

---

# Encyclopedia of Crash Dump Analysis Patterns

## Second Edition

---

Detecting Abnormal Software Structure and Behavior in Computer Memory

**Dmitry Vostokov**  
**Software Diagnostics Institute**

Published by OpenTask, Republic of Ireland

Copyright © 2017 by Dmitry Vostokov

Copyright © 2017 by Software Diagnostics Institute

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, without the prior written permission of the publisher.

You must not circulate this book in any other binding or cover, and you must impose the same condition on any acquirer.

OpenTask books are available through booksellers and distributors worldwide. For further information or comments send requests to [press@opentask.com](mailto:press@opentask.com).

Product and company names mentioned in this book may be trademarks of their owners.

A CIP catalog record for this book is available from the British Library.

ISBN-13: 978-1-908043-83-2 (Paperback)

First printing, 2017

Version 1.06 (March 2017)

## Summary of Contents

<b>Summary of Contents</b>	<b>3</b>
<b>Detailed Table of Contents</b>	<b>18</b>
<b>Preface to the First Edition</b>	<b>45</b>
<b>Preface to the Second Edition</b>	<b>46</b>
<b>Acknowledgements</b>	<b>47</b>
<b>About the Author</b>	<b>48</b>
<b>A</b>	<b>49</b>
Abridged Dump	49
Accidental Lock	53
Activation Context	60
Active Thread	63
Activity Resonance	70
Affine Thread	72
Annotated Disassembly	75
<b>B</b>	<b>76</b>
Blocked DPC	76
Blocked Queue	77
Blocked Thread	80
Blocking File	93
Blocking Module	96
Broken Link	97

---

Busy System	99
<b>C</b>	<b>108</b>
C++ Exception	108
Caller-n-Callee	111
Changed Environment	114
Clone Dump	118
Cloud Environment	122
CLR Thread	124
Coincidental Error Code	128
Coincidental Frames	130
Coincidental Symbolic Information	134
Constant Subtrace	141
Corrupt Dump	142
Corrupt Structure	144
Coupled Machines	146
Coupled Modules	147
Coupled Processes	148
Crash Signature	153
Crash Signature Invariant	155
Crashed Process	156
Critical Region	157
CriticalSection Corruption	161

---

Critical Stack Trace	168
Custom Exception Handler	169
<b>D</b>	<b>174</b>
Data Alignment	174
Data Contents Locality	175
Data Correlation	180
Deadlock	182
Debugger Bug	219
Debugger Omission	220
Design Value	221
Deviant Module	222
Deviant Token	229
Diachronic Module	230
Dialog Box	232
Directing Module	235
Disconnected Network Adapter	236
Disk Packet Buildup	238
Dispatch Level Spin	241
Distributed Exception	243
Distributed Spike	245
Distributed Wait Chain	253
Divide by Zero	255

---

Double Free	260
Double IRP Completion	279
Driver Device Collection	280
Dry Weight	281
Dual Stack Trace	282
Duplicate Extension	283
Duplicated Module	287
Dynamic Memory Corruption	292
<b>E</b>	<b>312</b>
Early Crash Dump	312
Effect Component	315
Embedded Comments	320
Empty Stack Trace	321
Environment Hint	324
Error Reporting Fault	325
Evental Dumps	328
Exception Module	361
Exception Stack Trace	363
Execution Residue	365
<b>F</b>	<b>385</b>
Fake Module	385
False Effective Address	389

---

False Function Parameters	390
False Positive Dump	393
Fat Process Dump	395
Fault Context	396
First Fault Stack Trace	397
Foreign Module Frame	398
FPU Exception	401
Frame Pointer Omission	403
Frozen Process	407
<b>G</b>	<b>411</b>
Ghost Thread	411
Glued Stack Trace	413
<b>H</b>	<b>416</b>
Handle Leak	416
Handle Limit	417
Handled Exception	428
Hardware Activity	437
Hardware Error	441
Hidden Call	450
Hidden Exception	455
Hidden IRP	462
Hidden Module	463

---

Hidden Parameter	465
Hidden Process	467
Hidden Stack Trace	469
High Contention	472
Historical Information	483
Hooked Functions	484
Hooked Modules	490
Hooking Level	492
<b>I</b>	<b>495</b>
Incomplete Stack Trace	495
Incomplete Session	496
Inconsistent Dump	498
Incorrect Stack Trace	499
Incorrect Symbolic Information	505
Injected Symbols	510
Inline Function Optimization	512
Instrumentation Information	516
Instrumentation Side Effect	520
Insufficient Memory	523
Internal Stack Trace	568
Invalid Exception Information	570
Invalid Handle	574

---

Invalid Parameter	586
Invalid Pointer	589
<b>J</b>	<b>591</b>
JIT Code	591
<b>L</b>	<b>596</b>
Last Error Collection	596
Last Object	599
Late Crash Dump	601
Lateral Damage	602
Least Common Frame	604
Livelock	606
Local Buffer Overflow	608
Lost Opportunity	612
<b>M</b>	<b>614</b>
Main Thread	614
Managed Code Exception	617
Managed Stack Trace	624
Manual Dump	625
Memory Fluctuation	634
Memory Leak	636
Message Box	660
Message Hooks	663

---

Mirror Dump Set	666
Missing Component	668
Missing Process	682
Missing Thread	683
Mixed Exception	688
Module Collection	693
Module Hint	696
Module Product Process	698
Module Stack Trace	699
Module Variable	701
Module Variety	703
Multiple Exceptions	706
<b>N</b>	<b>722</b>
Namespace	722
Nested Exceptions	723
Nested Offender	730
Network Packet Buildup	733
No Component Symbols	734
No Current Thread	737
No Data Types	739
No Process Dumps	740
No System Dumps	741

---

Not My Thread	742
Not My Version	743
NULL Pointer	745
<b>O</b>	<b>756</b>
Object Distribution Anomaly	756
OMAP Code Optimization	761
One-Thread Process	765
Optimized Code	767
Optimized VM Layout	769
Origin Module	771
Out-of-Module Pointer	773
Overaged System	774
<b>P</b>	<b>775</b>
Packed Code	775
Paged Out Data	778
Parameter Flow	780
Paratext	783
Pass Through Function	787
Passive System Thread	789
Passive Thread	793
Past Stack Trace	800
Patched Code	802

---

Pervasive System	803
Place Trace	804
Platform-Specific Debugger	806
Pleiades	808
Pre-Obfuscation Residue	809
Problem Exception Handler	810
Problem Module	812
Problem Vocabulary	813
Process Factory	814
Punctuated Memory Leak	819
<b>Q</b>	<b>823</b>
Quiet Dump	823
Quotient Stack Trace	824
<b>R</b>	<b>825</b>
Random Object	825
Raw Pointer	828
Reduced Symbolic Information	829
Reference Leak	830
Regular Data	833
Relative Memory Leak	834
RIP Stack Trace	837
Rough Stack Trace	839

---

<b>S</b>	<b>842</b>
Same Vendor	842
Screwbolt Wait Chain	843
Self-Diagnosis	844
Self-Dump	850
Semantic Split	853
Semantic Structure	860
Shared Buffer Overwrite	864
Shared Structure	872
Small Value	873
Software Exception	875
Special Process	877
Special Stack Trace	882
Special Thread	883
Spike Interval	884
Spiking Thread	885
Stack Overflow	895
Stack Trace	917
Stack Trace Change	932
Stack Trace Collection	933
Stack Trace Set	952
Stack Trace Signature	955

---

Stack Trace Surface	957
Step Dumps	958
Stored Exception	959
String Hint	960
String Parameter	962
Suspended Thread	964
Swarm of Shared Locks	966
System Object	971
<b>T</b>	<b>974</b>
Tampered Dump	974
Technology-Specific Subtrace	987
Template Module	997
Thread Age	1001
Thread Cluster	1003
Thread Poset	1004
Thread Starvation	1006
Top Module	1012
Translated Exception	1013
Truncated Dump	1014
Truncated Stack Trace	1017
<b>U</b>	<b>1020</b>
Ubiquitous Component	1020

---

Unified Stack Trace	1035
Unknown Component	1037
Unloaded Module	1041
Unrecognizable Symbolic Information	1045
Unsynchronized Dumps	1050
User Space Evidence	1051
<b>V</b>	<b>1052</b>
Value Adding Process	1052
Value Deviation	1053
Value References	1057
Variable Subtrace	1058
Version-Specific Extension	1064
Virtualized Process	1068
Virtualized System	1076
<b>W</b>	<b>1082</b>
Wait Chain	1082
Waiting Thread Time	1137
Well-Tested Function	1146
Well-Tested Module	1147
Wild Code	1148
Wild Pointer	1151
Window Hint	1153

---

<b>Y</b>		<b>1156</b>
Young System		1156
<b>Z</b>		<b>1158</b>
Zombie Processes		1158
<b>Bibliography</b>		<b>1165</b>
<b>Appendix A</b>		<b>1166</b>
Reference Stack Traces		1166
<b>Appendix B</b>		<b>1167</b>
.NET / CLR / Managed Space Patterns		1167
Contention Patterns		1168
Deadlock and Livelock Patterns		1169
DLL Link Patterns		1170
Dynamic Memory Corruption Patterns		1171
Executive Resource Patterns		1172
Exception Patterns		1173
Falsity and Coincidence Patterns		1174
Hookware Patterns		1175
Memory Consumption Patterns		1177
Meta-Memory Dump Patterns		1178
Module Patterns		1179
Optimization Patterns		1180
Process Patterns		1181

---

RPC, LPC and ALPC Patterns	1182
Stack Overflow Patterns	1183
Stack Trace Patterns	1184
Symbol Patterns	1186
Thread Patterns	1187
Wait Chain Patterns	1188
<b>Appendix C</b>	<b>1189</b>
Crash Dump Analysis Checklist	1189
<b>Index</b>	<b>1192</b>

## Detailed Table of Contents

<b>Summary of Contents</b>	<b>3</b>
<b>Detailed Table of Contents</b>	<b>18</b>
<b>Preface to the First Edition</b>	<b>45</b>
<b>Preface to the Second Edition</b>	<b>46</b>
<b>Acknowledgements</b>	<b>47</b>
<b>About the Author</b>	<b>48</b>
<b>A</b>	<b>49</b>
Abridged Dump	49
Accidental Lock	53
Activation Context	60
Active Thread	63
Linux	63
Mac OS X	64
Windows	66
Activity Resonance	70
Affine Thread	72
Annotated Disassembly	75
JIT .NET Code	75
<b>B</b>	<b>76</b>
Blocked DPC	76
Blocked Queue	77

---

LPC/ALPC	77
Comments	79
Blocked Thread	80
Hardware	80
Software	82
Comments	90
Timeout	92
Blocking File	93
Blocking Module	96
Comments	96
Broken Link	97
Busy System	99
<b>C</b>	<b>108</b>
C++ Exception	108
Linux	108
Mac OS X	109
Windows	110
Comments	110
Caller-n-Callee	111
Changed Environment	114
Comments	117
Clone Dump	118
Cloud Environment	122

---

CLR Thread	124
Comments	127
Coincidental Error Code	128
Coincidental Frames	130
Comments	133
Coincidental Symbolic Information	134
Linux	134
Mac OS X	135
Windows	137
Constant Subtrace	141
Corrupt Dump	142
Comments	143
Corrupt Structure	144
Coupled Machines	146
Coupled Modules	147
Coupled Processes	148
Semantics	148
Strong	149
Comments	150
Weak	151
Crash Signature	153
Crash Signature Invariant	155
Crashed Process	156

---

Critical Region	157
Linux	157
Critical Section Corruption	161
Critical Stack Trace	168
Custom Exception Handler	169
Kernel Space	169
User Space	171
<b>D</b>	<b>174</b>
Data Alignment	174
Page Boundary	174
Data Contents Locality	175
Data Correlation	180
Function Parameters	180
Deadlock	182
Critical Sections	182
Comments	189
Executive Resources	194
LPC	197
Managed Space	202
Mixed Objects	205
Kernel Space	205
User Space	210
Comments	217

---

Self	218
Comments	218
Debugger Bug	219
Debugger Omission	220
Design Value	221
Deviant Module	222
Comments	228
Deviant Token	229
Diachronic Module	230
Dialog Box	232
Directing Module	235
Disconnected Network Adapter	236
Disk Packet Buildup	238
Comments	240
Dispatch Level Spin	241
Distributed Exception	243
Managed Code	243
Distributed Spike	245
Comments	252
Distributed Wait Chain	253
Divide by Zero	255
Kernel Mode	255

---

User Mode	257
Linux	257
Mac OS X	258
Windows	259
Double Free	260
Kernel Pool	260
Comments	262
Process Heap	267
Windows	267
Comments	276
Mac OS X	278
Double IRP Completion	279
Driver Device Collection	280
Dry Weight	281
Dual Stack Trace	282
Duplicate Extension	283
Comments	286
Duplicated Module	287
Comments	291
Dynamic Memory Corruption	292
Kernel Pool	292
Comments	297
Managed Heap	301

---

Process Heap	304
Linux	304
Mac OS X	305
Windows	307
Comments	308
<b>E</b>	<b>312</b>
Early Crash Dump	312
Effect Component	315
Embedded Comments	320
Empty Stack Trace	321
Comments	323
Environment Hint	324
Comments	324
Error Reporting Fault	325
Evental Dumps	328
Exception Module	361
Exception Stack Trace	363
Comments	364
Execution Residue	365
Linux	365
Mac OS X	367
Windows	369
Managed Space	369

---

Comments	370
Unmanaged Space	371
Comments	382
<b>F</b>	<b>385</b>
Fake Module	385
False Effective Address	389
False Function Parameters	390
False Positive Dump	393
Fat Process Dump	395
Fault Context	396
First Fault Stack Trace	397
Foreign Module Frame	398
FPU Exception	401
Frame Pointer Omission	403
Frozen Process	407
<b>G</b>	<b>411</b>
Ghost Thread	411
Glued Stack Trace	413
<b>H</b>	<b>416</b>
Handle Leak	416
Handle Limit	417
GDI	417

---

Kernel Space	417
User Space	423
Handled Exception	428
.NET CLR	428
Kernel Space	433
User Space	434
Comments	436
Hardware Activity	437
Hardware Error	441
Comments	446
Hidden Call	450
Hidden Exception	455
Kernel Space	455
Comments	456
User Space	457
Comments	461
Hidden IRP	462
Hidden Module	463
Comments	464
Hidden Parameter	465
Hidden Process	467
Hidden Stack Trace	469
High Contention	472

---

.NET CLR Monitors	472
Critical Sections	475
Executive Resources	477
Comments	479
Processors	480
Historical Information	483
Comments	483
Hooked Functions	484
Kernel Space	484
Comments	487
User Space	488
Comments	489
Hooked Modules	490
Comments	491
Hooking Level	492
I	<b>495</b>
Incomplete Stack Trace	495
GDB	495
Incomplete Session	496
Comments	497
Inconsistent Dump	498
Comments	498
Incorrect Stack Trace	499

---

Comments	504
Incorrect Symbolic Information	505
Injected Symbols	510
Inline Function Optimization	512
Managed Code	512
Unmanaged Code	514
Instrumentation Information	516
Instrumentation Side Effect	520
Comments	522
Insufficient Memory	523
Committed Memory	523
Control Blocks	525
Handle Leak	526
Comments	530
Kernel Pool	535
Comments	543
Module Fragmentation	544
Comments	551
Physical Memory	552
PTE	555
Comments	556
Region	557
Reserved Virtual Memory	559
Session Pool	562

---

Stack Trace Database	563
Internal Stack Trace	568
Invalid Exception Information	570
Invalid Handle	574
General	574
Comments	577
Managed Space	578
Comments	585
Invalid Parameter	586
Process Heap	586
Invalid Pointer	589
General	589
<b>J</b>	<b>591</b>
JIT Code	591
.NET	591
Comments	593
Java	594
<b>L</b>	<b>596</b>
Last Error Collection	596
Comments	597
Last Object	599
Comments	600
Late Crash Dump	601

---

Lateral Damage	602
Linux	602
Windows	603
Comments	603
Least Common Frame	604
Livelock	606
Local Buffer Overflow	608
Linux	608
Mac OS X	609
Windows	611
Comments	611
Lost Opportunity	612
<b>M</b>	<b>614</b>
Main Thread	614
Managed Code Exception	617
Managed Stack Trace	624
Manual Dump	625
Kernel	625
Comments	626
Process	630
Comments	633
Memory Fluctuation	634
Process Heap	634

---

Comments	635
Memory Leak	636
.NET Heap	636
Comments	642
I/O Completion Packets	643
Page Tables	644
Process Heap	650
Comments	656
Regions	657
Message Box	660
Comments	662
Message Hooks	663
Comments	665
Mirror Dump Set	666
Missing Component	668
General	668
Static Linkage	672
User Mode	672
Comments	681
Missing Process	682
Comments	682
Missing Thread	683
Comments	687

---

Mixed Exception	688
Comments	692
Module Collection	693
General	693
Predicate	695
Module Hint	696
Comments	697
Module Product Process	698
Module Stack Trace	699
Linux	699
Windows	700
Module Variable	701
Module Variety	703
Multiple Exceptions	706
Mac OS X	706
Windows	708
Kernel Mode	708
Managed Space	713
Stowed	714
User Mode	720
<b>N</b>	<b>722</b>
Namespace	722
Nested Exceptions	723

---

Managed Code	723
Comments	725
Unmanaged Code	726
Nested Offender	730
Network Packet Buildup	733
No Component Symbols	734
No Current Thread	737
No Data Types	739
No Process Dumps	740
No System Dumps	741
Comments	741
Not My Thread	742
Not My Version	743
Hardware	743
Software	744
NULL Pointer	745
Linux	745
Code	745
Data	746
Mac OS X	747
Code	747
Data	749
Windows	750

---

Code	750
Data	752
Comments	752
<b>O</b>	<b>756</b>
Object Distribution Anomaly	756
.NET Heap	756
IRP	759
Comment	760
OMAP Code Optimization	761
Comments	764
One-Thread Process	765
Optimized Code	767
Comments	768
Optimized VM Layout	769
Origin Module	771
Out-of-Module Pointer	773
Overaged System	774
Comments	774
<b>P</b>	<b>775</b>
Packed Code	775
Paged Out Data	778
Parameter Flow	780

---

Paratext	783
Linux	783
Mac OS X	785
Comments	786
Pass Through Function	787
Comments	788
Passive System Thread	789
Kernel Space	789
Passive Thread	793
User Space	793
Comments	799
Past Stack Trace	800
Patched Code	802
Pervasive System	803
Place Trace	804
Comments	805
Platform-Specific Debugger	806
Pleiades	808
Pre-Obfuscation Residue	809
Problem Exception Handler	810
Comments	811
Problem Module	812

---

Comments	812
Problem Vocabulary	813
Process Factory	814
Punctuated Memory Leak	819
<b>Q</b>	<b>823</b>
Quiet Dump	823
Quotient Stack Trace	824
<b>R</b>	<b>825</b>
Random Object	825
Raw Pointer	828
Reduced Symbolic Information	829
Reference Leak	830
Regular Data	833
Relative Memory Leak	834
RIP Stack Trace	837
Rough Stack Trace	839
<b>S</b>	<b>842</b>
Same Vendor	842
Screwbolt Wait Chain	843
Self-Diagnosis	844
Kernel Mode	844
Comments	844

---

Registry	845
User Mode	847
Comments	848
Self-Dump	850
Comments	852
Semantic Split	853
Semantic Structure	860
PID.TID	860
Comments	863
Shared Buffer Overwrite	864
Mac OS X	864
Windows	868
Shared Structure	872
Small Value	873
Comments	874
Software Exception	875
Comments	875
Special Process	877
Comments	881
Special Stack Trace	882
Comments	882
Special Thread	883

---

.NET CLR	883
Spike Interval	884
Spiking Thread	885
Linux	885
Mac OS X	886
Windows	888
Comments	893
Stack Overflow	895
Linux	895
Mac OS X	897
Windows	900
Kernel Mode	900
Comments	908
Software Implementation	910
User Mode	912
Comments	915
Stack Trace	917
Linux	917
Mac OS X	918
Windows	919
Database	919
File System Filters	924
General	926
I/O Request	930

---

Stack Trace Change	932
Stack Trace Collection	933
CPUs	933
I/O Requests	936
Managed Space	940
Predicate	943
Comments	943
Unmanaged Space	944
Comments	951
Stack Trace Set	952
Stack Trace Signature	955
Stack Trace Surface	957
Step Dumps	958
Stored Exception	959
String Hint	960
String Parameter	962
Suspended Thread	964
Swarm of Shared Locks	966
System Object	971
<b>T</b>	<b>974</b>
Tampered Dump	974
Technology-Specific Subtrace	987
COM Client Call	987

---

COM Interface Invocation	988
Comments	991
Dynamic Memory	992
JIT .NET Code	994
Template Module	997
Thread Age	1001
Thread Cluster	1003
Thread Poset	1004
Thread Starvation	1006
Normal Priority	1006
Realtime Priority	1008
Top Module	1012
Translated Exception	1013
Truncated Dump	1014
Mac OS X	1014
Windows	1015
Truncated Stack Trace	1017
Comments	1017
<b>U</b>	<b>1020</b>
Ubiquitous Component	1020
Kernel Space	1020
User Space	1023
Unified Stack Trace	1035

---

Unknown Component	1037
Unloaded Module	1041
Unrecognizable Symbolic Information	1045
Unsynchronized Dumps	1050
User Space Evidence	1051
<b>V</b>	<b>1052</b>
Value Adding Process	1052
Value Deviation	1053
Stack Trace	1053
Value References	1057
Comments	1057
Variable Subtrace	1058
Version-Specific Extension	1064
Virtualized Process	1068
WOW64	1068
Comments	1075
Virtualized System	1076
<b>W</b>	<b>1082</b>
Wait Chain	1082
C++11, Condition Variable	1082
CLR Monitors	1085
Critical Sections	1086

---

Executive Resources	1089
General	1092
Comments	1096
LPC/ALPC	1097
Modules	1103
Comments	1103
Mutex Objects	1104
Named Pipes	1106
Nonstandard Synchronization	1108
Process Objects	1111
Pushlocks	1116
RPC	1118
RTL_RESOURCE	1122
Thread Objects	1128
Window Messaging	1132
Waiting Thread Time	1137
Kernel Dumps	1137
Comments	1142
User Dumps	1144
Comments	1145
Well-Tested Function	1146
Well-Tested Module	1147
Wild Code	1148
Comments	1149

---

Wild Pointer	1151
Comments	1152
Window Hint	1153
<b>Y</b>	<b>1156</b>
Young System	1156
Comments	1157
<b>Z</b>	<b>1158</b>
Zombie Processes	1158
Comments	1164
<b>Bibliography</b>	<b>1165</b>
<b>Appendix A</b>	<b>1166</b>
Reference Stack Traces	1166
<b>Appendix B</b>	<b>1167</b>
.NET / CLR / Managed Space Patterns	1167
Contention Patterns	1168
Deadlock and Livelock Patterns	1169
DLL Link Patterns	1170
Dynamic Memory Corruption Patterns	1171
Executive Resource Patterns	1172
Exception Patterns	1173
Falsity and Coincidence Patterns	1174
Hookware Patterns	1175

---

Memory Consumption Patterns	1177
Meta-Memory Dump Patterns	1178
Module Patterns	1179
Optimization Patterns	1180
Process Patterns	1181
RPC, LPC and ALPC Patterns	1182
Stack Overflow Patterns	1183
Stack Trace Patterns	1184
Symbol Patterns	1186
Thread Patterns	1187
Wait Chain Patterns	1188
<b>Appendix C</b>	<b>1189</b>
Crash Dump Analysis Checklist	1189
<b>Index</b>	<b>1192</b>

## Preface to the First Edition

We originally planned this book in 2009. At that time, there were less than 100 memory dump analysis patterns. Since then, Software Diagnostics Institute has already added many more patterns. All of them (326 patterns in total) are scattered among 3,300 pages of Memory Dump Analysis Anthology volumes (1 – 7, 8a), and a few can be found only in Software Diagnostics Library. So we decided to reprint all these patterns and their examples in one book for easy reference. During editing, we also corrected various mistakes, added additional comments and cross-references. Most of the patterns are for Windows platforms and WinDbg with a few examples for Mac OS X and GDB. However, this pattern language is easily extendable, and we plan to add more non-Windows examples (including Linux) in the future editions.

If you encounter any error, please contact me using this form

<http://www.dumpanalysis.org/contact>

Alternatively, send me a personal message using this contact e-mail:

dmitry.vostokov@dumpanalysis.org

Alternatively, via Twitter @ DumpAnalysis

## Preface to the Second Edition

Two years passed since the publication of the previous edition and after the release of volumes 8b, 9a, and 9b of Memory Dump Analysis Anthology containing new memory analysis patterns including Linux variants we decided to publish the updated edition. It now includes more than 50 new patterns and pattern variants including 5 analysis patterns from the forthcoming volume 10a at the time of this writing, and more than 70 new comments. WinDbg output and code sections were reformatted for easier screen and paperback reading. Some typos were corrected, and punctuation was improved.

In addition to contact details provided in the preface to the first edition, we suggest the following Facebook page and group:

<http://www.facebook.com/DumpAnalysis>

<http://www.facebook.com/groups/dumpanalysis>

## Acknowledgements

Special thanks to Igor Dzyubenko, who suggested corrections and improvements for the second edition, João Roque, and Malcolm McCaffery who provided encouraging support just after the first edition release, users of Software Diagnostics Library who contributed to comment sections, customers of Software Diagnostics Services training courses and reference materials who provided financial assistance to continue this project.

## About the Author



Dmitry Vostokov is an internationally recognized expert, speaker, educator, scientist, and author. He is the founder of pattern-oriented software diagnostics, forensics, and prognostics discipline and Software Diagnostics Institute (DA+TA: [DumpAnalysis.org](http://DumpAnalysis.org) + [TraceAnalysis.org](http://TraceAnalysis.org)). Vostokov has also authored more than 30 books on software diagnostics, forensics and problem-solving, memory dump analysis, debugging, software trace and log analysis, reverse engineering, and malware analysis. He has more than 20 years of experience in software architecture, design, development, and maintenance in a variety of industries including leadership, technical and people management roles. Dmitry also founded DiaThings, Logtellect, OpenTask Iterative and Incremental Publishing ([OpenTask.com](http://OpenTask.com)), Software Diagnostics Services (former Memory Dump Analysis Services) PatternDiagnostics.com and Software Prognostics. In his spare time, he presents various topics on Debugging.TV and explores Software Narratology, an applied science of software stories that he pioneered, and its further development as Narratology of Things and Diagnostics of Things (DoT). His current area of interest is theoretical software diagnostics.

# A

## Abridged Dump

Sometimes we get memory dumps that are difficult to analyze in full because some if not most of the information was omitted while saving them. These are usually small memory dumps (contrasted with kernel and complete) and user process minidumps. We can easily recognize that when we open a dump file:

User Mini Dump File: Only registers, stack and portions of memory are available

Mini Kernel Dump File: Only registers and stack trace are available

The same also applies to user dumps where thread times information is omitted (it is not possible to use **!runaway** WinDbg command) or to a dump saved with various options of **.dump** command (including privacy-aware<sup>1</sup>) instead of **/ma** or deprecated **/f** option. On the contrary, manually erased data<sup>2</sup> in crash dumps looks more like an example of another pattern called **Lateral Damage** (page 602).

The similar cases of abridged dumps are discussed in **Wrong Dump**<sup>3</sup> and **Missing Space**<sup>4</sup> antipatterns.

Anyway, we shouldn't dismiss such dump files and should try to analyze them. For example, some approaches (including using image binaries) are listed in kernel minidump analysis series<sup>5</sup>. We can even see portions of the raw stack data when looking for **Execution Residue** (page 371):

```
0: kd> !thread
GetPointerFromAddress: unable to read from 81d315b0
THREAD 82f49020 Cid 0004.0034 Teb: 00000000 Win32Thread: 00000000 RUNNING on processor 0
IRP List:
    Unable to read nt!_IRP @ 8391e008
Not impersonating
GetUlongFromAddress: unable to read from 81d0ad90
Owning Process      82f00ab0     Image:      System
Attached Process    N/A        Image:      N/A
ffd0000: Unable to get shared data
Wait Start TickCount   4000214
Context Switch Count 21886
ReadMemory error: Cannot get nt!KeMaximumIncrement value.
```

<sup>1</sup> WinDbg is Privacy-Aware, Memory Dump Analysis Anthology, Volume 1, page 600

<sup>2</sup> Data Hiding in Crash Dumps, Memory Dump Analysis Anthology, Volume 2, page 397

<sup>3</sup> Wrong Dump, Memory Dump Analysis Anthology, Volume 1, page 496

<sup>4</sup> Missing Space, Memory Dump Analysis Anthology, Volume 3, page 138

<sup>5</sup> Minidump Analysis, Memory Dump Analysis Anthology, Volume 1, page 43

```
UserTime          00:00:00.000
KernelTime        00:00:00.000
Win32 Start Address nt!ExpWorkerThread (0x81c78ea3)
Stack Init 85be0000 Current 85bdf7c0 Base 85be0000 Limit 85bdd000 Call 0
Priority 14 BasePriority 12 PriorityDecrement 0 IoPriority 2 PagePriority 5
[...]

0: kd> dps 85bdd000 85be0000
85bdd000  ????????
85bdd004  ????????
85bdd008  ????????
85bdd00c  ????????
85bdd010  ????????
85bdd014  ????????
85bdd018  ????????
85bdd01c  ????????
[...]
85bdf8c4  ????????
85bdf8c8  ????????
85bdf8cc  ????????
85bdf8d0  0000000a
85bdf8d4  a112883e
85bdf8d8  0000001b
85bdf8dc  00000000
85bdf8e0  81c28750 nt!KeSetEvent+0x4d
85bdf8e4  85bdf8e8
85bdf8e8  85bdf970
85bdf8ec  81c28750 nt!KeSetEvent+0x4d
85bdf8f0  badb0d00
85bdf8f4  00000000
85bdf8f8  00000000
85bdf8fc  81cf4820 nt!KiInitialPCR+0x120
85bdf900  00000000
85bdf904  85bdf938
85bdf908  81cf4820 nt!KiInitialPCR+0x120
85bdf90c  00000000
85bdf910  81d32300 nt!IopTimerLock
85bdf914  00000000
85bdf918  81fa0000 nt!_NULL_IMPORT_DESCRIPTOR <PERF> (nt+0x3a0000)
85bdf91c  85bd0023
85bdf920  00000023
85bdf924  00000000
85bdf928  81d323c0 nt!KiDispatcherLock
85bdf92c  a1128828
85bdf930  85bdf9b4
85bdf934  85bdfdb0
85bdf938  00000030
85bdf93c  84ca6f40
85bdf940  84ca6f38
85bdf944  00000001
85bdf948  85bdf970
85bdf94c  00000000
85bdf950  81c28750 nt!KeSetEvent+0x4d
85bdf954  00000008
85bdf958  00010246
85bdf95c  00000000
```

```
85bdf960 84ca68a0
[...]
85bdfd2c 82f49020
85bdfd30 835ca4d0
85bdfd34 a6684538
85bdfd38 81cfde7c nt!ExWorkerQueue+0x3c
85bdfd3c 00000001
85bdfd40 00000000
85bdfd44 85bdfd7c
85bdfd48 81c78fa0 nt!ExpWorkerThread+0xfd
85bdfd4c 835ca4d0
85bdfd50 00000000
85bdfd54 82f49020
85bdfd58 00000000
85bdfd5c 00000000
85bdfd60 0069000b
85bdfd64 00000000
85bdfd68 00000001
85bdfd6c 00000000
85bdfd70 835ca4d0
85bdfd74 81da9542 nt!PnpDeviceEventWorker
85bdfd78 00000000
85bdfd7c 85bdfd0
85bdfd80 81e254e0 nt!PspSystemThreadStartup+0x9d
85bdfd84 835ca4d0
85bdfd88 85bd4680
85bdfd8c 00000000
85bdfd90 00000000
85bdfd94 00000000
85bdfd98 00000002
85bdfd9c 00000000
85bdfda0 00000000
85bdfda4 00000001
85bdfda8 85bdfd88
85bdfdac 85bdfdbc
85bdfdb0 ffffffff
85bdfdb4 81c8aad5 nt!_except_handler4
85bdfdb8 81c9ddb8 nt!`string'+0x4
85bdfdbc 00000000
85bdfdc0 00000000
85bdfdc4 81c9159e nt!KiThreadStartup+0x16
85bdfdc8 81c78ea3 nt!ExpWorkerThread
85bdfdcc 00000001
85bdfdd0 00000000
85bdfdd4 00000000
85bdfdd8 002e0069
85bdfddc 006c0064
85bdfde0 004c006c
85bdfde4 00000000
85bdfde8 000007f0
85bdfdec 00010000
85bdfdf0 0000027f
85bdfdf4 00000000
85bdfdf8 00000000
85bdfdfc 00000000
85bdffe00 00000000
```

```

85bdfe04 00000000
85bdfe08 00001f80
85bdfe0c 0000ffff
85bdfe10 00000000
85bdfe14 00000000
85bdfe18 00000000
[...]
85bdffec 00000000
85bdfff0 00000000
85bdfff4 00000000
85bdfff8 00000000
85bdfffc 00000000
85be0000 ????????

```

User minidumps are similar here:

```

0:001> k
ChildEBP RetAddr
099bfe14 7c90daaa ntdll!KiFastSystemCallRet
099bfe18 77e765e3 ntdll!NtReplyWaitReceivePortEx+0xc
099bff80 77e76caf rpcrt4!LRPC_ADDRESS::ReceiveLotsaCalls+0x12a
099bff88 77e76ad1 rpcrt4!RecvLotsaCallsWrapper+0xd
099bffa8 77e76c97 rpcrt4!BaseCachedThreadRoutine+0x79
099bffb4 7c80b729 rpcrt4!ThreadStartRoutine+0x1a
099bfffec 00000000 kernel32!BaseThreadStart+0x37

0:001> dd 099bfe14
099bfe14 099bfe24 7c90daaa 77e765e3 00000224
099bfe24 099bff74 00000000 2db87ae8 099bff48
099bfe34 fbf58e18 00000040 fd629338 b279dbbc
099bfe44 fd5928b8 fbf58ebc b279dbbc e0c1e002
099bfe54 00000000 00000006 00000001 00000000
099bfe64 e637d218 00000000 00000006 00000006
099bfe74 00000006 e1f79698 e39b8b60 00000000
099bfe84 fbe33c40 00000001 e5ce12f8 b279db9c

0:001> dd 099bfe14-20
099bfd4 ???????? ???????? ???????? ???????? ???????
099bfe04 ???????? ???????? ???????? ???????? ???????
099bfe14 099bfe24 7c90daaa 77e765e3 00000224
099bfe24 099bff74 00000000 2db87ae8 099bff48
099bfe34 fbf58e18 00000040 fd629338 b279dbbc
099bfe44 fd5928b8 fbf58ebc b279dbbc e0c1e002
099bfe54 00000000 00000006 00000001 00000000
099bfe64 e637d218 00000000 00000006 00000006

```

As a warning here it is possible to conclude that minidumps can also reveal the private information especially when ASCII or Unicode buffers are seen in the raw stack data.

We named this pattern by analogy with an abridged book.

## Accidental Lock

When a system is unresponsive or sluggish, we usually check \_ERESOURCE locks in the kernel or complete memory dumps to see **Deadlock** (page 193) or **High Contention** (page 477) patterns. However, there is some chance that reported locks are purely accidental and appear in a crash dump because they just happened at that time. We need to look at *Contention Count*, *Ticks* and *KernelTime* in both blocking and blocked threads to recognize an **Accidental Lock**. Also, WinDbg may not distinguish between prolonged and accidental locks when we use `!analyze -v -hang` command, and merely reports some lock chain it finds among equal alternatives.

Here is an example. The system was reported hung, and kernel memory dump was saved. WinDbg analysis command reports one thread blocking 3 other threads and the driver on top of the blocking thread stack is *AVDriver.sys*. The algorithm WinDbg uses to point to specific image name is described in minidump analysis article<sup>6</sup> and in our case it chooses *AVDriver* module:

```
BLOCKED_THREAD: 8089d8c0

BLOCKING_THREAD: 8aab4700

LOCK_ADDRESS: 8859a570 -- (!locks 8859a570)

Resource @ 0x8859a570      Exclusively owned
Contention Count = 3
NumberOfExclusiveWaiters = 3
Threads: 8aab4700-01<*>
Threads Waiting On Exclusive Access:
     885d0020      88a7c020      8aafc7d8

1 total locks, 1 locks currently held

BUGCHECK_STR: LOCK_HELD

FAULTING_THREAD: 8aab4700

STACK_TEXT:
f592f698 80832f7a nt!KiSwapContext+0x26
f592f6c4 80828705 nt!KiSwapThread+0x284
f592f70c f720a394 nt!KeDelayExecutionThread+0x2ab
WARNING: Stack unwind information not available. Following frames may be wrong.
f592f734 f720ae35 AVDriver+0x1394
f592f750 f720b208 AVDriver+0x1e35
f592f794 f721945a AVDriver+0x2208
f592f7cc 8081dcdf AVDriver+0x1045a
f592f7e0 f5b9f76a nt!IoCallDriver+0x45
f592f7f0 f5b9c621 Driver!FS_Dispatch+0xa4
```

---

<sup>6</sup> Minidump Analysis, Memory Dump Analysis Anthology, Volume 1, page 43

```
f592f7fc 8081dcdf Driver!Kernel_dispatch+0x53
f592f810 f5eb2856 nt!IoCallDriver+0x45
f592f874 8081dcdf AVFilter!QueryFullName+0x5c10
f592f888 f5e9eae3 nt!IoCallDriver+0x45
f592f8b8 f5e9eca4 DrvFilter!PassThrough+0x115
f592f8d4 8081dcdf DrvFilter!Create+0xda
f592f8e8 808f8275 nt!IoCallDriver+0x45
f592f9d0 808f86bc nt!IopParseDevice+0xa35
f592fa08 80936689 nt!IopParseFile+0x46
f592fa88 80932e04 nt!ObpLookupObjectName+0x11f
f592fad0 808ea231 nt!ObOpenObjectByName+0xea
f592fb58 808eb4cb nt!IopCreateFile+0x447
f592fb5b4 f57c8efd nt!IoCreateFile+0xa3
f592fc24 f57c9f29 srv!SrvIoCreateFile+0x36d
f592fcf0 f57ca5e4 srv!SrvNtCreateFile+0x5cc
f592fd78 f57adbc6 srv!SrvSmbNtCreateAndX+0x15c
f592fd84 f57c3451 srv!SrvProcessSmb+0xb7
f592fdac 80948bd0 srv!WorkerThread+0x138
f592fddc 8088d4e2 nt!PspSystemThreadStartup+0x2e
00000000 00000000 nt!KiThreadStartup+0x16
```

STACK\_COMMAND: .thread 0xffffffff8aab4700 ; kb

FOLLOWUP\_IP:

```
AVDriver+1394
f720a394 eb85      jmp     AVDriver+0x131b (f720a31b)
```

MODULE\_NAME: AVDriver

IMAGE\_NAME: AVDriver.sys

Motivated by this “discovery” we want to see all locks:

```
0: kd> !locks
**** DUMP OF ALL RESOURCE OBJECTS ****
KD: Scanning for held locks...

Resource @ 0x895a62d8    Shared 1 owning threads
Threads: 89570520-01<*>

Resource @ 0x897ceba8    Shared 1 owning threads
Threads: 89584020-01<*>

Resource @ 0x8958e020    Shared 1 owning threads
Threads: 89555020-01<*>

Resource @ 0x89590608    Shared 1 owning threads
Threads: 89666020-01<*>

Resource @ 0x89efc398    Shared 1 owning threads
Threads: 89e277c0-01<*>
```

```
Resource @ 0x88d70820      Shared 1 owning threads
Threads: 88e43948-01<*>

Resource @ 0x89f2fb00      Shared 1 owning threads
Threads: 89674688-01<*>

Resource @ 0x89c80370      Shared 1 owning threads
Threads: 888496b8-01<*>

Resource @ 0x89bfd08       Shared 1 owning threads
Threads: 88b62910-01<*>

Resource @ 0x888b5488      Shared 1 owning threads
Threads: 88536730-01<*>

Resource @ 0x89f2e348       Shared 1 owning threads
Threads: 89295930-01<*>

Resource @ 0x891a0838      Shared 1 owning threads
Threads: 88949020-01<*>

Resource @ 0x8825bf08       Shared 1 owning threads
Threads: 882b9a08-01<*>

Resource @ 0x881a6510      Shared 1 owning threads
Threads: 88a88338-01<*>

Resource @ 0x885c5890      Shared 1 owning threads
Threads: 881ab020-01<*>

Resource @ 0x886633a8       Shared 1 owning threads
Threads: 89b5f8b0-01<*>

Resource @ 0x88216390      Shared 1 owning threads
Threads: 88820020-01<*>

Resource @ 0x88524490      Shared 1 owning threads
Threads: 88073020-01<*>

Resource @ 0x88f6a020       Shared 1 owning threads
Threads: 88e547b0-01<*>

Resource @ 0x88cf2020      Shared 1 owning threads
Threads: 89af32d8-01<*>

Resource @ 0x889cea80      Shared 1 owning threads
Threads: 88d18b40-01<*>

Resource @ 0x88486298      Shared 1 owning threads
Threads: 88af7db0-01<*>
```

```

Resource @ 0x88b22270    Exclusively owned
Contention Count = 4
NumberOfExclusiveWaiters = 4
Threads: 8aad07d8-01<*>
Threads Waiting On Exclusive Access:
8ad78020      887abdb0      88eb39a8      8aa1f668

Resource @ 0x88748c20    Exclusively owned
Contention Count = 2
NumberOfExclusiveWaiters = 2
Threads: 8873c8d8-01<*>
Threads Waiting On Exclusive Access:
88477478      88db6020

Resource @ 0x8859a570    Exclusively owned
Contention Count = 3
NumberOfExclusiveWaiters = 3
Threads: 8aab4700-01<*>
Threads Waiting On Exclusive Access:
885d0020      88a7c020      8aafc7d8

KD: Scanning for held locks...
18911 total locks, 25 locks currently held

```

We can ignore shared locks and then concentrate on the last 3 exclusively owned resources. It looks suspicious that Contention Count has the same number as the number of threads waiting for exclusive access (*NumberOfExclusiveWaiters*). This means that these resources had never been used before. If we dump locks verbosely, we see that blocked threads had been waiting no more than 2 seconds, for example, for resource 0x8859a570:

```

0: kd> !thread 885d0020; !thread 88a7c020; !thread 8aafc7d8
THREAD 885d0020  Cid 0004.1c34  Teb: 00000000 Win32Thread: 00000000 WAIT: (Unknown) KernelMode Non-
Alertable
 89908d50  SynchronizationEvent
 885d0098  NotificationTimer
Not impersonating
DeviceMap          e10022c8
Owning Process     8ad80648 Image:           System
Wait Start TickCount 7689055  Ticks: 127 (0:00:00:01.984)
Context Switch Count 248
UserTime           00:00:00.000
KernelTime         00:00:00.000
Start Address srv!WorkerThread (0xf57c3394)
Stack Init b4136000 Current b4135b74 Base b4136000 Limit b4133000 Call 0
Priority 9 BasePriority 9 PriorityDecrement 0
ChildEBP RetAddr
b4135b8c 80832f7a nt!KiSwapContext+0x26
b4135bb8 8082925c nt!KiSwapThread+0x284
b4135c00 8087c1ad nt!KeWaitForSingleObject+0x346
b4135c3c 8087c3a1 nt!ExpWaitForResource+0xd5
b4135c5c f57c9e95 nt!ExAcquireResourceExclusiveLite+0x8d
b4135cf0 f57ca5e4 srv!SrvNtCreateFile+0x510
b4135d78 f57adbc6 srv!SrvSmbNtCreateAndX+0x15c

```

```
b4135d84 f57c3451 srv!SrvProcessSmb+0xb7
b4135dac 80948bd0 srv!WorkerThread+0x138
b4135ddc 8088d4e2 nt!PspSystemThreadStartup+0x2e
00000000 00000000 nt!KiThreadStartup+0x16
```

THREAD 88a7c020 Cid 0004.3448 Peb: 00000000 Win32Thread: 00000000 WAIT: (Unknown) KernelMode Non-Alertable

```
89908d50 SynchronizationEvent
88a7c098 NotificationTimer
Not impersonating
DeviceMap e10022c8
Owning Process 8ad80648 Image: System
Wait Start TickCount 7689112 Ticks: 70 (0:00:00:01.093)
Context Switch Count 210
UserTime 00:00:00.000
KernelTime 00:00:00.000
Start Address srv!WorkerThread (0xf57c3394)
Stack Init b55dd000 Current b55dc74 Base b55dd000 Limit b55da000 Call 0
Priority 9 BasePriority 9 PriorityDecrement 0
ChildEBP RetAddr
b55dc8c 80832f7a nt!KiSwapContext+0x26
b55dcbb8 8082925c nt!KiSwapThread+0x284
b55dcc00 8087c1ad nt!KeWaitForSingleObject+0x346
b55dcc3c 8087c3a1 nt!ExpWaitForResource+0xd5
b55dcc5c f57c9e95 nt!ExAcquireResourceExclusiveLite+0x8d
b55dccf0 f57ca5e4 srv!SrvNtCreateFile+0x510
b55dcdf8 f57adbc6 srv!SrvSmbNtCreateAndX+0x15c
b55dcdf84 f57c3451 srv!SrvProcessSmb+0xb7
b55dcdac 80948bd0 srv!WorkerThread+0x138
b55dcddc 8088d4e2 nt!PspSystemThreadStartup+0x2e
00000000 00000000 nt!KiThreadStartup+0x16
```

THREAD 8aafc7d8 Cid 0004.058c Peb: 00000000 Win32Thread: 00000000 WAIT: (Unknown) KernelMode Non-Alertable

```
89908d50 SynchronizationEvent
8aafc850 NotificationTimer
Not impersonating
DeviceMap e10022c8
Owning Process 8ad80648 Image: System
Wait Start TickCount 7689171 Ticks: 11 (0:00:00:00.171)
Context Switch Count 310
UserTime 00:00:00.000
KernelTime 00:00:00.000
Start Address srv!WorkerThread (0xf57c3394)
Stack Init f592c000 Current f592bb18 Base f592c000 Limit f5929000 Call 0
Priority 9 BasePriority 9 PriorityDecrement 0
ChildEBP RetAddr
f592bb30 80832f7a nt!KiSwapContext+0x26
f592bb5c 8082925c nt!KiSwapThread+0x284
f592bba4 8087c1ad nt!KeWaitForSingleObject+0x346
f592bbe0 8087c3a1 nt!ExpWaitForResource+0xd5
f592bc00 f57c8267 nt!ExAcquireResourceExclusiveLite+0x8d
f592bc18 f57ff0ed srv!UnlinkRfcbsFromLfcbs+0x33
f592bc34 f57ff2ea srv!SrvCompleteRfcbsClose+0x1df
f592bc54 f57b5e8f srv!CloseRfcbsInternal+0xb6
f592bc78 f57ce8a9 srv!SrvCloseRfcbsOnSessionOrPid+0x74
```

```
f592bc94 f57e2b22 srv!SrvCloseSession+0xb0
f592bcb8 f57aeb12 srv!SrvCloseSessionsOnConnection+0xa9
f592bcd4 f57c79ed srv!SrvCloseConnection+0x143
f592bd04 f5808c50 srv!SrvCloseConnectionsFromClient+0x17f
f592bdac 80948bd0 srv!WorkerThread+0x138
f592bddc 8088d4e2 nt!PspSystemThreadStartup+0x2e
00000000 00000000 nt!KiThreadStartup+0x16
```

Blocking threads themselves are not blocked and active: the number of ticks passed since their last wait or preemption is 0. This could be a sign of CPU spike pattern. However, their accumulated KernelTime is less than a second:

```
0: kd> !thread 8aad07d8
THREAD 8aad07d8 Cid 0004.0580 Teb: 00000000 Win32Thread: 00000000 WAIT: (Unknown) KernelMode Non-
Alertable
  8aad0850 NotificationTimer
IRP List:
  8927ade0: (0006,0220) Flags: 00000884 Mdl: 00000000
Impersonation token: eafdc030 (Level Impersonation)
DeviceMap          e5d69340
Owning Process    8ad80648      Image:           System
Wait Start TickCount 7689182      Ticks: 0
Context Switch Count 915582
UserTime           00:00:00.000
KernelTime         00:00:00.125
Start Address srv!WorkerThread (0xf57c3394)
Stack Init f59d8000 Current f59d7680 Base f59d8000 Limit f59d5000 Call 0
Priority 9 BasePriority 9 PriorityDecrement 0

0: kd> !thread 8873c8d8
THREAD 8873c8d8 Cid 0004.2898 Teb: 00000000 Win32Thread: 00000000 WAIT: (Unknown) KernelMode Non-
Alertable
  8873c950 NotificationTimer
IRP List:
  882a8de0: (0006,0220) Flags: 00000884 Mdl: 00000000
Impersonation token: eafdc030 (Level Impersonation)
DeviceMap          e5d69340
Owning Process    8ad80648      Image:           System
Wait Start TickCount 7689182      Ticks: 0
Context Switch Count 917832
UserTime           00:00:00.000
KernelTime         00:00:00.031
Start Address srv!WorkerThread (0xf57c3394)
Stack Init ac320000 Current ac31f680 Base ac320000 Limit ac31d000 Call 0
Priority 9 BasePriority 9 PriorityDecrement 0
```

```
0: kd> !thread 8aab4700
THREAD 8aab4700  Cid 0004.0588  Teb: 00000000 Win32Thread: 00000000 WAIT: (Unknown) KernelMode Non-
Alertable
    8aab4778  NotificationTimer
IRP List:
    88453008: (0006,0220) Flags: 00000884  Mdl: 00000000
Impersonation token: e9a82728 (Level Impersonation)
DeviceMap          eb45f108
Owning Process    8ad80648      Image:           System
Wait Start TickCount 7689182      Ticks: 0
Context Switch Count 1028220
UserTime           00:00:00.000
KernelTime         00:00:00.765
Start Address srv!WorkerThread (0xf57c3394)
Stack Init f5930000 Current f592f680 Base f5930000 Limit f592d000 Call 0
Priority 9 BasePriority 9 PriorityDecrement 0
```

Based on this observation we could say that locks were accidental and when the problem happened again, the new dump didn't show them.

## Activation Context

This is a new pattern about activation contexts<sup>7</sup>. Here we have **Software Exceptions** (page 875) STATUS\_SXS\_\*, for example:

```
STATUS_SXS_EARLY_DEACTIVATION 0xC015000F
STATUS_SXS_INVALID_DEACTIVATION 0xC0150010

0:000> !analyze -v

[...]

EXCEPTION_RECORD: ffffffff -- (.exr 0xffffffffffff)
ExceptionAddress: 77a54441 (ntdll!RtlDeactivateActivationContext+0x00000154)
ExceptionCode: c015000f
ExceptionFlags: 00000000
NumberParameters: 3
Parameter[0]: 00000000
Parameter[1]: 0056cbe8
Parameter[2]: 0056cc18

EXCEPTION_CODE: (NTSTATUS) 0xc015000f - The activation context being deactivated is not the most recently activated one.

CONTEXT: 003df6c8 -- (.cxr 0x3df6c8)
eax=003df9bc ebx=13050002 ecx=00000000 edx=00000000 esi=0056cbe8 edi=0056cc18
eip=77a54441 esp=003df9b0 ebp=003dfa0c iopl=0 nv up ei pl nz na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000206
ntdll!RtlDeactivateActivationContext+0x154:
77a54441 8b36 mov esi,dword ptr [esi] ds:002b:0056cbe8=0056cbb8
Resetting default scope

STACK_TEXT:
003dfa0c 755aa138 005507d0 13050002 003dfa7c ntdll!RtlDeactivateActivationContext+0x154
003dfa1c 002b1235 00000000 13050002 3a92c68c kernel32!DeactivateActCtx+0x31
003dfa7c 002b13b5 00000001 01f01e98 01f01ec8 TestActCtx!wmain+0x225
003dfac4 75593677 7efde000 003dfb10 77a09f02 TestActCtx!_tmainCRTStartup+0xfa
003dfad0 77a09f02 7efde000 7e35c89d 00000000 kernel32!BaseThreadInitThunk+0xe
003dfb10 77a09ed5 002b140c 7efde000 ffffffff ntdll!__RtlUserThreadStart+0x70
003dfb28 00000000 002b140c 7efde000 00000000 ntdll!__RtlUserThreadStart+0x1b
```

<sup>7</sup> [http://msdn.microsoft.com/en-us/library/aa374153\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/aa374153(v=vs.85).aspx)

The ReactOS code for *RtlDeactivateActivationContext*<sup>8</sup> function suggests the following line of inquiry:

```
0:000> dt _TEB
TestActCtx!_TEB
+0x000 NtTib : _NT_TIB
+0x01c EnvironmentPointer : Ptr32 Void
+0x020 ClientId : _CLIENT_ID
+0x028 ActiveRpcHandle : Ptr32 Void
+0x02c ThreadLocalStoragePointer : Ptr32 Void
+0x030 ProcessEnvironmentBlock : Ptr32 _PEB
+0x034 LastErrorValue : Uint4B
+0x038 CountOfOwnedCriticalSection : Uint4B
+0x03c CsrClientThread : Ptr32 Void
+0x040 Win32ThreadInfo : Ptr32 Void
+0x044 User32Reserved : [26] Uint4B
+0x0ac UserReserved : [5] Uint4B
+0x0c0 WOW32Reserved : Ptr32 Void
+0x0c4 CurrentLocale : Uint4B
+0x0c8 FpSoftwareStatusRegister : Uint4B
+0x0cc SystemReserved1 : [54] Ptr32 Void
+0x1a4 ExceptionCode : Int4B
+0x1a8 ActivationContextStack : ACTIVATION CONTEXT STACK
+0x1bc SpareBytes1 : [24] UChar
+0x1d4 GdiTebBatch : _GDI_TEB_BATCH
+0x6b4 RealClientId : _CLIENT_ID
+0x6bc GdiCachedProcessHandle : Ptr32 Void
+0x6c0 GdiClientPID : Uint4B
+0x6c4 GdiClientTID : Uint4B
+0x6c8 GdiThreadLocalInfo : Ptr32 Void
+0x6cc Win32ClientInfo : [62] Uint4B
+0x7c4 glDispatchTable : [233] Ptr32 Void
+0xb68 glReserved1 : [29] Uint4B
+0xbd0 glReserved2 : Ptr32 Void
+0xbe0 glSectionInfo : Ptr32 Void
+0xbe4 glSection : Ptr32 Void
+0xbe8 glTable : Ptr32 Void
+0xbec glCurrentRC : Ptr32 Void
+0xbf0 glContext : Ptr32 Void
+0xbf4 LastStatusValue : Uint4B
+0xbff8 StaticUnicodeString : _UNICODE_STRING
+0xc00 StaticUnicodeBuffer : [261] Wchar
+0xe0c DeallocationStack : Ptr32 Void
+0xe10 TlsSlots : [64] Ptr32 Void
+0xf10 TlsLinks : _LIST_ENTRY
+0xf18 Vdm : Ptr32 Void
+0xf1c ReservedForNtRpc : Ptr32 Void
+0xf20 DbgSsReserved : [2] Ptr32 Void
+0xf28 HardErrorMode : Uint4B
+0xf2c Instrumentation : [16] Ptr32 Void
+0xf6c WinSockData : Ptr32 Void
```

---

<sup>8</sup> [https://doxygen.reactos.org/de/d9c/sdk\\_2lib\\_2rtl\\_2actctx\\_8c.html#a52533b501a01935d624ca160b7dd7dc7](https://doxygen.reactos.org/de/d9c/sdk_2lib_2rtl_2actctx_8c.html#a52533b501a01935d624ca160b7dd7dc7)

```
+0xf70 GdiBatchCount : Uint4B
+0xf74 InDbgPrint : UChar
+0xf75 FreeStackOnTermination : UChar
+0xf76 HasFiberData : UChar
+0xf77 IdealProcessor : UChar
+0xf78 Spare3 : Uint4B
+0xf7c ReservedForPerf : Ptr32 Void
+0xf80 ReservedForOle : Ptr32 Void
+0xf84 WaitingOnLoaderLock : Uint4B
+0xf88 Wx86Thread : _Wx86ThreadState
[...]

0:000> dt _ACTIVATION_CONTEXT_STACK
TestActCtx!_ACTIVATION_CONTEXT_STACK
+0x000 Flags : Uint4B
+0x004 NextCookieSequenceNumber : Uint4B
+0x008 ActiveFrame : Ptr32 _RTL_ACTIVATION_CONTEXT_STACK_FRAME
+0x00c FrameListCache : _LIST_ENTRY

0:000> dt _RTL_ACTIVATION_CONTEXT_STACK_FRAME
ntdll!_RTL_ACTIVATION_CONTEXT_STACK_FRAME
+0x000 Previous : Ptr32 _RTL_ACTIVATION_CONTEXT_STACK_FRAME
+0x004 ActivationContext : Ptr32 _ACTIVATION_CONTEXT
+0x008 Flags : Uint4B

0:000> dd 0056cc18 14
0056cc18 0056cbe8 0056ca6c 00000028 13050003

0:000> dd 0056cbe8
0056cbe8 0056cbb8 0056c934 00000028 13050002
0056cbf8 00000000 00000000 00000000 00000000
0056cc08 00000000 00000000 00000000 00000000
0056cc18 0056cbe8 0056ca6c 00000028 13050003
0056cc28 00000000 00000000 00000000 00000000
0056cc38 00000000 00000000 00000000 00000000
0056cc48 00000000 00000000 0000000c 00000000
0056cc58 00000000 00000000 00000000 00000000

0:000> dd 0056cbb8
0056cbb8 00000000 0056c7fc 00000028 13050001
0056cbc8 00000000 00000000 00000000 00000000
0056cbd8 00000000 00000000 00000000 00000000
0056cbe8 0056cbb8 0056c934 00000028 13050002
0056cbf8 00000000 00000000 00000000 00000000
0056cc08 00000000 00000000 00000000 00000000
0056cc18 0056cbe8 0056ca6c 00000028 13050003
0056cc28 00000000 00000000 00000000 00000000
```

We see that a different cookie was found on top of the thread activation stack and the code raised the runtime exception.

## Active Thread

### Linux

Here we publish a Linux variant of **Active Thread** pattern that was previously introduced for Mac OS X (page 64) and Windows (page 66). Basically, it is a thread that is not waiting, sleeping, or suspended (most threads are). However, from a memory dump, it is not possible to find out whether it was **Spiking Thread** (page 885) at the dump generation time (unless we have a set of memory snapshots and in each one we have the same or similar backtrace) and we don't have any **Paratext** (page 783) with CPU consumption stats for threads. For example, in one core dump we have this thread:

```
(gdb) info threads
Id Target Id Frame
6 Thread 0x7f560d467700 (LWP 3483) 0x0000000004324a9 in clone ()
5 Thread 0x7f560c465700 (LWP 3485) 0x00000000042fe31 in nanosleep ()
4 Thread 0x7f560bc64700 (LWP 3486) 0x00000000042fe31 in nanosleep ()
3 Thread 0x7f560b463700 (LWP 3487) 0x00000000042fe31 in nanosleep ()
2 Thread 0x18b9860 (LWP 3482) 0x00000000042fe31 in nanosleep ()
1 Thread 0x7f560cc66700 (LWP 3484) 0x00000000042fe31 in nanosleep ()
```

Thread #6 is not waiting so we inspect its back trace:

```
(gdb) thread 6
[Switching to thread 6 (Thread 0x7f560d467700 (LWP 3483))]
#0 0x0000000004324a9 in clone ()

(gdb) bt
#0 0x0000000004324a9 in clone ()
#1 0x000000000401560 in ?? () at pthread_create.c:217
#2 0x00007f560d467700 in ?? ()
#3 0x0000000000000000 in ?? ()

(gdb) x/i 0x4324a9
=> 0x4324a9 : test %rax,%rax
```

Perhaps the core dump was saved at the thread creation time.

## Mac OS X

This pattern was introduced in *Accelerated Mac OS X Core Dump Analysis*<sup>9</sup> training. Basically, it is a thread that is not waiting or suspended (most threads are). However, from a memory dump it is not possible to find out whether it was **Spiking Thread** (page 885) at the dump generation time (unless we have a set of memory snapshots and in each one we have the same or similar backtrace) and we don't have any **Paratext** (page 780) with CPU consumption stats for threads. For example, in one core dump we have these threads:

```
(gdb) info threads
12 0x98c450ee in __workq_kernreturn ()
11 0x98c4280e in semaphore_wait_trap ()
10 0x98c448e2 in __psynch_cvwait ()
9 0x00110171 in std::Rb_tree<int, std::pair<int const, _iCapture*>, std::_Select1st<std::pair<int const, _iCapture*> >, std::less<int>, std::allocator<std::pair<int const, _iCapture*> >::find ()
8 0x98c428e6 in mach_wait_until ()
7 0x98c448e2 in __psynch_cvwait ()
6 0x98c427d2 in mach_msg_trap ()
5 0x98c427d2 in mach_msg_trap ()
4 0x98c428e6 in mach_wait_until ()
3 0x98c427d2 in mach_msg_trap ()
2 0x98c459ae in kevent ()
* 1 0x014bcee0 in cgGLGetLatestProfile ()
```

Threads #9 and #1 are not waiting so we inspect their back traces:

```
(gdb) bt
#0 0x014bcee0 in cgGLGetLatestProfile ()
#1 0x99060dd5 in exit ()
#2 0x001ef859 in os_exit ()
#3 0x001dc873 in luaD_precall ()
#4 0x001e7d9e in luaV_execute ()
#5 0x001dc18b in luaD_rawrunprotected ()
#6 0x001dced4 in lua_resume ()
#7 0x0058a526 in ticLuaManager::executeProgram ()
#8 0x005a09af in ticLuaScript::_execute ()
#9 0x003a6480 in darcScript::execute ()
#10 0x003af4d8 in darcTimeline::execute ()
#11 0x0034a2ba in darcSequenceur::executeAll ()
#12 0x0036904b in darcEventManager::ExecuteEventHandler ()
#13 0x003a37d2 in darcScene::process ()
#14 0x0034a2ba in darcSequenceur::executeAll ()
#15 0x0036904b in darcEventManager::ExecuteEventHandler ()
#16 0x00343ec0 in darcContext::process ()
#17 0x00347339 in darcContext::main ()
#18 0x003cf73d in darcPlayerImpl::renderOneFrame ()
#19 0x003cf078 in darcPlayerImpl::render ()
#20 0x000b1f6f in Run ()
```

---

<sup>9</sup> <http://www.patterndiagnostics.com/accelerated-macosx-core-dump-analysis-book>

```
#21 0x000b1fe9 in tiMain ()
#22 0x000c73ee in main ()

(gdb) thread 9
[Switching to thread 9 (core thread 8)]
0x00110171 in std::_Rb_tree<int, std::pair<int const, _iCapture*>, std::_Select1st<std::pair<int const, _iCapture*> >, std::less<int>, std::allocator<std::pair<int const, _iCapture*> >>::find ()

(gdb) bt
#0 0x00110171 in std::_Rb_tree<int, std::pair<int const, _iCapture*>, std::_Select1st<std::pair<int const, _iCapture*> >, std::less<int>, std::allocator<std::pair<int const, _iCapture*> >>::find ()
#1 0x0010f936 in ticVideoManager::isPaused ()
#2 0x00201801 in ticMLT_VideoCapture::Execute ()
#3 0x0020aa0b in ticModuleGraph::runOnce ()
#4 0x002632be in TrackingApp::ProcessTracking ()
#5 0x005b2f5d in ticMLTTracking::processInternal ()
#6 0x005b322d in ticMLTTracking::processThread ()
#7 0x005b36f3 in trackingThread ()
#8 0x004eaf1e in ticThread::threadFunc ()
#9 0x99023557 in _pthread_start ()
#10 0x9900dcee in thread_start ()
```

Windows equivalent would be a process memory dump which doesn't have any information saved for !runaway WinDbg command.

## Windows

We already introduced **Active Thread** (page 63) pattern variant for Mac OS X. Here we provide an example for Windows. Unless we have **Eevental Dumps** (page 328), **Active Threads** in Windows are usually threads from **Busy System** (page 99) or **Spiking Threads** (page 888), and, therefore, represent an abnormal behavior since most threads are waiting or calling some API. For **Eevental Dumps** they may be just normal threads:

```
0:000> r
rax=0000000000000006 rbx=0000000000000003 rcx=0000000000000018
rdx=0000000000000000 rsi=00000000028c601 rdi=0000000002bee25e
rip=000007feff1a5a09 rsp=00000000028c380 rbp=0000000000000000
r8=0000000000000000 r9=00000000001653a0 r10=000000000000000e
r11=000000000000000a r12=0000000000000006 r13=000000000028ca38
r14=0000000002bec888 r15=0000000000173630
iopl=0 nv up ei pl nz ac po nc
cs=0033 ss=002b ds=002b es=002b fs=0053 gs=002b ef1=00000216
usp10!otlChainingLookup::apply+0x2f9:
000007fe`ff1a5a09 498d0c06 lea rcx,[r14+rax]

0:000> k
# Child-SP RetAddr Call Site
00 0000000`0028c380 000007fe`ff19f4f2 usp10!otlChainingLookup::apply+0x2f9
01 0000000`0028c4b0 000007fe`ff19e777 usp10!ApplyLookup+0x592
02 0000000`0028c5a0 000007fe`ff19a634 usp10!ApplyFeatures+0x777
03 0000000`0028c860 000007fe`ff181800 usp10!SubstituteOtlGlyphs+0x224
04 0000000`0028c910 000007fe`ff174cc0 usp10!GenericEngineGetGlyphs+0x1000
05 0000000`0028ccb0 000007fe`ff1389c5 usp10!ShlShape+0x7a0
06 0000000`0028ced0 000007fe`ff147363 usp10!ScriptShape+0x205
07 0000000`0028cf70 000007fe`ff148ac9 usp10!RenderItemNoFallback+0x433
08 0000000`0028d030 000007fe`ff148d86 usp10!RenderItemWithFallback+0x129
09 0000000`0028d080 000007fe`ff14a5f7 usp10!RenderItem+0x36
0a 0000000`0028d0d0 000007fe`ff13b2c9 usp10!ScriptStringAnalyzeGlyphs+0x277
0b 0000000`0028d170 000007fe`fdd616bf usp10!ScriptStringAnalyse+0x399
0c 0000000`0028d1f0 000007fe`fdd614cc lpk!LpkCharsetDraw+0x4eb
0d 0000000`0028d3b0 0000000`774e85c5 lpk!LpkDrawTextEx+0x68
0e 0000000`0028d420 0000000`774e865c user32!DT_DrawStr+0xa6
0f 0000000`0028d4c0 0000000`774e826c user32!DT_DrawJustifiedLine+0xa6
10 0000000`0028d530 0000000`774e6cc8 user32!DrawTextExWorker+0x442
11 0000000`0028d640 000007fe`fbdb840d1 user32!DrawTextW+0x57
12 0000000`0028d6b0 000007fe`fbdb83e49 comctl32!CLVView::_ComputeLabelSizeWorker+0x1d1
13 0000000`0028da40 000007fe`fbdb8cc48 comctl32!CLVView::v_RecomputeLabelSize+0x1f9
14 0000000`0028dd70 000007fe`fbda9d24 comctl32!CLVListView::v_DrawItem+0x284
15 0000000`0028e110 000007fe`fbda9773b comctl32!CLVDrawItemManager::DrawItem+0x4c0
16 0000000`0028e170 000007fe`fbdb95f8e comctl32!CLVDrawManager::_PaintItems+0x3df
17 0000000`0028e3b0 000007fe`fbdb95e87 comctl32!CLVDrawManager::_PaintWorkArea+0xda
18 0000000`0028e430 000007fe`fbdb95cff comctl32!CLVDrawManager::_OnPaintWorkAreas+0x147
19 0000000`0028e4c0 000007fe`fbdb06f1b comctl32!CLVDrawManager::_OnPaint+0x14b
1a 0000000`0028e570 000007fe`fbdb06011 comctl32!CListView::WndProc+0xebf
1b 0000000`0028e770 0000000`774e9bd1 comctl32!CListView::s_WndProc+0x6cd
1c 0000000`0028e7d0 0000000`774e3bfc user32!UserCallWinProcCheckWow+0x1ad
1d 0000000`0028e890 0000000`774e3b78 user32!CallWindowProcAorW+0xdc
1e 0000000`0028e8e0 000007fe`fbca6215 user32!CallWindowProcW+0x18
1f 0000000`0028e920 000007fe`fbca69a0 comctl32!CallOriginalWndProc+0x1d
20 0000000`0028e960 000007fe`fbca6768 comctl32!CallNextSubclassProc+0x8c
```

```

21 00000000`0028e9e0 000007fe`fde1096a comctl32!DefSubclassProc+0x7
22 00000000`0028ea30 000007fe`fdde9df4 shell32!DefSubclassProc+0x56
23 00000000`0028ea60 000007fe`fbca69a0 shell32!CListViewHost::s_ListViewSubclassWndProc+0x267
24 00000000`0028eb40 000007fe`fbca6877 comctl32!CallNextSubclassProc+0x8c
25 00000000`0028ebc0 00000000`774e9bd1 comctl32!MasterSubclassProc+0xe7
26 00000000`0028ec60 00000000`774e72cb user32!UserCallWinProcCheckWow+0x1ad
27 00000000`0028ed20 00000000`774e6829 user32!DispatchClientMessage+0xc3
28 00000000`0028ed80 00000000`776211f5 user32!_fnDWORD+0x2d
29 00000000`0028ede0 00000000`774e6e5a ntdll!KiUserCallbackDispatcherContinue
2a 00000000`0028ee68 00000000`774e6e6c user32!NtUserDispatchMessage+0xa
2b 00000000`0028ee70 00000000`774e67c2 user32!DispatchMessageWorker+0x55b
2c 00000000`0028eef0 000007fe`fbcc34a4 user32!IsDialogMessageW+0x153
2d 00000000`0028ef80 000007fe`fbcc583f comctl32!Prop_IsDialogMessage+0x1f0
2e 00000000`0028eff0 000007fe`fbcc5c05 comctl32!_RealPropertySheet+0x31b
2f 00000000`0028f0c0 000007fe`ff214e68 comctl32!_PropertySheet+0x55
30 00000000`0028f100 000007fe`ff214bb1 comdlg32!Print_InvokePropertySheets+0x2c6
31 00000000`0028f660 000007fe`ff21499c comdlg32!PrintDlgExX+0x2be
32 00000000`0028f6c0 00000000`ffec250f comdlg32!PrintDlgExW+0x38
33 00000000`0028f730 00000000`ffec242b notepad!GetPrinterDCviaDialog+0xab
34 00000000`0028f7f0 00000000`ffec23e8 notepad!PrintIt+0x46
35 00000000`0028fb70 00000000`ffec14eb notepad!NPCommand+0xdb
36 00000000`0028fcfa0 00000000`774e9bd1 notepad!NPWndProc+0x540
37 00000000`0028fce0 00000000`774e98da user32!UserCallWinProcCheckWow+0x1ad
38 00000000`0028fda0 00000000`ffec10bc user32!DispatchMessageWorker+0x3b5
39 00000000`0028fe20 00000000`ffec133c notepad!WinMain+0x16f
3a 00000000`0028fea0 00000000`773c59ed notepad!DisplayNonGenuineDlgWorker+0x2da
3b 00000000`0028ff60 00000000`775fc541 kernel32!BaseThreadInitThunk+0xd
3c 00000000`0028ff90 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

```

We see the thread is active, in the middle of the function. For comparison, the next two threads are waiting and calling API respectively:

```

0:000> ~1k
# Child-SP          RetAddr          Call Site
00 00000000`02edf748 00000000`775eb037 ntdll!NtWaitForMultipleObjects+0xa
01 00000000`02edf750 00000000`773c59ed ntdll!TppWaiterPThread+0x14d
02 00000000`02edf9f0 00000000`775fc541 kernel32!BaseThreadInitThunk+0xd
03 00000000`02edfa20 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

0:000> ~2k
# Child-SP          RetAddr          Call Site
00 00000000`0344e048 000007fe`fd3f403e ntdll!NtUnmapViewOfSection+0xa
01 00000000`0344e050 00000000`774e2edf KERNELBASE!FreeLibrary+0xa4
02 00000000`0344e080 000007fe`fdddaab3 user32!PrivateExtractIconsW+0x34b
03 00000000`0344e5a0 000007fe`fdddac28 shell32!SHPrivateExtractIcons+0x50a
04 00000000`0344e870 000007fe`fdde34b4 shell32!SHDefExtractIconW+0x254
05 00000000`0344eb60 000007fe`fdde3435 shell32!CEExtractIcon:::_ExtractW+0xcd
06 00000000`0344ebc0 000007fe`fdf0d529 shell32!CEExtractIconBase::Extract+0x21
07 00000000`0344ec20 000007fe`fdf0d2da shell32!IExtractIcon_Extract+0x43
08 00000000`0344ec60 000007fe`fddffff0 shell32!_GetILIndexGivenPXIcon+0x22e
09 00000000`0344f100 000007fe`fdde27a4 shell32!_GetILIndexFromItem+0x87
0a 00000000`0344f1a0 000007fe`fdedb6506 shell32!SHGetIconIndexFromPIDL+0x66
0b 00000000`0344f1d0 000007fe`fdedb186 shell32!MapIDListToIconILIndex+0x52
0c 00000000`0344f250 000007fe`fdcdc54c shell32!CLoadSystemIconTask::InternalResumeRT+0x110
0d 00000000`0344f2e0 000007fe`fde0efcb shell32!CRunnableTask::Run+0xda

```

```

0e 00000000`0344f310 000007fe`fde12b56 shell32!CShellTask::TT_Run+0x124
0f 00000000`0344f340 000007fe`fde12cb2 shell32!CShellTaskThread::ThreadProc+0x1d2
10 00000000`0344f3e0 000007fe`ff2b3843 shell32!CShellTaskThread::s_ThreadProc+0x22
11 00000000`0344f410 00000000`775f15db shlwapi!ExecuteWorkItemThreadProc+0xf
12 00000000`0344f440 00000000`775f0c56 ntdll!RtlpTpWorkCallback+0x16b
13 00000000`0344f520 00000000`773c59ed ntdll!TppWorkerThread+0x5ff
14 00000000`0344f820 00000000`775fc541 kernel32!BaseThreadInitThunk+0xd
15 00000000`0344f850 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

0:000> ub ntdll!NtUnmapViewOfSection+0xa
ntdll!NtAccessCheckAndAuditAlarm:
00000000`77621540 4c8bd1      mov     r10,rcx
00000000`77621543 b826000000  mov     eax,26h
00000000`77621548 0f05      syscall
00000000`7762154a c3      ret
00000000`7762154b 0f1f440000  nop     dword ptr [rax+rax]
ntdll!NtUnmapViewOfSection:
00000000`77621550 4c8bd1      mov     r10,rcx
00000000`77621553 b827000000  mov     eax,27h
00000000`77621558 0f05      syscall

```

Our **Active Thread** is not **Spiking Thread** since CPU consumption is minimal:

```

0:000> !runaway f
User Mode Time
Thread      Time
0:1ca4      0 days 0:00:00.171
11:1fb0      0 days 0:00:00.000
10:f98       0 days 0:00:00.000
9:eb8       0 days 0:00:00.000
8:1b80      0 days 0:00:00.000
7:139c      0 days 0:00:00.000
6:1d9c      0 days 0:00:00.000
5:1b44      0 days 0:00:00.000
4:1edc      0 days 0:00:00.000
3:830       0 days 0:00:00.000
2:1638      0 days 0:00:00.000
1:1ab0      0 days 0:00:00.000
Kernel Mode Time
Thread      Time
0:1ca4      0 days 0:00:00.421
11:1fb0      0 days 0:00:00.000
10:f98       0 days 0:00:00.000
9:eb8       0 days 0:00:00.000
8:1b80      0 days 0:00:00.000
7:139c      0 days 0:00:00.000
6:1d9c      0 days 0:00:00.000
5:1b44      0 days 0:00:00.000
4:1edc      0 days 0:00:00.000
3:830       0 days 0:00:00.000
2:1638      0 days 0:00:00.000
1:1ab0      0 days 0:00:00.000
Elapsed Time
Thread      Time
11:1fb0      24692 days 13:29:46.335

```

0:1ca4	0 days 0:02:39.671
1:1ab0	0 days 0:01:48.239
2:1638	0 days 0:01:18.837
4:1edc	0 days 0:01:18.697
3:830	0 days 0:01:18.697
5:1b44	0 days 0:01:18.497
6:1d9c	0 days 0:01:18.387
7:139c	0 days 0:01:15.957
8:1b80	0 days 0:01:14.397
9:eb8	0 days 0:01:01.485
10:f98	0 days 0:00:39.849

However, the huge *Elapsed Time* for the thread #11 (most likely the value is uninitialized) and its stack trace suggest that the dump was saved on *Create Thread* debugging event by a debugger:

```
0:000> ~11k
# Child-SP          RetAddr          Call Site
00 00000000`0666fb08 00000000`00000000 ntdll!RtlUserThreadStart
```

## Activity Resonance

This pattern is observed when two products from different vendors compete in some functional domain such as malware detection. In the example below *ApplicationA* and *AVDriverA* modules belong to *Vendor A*, and *AV-B* module belongs to *Vendor B*. Both threads are **Spiking Threads** (page 885) blocking all other activity in the system:

```
0: kd> !running

System Processors: (0000000000000003)
Idle Processors: (0000000000000000) (0000000000000000) (0000000000000000) (0000000000000000)

Prcbs          Current          Next
0   fffff80001845e80  ffffffa8004350060  .....
1   fffff880009c4180  ffffffa80028e7060  .....

0: kd> !thread ffffffa8004350060 1f
THREAD ffffffa8004350060  Cid 14424.14b34  Teb: 000000007efdb000 Win32Thread: fffff900c1d32c30 RUNNING on
processor 0
Not impersonating
DeviceMap          fffff8a00148fe80
Owning Process    ffffffa8003d6cb30  Image:      ApplicationA.exe
Attached Process  N/A           Image:      N/A
Wait Start TickCount 10568630  Ticks: 0
Context Switch Count 345          LargeStack
UserTime           00:02:21.360
KernelTime         01:09:32.130
Win32 Start Address ApplicationA!mainCRTStartup (0x000000000404c1b)
Stack Init fffff88006c71db0 Current fffff88006c71670
Base fffff88006c72000 Limit fffff88006c6a000 Call 0
Priority 9 BasePriority 8 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
Child-SP          RetAddr        Call Site
ffffff880`06c70ec0 fffff880`0197d53c AVDriverA+0x15d69
ffffff880`06c70f10 fffff880`01988556 AVDriverA+0x1453c
ffffff880`06c70fd0 fffff880`019886a8 AVDriverA+0x1f556
ffffff880`06c71000 fffff880`0198ebfd AVDriverA+0x1f6a8
ffffff880`06c71060 fffff880`019bf4f2 nt! ?? ::NNGAKEGL::`string'+0x2a6fd
ffffff880`06c711e0 fffff880`019c3385 nt!PspCreateThread+0x246
ffffff880`06c71460 fffff880`016d28d3 nt!NtCreateThreadEx+0x25d
ffffff880`06c71bb0 00000000`76e61d9a nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ fffff880`06c71c20)
00000000`0008e178 00000000`74990411 ntdll!ZwCreateThreadEx+0xa
00000000`0008e180 00000000`7497cf87 wow64!whNtCreateThreadEx+0x815
00000000`0008e350 00000000`748c2776 wow64!Wow64SystemServiceEx+0xd7
00000000`0008ec10 00000000`7497d07e wow64cpu!TurboDispatchJumpAddressEnd+0x2d
00000000`0008ecd0 00000000`7497c549 wow64!RunCpuSimulation+0xa
00000000`0008ed20 00000000`76e54956 wow64!Wow64LdrpInitialize+0x429
00000000`0008f270 00000000`76e51a17 ntdll!LdrpInitializeProcess+0x17e4
00000000`0008f760 00000000`76e3c32e ntdll! ?? ::FNODOBFM::`string'+0x29220
00000000`0008f7d0 00000000`00000000 ntdll!LdrInitializeThunk+0xe
```

```
0: kd> !thread ffffffa80028e7060 1f
THREAD ffffffa80028e7060 Cid 0dc4.0e5c Peb: 000000007efa4000 Win32Thread: 0000000000000000 RUNNING on
processor 1
Not impersonating
DeviceMap fffff8a000008b30
Owning Process ffffffa8002817060 Image: AV-B.exe
Attached Process N/A Image: N/A
Wait Start TickCount 10568617 Ticks: 13 (0:00:00:00.203)
Context Switch Count 1763138
UserTime 00:04:26.765
KernelTime 03:09:31.140
Win32 Start Address AV-B (0x0000000004289f2)
Stack Init fffff88003b88db0 Current fffff88003b88900
Base fffff88003b89000 Limit fffff88003b83000 Call 0
Priority 15 BasePriority 15 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
Child-SP RetAddr Call Site
ffffff880`03b88660 fffff800`019919a9 nt!ObReferenceObjectSafe+0xf
ffffff880`03b88690 fffff800`01991201 nt!PsGetNextProcess+0x81
ffffff880`03b886e0 fffff800`019dcef6 nt!ExpGetProcessInformation+0x774
ffffff880`03b88830 fffff800`019dd949 nt!ExpQuerySystemInformation+0xfb4
ffffff880`03b88be0 fffff800`016d28d3 nt!NtQuerySystemInformation+0x4d
ffffff880`03b88c20 00000000`76e6167a nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ fffff880`03b88c20)
00000000`0118e708 00000000`74987da7 ntdll!NtQuerySystemInformation+0xa
00000000`0118e710 00000000`74988636 wow64!whNT32QuerySystemProcessInformationEx+0x93
00000000`0118e760 00000000`7498a0e9 wow64!whNtQuerySystemInformation_SpecialQueryCase+0x466
00000000`0118e800 00000000`7497cf87 wow64!whNtQuerySystemInformation+0xf1
00000000`0118e840 00000000`748c2776 wow64!Wow64SystemServiceEx+0xd7
00000000`0118f100 00000000`7497d07e wow64cpu!TurboDispatchJumpAddressEnd+0x2d
00000000`0118f1c0 00000000`7497c549 wow64!RunCpuSimulation+0xa
00000000`0118f210 00000000`76e8e707 wow64!Wow64LdrpInitialize+0x429
00000000`0118f760 00000000`76e3c32e ntdll! ?? ::FNODOBFM::`string'+0x29364
00000000`0118f7d0 00000000`00000000 ntdll!LdrInitializeThunk+0xe
```

## Affine Thread

Setting a thread affinity mask to a specific processor or core makes that thread running in a single processor environment from that thread point of view. It is always scheduled to run on that processor. This potentially creates a problem found in real single processor environments if the processor runs another higher priority thread (**Thread Starvation** pattern, page 1004) or loops at dispatch level IRQL (**Dispatch Level Spin** pattern, page 240).

Here is one example. A dual core laptop was hanging, and kernel memory dump revealed the following **Wait Chain** pattern (page 1089):

```
Resource @ nt!PopPolicyLock (0x80563080)      Exclusively owned
  Contention Count = 32
  NumberOfExclusiveWaiters = 9
  Threads: 8b3b08b8-01<*>
  Threads Waiting On Exclusive Access:
    872935f0      8744cb30      87535da8      8755a6b0
    8588dba8      8a446c10      85891c50      87250020
    8a6e7da8
```

The thread 8b3b08b8 blocked other 9 threads and had the following stack trace:

```
0: kd> !thread 8b3b08b8 1f
THREAD 8b3b08b8  Cid 0004.002c  Teb: 00000000 Win32Thread: 00000000 READY
Not impersonating
DeviceMap          e1009248
Owning Process    8b3b2830  Image:           System
Wait Start TickCount 44419       Ticks: 8744 (0:00:02:16.625)
Context Switch Count 4579
UserTime           00:00:00.000
KernelTime         00:00:01.109
Start Address nt!ExpWorkerThread (0x8053867e)
Stack Init bad00000 Current bacffcb0 Base bad00000 Limit bacfd000 Call 0
Priority 15 BasePriority 12 PriorityDecrement 3 DecrementCount 16
ChildEBP RetAddr
bacffcc8 804fd2c9 nt!KiUnlockDispatcherDatabase+0x9e
bacffcdc 8052a16f nt!KeSetSystemAffinityThread+0x5b
bacffd04 805caf03 nt!PopCompositeBatteryUpdateThrottleLimit+0x2d
bacffd24 805ca767 nt!PopCompositeBatteryDeviceHandler+0x1c5
bacffd3c 80529d3b nt!PopPolicyWorkerMain+0x25
bacffd7c 8053876d nt!PopPolicyWorkerThread+0xbff
bacffdac 805cff64 nt!ExpWorkerThread+0xef
bacffddc 805460de nt!PspSystemThreadStartup+0x34
00000000 00000000 nt!KiThreadStartup+0x16
```

Note this function and its first parameter (shown in smaller font for visual clarity):

```
0: kd> !thread 8b3b08b8
...
bacffcdc 8052a16f 00000002 8a5b8cd8 00000030 nt!KeSetSystemAffinityThread+0x5b
...
```

The first parameter is KAFFINITY mask, and 0x2 is 0y10 (binary) which is the second core. This thread had been already set to run on that core only:

```
0: kd> dt _KTHREAD 8b3b08b8
nt!_KTHREAD
+0x000 Header : _DISPATCHER_HEADER
...
+0x124 Affinity : 2
...
```

Let's look at our second core:

```
0: kd> ~1s

1: kd> kL 100
ChildEBP RetAddr
a8f00618 acd21947 hal!KeAcquireInStackQueuedSpinLock+0x43
a8f00618 acd21947 tcpip!IndicateData+0x98
a8f00684 acd173e5 tcpip!IndicateData+0x98
a8f0070c acd14ef5 tcpip!TCPRecv+0xbb0
a8f0076c acd14b19 tcpip!DeliverToUser+0x18e
a8f007e8 acd14836 tcpip!DeliverToUserEx+0x95e
a8f008a0 acd13928 tcpip!IPRcvPacket+0x6cb
a8f008e0 acd13853 tcpip!ARPRcvIndicationNew+0x149
a8f0091c ba56be45 tcpip!ARPRcvPacket+0x68
a8f00970 b635801d NDIS!ethFilterDprIndicateReceivePacket+0x307
a8f00984 b63581b4 psched!PsFlushReceiveQueue+0x15
a8f009a8 b63585f9 psched!PsEnqueueReceivePacket+0xda
a8f009c0 ba56c8ed psched!C1ReceiveComplete+0x13
a8f009d8 b7defdb5 NDIS!EthFilterDprIndicateReceiveComplete+0x7c
a8f00a08 b7df0f78 driverA+0x17db5
a8f00a64 ba55ec2c driverA+0x18f78
a8f00a88 b6b0962c NDIS!ndisMSendCompleteX+0x8d
a8f00a9c b6b0a36d driverB+0x62c
a8f00ab8 ba55e88f driverB+0x136d
a8f00ae0 b7de003c NDIS!NdisReturnPackets+0xe9
a8f00af0 ba55e88f driverA+0x803c
a8f00b18 b6358061 NDIS!NdisReturnPackets+0xe9
a8f00b30 ba55e88f psched!MpReturnPacket+0x3b
a8f00b58 acc877cc NDIS!NdisReturnPackets+0xe9
87749da0 00000000 afd!AfdReturnBuffer+0xe1

1: kd> r
eax=a8f005f8 ebx=a8f00624 ecx=8a9862ed edx=a8f00b94 esi=874e2ed0 edi=8a9862d0
eip=806e6a33 esp=a8f005ec ebp=a8f00618 iopl=0 nv up ei pl nz na po nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00000202
hal!KeAcquireInStackQueuedSpinLock+0x43:
806e6a33 74ee je hal!KeAcquireInStackQueuedSpinLock+0x33 (806e6a23) [br=0]
```

```
1: kd> !running

System Processors 3 (affinity mask)
Idle Processors 1

Prcb      Current   Next
1 bab38120 8a0c8ae8 8b3a7318 .....
```

We see the thread 8a0c8ae8 had been spinning on the second core for more than 2 minutes:

```
1: kd> !thread 8a0c8ae8 1f
THREAD 8a0c8ae8 Cid 0660.0124 Peb: 7ffd7000 Win32Thread: e338c498 RUNNING on processor 1
IRP List:
  8a960008: (0006,01b4) Flags: 00000900 Mdl: 87535908
Not impersonating
DeviceMap          e2f155b8
Owning Process    87373020     Image:           APPLICATION.EXE
Wait Start TickCount 43918       Ticks: 9245 (0:00:02:24.453)
Context Switch Count 690         LargeStack
UserTime           00:00:00.000
KernelTime         00:02:24.453
...
```

Its kernel time looks consistent with the starved thread waiting time:

```
0: kd> !thread 8b3b08b8 1f
THREAD 8b3b08b8 Cid 0004.002c Peb: 00000000 Win32Thread: 00000000 READY
Not impersonating
DeviceMap          e1009248
Owning Process    8b3b2830     Image:           System
Wait Start TickCount 44419       Ticks: 8744 (0:00:02:16.625)
...
```

For comparison, the spinning thread has affinity mask 0x11 (0x3) which means that it could be scheduled to run on both cores:

```
0: kd> dt _KTHREAD 8a0c8ae8
nt!_KTHREAD
  +0x000 Header        : _DISPATCHER_HEADER
...
  +0x124 Affinity      : 3
...
```

## Annotated Disassembly

### JIT .NET Code

When disassembling JIT code, it is good to see annotated function calls with full type and token information:

```
0:000> !CLRStack
OS Thread Id: 0xbff8 (0)
ESP      EIP
001fef90 003200a4 ClassMain.DoWork()
001fef94 00320082 ClassMain.Main(System.String[])
001ff1b0 79e7c74b [GCFrame: 001ff1b0]

0:000> !U 00320082
Normal JIT generated code
ClassMain.Main(System.String[])
Begin 00320070, size 13
00320070 b960300d00    mov ecx,0D3060h (MT: ClassMain)
00320075 e8a21fdaff    call 000c201c (JitHelp: CORINFO_HELP_NEWSFAST)
0032007a 8bc8          mov ecx, eax
0032007c ff159c300d00  call dword ptr ds:[0D309Ch] (ClassMain.DoWork(), mdToken: 06000002)
>>> 00320082 c3        ret
```

However, this doesn't work when we disable the output of raw bytes:

```
0:000> !U 00320082
Normal JIT generated code
ClassMain.Main(System.String[])
Begin 00320070, size 13
00320070 mov    ecx,0D3060h
00320075 call   000c201c
0032007a mov    ecx, eax
0032007c call   dword ptr ds:[0D309Ch]
>>> 00320082 ret
```

Here we can still double check JIT-ed function calls manually:

```
0:000> dd 0D309Ch 11
000d309c 00320098

0:000> !IP2MD 00320098
MethodDesc: 000d3048
Method Name: ClassMain.DoWork()
Class:       000d1180
MethodTable: 000d3060
mdToken:     06000002
Module:      000d2c3c
IsJitted:   yes
m_CodeOrIL:  00320098
```

**B**

## Blocked DPC

In this pattern, we have blocked per-processor Deferred Procedure Call<sup>10</sup> queues because of threads running on processors with IRQL > DISPATCH\_LEVEL. For example, on the processor 11 (0x0b):

```
11: kd> !dpcs
CPU Type KDPC Function
3: Normal : 0x8accacec 0xf710567a DriverA

5: Normal : 0x89f449e4 0xf595b83a DriverB

7: Normal : 0x8a63664c 0xf59e3f04 USBPORT!USBPORT_IsrDpc

11: Normal : 0x8acb2cec 0xf710567a DriverA
11: Normal : 0x8b5e955c 0xf73484e6 ACPI!ACPIInterruptServiceRoutineDPC

11: kd> !thread
THREAD 89806428 Cid 0934.0944 Peb: 7ffdb000 Win32Thread: bc17dda0 RUNNING on processor b
Not impersonating
DeviceMap e1002258
Owning Process 89972290 Image: ApplicationA.exe
Attached Process N/A Image: N/A
Wait Start TickCount 2863772 Ticks: 368905 (0:01:36:04.140)
Context Switch Count 145085 LargeStack
UserTime 00:00:00.015
KernelTime 01:36:04.203
Win32 Start Address MSVCR90!_threadstartex (0x7854345e)
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init f3f63000 Current f3f62c4c Base f3f63000 Limit f3f5f000 Call 0
Priority 10 BasePriority 10 PriorityDecrement 0
ChildEBP RetAddr Args to Child
f777d3b0 f3f62d28 00000010 00000000 00000000 hal!KeAcquireInStackQueuedSpinLockRaiseToSynch+0x36
WARNING: Frame IP not in any known module. Following frames may be wrong.
f777d3b4 00000000 00000000 00000000 00000000 0xf3f62d28
```

---

<sup>10</sup> [http://en.wikipedia.org/wiki/Deferred\\_Procedure\\_Call](http://en.wikipedia.org/wiki/Deferred_Procedure_Call)

## Blocked Queue

### LPC/ALPC

We provide an example of an ALPC port here. If we see an LPC/ALPC wait chain (page 1097) endpoint or just have a message address (and optionally a port address), we can check the port queue length. For example, in a frozen system we have this WinDbg output:

```
THREAD ffffffa8009db7160 Cid 03b0.2ec0 Teb: 000007fffffd5000 Win32Thread: 0000000000000000 WAIT:
(WrLpcReply) UserMode Non-Alertable
    ffffffa8009db7520 Semaphore Limit 0x1
Waiting for reply to ALPC Message ffffff8a000dbc6650 : queued at port ffffffa800577ee60 : owned by process
ffffffa80056ddb30
Not impersonating
DeviceMap          ffffff8a000008b30
Owning Process    ffffffa8005691b30      Image:       ServiceA.exe
Attached Process   N/A           Image:       N/A
Wait Start TickCount 39742808      Ticks: 3469954 (0:15:02:11.629)
Context Switch Count 9
UserTime           00:00:00.000
KernelTime         00:00:00.000
Win32 Start Address 0x0000000076cd8e70
Stack Init ffffff8800bf60db0 Current ffffff8800bf60620
Base ffffff8800bf61000 Limit ffffff8800bf5b000 Call 0
Priority 10 BasePriority 9 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
Kernel stack not resident.
Child-SP RetAddr Call Site
ffffff880`0bf60660 ffffff800`016de992 nt!KiSwapContext+0x7a
ffffff880`0bf607a0 ffffff800`016e0cff nt!KiCommitThreadWait+0x1d2
ffffff880`0bf60830 ffffff800`016f5d1f nt!KeWaitForSingleObject+0x19f
ffffff880`0bf608d0 ffffff800`019ddac6 nt!AlpcpSignalAndWait+0x8f
ffffff880`0bf60980 ffffff800`019dba50 nt!AlpcpReceiveSynchronousReply+0x46
ffffff880`0bf609e0 ffffff800`019d8fc0 nt!AlpcpProcessSynchronousRequest+0x33d
ffffff880`0bf60b00 ffffff800`016d6993 nt!NtAlpcSendWaitReceivePort+0x1ab
ffffff880`0bf60bb0 00000000`76d105aa nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ ffffff880`0bf60c20)
00000000`01efe638 000007fe`fec0aa76 ntdll!ZwAlpcSendWaitReceivePort+0xa
00000000`01efe640 000007fe`fecacb64 RPCRT4!LRPC_CCALL::SendReceive+0x156
00000000`01efe700 000007fe`fecacd55 RPCRT4!NdrpClientCall3+0x244
00000000`01efe9c0 000007fe`fcbbf18a1 RPCRT4!NdrClientCall3+0xf2
[...]
```

```
0: kd> !alpc /m fffff8a00dbc6650
Message @ fffff8a00dbc6650
  MessageID          : 0x0720 (1824)
  CallbackID         : 0x257C575 (39306613)
  SequenceNumber     : 0x00000002 (2)
  Type               : LPC_REQUEST
  DataLength         : 0x0044 (68)
  TotalLength        : 0x006C (108)
  Canceled           : No
  Release             : No
  ReplyWaitReply     : No
  Continuation        : Yes
  OwnerPort          : ffffffa8006a4bb10 [ALPC_CLIENT_COMMUNICATION_PORT]
  WaitingThread      : ffffffa8009db7160
  QueueType          : ALPC_MSGQUEUE_PENDING
  QueuePort          : ffffffa800577ee60 [ALPC_CONNECTION_PORT]
  QueuePortOwnerProcess: ffffffa80056ddb30 (ServiceB.exe)
  ServerThread        : ffffffa8007ead4d0
  QuotaCharged        : No
  CancelQueuePort    : 0000000000000000
  CancelSequencePort : 0000000000000000
  CancelSequenceNumber: 0x00000000 (0)
  ClientContext       : 0000000002a60f40
  ServerContext       : 0000000000000000
  PortContext         : 000000000227a370
  CancelPortContext  : 0000000000000000
  SecurityData        : 0000000000000000
  View                : 0000000000000000
```

```
0: kd> !alpc /p fffffa800577ee60
Port @ fffffa800577ee60
  Type                 : ALPC_CONNECTION_PORT
  CommunicationInfo   : fffff8a0022435d0
  ConnectionPort       : ffffffa800577ee60
  ClientCommunicationPort: 0000000000000000
  ServerCommunicationPort: 0000000000000000
  OwnerProcess         : ffffffa80056ddb30 (ServiceB.exe)
  SequenceNo           : 0x0000481A (18458)
  CompletionPort       : ffffffa8005728e80
  CompletionList        : 0000000000000000
  MessageZone          : 0000000000000000
  ConnectionPending     : No
  ConnectionRefused    : No
  Disconnected          : No
  Closed                : No
  FlushOnClose          : Yes
  ReturnExtendedInfo   : No
  Waitable              : No
  Security              : Static
  Wow64CompletionList  : No
Main queue is empty.
```

Large message queue is empty.

Pending queue has 698 message(s)

```

fffff8a00235aa0 00000404 000000000001344:000000000001358 0000000000000000 fffffa8004c0cb60 LPC_REQUEST
fffff8a00a52f030 00000644 000000000001078:0000000000024c0 0000000000000000 fffffa80072f1b60 LPC_REQUEST
fffff8a00abb5030 000007a8 00000000000103c:00000000000050c 0000000000000000 fffffa800725b580 LPC_REQUEST
fffff8a00239cab0 000000b8 000000000000480:0000000000015f8 0000000000000000 fffffa80077f0b60 LPC_REQUEST
fffff8a00ac81a90 00000a18 0000000000028ac:000000000001e54 0000000000000000 fffffa8007fba060 LPC_CANCELED
fffff8a005879140 00000f80 000000000001260:000000000000730 fffffa8006432060 fffffa8006b18060 LPC_REQUEST
fffff8a013720d00 00000c6c 000000000003764:0000000000032a8 0000000000000000 fffffa8006b00a60 LPC_CANCELED
fffff8a00ac82660 00000810 000000000003af4:000000000002a98 0000000000000000 fffffa80068c0b60 LPC_CANCELED
fffff8a00bdec450 00000ec8 000000000000233c:0000000000013f8 0000000000000000 fffffa80079455b0 LPC_CANCELED
fffff8a00b662830 000005cc 00000000000005e4:000000000000e0c fffffa800791a7a0 fffffa8007376580 LPC_REQUEST
fffff8a003d57150 00000f08 000000000002678:000000000003e0c 0000000000000000 fffffa8007e4a870 LPC_CANCELED
fffff8a000cd08830 00000750 000000000003408:000000000003adc 0000000000000000 fffffa8008631b60 LPC_CANCELED
fffff8a01855b2f0 000004f4 000000000002c74:000000000002d00 0000000000000000 fffffa800746b890 LPC_CANCELED
fffff8a00da0db0 00000db0 000000000001a34:000000000002d80 0000000000000000 fffffa80080aff4b60 LPC_CANCELED
fffff8a00edd0b30 0000059c 000000000003f34:000000000003c8c 0000000000000000 fffffa8008f96060 LPC_CANCELED
fffff8a017a14d00 00000920 000000000003850:000000000002588 0000000000000000 fffffa8009f66060 LPC_CANCELED
fffff8a01792d030 000007f8 000000000003844:0000000000028d0 0000000000000000 fffffa800ad56260 LPC_CANCELED
fffff8a00f8d6ae0 00000f30 00000000000239c:000000000001694 0000000000000000 fffffa8008b86060 LPC_CANCELED
fffff8a01395ab80 00000cdc 000000000003630:0000000000018f8 0000000000000000 fffffa8005bc0770 LPC_CANCELED
fffff8a0166ff800 00000984 00000000000005e4:0000000000025f4 fffffa8009718910 fffffa8008cbfb60 LPC_REQUEST
fffff8a012b9f5a0 00000ac8 000000000002d34:000000000001b24 0000000000000000 fffffa8009cd8410 LPC_CANCELED
fffff8a014313830 00000afc 00000000000005e4:0000000000023bc fffffa80073f0230 fffffa80054d7060 LPC_REQUEST
fffff8a00a34a6b0 00000ca8 000000000002534:000000000002dd0 0000000000000000 fffffa80064c3980 LPC_CANCELED
[...]
fffff8a00ad8f610 00000e64 000000000003714:0000000000030b8 0000000000000000 fffffa800aeea9f0 LPC_REQUEST
fffff8a015720710 00001594 000000000003638:0000000000029b8 0000000000000000 fffffa800b5359a0 LPC_REQUEST
fffff8a009bac560 00001508 000000000003994:000000000001aac 0000000000000000 fffffa800b5359a0 LPC_REQUEST
fffff8a00b6e78f0 00001574 000000000002938:000000000001998 0000000000000000 fffffa800aea9f0 LPC_REQUEST
fffff8a00b5716b0 00001570 000000000002938:000000000001698 0000000000000000 fffffa800a3b8620 LPC_REQUEST
fffff8a018531d00 00000db8 0000000000016d8:0000000000031c4 0000000000000000 fffffa800b5359a0 LPC_REQUEST
fffff8a01112f410 000014b0 0000000000001b6c:0000000000001618 0000000000000000 fffffa800a3b8620 LPC_CANCELED

```

Canceled queue is empty.

## Comments

**Q.** Does this mean that no thread in *ServiceB.exe* can accept the ALPC request?

**A.** We think so because we haven't yet seen the opposite: there would be no need to save a memory dump file if ALPC works.

## Blocked Thread

### Hardware

This is a specialization of **Blocked Thread** pattern where a thread is waiting for hardware I/O response. For example, a frozen system initialization thread is waiting for a response from one of ACPI general register ports:

```
kd> kL 100
ChildEBP RetAddr
f7a010bc f74c5a57 hal!READ_PORT_UCHAR+0x7
f7a010c8 f74c5ba4 ACPI!DefReadAcpiRegister+0xa1
f7a010d8 f74b4d78 ACPI!ACPIReadGpeStatusRegister+0x10
f7a010e4 f74b6334 ACPI!ACPIGpeIsEvent+0x14
f7a01100 8054157d ACPI!ACPIInterruptServiceRoutine+0x16
f7a01100 806d687d nt!KiInterruptDispatch+0x3d
f7a01194 804f9487 hal!HalEnableSystemInterrupt+0x79
f7a011d8 8056aac4 nt!KeConnectInterrupt+0x95
f7a011fc f74c987c nt!IoConnectInterrupt+0xf2
f7a0123c f74d13f0 ACPI!OSInterruptVector+0x76
f7a01250 f74b5781 ACPI!ACPIInitialize+0x154
f7a01284 f74cf824 ACPI!ACPIInitStartACPI+0x71
f7a012b0 f74b1e12 ACPI!ACPIRootIrpStartDevice+0xc0
f7a012e0 804ee129 ACPI!ACPIDispatchIrp+0x15a
f7a012f0 8058803b nt!IopfCallDriver+0x31
f7a0131c 805880b9 nt!IopSynchronousCall+0xb7
f7a01360 804f515c nt!IopStartDevice+0x4d
f7a0137c 80587769 nt!PipProcessStartPhase1+0x4e
f7a015d4 804f5823 nt!PipProcessDevNodeTree+0x1db
f7a01618 804f5ab3 nt!PipDeviceActionWorker+0xa3
f7a01630 8068afc6 nt!PipRequestDeviceAction+0x107
f7a01694 80687e48 nt!IopInitializeBootDrivers+0x376
f7a0183c 806862dd nt!IoInitSystem+0x712
f7a01dac 805c61e0 nt!Phase1Initialization+0x9b5
f7a01ddc 80541e02 nt!PspSystemThreadStartup+0x34
00000000 00000000 nt!KiThreadStartup+0x16

kd> r
eax=00000000 ebx=00000000 ecx=00000002 edx=0000100c esi=00000000 edi=867d8008
eip=806d664b esp=f7a010c0 ebp=f7a010c8 iopl=1 nv up ei pl zr na pe nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00001246
hal!READ_PORT_UCHAR+0x7:
806d664b c20400 ret 4
```

```
kd> ub eip
hal!KdRestore+0x9:
806d663f cc          int     3
806d6640 cc          int     3
806d6641 cc          int     3
806d6642 cc          int     3
806d6643 cc          int     3
hal!READ_PORT_UCHAR:
806d6644 33c0         xor    eax,eax
806d6646 8b542404     mov    edx,dword ptr [esp+4]
806d664a ec          in     al,dx

kd> version
[...]
System Uptime: 0 days 0:03:42.140
[...]

kd> !thread
THREAD 867c63e8 Cid 0004.0008 Teb: 00000000 Win32Thread: 00000000 RUNNING on processor 0
IRP List:
 867df008: (0006,0190) Flags: 00000000 Mdl: 00000000
Not impersonating
DeviceMap           e1005460
Owning Process      0           Image: <Unknown>
Attached Process    867c6660   Image: System
Wait Start TickCount 39          Ticks: 1839 (0:00:00:18.416)
Context Switch Count 4
UserTime             00:00:00.000
KernelTime           00:00:00.911
Start Address nt!Phase1Initialization (0x80685928)
Stack Init f7a02000 Current f7a014a4 Base f7a02000 Limit f79ff000 Call 0
Priority 31 BasePriority 8 PriorityDecrement 0 DecrementCount 0
[...]
```

## Software

We often say that particular thread is blocked and/or it blocks other threads. At the same time, we know that almost all threads are “blocked” to some degree except those currently running on processors. They are either preempted and in the ready lists, voluntarily yielded their execution, or they are waiting for some synchronization object. Therefore the notion of **Blocked Thread** is highly context and problem dependent and usually we notice them when comparing current thread stack traces with their expected normal stack traces. Here reference guides (Appendix A) are indispensable especially those created for troubleshooting concrete products.

To show the diversity of “blocked” threads, we can propose the following thread classification:

### **Running threads**

Their EIP (RIP) points to some function different from *KiSwapContext*:

```
3: kd> !running

System Processors f (affinity mask)
Idle Processors 0

    Prcb      Current   Next
0  ffdff120  a30a9350      .....
1  f7727120  a3186448      .....
2  f772f120  a59a1b40      .....
3  f7737120  a3085888      .....

3: kd> !thread a59a1b40
THREAD a59a1b40  Cid 0004.00b8  Teb: 00000000 Win32Thread: 00000000 RUNNING on processor 2
Not impersonating
DeviceMap          e10028b0
Owning Process     a59aa648  Image:       System
Wait Start TickCount 1450446  Ticks: 1 (0:00:00:00.015)
Context Switch Count 308765
UserTime           00:00:00.000
KernelTime         00:00:01.250
Start Address nt!ExpWorkerThread (0x80880356)
Stack Init f7055000 Current f7054cec Base f7055000 Limit f7052000 Call 0
Priority 12 BasePriority 12 PriorityDecrement 0
ChildEBP RetAddr
f7054bc4 8093c55c nt!ObfReferenceObject+0x1c
f7054ca0 8093d2ae nt!ObpQueryNameString+0x2ba
f7054cbc 808f7d0f nt!ObQueryNameString+0x18
f7054d80 80880441 nt!IopErrorLogThread+0x197
f7054dac 80949b7c nt!ExpWorkerThread+0xeb
f7054ddc 8088e062 nt!PspSystemThreadStartup+0x2e
00000000 00000000 nt!KiThreadStartup+0x16

3: kd> .thread a59a1b40
Implicit thread is now a59a1b40
```

```
3: kd> r
Last set context:
eax=00000028 ebx=e1002b28 edx=e1000234 esi=e1002b18 edi=0000001a
eip=8086c73e esp=f7054bc4 ebp=f7054ca0 iopl=0 nv up ei pl nz na po nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00000202
nt!ObfReferenceObject+0x1c:
8086c73e 40          inc      eax
```

These threads can also be identified by RUNNING attribute in the output of **!process 0 3f** command applied for complete and kernel memory dumps.

### **Ready threads**

These threads can be seen in the output of **!ready** command or identified by READY attribute in the output of **!process 0 3f** command:

```
3: kd> !ready
Processor 0: No threads in READY state
Processor 1: Ready Threads at priority 11
    THREAD a3790790 Cid 0234.1108 Teb: 7ffab000 Win32Thread: 00000000 READY
    THREAD a32799a8 Cid 0234.061c Teb: 7ff83000 Win32Thread: 00000000 READY
    THREAD a3961798 Cid 0c04.0c68 Teb: 7ffab000 Win32Thread: bc204ea8 READY
Processor 1: Ready Threads at priority 10
    THREAD a32bedb0 Cid 1fc8.1a30 Teb: 7ffad000 Win32Thread: bc804468 READY
Processor 1: Ready Threads at priority 9
    THREAD a52dc4d8 Cid 0004.04d4 Teb: 00000000 Win32Thread: 00000000 READY
Processor 2: Ready Threads at priority 11
    THREAD a37fedb0 Cid 0c04.11f8 Teb: 7ff8e000 Win32Thread: 00000000 READY
Processor 3: Ready Threads at priority 11
    THREAD a5683db0 Cid 0234.0274 Teb: 7ffd6000 Win32Thread: 00000000 READY
    THREAD a3151b48 Cid 0234.2088 Teb: 7ff88000 Win32Thread: 00000000 READY
    THREAD a5099d80 Cid 0ecc.0d60 Teb: 7ffd4000 Win32Thread: 00000000 READY
    THREAD a3039498 Cid 0c04.275c Teb: 7ff7d000 Win32Thread: 00000000 READY
```

If we look at these threads we see that they were either scheduled to run because of a signaled object they were waiting for:

```
3: kd> !thread a3039498
THREAD a3039498 Cid 0c04.275c Teb: 7ff7d000 Win32Thread: 00000000 READY
IRP List:
    a2feb008: (0006,0094) Flags: 00000870 Mdl: 00000000
Not impersonating
DeviceMap           e10028b0
Owning Process     a399a770      Image:        svchost.exe
Wait Start TickCount 1450447      Ticks: 0
Context Switch Count 1069
UserTime            00:00:00.000
KernelTime          00:00:00.000
Win32 Start Address 0x001e4f22
LPC Server thread working on message Id 1e4f22
Start Address KERNEL32!BaseThreadStartThunk (0x77e617ec)
Stack Init f171b000 Current f171ac60 Base f171b000 Limit f1718000 Call 0
Priority 11 BasePriority 10 PriorityDecrement 0
```

```
ChildEBP RetAddr  Args to Child
f171ac78 80833465 a3039498 a3039540 e7561930 nt!KiSwapContext+0x26
f171aca4 80829a62 00000000 00000000 00000000 nt!KiSwapThread+0x2e5
f171acec 80938d0c a301cad8 00000006 f171ad01 nt!KeWaitForSingleObject+0x346
f171ad50 8088978c 00000c99 00000000 00000000 nt!NtWaitForSingleObject+0x9a
f171ad50 7c8285ec 00000c99 00000000 00000000 nt!KiFastCallEntry+0xfc
03d9efea8 00000000 00000000 00000000 00000000 ntdll!KiFastSystemCallRet
```

```
3: kd> !object a301cad8
Object: a301cad8  Type: (a59a0720) Event
    ObjectHeader: a301cac0 (old version)
    HandleCount: 1  PointerCount: 3
```

or they were boosted in priority:

```
3: kd> !thread a3790790
THREAD a3790790  Cid 0234.1108  Teb: 7ffab000 Win32Thread: 00000000 READY
IRP List:
    a2f8b7f8: (0006,0094) Flags: 00000900  Mdl: 00000000
Not impersonating
DeviceMap          e10028b0
Owning Process     a554bcc8      Image:        lsass.exe
Wait Start Ticks   1450447       Ticks: 0
Context Switch Count 384
UserTime           00:00:00.000
KernelTime         00:00:00.000
Win32 Start Address RPCRT4!ThreadStartRoutine (0x77c7b0f5)
Start Address KERNEL32!BaseThreadStartThunk (0x77e617ec)
Stack Init f3ac1000 Current f3ac0ce8 Base f3ac1000 Limit f3abe000 Call 0
Priority 11 BasePriority 10 PriorityDecrement 0
ChildEBP RetAddr  Args to Child
f3ac0d00 80831266 a3790790 a50f1870 a3186448 nt!KiSwapContext+0x26
f3ac0d20 8082833a 00000000 a50f1870 8098b56c nt!KiExitDispatcher+0xf8
f3ac0d3c 8098b5b9 a50f1870 00000000 00f5f8d0 nt!KeSetEventBoostPriority+0x156
f3ac0d58 8088978c a50f1870 00f5f8d4 7c8285ec nt!NtSetEventBoostPriority+0x4d
f3ac0d58 7c8285ec a50f1870 00f5f8d4 7c8285ec nt!KiFastCallEntry+0xfc
00f5f8d4 00000000 00000000 00000000 00000000 ntdll!KiFastSystemCallRet

3: kd> !object a50f1870
Object: a50f1870  Type: (a59a0720) Event
    ObjectHeader: a50f1858 (old version)
    HandleCount: 1  PointerCount: 15
```

or were interrupted and queued to be run again:

```

3: kd> !thread a5683db0
THREAD a5683db0 Cid 0234.0274 Teb: 7ffd6000 Win32Thread: 00000000 READY
IRP List:
 a324d498: (0006,0094) Flags: 00000900 Mdl: 00000000
 a2f97a20: (0006,0094) Flags: 00000900 Mdl: 00000000
 a50c3e70: (0006,0190) Flags: 00000000 Mdl: a50a22d0
 a5167750: (0006,0094) Flags: 00000800 Mdl: 00000000
Not impersonating
DeviceMap e10028b0
Owning Process a554bcc8 Image: lsass.exe
Wait Start TickCount 1450447 Ticks: 0
Context Switch Count 9619
UserTime 00:00:00.156
KernelTime 00:00:00.234
Win32 Start Address RPCRT4!ThreadStartRoutine (0x77c7b0f5)
Start Address KERNEL32!BaseThreadStartThunk (0x77e617ec)
Stack Init f59f3000 Current f59f2d00 Base f59f3000 Limit f59f0000 Call 0
Priority 11 BasePriority 10 PriorityDecrement 0
ChildEBP RetAddr
f59f2d18 80a5c1ae nt!KiDispatchInterrupt+0xb1
f59f2d48 80a5c577 hal!HalpDispatchSoftwareInterrupt+0x5e
f59f2d54 80a59902 hal!HalEndSystemInterrupt+0x67
f59f2d54 77c6928d hal!HalpIpiHandler+0xd2 (TrapFrame @ f59f2d64)
00c5f908 00000000 RPCRT4!OSF_SCALL::GetBuffer+0x37

3: kd> .thread a5683db0
Implicit thread is now a5683db0

3: kd> r
Last set context:
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
eip=8088dba1 esp=f59f2d0c ebp=f59f2d2c iopl=0 nv up di pl nz na po nc
cs=0008 ss=0010 ds=0000 es=0000 fs=0000 gs=0000 ef1=00000000
nt!KiDispatchInterrupt+0xb1:
8088dba1 b902000000 mov ecx,2

3: kd> ub
nt!KiDispatchInterrupt+0x8f:
8088db7f mov dword ptr [ebx+124h],esi
8088db85 mov byte ptr [esi+4Ch],2
8088db89 mov byte ptr [edi+5Ah],1Fh
8088db8d mov ecx,edi
8088db8f lea edx,[ebx+120h]
8088db95 call nt!KiQueueReadyThread (80833490)
8088db9a mov cl,1
8088db9c call nt!SwapContext (8088dbd0)

3: kd> u
nt!KiDispatchInterrupt+0xb1:
8088dba1 mov ecx,2
8088dba6 call dword ptr [nt!_imp_KfLowerIrql (80801108)]
8088dbac mov ebp,dword ptr [esp]
8088dbaf mov edi,dword ptr [esp+4]
8088dbb3 mov esi,dword ptr [esp+8]
```

```
8088dbb7 add esp,0Ch
8088dbba pop ebx
8088dbbb ret
```

We can get user space thread stack by using `.trap` WinDbg command, but we need to switch to the corresponding process context first:

```
3: kd> .process /r /p a554bcc8
Implicit process is now a554bcc8
Loading User Symbols

3: kd> kL
*** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr
00c5f908 77c7ed60 RPCRT4!OSF_SCALL::GetBuffer+0x37
00c5f924 77c7ed14 RPCRT4!I_RpcGetBufferWithObject+0x7f
00c5f934 77c7f464 RPCRT4!I_RpcGetBuffer+0xf
00c5f944 77ce3470 RPCRT4!NdrGetBuffer+0xe
00c5fd44 77ce35c4 RPCRT4!NdrStubCall2+0x35c
00c5fd60 77c7ff7a RPCRT4!NdrServerCall2+0x19
00c5fd94 77c8042d RPCRT4!DispatchToStubInCNoAvrf+0x38
00c5fde8 77c80353 RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x11f
00c5fe0c 77c68e0d RPCRT4!RPC_INTERFACE::DispatchToStub+0xa3
00c5fe40 77c68cb3 RPCRT4!OSF_SCALL::DispatchHelper+0x149
00c5fe54 77c68c2b RPCRT4!OSF_SCALL::DispatchRPCCall+0x10d
00c5fe84 77c68b5e RPCRT4!OSF_SCALL::ProcessReceivedPDU+0x57f
00c5fea4 77c6e8db RPCRT4!OSF_SCALL::BeginRpcCall+0x194
00c5ff04 77c6e7b4 RPCRT4!OSF_SCONNECTION::ProcessReceiveComplete+0x435
00c5ff18 77c7b799 RPCRT4!ProcessConnectionServerReceivedEvent+0x21
00c5ff84 77c7b9b5 RPCRT4!LOADABLE_TRANSPORT::ProcessIOEvents+0x1b8
00c5ff8c 77c8872d RPCRT4!ProcessIOEventsWrapper+0xd
00c5ffac 77c7b110 RPCRT4!BaseCachedThreadRoutine+0x9d
00c5ffb8 77e64829 RPCRT4!ThreadStartRoutine+0x1b
00c5ffec 00000000 kernel32!BaseThreadStart+0x34
```

### ***Waiting threads (wait originated from user space)***

```
THREAD a34369d0 Cid 1fc8.1e88 Teb: 7ffae000 Win32Thread: bc6d5818 WAIT: (Unknown) UserMode Non-
Alertable
    a34d9940 SynchronizationEvent
    a3436a48 NotificationTimer
Not impersonating
DeviceMap          e12256a0
Owning Process    a3340a10      Image:           IEXPLORE.EXE
Wait Start TickCount 1450409      Ticks: 38 (0:00:00:00.593)
Context Switch Count 7091        LargeStack
UserTime           00:00:01.015
KernelTime         00:00:02.250
Win32 Start Address mshtml!CExecFT::StaticThreadProc (0x7fab1061)
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init f252b000 Current f252ac60 Base f252b000 Limit f2528000 Call 0
Priority 11 BasePriority 10 PriorityDecrement 0
ChildEBP RetAddr
f252ac78 80833465 nt!KiSwapContext+0x26
f252aca4 80829a62 nt!KiSwapThread+0x2e5
```

```
f252acec 80938d0c nt!KeWaitForSingleObject+0x346
f252ad50 8088978c nt!NtWaitForSingleObject+0x9a
f252ad50 7c8285ec nt!KiFastCallEntry+0xfc (TrapFrame @ f252ad64)
030dff08 7c827d0b ntdll!KiFastSystemCallRet
030dff0c 77e61d1e ntdll!NtWaitForSingleObject+0xc
030dff7c 77e61c8d kernel32!WaitForSingleObjectEx+0xac
030dff90 7fab08a3 kernel32!WaitForSingleObject+0x12
030dfffa8 7fab109c mshtml!CDwnTaskExec::ThreadExec+0xae
030dffb0 7fab106e mshtml!CExecFT::ThreadProc+0x28
030dffb8 77e64829 mshtml!CExecFT::StaticThreadProc+0xd
030dffec 00000000 kernel32!BaseThreadStart+0x34
```

If we had taken user dump of iexplore.exe we would have seen the following stack trace there:

```
030dff08 7c827d0b ntdll!KiFastSystemCallRet
030dff0c 77e61d1e ntdll!NtWaitForSingleObject+0xc
030dff7c 77e61c8d kernel32!WaitForSingleObjectEx+0xac
030dff90 7fab08a3 kernel32!WaitForSingleObject+0x12
030dfffa8 7fab109c mshtml!CDwnTaskExec::ThreadExec+0xae
030dffb0 7fab106e mshtml!CExecFT::ThreadProc+0x28
030dffb8 77e64829 mshtml!CExecFT::StaticThreadProc+0xd
030dffec 00000000 kernel32!BaseThreadStart+0x34
```

Another example:

```
THREAD a31f2438 Cid 1fc8.181c Teb: 7ffaa000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Non-Alertable
    a30f8c20 NotificationEvent
    a5146720 NotificationEvent
    a376fb0 NotificationEvent
Not impersonating
DeviceMap          e12256a0
Owning Process     a3340a10      Image:      IEXPLORE.EXE
Wait Start TickCount 1419690      Ticks: 30757 (0:00:08:00.578)
Context Switch Count 2
UserTime           00:00:00.000
KernelTime         00:00:00.000
Win32 Start Address USERENV!NotificationThread (0x76929dd9)
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init f5538000 Current f5537900 Base f5538000 Limit f5535000 Call 0
Priority 10 BasePriority 10 PriorityDecrement 0
Kernel stack not resident.
ChildEBP RetAddr
f5537918 80833465 nt!KiSwapContext+0x26
f5537944 80829499 nt!KiSwapThread+0x2e5
f5537978 80938f68 nt!KeWaitForMultipleObjects+0x3d7
f5537bf4 809390ca nt!ObpWaitForMultipleObjects+0x202
f5537d48 8088978c nt!NtWaitForMultipleObjects+0xc8
f5537d48 7c8285ec nt!KiFastCallEntry+0xfc (TrapFrame @ f5537d64)
0851fec0 7c827cfb ntdll!KiFastSystemCallRet
0851fec4 77e6202c ntdll!NtWaitForMultipleObjects+0xc
0851ff6c 77e62fbe kernel32!WaitForMultipleObjectsEx+0x11a
0851ff88 76929e35 kernel32!WaitForMultipleObjects+0x18
0851ffb8 77e64829 USERENV!NotificationThread+0x5f
0851ffec 00000000 kernel32!BaseThreadStart+0x34
```

***Waiting threads (wait originated from kernel space)***

Examples include explicit wait as a result from calling potentially blocking API:

```
THREAD a33a9740 Cid 1980.1960 Peb: 7ffdde000 Win32Thread: bc283ea8 WAIT: (Unknown) UserMode Non-Alertable
    a35e3168 SynchronizationEvent
Not impersonating
DeviceMap          e689f298
Owning Process     a342d3a0      Image:           explorer.exe
Wait Start TickCount 1369801      Ticks: 80646 (0:00:21:00.093)
Context Switch Count 1667          LargeStack
UserTime            00:00:00.015
KernelTime          00:00:00.093
Win32 Start Address Explorer!ModuleEntry (0x010148a4)
Start Address kernel32!BaseProcessStartThunk (0x77e617f8)
Stack Init f258b000 Current f258ac50 Base f258b000 Limit f2585000 Call 0
Priority 13 BasePriority 10 PriorityDecrement 1
Kernel stack not resident.
ChildEBP RetAddr
f258ac68 80833465 nt!KiSwapContext+0x26
f258ac94 80829a62 nt!KiSwapThread+0x2e5
f258acdc bf89abd3 nt!KeWaitForSingleObject+0x346
f258ad38 bf89da43 win32k!xxxSleepThread+0x1be
f258ad4c bf89e401 win32k!xxxRealWaitMessageEx+0x12
f258ad5c 8088978c win32k!NtUserWaitMessage+0x14
f258ad5c 7c8285ec nt!KiFastCallEntry+0xfc (TrapFrame @ f258ad64)
0007feec 7739bf53 ntdll!KiFastSystemCallRet
0007ff08 7c8fadbd USER32!NtUserWaitMessage+0xc
0007ff14 0100ffff SHELL32!SHDesktopMessageLoop+0x24
0007ff5c 0101490c Explorer!ExplorerWinMain+0x2c4
0007ffc0 77e6f23b Explorer!ModuleEntry+0x6d
0007fff0 00000000 kernel32!BaseProcessStart+0x23
```

and implicit wait when a thread yields execution to another thread voluntarily via explicit context swap:

```
THREAD a3072b68 Cid 1fc8.1d94 Peb: 7ffaf000 Win32Thread: bc1e3c20 WAIT: (Unknown) UserMode Non-Alertable
    a37004d8 QueueObject
    a3072be0 NotificationTimer
IRP List:
    a322be0: (0006,01fc) Flags: 00000000 Mdl: a30b8e30
    a30bcc38: (0006,01fc) Flags: 00000000 Mdl: a35bf530
Not impersonating
DeviceMap          e12256a0
Owning Process     a3340a10      Image:           IEXPLORE.EXE
Wait Start TickCount 1447963      Ticks: 2484 (0:00:00:38.812)
Context Switch Count 3972          LargeStack
UserTime            00:00:00.140
KernelTime          00:00:00.250
Win32 Start Address ntdll!RtlpWorkerThread (0x7c839efb)
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init f1cc3000 Current f1cc2c38 Base f1cc3000 Limit f1cbf000 Call 0
Priority 10 BasePriority 10 PriorityDecrement 0
ChildEBP RetAddr
```

```

f1cc2c50 80833465 nt!KiSwapContext+0x26
f1cc2c7c 8082b60f nt!KiSwapThread+0x2e5
f1cc2cc4 808ed620 nt!KeRemoveQueue+0x417
f1cc2d48 8088978c nt!NtRemoveIoCompletion+0xdc
f1cc2d48 7c8285ec nt!KiFastCallEntry+0xfc (TrapFrame @ f1cc2d64)
06ceff70 7c8277db ntdll!KiFastSystemCallRet
06ceff74 7c839f38 ntdll!ZwRemoveIoCompletion+0xc
06ceffb8 77e64829 ntdll!RtlpWorkerThread+0x3d
06ceffec 00000000 kernel32!BaseThreadStart+0x34

THREAD a3612020 Cid 1980.1a48 Teb: 7ffd9000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Alertable
    a3612098 NotificationTimer
Not impersonating
DeviceMap          e689f298
Owning Process     a342d3a0      Image:           explorer.exe
Wait Start TickCount 1346718      Ticks: 103729 (0:00:27:00.765)
Context Switch Count 4
UserTime            00:00:00.000
KernelTime          00:00:00.000
Win32 Start Address ntdll!RtlpTimerThread (0x7c83d3dd)
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init f2453000 Current f2452c80 Base f2453000 Limit f2450000 Call 0
Priority 10 BasePriority 10 PriorityDecrement 0
Kernel stack not resident.
ChildEBP RetAddr
f2452c98 80833465 nt!KiSwapContext+0x26
f2452cc4 80828f0b nt!KiSwapThread+0x2e5
f2452d0c 80994812 nt!KeDelayExecutionThread+0x2ab
f2452d54 8088978c nt!NtDelayExecution+0x84
f2452d54 7c8285ec nt!KiFastCallEntry+0xfc (TrapFrame @ f2452d64)
0149ff9c 7c826f4b ntdll!KiFastSystemCallRet
0149ffa0 7c83d424 ntdll!NtDelayExecution+0xc
0149ffb8 77e64829 ntdll!RtlpTimerThread+0x47
0149ffec 00000000 kernel32!BaseThreadStart+0x34

```

Explicit waits in the kernel can be originated from GUI threads and their message loops, for example, **Main Thread** (page 614). Blocked GUI thread, **Message Box** pattern (page 660) can be seen as an example of a genuine **Blocked Thread**. Some “blocked” threads are just really **Passive Threads** (page 793).

## Comments

An example of a blocked thread that is trying to load a library and blocked in the kernel:

```
ntdll!KiFastSystemCallRet
ntdll!NtQueryAttributesFile+0xc)
ntdll!RtlDoesFileExists_UstrEx+0x6b
ntdll!RtlDoesFileExists_UEx+0x27
ntdll!RtlDosSearchPath_U+0x14f
ntdll!LdrpResolveD11Name+0x12d
ntdll!LdrpMapD11+0x140
ntdll!LdrpLoadD11+0x1e9
ntdll!LdrLoadD11+0x230
kernel32!LoadLibraryExW+0x18e
kernel32!LoadLibraryExA+0x1f
[...]
WINMM!DrvSendMessage+0x18
MSACM32!IDriverMessageId+0x81
MSACM32!acmFormatSuggest+0x28b
mmdriver!DriverProc+0x8e52
[...]
mmdriver!wodMessage+0x76
WINMM!waveOutOpen+0x2a2
[...]
```

Another example of the blocked thread on a uniprocessor VM:

```
kd> !running

System Processors: (00000001)
Idle Processors: (00000000)

Prcbs Current Next
0 82944d20 85a575c0 ......

kd> !thread
THREAD 85a575c0 Cid 0004.00e0 Teb: 00000000 Win32Thread: 00000000 RUNNING on processor 0
Not impersonating
DeviceMap 8a209d68
Owning Process 8523e660 Image: System
Attached Process N/A Image: N/A
Wait Start TickCount 4392621 Ticks: 0
Context Switch Count 414195
UserTime 00:00:00.000
KernelTime 17:59:14.739
Win32 Start Address DriverA (0x8900ab4e)
Stack Init 8d1c0fd0 Current 8d1c0950 Base 8d1c1000 Limit 8d1be000 Call 0
Priority 16 BasePriority 8 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
ChildEBP RetAddr Args to Child
8d1c0c48 82c34117 ffd090f0 85a32008 85a5a000 nt!READ_REGISTER ULONG+0x6 (FPO: [1,0,0])
8d1c0c68 82c347a1 8d1c0c84 82c38b53 8d1c0c7c hal!HalpHpetQueryCount+0x4b (FPO: [Non-Fpo])
8d1c0c70 82c38b53 8d1c0c7c 00da7a64 00000000 hal!HalpHpetQueryPerformanceCounter+0x1d (FPO: [Non-Fpo])
8d1c0c84 89007e52 8d1c0cbc 85a5a0d0 85a5a000 hal!KeQueryPerformanceCounter+0x3d (FPO: [Non-Fpo])
```

```
WARNING: Stack unwind information not available. Following frames may be wrong.  
8d1c0ce4 89007938 85a5a000 00000000 85a5a000 DriverA+0x7e52  
8d1c0cf8 8900ae9a 85a5a000 00000000 00000000 DriverA+0x7938  
8d1c0d50 82a23056 85a32008 a9492f18 00000000 DriverA+0xae9a  
8d1c0d90 828cb1a9 8900ab4e 85a5a000 00000000 nt!PspSystemThreadStartup+0x9e  
00000000 00000000 00000000 00000000 nt!KiThreadStartup+0x19
```

```
kd> !ready  
Processor 0: Ready Threads at priority 15  
THREAD 861a2590 Cid 0460.0464 Teb: 7ffdf000 Win32Thread: fe9dbb18 ????  
THREAD 853cac98 Cid 0460.0670 Teb: 7ffd6000 Win32Thread: 00000000 ????  
THREAD 852fd488 Cid 0004.0084 Teb: 00000000 Win32Thread: 00000000 ????  
THREAD 8534ad48 Cid 0004.0090 Teb: 00000000 Win32Thread: 00000000 ????  
THREAD 852f1d48 Cid 0004.0080 Teb: 00000000 Win32Thread: 00000000 ????  
THREAD 852fd4f0 Cid 0004.0088 Teb: 00000000 Win32Thread: 00000000 ????  
THREAD 85340020 Cid 0004.008c Teb: 00000000 Win32Thread: 00000000 ????  
THREAD 852e67c8 Cid 0004.0078 Teb: 00000000 Win32Thread: 00000000 ????  
THREAD 85de77a8 Cid 01e4.026c Teb: 7ffd0000 Win32Thread: ffb2c008 ????  
THREAD 85da7d48 Cid 01e4.0234 Teb: 7ffdf000 Win32Thread: ffad0d38 ????  
THREAD 85a7c678 Cid 0004.00f0 Teb: 00000000 Win32Thread: 00000000 ????  
[...]
```

## Timeout

This is a special variant of **Blocked Thread** pattern where we have a timeout value: a thread is temporarily blocked. For example, this **Main Thread** (page 614) is blocked while waiting for the beep sound to finish after a minute:

```
0:000> kvL
ChildEBP RetAddr Args to Child
0291f354 7c90d21a 7c8023f1 00000001 0291f388 ntdll!KiFastSystemCallRet
0291f358 7c8023f1 00000001 0291f388 7c90d27e ntdll!NtDelayExecution+0xc
0291f3b0 7c837beb 0000ea60 00000001 00000004 kernel32!SleepEx+0x61
0291f404 004952a2 00000370 0000ea60 004d6ae2 kernel32!Beep+0x1b3
0291f410 004d6ae2 00000370 0000ea60 004d6ed4 Application!DoBeep+0x16
[...]
0291ffec 00000000 0045aad0 00e470a0 00000000 kernel32!BaseThreadStart+0x37

0:000> ? ea60/0n1000
Evaluate expression: 60 = 0000003c
```

## Blocking File

This pattern often happens (but not limited to) in roaming profile scenarios In addition to **Blocked Thread** (page 82), endpoint threads of **Wait Chain** patterns (page 1092), and **Blocking Module** (page 96). For example, an application was reported hanging, and in a complete memory dump we could see a thread in **Stack Trace Collection** (page 943):

```
THREAD ffffffa8005eca060 Cid 14b0.1fec Peb: 000000007ef84000 Win32Thread: ffffff900c26c2c30 WAIT:
(Executive) KernelMode Non-Alertable
ffffffa80048e6758 NotificationEvent
IRP List:
ffffffa8005a6c160: (0006,03e8) Flags: 00060000 Mdl: 00000000
Not impersonating
DeviceMap fffff8a0055b6620
Owning Process ffffffa80063dd970 Image: Application.exe
Attached Process N/A Image: N/A
Wait Start TickCount 171988390 Ticks: 26963639 (4:21:01:46.859)
Context Switch Count 226 LargeStack
UserTime 00:00:00.015
KernelTime 00:00:00.015
Win32 Start Address 0x000000006d851f62
Stack Init fffff880075a9db0 Current fffff880075a9770
Base fffff880075aa000 Limit fffff880075a4000 Call 0
Priority 10 BasePriority 8 UnusualBoost 0 ForegroundBoost 2 IoPriority 2 PagePriority 5
Child-SP RetAddr Call Site
ffffff880`075a97b0 fffff800`0167f752 nt!KiSwapContext+0x7a
ffffff880`075a98f0 fffff800`016818af nt!KiCommitThreadWait+0x1d2
ffffff880`075a9980 fffff800`019b612a nt!KeWaitForSingleObject+0x19f
ffffff880`075a9a20 fffff800`0198feaa nt! ?? ::NNGAKEGL::`string'+0x1d61a
ffffff880`075a9a60 fffff800`018ed0e3 nt!IopSynchronousServiceTail+0x35a
ffffff880`075a9ad0 fffff800`01677853 nt!NtLockFile+0x514
ffffff880`075a9bb0 00000000`77840cea nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ fffff880`075a9c20)
00000000`0798e488 00000000`7543293b ntdll!ZwLockFile+0xa
00000000`0798e490 00000000`7541cf87 wow64!whNtLockFile+0x7f
00000000`0798e510 00000000`7536276d wow64!Wow64SystemServiceEx+0xd7
00000000`0798edd0 00000000`7541d07e wow64cpu!TurboDispatchJumpAddressEnd+0x24
00000000`0798ee90 00000000`7541c549 wow64!RunCpuSimulation+0xa
00000000`0798eee0 00000000`7786d177 wow64!Wow64LdrpInitialize+0x429
00000000`0798f430 00000000`7782308e ntdll! ?? ::FNODOBFM::`string'+0x2bfe4
00000000`0798f4a0 00000000`00000000 ntdll!LdrInitializeThunk+0xe
```

We immediately spot the anomaly of a lock file attempt and look at its IRP:

```
0: kd> !irp ffffffa8005a6c160
Irp is active with 7 stacks 7 is current (= 0xffffffa8005a6c3e0)
No Mdl: No System Buffer: Thread ffffffa8005eca060: Irp stack trace.
cmd flg cl Device File Completion-Context
[ 0, 0] 0 2 00000000 00000000 00000000-00000000

Args: 00000000 00000000 00000000 ffffffc000020c
[ 0, 0] 0 0 00000000 00000000 00000000-00000000
```

```
Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

Args: 00000000 00000000 00000000 00000000
[ 11, 0] 0 2 ffffffa8004da0620 00000000 ffffff8000177d9cc-fffffa800710e580
\FileSystem\mrxsmb mup!MupiUncProviderCompletion
Args: 00000000 00000000 00000000 00000000
>[ 11, 1] 0 0 ffffffa8004066400 ffffffa80048e66c0 00000000-00000000
\FileSystem\Mup
Args: ffffffa8004a98120 00000001 00000000 00000000
```

From that IRP we see a file name:

```
0: kd> !fileobj ffffffa80048e66c0

[...]\\AppData\\Roaming\\Vendor\\Product\\Recent\\index.dat

LockOperation Set Device Object: 0xfffffa8004066400 \FileSystem\\Mup
Vpb is NULL
Access: Read SharedRead SharedWrite SharedDelete

Flags: 0x40002
Synchronous IO
Handle Created

File Object is currently busy and has 0 waiters.

FsContext: 0xfffffa8a00e8d9010 FsContext2: 0xfffffa8a012e4d688
CurrentByteOffset: 0
Cache Data:
Section Object Pointers: ffffffa8006086928
Shared Cache Map: 00000000
File object extension is at ffffffa8005c8cbe0:
```

Alternatively we get a 32-bit stack trace from **Virtualized Process** (page 1068):

```
0: kd> .process /r /p ffffffa80063dd970
Implicit process is now ffffffa80`063dd970
Loading User Symbols

0: kd> .thread /w ffffffa8005eca060
Implicit thread is now ffffffa80`05eca060
The context is partially valid. Only x86 user-mode context is available.
x86 context set
```

```
0: kd:x86> .reload
Loading Kernel Symbols
Loading User Symbols
Loading unloaded module list
Loading Wow64 Symbols

0: kd:x86> kv
*** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr Args to Child
07ac8510 774f033f 00000390 00000000 00000000 ntdll_779d0000!ZwLockFile+0x12
07ac8590 774f00d3 061b2b68 ada9964d c0000016 kernel32!BaseDllOpenIniFileOnDisk+0x246
07ac85d0 774efae9 061b2b68 00001000 6d352f20 kernel32!BaseDllReadWriteIniFileOnDisk+0x2d
07ac85e8 775001bf 00000001 00000000 061b2b68 kernel32!BaseDllReadWriteIniFile+0xed
07ac861c 6d928401 07aca71c 00000000 00001000 kernel32!GetPrivateProfileStringW+0x35
WARNING: Stack unwind information not available. Following frames may be wrong.
07ac8640 6d9282f5 07aca71c 00000000 00000000 DLL+0x618401
[...]
07acfb14 774e3677 06757d20 07acfb60 77a09d72 DLL+0x541f6d
07acfb20 77a09d72 06757d20 eca51e43 00000000 kernel32!BaseThreadInitThunk+0xe
07acfb60 77a09d45 6d851f62 06757d20 ffffffff ntdll_779d0000!_RtlUserThreadStart+0x70
07acfb78 00000000 6d851f62 06757d20 00000000 ntdll_779d0000!_RtlUserThreadStart+0x1b
```

We get the same file name from a file handle:

```
0: kd> !handle 00000390
processor number 0, process ffffffa80063dd970
PROCESS ffffffa80063dd970
SessionId: 5 Cid: 14b0 Peb: 7efdf000 ParentCid: 1fac
DirBase: 48293000 ObjectTable: fffff8a010515f90 HandleCount: 342.
Image: Application.exe

Handle table at fffff8a0083e9000 with 444 Entries in use
0390: Object: ffffffa80048e66c0 GrantedAccess: 00120089 Entry: fffff8a00866fe40
Object: ffffffa80048e66c0 Type: (fffffa8003cf0b40) File
ObjectHeader: ffffffa80048e6690 (new version)
HandleCount: 1 PointerCount: 3
Directory Object: 00000000 Name: [...]\\AppData\\Roaming\\Vendor\\Product\\Recent\\index.dat {Mup}
```

Also, we have c0000016 error code on raw stack and examine it too:

```
0: kd> !error c0000016
Error code: (NTSTATUS) 0xc0000016 (3221225494) - {Still Busy} The specified I/O request packet (IRP)
cannot be disposed of because the I/O operation is not complete.
```

## Blocking Module

We would like to add this pattern in addition to **Blocked Thread** (page 82) and endpoint threads of **Wait Chain** (page 1092) patterns to account for modules calling waiting or delaying functions, for example:

```
0:017> kL  
ChildEBP RetAddr  
02c34100 7c90df5a ntdll!KiFastSystemCallRet  
02c34104 7c8025db ntdll!ZwWaitForSingleObject+0xc  
02c34168 7c802542 kernel32!WaitForSingleObjectEx+0xa8  
02c3417c 009f0ed9 kernel32!WaitForSingleObject+0x12  
02c34a08 00bc2c9a ModuleA!DLLCanUnloadNow+0x6db39  
02c3526c 00bc2fa4 ModuleA!DLLCanUnloadNow+0x23f8fa  
02c35ae0 00f6413c ModuleA!DLLCanUnloadNow+0x23fc04  
02c363e8 00c761ab ModuleA!DLLCanUnloadNow+0x5e0d9c  
02c36c74 00c74daa ModuleA!DLLCanUnloadNow+0x2f2e0b  
02c374e4 3d1a9eb4 ModuleA!DLLCanUnloadNow+0x2f1a0a  
02c3753c 3d0ed032 mshtml!CView::SetObjectRectsHelper+0x98  
02c37578 3cf7e43b mshtml!CView::EndDeferSetObjectRects+0x75  
02c375bc 3cf2542d mshtml!CView::EnsureView+0x39f  
02c375d8 3cf4072c mshtml!CElement::EnsureRecalcNotify+0x17c  
02c37614 3cf406ce mshtml!CElement::get_clientHeight_Logical+0x54  
02c37628 3d0822a1 mshtml!CElement::get_clientHeight+0x27  
02c37648 3cf8ad53 mshtml!G_LONG+0x7b  
02c376bc 3cf96e21 mshtml!CBase::ContextInvokeEx+0x5d1  
02c3770c 3cfa2baf mshtml!CElement::ContextInvokeEx+0x9d  
02c37738 3cf8a751 mshtml!CElement::VersionedInvokeEx+0x2d  
[ ... ]
```

## Comments

**!stacks** command may show the possible candidates too.

## Broken Link

Sometimes we have a broken linked list for some reason, either from memory corruption, **Lateral Damage** (page 602) or **Truncated Dump** (page 1015). For example, an active process list enumeration stopped after showing some processes (**!for\_each\_thread** and **lvm** also don't work):

```
0: kd> !process 0 3f
[...]
TYPE mismatch for process object at ffffffa80041da5c0
0: kd> !validateplist nt!PsActiveProcessHead
Blink at address ffffffa80041da748 does not point back to previous at ffffffa8005bc8cb8
```

Here we can either try to repair or navigate links manually or use other means such as dumping pool allocations for process structures with *Proc* pool tag:

```
0: kd> !poolfind Proc

Searching NonPaged pool (ffffffa80032fc000 : ffffffe000000000) for Tag: Proc

*ffffffa80033879a0 size: 510 previous size: a0 (Allocated) Proc (Protected)
*ffffffa80033ffad0 size: 530 previous size: 280 (Allocated) Proc (Protected)
*ffffffa80041a2af0 size: 510 previous size: 90 (Allocated) Proc (Protected)
*ffffffa800439c5c0 size: 530 previous size: 80 (Allocated) Proc (Protected)
[...]
*ffffffa8007475ad0 size: 530 previous size: 30 (Allocated) Proc (Protected)
*ffffffa80074e8490 size: 530 previous size: 100 (Allocated) Proc (Protected)
*ffffffa80075ee0b0 size: 530 previous size: b0 (Free) Pro.
*ffffffa800761d000 size: 530 previous size: 0 (Free) Pro.
*ffffffa8007645ad0 size: 530 previous size: b0 (Allocated) Proc (Protected)

0: kd> dc ffffffa8007645ad0
ffffffa80'07645ad0 0253000b e36f7250 07644030 ffffffa80 ..S.Pro.0.d.....
ffffffa80'07645ae0 00001000 00000528 00000068 fffff800 ....(...h.....
ffffffa80'07645af0 01a1a940 fffff800 00080090 00490024 @.....$.
ffffffa80'07645b00 000000c4 00000000 00000008 00000000 .....
ffffffa80'07645b10 00000000 00000000 00080007 00300033 .....3.0.
ffffffa80'07645b20 01a1a940 fffff800 013cfeae fffff8a0 @.....<.....
ffffffa80'07645b30 00580003 00000000 05ba19a0 fffffa80 ..X.....
ffffffa80'07645b40 05ba19a0 ffffffa80 07645b48 ffffffa80 .....H[d.....
```

```
0: kd> !process ffffffa80`07645b30 3f
PROCESS ffffffa8007645b30
SessionId: 0 Cid: 14c4 Peb: 7fffffff4000 ParentCid: 02c4
DirBase: 7233e000 ObjectTable: ffffff8a0014d4220 HandleCount: 399.
Image: AppA.exe
VadRoot ffffffa80072bc5b0 Vads 239 Clone 0 Private 24675. Modified 23838. Locked 0.
DeviceMap ffffff8a0000088f0
Token ffffff8a000f28060
ElapsedTime 00:00:53.066
UserTime 00:00:00.000
KernelTime 00:00:00.000
QuotaPoolUsage[PagedPool] 0
QuotaPoolUsage[NonPagedPool] 0
Working Set Sizes (now,min,max) (11960, 50, 345) (47840KB, 200KB, 1380KB)
PeakWorkingSetSize 74346
VirtualSize 331 Mb
PeakVirtualSize 478 Mb
PageFaultCount 92214
MemoryPriority BACKGROUND
BasePriority 8
CommitCharge 25905

[...]
```

## Busy System

If there are no CPU-bound threads in a system, then most of the time processors are looping in the so-called idle thread where they are halted waiting for an interrupt to occur (HLT instruction<sup>11</sup>). When an interrupt occurs, they process a DPC list and then do thread scheduling if necessary as evident from the stack trace and its functions disassembly below. If we have a memory dump, one of running threads would be the one that called *KeBugCheck(Ex)* function.

```
3: kd> !running

System Processors f (affinity mask)
Idle Processors d

      Prcb      Current   Next
1  f7737120  8a3da020          .....
3: kd> !thread 8a3da020 1f
THREAD 8a3da020  Cid 0ebc.0dec  Teb: 7fffd000 Win32Thread: bc002328 RUNNING on processor 1
Not impersonating
DeviceMap          e3e3e080
Owning Process    8a0aea88  Image:       SystemDump.exe
Wait Start TickCount 17154        Ticks: 0
Context Switch Count 568          LargeStack
UserTime           00:00:00.046
KernelTime         00:00:00.375
Win32 Start Address 0x0040fe92
Start Address 0x77e6b5c7
Stack Init f4266000 Current f4265c08 Base f4266000 Limit f4261000 Call 0
Priority 11 BasePriority 10 PriorityDecrement 0
ChildEBP RetAddr
f4265bec f79c9743 nt!KeBugCheckEx+0x1b
WARNING: Stack unwind information not available. Following frames may be wrong.
f4265c38 8081dce5 SystemDump+0x743
f4265c4c 808f4797 nt!IofCallDriver+0x45
f4265c60 808f5515 nt!IoSynchronousServiceTail+0x10b
f4265d00 808ee0e4 nt!IoPxxxControlFile+0x5db
f4265d34 80888c6c nt!NtDeviceIoControlFile+0x2a
f4265d34 7c82ed54 nt!KiFastCallEntry+0xfc

3: kd> !ready
Processor 0: No threads in READY state
Processor 1: No threads in READY state
Processor 2: No threads in READY state
Processor 3: No threads in READY state

3: kd> ~2s
```

---

<sup>11</sup> <http://www.asmpedia.org/index.php?title=HLT>

```
2: kd> !thread -1 1f
THREAD f7742090 Cid 0000.0000 Peb: 00000000 Win32Thread: 00000000 RUNNING on processor 2
Not impersonating
Owning Process          8089db40      Image:           Idle
Wait Start TickCount     0                  Ticks: 17154 (0:00:04:28.031)
Context Switch Count    193155
UserTime                 00:00:00.000
KernelTime               00:03:23.328
Stack Init f78b7000 Current f78b6d4c Base f78b7000 Limit f78b4000 Call 0
Priority 0 BasePriority 0 PriorityDecrement 0
ChildEBP RetAddr
f78b6d50 8088d262 intelppm!AcpiC1Idle+0x12
f78b6d54 00000000 nt!KiIdleLoop+0xa

2: kd> .asm no_code_bytes
Assembly options: no_code_bytes

2: kd> uf intelppm!AcpiC1Idle
intelppm!AcpiC1Idle:
f6e73c90 push    ecx
f6e73c91 push    0
f6e73c93 call    intelppm!KeQueryPerformanceCounter (f6e740c6)
f6e73c98 mov     ecx,dword ptr [esp]
f6e73c9b mov     dword ptr [ecx],eax
f6e73c9d mov     dword ptr [ecx+4],edx
f6e73ca0 sti
f6e73ca1 hlt
f6e73ca2 push    0
f6e73ca4 call    intelppm!KeQueryPerformanceCounter (f6e740c6)
f6e73ca9 pop     ecx
f6e73caa mov     dword ptr [ecx+8],eax
f6e73cad mov     dword ptr [ecx+0Ch],edx
f6e73cb0 xor     eax,eax
f6e73cb2 ret

2: kd> uf nt!KiIdleLoop
nt!KiIdleLoop:
8088d258 jmp     nt!KiIdleLoop+0xa (8088d262)

nt!KiIdleLoop+0x2:
8088d25a lea     ecx,[ebx+0EC0h]
8088d260 call   dword ptr [ecx]

nt!KiIdleLoop+0xa:
8088d262 pause ; http://www.asmpedia.org/index.php?title=PAUSE
8088d264 sti
8088d265 nop
8088d266 nop
8088d267 cli
8088d268 mov     eax,dword ptr [ebx+0A4Ch]
8088d26e or      eax,dword ptr [ebx+0A88h]
8088d274 or      eax,dword ptr [ebx+0C10h]
8088d27a je      nt!KiIdleLoop+0x37 (8088d28f)
```

```

nt!KiIdleLoop+0x24:
8088d27c mov     cl,2
8088d27e call    dword ptr [nt!_imp_HalClearSoftwareInterrupt (808010a8)]
8088d284 lea     ecx,[ebx+120h]
8088d28a call    nt!KiRetireDpcList (80831be8)

nt!KiIdleLoop+0x37:
8088d28f cmp     dword ptr [ebx+128h],0
8088d296 je      nt!KiIdleLoop+0xca (8088d322)

nt!KiIdleLoop+0x44:
8088d29c mov     ecx,1Bh
8088d2a1 call    dword ptr [nt!_imp_KfRaiseIrql (80801100)]
8088d2a7 sti
8088d2a8 mov     edi,dword ptr [ebx+124h]
8088d2ae mov     byte ptr [edi+5Dh],1
8088d2b2 lock bts dword ptr [ebx+0A7Ch],0
8088d2bb jae    nt!KiIdleLoop+0x70 (8088d2c8)

nt!KiIdleLoop+0x65:
8088d2bd lea     ecx,[ebx+0A7Ch]
8088d2c3 call   nt!KefAcquireSpinLockAtDpcLevel (80887fd0)

nt!KiIdleLoop+0x70:
8088d2c8 mov     esi,dword ptr [ebx+128h]
8088d2ce cmp     esi,edi
8088d2d0 je      nt!KiIdleLoop+0xb3 (8088d30b)

nt!KiIdleLoop+0x7a:
8088d2d2 and    dword ptr [ebx+128h],0
8088d2d9 mov     dword ptr [ebx+124h],esi
8088d2df mov     byte ptr [esi+4Ch],2
8088d2e3 and    byte ptr [ebx+0AA3h],0
8088d2ea and    dword ptr [ebx+0A7Ch],0

nt!KiIdleLoop+0x99:
8088d2f1 mov     ecx,1
8088d2f6 call   nt!SwapContext (8088d040)
8088d2fb mov     ecx,2
8088d300 call   dword ptr [nt!_imp_KfLowerIrql (80801104)]
8088d306 jmp    nt!KiIdleLoop+0xa (8088d262)

nt!KiIdleLoop+0xb3:
8088d30b and    dword ptr [ebx+128h],0
8088d312 and    dword ptr [ebx+0A7Ch],0
8088d319 and    byte ptr [edi+5Dh],0
8088d31d jmp    nt!KiIdleLoop+0xa (8088d262)

nt!KiIdleLoop+0xca:
8088d322 cmp     byte ptr [ebx+0AA3h],0
8088d329 je      nt!KiIdleLoop+0x2 (8088d25a)

```

```

nt!KiIdleLoop+0xd7:
8088d32f sti
8088d330 lea    ecx,[ebx+120h]
8088d336 call   nt!KiIdleSchedule (808343e6)
8088d33b test   eax,eax
8088d33d mov    esi,eax
8088d33f mov    edi,dword ptr [ebx+12Ch]
8088d345 jne    nt!KiIdleLoop+0x99 (8088d2f1)

nt!KiIdleLoop+0xef:
8088d347 jmp    nt!KiIdleLoop+0xa (8088d262)

```

In some memory dumps taken when systems or sessions were hanging or very slow for some time we might see **Busy System** pattern where all processors execute non-idle threads, and there are threads in ready queues waiting to be scheduled:

```

3: kd> !running

System Processors f (affinity mask)
  Idle Processors 0

      Prcb      Current    Next
  0  ffdff120  88cef850      .....
  1  f7727120  8940b7a0      .....
  2  f772f120  8776f020      .....
  3  f7737120  87b25360      .....

3: kd> !ready
Processor 0: Ready Threads at priority 8
  THREAD 88161668 Cid 3d58.43a0 Teb: 7ffd000 Win32Thread: bc1eba48 READY
  THREAD 882d0020 Cid 1004.0520 Teb: 7ffd000 Win32Thread: bc230838 READY
  THREAD 88716b40 Cid 2034.241c Teb: 7ffd000 Win32Thread: bc11b388 READY
  THREAD 88bf7978 Cid 2444.2564 Teb: 7ffde000 Win32Thread: bc1ccc18 READY
  THREAD 876f7a28 Cid 2308.4bfc Teb: 7ffd000 Win32Thread: bc1f7b98 READY
Processor 0: Ready Threads at priority 0
  THREAD 8a3925a8 Cid 0004.0008 Teb: 00000000 Win32Thread: 00000000 READY
Processor 1: Ready Threads at priority 9
  THREAD 87e69db0 Cid 067c.3930 Teb: 7ffdb000 Win32Thread: bc180990 READY
Processor 1: Ready Threads at priority 8
  THREAD 88398c70 Cid 27cc.15b4 Teb: 7ffde000 Win32Thread: bc159ea8 READY
Processor 2: Ready Threads at priority 8
  THREAD 8873cdb0 Cid 4c24.4384 Teb: 7ffd000 Win32Thread: bc1c9838 READY
  THREAD 89f331e0 Cid 453c.4c68 Teb: 7ffd000 Win32Thread: bc21dbd0 READY
  THREAD 889a03f0 Cid 339c.2fcc Teb: 7ffd000 Win32Thread: bc1cdbe8 READY
  THREAD 87aacdb0 Cid 3b80.4ed0 Teb: 7ffde000 Win32Thread: bc1c5d10 READY
Processor 3: No threads in READY state

```

Here is another example from a busy 8-processor system where only one processor was idle at the time of the bugcheck:

```

5: kd> !ready
Processor 0: No threads in READY state
Processor 1: No threads in READY state
Processor 2: No threads in READY state
Processor 3: No threads in READY state
Processor 4: No threads in READY state
Processor 5: No threads in READY state
Processor 6: No threads in READY state
Processor 7: No threads in READY state

5: kd> !running

System Processors ff (affinity mask)
  Idle Processors 1

      Prcb      Current    Next
1 f7727120 8713a5a0      .....
2 f772f120 86214750      .....
3 f7737120 86f87020      .....
4 f773f120 86ffe700      .....
5 f7747120 86803a90      .....
6 f774f120 86043db0      .....
7 f7757120 86bcbdb0      .....

5: kd> !thread 8713a5a0 1f
THREAD 8713a5a0 Cid 4ef4.4f04 Peb: 7fffd000 Win32Thread: bc423920 RUNNING on processor 1
Not impersonating
DeviceMap          e44e9a40
Owning Process     864d1d88 Image:      SomeExe.exe
Wait Start TickCount 1415535 Ticks: 0
Context Switch Count 7621092 LargeStack
UserTime           00:06:59.218
KernelTime         00:19:26.359
Win32 Start Address BROWSEUI!BrowserProtectedThreadProc (0x75ec1c3f)
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init b68b8a70 Current b68b8c28 Base b68b9000 Limit b68b1000 Call b68b8a7c
Priority 13 BasePriority 13 PriorityDecrement 0
ChildEBP RetAddr
00c1f4fc 773dc4e4 USER32!DispatchHookA+0x35
00c1f528 7739c9c6 USER32!fnHkINLPCWPRETSTRUCTA+0x60
00c1f550 7c828536 USER32!__fnDWORD+0x24
00c1f550 808308f4 ntdll!KiUserCallbackDispatcher+0x2e
b68b8a94 8091d6d1 nt!KiCallUserMode+0x4
b68b8aec bf8a26d3 nt!KeUserModeCallback+0x8f
b68b8b70 bf89dd4d win32k!SfnDWORD+0xb4
b68b8be8 bf89d79d win32k!xxxHkCallHook+0x22c
b68b8c90 bf89da19 win32k!xxxCallHook2+0x245
b68b8cac bf8a137a win32k!xxxCallHook+0x26
b68b8cec bf85af67 win32k!xxxSendMessageTimeout+0x1e3
b68b8d10 bf8c182c win32k!xxxWrapSendMessage+0x1b
b68b8d40 8088978c win32k!NtUserMessageCall+0x9d
b68b8d40 7c8285ec nt!KiFastCallEntry+0xfc
00c1f550 7c828536 ntdll!KiFastSystemCallRet
00c1f57c 7739d1ec ntdll!KiUserCallbackDispatcher+0x2e

```

```

00c1f5b8 7738cee9 USER32!NtUserMessageCall+0xc
00c1f5d8 01438f73 USER32!SendMessageA+0x7f

5: kd> !thread 86214750
THREAD 86214750 Cid 0b94.1238 Peb: 7ffdb000 Win32Thread: bc2f5ea8 RUNNING on processor 2
Not impersonating
DeviceMap          e3482310
Owning Process    85790020 Image:      SomeExe.exe
Wait Start TickCount 1415535 Ticks: 0
Context Switch Count 1745682 LargeStack
UserTime           00:01:20.031
KernelTime         00:04:03.484
Win32 Start Address 0x75ec1c3f
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init b4861000 Current b4860558 Base b4861000 Limit b4856000 Call 0
Priority 13 BasePriority 13 PriorityDecrement 0
ChildEBP RetAddr
b4860bd8 bf8da699 nt!PsGetThreadProcess
b4860bf4 bf89d6e6 win32k!IsRestricted+0x2f
b4860c90 bf89da19 win32k!xxxCallHook2+0x12d
b4860cac bf8a137a win32k!xxxCallHook+0x26
b4860cec bf85af67 win32k!xxxSendMessageTimeout+0x1e3
b4860d10 bf8c182c win32k!xxxWrapSendMessage+0x1b
b4860d40 8088978c win32k!NtUserMessageCall+0x9d
b4860d40 7c8285ec nt!KiFastCallEntry+0xfc
00c1f5fc 00000000 ntdll!KiFastSystemCallRet

5: kd> !thread 86f87020 1f
THREAD 86f87020 Cid 0238.0ae8 Peb: 7ffa5000 Win32Thread: 00000000 RUNNING on processor 3
IRP List:
86869200: (0006,0094) Flags: 00000900 Mdl: 00000000
85b2a7f0: (0006,0094) Flags: 00000900 Mdl: 00000000
86f80a20: (0006,0094) Flags: 00000800 Mdl: 00000000
85e6af68: (0006,0094) Flags: 00000900 Mdl: 00000000
892a6c78: (0006,0094) Flags: 00000900 Mdl: 00000000
85d06070: (0006,0094) Flags: 00000900 Mdl: 00000000
85da35e0: (0006,0094) Flags: 00000900 Mdl: 00000000
87216340: (0006,0094) Flags: 00000900 Mdl: 00000000
Not impersonating
DeviceMap          e1003940
Owning Process    8850e020 Image:      lsass.exe
Wait Start TickCount 1415535 Ticks: 0
Context Switch Count 39608
UserTime           00:00:01.625
KernelTime         00:00:05.437
Win32 Start Address RPCRT4!ThreadStartRoutine (0x77c7b0f5)
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init f4925000 Current f4924c38 Base f4925000 Limit f4922000 Call 0
Priority 10 BasePriority 9 PriorityDecrement 0
ChildEBP RetAddr
f4924640 80972e8e nt!SePrivilegeCheck+0x24
f4924678 80944aa0 nt!SeSinglePrivilegeCheck+0x3a
f4924770 8088978c nt!NtOpenProcess+0x13a
f4924770 8082eff5 nt!KiFastCallEntry+0xfc
f49247f8 f6037bee nt!ZwOpenProcess+0x11

```

```
WARNING: Stack unwind information not available. Following frames may be wrong.  
f4924830 f6002996 SomeDrv+0x48bee
```

```
5: kd> !thread 86ffe700 1f  
THREAD 86ffe700 Cid 1ba4.1ba8 Teb: 7fffd000 Win32Thread: bc23cea8 RUNNING on processor 4  
Not impersonating  
DeviceMap e44e9a40  
Owning Process 87005708 Image: WINWORD.EXE  
Wait Start TickCount 1415535 Ticks: 0  
Context Switch Count 1547251 LargeStack  
UserTime 00:01:00.750  
KernelTime 00:00:45.265  
Win32 Start Address WINWORD (0x300019b0)  
Start Address kernel32!BaseProcessStartThunk (0x77e617f8)  
Stack Init f3465000 Current f3464c48 Base f3465000 Limit f345e000 Call 0  
Priority 8 BasePriority 8 PriorityDecrement 0  
ChildEBP RetAddr  
f3464d64 7c8285eb nt!KiFastCallEntry+0x91  
f3464d68 badb0d00 ntdll!KiFastSystemCall+0x3
```

```
5: kd> !thread 86803a90 1f  
THREAD 86803a90 Cid 3c20.29f8 Teb: 7fffd000 Win32Thread: bc295480 RUNNING on processor 5  
Not impersonating  
DeviceMap e518c6b8  
Owning Process 857d5500 Image: SystemDump.exe  
Wait Start TickCount 1415535 Ticks: 0  
Context Switch Count 310 LargeStack  
UserTime 00:00:00.015  
KernelTime 00:00:00.046  
*** ERROR: Module load completed but symbols could not be loaded for SystemDump.exe  
Win32 Start Address SystemDump_400000 (0x0040fe92)  
Start Address kernel32!BaseProcessStartThunk (0x77e617f8)  
Stack Init b38a4000 Current b38a3c08 Base b38a4000 Limit b389f000 Call 0  
Priority 11 BasePriority 8 PriorityDecrement 2  
ChildEBP RetAddr Args to Child  
b38a3bf0 f79e3743 000000e2 cccccccc 866962b0 nt!KeBugCheckEx+0x1b  
WARNING: Stack unwind information not available. Following frames may be wrong.  
b38a3c3c 8081df65 SystemDump+0x743  
b38a3c50 808f5437 nt!IofCallDriver+0x45  
b38a3c64 808f61bf nt!IopSynchronousServiceTail+0x10b  
b38a3d00 808eed08 nt!IopXxxControlFile+0x5e5  
b38a3d34 8088978c nt!NtDeviceIoControlFile+0x2a  
b38a3d34 7c8285ec nt!KiFastCallEntry+0xfc  
0012efc4 7c826fcb ntdll!KiFastSystemCallRet  
0012efc8 77e416f5 ntdll!NtDeviceIoControlFile+0xc  
0012f02c 00402208 kernel32!DeviceIoControl+0x137  
0012f884 00404f8e SystemDump_400000+0x2208
```

```

5: kd> !thread 86043db0 1f
THREAD 86043db0  Cid 0610.55dc  Teb: 7ffa1000 Win32Thread: 00000000 RUNNING on processor 6
IRP List:
  86dc99a0: (0006,0094) Flags: 00000a00  Mdl: 00000000
Impersonation token: e7b30030 (Level Impersonation)
DeviceMap          e4e470a8
Owning Process    891374a8      Image:      SomeSvc.exe
Wait Start TickCount 1415215      Ticks: 320 (0:00:00:05.000)
Context Switch Count 11728
UserTime           00:00:02.546
KernelTime         00:02:57.765
Win32 Start Address 0x0082b983
LPC Server thread working on message Id 82b983
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init b49c1000 Current b49c0a7c Base b49c1000 Limit b49be000 Call 0
Priority 8 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr
b49c0b80 8087c9c0 hal!KeReleaseQueuedSpinLock+0x2d
b49c0ba0 8087ca95 nt!ExReleaseResourceLite+0xac
b49c0ba4 f6faa5ae nt!ExReleaseResourceAndLeaveCriticalSection+0x5
b49c0bb8 f6faad05 termdd!_IcaCallStack+0x60
b49c0bdc f6fa6bda termdd!IcaCallDriver+0x71
b49c0c34 f6fa86dc termdd!IcaWriteChannel+0xd8
b49c0c50 f6fa8cc6 termdd!IcaWrite+0x40
b49c0c68 8081df65 termdd!IcaDispatch+0xd0
b49c0c7c 808f5437 nt!IofCallDriver+0x45
b49c0c90 808f3157 nt!IopSynchronousServiceTail+0x10b
b49c0d38 8088978c nt!NtWriteFile+0x663
b49c0d38 7c8285ec nt!KiFastCallEntry+0xfc
0254d814 7c827d3b ntdll!KiFastSystemCallRet
0254d818 77e5b012 ntdll!NtWriteFile+0xc
0254d878 004389f2 kernel32!WriteFile+0xa9

```

```

5: kd> !thread 86bcdb0 1f
THREAD 86bcdb0  Cid 34ac.1b04  Teb: 7fffd000 Win32Thread: bc3d9a48 RUNNING on processor 7
IRP List:
  8581d900: (0006,01fc) Flags: 00000884  Mdl: 00000000
Not impersonating
DeviceMap          e153fc48
Owning Process    872fb708      Image:      SomeExe.exe
Wait Start TickCount 1415535      Ticks: 0
Context Switch Count 7655285      LargeStack
UserTime           00:10:09.343
KernelTime         00:30:21.296
Win32 Start Address 0x75ec1c3f
Start Address 0x77e617ec
Stack Init b86cb000 Current b86ca58c Base b86cb000 Limit b86c2000 Call 0
Priority 13 BasePriority 13 PriorityDecrement 0
ChildEBP RetAddr
b86ca974 f724ffc2 fltmgr!FltpPerformPostCallbacks+0x260
b86ca988 f72504f1 fltmgr!FltpProcessIoCompletion+0x10
b86ca998 f7250b83 fltmgr!FltpPassThroughCompletion+0x89
b86ca9c8 f725e5de fltmgr!FltpLegacyProcessingAfterPreCallbacksCompleted+0x269
b86caa04 8081df65 fltmgr!FltpCreate+0x26a
b86caa18 f75fa8c7 nt!IofCallDriver+0x45
b86caa40 f75faa5a SomeFlt!PassThrough+0xbb

```

```
b86caa5c 8081df65 SomeFlt!Create+0xda
b86caa70 808f8f71 nt!IoCallDriver+0x45
b86cab58 80937942 nt!IopParseDevice+0xa35
b86cabd8 80933a76 nt!ObpLookupObjectName+0x5b0
b86cac2c 808eae25 nt!ObOpenObjectByName+0xea
b86caca8 808ec0bf nt!IopCreateFile+0x447
b86cad04 808efc4f nt!IoCreateFile+0xa3
b86cad44 8088978c nt!NtOpenFile+0x27
b86cad44 7c8285ec nt!KiFastCallEntry+0xfc
```

Running threads have good chance to be **Spiking Threads** (page 888).

# C

## C++ Exception

### Linux

This is a Linux variant of **C++ Exception** pattern previously described for Mac OS X (page 109) and Windows (page 110) platforms:

```
(gdb) bt
#0 0x00007f0a1d0e5165 in *__GI_raise ()
at ../nptl/sysdeps/unix/sysv/linux/raise.c:64
#1 0x00007f0a1d0e83e0 in *__GI_abort () at abort.c:92
#2 0x00007f0a1db5789d in __gnu_cxx::__verbose_terminate_handler() ()
from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
#3 0x00007f0a1db55996 in ?? () from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
#4 0x00007f0a1db559c3 in std::terminate() ()
from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
#5 0x00007f0a1db55bee in __cxa_throw ()
from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
#6 0x0000000000400dcf in procB() ()
#7 0x0000000000400e26 in procA() ()
#8 0x0000000000400e88 in procNH() ()
#9 0x0000000000400ea8 in bar_one() ()
#10 0x0000000000400eb3 in foo_one() ()
#11 0x0000000000400ec6 in thread_one(void*) ()
#12 0x00007f0a1d444b50 in start_thread ()
#13 0x00007f0a1d18e95d in clone ()
at ../sysdeps/unix/sysv/linux/x86_64/clone.S:112
#14 0x0000000000000000 in ?? ()
```

## Mac OS X

This is a Mac OS X / GDB counterpart to **C++ Exception** pattern:

```
(gdb) bt
#0 0x00007fff88bd582a in __kill ()
#1 0x00007fff8c184a9c in abort ()
#2 0x00007fff852f57bc in abort_message ()
#3 0x00007fff852f2fcf in default_terminate ()
#4 0x00007fff852f3001 in safe_handler_caller ()
#5 0x00007fff852f305c in std::terminate ()
#6 0x00007fff852f4152 in __cxa_throw ()
#7 0x000000010e402be8 in bar ()
#8 0x000000010e402c99 in foo ()
#9 0x000000010e402cbb in main (argc=1, argv=0x7fff6e001b18)
```

The modeling application source code:

```
class Exception
{
    int code;
    std::string description;
public:
    Exception(int _code, std::string _desc) : code(_code), description(_desc) {}
};

void bar()
{
    throw new Exception(5, "Access Denied");
}

void foo()
{
    bar();
}

int main(int argc, const char * argv[])
{
    foo();
    return 0;
}
```

## Windows

This is a very simple pattern, and it is similar to **Managed Code Exception** (page 617) and can be manifested by the same *RaiseException* function call on top of the stack (bold). It is called by Visual C runtime (I consider Microsoft C/C++ implementation here, msvcr.dll, bold italic). The typical example of it might be checking the validity of a C++ stream operator data format (bold underlined):

```
STACK_TEXT:
09d6f264 78007108 KERNEL32!RaiseException+0x56
09d6f2a4 677f2a88 msvcr!_CxxThrowException+0x34
09d6f2bc 6759afff DLL!MyInputStream::operator>>+0x34
```

Also, some Visual C++ STL implementations check for out of bounds or invalid parameters and call unhandled exception filter directly, for example:

```
STACK_TEXT:
0012d2e8 7c90e9ab ntdll!KiFastSystemCallRet
0012d2ec 7c8094e2 ntdll!ZwWaitForMultipleObjects+0xc
0012d388 7c80a075 kernel32!WaitForMultipleObjectsEx+0x12c
0012d3a4 6945763c kernel32!WaitForMultipleObjects+0x18
0012dd38 694582b1 faultrep!StartDWEException+0x5df
0012edac 7c8633b1 faultrep!ReportFault+0x533
0012f44c 004409b3 kernel32!UnhandledExceptionFilter+0x587
0012f784 00440a1b Application!_invoke_watson+0xc4
0012f79c 00406f4f Application!_invalid_parameter_noinfo+0xc
0012f7a0 0040566b Application!std::vector<std::basic_string<char, std::char_traits<char>, std::allocator<char> >, std::allocator<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >::operator[]+0x12
```

The latter example also shows how an unhandled exception filter in an application itself calls a postmortem debugger specified by *AeDebug* registry key (see also the article<sup>12</sup> for the detailed explanation).

## Comments

There is also **Icppexpr** WinDbg extension command to format C++ exception record contents.

---

<sup>12</sup> Who Calls the Postmortem Debugger?, Memory Dump Analysis Anthology, Volume 1, page 113

## Caller-n-Callee

We noticed this pattern when analyzing the output of **!DumpStack** WinDbg SOS extension command:

```
0:011> !DumpStack
OS Thread Id: 0xac (11)
[...]
ChildEBP RetAddr  Caller, Callee
[...]
0b73f65c 77c416dc ntdll!RtlAllocateHeap+0x17c, calling ntdll!RtlpLowFragHeapAllocFromContext
0b73f688 77c486cd ntdll!RtlAllocateHeap+0x193, calling ntdll!memset
0b73f6b0 7653a467 kernel32!TlsSetValue+0x4c, calling ntdll!RtlAllocateHeap
0b73f6cc 77a01c48 urlmon!CUrlMkTls::TLSAllocData+0x3f, calling kernel32!TlsSetValue
0b73f6dc 77a0198d urlmon!CUrlMkTls::CUrlMkTls+0x29, calling urlmon!CUrlMkTls::TLSAllocData
0b73f6e8 77a01be5 urlmon!TlsD11Main+0x100, calling urlmon!EnsureFeatureCache
0b73f6f4 6d016a21 mshtml!DllMain+0x10, calling kernel32!GetCurrentThreadId
0b73f704 6d016b6c mshtml!_CRT_INIT+0x281, calling mshtml!DllMain
0b73f71c 7239133e msimtf!_CRT_INIT+0x281, calling msimtf!DllMain
0b73f728 72391375 msimtf!_CRT_INIT+0x3e7, calling msimtf!_SEH_epilog4
0b73f764 6d016ad0 mshtml!_DllMainStartup+0x56, calling mshtml!_DllMainCRTStartup
0b73f778 72391375 msimtf!_CRT_INIT+0x3e7, calling msimtf!_SEH_epilog4
0b73f77c 77c4a604 ntdll!LdrpCallInitRoutine+0x14
0b73f7a4 77c1ab6c ntdll!LdrpInitializeThread+0x1e9, calling ntdll!RtlLeaveCriticalSection
0b73f7ac 77c1a9ea ntdll!LdrpInitializeThread+0x1cd, calling ntdll!_SEH_epilog4
0b73f800 77c1ab15 ntdll!LdrpInitializeThread+0x11f, calling ntdll!RtlActivateActivationContextUnsafeFast
0b73f804 77c1ab53 ntdll!LdrpInitializeThread+0x167, calling
ntdll!RtlDeactivateActivationContextUnsafeFast
0b73f838 77c1a9ea ntdll!LdrpInitializeThread+0x1cd, calling ntdll!_SEH_epilog4
0b73f83c 77c405a0 ntdll!NtTestAlert+0xc
0b73f840 77c1a968 ntdll!_LdrpInitialize+0x29c, calling ntdll!_SEH_epilog4
0b73f8a0 77c3f3d0 ntdll!NtContinue+0xc
0b73f8a4 77c1a98a ntdll!LdrInitializeThunk+0x1a, calling ntdll!NtContinue
0b73fb30 6afdf59f6 clr!Thread::intermediateThreadProc+0x39, calling clr!_alloca_probe_16
0b73fb44 76573833 kernel32!BaseThreadInitThunk+0xe
0b73fb50 77c1a9bd ntdll!_RtlUserThreadStart+0x23
```

Obviously the command collected “call-type” **Execution Residue** (page 371) from the raw stack. The “calling” part wasn’t found in the nearby region:

```
0:011> dps 0b73f7a4-20 0b73f7a4+20
0b73f784 72390000 msimtf!__imp__RegOpenKeyW <PERF> (msimtf+0x0)
0b73f788 00000002
0b73f78c 00000000
0b73f790 00000001
0b73f794 0b73f80c
0b73f798 0b73f80c
0b73f79c 00000001
0b73f7a0 05636578
0b73f7a4 0b73f83c
0b73f7a8 77c1ab6c ntdll!LdrpInitializeThread+0x1e9
0b73f7ac 77ca5340 ntdll!LdrpLoaderLock
0b73f7b0 77c1a9ea ntdll!LdrpInitializeThread+0x1cd
0b73f7b4 0b7321f2
```

```
0b73f7b8 7ff4e000
0b73f7bc 7ffd000
0b73f7c0 77ca51f4 ntdll!LdrpProcessInitialized
0b73f7c4 00000000
```

We tried to disassemble backward the addresses and found the callees:

```
0:011> ub 77c1ab6c
ntdll!LdrpInitializeThread+0x16b:
77c1ab57 90 nop
77c1ab58 90 nop
77c1ab59 90 nop
77c1ab5a 90 nop
77c1ab5b 90 nop
77c1ab5c ff054452ca77 inc dword ptr [ntdll!LdrpActiveThreadCount (77ca5244)]
77c1ab62 684053ca77 push offset ntdll!LdrpLoaderLock (77ca5340)
77c1ab67 e8bd820000 call ntdll!RtlLeaveCriticalSection (77c22e29)

0:011> ub 77a01be5
urlmon!TlsDllMain+0x2f:
77a01bce 8d4510 lea eax,[ebp+10h]
77a01bd1 50 push eax
77a01bd2 8d4d0c lea ecx,[ebp+0Ch]
77a01bd5 e88efdf7ff call urlmon!CUrlMkTls::CUrlMkTls (77a01968)
77a01bda 397d10 cmp dword ptr [ebp+10h],edi
77a01bdd 7c09 jl urlmon!TlsDllMain+0x103 (77a01be8)
77a01bdf 56 push esi
77a01be0 e887fcffff call urlmon!EnsureFeatureCache (77a0186c)
```

In the past, we were frequently referencing this pattern especially when discussing **Coincidental Symbolic Information** (page 137) but didn't name it.

We can also run **!DumpStack** command against every thread (including non-managed) to get the summary of the call-type execution residue:

```
0:011> ~4s
eax=76573821 ebx=00000002 ecx=00000000 edx=74d01909 esi=00000000 edi=00000000
eip=77c40f34 esp=0478f8a0 ebp=0478f93c iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
ntdll!KiFastSystemCallRet:
77c40f34 c3 ret

0:004> k
ChildEBP RetAddr
0478f89c 77c40690 ntdll!KiFastSystemCallRet
0478f8a0 76577e09 ntdll!ZwWaitForMultipleObjects+0xc
0478f93c 7674c4af kernel32!WaitForMultipleObjectsEx+0x11d
0478f990 76748b7b user32!RealMsgWaitForMultipleObjectsEx+0x13c
0478f9ac 74d01965 user32!MsgWaitForMultipleObjects+0x1f
0478f9f8 76573833 GdiPlus!BackgroundThreadProc+0x59
0478fa04 77c1a9bd kernel32!BaseThreadInitThunk+0xe
0478fa44 00000000 ntdll!_RtlUserThreadStart+0x23
```

```
0:004> !DumpStack
OS Thread Id: 0x950 (4)
Current frame: ntdll!KiFastSystemCallRet
ChildEBP RetAddr Caller, Callee
0478f89c 77c40690 ntdll!ZwWaitForMultipleObjects+0xc
0478f8a0 76577e09 kernel32!WaitForMultipleObjectsEx+0x11d, calling ntdll!NtWaitForMultipleObjects
0478f914 76751a91 user32!UserCallWinProcCheckWow+0x5c, calling
ntdll!RtlActivateActivationContextUnsafeFast
0478f918 76751b41 user32!UserCallWinProcCheckWow+0x16a, calling
ntdll!RtlDeactivateActivationContextUnsafeFast
0478f93c 7674c4af user32!RealMsgWaitForMultipleObjectsEx+0x13c, calling kernel32!WaitForMultipleObjectsEx
0478f968 76752a65 user32!DispatchMessageWorker+0x396, calling user32!_SEH_epilog4
0478f980 76743c64 user32!PeekMessageA+0x129, calling user32!_PeekMessage
0478f990 76748b7b user32!MsgWaitForMultipleObjects+0x1f, calling user32!MsgWaitForMultipleObjectsEx
0478f9ac 74d01965 GdiPlus!BackgroundThreadProc+0x59, calling user32!MsgWaitForMultipleObjects
0478f9f8 76573833 kernel32!BaseThreadInitThunk+0xe
0478fa04 77c1a9bd ntdll!_RtlUserThreadStart+0x23
```

## Changed Environment

Sometimes the change of operating system version or installing an intrusive product reveals hidden bugs in software that was working perfectly before that.

What happens after installing the new software? If we look at the process dump, we see many DLLs loaded at their specific virtual addresses. Here is the output from `!m` WinDbg command after attaching to `iexplore.exe` process running on Windows XP SP2 workstation:

```
0:000> !m
start      end        module name
00400000 00419000 iexplore
01c80000 01d08000 shdoclc
01d10000 01fd5000 xpsp2res
022b0000 022cd000 xpsp3res
02680000 02946000 msi
031f0000 031fd000 LvHook
03520000 03578000 PortableDeviceApi
037e0000 037f7000 odbcint
0ffd0000 0fff8000 rsaenh
20000000 20012000 browselc
30000000 302ee000 Flash9b
325c0000 325d2000 mshev
4d4f0000 4d548000 WINHTTP
5ad70000 5ada8000 UxTheme
5b860000 5b8b4000 NETAPI32
5d090000 5d12a000 comctl32_5d090000
5e310000 5e31c000 pngfilt
63000000 63014000 SynTPFcs
662b0000 66308000 hnetcfg
66880000 6688c000 ImgUtil
6bdd0000 6be06000 dxtrans
6be10000 6be6a000 dxtmsft
6d430000 6d43a000 ddrawex
71a50000 71a8f000 mssock
71a90000 71a98000 wshtcpip
71aa0000 71aa8000 WS2HELP
71ab0000 71ac7000 WS2_32
71ad0000 71ad9000 wssock32
71b20000 71b32000 MPR
71bf0000 71c03000 SAMLIB
71c10000 71c1e000 ntlanman
71c80000 71c87000 NETRAP
71c90000 71cd0000 NETUI1
71cd0000 71ce7000 NETUI0
71d40000 71d5c000 actxprxy
722b0000 722b5000 sensapi
72d10000 72d18000 msacm32
72d20000 72d29000 wdmaud
73300000 73367000 vbscript
73760000 737a9000 DDRAW
73bc0000 73bc6000 DCIMAN32
73dd0000 73ece000 MFC42
```

74320000 7435d000 ODBC32  
746c0000 746e7000 mslns31  
746f0000 7471a000 msimtf  
74720000 7476b000 MSCTF  
754d0000 75550000 CRYPTUI  
75970000 75a67000 MSGINA  
75c50000 75cbe000 jscript  
75cf0000 75d81000 mlang  
75e90000 75f40000 SXS  
75f60000 75f67000 drprov  
75f70000 75f79000 davclnt  
75f80000 7607d000 BROWSEUI  
76200000 76271000 mshtmlled  
76360000 76370000 WINSTA  
76390000 763ad000 IMM32  
763b0000 763f9000 comdlg32  
76600000 7661d000 CSCDLL  
767f0000 76817000 schannel  
769c0000 76a73000 USERENV  
76b20000 76b31000 ATL  
76b40000 76b6d000 WINMM  
76bf0000 76bfb000 PSAPI  
76c30000 76c5e000 WINTRUST  
76c90000 76cb8000 IMAGEHLP  
76d60000 76d79000 iphlpapi  
76e80000 76e8e000 rtutils  
76e90000 76ea2000 rasman  
76eb0000 76edf000 TAPI32  
76ee0000 76f1c000 RASAPI32  
76f20000 76f47000 DNSAPI  
76f60000 76f8c000 WLDAP32  
76fc0000 76fc6000 rasadhlp  
76fd0000 7704f000 CLBCATQ  
77050000 77115000 COMRes  
77120000 771ac000 OLEAUT32  
771b0000 77256000 WININET  
773d0000 774d3000 comctl32  
774e0000 7761d000 ole32  
77920000 77a13000 SETUPAPI  
77a20000 77a74000 cscui  
77a80000 77b14000 CRYPT32  
77b20000 77b32000 MSASN1  
77b40000 77b62000 appHelp  
77bd0000 77bd7000 midimap  
77be0000 77bf5000 MSACM32\_77be0000  
77c00000 77c08000 VERSION  
77c10000 77c68000 msvcrt  
77c70000 77c93000 msv1\_0  
77d40000 77dd0000 USER32  
77dd0000 77e6b000 ADVAPI32  
77e70000 77f01000 RPCRT4  
77f10000 77f57000 GDI32  
77f60000 77fd6000 SHLWAPI  
77fe0000 77ff1000 Secur32  
7c800000 7c8f4000 kernel32  
7c900000 7c9b0000 ntdll

```
7c9c0000 7d1d5000 SHELL32
7dc30000 7df20000 mshtml
7e1e0000 7e280000 urlmon
7e290000 7e3ff000 SHDOCVW
```

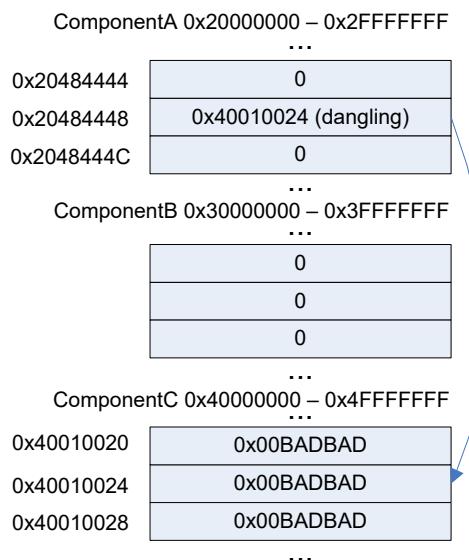
Installing or upgrading software can change the distribution of loaded DLLs and their addresses. This also happens when we install some monitoring software which usually injects their DLLs into every process. As a result, some DLLs might be relocated, or even the new ones appear loaded. And this might influence 3rd-party program behavior therefore exposing its hidden bugs being dormant when executing the process in the old environment. I call this pattern **Changed Environment**.

Let's look at some hypothetical example. Suppose our program has the following code fragment

```
if (*p)
{
// do something useful
}
```

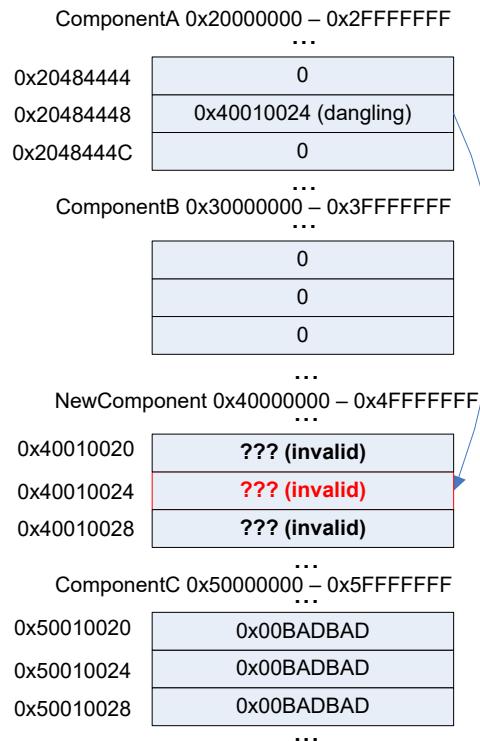
Suppose the pointer p is invalid, dangling, its value has been overwritten, and this happened because of some bug. Being invalid that pointer can point to a valid memory location nevertheless and the value it points to most likely is non-zero. Therefore, the body of the "if" statement will be executed. Suppose it always happens when we run the program and every time we execute it the value of the pointer happens to be the same.

Here is the picture illustrating the point:



The pointer value 0x40010024 due to some reason always points to the value 0x00BADBAD. Although in the correct program the pointer itself should have had a completely different value and pointed to 0x1, for example, we see that dereferencing its current invalid value doesn't crash the process.

After installing the new software, *NewComponent* DLL is loaded at the address range previously occupied by *ComponentC*:



Now the address 0x40010024 happens to be completely invalid, and we have an access violation and the crash dump.

## Comments

Sometimes changes in physical memory size may also affect process behavior. Other examples include an application running under a user mode debugger which effects a different type of runtime heap used.

## Clone Dump

With the possibility of process cloning (reflection) starting from Windows 7 it is possible to get memory snapshots (**Clone Dump**) from a process clone (similar to *fork* API in Unix). The procdump<sup>13</sup> tool has -r switch for that purpose. We checked this with x64 Windows 7 *notepad.exe*. We got two memory dumps: one is a clone with this stack trace:

```
Loading Dump File [C:\DebuggingTV\Procdump\notepad.exe_151117_000755.dbgcfg.dmp]
User Mini Dump File with Full Memory: Only application data is available

Comment: '
*** procdump -ma -r Notepad.exe
*** Manual dump'

0:000> ~*k

. 0 Id: 25ec.147c Suspend: 1 Teb: 000007ff`ffffdb000 Unfrozen
# Child-SP RetAddr Call Site
00 00000000`02c8fd38 00000000`773aae7 ntdll!NtSuspendThread+0xa
01 00000000`02c8fd40 00000000`77165a4d ntdll!RtlpProcessReflectionStartup+0x2e7
02 00000000`02c8fe30 00000000`7729b831 kernel32!BaseThreadInitThunk+0xd
03 00000000`02c8fe60 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

The process memory has all address space of the original process including module list and heap structure:

```
0:000> lmn
start end module name
00000000`77050000 00000000`7714a000 user32 user32.dll
00000000`77150000 00000000`77270000 kernel32 kernel32.dll
00000000`77270000 00000000`77419000 ntdll ntdll.dll
00000000`ff030000 00000000`ff065000 notepad notepad.exe
000007fe`f57d0000 000007fe`f5841000 winspool winspool.drv
000007fe`fb730000 000007fe`fb786000 uxtheme uxtheme.dll
000007fe`fb910000 000007fe`fbb04000 comctl32 comctl32.dll
000007fe`fbf00000 000007fe`fbf0c000 version version.dll
000007fe`fce80000 000007fe`fcebf000 CRYPTBASE CRYPTBASE.dll
000007fe`fd310000 000007fe`fd37c000 KERNELBASE KERNELBASE.dll
000007fe`fd3d0000 000007fe`fd499000 usp10 usp10.dll
000007fe`fd4a0000 000007fe`fd511000 shlwapi shlwapi.dll
000007fe`fd520000 000007fe`fd64d000 rpcrt4 rpcrt4.dll
000007fe`fd650000 000007fe`fd66f000 sechost sechost.dll
000007fe`fd680000 000007fe`fd6ae000 imm32 imm32.dll
000007fe`fd6b0000 000007fe`fd78b000 advapi32 advapi32.dll
000007fe`fd810000 000007fe`fd8a7000 comdlg32 comdlg32.dll
000007fe`fdd60000 000007fe`fddff000 msvcrt msvcrt.dll
000007fe`fdfe0000 000007fe`fed69000 shell32 shell32.dll
000007fe`fed70000 000007fe`fedd7000 gdi32 gdi32.dll
```

<sup>13</sup> <http://technet.microsoft.com/en-us/sysinternals/dd996900.aspx>

```
000007fe`ff0b0000 000007fe`ff1b9000 msctf    msctf.dll
000007fe`ff1c0000 000007fe`ff1ce000 lpk      lpk.dll
000007fe`ff1d0000 000007fe`ff2a7000 oleaut32 oleaut32.dll
000007fe`ff2b0000 000007fe`ff349000 clbcatq clbcatq.dll
000007fe`ff350000 000007fe`ff553000 ole32    ole32.dll
```

```
0:000> !address -summary
```

Mapping file section regions...  
 Mapping module regions...  
 Mapping PEB regions...  
 Mapping TEB and stack regions...  
 Mapping heap regions...  
 Mapping page heap regions...  
 Mapping other regions...  
 Mapping stack trace database regions...  
 Mapping activation context regions...

	RgnCount	Total Size	%ofBusy	%ofTotal
Free	48	7ff`faa06000 ( 8.000 TB)	100.00%	
Image	129	0`01e79000 ( 30.473 MB)	35.47%	0.00%
<unknown>	21	0`01d10000 ( 29.063 MB)	33.83%	0.00%
Other	9	0`016be000 ( 22.742 MB)	26.47%	0.00%
Heap	26	0`00320000 ( 3.125 MB)	3.64%	0.00%
Stack	3	0`00080000 ( 512.000 kB)	0.58%	0.00%
TEB	1	0`00002000 ( 8.000 kB)	0.01%	0.00%
PEB	1	0`00001000 ( 4.000 kB)	0.00%	0.00%

	RgnCount	Total Size	%ofBusy	%ofTotal
MEM_IMAGE	130	0`01e7a000 ( 30.477 MB)	35.47%	0.00%
MEM_PRIVATE	47	0`01449000 ( 20.285 MB)	23.61%	0.00%
MEM_MAPPED	11	0`00c97000 ( 12.590 MB)	14.65%	0.00%

	RgnCount	Total Size	%ofBusy	%ofTotal
MEM_FREE	50	7ff`fc096000 ( 8.000 TB)	100.00%	
MEM_COMMIT	176	0`02c10000 ( 44.063 MB)	51.29%	0.00%
MEM_RESERVE	12	0`0134a000 ( 19.289 MB)	22.45%	0.00%

	RgnCount	Total Size	%ofBusy	%ofTotal
PAGE_READONLY	83	0`01b46000 ( 27.273 MB)	31.75%	0.00%
PAGE_EXECUTE_READ	25	0`00f6d000 ( 15.426 MB)	17.95%	0.00%
PAGE_WRITECOPY	48	0`00126000 ( 1.148 MB)	1.34%	0.00%
PAGE_READWRITE	18	0`00032000 ( 200.000 kB)	0.23%	0.00%
PAGE_READWRITE PAGE_GUARD	2	0`00005000 ( 20.000 kB)	0.02%	0.00%

	Base Address	Region Size
Free	0`ff065000	7fd`f676b000 ( 7.992 TB)
Image	7fe`fe4cb000	0`0089e000 ( 8.617 MB)
<unknown>	0`7f0e0000	0`00f00000 ( 15.000 MB)
Other	0`00610000	0`01590000 ( 21.563 MB)
Heap	0`003b8000	0`000c8000 ( 800.000 kB)
Stack	0`02c10000	0`0006c000 ( 432.000 kB)
TEB	7ff`ffffdb000	0`00002000 ( 8.000 kB)
PEB	7ff`ffffdf000	0`00001000 ( 4.000 kB)

```
0:000> !heap -s
*****
NT HEAP STATS BELOW
*****
LFH Key : 0x000000381021167d
Termination on corruption : ENABLED
Heap Flags Reserv Commit Virt Free List UCR Virt Lock Fast
(k) (k) (k) (k) length blocks cont. heap
-----
000000000280000 00000002 1024 412 1024 14 4 1 0 0 LFH
000000000250000 00001002 1088 256 1088 5 2 2 0 0 LFH
0000000001c0000 00001002 64 8 64 3 1 1 0 0
0000000001e60000 00001002 512 120 512 49 3 1 0 0
0000000001dc0000 00001002 512 8 512 2 1 1 0 0
-----
```

The other dump saved is a minidump from which we can get thread information for **Execution Residue** (page 371, raw stack data) and reconstruct stack traces in **Clone Dump**:

```
Loading Dump File [C:\DebuggingTV\Procdump\notepad.exe_151117_000755.dmp]
Comment: '
*** procdump -ma -r Notepad.exe
*** Manual dump'
User Mini Dump File: Only registers, stack and portions of memory are available

0:000> ~
. 0 Id: 87c.27f4 Suspend: 0 Teb: 000007ff`ffffdd000 Unfrozen

0:000> k
# Child-SP RetAddr Call Site
00 00000000`0016fac8 00000000`77069e9e 0x77069e6a
01 00000000`0016fad0 00000000`00000000 0x77069e9e

0:000> r
rax=0000000000000000 rbx=00000000016fb40 rcx=000000000280000
rdx=0000000000000000 rsi=0000000000000001 rdi=0000000000000000
rip=0000000077069e6a rsp=000000000016fac8 rbp=00000000ff030000
r8=000000000016f8e8 r9=00000000000a0cdc r10=0000000000000000
r11=0000000000000000 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0 nv up ei pl nz na po nc
cs=0033 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000246
00000000`77069e6a c3 ret
```

Now we can see the original stack trace in **Clone Dump**:

```
0:000> k =000000000016fac8 ff
# Child-SP RetAddr Call Site
00 00000000`0016fac8 00000000`77069e9e ntdll!NtSuspendThread+0xa
01 00000000`0016fad0 00000000`ff031064 user32!GetMessageW+0x34
02 00000000`0016fb00 00000000`ff03133c notepad!WinMain+0x182
03 00000000`0016fb80 00000000`77165a4d notepad!DisplayNonGenuineDlgWorker+0x2da
```

```
04 00000000`0016fc40 00000000`7729b831 kernel32!BaseThreadInitThunk+0xd
05 00000000`0016fc70 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

Since we know TEB address from the minidump we can get stack region boundaries in **Clone Dump**:

```
0:000> dt _NT_TIB 000007fffffd000
ntdll!_NT_TIB
+0x000 ExceptionList      : (null)
+0x008 StackBase          : 0x00000000`00170000 Void
+0x010 StackLimit         : 0x00000000`0015b000 Void
+0x018 SubSystemTib       : (null)
+0x020 FiberData          : 0x00000000`00001e00 Void
+0x020 Version            : 0x1e00
+0x028 ArbitraryUserPointer : (null)
+0x030 Self               : 0x000007ff`fffffd000 _NT_TIB
```

Now we can check **Execution Residue** (for example, for signs of **Hidden Exceptions**, page 457):

```
0:000> dps 0x00000000`0015b000 0x00000000`00170000
[...]
```

## Cloud Environment

This pattern is specific to cloud platforms. It covers both development (emulator, if it exists) and real (staging and deployment) environments and is best diagnosed by looking at specific infrastructure modules:

```
0:016> lm m Wa*
start end module name
00000000`00b00000 00000000`00b0c000 WaWorkerHost
00000000`74fb0000 00000000`74fdb000 WaRuntimeProxy

0:016> lm m *Azure*
start end module name
00000000`57cd0000 00000000`57d26000 Microsoft_WindowsAzure_StorageClient
00000000`58820000 00000000`5886c000 Microsoft_WindowsAzure_Diagnostics
00000000`5c750000 00000000`5c764000 Microsoft_WindowsAzure_ServiceRuntime
```

Development platform can be distinguished by looking at versions of system modules such as *ntdll*:

```
0:016> lmv m ntdll
start           end           module name
00000000`76de0000 00000000`76f5f000 ntdll
Loaded symbol image file:      ntdll.dll
Image path:                 D:\Windows\System32\ntdll.dll
Image name:                  ntdll.dll
Timestamp:                  Fri May 13 21:45:21 2011 (4DCD9861)
CheckSum:                   00188814
ImageSize:                  0017F000
File version:                6.0.6002.18446
Product version:             6.0.6002.18446
File flags:                  0 (Mask 3F)
File OS:                     40004 NT Win32
File type:                   2.0 Dll
File date:                  00000000.00000000
Translations:                0409.04b0
CompanyName:                Microsoft Corporation
ProductName:                 Microsoft® Windows® Operating System
InternalName:                ntdll.dll
OriginalFilename:             ntdll.dll
ProductVersion:              6.0.6002.18446
FileVersion:                 6.0.6002.18446 (rd_os_v1.110513-1321)
FileDescription:              NT Layer DLL
LegalCopyright:               © Microsoft Corporation. All rights reserved.
```

```
0:016> lmv m ntdll
start           end             module name
00000000`775a0000 00000000`7774b000 ntdll
Loaded symbol image file:      ntdll.dll
Image path:                  C:\Windows\System32\ntdll.dll
Image name:                  ntdll.dll
Timestamp:                   Tue Jul 14 02:32:27 2009 (4A5BE02B)
CheckSum:                    001B1CB5
ImageSize:                   001AB000
File version:                6.1.7600.16385
Product version:              6.1.7600.16385
File flags:                  0 (Mask 3F)
File OS:                     40004 NT Win32
File type:                   2.0 Dll
File date:                  00000000.00000000
Translations:                0409.04b0
CompanyName:                 Microsoft Corporation
ProductName:                  Microsoft® Windows® Operating System
InternalName:                ntdll.dll
OriginalFilename:             ntdll.dll
ProductVersion:               6.1.7600.16385
FileVersion:                  6.1.7600.16385 (win7_rtm.090713-1255)
FileDescription:              NT Layer DLL
LegalCopyright:              © Microsoft Corporation. All rights reserved.
```

## CLR Thread

In cases where we don't see **Managed Code Exceptions** (page 617) or **Managed Stack Traces** (page 624) by default, we need to identify **CLR Threads** in order to try various SOS commands and start digging into a managed realm. These threads are easily distinguished by *mscorwks* module on their stack traces (don't forget to list full stack traces<sup>14</sup>):

```
0:000> ~*kL 100

. 0 Id: 658.4ec Suspend: 1 Teb: 7ffdf000 Unfrozen
ChildEBP RetAddr
0007fc98 7c827d19 ntdll!KiFastSystemCallRet
0007fc9c 77e6202c ntdll!NtWaitForMultipleObjects+0xc
0007fd44 7739bbd1 kernel32!WaitForMultipleObjectsEx+0x11a
0007fda0 6c296601 user32!RealMsgWaitForMultipleObjectsEx+0x141
0007fdc0 6c29684b duser!CoreSC::Wait+0x3a
0007fdf4 6c29693d duser!CoreSC::xwProcessNL+0xab
0007fe14 773b0c02 duser!MphProcessMessage+0x2e
0007fe5c 7c828556 user32!__ClientGetMessageMPH+0x30
0007fe84 7739c811 ntdll!KiUserCallbackDispatcher+0x2e
0007fea4 7f072fd6 user32!NtUserGetMessage+0xc
0007fec0 010080ef mfc42u!CWinThread::PumpMessage+0x16
0007fef0 7f072dda mmc!CAMCAApp::PumpMessage+0x37
0007ff08 7f044d5b mfc42u!CWinThread::Run+0x4a
0007ff1c 01034e19 mfc42u!AfxWinMain+0x7b
0007ffc0 77e6f23b mmc!wWinMainCRTStartup+0x19d
0007fff0 00000000 kernel32!BaseProcessStart+0x23

1 Id: 658.82c Suspend: 1 Teb: 7ffdde000 Unfrozen
ChildEBP RetAddr
003afea0 7c827d19 ntdll!KiFastSystemCallRet
003afea4 7c80e5bb ntdll!NtWaitForMultipleObjects+0xc
003aff48 7c80e4a2 ntdll!EtwpWaitForMultipleObjectsEx+0xf7
003affb8 77e6482f ntdll!EtwpEventPump+0x27f
003affec 00000000 kernel32!BaseThreadStart+0x34

2 Id: 658.648 Suspend: 1 Teb: 7ffdd000 Unfrozen
ChildEBP RetAddr
00f3fe18 7c827859 ntdll!KiFastSystemCallRet
00f3fe1c 77c885ac ntdll!NtReplyWaitReceivePortEx+0xc
00f3ff84 77c88792 rpcrt4!LRPC_ADDRESS::ReceiveLotsaCalls+0x198
00f3ff8c 77c8872d rpcrt4!RecvLotsaCallsWrapper+0xd
00f3ffac 77c7b110 rpcrt4!BaseCachedThreadRoutine+0x9d
00f3ffb8 77e6482f rpcrt4!ThreadStartRoutine+0x1b
00f3ffec 00000000 kernel32!BaseThreadStart+0x34
```

<sup>14</sup> Common Mistakes, Memory Dump Analysis Anthology, Volume 2, page 39

3 Id: 658.640 Suspend: 1 Teb: 7ffdb000 Unfrozen  
ChildEBP RetAddr  
0156fdb4 7c827d19 ntdll!KiFastSystemCallRet  
0156fdb8 77e6202c ntdll!NtWaitForMultipleObjects+0xc  
0156fe60 7739bbd1 kernel32!WaitForMultipleObjectsEx+0x11a  
0156feb0 6c296601 user32!RealMsgWaitForMultipleObjectsEx+0x141  
0156fedc 6c29684b duser!CoreSC::Wait+0x3a  
0156ff10 6c28f9e6 duser!CoreSC::xwProcessNL+0xab  
0156ff30 6c28bce1 duser!GetMessageExA+0x44  
0156ff84 77bcb530 duser!ResourceManager::SharedThreadProc+0xb6  
0156ffb8 77e6482f msrvct!\_endthreadex+0xa3  
0156ffec 00000000 kernel32!BaseThreadStart+0x34

4 Id: 658.e74 Suspend: 1 Teb: 7ffd000 Unfrozen  
ChildEBP RetAddr  
01d1fe30 7c827d19 ntdll!KiFastSystemCallRet  
01d1fe34 77e6202c ntdll!NtWaitForMultipleObjects+0xc  
01d1fedc 77e62fbe kernel32!WaitForMultipleObjectsEx+0x11a  
01d1fef8 79f02541 kernel32!WaitForMultipleObjects+0x18  
01d1ff58 79f0249e msrvwks!DebuggerRCThread::MainLoop+0xe9  
01d1ff88 79f023c5 msrvwks!DebuggerRCThread::ThreadProc+0xe5  
01d1ffb8 77e6482f msrvwks!DebuggerRCThread::ThreadProcStatic+0x9c  
01d1ffec 00000000 kernel32!BaseThreadStart+0x34

5 Id: 658.4d4 Suspend: 1 Teb: 7ffd8000 Unfrozen  
ChildEBP RetAddr  
03dffcc4 7c827d19 ntdll!KiFastSystemCallRet  
03dffcc8 77e6202c ntdll!NtWaitForMultipleObjects+0xc  
03dfffd0 77e62fbe kernel32!WaitForMultipleObjectsEx+0x11a  
03dfffd8c 79f92bcb kernel32!WaitForMultipleObjects+0x18  
03dffdac 79f97028 msrvwks!WKS::WaitForFinalizerEvent+0x77  
03dffdc0 79e9845f msrvwks!WKS::GCHeap::FinalizerThreadWorker+0x49  
03dffdd4 79e983fb msrvwks!Thread::DoADCCallBack+0x32a  
03dffe68 79e98321 msrvwks!Thread::ShouldChangeAbortToUnload+0xe3  
03dffa4 79eef6cc msrvwks!Thread::ShouldChangeAbortToUnload+0x30a  
03dffecc 79eef6dd msrvwks!ManagedThreadBase\_NoADTransition+0x32  
03dffffdc 79f3c63c msrvwks!ManagedThreadBase::FinalizerBase+0xd  
03dfff14 79f92015 msrvwks!WKS::GCHeap::FinalizerThreadStart+0xbb  
03dfff88 77e6482f msrvwks!Thread::intermediateThreadProc+0x49  
03dffffec 00000000 kernel32!BaseThreadStart+0x34

6 Id: 658.f54 Suspend: 1 Teb: 7ffd6000 Unfrozen  
ChildEBP RetAddr  
040afec4 7c826f69 ntdll!KiFastSystemCallRet  
040afec8 77e41ed5 ntdll!NtDelayExecution+0xc  
040aff30 79fd8a41 kernel32!SleepEx+0x68  
040affac 79fd88ef msrvwks!ThreadpoolMgr::TimerThreadFire+0x6d  
040affb8 77e6482f msrvwks!ThreadpoolMgr::TimerThreadStart+0x57  
040affec 00000000 kernel32!BaseThreadStart+0x34

```
7 Id: 658.988 Suspend: 1 Teb: 7ffd5000 Unfrozen
ChildEBP RetAddr
0410fc2c 7c827d29 ntdll!KiFastSystemCallRet
0410fc30 77e61d1e ntdll!ZwWaitForSingleObject+0xc
0410fcfa0 79e8c5f9 kernel32!WaitForSingleObjectEx+0xac
0410fce4 79e8c52f mscorewks!PEImage::LoadImage+0x1af
0410fd34 79e8c54e mscorewks!CLREvent::WaitEx+0x117
0410fd48 79ee3f35 mscorewks!CLREvent::Wait+0x17
0410fe14 79f92015 mscorewks!AppDomain::ADUnloadThreadStart+0x308
0410ffb8 77e6482f mscorewks!Thread::intermediateThreadProc+0x49
0410ffec 00000000 kernel32!BaseThreadStart+0x34

8 Id: 658.e0 Suspend: 1 Teb: 7ff4f000 Unfrozen
ChildEBP RetAddr
0422fcce 7c827d19 ntdll!KiFastSystemCallRet
0422fcf0 7c83c7be ntdll!NtWaitForMultipleObjects+0xc
0422ffb8 77e6482f ntdll!RtlpWaitThread+0x161
0422ffec 00000000 kernel32!BaseThreadStart+0x34

9 Id: 658.db4 Suspend: 1 Teb: 7ff4e000 Unfrozen
ChildEBP RetAddr
0447fec0 7c827d19 ntdll!KiFastSystemCallRet
0447fec4 77e6202c ntdll!NtWaitForMultipleObjects+0xc
0447ff6c 77e62fbe kernel32!WaitForMultipleObjectsEx+0x11a
0447ff88 76929e35 kernel32!WaitForMultipleObjects+0x18
0447ffb8 77e6482f userenv!NotificationThread+0x5f
0447ffec 00000000 kernel32!BaseThreadStart+0x34

10 Id: 658.e7c Suspend: 1 Teb: 7ff4c000 Unfrozen
ChildEBP RetAddr
0550ff7c 7c8277f9 ntdll!KiFastSystemCallRet
0550ff80 71b25914 ntdll!NtRemoveIoCompletion+0xc
0550ffb8 77e6482f msowsock!SockAsyncThread+0x69
0550ffec 00000000 kernel32!BaseThreadStart+0x34

[...]
```

## Comments

---

Silverlight applications use *coreclr*<sup>15</sup>.

In CLR 4.0 the module has changed to just *clr*:

```
0:000> lmv m clr
start          end            module name
000007fe`eadc0000 000007fe`eb725000  clr      (pdb symbols)
  Loaded symbol image file: clr.dll
  Image path: C:\Windows\Microsoft.NET\Framework64\
v4.0.30319\clr.dll
  Image name: clr.dll
  Timestamp:     Thu Mar 18 12:39:07 2010 (4BA21EEB)
  CheckSum:      00959DBD
  ImageSize:     00965000
  File version:  4.0.30319.1
  Product version: 4.0.30319.1
  File flags:    8 (Mask 3F) Private
  File OS:       4 Unknown Win32
  File type:     2.0 Dll
  File date:    00000000.00000000
  Translations: 0409.04b0
  CompanyName:  Microsoft Corporation
  ProductName:  Microsoft® .NET Framework
  InternalName: clr.dll
  OriginalFilename: clr.dll
  ProductVersion: 4.0.30319.1
  FileVersion:   4.0.30319.1 (RTMRel.030319-0100)
  PrivateBuild:  DDBLD431
  FileDescription: Microsoft .NET Runtime Common Language Runtime - WorkStation
  LegalCopyright: © Microsoft Corporation. All rights reserved.
  Comments:     Flavor=Retail

0:000> ~16kc
Call Site
ntdll!NtWaitForSingleObject
KERNELBASE!WaitForSingleObjectEx
clr!CLREvent::WaitEx
clr!CLREvent::WaitEx
clr!CLREvent::WaitEx
clr!Thread::WaitSuspendEventsHelper
clr!Thread::WaitSuspendEvents
clr! ?? ::FNODOBFM::`string'
clr!Thread::RareDisablePreemptiveGC
clr!GCHolderBase<1,0,0,1>::EnterInternal
clr!AddTimerCallbackEx
clr!ThreadpoolMgr::AsyncTimerCallbackCompletion
clr!UnManagedPerAppDomainTPCount::DispatchWorkItem
clr!ThreadpoolMgr::NewWorkerThreadStart
clr!ThreadpoolMgr::WorkerThreadStart
clr!Thread::intermediateThreadProc
kernel32!BaseThreadInitThunk
ntdll!RtlUserThreadStart
```

---

<sup>15</sup> <http://blogs.msdn.com/tess/archive/2008/08/21/debugging-silverlight-applications-with-windbg-and-sos-dll.aspx>

## Coincidental Error Code

Address space-wide search for errors and status codes<sup>16</sup> may show **Coincidental Error Codes**:

```
0:000> !heap -x -v c0000005
Search VM for address range c0000005 - c0000005 : 028690b8 (c0000005), [...]

0:000> dd 028690b8 11
028690b8 c0000005
```

In such cases we need to check whether the addresses belong to volatile regions such as stack because it is possible to have such values as legitimate code and image data:

```
0:000> !address 028690b8
Usage: Image
Allocation Base: 02700000
Base Address: 02869000
End Address: 02874000
Region Size: 0000b000
Type: 01000000 MEM_IMAGE
State: 00001000 MEM_COMMIT
Protect: 00000002 PAGE_READONLY
More info: lm v m ModuleA
More info: !lmi ModuleA
More info: ln 0x28690b8

0:000> u 028690b8
ModuleA!ComputeB:
028690b8 050000c000 add eax,0C00000h
[...]
```

Another example (x64):

```
0:000> !heap -x -v c0000005
Search VM for address range 00000000c0000005 - 00000000c0000005 : 7feff63ab60 (c0000005),

0:000> !address 7feff63ab60
Usage: Image
Allocation Base: 000007fe`ff460000
Base Address: 000007fe`ff635000
End Address: 000007fe`ff63c000
Region Size: 00000000`00007000
Type: 01000000 MEM_IMAGE
State: 00001000 MEM_COMMIT
Protect: 00000004 PAGE_READWRITE
More info: lm v m ole32
```

---

<sup>16</sup> WinDbg Shortcuts, Memory Dump Analysis Anthology, Volume 7, page 29

```
More info: !lmi ole32
More info: ln 0x7feff63ab60
```

```
0:000> dp 7feff63ab60
000007fe`ff63ab60 00000000`c0000005 c0000194`00000001
000007fe`ff63ab70 00000001`00000000 00000000`c00000aa
000007fe`ff63ab80 80000002`00000001 00000001`00000000
000007fe`ff63ab90 00000000`c0000096 c000001d`00000001
000007fe`ff63aba0 00000001`00000000 00000000`80000003
000007fe`ff63abb0 c00000fd`00000001 00000001`00000000
000007fe`ff63abc0 00000000`c0000235 c0000006`00000001
000007fe`ff63abd0 00000001`00000000 00000000`c0000420
```

In the latter case, the data structure suggests a table of errors:

```
0:000> ln 7feff63ab60
(000007fe`ff63ab60) ole32!gReportedExceptions
```

## Coincidental Frames

For certain stack traces, we should always be aware of coincidental frames similar to **Coincidental Symbolic Information** pattern (page 137) for raw stack data. Such frames can lead to a wrong analysis conclusion. Consider this stack trace fragment from a kernel memory dump:

```
0: kd> kL 100
ChildEBP RetAddr
9c5b6550 8082d9a4 nt!KeBugCheckEx+0x1b
9c5b6914 8088befa nt!KiDispatchException+0x3a2
9c5b697c 8088beae nt!CommonDispatchException+0x4a
9c5b699c 80a6056d nt!KiExceptionExit+0x186
9c5b69a0 80893ae2 hal!KeReleaseQueuedSpinLock+0x2d
9c5b6a08 b20c3de5 nt!MiFreePoolPages+0x7dc
WARNING: Stack unwind information not available. Following frames may be wrong.
9c5b6a48 b20c4107 DriverA+0x17de5
[...]
```

The frame with *MiFreePoolPages* symbol might suggest some sort of a pool corruption. We can even double check return addresses and see the valid common sense assembly language code:

```
0: kd> ub 8088beae
nt!KiExceptionExit+0x167:
8088be8f 33c9          xor    ecx,ecx
8088be91 e81a000000    call   nt!CommonDispatchException (8088beb0)
8088be96 33d2          xor    edx,edx
8088be98 b901000000    mov    ecx,1
8088be9d e80e000000    call   nt!CommonDispatchException (8088beb0)
8088bea2 33d2          xor    edx,edx
8088bea4 b902000000    mov    ecx,2
8088bea9 e802000000    call   nt!CommonDispatchException (8088beb0)

0: kd> ub 80a6056d
hal!KeReleaseQueuedSpinLock+0x1b:
80a6055b 7511          jne    hal!KeReleaseQueuedSpinLock+0x2e (80a6056e)
80a6055d 50             push   eax
80a6055e f00fb119     lock   cmpxchg dword ptr [ecx],ebx
80a60562 58             pop    eax
80a60563 7512          jne    hal!KeReleaseQueuedSpinLock+0x37 (80a60577)
80a60565 5b             pop    ebx
80a60566 8aca           mov    cl,d1
80a60568 e8871e0000    call   hal!KfLowerIrql (80a623f4)

0: kd> ub 80893ae2
nt!MiFreePoolPages+0x7c3:
80893ac9 761c          jbe    nt!MiFreePoolPages+0x7e1 (80893ae7)
80893acb ff75f8        push   dword ptr [ebp-8]
80893ace ff7508        push   dword ptr [ebp+8]
80893ad1 e87ea1fcff    call   nt!MiFreeNonPagedPool (8085dc54)
80893ad6 8a55ff        mov    dl,byte ptr [ebp-1]
80893ad9 6a0f          push   0Fh
```

```

80893adb 59          pop   ecx
80893adc ff1524118080  call  dword ptr [nt!_imp_KeReleaseQueuedSpinLock (80801124)]  

0: kd> ub b20c3de5
DriverA+0x17dcf:  

b20c3dcf 51          push  ecx
b20c3dd0 ff5010  call  dword ptr [eax+10h]
b20c3dd3 eb10          jmp   DriverA+0x17de5 (b20c3de5)
b20c3dd5 8b5508  mov    edx,dword ptr [ebp+8]
b20c3dd8 52          push  edx
b20c3dd9 8d86a0000000 lea    eax,[esi+0A0h]
b20c3ddf 50          push  eax
b20c3de0 e8ebf1ffff  call  DriverA+0x16fd0 (b20c2fd0)

```

However, if we try to reconstruct the stack trace manually<sup>17</sup> we would naturally skip these three frames (shown in underlined bold):

```

9c5b6550 8082d9a4 nt!KeBugCheckEx+0x1b
9c5b6914 8088befa nt!KiDispatchException+0x3a2
9c5b697c 8088beae nt!CommonDispatchException+0x4a
9c5b699c 80a6056d nt!KiExceptionExit+0x186
9c5b69a0 80893ae2 hal!KeReleaseQueuedSpinLock+0x2d
9c5b6a08 b20c3de5 nt!MiFreePoolPages+0x7dc
9c5b6a48 b20c4107 DriverA+0x17de5
[...]  

0: kd> !thread
THREAD 8f277020 Cid 081c.7298 Teb: 7fff11000 Win32Thread: 00000000 RUNNING on processor 0
IRP List:
  8e234b60: (0006,0094) Flags: 00000000 Mdl: 00000000
Not impersonating
DeviceMap           e1002880
Owning Process     8fc78b80      Image:       ProcessA.exe
Attached Process   N/A          Image:       N/A
Wait Start TickCount 49046879    Ticks: 0
Context Switch Count 10
UserTime            00:00:00.000
KernelTime          00:00:00.000
Win32 Start Address D11A!ThreadA (0x7654dc90)
Start Address kernel32!BaseThreadStartThunk (0x77e617dc)
Stack Init 9c5b7000 Current 9c5b6c50 Base 9c5b7000 Limit 9c5b4000 Call 0
Priority 10 BasePriority 10 PriorityDecrement 0
ChildEBP RetAddr  Args to Child
[...]  

0: kd> dds 9c5b4000 9c5b7000
9c5b4000 00000000
9c5b4004 00000000
9c5b4008 00000000

```

<sup>17</sup> Manual Stack Trace Reconstruction, Memory Dump Analysis Anthology, Volume 1, page 157

```
[...]
9c5b6290  ffdfff13c
9c5b6294  9c5b6550
9c5b6298  80827e01 nt!KeBugCheckEx+0x1b
9c5b629c  00000008
9c5b62a0  00000286
[...]
9c5b654c  00000000
9c5b6550  9c5b6914
9c5b6554  8082d9a4 nt!KiDispatchException+0x3a2
9c5b6558  0000008e
9c5b655c  c0000005
[...]
9c5b6910  ffffffff
9c5b6914  9c5b6984
9c5b6918  8088befa nt!CommonDispatchException+0x4a
9c5b691c  9c5b6930
9c5b6920  00000000
[...]
9c5b6980  8088beae nt!KiExceptionExit+0x186
9c5b6984  9c5b6a08
9c5b6988  b20c3032 DriverA+0x17032
9c5b698c  badb0d00
9c5b6990  00000006
9c5b6994  8dc11cec
9c5b6998  808b6900 nt!KiTimerTableLock+0x3c0
9c5b699c  9c5b69d4
9c5b69a0  80a6056d hal!KeReleaseQueuedSpinLock+0x2d
9c5b69a4  80893ae2 nt!MiFreePoolPages+0x7dc
9c5b69a8  808b0b40 nt!NonPagedPoolDescriptor
9c5b69ac  03151fd0
9c5b69b0  00000000
9c5b69b4  00000000
[...]
9c5b6a04  8f47123b
9c5b6a08  9c5b6a48
9c5b6a0c  b20c3de5 DriverA+0x17de5
9c5b6a10  8e3640a0
9c5b6a14  8f4710d0
[...]
9c5b6a44  00000000
9c5b6a48  9c5b6a80
9c5b6a4c  b20c4107 DriverA+0x18107
9c5b6a50  8f4710d0
9c5b6a54  9c5b6a6c
[...]
```

If we try to find a pointer to the exception record we get this crash address:

```
0: kd> .exr 9c5b6930
ExceptionAddress: b20c3032 (DriverA+0x00017032)
  ExceptionCode: c0000005 (Access violation)
  ExceptionFlags: 00000000
NumberParameters: 2
  Parameter[0]: 00000000
  Parameter[1]: 00000157
Attempt to read from address 00000157
```

If we disassemble it we see **Inline Function Optimization** pattern (page 514) for string or memory copy, perhaps *wcsncpy* function:

```
0: kd> u b20c3032
DriverA+0x17032:
b20c3032 f3a5      rep movs dword ptr es:[edi],dword ptr [esi]
b20c3034 8bcb      mov     ecx,ebx
b20c3036 83e103    and     ecx,3
b20c3039 f3a4      rep movs byte ptr es:[edi],byte ptr [esi]
b20c303b 8b750c    mov     esi,dword ptr [ebp+0Ch]
b20c303e 0fb7ca    movzx   ecx,dx
b20c3041 894e14    mov     dword ptr [esi+14h],ecx
b20c3044 8b700c    mov     esi,dword ptr [eax+0Ch]
```

So the problem happened in *DriverA* code, not in functions *MiFreePoolPages* or *KeReleaseQueuedSpinLock*.

## Comments

This often happens when we have some “boundary” such as in **Exception Stack Trace** pattern (page 363).

## Coincidental Symbolic Information

### Linux

This is a Linux variant of **Coincidental Symbolic Information** pattern previously described for Mac OS X (page 135) and Windows (page 137) platforms. The idea is the same: to disassemble the address to see if the preceding instruction is a call. If it is indeed then most likely the symbolic address is a return address from past **Execution Residue** (page 365):

```
(gdb) x/i 0x4005e6
0x4005e6 <_Z6work_3v+9>: pop    %rbp

(gdb) disassemble 0x4005e6
Dump of assembler code for function _Z6work_3v:
0x00000000004005dd <+0>: push    %rbp
0x00000000004005de <+1>: mov     %rsp,%rbp
0x00000000004005e1 <+4>: callq   0x4005d2 <_Z6work_4v>
0x00000000004005e6 <+9>: pop     %rbp
0x00000000004005e7 <+10>: retq
End of assembler dump.

(gdb) x/4i 0x49c740-4
0x49c73c: add    %al,(%rax)
0x49c73e: add    %al,(%rax)
0x49c740 <default_attr>: add    %al,(%rax)
0x49c742 <default_attr+2>: add    %al,(%rax)
```

## Mac OS X

This is a Mac OS X / GDB counterpart to **Coincidental Symbolic Information** pattern previously described for Windows platforms. The idea is the same: to disassemble the address to see if the preceding instruction is a call. If it is indeed then most likely the symbolic address is a return address from past **Execution Residue** (page 365):

```
(gdb) x $rsp
0x7fff6a162a38: 0x8fab9a9c

(gdb) x/1000a 0x7fff6a162000
[...]
0x7fff6a162960: 0x7fff6a162980 0x7fff6a167922
0x7fff6a162970: 0x0 0x0
0x7fff6a162980: 0x7fff6a162a50 0x7fff8a31e716 <dyld_stub_binder_+13>
0x7fff6a162990: 0x1 0x7fff6a162b00
0x7fff6a1629a0: 0x7fff6a162b10 0x7fff6a162bc0
0x7fff6a1629b0: 0x8 0x0
[...]
0x7fff6a162a00: 0x0 0x0
0x7fff6a162a10: 0x0 0x0
0x7fff6a162a20: 0x0 0x0
0x7fff6a162a30: 0x7fff6a162a60 0x7fff8fab9a9c <abort+177>
0x7fff6a162a40: 0x0 0x0
0x7fff6a162a50: 0x7fffffffdf 0x0
[...]
0x7fff6a163040: 0x35000 0x0
0x7fff6a163050: 0x35000 0x500000007
0x7fff6a163060: 0x7 0x747865745f5f
0x7fff6a163070: 0x0 0x545845545f5f
0x7fff6a163080: 0x0 0x7fff5fc01000 <__dyld_stub_binding_helper>
0x7fff6a163090: 0x22c9d 0xc00001000
0x7fff6a1630a0: 0x0 0x80000400
[...]

(gdb) disass 0x7fff8a31e716
Dump of assembler code for function dyld_stub_binder_:
0x00007fff8a31e709 <dyld_stub_binder_+0>: mov 0x8(%rbp),%rdi
0x00007fff8a31e70d <dyld_stub_binder_+4>: mov 0x10(%rbp),%rsi
0x00007fff8a31e711 <dyld_stub_binder_+8>: callq 0x7fff8a31e86d <_Z21_dyld_fast_stub_entryPv>
0x00007fff8a31e716 <dyld_stub_binder_+13>: mov %rax,%r11
0x00007fff8a31e719 <dyld_stub_binder_+16>: movdqa 0x40(%rsp),%xmm0
0x00007fff8a31e71f <dyld_stub_binder_+22>: movdqa 0x50(%rsp),%xmm1
0x00007fff8a31e725 <dyld_stub_binder_+28>: movdqa 0x60(%rsp),%xmm2
0x00007fff8a31e72b <dyld_stub_binder_+34>: movdqa 0x70(%rsp),%xmm3
0x00007fff8a31e731 <dyld_stub_binder_+40>: movdqa 0x80(%rsp),%xmm4
0x00007fff8a31e73a <dyld_stub_binder_+49>: movdqa 0x90(%rsp),%xmm5
0x00007fff8a31e743 <dyld_stub_binder_+58>: movdqa 0xa0(%rsp),%xmm6
0x00007fff8a31e74c <dyld_stub_binder_+67>: movdqa 0xb0(%rsp),%xmm7
0x00007fff8a31e755 <dyld_stub_binder_+76>: mov (%rsp),%rdi
0x00007fff8a31e759 <dyld_stub_binder_+80>: mov 0x8(%rsp),%rsi
0x00007fff8a31e75e <dyld_stub_binder_+85>: mov 0x10(%rsp),%rdx
0x00007fff8a31e763 <dyld_stub_binder_+90>: mov 0x18(%rsp),%rcx
0x00007fff8a31e768 <dyld_stub_binder_+95>: mov 0x20(%rsp),%r8
0x00007fff8a31e76d <dyld_stub_binder_+100>: mov 0x28(%rsp),%r9
```

```
0x00007fff8a31e772 <dyld_stub_binder_+105>: mov 0x30(%rsp),%rax
0x00007fff8a31e777 <dyld_stub_binder_+110>: add $0xc0,%rsp
0x00007fff8a31e77e <dyld_stub_binder_+117>: pop %rbp
0x00007fff8a31e77f <dyld_stub_binder_+118>: add $0x10,%rsp
0x00007fff8a31e783 <dyld_stub_binder_+122>: jmpq *%r11

(gdb) x/2i 0x7fff8fab9a9c
0x7fff8fab9a9c <abort+177>: mov $0x2710,%edi
0x7fff8fab9aa1 <abort+182>: callq 0x7fff8fab9c43 <usleep$nocancel>

(gdb) disass 0x7fff8fab9a9c-5 0x7fff8fab9a9c
Dump of assembler code from 0x7fff8fab9a97 to 0x7fff8fab9a9c:
0x00007fff8fab9a97 <abort+172>: callq 0x7fff8fb1f54a <dyld_stub_kill>
End of assembler dump.

(gdb) disass 0x7fff5fc01000
Dump of assembler code for function __dyld_stub_binding_helper:
0x00007fff5fc01000 <__dyld_stub_binding_helper+0>: add %al,(%rax)
0x00007fff5fc01002 <__dyld_stub_binding_helper+2>: add %al,(%rax)
0x00007fff5fc01004 <__dyld_stub_binding_helper+4>: add %al,(%rax)
0x00007fff5fc01006 <__dyld_stub_binding_helper+6>: add %al,(%rax)
End of assembler dump.

(gdb) x/10 0x7fff5fc01000-0x10
0x7fff5fc00ff0: 0x00000000 0x00000000 0x00000000 0x00000000
0x7fff5fc01000 <__dyld_stub_binding_helper>: 0x00000000 0x00000000 0x00000000 0x00000000
0x7fff5fc01010 <__dyld_offset_to_dyld_all_image_infos>: 0x00000000 0x00000000
```

## Windows

Raw stack dumps can be useful for finding any suspicious modules that might have caused the problem. For example, it is common for some programs to install hooks to monitor GUI changes, intercept window messages to provide value added services on top of the existing applications. These hooks are implemented as DLLs. Another use would be to examine raw stack data for printer drivers that caused problems before. The fact that these modules had been loaded doesn't mean that they were used. If we find references to their code, it means that they might have been used.

However, when looking at raw stack dump with symbol information, we should be aware of **Coincidental Symbolic Information** pattern. Here is the first example. Loading the crash dump and displaying the problem thread stack shows the following reference:

```
...
...
...
00b1ed00 0063006f
00b1ed04 006d0075
00b1ed08 006e0065
00b1ed0c 00200074
00b1ed10 006f004c
00b1ed14 00640061
00b1ed18 00720065
00b1ed1c 005b0020
00b1ed20 00500055
00b1ed24 003a0044
00b1ed28 00430050 Application!Array::operator=+0x2f035
00b1ed2c 0035004c
00b1ed30 005d0063
00b1ed34 00630000
00b1ed38 0000005d
...
...
...
```

Applying symbols gives us a more meaningful name:

```
...
...
...
00b1ed00 0063006f
00b1ed04 006d0075
00b1ed08 006e0065
00b1ed0c 00200074
00b1ed10 006f004c
00b1ed14 00640061
00b1ed18 00720065
00b1ed1c 005b0020
00b1ed20 00500055
00b1ed24 003a0044
00b1ed28 00430050 Application!Print::DocumentLoad+0x5f
```

```
00b1ed2c 0035004c
00b1ed30 005d0063
00b1ed34 00630000
...
...
...
```

However, this is the pure coincidence. The data pattern 00NN00NN clearly belongs to a Unicode string:

```
0:020> du 00b1ed00
00b1ed00 "ocument Loader [UPD:PCL5c]"
```

It just happens that 00430050 value can be interpreted as an address that falls within *Application* module address range and its code section:

```
0:020> lm
start      end          module name
00400000  0044d000  Application
```

In the second example, the crash dump is from some 3rd-party application called *AppSql* for which we don't have PDB files. Also, we know that *myhook.dll* is installed as a system-wide hook, and it had some problems in the past. It is loaded into any address space but is not necessarily used. We want to see if there are traces of it on the problem thread stack. Dumping stack contents shows us the only one reference:

```
...
...
...
00118cb0  37302f38
00118cb4  00000000
00118cb8  10008e00 myhook!notify_me+0x22c
00118cbc  01400000
00118cc0  00118abc
00118cc4  06a129f0
00118cc8  00118d04
00118ccc  02bc57d0
00118cd0  04ba5d74
00118cd4  00118d30
00118cd8  0000001c
00118cdc  00000010
00118ce0  075922bc
00118ce4  04a732e0
00118ce8  075922bc
00118cec  04a732e0
00118cf0  0066a831 AppSql+0x26a831
00118cf4  04a732d0
00118cf8  02c43190
00118cfc  00000001
00118d00  0000001c
00118d04  00118d14
00118d08  0049e180 AppSql+0x9e180
00118d0c  02c43190
00118d10  0000001c
```

```
00118d14 00118d34
...
...
...
0:020> lm
start      end        module name
00400000 00ba8000  AppSql
...
...
10000000 100e0000  myhook
```

The address 10008e00 looks very “round” and it might be the set of bit flags, and also, if we disassemble the code at this address backward, we don’t see the usual *call* instruction that saved that address on the stack:

```
0:000> ub 10008e00
myhook!notify_me+0x211
10008de5 81c180000000    add    ecx,80h
10008deb 899578fffffff    mov    dword ptr [ebp-88h],edx
10008df1 89458c          mov    dword ptr [ebp-74h],eax
10008df4 894d98          mov    dword ptr [ebp-68h],ecx
10008df7 6a01            push   1
10008df9 8d45ec          lea    eax,[ebp-14h]
10008dfc 50              push   eax
10008dfd ff75e0          push   dword ptr [ebp-20h]
```

In contrast, the other two addresses are return addresses saved on the stack:

```
0:000> ub 0066a831
AppSql+0x26a81e:
0066a81e 8bfb          mov    edi,ebx
0066a820 f3a5          rep    movs dword ptr es:[edi],dword ptr [esi]
0066a822 8bca          mov    ecx,edx
0066a824 83e103         and    ecx,3
0066a827 f3a4          rep    movs byte ptr es:[edi],byte ptr [esi]
0066a829 8b00          mov    eax,dword ptr [eax]
0066a82b 50              push   eax
0066a82c e8afffffff    call   AppSql+0x26a6e0 (0066a6e0)

0:000> ub 0049e180
AppSql+0x9e16f:
0049e16f cc            int    3
0049e170 55            push   ebp
0049e171 8bec          mov    ebp,esp
0049e173 8b4510         mov    eax,dword ptr [ebp+10h]
0049e176 8b4d0c         mov    ecx,dword ptr [ebp+0Ch]
0049e179 50            push   eax
0049e17a 51            push   ecx
0049e17b e840c61c00    call   AppSql+0x26a7c0 (0066a7c0)
```

Therefore, the appearance of *myhook!notify\_me+0x22c* could be a coincidence unless it was a pointer to a function. However, if it was the function pointer address then it wouldn't have pointed to the middle of the function call sequence that pushes arguments:

```
0:000> ub 10008e00
myhook!notify_me+0x211
10008de5 81c180000000    add    ecx,80h
10008deb 899578fffff    mov    dword ptr [ebp-88h],edx
10008df1 89458c          mov    dword ptr [ebp-74h],eax
10008df4 894d98          mov    dword ptr [ebp-68h],ecx
10008df7 6a01            push   1
10008df9 8d45ec          lea    eax,[ebp-14h]
10008dfc 50              push   eax
10008dfd ff75e0          push   dword ptr [ebp-20h]
0:000> u 10008e00
myhook!notify_me+0x22c
10008e00 e82ff1ffff      call   myhook!copy_data (10007f34)
10008e05 8b8578fffff    mov    eax,dword ptr [ebp-88h]
10008e0b 3945ac          cmp    dword ptr [ebp-54h],eax
10008e0e 731f            jae   myhook!notify_me+0x25b (10008e2f)
10008e10 8b4598          mov    eax,dword ptr [ebp-68h]
10008e13 0fbf00          movsx  eax,word ptr [eax]
10008e16 8945a8          mov    dword ptr [ebp-58h],eax
10008e19 8b45e0          mov    eax,dword ptr [ebp-20h]
```

Also, because we have a source code and private symbols, we know that if it was a function pointer, then it would have been *myhook!notify\_me* address and not *notify\_me+0x22c* address.

All this evidence supports the hypothesis that *myhook* occurrence on the problem stack is just the coincidence and should be ignored.

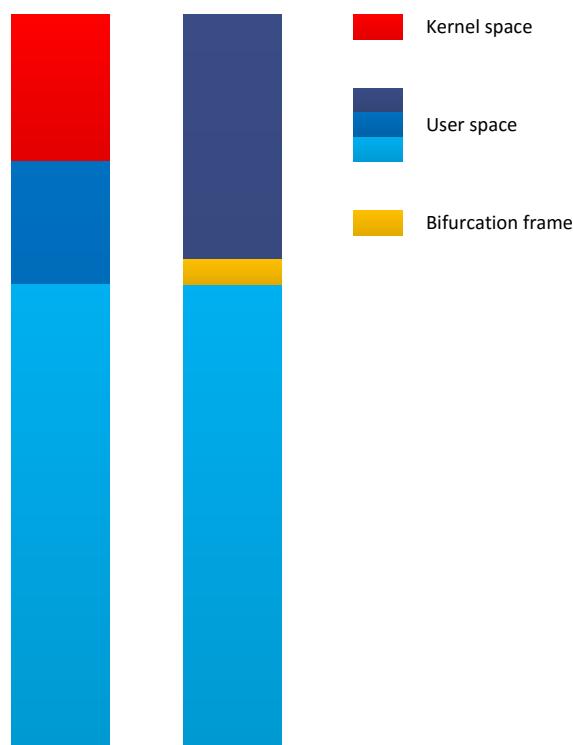
To add, the most coincidental symbolic information we have found so far in one crash dump is an accidental correspondence between exported *\_DebuggerHookData* and the location of the postmortem debugger NTSd:

```
002dd434 003a0043
002dd438 0057005c
002dd43c 004e0049 LegacyApp!_DebuggerHookData+0xc4a5
002dd440 004f0044 LegacyApp!_DebuggerHookData+0x1c4a0
002dd444 00530057
002dd448 0073005c
002dd44c 00730079
002dd450 00650074
002dd454 0033006d
002dd458 005c0032
002dd45c 0074006e
002dd460 00640073
002dd464 0065002e
002dd468 00650078

0:000> du 002dd434
002dd434 "C:\WINDOWS\system32\ntsd.exe"
```

## Constant Subtrace

**Variable Subtrace** (page 1058) analysis pattern was introduced for inter-correlational (**Inter-Correlation**<sup>18</sup>) analysis of CPU spikes across memory snapshots with just one thread involved. In contrast, we found **Constant Subtrace** pattern useful in **Wait Chain** (page 1082) analysis involving several threads in just one memory snapshot (intra-correlational analysis, **Intra-Correlation**<sup>19</sup>). Here a constant subtrace groups stack traces from **Stack Trace Collection** (page 943) with a bifurcation stack trace frame (similar to **Bifurcation Point** trace analysis pattern<sup>20</sup>) providing some wait chain relationship hint. Such traces may be initially found by the preceding wait chain analysis or by technology-specific subtraces such as ALPC/RPC server thread frames (as seen in an example stack from COM interface invocation, **Technology-Specific Subtrace**, page 988). Here is a minimal stack trace diagram (similar to minimal trace graphs introduced in *Accelerated Windows Software Trace Analysis* training<sup>21</sup>) illustrating the pattern (it also shows **Spiking Thread** pattern in user space as seen from a complete memory dump, page 888):



<sup>18</sup> Memory Dump Analysis Anthology, Volume 4, page 350

<sup>19</sup> Ibid., Volume 3, page 347

<sup>20</sup> Ibid., Volume 4, page 343

<sup>21</sup> <http://www.patterndiagnostics.com/Training/Accelerated-Windows-Software-Trace-Analysis-Public.pdf>

## Corrupt Dump

This is quite a frequent pattern and usually becomes the consequence of **Truncated Dump** pattern (page 1015). When we open such crash dumps, we usually notice immediate errors in WinDbg output. We can distinguish between two classes of corrupt memory dumps: totally corrupt and partially corrupt. Total corruption is less frequent, results from invalid file header and manifests itself in an error message box with the following Win32 error:

```
Loading Dump File [C:\Documents and Settings\All Users\Application Data\Microsoft\Dr
Watson\user_corrupted.dmp]
ERROR: Directory not present in dump (RVA 0x20202020)
Could not open dump file [C:\Documents and Settings\All Users\Application Data\Microsoft\Dr
Watson\user_corrupted.dmp], Win32 error 1392
"The file or directory is corrupted and unreadable."
```

Partially corrupt files can be loaded, but some critical information is missing, for example, the list of loaded modules and context for all or some processors. We can see lots of messages in WinDbg output like:

```
GetContextState failed, 0x80070026
Unable to get current machine context, Win32 error 0n38
```

or

```
GetContextState failed, 0x80004005
```

or

```
GetContextState failed, 0xD0000147
```

which mean:

```
? : kd> !error 0x80070026
Error code: (HRESULT) 0x80070026 (2147942438) - Reached the end of the file.

? : kd> !error 0x80004005
Error code: (HRESULT) 0x80004005 (2147500037) - Unspecified error

? : kd> !error 0xD0000147
Error code: (NTSTATUS) 0xd0000147 (3489661255) - {No Paging File Specified} No paging file was specified
in the system configuration.
```

However, in many such cases we can still see system information and bugcheck parameters:

```
*****
THIS DUMP FILE IS PARTIALLY CORRUPT.
KdDebuggerDataBlock is not present or unreadable.
*****
Unable to read PsLoadedModuleList
KdDebuggerData.KernBase < SystemRangeStart
Windows Server 2003 Kernel Version 3790 MP (4 procs) Free x86 compatible
Product: Server, suite: TerminalServer
Kernel base = 0x00000000 PsLoadedModuleList = 0x808af9c8
```

```
Debug session time: Wed Nov 21 20:29:31.373 2007 (GMT+0)
System Uptime: 0 days 0:45:02.312
Unable to read PsLoadedModuleList
KdDebuggerData.KernBase < SystemRangeStart
Loading Kernel Symbols
Unable to read PsLoadedModuleList
GetContextState failed, 0x80070026
GetContextState failed, 0x80070026
CS descriptor lookup failed
GetContextState failed, 0x80070026
GetContextState failed, 0x80070026
GetContextState failed, 0x80070026
GetContextState failed, 0x80070026
Unable to get program counter
GetContextState failed, 0x80070026
Unable to get current machine context, Win32 error 0n38
GetContextState failed, 0x80070026
GetContextState failed, 0x80070026

Use !analyze -v to get detailed debugging information.

BugCheck 20, {0, ffff, 0, 1}

***** Debugger could not find nt in module list, module list might be corrupt, error 0x80070057.

GetContextState failed, 0x80070026
Unable to read selector for PCR for processor 0
GetContextState failed, 0x80070026
Unable to read selector for PCR for processor 0
GetContextState failed, 0x80070026
Unable to read selector for PCR for processor 0
GetContextState failed, 0x80070026
GetContextState failed, 0x80070026
Unable to get current machine context, Win32 error 0n38
GetContextState failed, 0x80070026
Unable to get current machine context, Win32 error 0n38
GetContextState failed, 0x80070026
```

Looking at bugcheck number and parameters, we can form some signature and check in our crash database. We can also request a kernel minidump corresponding to debug session time.

## Comments

The main point is that if you have a corrupt dump, you may still identify the problem. Also, system administrators and support engineers can identify corrupt dumps earlier and request the new ones.

## Corrupt Structure

Typical signals of **Corrupt Structure** include:

- **Regular Data** (page 833) such as ASCII and UNICODE fragments over substructures and pointer areas
- Large values where you expect small and vice versa
- User space address values where we expect kernel space and vice versa
- Malformed and partially zeroed \_LIST\_ENTRY data (see exercise C3<sup>22</sup> for linked list navigation)
- Memory read errors for pointer dereferences or inaccessible memory indicators (??)
- Memory read error at the end of the linked list while traversing structures

```
0: kd> dt _ERESOURCE fffffd0002299f830
ntdll!_ERESOURCE
+0x000 SystemResourcesList : _LIST_ENTRY [ 0xfffffc000`07b64800 - 0xfffffe000`02a79970 ]
+0x010 OwnerTable : 0xfffffe000`02a79940 _OWNER_ENTRY
+0x018 ActiveCount : 0n0
+0x01a Flag : 0
+0x01a ReservedLowFlags : 0 ''
+0x01b WaiterPriority : 0 ''
+0x020 SharedWaiters : 0x00000000`00000001 _KSEMAPHORE
+0x028 ExclusiveWaiters : 0xfffffe000`02a79a58 _KEVENT
+0x030 OwnerEntry : _OWNER_ENTRY
+0x040 ActiveEntries : 0
+0x044 ContentionCount : 0
+0x048 NumberOfSharedWaiters : 0x7b64800
+0x04c NumberOfExclusiveWaiters : 0xfffffc000
+0x050 Reserved2 : (null)
+0x058 Address : 0xfffffd000`2299f870 Void
+0x058 CreatorBackTraceIndex : 0xfffffd000`2299f870
+0x060 SpinLock : 1

0: kd> dt _ERESOURCE fffffd0002299d830
ntdll!_ERESOURCE
+0x000 SystemResourcesList : _LIST_ENTRY [ 0x000001e0`00000280 - 0x00000000`00000004 ]
+0x010 OwnerTable : 0x00000000`0000003c _OWNER_ENTRY
+0x018 ActiveCount : 0n0
+0x01a Flag : 0
+0x01a ReservedLowFlags : 0 ''
+0x01b WaiterPriority : 0 ''
+0x020 SharedWaiters : 0x0000003c`000001e0 _KSEMAPHORE
+0x028 ExclusiveWaiters : (null)
+0x030 OwnerEntry : _OWNER_ENTRY
+0x040 ActiveEntries : 0
+0x044 ContentionCount : 0x7f
+0x048 NumberOfSharedWaiters : 0x7f
+0x04c NumberOfExclusiveWaiters : 0x7f
```

---

<sup>22</sup> Advanced Windows Memory Dump Analysis with Data Structures, Second Edition, <http://www.patterndiagnostics.com/advanced-windows-memory-dump-analysis-book>

```
+0x050 Reserved2      : 0x00000001`00000001 Void
+0x058 Address       : 0x00000000`00000005 Void
+0x058 CreatorBackTraceIndex : 5
+0x060 SpinLock       : 0
```

However, we need to be sure that we supplied the correct pointer to **dt** WinDbg command. One of the signs that the pointer was incorrect is memory read errors or all zeroes:

```
0: kd> dt _ERESOURCE fffffd000229af830
ntdll!_ERESOURCE
+0x000 SystemResourcesList : _LIST_ENTRY [ 0x00000000`00000000 - 0x00000000`00000000 ]
+0x010 OwnerTable : (null)
+0x018 ActiveCount : 0n0
+0x01a Flag : 0
+0x01a ReservedLowFlags : 0 ''
+0x01b WaiterPriority : 0 ''
+0x020 SharedWaiters : (null)
+0x028 ExclusiveWaiters : (null)
+0x030 OwnerEntry : _OWNER_ENTRY
+0x040 ActiveEntries : 0
+0x044 ContentionCount : 0
+0x048 NumberOfSharedWaiters : 0
+0x04c NumberOfExclusiveWaiters : 0
+0x050 Reserved2 : (null)
+0x058 Address : (null)
+0x058 CreatorBackTraceIndex : 0
+0x060 SpinLock : 0

0: kd> dt _ERESOURCE fffffd00022faf830
ntdll!_ERESOURCE
+0x000 SystemResourcesList : _LIST_ENTRY
+0x010 OwnerTable      : ???
+0x018 ActiveCount     : ??
+0x01a Flag            : ??
+0x01a ReservedLowFlags : ??
+0x01b WaiterPriority   : ??
+0x020 SharedWaiters    : ???
+0x028 ExclusiveWaiters : ???
+0x030 OwnerEntry       : _OWNER_ENTRY
+0x040 ActiveEntries    : ??
+0x044 ContentionCount  : ??
+0x048 NumberOfSharedWaiters : ??
+0x04c NumberOfExclusiveWaiters : ??
+0x050 Reserved2        : ???
+0x058 Address          : ???
+0x058 CreatorBackTraceIndex : ??
+0x060 SpinLock         : ??
Memory read error fffffd00022faf890
```

## Coupled Machines

Sometimes we have threads that wait for a response from another machine (for example, via RPC). For most of the time, **Coupled Processes** pattern (page 149) covers that if we assume that processes in that pattern are not restricted to the same machine. However, sometimes we have threads that provide hints for dependency on another machine through their data, and that could also involve additional threads from different processes to accomplish the task. Here we need another pattern that we call **Coupled Machines**. For example, the following thread on a computer SERVER\_A is trying to set the current working directory that resides on a computer SERVER\_B:

```
kd> kv 100
ChildEBP RetAddr  Args to Child
b881c8d4 804e1bf2 89cd9c80 89cd9c10 804e1c3e nt!KiSwapContext+0x2f
b881c8e0 804e1c3e 00000000 89e35b08 89e35b34 nt!KiSwapThread+0x8a
b881c908 f783092e 00000000 00000006 00000000 nt!KeWaitForSingleObject+0x1c2
b881c930 f7830a3b 89e35b08 00000000 f78356d8 Mup!PktPostSystemWork+0x3d
b881c94c f7836712 b881c9b0 b881c9b0 b881c9b8 Mup!PktGetReferral+0xce
b881c980 f783644f b881c9b0 b881c9b8 00000000 Mup!PktCreateDomainEntry+0x224
b881c9d0 f7836018 0000000b 00000000 b881c9f0 Mup!DfsFsctrlIsThisADfsPath+0x2bb
b881ca14 f7835829 89a2e130 899ba350 b881caac Mup!CreateRedirectedFile+0x2cd
b881ca70 804e13eb 89f46ee8 89a2e130 89a2e130 Mup!MupCreate+0x1cb
b881ca80 805794b6 89f46ed0 89df3c44 b881cc18 nt!IopfCallDriver+0x31
b881cb60 8056d03b 89f46ee8 00000000 89df3ba0 nt!IopParseDevice+0xa12
b881cbd8 805701e7 00000000 b881cc18 00000042 nt!ObpLookupObjectName+0x53c
b881cc2c 80579b12 00000000 00000000 00003801 nt!ObOpenObjectByName+0xea
b881cca8 80579be1 00cff67c 00100020 00cff620 nt!IopCreateFile+0x407
b881cd04 80579d18 00cff67c 00100020 00cff620 nt!IoCreateFile+0x8e
b881cd44 804dd99f 00cff67c 00100020 00cff620 nt!NtOpenFile+0x27
b881cd44 7c90e514 00cff67c 00100020 00cff620 nt!KiFastCallEntry+0xfc
00cff5f0 7c90d5aa 7c91e8dd 00cff67c 00100020 ntdll!KiFastSystemCallRet
00cff5f4 7c91e8dd 00cff67c 00100020 00cff620 ntdll!ZwOpenFile+0xc
00cff69c 7c831e58 00cff6a8 00460044 0078894a ntdll!RtlSetCurrentDirectory_U+0x169
00cff6b0 7731889e 0078894a 00000000 00000001 kernel32!SetCurrentDirectoryW+0x2b
00cffb84 7730ffbb 00788450 00788b38 00cffbe0 schedsvc!CSchedWorker::RunNTJob+0x221
00cffe34 7730c03a 01ea9108 8ed032d4 00787df8 schedsvc!CSchedWorker::RunJobs+0x304
00cfffe74 77310e4d 7c80a749 00000000 00000000 schedsvc!CSchedWorker::RunNextJobs+0x129
00cfffb28 77310efc 7730b592 00000000 000ba4bc schedsvc!CSchedWorker::MainServiceLoop+0x6d9
00cfffb2c 7730b592 00000000 000ba4bc 0009a2bc schedsvc!SchedMain+0xb
00cfffb5c 7730b69f 00000001 000ba4b8 00cfffa0 schedsvc!SchedStart+0x266
00cfffb6c 010011cc 00000001 000ba4b8 00000000 schedsvc!SchedServiceMain+0x33
00cfffa0 77df354b 00000001 000ba4b8 0007e898 svchost!ServiceStarter+0x9e
00cfffb4 7c80b729 000ba4b0 00000000 0007e898 ADVAPI32!ScSvcctrlThreadA+0x12
00cfffec 00000000 77df3539 000ba4b0 00000000 kernel32!BaseThreadStart+0x37

kd> du /c 90 0078894a
0078894a  "\\SERVER_B\Share_X$\Folder_Q"
```

## Coupled Modules

Often we identify a pattern that points to a particular module such as a driver or DLL other modules could use functional services from and, therefore, the latter modules can be implicated in abnormal software behavior. For example, detected **Insufficient Memory** in kernel paged pool (page 535) pointed to a driver that owns a pool tag DRV:

```
1: kd> !poolused 4
Sorting by Paged Pool Consumed

Tag   Allocs   Frees   Diff   Used
DRV  1466496  1422361  44135  188917256 UNKNOWN pooltag 'DRV ', please update pooltag.txt
File 6334830 6284036 50794 6735720 File objects
Thre 53721 45152 8569 4346432 Thread objects , Binary: nt!ps
[...]
```

This module is known to be **Directing Module** (page 235) to other drivers (from the stack trace perspective), but we also know that other (directed) modules use its services that require memory allocation.

## Coupled Processes

### Semantics

In addition to **Strong** (page 149) and **Weak** (page 151) process coupling patterns, we also have another variant that we call **Semantic** coupling. Some processes (not necessarily from the same vendor) cooperate to provide certain functionality. The cooperation might not involve trackable and visible inter-process communication such as (A)LPC/RPC or pipes but involve events, shared memory and other possible mechanisms not explicitly visible when we look at memory dumps. In many cases, after finding problems in one or several processes from a semantic group, we also look at the remaining processes from that group to see if there are some anomalies there as well. The one example I encounter often can be generalized as follows: we have an ALPC wait chain *ProcessA* → *ProcessB* <-> *ProcessC* (not necessarily **Deadlock**, page 197) but the crucial piece of functionality is also implemented in *ProcessD*. Sometimes *ProcessD* is healthy, and the problem resides in *ProcessC* or *ProcessB*, and sometimes, when we look at *ProcessD* we find evidence of an earlier problem pattern there so the focus of recommendations shifts to one of the *ProcessD* modules.

## Strong

Sometimes we have a problem that some functionality is not available, or it is unresponsive when we request it. Then we can suppose that the process implementing that functionality has crashed or hangs. If we know the relationship between processes, we can request several user dumps at once or a complete memory dump to analyze the dependency between processes by looking at their stack traces. This is an example of the system level crash dump analysis pattern that we call **Coupled Processes**.

Process relationship can be implemented via different interprocess communication mechanisms (IPC), for example, Remote Procedure Call (RPC) via LPC (Local Procedure Call) which can be easily identified in **Stack Traces** (page 926).

Our favorite example here is when some application tries to print and hangs. Printing API is exported from *WINSPOOL.DLL*, and it forwards via RPC most requests to Windows Print Spooler service. Therefore, it is logical to take two dumps, one from that application, and one from *spoolsv.exe*. A similar example is from Citrix terminal services environments related to printer autocreation when there are dependencies between Citrix Printing Service *CpSvc.exe* and *spoolsv.exe*. Therefore if new user connections hang and restarting both printing services resolves the issue then we might need to analyze memory dumps from both services together to confirm this **Procedure Call Chain**<sup>23</sup> and find the problem 3rd-party printing component or driver.

Back to our favorite example. In the hang application we have the following thread:

```
18 Id: 2130.6320 Suspend: 1 Teb: 7ffa8000 Unfrozen
ChildEBP RetAddr
01eae170 7c821c94 ntdll!KiFastSystemCallRet
01eae174 77c72700 ntdll!NtRequestWaitReplyPort+0xc
01eae1c8 77c713ba rpcrt4!LRPC_CCALL::SendReceive+0x230
01eae1d4 77c72c7f rpcrt4!I_RpcSendReceive+0x24
01eae1e8 77ce219b rpcrt4!NdrSendReceive+0x2b
01eae5d0 7307c9ef rpcrt4!NdrClientCall2+0x22e
01eae5e8 73082d8d winspool!RpcAddPrinter+0x1c
01eaea70 0040d81a winspool!AddPrinterW+0x102
01eaef58 0040ee7c App!AddNewPrinter+0x816
...
...
...
```

Notice *winspool* and *rpcrt4* modules. The application is calling spooler service using RPC to add a new printer and waiting for a reply back. Looking at spooler service dump shows several threads displaying message boxes and waiting for user input:

---

<sup>23</sup> This is a pattern we may add in the future

```

20 Id: 790.5950 Suspend: 1 Teb: 7ffa2000 Unfrozen
ChildEBP RetAddr Args to Child
03deea70 7739d02f 77392bf3 00000000 00000000 ntdll!KiFastSystemCallRet
03deea8 7738f122 03dd0058 00000000 00000001 user32!NtUserWaitMessage+0xc
03deead0 773a1722 77380000 00123690 00000000 user32!InternalDialogBox+0xd0
03deed90 773a1004 03eeeeec 03dae378 03dae160 user32!SoftModalMessageBox+0x94b
03deeee0 773b1a28 03eeeeec 00000028 00000000 user32!MessageBoxWorker+0x2ba
03deef38 773b19c4 00000000 03defb9c 03def39c user32!MessageBoxTimeoutW+0x7a
03deef58 773b19a0 00000000 03defb9c 03def39c user32!MessageBoxExW+0x1b
03deef74 021f265b 00000000 03defb9c 03def39c user32!MessageBoxW+0x45
WARNING: Stack unwind information not available. Following frames may be wrong.
03def88 00000000 03dae160 03deffec 03dae16a PrinterDriver!UninstallerInstall+0x2cb

```

Dumping the 3rd parameter of *MessageBoxW* using WinDbg **du** command shows the message:

“Installation of the software for your printer is now complete. Restart your computer to make the new settings active.”

Another example is when one process starts another and then waiting for it to finish:

```

0 Id: 2a34.24d0 Suspend: 1 Teb: 7ffde000 Unfrozen
ChildEBP RetAddr
0007ec8c 7c822124 ntdll!KiFastSystemCallRet
0007ec90 77e6bad8 ntdll!NtWaitForSingleObject+0xc
0007ed00 77e6ba42 kernel32!WaitForSingleObjectEx+0xac
0007ed14 01002f4c kernel32!WaitForSingleObject+0x12
0007f79c 01003137 userinit!ExecApplication+0x2d3
0007f7dc 0100366b userinit!ExecProcesses+0x1bb
0007fe68 010041fd userinit!StartTheShell+0x132
0007ff1c 010056f1 userinit!WinMain+0x263
0007ffc0 77e523e5 userinit!WinMainCRTStartup+0x186

```

## Comments

Good diagrams explaining basic printing architecture<sup>24</sup>.

Print driver isolation in W2K8 R2/W7<sup>25</sup>.

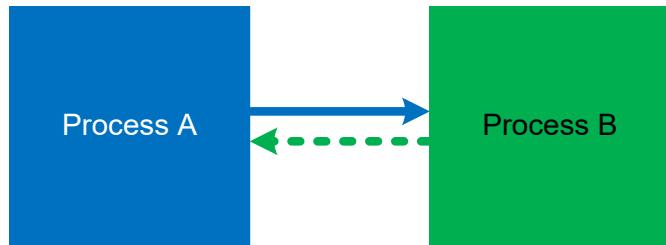
---

<sup>24</sup> <http://blogs.technet.com/b/askperf/archive/2007/06/19/basic-printing-architecture.aspx>

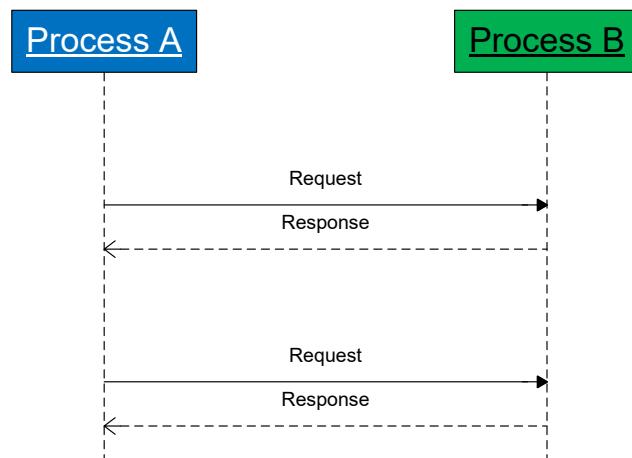
<sup>25</sup> <http://blogs.technet.com/b/askperf/archive/2009/10/08/windows-7-windows-server-2008-r2-print-driver-isolation.aspx>

## Weak

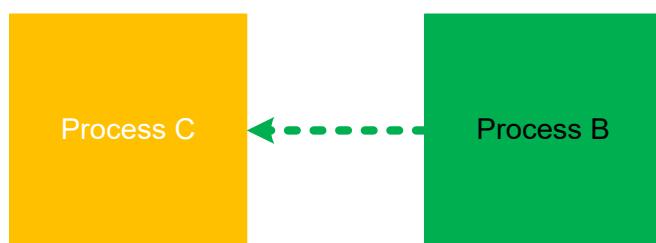
Previously introduced **Strong** version of **Coupled Processes** (page 149) pattern involves an active request (or an action) and an active wait for a response (or the action status):



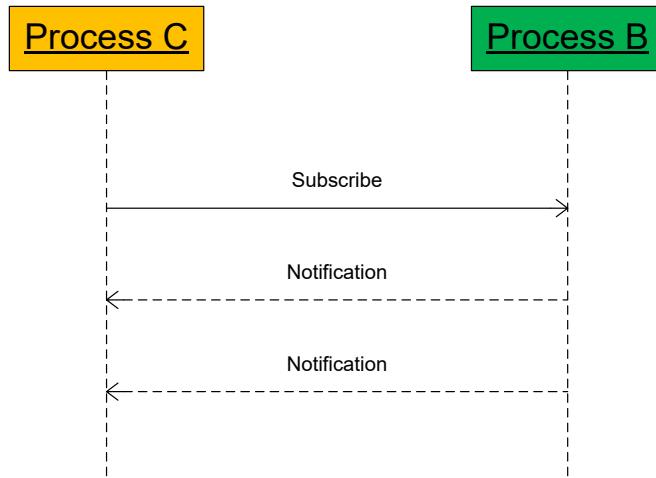
It is illustrated on this simple UML sequence diagram (process timeline represent collective request-response threads):



However, there is so-called **Weak** coupling when a process subscribes for notifications. Such threads, for the most time, are **Passive Threads** (page 793), and processes are not blocked:



The coupling manifests itself when notifier threads start spiking CPU (**Spiking Thread**, page 888) and bring their share of CPU consumption to the notified threads:



Here is an example of such threads:

```

5  Id: 61018.dbec Suspend: 1 Teb: 7ffae000 Unfrozen
ChildEBP RetAddr
01e3fa68 7c82787b nt!KiFastSystemCallRet
01e3fa6c 77c80a6e nt!NtRequestWaitReplyPort+0xc
01e3fab8 77c7fcf0 rpct4!LRPC_CCALL::SendReceive+0x230
01e3fac4 77c80673 rpct4!I_RpcSendReceive+0x24
01e3fad8 77ce315a rpct4!NdrSendReceive+0x2b
01e3fec0 771f4fb0 rpct4!NdrClientCall2+0x22e
01e3fed8 771f4f60 winsta!RpcWinStationWaitSystemEvent+0x1c
01e3ff20 6582116c winsta!WinStationWaitSystemEvent+0x51
[...]
01e3ffec 00000000 kernel32!BaseThreadStart+0x34
  
```

In cases of synchronous notifications, if a notified thread is blocked we have an instance of a reversed strong coupling.

## Crash Signature

This is one of the most common patterns. It consists of a set of attributes derivable from saved execution context for exceptions, faults, and traps. For example, on x64 Windows, it is usually RIP and RSP addresses. For x86, it is usually EIP, ESP, and EBP. It can also include the application module name.

```
0:009> !analyze -v

[...]

FAILURE_BUCKET_ID: SOFTWARE_NX_FAULT_c0000005_ApplicationA.exe!Unknown

BUCKET_ID: APPLICATIONFAULT_SOFTWARE_NXFAULT_STACKCORRUPTION_BAD_IP_ApplicationA+2d560

[...]

0:009> kL
ChildEBP RetAddr
0354f270 75bc0962 ntdll!NtWaitForMultipleObjects+0x15
0354f30c 7651162d KERNELBASE!WaitForMultipleObjectsEx+0x100
0354f354 76511921 kernel32!WaitForMultipleObjectsExImplementation+0xe0
0354f370 76539b0d kernel32!WaitForMultipleObjects+0x18
0354f3dc 76539baa kernel32!WerReportFaultInternal+0x186
0354f3f0 765398d8 kernel32!WerReportFault+0x70
0354f400 76539855 kernel32!BaseReportFault+0x20
0354f48c 77750727 kernel32!UnhandledExceptionFilter+0x1af
0354f494 77750604 ntdll!__RtlUserThreadStart+0x62
0354f4a8 777504a9 ntdll!_EH4_CallFilterFunc+0x12
0354f4d0 777387b9 ntdll!_except_handler4+0x8e
0354f4f4 7773878b ntdll!ExecuteHandler2+0x26
0354f5a4 776f010f ntdll!ExecuteHandler+0x24
0354f5a4 0354f958 ntdll!KiUserExceptionDispatcher+0xf
WARNING: Frame IP not in any known module. Following frames may be wrong.
0354f908 02ff0340 0x354f958
00000000 00000000 0x2ff0340

0:009> kv
ChildEBP RetAddr Args to Child
[...]
0354f5a4 0354f958 0154f5bc 0354f60c 0354f5bc ntdll!KiUserExceptionDispatcher+0xf (CONTEXT @ 0354f60c)
WARNING: Frame IP not in any known module. Following frames may be wrong.
0354f908 02ff0340 00000000 00000000 00000000 0x354f958
00000000 00000000 00000000 00000000 00000000 0x2ff0340

0:009> .cxr 0354f60c
eax=80010105 ebx=0354f924 ecx=00000003 edx=0000ffff esi=00d7dce0 edi=00d7e0c8
eip=0354f958 esp=0354f8f4 ebp=0354f908 iopl=0 nv up ei pl zr na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010246
0354f958 64f9 stc
```

```
0:009> !address 0354f958
TEB 7efdd000 in range 7efdb000 7efde000
TEB 7efda000 in range 7efd8000 7efdb000
TEB 7efd7000 in range 7efd5000 7efd8000
TEB 7efaf000 in range 7efad000 7efb0000
TEB 7efac000 in range 7efaa000 7efad000
TEB 7efa9000 in range 7efa7000 7efaa000
TEB 7efa6000 in range 7efa4000 7efa7000
TEB 7efa3000 in range 7efa1000 7efa4000
TEB 7ef9f000 in range 7ef9d000 7efa0000
TEB 7ef9c000 in range 7ef9a000 7ef9d000
TEB 7ef99000 in range 7ef97000 7ef9a000
ProcessParametrs      007714b0 in range 00770000 00870000
Environment          007707f0 in range 00770000 00870000
03450000 : 0354d000 - 00003000
Type                 00020000 MEM_PRIVATE
Protect              00000004 PAGE_READWRITE
State                00001000 MEM_COMMIT
Usage                RegionUsageStack
Pid.Tid              1ea0.12dc

0:009> !address 02ff0340
TEB 7efdd000 in range 7efdb000 7efde000
TEB 7efda000 in range 7efd8000 7efdb000
TEB 7efd7000 in range 7efd5000 7efd8000
TEB 7efaf000 in range 7efad000 7efb0000
TEB 7efac000 in range 7efaa000 7efad000
TEB 7efa9000 in range 7efa7000 7efaa000
TEB 7efa6000 in range 7efa4000 7efa7000
TEB 7efa3000 in range 7efa1000 7efa4000
TEB 7ef9f000 in range 7ef9d000 7efa0000
TEB 7ef9c000 in range 7ef9a000 7ef9d000
TEB 7ef99000 in range 7ef97000 7ef9a000
ProcessParametrs      007714b0 in range 00770000 00870000
Environment          007707f0 in range 00770000 00870000
02fc0000 : 02fc0000 - 00043000
Type                 00020000 MEM_PRIVATE
Protect              00000004 PAGE_READWRITE
State                00001000 MEM_COMMIT
Usage                RegionUsageHeap
Handle 00d70000
```

**Stack Trace** (page 926) may or may not be included here, and it might be **Incorrect Stack Trace** (page 499), heuristic<sup>26</sup> and not fully discernible automatically (requires raw stack semantic analysis) like in the example above. In some cases, we may have **Invalid Exception Information** (page 568) though, for example, in the case of **Laterally Damaged** (page 602) memory dumps or **Truncated Dump** (page 1015) files.

---

<sup>26</sup> Heuristic Stack Trace, Memory Dump Analysis Anthology, Volume 2, page 43

## Crash Signature Invariant

Sometimes there are crashes in multiplatform products where only some portion of **Crash Signature** (page 153) is similar, for example:

```
x86: cmp dword ptr [eax], 1
x64: cmp dword ptr [r10], 1
```

One crash dump had the following condensed stack trace:

```
0: kd> kc
DriverA
win32k!DrvSetMonitorPowerState
win32k!xxxSysCommand
win32k!xxxRealDefWindowProc
win32k!NtUserfnNCDESTROY
win32k!NtUserMessageCall
nt!KiSystemServiceCopyEnd
```

With the following faulting instruction:

```
DriverA+0x1234:
cmp     dword ptr [r11],1 ds:002b:00000000`00000000=????????
```

A search for DriverA led to this x86 crash analyzed some time ago:

```
0: kd> kc
DriverA
nt!IopfCallDriver
win32k!GreDeviceIoControl
win32k!DrvSetMonitorPowerState
win32k!xxxSysCommand
win32k!xxxRealDefWindowProc
win32k!xxxWrapRealDefWindowProc
win32k!NtUserfnNCDESTROY
win32k!NtUserMessageCall
nt!KiSystemServicePostCall

0: kd> r
DtiverA+0x1423:
cmp     dword ptr [ecx],1     ds:0023:00000000=????????
```

We see common function names on both stack traces, and overall flow is the same (only 3 functions are omitted in x64 trace); we see the same NULL pointer dereference for the same comparison instruction with the same comparison operand, #1.

## Crashed Process

Sometimes we can see signs of **Crashed Processes** in the kernel and complete memory dumps. By “crashes”<sup>27</sup> we mean the sudden disappearance of processes from Task Manager, for example. In memory dumps, we can still see such processes as **Zombie Processes** (page 1158). **Special Processes** (page 877) found in the process list may help to select the possible candidate among many **Zombie Processes**. If a process is supposed to be launched only once (as a service) but found several times as **Zombie Process** and also as a normal process later in the process list (for example, as **Last Object**, page 599), then this may point to possible past crashes (or silent terminations). We also have a similar trace analysis pattern: **Singleton Event**<sup>28</sup>. The following example illustrates both signs:

```
0: kd> !process 0 0  
[...]  
PROCESS ffffffa80088a5640  
SessionId: 0 Cid: 2184 Peb: 7fffffd7000 ParentCid: 0888  
DirBase: 381b8000 ObjectTable: 00000000 HandleCount: 0.  
Image: WerFault.exe  
  
PROCESS ffffffa8007254b30  
SessionId: 0 Cid: 20ac Peb: 7fffffd000 ParentCid: 02cc  
DirBase: b3306000 ObjectTable: 00000000 HandleCount: 0.  
Image: ServiceA.exe  
  
[...]  
  
PROCESS ffffffa8007fe2b30  
SessionId: 0 Cid: 2a1c Peb: 7fffffd000 ParentCid: 02cc  
DirBase: 11b649000 ObjectTable: fffff8a014939530 HandleCount: 112.  
Image: ServiceA.exe
```

<sup>27</sup> Crashes and Hangs Differentiated, Memory Dump Analysis Anthology, Volume 1, page 36

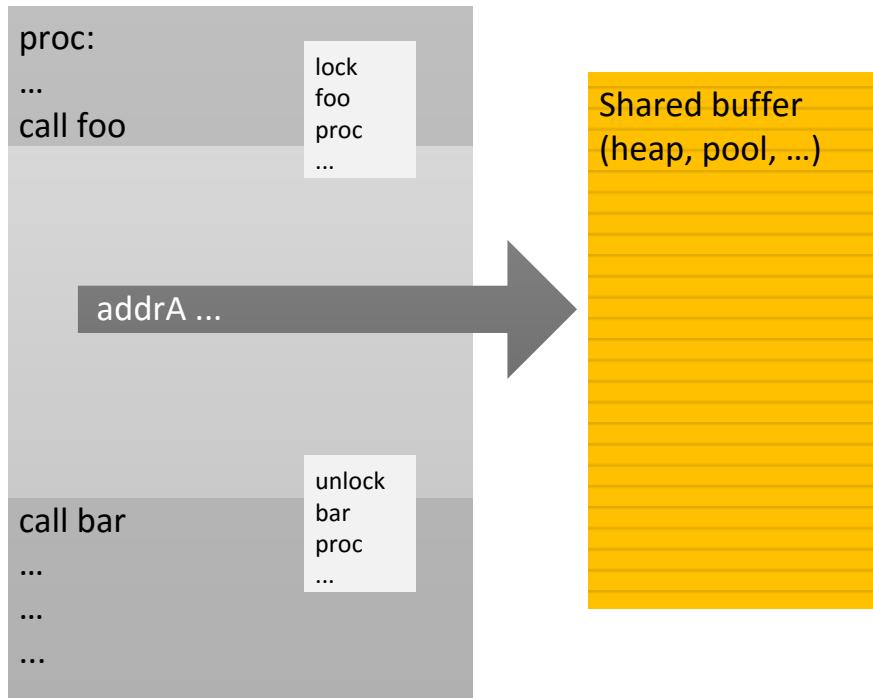
<sup>28</sup> Singleton Event, Memory Dump Analysis Anthology, Volume 8a, page 108

## Critical Region

### Linux

We first introduced **Critical Region** pattern in *Accelerated Mac OS X Core Dump Analysis*<sup>29</sup> training but didn't submit the pattern itself to the catalog at that time.

A critical region is usually a region of code protected by synchronization objects such as critical sections and mutexes. However, **Critical Region** analysis pattern is about identifying code regions "sandwiched" between contending function calls (which may or may not involve synchronization objects and corresponding synchronization calls such as identified in **Contention** patterns, page 1168), and then identifying any possibly shared data referenced by such code regions:



<sup>29</sup> <http://www.patterndiagnostics.com/accelerated-macosx-core-dump-analysis-book>

```
(gdb) thread apply all bt

Thread 6 (Thread 0x7f2665377700 (LWP 17000)):
#0  0x00000000004151a1 in _int_malloc ()
#1  0x0000000000416cf8 in malloc ()
#2  0x00000000004005a4 in proc ()
#3  0x0000000000400604 in bar_two ()
#4  0x0000000000400614 in foo_two ()
#5  0x000000000040062c in thread_two ()
#6  0x00000000004016c0 in start_thread (arg=<optimized out>
at pthread_create.c:304
#7  0x0000000000432589 in clone ()
#8  0x0000000000000000 in ?? ()

Thread 5 (Thread 0x7f2664b76700 (LWP 17001)):
#0  __l11_unlock_wake_private ()
at ../nptl/sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:343
#1  0x000000000041886d in _L_unlock_9670 ()
#2  0x0000000000416d22 in malloc ()
#3  0x00000000004005a4 in proc ()
#4  0x0000000000400641 in bar_three ()
#5  0x0000000000400651 in foo_three ()
#6  0x0000000000400669 in thread_three ()
#7  0x00000000004016c0 in start_thread (arg=<optimized out>
at pthread_create.c:304
#8  0x0000000000432589 in clone ()
#9  0x0000000000000000 in ?? ()

Thread 4 (Thread 0x7f2665b78700 (LWP 16999)):
#0  __l11_lock_wait_private ()
at ../nptl/sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:97
#1  0x0000000000418836 in _L_lock_9558 ()
#2  0x0000000000416c1c in free ()
#3  0x0000000000400586 in proc ()
#4  0x00000000004005c7 in bar_one ()
#5  0x00000000004005d7 in foo_one ()
#6  0x00000000004005ef in thread_one ()
#7  0x00000000004016c0 in start_thread (arg=<optimized out>
at pthread_create.c:304
#8  0x0000000000432589 in clone ()
#9  0x0000000000000000 in ?? ()

Thread 3 (Thread 0x1ab1860 (LWP 16998)):
#0  0x000000000042fed1 in nanosleep ()
#1  0x000000000042fd0 in sleep ()
#2  0x000000000040078a in main ()

Thread 2 (Thread 0x7f2663b74700 (LWP 17003)):
#0  __l11_lock_wait_private ()
at ../nptl/sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:97
#1  0x0000000000418836 in _L_lock_9558 ()
#2  0x0000000000416c1c in free ()
#3  0x0000000000400586 in proc ()
#4  0x00000000004006bb in bar_five ()
#5  0x00000000004006cb in foo_five ()
```

```
#6 0x00000000004006e3 in thread_five ()
#7 0x00000000004016c0 in start_thread (arg=<optimized out>
at pthread_create.c:304
#8 0x0000000000432589 in clone ()
#9 0x0000000000000000 in ?? ()

Thread 1 (Thread 0x7f2664375700 (LWP 17002)):
#0 0x000000000043ef65 in raise ()
#1 0x0000000000409fc0 in abort ()
#2 0x000000000040bf5b in __libc_message ()
#3 0x0000000000412042 in malloc_printerr ()
#4 0x0000000000416c27 in free ()
#5 0x0000000000400586 in proc ()
#6 0x000000000040067e in bar_four ()
#7 0x000000000040068e in foo_four ()
#8 0x00000000004006a6 in thread_four ()
#9 0x00000000004016c0 in start_thread (arg=<optimized out>
at pthread_create.c:304
#10 0x0000000000432589 in clone ()
#11 0x0000000000000000 in ?? ()
```

From threads #4 and #5 we can identify one such a region with a shared buffer 0xb8fc0 which may further point to heap entries.

```
(gdb) disassemble proc
Dump of assembler code for function proc:
0x00000000004004f0 <+0>: push    %rbp
0x00000000004004f1 <+1>: mov     %rsp,%rbp
0x00000000004004f4 <+4>: push    %rbx
0x00000000004004f5 <+5>: sub     $0x18,%rsp
0x00000000004004f9 <+9>: callq   0x40ac70 <rand>
0x00000000004004fe <+14>: mov     %eax,%ecx
0x0000000000400500 <+16>: mov     $0x68db8bad,%edx
0x0000000000400505 <+21>: mov     %ecx,%eax
0x0000000000400507 <+23>: imul   %edx
0x0000000000400509 <+25>: sar    $0xc,%edx
0x000000000040050c <+28>: mov     %ecx,%eax
0x000000000040050e <+30>: sar    $0x1f,%eax
0x0000000000400511 <+33>: mov     %edx,%ebx
0x0000000000400513 <+35>: sub    %eax,%ebx
0x0000000000400515 <+37>: mov     %ebx,%eax
0x0000000000400517 <+39>: mov     %eax,-0x14(%rbp)
0x000000000040051a <+42>: mov     -0x14(%rbp),%eax
0x000000000040051d <+45>: imul   $0x2710,%eax,%eax
0x0000000000400523 <+51>: mov     %ecx,%edx
0x0000000000400525 <+53>: sub    %eax,%edx
0x0000000000400527 <+55>: mov     %edx,%eax
0x0000000000400529 <+57>: mov     %eax,-0x14(%rbp)
0x000000000040052c <+60>: callq  0x40ac70 <rand>
0x0000000000400531 <+65>: mov     %eax,%ecx
0x0000000000400533 <+67>: mov     $0x68db8bad,%edx
0x0000000000400538 <+72>: mov     %ecx,%eax
0x000000000040053a <+74>: imul   %edx
0x000000000040053c <+76>: sar    $0xc,%edx
0x000000000040053f <+79>: mov     %ecx,%eax
```

```

0x0000000000400541 <+81>: sar    $0x1f,%eax
0x0000000000400544 <+84>: mov    %edx,%ebx
0x0000000000400546 <+86>: sub    %eax,%ebx
0x0000000000400548 <+88>: mov    %ebx,%eax
0x000000000040054a <+90>: mov    %eax,-0x18(%rbp)
0x000000000040054d <+93>: mov    -0x18(%rbp),%eax
0x0000000000400550 <+96>: imul   $0x2710,%eax,%eax
0x0000000000400556 <+102>: mov    %ecx,%edx
0x0000000000400558 <+104>: sub    %eax,%edx
0x000000000040055a <+106>: mov    %edx,%eax
0x000000000040055c <+108>: mov    %eax,-0x18(%rbp)
0x000000000040055f <+111>: mov    -0x14(%rbp),%eax
0x0000000000400562 <+114>: cltq   0x6b8fc0(%rax,8),%rax
0x0000000000400564 <+116>: mov    %rax,%rax
0x000000000040056c <+124>: test   0x400597 <proc+167>
0x000000000040056f <+127>: je     -0x14(%rbp),%eax
0x0000000000400571 <+129>: mov    0x416bc0 <free>
0x0000000000400574 <+132>: cltq   -0x14(%rbp),%eax
0x0000000000400576 <+134>: mov    0x6b8fc0(%rax,8),%rax
0x000000000040057e <+142>: mov    %rax,%rdi
0x0000000000400581 <+145>: callq  0x416bc0 <free>
0x0000000000400586 <+150>: mov    -0x14(%rbp),%eax
0x0000000000400589 <+153>: cltq   0x6b8fc0(%rax,8)
0x000000000040058b <+155>: movq   $0x0,0x6b8fc0(%rax,8)
0x0000000000400597 <+167>: mov    -0x18(%rbp),%eax
0x000000000040059a <+170>: cltq   0x416c90 <malloc>
0x000000000040059c <+172>: mov    %rax,%rdi
0x000000000040059f <+175>: callq  %rax,%rdx
0x00000000004005a4 <+180>: mov    0x4004f9 <proc+9>
0x00000000004005a7 <+183>: mov    -0x14(%rbp),%eax
0x00000000004005aa <+186>: cltq   %rdx,0x6b8fc0(%rax,8)
0x00000000004005ac <+188>: mov    0x6b8fc0(%rax,8)
0x00000000004005b4 <+196>: jmpq   End of assembler dump.

```

## Critical Section Corruption

**Dynamic Memory Corruption** patterns in user (page 307) and kernel (page 292) spaces are specializations of one big parent pattern called **Corrupt Structure** (page 144) because crashes happen there due to corrupt or overwritten heap or pool control structures (for the latter see **Double Free** pattern, page 259). Critical sections are linked together through statically pre-allocated or heap-allocated helper structure (shown in bold italic) although themselves they can be stored anywhere from static and stack area to process heap:

```
0:001> dt -r1 ntdll!_RTL_CRITICAL_SECTION 77795240
+0x000 DebugInfo      : 0x00175d28 _RTL_CRITICAL_SECTION_DEBUG
+0x000 Type          : 0
+0x002 CreatorBackTraceIndex : 0
+0x004 CriticalSection : 0x77795240 _RTL_CRITICAL_SECTION
+0x008 ProcessLocksList : _LIST_ENTRY [ 0x173a08 - 0x173298 ]
+0x010 EntryCount    : 0
+0x014 ContentionCount : 0
+0x018 Spare         : [2] 0
+0x004 LockCount     : -1
+0x008 RecursionCount : 0
+0x00c OwningThread   : (null)
+0x010 LockSemaphore  : (null)
+0x014 SpinCount      : 0

0:001> !address 77795240
77670000 : 77792000 - 000005000
  Type      01000000 MEM_IMAGE
  Protect   00000004 PAGE_READWRITE
  State     00001000 MEM_COMMIT
  Usage     RegionUsageImage
  FullPath  C:\WINDOWS\system32\ole32.dll

0:001> !address 0x00175d28
00140000 : 00173000 - 00000d000
  Type      00020000 MEM_PRIVATE
  Protect   00000004 PAGE_READWRITE
  State     00001000 MEM_COMMIT
  Usage     RegionUsageHeap
  Handle   00140000
```

```

0:000> !locks

CritSec ntdll!LdrpLoaderLock+0 at 7c8877a0
WaiterWoken      No
LockCount        0
RecursionCount   1
OwningThread    1184
EntryCount       0
ContentionCount b04707
*** Locked

0:000> dt -r1 _RTL_CRITICAL_SECTION 7c8877a0
+0x000 DebugInfo      : 0x7c8877c0 _RTL_CRITICAL_SECTION_DEBUG
+0x000 Type           : 0
+0x002 CreatorBackTraceIndex : 0
+0x004 CriticalSection : 0x7c8877a0 _RTL_CRITICAL_SECTION
+0x008 ProcessLocksList : _LIST_ENTRY [ 0x7c887be8 - 0x7c887bc8 ]
+0x010 EntryCount     : 0
+0x014 ContentionCount : 0xb04707
+0x018 Spare          : [2] 0
+0x004 LockCount      : -2
+0x008 RecursionCount : 1
+0x00c OwningThread   : 0x00001184
+0x010 LockSemaphore  : 0x0000013c
+0x014 SpinCount      : 0

0:000> !address 7c8877a0
7c800000 : 7c887000 - 00003000
    Type      01000000 MEM_IMAGE
    Protect   00000004 PAGE_READWRITE
    State     00001000 MEM_COMMIT
    Usage     RegionUsageImage
   FullPath C:\WINDOWS\system32\ntdll.dll

0:000> !address 0x7c8877c0
7c800000 : 7c887000 - 00003000
    Type      01000000 MEM_IMAGE
    Protect   00000004 PAGE_READWRITE
    State     00001000 MEM_COMMIT
    Usage     RegionUsageImage
   FullPath C:\WINDOWS\system32\ntdll.dll

```

Consider the case when CRITICAL\_SECTION structure is defined on a stack, and there was **Local Buffer Overflow** (page 608) overwriting *DebugInfo* pointer. Then we have an example of **Wild Pointer** pattern (page 1151) and traversing the list of critical sections from this point will diverge into completely unrelated memory area or stop there. Consider another example of heap corruption or race condition overwriting *ProcessLocksList* or *CriticalSection* pointer. Then we have another instance of **Wild Pointer** pattern illustrated below:

```
0:000> !locks

CritSec ntdll!LdrpLoaderLock+0 at 7c8877a0
WaiterWoken      No
LockCount        0
RecursionCount   1
OwningThread    1184
EntryCount       0
ContentionCount b04707
*** Locked

CritSec +1018de08 at 1018de08
WaiterWoken      Yes
LockCount        -49153
RecursionCount   5046347
OwningThread    460050
EntryCount       0
ContentionCount 0
*** Locked

CritSec +1018ddd8 at 1018ddd8
WaiterWoken      Yes
LockCount        -1
RecursionCount   0
OwningThread    0
EntryCount       0
ContentionCount 0
*** Locked

CritSec +1018de28 at 1018de28
WaiterWoken      Yes
LockCount        -1
RecursionCount   0
OwningThread    460050
EntryCount       0
ContentionCount 0
*** Locked

CritSec +1018de08 at 1018de08
WaiterWoken      Yes
LockCount        -49153
RecursionCount   5046347
OwningThread    460050
EntryCount       0
ContentionCount 0
*** Locked

CritSec +1018de28 at 1018de28
WaiterWoken      Yes
LockCount        -1
RecursionCount   0
OwningThread    0
EntryCount       0
ContentionCount 0
*** Locked
```

```
CritSec +1018ddd8 at 1018ddd8
WaiterWoken      Yes
LockCount        -1
RecursionCount   0
OwningThread    0
*** Locked
```

```
Scanned 841 critical sections
```

We see the signs of corruption at 1018de08 address which is interpreted as pointing to a locked critical section. To see where the corruption started we need to look at the list of all critical sections either locked or not locked:

```
0:000> !locks -v

CritSec ntdll!RtlCriticalSectionLock+0 at 7c887780
LockCount      NOT LOCKED
RecursionCount 0
OwningThread   0
EntryCount     0
ContentionCount 28

CritSec ntdll!LdrpLoaderLock+0 at 7c8877a0
WaiterWoken    No
LockCount      0
RecursionCount 1
OwningThread   1184
EntryCount     0
ContentionCount b04707
*** Locked

CritSec ntdll!FastPebLock+0 at 7c887740
LockCount      NOT LOCKED
RecursionCount 0
OwningThread   0
EntryCount     0
ContentionCount 42c9

CritSec ntdll!RtlpCalloutEntryLock+0 at 7c888ea0
LockCount      NOT LOCKED
RecursionCount 0
OwningThread   0
EntryCount     0
ContentionCount 0

CritSec ntdll!PMCritSect+0 at 7c8883c0
LockCount      NOT LOCKED
RecursionCount 0
OwningThread   0
EntryCount     0
ContentionCount 0

CritSec ntdll!UMLogCritSect+0 at 7c888400
LockCount      NOT LOCKED
RecursionCount 0
OwningThread   0
```

```
EntryCount      0
ContentionCount 0

CriticalSection ntdll!RtlpProcessHeapsListLock+0 at 7c887960
LockCount       NOT LOCKED
RecursionCount   0
OwningThread     0
EntryCount       0
ContentionCount  0

CriticalSection +80608 at 00080608
LockCount       NOT LOCKED
RecursionCount   0
OwningThread     0
EntryCount       0
ContentionCount  22

...

CriticalSection cabinet!_adbmsg+13c at 74fb4658
LockCount       NOT LOCKED
RecursionCount   0
OwningThread     0
EntryCount       0
ContentionCount  0

CriticalSection +c6c17c at 00c6c17c
LockCount       NOT LOCKED
RecursionCount   0
OwningThread     0
EntryCount       0
ContentionCount  0

CriticalSection +c6c0e4 at 00c6c0e4
LockCount       NOT LOCKED
RecursionCount   0
OwningThread     0
EntryCount       0
ContentionCount  0

CriticalSection at 1018de08 does not point back to the debug info at 00136a40
Perhaps the memory that held the critical section has been reused without calling DeleteCriticalSection()
?

CriticalSection +1018de08 at 1018de08
WaiterWoken      Yes
LockCount        -49153
RecursionCount   5046347
OwningThread     460050
EntryCount       0
ContentionCount  0
*** Locked

CriticalSection at 1018ddd8 does not point back to the debug info at 00136a68
Perhaps the memory that held the critical section has been reused without calling DeleteCriticalSection()
?
```

```
CritSec +1018ddd8 at 1018ddd8
WaiterWoken      Yes
LockCount        -1
RecursionCount   0
OwningThread    0
*** Locked
```

...

We see that the problem appears when the heap-allocated critical section at 00c6c0e4 address is linked to an inconsistent critical section at the 0x1018de08 address where its memory data contains UNICODE string fragment:

```
0:000> !address 00c6c0e4
00c60000 : 00c60000 - 00010000
          Type      00020000 MEM_PRIVATE
          Protect   00000004 PAGE_READWRITE
          State     00001000 MEM_COMMIT
          Usage     RegionUsageHeap
          Handle    00c60000

0:000> dt -r1 _RTL_CRITICAL_SECTION 00c6c0e4
+0x000 DebugInfo      : 0x00161140 _RTL_CRITICAL_SECTION_DEBUG
+0x000 Type           : 0
+0x002 CreatorBackTraceIndex : 0
+0x004 CriticalSection : 0x00c6c0e4 _RTL_CRITICAL_SECTION
+0x008 ProcessLocksList : _LIST_ENTRY [ 0x136a48 - 0x119f58 ]
+0x010 EntryCount     : 0
+0x014 ContentionCount : 0
+0x018 Spare          : [2] 0
+0x004 LockCount      : -1
+0x008 RecursionCount : 0
+0x00c OwningThread   : (null)
+0x010 LockSemaphore  : (null)
+0x014 SpinCount      : 0

0:000> dt -r _RTL_CRITICAL_SECTION_DEBUG 0x00136a48-0x008
+0x000 Type           : 0
+0x002 CreatorBackTraceIndex : 0
+0x004 CriticalSection : 0x1018de08 _RTL_CRITICAL_SECTION
+0x000 DebugInfo      : 0x000d001b _RTL_CRITICAL_SECTION_DEBUG
+0x000 Type           : 0
+0x002 CreatorBackTraceIndex : 0
+0x004 CriticalSection : (null)
+0x008 ProcessLocksList : _LIST_ENTRY [ 0x0 - 0x0 ]
+0x010 EntryCount     : 0
+0x014 ContentionCount : 0x37e3c700
+0x018 Spare          : [2] 0x8000025
+0x004 LockCount      : 196609
+0x008 RecursionCount : 5046347
+0x00c OwningThread   : 0x00460050
+0x010 LockSemaphore  : 0x00310033
+0x014 SpinCount      : 0x520044
+0x008 ProcessLocksList : _LIST_ENTRY [ 0x136a70 - 0x161148 ]
+0x000 Flink           : 0x00136a70 _LIST_ENTRY [ 0x136a98 - 0x136a48 ]
+0x000 Flink           : 0x00136a98 _LIST_ENTRY [ 0x136ae8 - 0x136a70 ]
```

```

+0x004 Blink      : 0x00136a48 _LIST_ENTRY [ 0x136a70 - 0x161148 ]
+0x004 Blink      : 0x00161148 _LIST_ENTRY [ 0x136a48 - 0x119f58 ]
+0x000 Flink      : 0x00136a48 _LIST_ENTRY [ 0x136a70 - 0x161148 ]
+0x004 Blink      : 0x00119f58 _LIST_ENTRY [ 0x161148 - 0x16cc3c0 ]
+0x010 EntryCount : 0
+0x014 ContentionCount : 0
+0x018 Spare      : [2] 0x2e760000

0:000> !address 0x1018de08
10120000 : 10120000 - 00100000
    Type      00020000 MEM_PRIVATE
    Protect   00000004 PAGE_READWRITE
    State     00001000 MEM_COMMIT
    Usage     RegionUsageIsVAD

```

The address points miraculously to some DLL:

```

0:000> du 1018de08
1018de08  "....componentA.dll"

```

We might suggest that componentA.dll played some role there.

There are other messages from verbose version of **!locks** WinDbg command pointing to critical section problems:

```

CritSec componentB!Section+0 at 74004008
LockCount      NOT LOCKED
RecursionCount 0
OwningThread   0
EntryCount     0
ContentionCount 0

The CritSec componentC!Info+c at 72455074 has been RE-INITIALIZED.
The critical section points to DebugInfo at 00107cc8 instead of 000f4788

CritSec componentC!Info+c at 72455074
LockCount      NOT LOCKED
RecursionCount 0
OwningThread   0
EntryCount     0
ContentionCount 0

CritSec componentD!foo+8ec0 at 0101add0
LockCount      NOT LOCKED
RecursionCount 0
OwningThread   0
EntryCount     0
ContentionCount 0

```

## Critical Stack Trace

This pattern addresses abnormal behavior such as the page fault processing or any other critical system activity that is waiting for too long. Such activity is either finishes quickly or leads to normal bugcheck processing code. For example, this thread is stuck in the page fault processing for 32 minutes while loading a resource:

```
THREAD ffffffa80f0603c00 Cid 376.3d6 Peb: 000007fffffd6000 Win32Thread: ffffff900c09e0640 WAIT:
(Executive) KernelMode Non-Alertable
[...]
Wait Start TickCount      6281298           Ticks: 123391 (0:00:32:04.102)
[...]
Child-SP      RetAddr          Call Site
ffffff880`3fc99030 ffffff800`01882bd2 nt!KiSwapContext+0x7a
ffffff880`3fc99170 ffffff800`01893f8f nt!KiCommitThreadWait+0x1d2
ffffff880`3fc99200 ffffff880`016283ff nt!KeWaitForSingleObject+0x19f
ffffff880`3fc992a0 ffffff880`01620fc6 Ntfs!NtfsNonCachedIo+0x23f
ffffff880`3fc99470 ffffff880`01622a68 Ntfs!NtfsCommonRead+0x7a6
ffffff880`3fc99610 ffffff880`00fb4bcf Ntfs!NtfsFsdRead+0x1b8
ffffff880`3fc99820 ffffff880`00fb36df fltmgr!FltpLegacyProcessingAfterPreCallbacksCompleted+0x24f
ffffff880`3fc998b0 ffffff800`018b44f5 fltmgr!FltpDispatch+0xcf
ffffff880`3fc999a0 ffffff800`018b3fc9 nt!IoPageRead+0x255
ffffff880`3fc99a30 ffffff800`0189a85a nt!MiIssueHardFault+0x255
ffffff880`3fc99ac0 ffffff800`0188b2ee nt!MmAccessFault+0x146a
ffffff880`3fc99c20 00000000`779da643 nt!KiPageFault+0x16e (TrapFrame @ ffffff880`3fd99c20)
00000000`039ff4f0 00000000`779d8b1e ntdll!LdrpGetRcConfig+0xcd
00000000`039ff580 00000000`779da222 ntdll!LdrIsResItemExist+0x1e
00000000`039ff5c0 00000000`779f82c4 ntdll!LdrpSearchResourceSection_U+0xa4
00000000`039ff6e0 000007fe`fe0075c1 ntdll!LdrFindResource_U+0x44
00000000`039ff720 000007fe`fb217777 KERNELBASE!FindResourceExW+0x85
[...]
```

The **Top Module** (page 1012) and **Blocking Module** (page 96) is NTFS, so we might want to look for other similar stack traces from **Stack Trace Collection** (page 943).

## Custom Exception Handler

### Kernel Space

This is a kernel space counterpart to **Custom Exception Handler** pattern in user space (page 171). In the following stack trace below we see that *DriverA* code intercepted an access violation exception resulted from dereferencing **NULL Pointer** (page 752) and generated a custom bugcheck:

```
kd> !analyze -v

[...]

EXCEPTION_RECORD: fffff8801c757158 -- (.exr 0xfffff8801c757158)
ExceptionAddress: fffff88003977de1 (DriverA!foo+0x0000000000000381)
ExceptionCode: c0000005 (Access violation)
ExceptionFlags: 00000000
NumberParameters: 2
Parameter[0]: 0000000000000000
Parameter[1]: 0000000000000070
Attempt to read from address 0000000000000070

TRAP_FRAME: fffff8801c757200 -- (.trap 0xfffff8801c757200)
NOTE: The trap frame does not contain all registers.
Some register values may be zeroed or incorrect.
rax=0000000000000000 rbx=0000000000000000 rcx=fffff8a00da3f3c0
rdx=0000000000000000 rsi=0000000000000000 rdi=0000000000000000
rip=fffff88003977de1 rsp=fffff8801c757390 rbp=fffffa8009a853f0
r8=0000000000000000 r9=0000000000000000 r10=006800740020006e
r11=fffff8a00da3f3c6 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0 nv up ei pl zr na po nc
DriverA!foo+0x381:
fffff880`03977de1 0fb74070 movzx eax,word ptr [rax+70h] ds:0703:0070=?????
Resetting default scope

[...]

kd> kL 100
Child-SP RetAddr Call Site
fffff880`1c7560f8 fffff880`039498f7 nt!KeBugCheckEx
fffff880`1c756100 fffff880`039352a0 DriverA!MyBugCheckEx+0x93
fffff880`1c756140 fffff800`016f1d1c DriverA!MyExceptionFilter+0x1d0
fffff880`1c756210 fffff800`016e940d nt!_C_specific_handler+0x8c
fffff880`1c756280 fffff800`016f0a90 nt!RtlpExecuteHandlerForException+0xd
fffff880`1c7562b0 fffff800`016fd9ef nt!RtlDispatchException+0x410
fffff880`1c756990 fffff800`016c2d82 nt!KiDispatchException+0x16f
fffff880`1c757020 fffff800`016c18fa nt!KiExceptionDispatch+0xc2
fffff880`1c757200 fffff880`03977de1 nt!KiPageFault+0x23a
fffff880`1c757390 fffff880`03977754 DriverA!foo+0x381
fffff880`1c757430 fffff880`0396f006 DriverA!bar+0x74
[...]
fffff880`1c7579b0 fffff800`019a6e0a DriverA!QueryInformation+0x30b
```

```
fffff880`1c757a70 fffff800`016c2993 nt!NtQueryInformationFile+0x535
fffff880`1c757bb0 00000000`76e5fe6a nt!KiSystemServiceCopyEnd+0x13
00000000`0a08dfe8 00000000`00000000 0x76e5fe6a

kd> !exchain
24 stack frames, scanning for handlers...
Frame 0x05: nt!RtlpExecuteHandlerForException+0xd (fffff800`016e940d)
ehandler nt!RtlpExceptionHandler (fffff800`016e93d0)
Frame 0x07: nt!KiDispatchException+0x16f (fffff800`016fd9ef)
ehandler nt!_GSHandlerCheck_SEH (fffff800`0169aec0)
Frame 0x0b: DriverA!bar+0x74 (fffff880`03977754)
ehandler DriverA!__GSHandlerCheck (fffff880`039a12fc)
[...]
Frame 0x14: DriverA!QueryInformation+0x30b (fffff880`039303ab)
ehandler DriverA!_C_specific_handler (fffff880`039a1864)
Frame 0x15: nt!NtQueryInformationFile+0x535 (fffff800`019a6e0a)
ehandler nt!_C_specific_handler (fffff800`016f1c90)
Frame 0x16: nt!KiSystemServiceCopyEnd+0x13 (fffff800`016c2993)
ehandler nt!KiSystemServiceHandler (fffff800`016c2580)
Frame 0x17: error getting module for 0000000076e5fe6a
```

## User Space

As discussed in **Early Crash Dump** pattern (page 312) saving crash dumps on first-chance exceptions helps to diagnose components that might have caused corruption and later crashes, hangs or CPU spikes by ignoring abnormal exceptions like access violation. In such cases, we need to know whether an application installs its own **Custom Exception Handler** or several of them. If it uses only default handlers provided by runtime or windows subsystem then most likely a first-chance access violation exception will result in a last-chance exception and a postmortem dump. To check a chain of exception handlers, we can use WinDbg !exchain extension command. For example:

```
0:000> !exchain
0017f9d8: TestDefaultDebugger!AfxWinMain+3f5 (00420aa9)
0017fa60: TestDefaultDebugger!AfxWinMain+34c (00420a00)
0017fb20: user32!_except_handler4+0 (770780eb)
0017fcc0: user32!_except_handler4+0 (770780eb)
0017fd24: user32!_except_handler4+0 (770780eb)
0017fe40: TestDefaultDebugger!AfxWinMain+16e (00420822)
0017feec: TestDefaultDebugger!AfxWinMain+797 (00420e4b)
0017ff90: TestDefaultDebugger!_except_handler4+0 (00410e00)
0017ffd8: ntdll!_except_handler4+0 (77961c78)
```

We see that *TestDefaultDebugger* doesn't have its own exception handlers except ones provided by MFC and C/C++ runtime libraries which were linked statically. Here is another example. It was reported that a 3rd-party application was hanging and spiking CPU (**Spiking Thread** pattern, page 888), so a user dump was saved using command line *userdump.exe*:

```
0:000> vertarget
Windows Server 2003 Version 3790 (Service Pack 2) MP (4 procs) Free x86 compatible
Product: Server, suite: TerminalServer
kernel32.dll version: 5.2.3790.4062 (srv03_sp2_gdr.070417-0203)
Debug session time: Thu Nov 22 12:45:59.000 2007 (GMT+0)
System Uptime: 0 days 10:43:07.667
Process Uptime: 0 days 4:51:32.000
Kernel time: 0 days 0:08:04.000
User time: 0 days 0:23:09.000

0:000> !runaway 3
User Mode Time
Thread Time
0:1c1c      0 days 0:08:04.218
1:2e04      0 days 0:00:00.015
Kernel Mode Time
Thread Time
0:1c1c      0 days 0:23:09.156
1:2e04      0 days 0:00:00.031
```

```
0:000> kL
ChildEBP RetAddr
0012fb80 7739bf53 ntdll!KiFastSystemCallRet
0012fbb4 05ca73b0 user32!NtUserWaitMessage+0xc
WARNING: Stack unwind information not available. Following frames may be wrong.
0012fd20 05c8be3f 3rdPartyDLL+0x573b0
0012fd50 05c9e9ea 3rdPartyDLL+0x3be3f
0012fd68 7739b6e3 3rdPartyDLL+0x4e9ea
0012fd94 7739b874 user32!InternalCallWinProc+0x28
0012fe0c 7739c8b8 user32!UserCallWinProcCheckWow+0x151
0012fe68 7739c9c6 user32!DispatchClientMessage+0xd9
0012fe90 7c828536 user32!__fnDWORD+0x24
0012feb2 7739d1ec ntdll!KiUserCallbackDispatcher+0x2e
0012fef8 7738cee9 user32!NtUserMessageCall+0xc
0012ff18 0050aea9 user32!SendMessageA+0x7f
0012ff70 00452ae4 3rdPartyApp+0x10aea9
0012ffac 00511941 3rdPartyApp+0x52ae4
0012ffc0 77e6f23b 3rdPartyApp+0x111941
0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

Exception chain listed **Custom Exception Handlers**:

```
0:000> !exchain
0012fb8c: 3rdPartyDLL+57acb (05ca7acb)
0012fd28: 3rdPartyDLL+3be57 (05c8be57)
0012fd34: 3rdPartyDLL+3be68 (05c8be68)
0012fdfc: user32!_except_handler3+0 (773aaaf18)
    CRT scope 0, func: user32!UserCallWinProcCheckWow+156 (773ba9ad)
0012fe58: user32!_except_handler3+0 (773aaaf18)
0012fea0: ntdll!KiUserCallbackExceptionHandler+0 (7c8284e8)
0012ff3c: 3rdPartyApp+53310 (00453310)
0012ff48: 3rdPartyApp+5334b (0045334b)
0012ff9c: 3rdPartyApp+52d06 (00452d06)
0012ffb4: 3rdPartyApp+38d4 (004038d4)
0012ffe0: kernel32!_except_handler3+0 (77e61a60)
    CRT scope 0, filter: kernel32!BaseProcessStart+29 (77e76a10)
        func: kernel32!BaseProcessStart+3a (77e81469)
```

The customer then enabled MS Exception Monitor and selected only “Access violation exception code” (c0000005) to avoid **False Positive Dumps** (page 393). During application execution various 1st-chance exception crash dumps were saved pointing to numerous access violations including function calls into unloaded modules, for example:

```
0:000> kL 100
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
0012f910 7739b6e3 <Unloaded_Another3rdParty.dll>+0x4ce58
0012f93c 7739b874 user32!InternalCallWinProc+0x28
0012f9b4 7739c8b8 user32!UserCallWinProcCheckWow+0x151
0012fa10 7739c9c6 user32!DispatchClientMessage+0xd9
0012fa38 7c828536 user32!__fnDWORD+0x24
0012fa64 7739d1ec ntdll!KiUserCallbackDispatcher+0x2e
0012faa0 7738cee9 user32!NtUserMessageCall+0xc
0012fac0 0a0f2e01 user32!SendMessageA+0x7f
0012fae4 0a0f2ac7 3rdPartyDLL+0x52e01
0012fb60 7c81a352 3rdPartyDLL+0x52ac7
0012fb80 7c839dee ntdll!LdrpCallInitRoutine+0x14
0012fc94 77e6b1bb ntdll!LdrUnloadDll+0x41a
0012fca8 0050c9c1 kernel32!FreeLibrary+0x41
0012fdf4 004374af 3rdPartyApp+0x10c9c1
0012fe24 0044a076 3rdPartyApp+0x374af
0012fe3c 7739b6e3 3rdPartyApp+0x4a076
0012fe68 7739b874 user32!InternalCallWinProc+0x28
0012fee0 7739ba92 user32!UserCallWinProcCheckWow+0x151
0012ff48 773a16e5 user32!DispatchMessageWorker+0x327
0012ff58 00452aa0 user32!DispatchMessageA+0xf
0012ffac 00511941 3rdPartyApp+0x52aa0
0012ffc0 77e6f23b 3rdPartyApp+0x111941
0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

**D****Data Alignment****Page Boundary**

Most of the time, this pattern manifests itself on Intel platforms from performance perspective and via GP faults for some instructions that require natural boundary for their qword operands. There are no exceptions generally if we move a dword value from or to an odd memory location address when the whole operand fits into one page. However, we need to take the possibility of the page boundary spans into account when checking memory addresses for their validity. Consider this exception:

```
0: kd> .trap 0xfffffffffa38df520
ErrCode = 00000002
eax=b6d9220f ebx=b6ab4ffb ecx=00000304 edx=eaf2fdea esi=b6d9214c edi=b6ab8189
eip=bfa10e6e esp=a38df594 ebp=a38df5ac iopl=0 nv up ei ng nz ac po cy
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010293
driver+0x2ae6e:
bfa10e6e 895304 mov     dword ptr [ebx+4],edx ds:0023:b6ab4fff=????????
```

The address seems to be valid:

```
0: kd> !pte b6ab4fff
      VA b6ab4fff
PDE at    C0300B68          PTE at C02DAAD0
contains 7F0DD863          contains 426B0863
pfn 7f0dd -DA-KWEV      pfns 426b0 -DA-KWEV
```

But a careful examination of the instruction reveals that it writes 32-bit value, so we need to inspect the next byte too because it is on another page:

```
0: kd> !pte b6ab4fff+1
      VA b6ab5000
PDE at    C0300B68          PTE at C02DAAD4
contains 7F0DD863          contains 00000080
pfns 7f0dd -DA-KWEV      not valid
                           DemandZero
                           Protect: 4 - ReadWrite
```

Although the page is demand zero and this should have been satisfied by creating a new page filled with zeroes, the point here is that the page could have been completely invalid or paged out in the case of IRQL  $\geq 2$ .

## Data Contents Locality

This is a comparative pattern that helps not only in identifying the class of the problem but increases our confidence and degree of belief in the specific hypothesis. Suppose we have a database of notes on previous problems. If we see the same or similar data accessed in the new memory dump, we may suppose that the issue is similar. If **Data Contents Locality** is complemented by **Code Path Locality**<sup>30</sup> (similar partial stack traces and code **Execution Residues**, page 371) it even greater boosts our confidence in suggesting specific troubleshooting steps, recommending fixes and service packs or routing the problem to the next support or development service supply chain (like escalating the issue).

Suppose we got a new kernel memory dump with IRQL\_NOT\_LESS\_OR\_EQUAL (A) bugcheck pointing to our module, and we notice the write access to a structure in the nonpaged pool having specific pool tag:

```
3: kd> .trap 9ee8d9b0
ErrCode = 00000002
eax=85407650 ebx=858f6650 ecx=ffffffff edx=85407648 esi=858f65a8 edi=858f6620
eip=8083df4c esp=9ee8da24 ebp=9ee8da64 iopl=0 nv up ei pl zr na pe nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010246
nt!KeWaitForSingleObject+0x24f:
8083df4c 8919    mov     dword ptr [ecx],ebx  ds:0023:ffffffffff=?????????

STACK_TEXT:
9ee8d9b0 8083df4c badb0d00 85407648 00000000 nt!KiTrap0E+0x2a7
9ee8da64 80853f3f 85407648 00000001 nt!KeWaitForSingleObject+0x24f
9ee8da7c 8081d45f 865b18d8 854076b0 f4b9e53b nt!KiAcquireFastMutex+0x13
9ee8da88 f4b9e53b 00000004 86940110 85407638 nt!ExAcquireFastMutex+0x20
9ee8daa8 f4b9ed98 85407638 00000000 86940110 driver!Query+0x143
...
3: kd> !pool 85407648
Pool page 85407648 region is Nonpaged pool
85407000 size: 80 previous size: 0 (Allocated) Mdl
85407080 size: 30 previous size: 80 (Allocated) Even (Protected)
854070b0 size: 28 previous size: 30 (Allocated) Ntfn
854070d8 size: 28 previous size: 28 (Allocated) NtFs
85407100 size: 28 previous size: 28 (Allocated) Ntfn
...
85407570 size: 28 previous size: 70 (Allocated) Ntfn
85407598 size: 98 previous size: 28 (Allocated) File (Protected)
*85407630 size: b0 previous size: 98 (Free ) *DrvA
```

<sup>30</sup> This is a pattern we may add in the future

Dumping the memory address passed to *KeWaitForSingleObject* function shows simple but a peculiar pattern:

```
3: kd> dd 85407648
85407648  ffffffff ffffffff ffffffff ffffffff
85407658  ffffffff ffffffff ffffffff ffffffff
85407668  ffffffff ffffffff ffffffff ffffffff
85407678  ffffffff ffffffff ffffffff ffffffff
85407688  ffffffff ffffffff ffffffff ffffffff
85407698  ffffffff ffffffff ffffffff ffffffff
854076a8  ffffffff ffffffff ffffffff ffffffff
854076b8  ffffffff ffffffff ffffffff ffffffff
```

We find several similar cases in our database but with different overall call stacks, except the topmost wait call. Then we notice that in previous cases there were mutants associated with their thread structure, and we have the same now:

```
0: kd> !thread
THREAD 858f65a8 Cid 474c.4530 Teb: 7ffdf000 Win32Thread: bc012410 RUNNING on processor 0
...
3: kd> dt /r _KTHREAD 858f65a8 MutantListHead
nt!_KTHREAD
+0x010 MutantListHead : _LIST_ENTRY [ 0x86773040 - 0x86773040 ]
3: kd> !pool 86773040
Pool page 86773040 region is Nonpaged pool
*86773000 size: 50 previous size: 0 (Allocated) *Muta (Protected)
  Pooltag Muta : Mutant objects
...
```

This narrows the issue to only a few previous cases. In one previous case, *WaitBlockList* associated with a thread structure had 0xffffffff in its pointers. Our block shows the same pattern:

```
0: kd> dt -r _KTHREAD 858f65a8 WaitBlockList
nt!_KTHREAD
+0x054 WaitBlockList : 0x858f6650 _KWAIT_BLOCK

0: kd> dt _KWAIT_BLOCK 0x858f6650
nt!_KWAIT_BLOCK
+0x000 WaitListEntry : _LIST_ENTRY [ 0x85407650 - 0xffffffff ]
+0x008 Thread : 0x858f65a8 _KTHREAD
+0x00c Object : 0x85407648
+0x010 NextWaitBlock : 0x858f6650 _KWAIT_BLOCK
+0x014 WaitKey : 0
+0x016 WaitType : 0x1 "
+0x017 SpareByte : 0 "
```

We have probably narrowed down the issue to a specific case. Although this doesn't work always and mostly based on intuition, there are spectacular cases where it really helps in troubleshooting. Here is another example where the contents of EDI register from exception context provided specific recommendation hints. When looking at the crash point, we see an instance of **Wild Code** pattern (page 1148):

```

0:000> kv
ChildEBP RetAddr  Args to Child
WARNING: Frame IP not in any known module. Following frames may be wrong.
49ab5bba 00000000 00000000 00000000 0x60f1011a

0:000> r
eax=38084fff ebx=52303340 ecx=963f1416 edx=0000063d esi=baaff395 edi=678c5804
eip=60f1011a esp=5a9d0f48 ebp=49ab5bba iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00210206
60f1011a cd01      int     1

0:000> u
60f1011a cd01      int     1
60f1011c cc          int     3
60f1011d 8d          ????
60f1011e c0eb02      shr     bl,2
60f10121 0f840f31cd01 je      62be3236
60f10127 8d          ????
60f10128 c0cc0f      ror     ah,0Fh
60f1012b 0bce        or      ecx,esi

```

Looking at raw stack data, we notice the presence of a specific component that is known to patch the process import table. Applying techniques outlined in **Hooked Functions** pattern (page 488) we notice two different 3rd-party components that patched two different modules (*kernel32* and *user32*):

```

0:000> !chkimg -lo 50 -d !kernel32 -v
Searching for module with expression: !kernel32
Will apply relocation fixups to file used for comparison
Will ignore NOP/LOCK errors
Will ignore patched instructions
Image specific ignores will be applied
Comparison image path: c:\mss\kernel32.dll\4626487F102000\kernel32.dll
No range specified

Scanning section:    .text
Size: 564709
Range to scan: 77e41000-77ecade5
 77e41ae5-77e41ae9 5 bytes - kernel32!LoadLibraryExW
 [ 6a 34 68 48 7b:e9 16 e5 f4 07 ]
 77e44a8a-77e44a8e 5 bytes - kernel32!WaitNamedPipeW (+0x2fa5)
 [ 8b ff 55 8b ec:e9 71 b5 f9 07 ]
 77e5106a-77e5106e 5 bytes - kernel32!CreateProcessInternalW (+0xc5e0)
...
Total bytes compared: 564709(100%)
Number of errors: 49
49 errors : !kernel32 (77e41ae5-77e9aa16)

```

```

0:000> u 77e41ae5
kernel32!LoadLibraryExW:
77e41ae5 jmp      7fd90000
77e41aea out     77h,al
77e41aec call    kernel32!_SEH_prolog (77e6b779)
77e41af1 xor     edi,edi
77e41af3 mov     dword ptr [ebp-28h],edi
77e41af6 mov     dword ptr [ebp-2Ch],edi
77e41af9 mov     dword ptr [ebp-20h],edi
77e41afc cmp     dword ptr [ebp+8],edi

0:000> u 7fd90000
*** ERROR: Symbol file could not be found. Defaulted to export symbols for ComponentA.dll -
7fd90000 jmp      ComponentA!DllUnregisterServer+0x2700 (678c4280)
7fd90005 push    34h
7fd90007 push    offset kernel32!`string'+0xc (77e67b48)
7fd9000c jmp     kernel32!LoadLibraryExW+0x7 (77e41aec)
7fd90011 add     byte ptr [eax],al
7fd90013 add     byte ptr [eax],al
7fd90015 add     byte ptr [eax],al
7fd90017 add     byte ptr [eax],al

0:000> !chkimg -lo 50 -d !user32 -v
Searching for module with expression: !user32
Will apply relocation fixups to file used for comparison
Will ignore NOP/LOCK errors
Will ignore patched instructions
Image specific ignores will be applied
Comparison image path: c:\mss\user32.dll\45E7BFD692000\user32.dll
No range specified

Scanning section: .text
Size: 396943
Range to scan: 77381000-773e1e8f
77383f38-77383f3c 5 bytes - user32!EnumDisplayDevicesW
[ 8b ff 55 8b ec:e9 c3 c0 82 08 ]
  77384406-7738440a 5 bytes - user32!EnumDisplaySettingsExW (+0x4ce)
[ 8b ff 55 8b ec:e9 f5 bb 7e 08 ]
  773844d9-773844dd 5 bytes - user32!EnumDisplaySettingsW (+0xd3)
[ 8b ff 55 8b ec:e9 22 bb 80 08 ]
  7738619b-7738619f 5 bytes - user32!EnumDisplayDevicesA (+0x1cc2)
[ 8b ff 55 8b ec:e9 60 9e 83 08 ]
  7738e985-7738e989 5 bytes - user32!CreateWindowExA (+0x87ea)
[ 8b ff 55 8b ec:e9 76 16 8c 08 ]
...
Total bytes compared: 396943(100%)
Number of errors: 119
119 errors : !user32 (77383f38-773c960c)

```

```

0:000> u 77383f38
user32!EnumDisplayDevicesW:
77383f38 e9c3c08208      jmp    7fb0000
77383f3d 81ec58030000    sub    esp,358h
77383f43 a1ac243e77      mov    eax,dword ptr [user32!__security_cookie (773e24ac)]
77383f48 8b5508          mov    edx,dword ptr [ebp+8]
77383f4b 83a5acfccfff00 and   dword ptr [ebp-354h],0
77383f52 53              push   ebx
77383f53 56              push   esi
77383f54 8b7510          mov    esi,dword ptr [ebp+10h]

0:000> u 7fb0000
*** ERROR: Symbol file could not be found. Defaulted to export symbols for ComponentB.dll -
7fb0000 e91b43d5e5      jmp    ComponentB+0x4320 (65904320)
7fb0005 8bff            mov    edi,edi
7fb0007 55              push   ebp
7fb0008 8bec            mov    ebp,esp
7fb000a e92e3f7df7      jmp    user32!EnumDisplayDevicesW+0x5 (77383f3d)
7fb000f 0000            add    byte ptr [eax],al
7fb0011 0000            add    byte ptr [eax],al
7fb0013 0000            add    byte ptr [eax],al

```

Which one should we try to eliminate first to test our assumption that they somehow resulted in application faults? Looking at register context again we see that one specific register (EDI) has a value that lies in *ComponentA* address range:

```

0:000> r
eax=38084ff0 ebx=52303340 ecx=963f1416 edx=0000063d esi=baaff395 edi=678c5804
eip=60f1011a esp=5a9d0f48 ebp=49ab5bba iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00210206
60f1011a cd01          int     1

0:000> lm
start   end     module name
00400000 01901000 Application
...
678c0000 6791d000 ComponentA ComponentA.DLL
...

```

## Data Correlation

### Function Parameters

This is a general pattern where values found in different parts of a memory dump correlate between each other according to some rules, for example, in some proportion. Here we show a variant for function parameters.

A process user memory dump had **C++ Exception** (page 108) inside:

```
0:000> kL
*** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr
0012e950 78158e89 kernel32!RaiseException+0x53
0012e988 7830770c msrvcr80!_CxxThrowException+0x46
0012e99c 783095bc mfc80u!AfxThrowMemoryException+0x19
0012e9b4 02afa8ca mfc80u!operator new+0x27
0012e9c8 02b0992f ModuleA!std::_Allocate<...>+0x1a
0012e9e0 02b09e7c ModuleA!std::vector<double,std::allocator<double>
>::vector<double,std::allocator<double>>+0x3f
[...]
```

We suspected an out-of-memory condition and looked for function parameters:

```
0:000> kv 5
ChildEBP RetAddr Args to Child
0012e950 78158e89 e06d7363 00000001 00000003 kernel32!RaiseException+0x53
0012e988 7830770c 0012e998 783b0110 783c8d68 msrvcr80!_CxxThrowException+0x46
0012e99c 783095bc 0000a7c0 0012ea40 000014f8 mfc80u!AfxThrowMemoryException+0x19
0012e9b4 02afa8ca 0000a7c0 089321b0 089321f0 mfc80u!operator new+0x27 (FPO: [Uses EBP] [1,0,0])
0012e9c8 02b0992f 000014f8 00000000 00000008 ModuleA!std::_Allocate<...>+0x1a (FPO: [2,3,0])
```

Because of **Frame Pointer Omission** (page 403) we originally thought that stack arguments would be invalid. However, knowing the function prototype and semantics of operator *new*<sup>31</sup> and *std::vector double* element type we immediately see the correlation between 0xa7c0 and 0x14f8 which are proportional to *sizeof(double)* == 8:

```
0:000> ? 0000a7c0/000014f8
Evaluate expression: 8 = 00000000`00000008
```

---

<sup>31</sup> <https://msdn.microsoft.com/en-us/library/kftdy56f.aspx>

We, therefore, conclude without looking at the disassembly that memory allocation size was 42944 bytes:

```
0:000> .formats 0000a7c0
Evaluate expression:
Hex: 00000000`0000a7c0
Decimal: 42944
Octal: 000000000000000123700
Binary: 00000000 00000000 00000000 00000000 00000000 10100111 11000000
Chars: .....
Time: Thu Jan 01 11:55:44 1970
Float: low 6.01774e-041 high 0
Double: 2.12172e-319
```

## Deadlock

### Critical Sections

For the high-level explanation of “deadlock” terminology, please refer to the discussion of “hangs” in Memory Dump Analysis Anthology<sup>32</sup>. **Deadlocks** do not only happen with synchronization primitives like mutexes, events or more complex objects (built upon primitives) like critical sections or executive resources (ERESOURCE). They can happen from a high level or systems perspective in inter-process or inter-component communication, for example, mutually waiting on messages: GUI window messages, LPC messages, RPC calls.

How can we see **Deadlocks** in memory dumps? Let’s start with user dumps and critical sections.

First, we would recommend reading the following excellent MSDN article to understand various members of CRITICAL\_SECTION structure<sup>33</sup>.

WinDbg **!locks** command examines process critical section list and displays all locked critical sections, lock count and thread id of current critical section owner. This is the output from a memory dump of hanging Windows print spooler process (*spoolsv.exe*):

```
0:000> !locks
CritSec NTDLL!LoaderLock+0 at 784B0348
LockCount      4
RecursionCount 1
OwningThread   624
EntryCount     6c3
ContentionCount 6c3
*** Locked

CritSec LOCALSPL!SpoolerSection+0 at 76AB8070
LockCount      3
RecursionCount 1
OwningThread   1c48
EntryCount     646
ContentionCount 646
*** Locked
```

If we look at threads #624 and #1c48, we see them mutually waiting for each other:

- TID#**624** owns CritSec **784B0348** and is waiting for CritSec **76AB8070**
- TID#**1c48** owns CritSec **76AB8070** and is waiting for CritSec **784B0348**

<sup>32</sup> Hangs Explained, Memory Dump Analysis Anthology, Volume 1, page 31

<sup>33</sup> Break Free of Code Deadlocks in Critical Sections Under Windows, MSDN Magazine, December 2003 (<https://web.archive.org/web/20150419055323/https://msdn.microsoft.com/en-us/magazine/cc164040.aspx>)

0:000>~\*kv

```
. 12 Id: bc0.624 Suspend: 1 Teb: 7ffd3000 Unfrozen
0000024c 00000000 00000000 NTDLL!ZwWaitForSingleObject+0xb
76ab8000 76a815ef 76ab8070 NTDLL!RtlpWaitForCriticalSection+0x9e
76ab8070 76a844f8 00cd1f38 NTDLL!RtlEnterCriticalSection+0x46
00cd1f38 76a8a1d7 00000000 LOCALSPL!EnterSplSem+0xb
00000000 00000000 00cd1f38 LOCALSPL!FindSpoolerByNameIncRef+0x1f
00000000 777f19bc 00000001 LOCALSPL!LocalGetPrinterDriverDirectory+0xe
00000000 777f19bc 00000001 spoolss!GetPrinterDriverDirectoryW+0x59
00000000 777f19bc 00000001 spoolsrv!YGetPrinterDriverDirectory+0x27
00000000 777f19bc 00000001 WINSPOOL!GetPrinterDriverDirectoryW+0x7b
50000000 00000001 00000000 BRHLUI04+0x14ea
50002ea0 50000000 00000001 BRHLUI04!DllGetClassObject+0x1705
00000000 00000000 000cb570 NTDLL!LdrpRunInitializeRoutines+0x1df
000cc8f8 0288ea30 0288ea38 NTDLL!LdrpLoadDll+0x2e6
000cc8f8 0288ea30 0288ea38 NTDLL!LdrLoadDll+0x17)
000c1258 00000000 00000008 KERNEL32!LoadLibraryExW+0x231
000c150c 0288efd8 00000000 UNIDRVUI!PLoadCommonInfo+0x17e
000c150c 0288efd8 00000007 UNIDRVUI!DwDeviceCapabilities+0x1a
00070000 00071378 00000045 UNIDRVUI!DrvDeviceCapabilities+0x19
```

```
. 13 Id: bc0.1c48 Suspend: 1 Teb: 7ffd2000 Unfrozen
0000010c 00000000 00000000 NTDLL!ZwWaitForSingleObject+0xb
784b0301 78468d38 784b0348 NTDLL!RtlpWaitForCriticalSection+0x9e
784b0348 74fb4344 00000000 NTDLL!RtlEnterCriticalSection+0x46
74fb0000 02c0f2a8 00000000 NTDLL!LdrpGetProcedureAddress+0x122
74fb0000 02c0f2a8 00000000 NTDLL!LdrGetProcedureAddress+0x17
74fb0000 74fb4344 02c0f449 KERNEL32!GetProcAddress+0x41
017924b0 00000000 00000001 ws2_32!CheckForHookersOrChainers+0x1f
00000101 02c0f344 017924b0 ws2_32!WSAStartup+0x10f
00cdf79c 02c0f4f4 76a8c9bc LOCALSPL!GetDNSMachineName+0x1e
00000000 76a8c9bc 780276a2 LOCALSPL!GetPrinterUrl+0x2c
0176f570 ffffffff 01000000 LOCALSPL!UpdateDsSpoolerKey+0x322
0176f570 76a8c9bc 01792b90 LOCALSPL!RecreateDsKey+0x50
00000000 00000002 01792b90 LOCALSPL!SplAddPrinter+0x521
01791faa 0176a684 76a5cd34 WIN32SPL!InternalAddPrinterConnection+0x1b4
01791faa 02c0fa00 02c0fabc WIN32SPL!AddPrinterConnectionW+0x15
00076f1c 02c0fabc 01006873 spoolss!AddPrinterConnectionW+0x49
00076f1c 00000001 77107fb0 spoolsrv!YAddPrinterConnection+0x17
00076f1c 02020202 00000001 spoolsrv!RpcAddPrinterConnection+0xb
01006868 02c0fac0 00000001 rpcrt4!Invoke+0x30
00000000 00000000 000d22c8 rpcrt4!NdrStubCall12+0x655
000d22c8 00076fe0 000d22c8 rpcrt4!NdrServerCall12+0x17
010045fc 000d22c8 02c0fe0c rpcrt4!DispatchToStubInC+0x32
0000002b 00000000 02c0fe0c rpcrt4!RPC_INTERFACE::DispatchToStubWorker+0x100
000d22c8 00000000 02c0fe0c rpcrt4!RPC_INTERFACE::DispatchToStub+0x5e
000d3210 00076608 813b0013 rpcrt4!LRPC_SCALL::DealWithRequestMessage+0x1dd
000d21d0 02c0fe50 000d3210 rpcrt4!LRPC_ADDRESS::DealWithLRPCRequest+0x10c
770c9ad0 00076608 770cb6d8 rpcrt4!LRPC_ADDRESS::ReceiveLotsaCalls+0x229
00076608 770cb6d8 0288f9a8 rpcrt4!RecvLotsaCallsWrapper+0x9
00074a50 02c0ffec 77e7438b rpcrt4!BaseCachedThreadRoutine+0x11f
00076e68 770cb6d8 0288f9a8 rpcrt4!ThreadStartRoutine+0x18
770d1c54 00076e68 00000000 KERNEL32!BaseThreadStart+0x52
```

This analysis looks pretty simple and easy. What about the kernel and complete memory dumps? Of course, we cannot see user space critical sections in kernel memory dumps but we can see them in complete memory dumps after switching to the appropriate process context and using `!Intsdexts.locks`. This can be done via a simple script adapted from debugger.chm (see Deadlocks and Critical Sections section there).

Why is it so easy to see deadlocks when critical sections are involved? This is because their structures have a member that records their owner. So it is very easy to map them to corresponding threads. The same is with kernel ERESOURCE synchronization objects. Other objects do not have an owner, for example, in the case of events it is not so easy to find an owner just by looking at an event object. We need to examine thread call stacks, other structures or have access to source code.

There is also `!ics` WinDbg extension where `!ics -l` command lists all locked sections with stack traces and `!ics -t` shows critical section tree. For the latter we need to enable Application Verifier using `gflags.exe` or set 0x100 in the registry for your image:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
GlobalFlag=0x00000100
```

Here is another **Deadlock** example in hanging IE process:

```
0:000> !locks

CritSec ntdll!LdrpLoaderLock+0 at 7c8877a0
WaiterWoken No
LockCount 3
RecursionCount 2
OwningThread d5a8
EntryCount 0
ContentionCount 5a
*** Locked

CritSec shell32!CMountPoint::_csDL+0 at 7cae42d0
WaiterWoken No
LockCount 1
RecursionCount 1
OwningThread b7b4
EntryCount 0
ContentionCount 7
*** Locked

Scanned 1024 critical sections
```

```
0:000> ~*kb 100
```

```
. 0 Id: c068.b7b4 Suspend: 1 Teb: 7ffdd000 Unfrozen
ChildEBP RetAddr Args to Child
0013bd0c 7c827d0b 7c83d236 000001d0 00000000 ntdll!KiFastSystemCallRet
0013bd10 7c83d236 000001d0 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
0013bd4c 7c83d281 000001d0 00000004 00000001 ntdll!RtlpWaitOnCriticalSection+0x1a3
0013bd6c 7c82f20c 7c8877a0 00000000 0013be68 ntdll!RtlEnterCriticalSection+0xa8
0013bda0 7c82f336 00000000 00000000 0013bde8 ntdll!LdrLockLoaderLock+0x133
0013be1c 7c82f2a3 00000001 00000001 00000000 ntdll!LdrGetD1lHandleEx+0x94
0013be38 77e65185 00000001 00000000 0013bea0 ntdll!LdrGetD1lHandle+0x18
0013be84 77e6528f 0013bea0 00000000 7cae2f60 kernel32!GetModuleHandleForUnicodeString+0x20
0013c2fc 77e65155 00000001 00000002 7c8d8828 kernel32!BasepGetModuleHandleExW+0x17f
0013c314 7c91079e 7c8d8828 7c9107b8 0013c350 kernel32!GetModuleHandleW+0x29
0013c31c 7c9107b8 0013c350 7c91078d 00000001 shell32!IsProcessAnExplorer+0xb
0013c324 7c91078d 00000001 7c91373b 00000018 shell32!IsMainShellProcess2+0x46
0013c32c 7c91373b 00000018 00000000 7cae42d0 shell32!_Shell32LoadedInDesktop+0x7
0013c350 7c913776 00000018 00000000 7cae42d0 shell32!CMountPoint::IsNetDriveLazyLoadNetDLLs+0x7b
0013c37c 7c9136dc 00000018 00000001 0013c634 shell32!CMountPoint::GetMountPointDL+0x1c
0013c398 7c96df7 00000018 00000001 00000001 shell32!CMountPoint::GetMountPoint+0x46
0013c5e4 7c90f37d 0018e988 00000001 001a0ea8 shell32!CDrivesFolder::GetAttributesOf+0x7b
0013c624 779cc875 0018e9b0 00000001 04002000 shell32!CRegFolder::GetAttributesOf+0x122
0013c648 779cc917 0018e9b0 001e4dc8 04002000 shdocvw!SHGetAttributes+0x53
0013d728 779cd9c8 0013ddac 00193a50 80004005 shdocvw!CNscTree::_OnCDNotify+0x85
0013d754 779cd964 0013ddac 001a06c8 11281f2a shdocvw!CNscTree::_OnNotify+0x2e1
0013d768 779cd8ff 001a06c8 00010090 0000004e shdocvw!CNscTree::OnWinEvent+0x51
0013d798 75eba756 00193a50 00010090 0000004e shdocvw!CNSCBand::OnWinEvent+0x70
0013d7b8 75eba2a2 00193a50 00010090 0000004e browseui!_FwdWinEvent+0x1d
0013d7ec 75eba357 00010090 0000004e 00000064 browseui!CBandSite::_SendToToolband+0x44
0013d818 75ee2a72 0017de98 00010088 00000000 browseui!CBandSite::OnWinEvent+0x143
0013d864 75ee2b32 0017de98 00010088 0000004e browseui!CBrowserBandSite::OnWinEvent+0x14c
0013d890 75ee2a9a 0000004e 00000064 0013ddac browseui!CBaseBar::_CheckForwardWinEvent+0x88
0013d8ac 75ee29dc 0000004e 00000064 0013ddac browseui!CBaseBar::_OnNotify+0x1c
0013d8c8 75ee2965 00010088 0000004e 00000064 browseui!CBaseBar::v_WndProc+0xd4
0013d918 75ee28fa 00010088 0000004e 00000064 browseui!CDockingBar::v_WndProc+0x447
0013d948 75ee2880 00010088 0000004e 00000064 browseui!CBrowserBar::v_WndProc+0x99
0013d96c 7739b6e3 00010088 0000004e 00000064 browseui!CImplWndProc::s_WndProc+0x65
0013d998 7739b874 75ee2841 00010088 0000004e user32!InternalCallWinProc+0x28
0013da10 7739c2d3 00172e34 75ee2841 00010088 user32!UserCallWinProcCheckWow+0x151
0013da4c 7739c337 006172a0 00618f18 00000064 user32!SendMessageWorker+0x4bd
0013da6c 7743b07f 00010088 0000004e 00000064 user32!SendMessageW+0x7f
0013db04 7743b1ef 0013db1c ffffffff 0013ddac comctl32!CCSendNotify+0xc24
0013db40 774a5ab0 00010088 ffffffff ffffffff4 comctl32!SendNotifyEx+0x57
0013dbac 774a652d 0001008a 0000004e 00000064 comctl32!CReBar::_WndProc+0x257
0013dbd0 7739b6e3 0001008a 0000004e 00000064 comctl32!CReBar::s_WndProc+0x2c
0013dbfc 7739b874 774a6501 0001008a 0000004e user32!InternalCallWinProc+0x28
0013dc74 7739c2d3 00172e34 774a6501 0001008a user32!UserCallWinProcCheckWow+0x151
0013dcbb 7739c337 00617350 0060a9c0 00000064 user32!SendMessageWorker+0x4bd
0013dcfd 7743b07f 0001008a 0000004e 00000064 user32!SendMessageW+0x7f
0013dd68 7743b10d 001c8900 ffffffff4 0013ddac comctl32!CCSendNotify+0xc24
0013dd7c 7748a032 001c8900 00010001 0013ddac comctl32!CICustomDrawNotify+0x2c
0013e070 7748a8bb 001c8900 001d2aa8 01010060 comctl32!TV_DrawItem+0x356
0013e0f4 7748a9ac 00000154 01010060 00000000 comctl32!TV_DrawTree+0x136
0013e158 7745bdd0 001c8900 00000000 0013e21c comctl32!TV_Paint+0x65
0013e1a4 7739b6e3 00010090 0000000f 00000000 comctl32!TV_WndProc+0x6ea
0013e1d0 7739b874 7745b6e6 00010090 0000000f user32!InternalCallWinProc+0x28
```

```

0013e248 7739bfce 0015fce4 7745b6e6 00010090 user32!UserCallWinProcCheckWow+0x151
0013e278 7739bf74 7745b6e6 00010090 0000000f user32!CallWindowProcAorW+0x98
0013e298 77431848 7745b6e6 00010090 0000000f user32!CallWindowProcW+0x1b
0013e2b4 77431b9b 00010090 0000000f 00000000 comctl32!CallOriginalWndProc+0x1a
0013e310 77431d5d 001cf0f8 00010090 0000000f comctl32!CallNextSubclassProc+0x3c
0013e334 779cd761 00010090 0000000f 00000000 comctl32!DefSubclassProc+0x46
0013e350 77431b9b 00010090 0000000f 00000000 shdocvw!CNotifySubclassWndProc::_SubclassWndProc+0xa7
0013e3ac 77431d5d 001cf0f8 00010090 0000000f comctl32!CallNextSubclassProc+0x3c
0013e3d0 779cd86f 00010090 0000000f 00000000 comctl32!DefSubclassProc+0x46
0013e41c 779cd7e4 00010090 0000000f 00000000 shdocvw!CNscTree::_SubClassTreeWndProc+0x3ae
0013e43c 77431b9b 00010090 0000000f 00000000 shdocvw!CNscTree::s_SubClassTreeWndProc+0x34
0013e498 77431dc0 001cf0f8 00010090 0000000f comctl32!CallNextSubclassProc+0x3c
0013e4ec 7739b6e3 00010090 0000000f 00000000 comctl32!MasterSubclassProc+0x54
0013e518 7739b874 77431d6c 00010090 0000000f user32!InternalCallWinProc+0x28
0013e590 7739c8b8 0015fce4 77431d6c 00010090 user32!UserCallWinProcCheckWow+0x151
0013e5ec 7739c9c6 00617618 0000000f 00000000 user32!DispatchClientMessage+0xd9
0013e614 7c828536 0013e62c 00000018 0013e750 user32!__fnDWORD+0x24
0013e640 7739ccb2 7739cb75 00010090 0000005e ntdll!KiUserCallbackDispatcher+0x2e
0013e654 77459d14 00010090 00000200 001c8900 user32!NtUserCallHwndLock+0xc
0013e66c 7745bd2d 00000004 016b0055 00000000 comctl32!TV_OnMouseMove+0x62
0013e6bc 7739b6e3 00010090 00000200 00000000 comctl32!TV_WndProc+0x647
0013e6e8 7739b874 7745b6e6 00010090 00000200 user32!InternalCallWinProc+0x28
0013e760 7739bfce 0015fce4 7745b6e6 00010090 user32!UserCallWinProcCheckWow+0x151
0013e790 7739bf74 7745b6e6 00010090 00000200 user32!CallWindowProcAorW+0x98
0013e7b0 77431848 7745b6e6 00010090 00000200 user32!CallWindowProcW+0x1b
0013e7cc 77431b9b 00010090 00000200 00000000 comctl32!CallOriginalWndProc+0x1a
0013e828 77431d5d 001cf0f8 00010090 00000200 comctl32!CallNextSubclassProc+0x3c
0013e84c 779cd761 00010090 00000200 00000000 comctl32!DefSubclassProc+0x46
0013e868 77431b9b 00010090 00000200 00000000 shdocvw!CNotifySubclassWndProc::_SubclassWndProc+0xa7
0013e8c4 77431d5d 001cf0f8 00010090 00000200 comctl32!CallNextSubclassProc+0x3c
0013e8e8 779cd86f 00010090 00000200 00000000 comctl32!DefSubclassProc+0x46
0013e934 779cd7e4 00010090 00000200 00000000 shdocvw!CNscTree::_SubClassTreeWndProc+0x3ae
0013e954 77431b9b 00010090 00000200 00000000 shdocvw!CNscTree::s_SubClassTreeWndProc+0x34
0013e9b0 77431dc0 001cf0f8 00010090 00000200 comctl32!CallNextSubclassProc+0x3c
0013ea04 7739b6e3 00010090 00000200 00000000 comctl32!MasterSubclassProc+0x54
0013ea30 7739b874 77431d6c 00010090 00000200 user32!InternalCallWinProc+0x28
0013eaa8 7739ba92 0015fce4 77431d6c 00010090 user32!UserCallWinProcCheckWow+0x151
0013eb10 7739bad0 0013eb50 00000000 0013eb38 user32!DispatchMessageWorker+0x327
0013eb20 75ed1410 0013eb50 00000000 00176388 user32!DispatchMessageW+0xf
0013eb38 75ed14fc 0013eb50 0013ee50 00000000 browseui!TimedDispatchMessage+0x33
0013ed98 75ec1c83 0015f7e8 0013ee50 0015f7e8 browseui!BrowserThreadProc+0x336
0013ee24 75ec61ef 0015f7e8 0015f7e8 00000000 browseui!BrowserProtectedThreadProc+0x44
0013fea8 779ba3a6 0015f7e8 00000001 00000000 browseui!SHOpenFolderWindow+0x22c
0013fec8 0040243d 00152552 00020d02 ffffffff shdocvw!IEWinMain+0x129
0013ff1c 00402744 00400000 00000000 00152552 iexplore!WinMain+0x316
0013ffc0 77e6f23b 00000000 00000000 7ffd000 iexplore!WinMainCRTStartup+0x182
0013fff0 00000000 004025c2 00000000 78746341 kernel32!BaseProcessStart+0x23

```

```

1 Id: c068.d71c Suspend: 1 Teb: 7ffdc000 Unfrozen
ChildEBP RetAddr Args to Child
00d4fea0 7c827cfb 7c80e5bb 00000002 00d4fef0 ntdll!KiFastSystemCallRet
00d4fea4 7c80e5bb 00000002 00d4fef0 00000001 ntdll!NtWaitForMultipleObjects+0xc
00d4ff48 7c80e4a2 00000002 00d4ff70 00000000 ntdll!EtwpWaitForMultipleObjectsEx+0xf7
00d4ffb8 77e64829 00000000 00000000 00000000 ntdll!EtwpEventPump+0x27f
00d4ffec 00000000 7c80e1fa 00000000 00000000 kernel32!BaseThreadStart+0x34

2 Id: c068.cba4 Suspend: 1 Teb: 7ffdb000 Unfrozen
ChildEBP RetAddr Args to Child
012bfe18 7c82783b 77c885ac 000001c4 012bff74 ntdll!KiFastSystemCallRet
012bfe1c 77c885ac 000001c4 012bff74 00000000 ntdll!NtReplyWaitReceivePortEx+0xc
012bff84 77c88792 012bfffac 77c8872d 00153cf0 rpcrt4!LRPC_ADDRESS::ReceiveLotsaCalls+0x198
012bff8c 77c8872d 00153cf0 00000000 00000000 rpcrt4!RecvLotsaCallsWrapper+0xd
012bfffac 77c7b110 00167030 012bfffec 77e64829 rpcrt4!BaseCachedThreadRoutine+0x9d
012bffb8 77e64829 00172088 00000000 00000000 rpcrt4!ThreadStartRoutine+0x1b
012bfffec 00000000 77c7b0f5 00172088 00000000 kernel32!BaseThreadStart+0x34

3 Id: c068.8604 Suspend: 1 Teb: 7ffda000 Unfrozen
ChildEBP RetAddr Args to Child
013bfe28 7c827d0b 7c83d236 000001d0 00000000 ntdll!KiFastSystemCallRet
013bfe2c 7c83d236 000001d0 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
013bfe68 7c83d281 000001d0 00000004 00000000 ntdll!RtlpWaitOnCriticalSection+0x1a3
013bfe88 7c839844 7c8877a0 00000000 77670000 ntdll!RtlEnterCriticalSection+0xa8
013bfff90 77e52860 77670000 77670000 00171698 ntdll!LdrUnloadDll+0x35
013bffa4 776b171d 77670000 00000000 00000000 kernel32!FreeLibraryAndExitThread+0x38
013bffb8 77e64829 00171698 00000000 00000000 ole32!CRpcThreadCache::RpcWorkerThreadEntry+0x39
013bfffec 00000000 776b16e4 00171698 00000000 kernel32!BaseThreadStart+0x34

4 Id: c068.d6dc Suspend: 1 Teb: 7ffd9000 Unfrozen
ChildEBP RetAddr Args to Child
016dfd24 7c827cfb 77e6202c 00000005 016dfd74 ntdll!KiFastSystemCallRet
016dfd28 77e6202c 00000005 016dfd74 00000001 ntdll!NtWaitForMultipleObjects+0xc
016dfdd0 7739bbd1 00000005 016dfdf8 00000000 kernel32!WaitForMultipleObjectsEx+0x11a
016dfe2c 7c919b2e 00000004 016dfe54 ffffffff user32!RealMsgWaitForMultipleObjectsEx+0x141
016dff50 7c8f7ada 77da3f12 00000000 00000000 shell32!CChangeNotify::_MessagePump+0x3b
016dff54 77da3f12 00000000 00000000 00000000 shell32!CChangeNotify::ThreadProc+0x1e
016dfffb8 77e64829 00000000 00000000 00000000 shlwapi!WrapperThreadProc+0x94
016dfffec 00000000 77da3ea5 0013dea8 00000000 kernel32!BaseThreadStart+0x34

5 Id: c068.caf4 Suspend: 1 Teb: 7ffd8000 Unfrozen
ChildEBP RetAddr Args to Child
01b1fdb4 7c827cfb 77e6202c 00000002 01b1fe04 ntdll!KiFastSystemCallRet
01b1fdb8 77e6202c 00000002 01b1fe04 00000001 ntdll!NtWaitForMultipleObjects+0xc
01b1fe60 7739bbd1 00000002 01b1fe88 00000000 kernel32!WaitForMultipleObjectsEx+0x11a
01b1febcb 6c296601 00000001 01b1fef0 ffffffff user32!RealMsgWaitForMultipleObjectsEx+0x141
01b1fedc 6c29684b 000004ff ffffffff 00000001 duser!CoreSC::Wait+0x3a
01b1ff10 6c28f9e6 01b1ff50 00000000 00000000 duser!CoreSC::xwProcessNL+0xab
01b1ff30 6c28bcce1 01b1ff50 00000000 00000000 duser!GetMessageExA+0x44
01b1ff84 77bcb530 00000000 00000000 00000000 duser!ResourceManager::SharedThreadProc+0xb6
01b1ffb8 77e64829 000385f0 00000000 00000000 msvcr!_endthreadex+0xa3
01b1ffec 00000000 77bcb4bc 000385f0 00000000 kernel32!BaseThreadStart+0x34

```

```

6 Id: c068.d624 Suspend: 1 Teb: 7ffd7000 Unfrozen
ChildEBP RetAddr Args to Child
01c9ff9c 7c826f4b 7c83d424 00000001 01c9ffb0 ntdll!KiFastSystemCallRet
01c9ffa0 7c83d424 00000001 01c9ffb0 00000000 ntdll!NtDelayExecution+0xc
01c9ffb8 77e64829 00000000 00000000 00000000 ntdll!RtlpTimerThread+0x47
01c9ffec 00000000 7c83d3dd 00000000 00000000 kernel32!BaseThreadStart+0x34

7 Id: c068.b4e0 Suspend: 1 Teb: 7ffd6000 Unfrozen
ChildEBP RetAddr Args to Child
01d9fd58 7c827d0b 7c83d236 000001d0 00000000 ntdll!KiFastSystemCallRet
01d9fd5c 7c83d236 000001d0 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
01d9fd98 7c83d281 000001d0 00000004 00000000 ntdll!RtlpWaitOnCriticalSection+0x1a3
01d9fdb8 7c839844 7c8877a0 75eb8b7c 75eb0000 ntdll!RtlEnterCriticalSection+0xa8
01d9fec0 77e6b1bb 75eb0000 75eb0000 001e2f98 ntdll!LdrUnloadD1l+0x35
01d9fed4 77da4c1c 75eb0000 0020eec8 77da591b kernel32!FreeLibrary+0x41
01d9feec 7c83a827 0020eec8 7c889080 001e4ec0 shlwapi!ExecuteWorkItem+0x28
01d9ff44 7c83aa0b 77da591b 0020eec8 00000000 ntdll!RtlpWorkerCallout+0x71
01d9ff64 7c83aa82 00000000 0020eec8 001e4ec0 ntdll!RtlpExecuteWorkerRequest+0x4f
01d9ff78 7c839f60 7c83a9ca 00000000 0020eec8 ntdll!RtlpApcCallout+0x11
01d9ffb8 77e64829 00000000 00000000 00000000 ntdll!RtlpWorkerThread+0x61
01d9ffec 00000000 7c839efb 00000000 00000000 kernel32!BaseThreadStart+0x34

8 Id: c068.d5a8 Suspend: 1 Teb: 7ffd5000 Unfrozen
ChildEBP RetAddr Args to Child
01fbba41c 7c827d0b 7c83d236 00000468 00000000 ntdll!KiFastSystemCallRet
01fbba420 7c83d236 00000468 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
01fbba45c 7c83d281 00000468 00000004 00000000 ntdll!RtlpWaitOnCriticalSection+0x1a3
01fbba47c 7c9136c9 7cae42d0 001c97b0 80070003 ntdll!RtlEnterCriticalSection+0xa8
01fbba494 7c913b75 0000000c 00000000 00000001 shell32!CMountPoint::GetMountPoint+0x33
01fbba4c8 7c91358d 01fbba4fc 0000000c 00000000 shell32!CDrivesFolder::_FillIDDrive+0x5c
01fbba52c 7c9109e7 0018e988 00000000 001c97b0 shell32!CDrivesFolder::ParseDisplayName+0x9f
01fbba594 7c9119ff 0018e9b0 00000000 001c97b0 shell32!CRegFolder::ParseDisplayName+0x93
01fbba5bc 7c910bb8 00000000 001a8e30 00000000 shell32!CDesktopFolder::_ChildParseDisplayName+0x22
01fbba60c 7c9109e7 0017cde0 00000000 001c97b0 shell32!CDesktopFolder::ParseDisplayName+0x7e
01fbba674 7c9109ab 0015f058 00000000 001c97b0 shell32!CRegFolder::ParseDisplayName+0x93
01fbba6ac 7c911ab4 00000000 00000000 00000000 shell32!SHParseDisplayName+0xa3
01fbba6d0 7c911a6e 01fbbe60 00000000 00000002 shell32!ILCreateFromPathEx+0x3d
01fbba6ec 7c911a4b 01fbbe60 01fbba700 00000000 shell32!SHILCreateFromPath+0x17
01fbba704 7c95e055 01fbbe60 00000104 01fbca0 shell32!ILCreateFromPathW+0x18
01fbba84 7c9ef49d 01fbbe60 00000000 01fbba8c shell32!SHGetFileInfoW+0x117
01fbca06c 01b4d195 01fbca00 00000000 01fbca0 shell32!SHGetFileInfoA+0x6a
WARNING: Stack unwind information not available. Following frames may be wrong.
01fbca04 01b54a20 0000073c 02541f28 00000000 issftran!SSCopyFile+0x27ad
00000000 00000000 00000000 00000000 issftran!DllUnregisterServer+0x70ad

9 Id: c068.d750 Suspend: 1 Teb: 7ffd4000 Unfrozen
ChildEBP RetAddr Args to Child
0228ff7c 7c8277db 71b25914 000004b4 0228ffc0 ntdll!KiFastSystemCallRet
0228ff80 71b25914 000004b4 0228ffc0 0228ffb4 ntdll!ZwRemoveIoCompletion+0xc
0228ffb8 77e64829 71b259de 00000000 00000000 mssock!SockAsyncThread+0x69
0228ffec 00000000 71b258ab 001fcda0 00000000 kernel32!BaseThreadStart+0x34

```

0:000> du 7c8d8828  
7c8d8828 "EXPLORER.EXE"

```
0:000> da 01fbc200
01fbc200 "M:\WINDOWS"
```

## Comments

---

Another example of the critical section analysis:

```
0:005> !locks

CritSec ModuleA!__onexitbegin+50cbc at 006f1da8
WaiterWoken No
LockCount 49
RecursionCount 1
OwningThread 19a4
EntryCount 0
ContentionCount 2a51
*** Locked

CritSec ModuleA!__onexitbegin+5131c at 006f2408
WaiterWoken No
LockCount 0
RecursionCount 1
OwningThread 10f4
EntryCount 0
ContentionCount 0
*** Locked

CritSec +6e8827c at 06e8827c
WaiterWoken No
LockCount 0
RecursionCount 1
OwningThread 19a4
EntryCount 0
ContentionCount 0
*** Locked

Scanned 2404 critical sections
```

In this example, we have only one critical section which blocks other threads according to this formula:

```
BlockedThreads = LockCount - (RecursionCount - 1) = 49 - (1 - 1)

CritSec ModuleA!__onexitbegin+50cbc at 006f1da8
WaiterWoken No
LockCount 49
RecursionCount 1
OwningThread 19a4
EntryCount 0
ContentionCount 2a51
*** Locked
```

Then we can examine owner thread's call stack 19a4 by looking at `~*kv` output and find other waiting threads by searching for critical section address: 006f1da8.

All other two critical sections are being held by *OwningThread*, and there are no waiting threads for them:

```
BlockedThreads = LockCount - (RecursionCount - 1) = 0 - (1 - 1) = 0
```

Therefore, the rule of thumb is to look at *LockCount* values.

One of the questions asked is if there any way to see the stack trace if the thread-owner of a critical section is already dead?

```
0:022> !locks
```

```
CritSec ole32!g_mxSingleThreadOle+18 at 76a40664
WaiterWoken No
LockCount 4
RecursionCount 1
OwningThread 1ec
EntryCount 0
ContentionCount 6
*** Locked
```

Virtual memory for TEB is decommitted already, and, therefore, no data for the stack start address and its limit is available. We might guess that stack region itself is decommitted during thread termination so we cannot use memory search here to find its raw stack data. Live debugging might help here with scripts to set conditional breakpoints and saving dumps automatically upon some condition.

Another example:

```
0:000> !locks
```

```
CritSec +83eb6d10 at 83eb6d10
WaiterWoken No
LockCount -452697857
RecursionCount 36294152
OwningThread a
EntryCount 1379620
ContentionCount 1379620
*** Locked
```

```
CritSec +840d6d10 at 840d6d10
WaiterWoken Yes
LockCount -60090439
RecursionCount 1093776890
OwningThread ffffffffadcb569d
EntryCount 0
ContentionCount 0
*** Locked
```

```
CritSec +840d6d10 at 840d6d10
WaiterWoken Yes
LockCount -60090439
RecursionCount 1093776890
OwningThread ffffffffadcb569d
EntryCount 0
ContentionCount 0
*** Locked
```

```
CritSec +86156d10 at 86156d10
WaiterWoken Yes
LockCount -278997059
RecursionCount 1224415142
OwningThread ffffffff9e8e4272
EntryCount 0
ContentionCount 0
*** Locked
```

```
CritSec +83c96d10 at 83c96d10
WaiterWoken Yes
LockCount 520027693
RecursionCount 1097194433
OwningThread ffffffff8c000036
EntryCount 0
ContentionCount 0
*** Locked
```

```
CritSec +83c96d10 at 83c96d10
WaiterWoken Yes
LockCount 520027693
RecursionCount 1097194433
OwningThread ffffffff8c000036
EntryCount 0
ContentionCount 0
*** Locked
```

Scanned 1223 critical sections

We can see here that *LockCount* and *RecursionCount* have strange numbers. Here we have **Critical Section Corruption** pattern (page 156). We can also try **!cs -l -o -s** command. Sometimes, if we dump all sections, we can see where corruption starts (**!locks -v** or **!cs**).

Example for **Virtualized Process** (page 1068):

```
0:000:x86> ~*kv

. 0 Id: 2e24.35f8 Suspend: 0 Teb: 7efdb000 Unfrozen
ChildEBP RetAddr Args to Child
0015cedc 77748e44 000004b8 00000000 00000000 ntdll_77710000!ZwWaitForSingleObject+0x15 (FPO: [3,0,0])
0015cf40 77748d28 00000000 00000000 000035f8 ntdll_77710000!RtlpWaitOnCriticalSection+0x13e (FPO: [Non-Fpo])
0015cf68 558f829b 00e531d8 d3c25775 058f0a0c ntdll_77710000!RtlEnterCriticalSection+0x150 (FPO: [Non-Fpo])
WARNING: Stack unwind information not available. Following frames may be wrong.
0015d4a4 558f9e66 d3c256e1 000000df 00000000 libdjvullibre!DJVU::DjVuDocument::process_threqs+0x6b
0015d530 558cdea5 0015d57c 000000df 00000001 libdjvullibre!DJVU::DjVuDocument::get_thumbnail+0x8c6
0015d594 0103d8d1 00c6a1e0 000000df 00000000 libdjvullibre!ddjvu_thumbnail_status+0x115
0015d604 55107cd9 00c3c5a8 00000000 00000006 djview+0x4d8d1
0015d658 53f3fc29 00000000 00c3c5a8 00b8ed18 QtCore4!QMetaCallEvent::placeMetaCall+0x19
0015d8a0 550f948d 00c3c5a8 00eeb598 d3c25ad6 QtGui4!QApplicationPrivate::notify_helper+0xb9
0015d8e0 550fb07f 00b8e640 00eeb598 d3c25b12 QtCore4!QCoreApplication::notifyInternal+0x8d
0015d924 5511e835 00000000 00000000 00b8e640 QtCore4!QCoreApplicationPrivate::sendPostedEvents+0x1cf
00000000 00000000 00000000 00000000 QtCore4!QEventDispatcherWin32::event+0x555
```

```
1 Id: 2e24.1390 Suspend: 0 Teb: 7efd8000 Unfrozen
ChildEBP RetAddr Args to Child
0296f884 7344a41c 00000001 0296f8e4 00000001 ntdll_77710000!NtWaitForMultipleObjects+0x15 (FPO: [5,0,0])
0296f92c 7702338a 00000000 0296f978 77749f72 winmm!timeThread+0x3c (FPO: [Non-Fpo])
0296f938 77749f72 00000000 55277637 00000000 kernel32!BaseThreadInitThunk+0xe (FPO: [Non-Fpo])
0296f978 77749f45 7344a3e0 00000000 00000000 ntdll_77710000!_RtlUserThreadStart+0x70 (FPO: [Non-Fpo])
0296f990 00000000 7344a3e0 00000000 00000000 ntdll_77710000!_RtlUserThreadStart+0x1b (FPO: [Non-Fpo])
```

2 Id: 2e24.37a0 Suspend: 0 Teb: 7efd5000 Unfrozen  
ChildEBP RetAddr Args to Child  
02c3facc 77762f91 00000003 005fb610 00000001 ntdll\_77710000!NtWaitForMultipleObjects+0x15 (FPO: [5,0,0])  
02c3fc60 7702338a 00000000 02c3fcac 77749f72 ntdll\_77710000!TppWaiterpThread+0x33d (FPO: [Non-Fpo])  
02c3fc6c 77749f72 005fb5e0 557273e3 00000000 kernel32!BaseThreadInitThunk+0xe (FPO: [Non-Fpo])  
02c3fcac 77749f45 77762e65 005fb5e0 00000000 ntdll\_77710000!\_RtlUserThreadStart+0x70 (FPO: [Non-Fpo])  
02c3fcc4 00000000 77762e65 005fb5e0 00000000 ntdll\_77710000!\_RtlUserThreadStart+0x1b (FPO: [Non-Fpo])

3 Id: 2e24.2150 Suspend: 0 Teb: 7ef4a000 Unfrozen  
ChildEBP RetAddr Args to Child  
0300f940 77763392 0000039c 0300f9f4 54b175ef ntdll\_77710000!NtWaitForWorkViaWorkerFactory+0x12 (FPO: [2,0,0])  
0300faa0 7702338a 005fa788 0300faec 77749f72 ntdll\_77710000!TppWorkerThread+0x216 (FPO: [Non-Fpo])  
0300faa0 77749f72 005fa788 54b175a3 00000000 kernel32!BaseThreadInitThunk+0xe (FPO: [Non-Fpo])  
0300faec 77749f45 77763e85 005fa788 00000000 ntdll\_77710000!\_RtlUserThreadStart+0x70 (FPO: [Non-Fpo])  
0300fb04 00000000 77763e85 005fa788 00000000 ntdll\_77710000!\_RtlUserThreadStart+0x1b (FPO: [Non-Fpo])

4 Id: 2e24.3478 Suspend: 0 Teb: 7ef4d000 Unfrozen  
ChildEBP RetAddr Args to Child  
0320f3bc 77748e44 000004c4 00000000 00000000 ntdll\_77710000!ZwWaitForSingleObject+0x15 (FPO: [3,0,0])  
0320f420 77748d28 00000000 00000000 058f0a18 ntdll\_77710000!RtlpWaitOnCriticalSection+0x13e (FPO: [Non-Fpo])  
0320f448 558f8750 04f3538c d0f77a55 058dbb18 ntdll\_77710000!RtlEnterCriticalSection+0x150 (FPO: [Non-Fpo])  
WARNING: Stack unwind information not available. Following frames may be wrong.  
0320f984 558fa26c d0f77a41 058dbb18 04ffb618 libdjvulibre!DJVU::DjVuDocument::process\_threqs+0x520  
0320f9b0 5591ec70 058dbb18 00000042 00000001 libdjvulibre!DJVU::DjVuDocument::notify\_file\_flags\_changed+0xd0  
0320fa00 55900ac1 058dbb18 00000042 00000001 libdjvulibre!DJVU::DjVuPortcaster::notify\_file\_flags\_changed+0x80  
0320faf4 559005a7 d0f778f1 00000000 00000000 libdjvulibre!DJVU::DjVuFile::decode\_func+0x4c1  
0320fb20 55945bf2 058dbb18 d0f778b1 00000000 libdjvulibre!DJVU::DjVuFile::static\_decode\_func+0x87  
0320fb60 55b4c556 04d95f90 d0f779b8 00000000 libdjvulibre!DJVU::GNativeString::setat+0x282  
0320fb98 55b4c600 00000000 0320fb00 7702338a msvcr100!\_endthreadex+0x3f (FPO: [Non-Fpo])  
0320fba4 7702338a 04ad13d8 0320fb0 77749f72 msvcr100!\_endthreadex+0xce (FPO: [Non-Fpo])  
0320fb00 77749f72 04ad13d8 549174bf 00000000 kernel32!BaseThreadInitThunk+0xe (FPO: [Non-Fpo])  
0320fb00 77749f45 55b4c59c 04ad13d8 00000000 ntdll\_77710000!\_RtlUserThreadStart+0x70 (FPO: [Non-Fpo])  
0320fc08 00000000 55b4c59c 04ad13d8 00000000 ntdll\_77710000!\_RtlUserThreadStart+0x1b (FPO: [Non-Fpo])

0:000:x86> !cs -l -o -s  
-----  
DebugInfo = 0x0000000005f05c0  
Critical section = 0x0000000000e531d8 (+0xE531D8)  
LOCKED  
LockCount = 0x1  
WaiterWoken = No  
OwningThread = 0x000000000003478  
RecursionCount = 0x1  
LockSemaphore = 0x4B8  
SpinCount = 0x0000000000000000  
OwningThread DbgId = ~45  
OwningThread Stack =  
ChildEBP RetAddr Args to Child  
0320f3bc 77748e44 000004c4 00000000 00000000 ntdll\_77710000!ZwWaitForSingleObject+0x15 (FPO: [3,0,0])  
0320f420 77748d28 00000000 00000000 058f0a18 ntdll\_77710000!RtlpWaitOnCriticalSection+0x13e (FPO: [Non-Fpo])  
0320f448 558f8750 04f3538c d0f77a55 058dbb18 ntdll\_77710000!RtlEnterCriticalSection+0x150 (FPO: [Non-Fpo])  
0320f984 558fa26c d0f77a41 058dbb18 04ffb618 libdjvulibre!DJVU::DjVuDocument::process\_threqs+0x520  
0320f9b0 5591ec70 058dbb18 00000042 00000001 libdjvulibre!DJVU::DjVuDocument::notify\_file\_flags\_changed+0xd0  
0320fa00 55900ac1 058dbb18 00000042 00000001 libdjvulibre!DJVU::DjVuPortcaster::notify\_file\_flags\_changed+0x80  
0320faf4 559005a7 d0f778f1 00000000 00000000 libdjvulibre!DJVU::DjVuFile::decode\_func+0x4c1  
0320fb20 55945bf2 058dbb18 d0f778b1 00000000 libdjvulibre!DJVU::DjVuFile::static\_decode\_func+0x87  
0320fb60 55b4c556 04d95f90 d0f779b8 00000000 libdjvulibre!DJVU::GNativeString::setat+0x282  
0320fb98 55b4c600 00000000 0320fb0 7702338a msvcr100!\_endthreadex+0x3f (FPO: [Non-Fpo])  
0320fba4 7702338a 04ad13d8 0320fb0 77749f72 msvcr100!\_endthreadex+0xce (FPO: [Non-Fpo])  
0320fb00 77749f72 04ad13d8 549174bf 00000000 kernel32!BaseThreadInitThunk+0xe (FPO: [Non-Fpo])  
0320fb00 77749f45 55b4c59c 04ad13d8 00000000 ntdll\_77710000!\_RtlUserThreadStart+0x70 (FPO: [Non-Fpo])  
0320fc08 00000000 55b4c59c 04ad13d8 00000000 ntdll\_77710000!\_RtlUserThreadStart+0x1b (FPO: [Non-Fpo])  
\${\$ntdllwsym}!RtlpStackTraceDataBase is NULL. Probably the stack traces are not enabled.  
-----  
DebugInfo = 0x0000000000625d80  
Critical section = 0x0000000004f3538c (+0x4F3538C)  
LOCKED  
LockCount = 0x1  
WaiterWoken = No  
OwningThread = 0x0000000000035f8  
RecursionCount = 0x1  
LockSemaphore = 0x4C4

```

SpinCount = 0x0000000000000000
OwningThread DbgId = ~0s
OwningThread Stack =
ChildEBP RetAddr Args to Child
0015ced4 77748e44 000004b8 00000000 ntdll_77710000!ZwWaitForSingleObject+0x15 (FPO: [3,0,0])
0015cf40 77748d28 00000000 00000000 000035f8 ntdll_77710000!RtlpWaitOnCriticalSection+0x13e (FPO: [Non-Fpo])
0015cf68 558f829b 00e531d8 d3c25775 058f0a0c ntdll_77710000!RtlEnterCriticalSection+0x150 (FPO: [Non-Fpo])
0015d4a4 558f9e66 d3c256e1 000000df 00000000 libdjvullibre!DJVU::DjVuDocument::process_threqs+0x6b
0015d530 558cdea5 0015d57c 000000df 00000001 libdjvullibre!DJVU::DjVuDocument::get_thumbnail+0x8c6
0015d594 0103d8d1 00c6a1e0 000000df 00000000 libdjvullibre!ddjvu_thumbnail_status+0x115
0015d604 55107cd9 00c3c5a8 00000000 00000006 djview+0x4d8d1
0015d658 53f3fc29 00000000 00c3c5a8 00b8ed18 QtCore4!QMetaCallEvent::placeMetaCall+0x19
0015d8a0 550f948d 00c3c5a8 00eeb598 d3c25ad6 QtGui4!QApplicationPrivate::notify_helper+0xb9
0015d8e0 550fb07f 00b8e640 00eeb598 d3c25b12 QtCore4!QCoreApplication::notifyInternal+0x8d
0015d924 5511e835 00000000 00000000 00b8e640 QtCore4!QCoreApplicationPrivate::sendPostedEvents+0x1cf
00000000 00000000 00000000 00000000 00000000 QtCore4!QEventDispatcherWin32::event+0x555
${$ntdllwsym}!RtlpStackTraceDataBase is NULL. Probably the stack traces are not enabled.

```

Often, we can preliminary suppose a critical section deadlock if all locked critical section owner threads are waiting for critical sections (as seen from **!cs -l -o -s** WinDbg command output and corresponding stack traces, **k** or **!thread**).

## Executive Resources

ERESOURCE (executive resource) is a Windows synchronization object that has ownership semantics.

An executive resource can be owned exclusively or can have a shared ownership. This is similar to the following file sharing analogy: when a file is opened for writing others can't write or read it; if we have that file opened for reading others can read it but can't write to it.

ERESOURCE structure is linked into a list and has threads as owners which allows us to quickly find deadlocks using **!locks** command in the kernel and complete memory dumps. Here is the definition of \_ERESOURCE from x86 and x64 Windows:

```
0: kd> dt -r1 _ERESOURCE
+0x000 SystemResourcesList : _LIST_ENTRY
    +0x000 Flink          : Ptr32 _LIST_ENTRY
    +0x004 Blink          : Ptr32 _LIST_ENTRY
+0x008 OwnerTable      : Ptr32 _OWNER_ENTRY
    +0x000 OwnerThread   : Uint4B
    +0x004 OwnerCount    : Int4B
    +0x004 TableSize     : Uint4B
+0x00c ActiveCount     : Int2B
+0x00e Flag             : Uint2B
+0x010 SharedWaiters   : Ptr32 _KSEMAPHORE
    +0x000 Header        : _DISPATCHER_HEADER
    +0x010 Limit         : Int4B
+0x014 ExclusiveWaiters : Ptr32 _KEVENT
    +0x000 Header        : _DISPATCHER_HEADER
+0x018 OwnerThreads    : [2] _OWNER_ENTRY
    +0x000 OwnerThread   : Uint4B
    +0x004 OwnerCount    : Int4B
    +0x004 TableSize     : Uint4B
+0x028 ContentionCount : Uint4B
+0x02c NumberOfSharedWaiters : Uint2B
+0x02e NumberOfExclusiveWaiters : Uint2B
+0x030 Address          : Ptr32 Void
+0x030 CreatorBackTraceIndex : Uint4B
+0x034 SpinLock         : Uint4B

0: kd> dt -r1 _ERESOURCE
nt!_ERESOURCE
+0x000 SystemResourcesList : _LIST_ENTRY
    +0x000 Flink          : Ptr64 _LIST_ENTRY
    +0x008 Blink          : Ptr64 _LIST_ENTRY
+0x010 OwnerTable      : Ptr64 _OWNER_ENTRY
    +0x000 OwnerThread   : Uint8B
    +0x008 OwnerCount    : Int4B
    +0x008 TableSize     : Uint4B
+0x018 ActiveCount     : Int2B
+0x01a Flag             : Uint2B
+0x020 SharedWaiters   : Ptr64 _KSEMAPHORE
    +0x000 Header        : _DISPATCHER_HEADER
    +0x018 Limit         : Int4B
```

```
+0x028 ExclusiveWaiters : Ptr64 _KEVENT
  +0x000 Header      : _DISPATCHER_HEADER
+0x030 OwnerThreads   : [2] _OWNER_ENTRY
  +0x000 OwnerThread  : UInt8B
  +0x008 OwnerCount    : Int4B
  +0x008 TableSize     : UInt4B
+0x050 ContentionCount : UInt4B
+0x054 NumberOfSharedWaiters : UInt2B
+0x056 NumberOfExclusiveWaiters : UInt2B
+0x058 Address        : Ptr64 Void
+0x058 CreatorBackTraceIndex : UInt8B
+0x060 SpinLock       : UInt8B
```

If we have a list of resources from **!locks** output, we can start following threads that own these resources. Owner threads are marked with a star character (\*):

```
0: kd> !locks
**** DUMP OF ALL RESOURCE OBJECTS ****
KD: Scanning for held locks.....
Resource @ 0x8815b928    Exclusively owned
Contention Count = 6234751
NumberOfExclusiveWaiters = 53
Threads: 89ab8db0-01<*>
Threads Waiting On Exclusive Access:
  8810fa08    880f5b40    88831020    87e33020
  880353f0    88115020    88131678    880f5db0
  89295420    88255378    880f8b40    8940d020
  880f58d0    893ee500    880edac8    880f8db0
  89172938    879b3020    88091510    88038020
  880407b8    88051020    89511db0    8921f020
  880e9db0    87c33020    88064cc0    88044730
  8803f020    87a2a020    89529380    8802d330
  89a53020    89231b28    880285b8    88106b90
  8803cbc8    88aa3020    88093400    8809aab0
  880ea540    87d46948    88036020    8806e198
  8802d020    88038b40    8826b020    88231020
  890a2020    8807f5d0
```

We see that 53 threads are waiting for **\_KTHREAD** 89ab8db0 to release **\_ERESOURCE** 8815b928. Searching for this thread address reveals the following:

```
Resource @ 0x88159560    Exclusively owned
Contention Count = 166896
NumberOfExclusiveWaiters = 1
Threads: 8802a790-01<*>
Threads Waiting On Exclusive Access:
  89ab8db0
```

We see that the thread 89ab8db0 is waiting for 8802a790 to release the resource 88159560. We continue searching for the thread 8802a790 waiting for another thread, but we skip occurrences when this thread is not waiting:

```

Resource @ 0x881f7b60    Exclusively owned
Threads: 8802a790-01<*>

Resource @ 0x8824b418    Exclusively owned
Contention Count = 34
Threads: 8802a790-01<*>

Resource @ 0x8825e5a0    Exclusively owned
Threads: 8802a790-01<*>

Resource @ 0x88172428    Exclusively owned
Contention Count = 5
NumberOfExclusiveWaiters = 1
Threads: 8802a790-01<*>
Threads Waiting On Exclusive Access:
880f5020

```

Searching further we see that the thread 8802a790 is waiting for the thread 880f5020 to release the resource 89bd7bf0:

```

Resource @ 0x89bd7bf0    Exclusively owned
Contention Count = 1
NumberOfExclusiveWaiters = 1
Threads: 880f5020-01<*>
Threads Waiting On Exclusive Access:
8802a790

```

If we look carefully we see that we have already seen the thread 880f5020 above, and we repeat the fragment:

```

Resource @ 0x88172428    Exclusively owned
Contention Count = 5
NumberOfExclusiveWaiters = 1
Threads: 8802a790-01<*>
Threads Waiting On Exclusive Access:
880f5020

```

We see that the thread 880f5020 is waiting for the thread 8802a790, and the thread 8802a790 is waiting for the thread 880f5020.

Therefore, we have identified the classical **Deadlock**. What we have to do now is to look at **Stack Traces** (page 926) of these threads to see involved components.

## LPC

Here is an example of **Deadlock** pattern involving LPC. In the stack trace below, *svchost.exe* thread (we call it thread A) receives an LPC call and dispatches it to *componentA* module which makes another LPC call (MessageId 000135b8) and then waiting for a reply:

```

THREAD 89143020 Cid 09b4.10dc Peb: 7ff91000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Non-Alertable
8914320c Semaphore Limit 0x1
Waiting for reply to LPC MessageId 000135b8:
Current LPC port d64a5328
Not impersonating
DeviceMap          d64028f0
Owning Process    891b8b80      Image:      svchost.exe
Wait Start TickCount 237408      Ticks: 1890 (0:00:00:29.531)
Context Switch Count 866
UserTime           00:00:00.031
KernelTime         00:00:00.015
Win32 Start Address 0x000135b2
LPC Server thread working on message Id 135b2
Start Address kernel32!BaseThreadStartThunk (0x7c82b5f3)
Stack Init b91f9000 Current b91f8c08 Base b91f9000 Limit b91f6000 Call 0
Priority 9 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr
b91f8c20 8083e6a2 nt!KiSwapContext+0x26
b91f8c4c 8083f164 nt!KiSwapThread+0x284
b91f8c94 8093983f nt!KeWaitForSingleObject+0x346
b91f8d50 80834d3f nt!NtRequestWaitReplyPort+0x776
b91f8d50 7c94ed54 nt!KiFastCallEntry+0xfc
02bae928 7c941c94 nt!ntdll!KiFastSystemCallRet
02bae92c 77c42700 nt!ntdll!NtRequestWaitReplyPort+0xc
02bae984 77c413ba RPCRT4!LRPC_CCALL::SendReceive+0x230
02bae990 77c42c7f RPCRT4!I_RpcSendReceive+0x24
02bae9a4 77cb5d63 RPCRT4!NdrSendReceive+0x2b
02baec48 674825b6 RPCRT4!NdrClientCall+0x334
02baec5c 67486776 componentA!bar+0x16
...
...
...
02baf8d4 77c40f3b componentA!foo+0x157
02baf8f8 77cb23f7 RPCRT4!Invoke+0x30
02bafcf8 77cb26ed RPCRT4!NdrStubCall2+0x299
02bafd14 77c409be RPCRT4!NdrServerCall2+0x19
02bafd48 77c4093f RPCRT4!DispatchToStubInCNoAvrf+0x38
02bafd9c 77c40865 RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x117
02bafdc0 77c434b1 RPCRT4!RPC_INTERFACE::DispatchToStub+0xa3
02bafdfc 77c41bb3 RPCRT4!LRPC_SCALL::DealWithRequestMessage+0x42c
02bafe20 77c45458 RPCRT4!LRPC_ADDRESS::DealWithLRPCRequest+0x127
02baff84 77c2778f RPCRT4!LRPC_ADDRESS::ReceiveLotsaCalls+0x430
02baff8c 77c2f7dd RPCRT4!RecvLotsaCallsWrapper+0xd
02baffac 77c2de88 RPCRT4!BaseCachedThreadRoutine+0x9d
02baffb8 7c82608b RPCRT4!ThreadStartRoutine+0x1b
02baffec 00000000 kernel32!BaseThreadStart+0x34

```

We search for that LPC message to find the server thread:

```
1: kd> !lpc message 000135b8
Searching message 135b8 in threads ...
  Server thread 89115db0 is working on message 135b8
Client thread 89143020 waiting a reply from 135b8
...
...
...
```

It belongs to *Process.exe*, and we call it thread B:

```
1: kd> !thread 89115db0 0x16
THREAD 89115db0  Cid 098c.0384  Teb: 7ff79000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Non-
Alertable
  8a114628  SynchronizationEvent
Not impersonating
DeviceMap          d64028f0
Owning Process    8a2c9d88      Image:       Process.exe
Wait Start TickCount 237408      Ticks: 1890 (0:00:00:29.531)
Context Switch Count 1590
UserTime           00:00:03.265
KernelTime         00:00:01.671
Win32 Start Address 0x000135b8
LPC Server thread working on message Id 135b8
Start Address kernel32!BaseThreadStartThunk (0x7c82b5f3)
Stack Init b952d000 Current b952cc60 Base b952d000 Limit b952a000 Call 0
Priority 9 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr Args to Child
b952cc78 8083e6a2 89115e28 89115db0 89115e58 nt!KiSwapContext+0x26
b952cca4 8083f164 00000000 00000000 00000000 nt!KiSwapThread+0x284
b952ccce 8092db70 8a114628 00000006 ffffff01 nt!KeWaitForSingleObject+0x346
b952cd50 80834d3f 00000a7c 00000000 00000000 nt!NtWaitForSingleObject+0x9a
b952cd50 7c94ed54 00000a7c 00000000 00000000 nt!KiFastCallEntry+0xfc
22aceb48 7c942124 7c95970f 00000a7c 00000000 ntdll!KiFastSystemCallRet
22aceb4c 7c95970f 00000a7c 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
22aceb88 7c959620 00000000 00000004 00002000 ntdll!RtlpWaitOnCriticalSection+0x19c
22acea88 1b005744 06d30940 1b05ea80 06d30940 ntdll!RtlEnterCriticalSection+0xa8
22acebb0 1b05ea80 06d30940 fefffff 0cd410c0 componentB!bar+0xb
...
...
...
22acf8b0 77c40f3b 00080002 000800e2 00000001 componentB!foo+0xeb
22acf8e0 77cb23f7 0de110dc 22acf8c 00000007 RPCRT4!Invoke+0x30
22acfce0 77cb26ed 00000000 00000000 19f38f94 RPCRT4!NdrStubCall12+0x299
22acfcfc 77c409be 19f38f94 17316ef0 19f38f94 RPCRT4!NdrServerCall12+0x19
22acfd30 77c75e41 0de1dc58 19f38f94 22acfdec RPCRT4!DispatchToStubInCNoAvrf+0x38
22acfd48 77c4093f 0de1dc58 19f38f94 22acfdec RPCRT4!DispatchToStubInCAvrf+0x14
22acfd9c 77c40865 00000041 00000000 0de2b398 RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x117
22acfdc0 77c434b1 19f38f94 00000000 0de2b398 RPCRT4!RPC_INTERFACE::DispatchToStub+0xa3
22acfd9c 77c41bb3 1beeae8 16b96f50 1baeef0 RPCRT4!LRPC_SCALL::DealWithRequestMessage+0x42c
22acf8e0 77c45458 16b96f88 22acf8e8 1beeae8 RPCRT4!LRPC_ADDRESS::DealWithLRPCRequest+0x127
```

0x16 flags for **Ithread** extension command are used to temporarily set the process context to the owning process and show the first three function call parameters. We see that the thread B is waiting for the critical section 06d30940, and we use user space **!locks** extension command to find who owns it after switching process context:

```
1: kd> .process /r /p 8a2c9d88
Implicit process is now 8a2c9d88
Loading User Symbols

1: kd> !ntsdexts.locks

CritSec +6d30940 at 06d30940
WaiterWoken      No
LockCount        1
RecursionCount   1
OwningThread    d6c
EntryCount       0
ContentionCount 1
*** Locked
```

Now we try to find a thread with TID d6c (thread C):

```
1: kd> !thread -t d6c
Looking for thread Cid = d6c ...
THREAD 890d8bb8  Cid 098c.0d6c  Teb: 7ff71000 Win32Thread: bc23cc20 WAIT: (Unknown) UserMode Non-
Alertable
     890d8da4  Semaphore Limit 0x1
Waiting for reply to LPC MessageId 000135ea:
Current LPC port d649a678
Not impersonating
DeviceMap          d64028f0
Owning Process    8a2c9d88  Image:      Process.exe
Wait Start TickCount 237641  Ticks: 1657 (0:00:00:25.890)
Context Switch Count 2102           LargeStack
UserTime           00:00:00.734
KernelTime         00:00:00.234
Win32 Start Address msvcrt!_endthreadex (0x77b9b4bc)
Start Address kernel32!BaseThreadStartThunk (0x7c82b5f3)
Stack Init ba91d000 Current ba91cc08 Base ba91d000 Limit ba919000 Call 0
Priority 13 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr  Args to Child
ba91cc20 8083e6a2 890d8c30 890d8bb8 890d8c60 nt!KiSwapContext+0x26
ba91cc4c 8083f164 890d8da4 890d8d78 890d8bb8 nt!KiSwapThread+0x284
ba91cc94 8093983f 890d8da4 00000011 8a2c9d01 nt!KeWaitForSingleObject+0x346
ba91cd50 80834d3f 000008bc 19c94f00 19c94f00 nt!NtRequestWaitReplyPort+0x776
ba91cd50 7c94ed54 000008bc 19c94f00 19c94f00 nt!KiFastCallEntry+0xfc
2709ebf4 7c941c94 77c42700 000008bc 19c94f00 ntdll!KiFastSystemCallRet
2709ebf8 77c42700 000008bc 19c94f00 19c94f00 ntdll!NtRequestWaitReplyPort+0xc
2709ec44 77c413ba 2709ec80 2709ec64 77c42c7f RPCRT4!LRPC CCALL::SendReceive+0x230
2709ec50 77c42c7f 2709ec80 779b2770 2709f06c RPCRT4!I RpcSendReceive+0x24
2709ec64 77cb219b 2709ecac 1957cf4 1957ab38 RPCRT4!NdrSendReceive+0x2b
2709f04c 779b43a3 779b2770 779b1398 2709f06c RPCRT4!NdrClientCall2+0x22e
...
```

```
...
2709ff84 77b9b530 26658fb0 00000000 00000000 ComponentC!foo+0x18d
2709ffb8 7c82608b 26d9af70 00000000 00000000 msvcrt!_endthreadex+0xa3
2709ffec 00000000 77b9b4bc 26d9af70 00000000 kernel32!BaseThreadStart+0x34
```

We see that the thread C makes another LPC call (MessageId 000135e) and waiting for a reply. Let's find the server thread processing the message (thread D):

```
1: kd> !lpc message 000135ea
Searching message 135ea in threads ...
Client thread 890d8bb8 waiting a reply from 135ea
Server thread 89010020 is working on message 135ea
...
...
...

1: kd> !thread 89010020 16
THREAD 89010020 Cid 09b4.1530 Peb: 7ff93000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Non-Alertable
8903ba28 Mutant - owning thread 89143020
Not impersonating
DeviceMap d64028f0
Owning Process 891b8b80 Image: svchost.exe
Wait Start TickCount 237641 Ticks: 1657 (0:00:00:25.890)
Context Switch Count 8
UserTime 00:00:00.000
KernelTime 00:00:00.000
Win32 Start Address 0x000135ea
LPC Server thread working on message Id 135ea
Start Address kernel32!BaseThreadStartThunk (0x7c82b5f3)
Stack Init b9455000 Current b9454c60 Base b9455000 Limit b9452000 Call 0
Priority 9 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr Args to Child
b9454c78 8083e6a2 89010098 89010020 890100c8 nt!KiSwapContext+0x26
b9454ca4 8083f164 00000000 00000000 00000000 nt!KiSwapThread+0x284
b9454cec 8092db70 8903ba28 00000006 00000001 nt!KeWaitForSingleObject+0x346
b9454d50 80834d3f 00000514 00000000 00000000 nt!NtWaitForSingleObject+0x9a
b9454d50 7c94ed54 00000514 00000000 00000000 nt!KiFastCallEntry+0xfc
02b5f720 7c942124 75fdbbe44 00000514 00000000 ntdll!KiFastSystemCallRet
02b5f724 75fdbbe44 00000514 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
02b5f744 75fdc57f 000e6014 000da62c 02b5fc0 ComponentD!bar+0x42
...
...
...
02b5f8c8 77c40f3b 000d0a48 02b5fc90 00000001 ComponentD!foo+0x49
02b5f8f8 77cb23f7 75fdf8f2 02b5fae0 00000007 RPCRT4!Invoke+0x30
02b5fcf8 77cb26ed 00000000 00000000 000d4f24 RPCRT4!NdrStubCall12+0x299
02b5fd14 77c409be 000d4f24 000b5d70 000d4f24 RPCRT4!NdrServerCall12+0x19
02b5fd48 77c4093f 75fff834 000d4f24 02b5fdec RPCRT4!DispatchToStubInCNoAvrf+0x38
02b5fd9c 77c40865 00000005 00000000 7600589c RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x117
02b5fdc0 77c434b1 000d4f24 00000000 7600589c RPCRT4!RPC_INTERFACE::DispatchToStub+0xa3
02b5fdfe 77c41bb3 000d3550 000a78d0 001054b8 RPCRT4!LRPC_SCALL::DealWithRequestMessage+0x42c
02b5fe20 77c45458 000a7908 02b5fe38 000d3550 RPCRT4!LRPC_ADDRESS::DealWithLRPCRequest+0x127
02b5ff84 77c2778f 02b5ffac 77c2f7dd 000a78d0 RPCRT4!LRPC_ADDRESS::ReceiveLotsaCalls+0x430
```

```
02b5ff8c 77c2f7dd 000a78d0 00000000 00000000 RPCRT4!RecvLotsaCallsWrapper+0xd  
02b5ffac 77c2de88 0008ae00 02b5ffec 7c82608b RPCRT4!BaseCachedThreadRoutine+0x9d  
02b5ffb8 7c82608b 000d5c20 00000000 00000000 RPCRT4!ThreadStartRoutine+0x1b  
02b5ffec 00000000 77c2de6d 000d5c20 00000000 kernel32!BaseThreadStart+0x34
```

We see that the thread D is waiting for the mutant object owned by the thread A (89143020). Therefore we have **Deadlock** spanning 2 process boundaries via RPC/LPC calls with the following dependency graph:

```
A (svchost.exe) LPC-> B (Process.exe) CritSec-> C (Process.exe) LPC-> D (svchost.exe) Obj-> A (svchost.exe)
```

## Managed Space

Now we illustrate a synchronization block **Deadlock** pattern in managed code. Here we can use either a manual **!syncblk** WinDbg command coupled with a stack trace and disassembly analysis or SOSEX<sup>34</sup> extension **!dlk** command (which automates the whole detection process).

```
0:011> !syncblk
Index SyncBlock MonitorHeld Recursion Owning Thread Info SyncBlock Owner
373 052cbf1c 3 1 08f69280 bc0 14 0a1ffd84 System.String
375 052cbd3c 3 1 08f68728 b6c 12 0a1ffd4c System.String

0:011> ~12s
[...]

0:012> k
ChildEBP RetAddr
WARNING: Stack unwind information not available. Following frames may be wrong.
09c8ebd0 79ed98fd ntdll!KiFastSystemCallRet
09c8ec38 79ed9889 mscorewks!WaitForMultipleObjectsEx_SO_TOLERANT+0x6f
09c8ec58 79ed9808 mscorewks!Thread::DoAppropriateAptStateWait+0x3c
09c8ecd0 79ed96c4 mscorewks!Thread::DoAppropriateWaitWorker+0x13c
09c8ed2c 79ed9a62 mscorewks!Thread::DoAppropriateWait+0x40
09c8ed88 79e78944 mscorewks!CLREvent::WaitEx+0xf7
09c8ed9c 79ed7b37 mscorewks!CLREvent::Wait+0x17
09c8ee28 79ed7a9e mscorewks!AwareLock::EnterEpilog+0x8c
09c8ee44 79ebd7e4 mscorewks!AwareLock::Enter+0x61
09c8eee4 074c1f38 mscorewks!JIT_MonEnterWorker_Portable+0xb3
09c8ef0c 793b0d1f 0x74c1f38
09c8ef14 79373ecd mscorlib_ni+0x2f0d1f
09c8ef28 793b0c68 mscorlib_ni+0x2b3ecd
09c8ef40 79e7c74b mscorlib_ni+0x2f0c68
09c8ef50 79e7c6cc mscorewks!CallDescrWorker+0x33
09c8efd0 79e7c8e1 mscorewks!CallDescrWorkerWithHandler+0xa3
09c8f110 79e7c783 mscorewks!MethodDesc::CallDescr+0x19c
09c8f12c 79e7c90d mscorewks!MethodDesc::CallTargetWorker+0x1f
09c8f140 79fc58cd mscorewks!MethodDescCallSite::Call_RetArgSlot+0x18
09c8f328 79ef3207 mscorewks!ThreadNative::KickOffThread_Worker+0x190
09c8f33c 79ef31a3 mscorewks!Thread::DoADCallBack+0x32a
09c8f3d0 79ef30c3 mscorewks!Thread::ShouldChangeAbortToUnload+0xe3
09c8f40c 79f01723 mscorewks!Thread::ShouldChangeAbortToUnload+0x30a
09c8f41c 79f02a5d mscorewks!Thread::RaiseCrossContextException+0x434
09c8f4cc 79f02ab7 mscorewks!Thread::DoADCallBack+0xda
09c8f4e8 79ef31a3 mscorewks!Thread::DoADCallBack+0x310
09c8f57c 79ef30c3 mscorewks!Thread::ShouldChangeAbortToUnload+0xe3
09c8f5b8 79ef4826 mscorewks!Thread::ShouldChangeAbortToUnload+0x30a
09c8f5e0 79fc57b1 mscorewks!Thread::ShouldChangeAbortToUnload+0x33e
09c8f5f8 79fc56ac mscorewks!ManagedThreadBase::KickOff+0x13
09c8f694 79f95a2e mscorewks!ThreadNative::KickOffThread+0x269
```

<sup>34</sup> <http://www.stevestechspot.com/default.aspx>

```

09c8fd34 76573833 mscorewks!Thread::intermediateThreadProc+0x49
09c8fd40 77c1a9bd kernel32!BaseThreadInitThunk+0xe
09c8fd80 00000000 ntdll!LdrInitializeThunk+0x4d

0:012> ub 074c1f38
074c1f11 eb10 jmp 074c1f23
074c1f13 8b0df8927b02 mov ecx,dword ptr ds:[27B92F8h]
074c1f19 e8367ef271 call mscorelib_ni+0x329d54 (793e9d54)
074c1f1e e89272a472 call mscorewks!JIT_EndCatch (79f091b5)
074c1f23 b9d0070000 mov ecx,7D0h
074c1f28 e8c432b072 call mscorewks!ThreadNative::Sleep (79fc51f1)
074c1f2d 8b0d88dc7b02 mov ecx,dword ptr ds:[27BDC88h]
074c1f33 e811389b72 call mscorewks!JIT_MonEnterWorker (79e75749)

0:012> dp 27BDC88h 11
027bdc88 0a1ffd84

0:012> ~14s

0:014> k
ChildEBP RetAddr
WARNING: Stack unwind information not available. Following frames may be wrong.
0b83ed04 79ed98fd ntdll!KiFastSystemCallRet
0b83ed6c 79ed9889 mscorewks!WaitForMultipleObjectsEx_SO_TOLERANT+0x6f
0b83ed8c 79ed9808 mscorewks!Thread::DoAppropriateAptStateWait+0x3c
0b83ee10 79ed96c4 mscorewks!Thread::DoAppropriateWaitWorker+0x13c
0b83ee60 79ed9a62 mscorewks!Thread::DoAppropriateWait+0x40
0b83eebc 79e78944 mscorewks!CLREvent::WaitEx+0xf7
0b83eed0 79ed7b37 mscorewks!CLREvent::Wait+0x17
0b83ef5c 79ed7a9e mscorewks!AwareLock::EnterEpilog+0x8c
0b83ef78 79ebd7e4 mscorewks!AwareLock::Enter+0x61
0b83f018 074c5681 mscorewks!JIT_MonEnterWorker_Portable+0xb3
0b83f01c 793b0d1f 0x74c5681
0b83f024 79373ecd mscorelib_ni+0x2f0d1f
0b83f038 793b0c68 mscorelib_ni+0x2b3ecd
0b83f050 79e7c74b mscorelib_ni+0x2f0c68
0b83f060 79e7c6cc mscorewks!CallDescrWorker+0x33
0b83f0e0 79e7c8e1 mscorewks!CallDescrWorkerWithHandler+0xa3
0b83f220 79e7c783 mscorewks!MethodDesc::CallDescr+0x19c
0b83f23c 79e7c90d mscorewks!MethodDesc::CallTargetWorker+0x1f
0b83f250 79fc58cd mscorewks!MethodDescCallSite::Call_RetArgSlot+0x18
0b83f438 79ef3207 mscorewks!ThreadNative::KickOffThread_Worker+0x190
0b83f44c 79ef31a3 mscorewks!Thread::DoADCallBack+0x32a
0b83f4e0 79ef30c3 mscorewks!Thread::ShouldChangeAbortToUnload+0xe3
0b83f51c 79f01723 mscorewks!Thread::ShouldChangeAbortToUnload+0x30a
0b83f52c 79f02a5d mscorewks!Thread::RaiseCrossContextException+0x434
0b83f5dc 79f02ab7 mscorewks!Thread::DoADCallBack+0xda
0b83f5f8 79ef31a3 mscorewks!Thread::DoADCallBack+0x310
0b83f68c 79ef30c3 mscorewks!Thread::ShouldChangeAbortToUnload+0xe3
0b83f6c8 79ef4826 mscorewks!Thread::ShouldChangeAbortToUnload+0x30a
0b83f6f0 79fc57b1 mscorewks!Thread::ShouldChangeAbortToUnload+0x33e
0b83f708 79fc56ac mscorewks!ManagedThreadBase::KickOff+0x13
0b83f7a4 79f95a2e mscorewks!ThreadNative::KickOffThread+0x269
0b83ff3c 76573833 mscorewks!Thread::intermediateThreadProc+0x49

```

```
0b83ff48 77c1a9bd kernel32!BaseThreadInitThunk+0xe
0b83ff88 00000000 ntdll!LdrInitializeThunk+0x4d

0:014> ub 074c5681
074c565c 080c54 or byte ptr [esp+edx*2],cl
074c565f 07 pop es
074c5660 8b0d88dc7b02 mov ecx,dword ptr ds:[27BDC88h]
074c5666 e8de009b72 call mscorewks!JIT_MonEnterWorker (79e75749)
074c566b a1240a5407 mov eax,dword ptr ds:[07540A24h]
074c5670 3105280a5407 xor dword ptr ds:[7540A28h],eax
074c5676 8b0d84dc7b02 mov ecx,dword ptr ds:[27BDC84h]
074c567c e8c8009b72 call mscorewks!JIT_MonEnterWorker (79e75749)

0:014> dp 27BDC84h 11
027bdc84 0a1ffd4c

0:014> !dlk
Examining SyncBlocks...
Scanning for ReaderWriterLock instances...
Scanning for holders of ReaderWriterLock locks...
Scanning for ReaderWriterLockSlim instances...
Scanning for holders of ReaderWriterLockSlim locks...
Examining CriticalSections...
Could not find symbol ntdll!RtlCriticalSectionList.
Scanning for threads waiting on SyncBlocks...
Scanning for threads waiting on ReaderWriterLock locks...
Scanning for threads waiting on ReaderWriterLockSlim locks...
Scanning for threads waiting on CriticalSections...
*DEADLOCK DETECTED*
CLR thread 0xd holds the lock on SyncBlock 052cbd3c OBJ:0a1ffd4c[System.String] STRVAL=critical section 1
...and is waiting for the lock on SyncBlock 052cbf1c OBJ:0a1ffd84[System.String] STRVAL=critical section 2
CLR thread 0xb holds the lock on SyncBlock 052cbf1c OBJ:0a1ffd84[System.String] STRVAL=critical section 2
...and is waiting for the lock on SyncBlock 052cbd3c OBJ:0a1ffd4c[System.String] STRVAL=critical section 1
CLR Thread 0xd is waiting at UserQuery+ClassMain.thread_proc_1()(+0x42 IL)(+0x60 Native)
CLR Thread 0xb is waiting at UserQuery+ClassMain.thread_proc_2()(+0x19 IL)(+0x21 Native)

1 deadlock detected.
```

## Mixed Objects

### Kernel Space

Here is another pattern of a **Deadlock** variety involving mixed objects in kernel space. Let's look at a complete **Manual Dump** (page 625) file from a hanging system:

```
0: kd> !analyze -v

NMI_HARDWARE_FAILURE (80)
This is typically due to a hardware malfunction. The hardware supplier should
be called.

Arguments:
Arg1: 004f4454
Arg2: 00000000
Arg3: 00000000
Arg4: 00000000
```

Here we have a problem to read all executive resource locks:

```
3: kd> !locks
**** DUMP OF ALL RESOURCE OBJECTS ****

Resource @ nt!CmpRegistryLock (0x808a48c0)      Shared 36 owning threads
Contention Count = 48
Threads: 86aecae0-01<*> 8b76db40-01<*> 8b76ddb0-01<*> 89773020-01<*>
          87222db0-01<*> 87024ba8-01<*> 89a324f0-01<*> 86b4e298-01<*>
          87925b40-01<*> 86b4db40-01<*> 8701f738-01<*> 86ffb198-01<*>
          86b492f0-01<*> 8701bad8-01<*> 86ae2db0-01<*> 86c85db0-01<*>
          86a9ddb0-01<*> 86a86db0-01<*> 86aa7db0-01<*> 86a9f5c0-01<*>
          86c5adb0-01<*> 8767ba38-01<*> 86afedb0-01<*> 89877960-01<*>
          8772cdb0-01<*> 87348628-01<*> 874d6748-01<*> 872365e0-01<*>
          87263970-01<*> 873bf020-01<*> 86c13db0-01<*> 893dcdb0-01<*>
          86afa020-01<*> 878e5020-01<*> 874959f8-01<*> 86b2dc70-01<*>

KD: Scanning for held locks...Error 1 in reading nt!_ERESOURCE.SystemResourcesList.Flink @ f76ee2a0
```

This is probably because the dump was **Truncated** (page 1015):

```
Loading Dump File [MEMORY.DMP]
Kernel Complete Dump File: Full address space is available

WARNING: Dump file has been truncated. Data may be missing.
```

However looking at the resource 808a48c0 closely we see that it is owned by the thread 86aecae0 (Cid 2810.2910) which is blocked on a mutant owned by the thread 86dcf3a8:

```
3: kd> !locks -v 0x808a48c0

Resource @ nt!CmpRegistryLock (0x808a48c0)      Shared 36 owning threads
Contention Count = 48
Threads: 86aecae0-01<*>

THREAD 86aecae0  Cid 2810.2910  Teb: 7fffd000 Win32Thread: bc54ab88 WAIT: (Unknown) KernelMode Non-
Alertable
  86dda264  Mutant - owning thread 86dcf3a8
  Not impersonating
  DeviceMap          da534618
  Owning Process    86f30b70      Image:       ApplicationA.exe
  Wait Start TickCount   1074481      Ticks: 51601 (0:00:13:26.265)
  Context Switch Count 9860        LargeStack
  UserTime           00:00:01.125
  KernelTime         00:00:00.890
  Win32 Start Address 0x300019f0
  Start Address kernel32!BaseProcessStartThunk (0x7c8217f8)
  Stack Init b5342000 Current b5341150 Base b5342000 Limit b533d000 Call 0
  Priority 12 BasePriority 10 PriorityDecrement 0
  ChildEBP RetAddr
  b5341168 80833465 nt!KiSwapContext+0x26
  b5341194 80829a62 nt!KiSwapThread+0x2e5
  b53411dc b91f4c08 nt!KeWaitForSingleObject+0x346
WARNING: Stack unwind information not available. Following frames may be wrong.
  b5341200 b91ee770 driverA+0xec08
  b5341658 b91e9ca7 driverA+0x8770
  b5341af0 8088978c driverA+0x3ca7
  b5341af0 8082f829 nt!KiFastCallEntry+0xfc
  b5341b7c 808ce716 nt!ZwSetInformationFile+0x11
  b5341bbc 808dd8d8 nt!CmpDoFileSetSize+0x5e
  b5341bd4 808bd798 nt!CmpFileSetSize+0x16
  b5341bf4 808be23f nt!HvpGrowLog1+0x52
  b5341c18 808bfc6b nt!HvMarkDirty+0x453
  b5341c40 808c3fd4 nt!HvMarkCellDirty+0x255
  b5341cb4 808b7e2f nt!CmSetValueKey+0x390
  b5341d44 8088978c nt!NtSetValueKey+0x241
  b5341d44 7c9485ec nt!KiFastCallEntry+0xfc
  0013f5fc 00000000 ntdll!KiFastSystemCallRet

8b76db40-01<*>
```

```

THREAD 8b76db40 Cid 0004.00c8 Teb: 00000000 Win32Thread: 00000000 GATEWAIT
    Not impersonating
    DeviceMap          d6600900
    Owning Process     8b7772a8      Image:           System
    Wait Start TickCount 1074667      Ticks: 51415 (0:00:13:23.359)
    Context Switch Count 65106
    UserTime           00:00:00.000
    KernelTime          00:00:00.781
    Start Address nt!ExpWorkerThread (0x80880352)
    Stack Init bae35000 Current bae34c68 Base bae35000 Limit bae32000 Call 0
    Priority 12 BasePriority 12 PriorityDecrement 0
    ChildEBP RetAddr
    bae34c80 80833465 nt!KiSwapContext+0x26
    bae34cac 8082ffc0 nt!KiSwapThread+0x2e5
    bae34cd4 8087d6f6 nt!KeWaitForGate+0x152
    dbba6d78 00000000 nt!ExfAcquirePushLockExclusive+0x112

```

[...]

A reminder about **Cid**: it is the so-called **Client id** composed of **Process id** and **Thread id (Pid.Tid)**. Also, a mutant is just another name for a mutex object which has ownership semantics:

```

0: kd> dt _KMUTANT 86dda264
nt!_KMUTANT
+0x000 Header          : _DISPATCHER_HEADER
+0x010 MutantListEntry : _LIST_ENTRY [ 0x86dcf3a8 - 0x86dcf3a8 ]
+0x018 OwnerThread     : 86dcf3a8 _KTHREAD
+0x01c Abandoned       : 0 "
+0x01d ApcDisable       : 0x1 "

```

Now we look at that thread 86dcf3a8 and see that it belongs to *ApplicationB* (Cid 25a0.14b8):

```

3: kd> !thread 86dcf3a8
THREAD 86dcf3a8 Cid 25a0.14b8 Teb: 7ffa9000 Win32Thread: bc3e0d20 WAIT: (Unknown) UserMode Non-
Alertable
  8708b888 Thread
    86dcf420 NotificationTimer
Not impersonating
DeviceMap          da534618
Owning Process     87272d88      Image:           ApplicationB.exe
Wait Start TickCount 1126054      Ticks: 28 (0:00:00:00.437)
Context Switch Count 2291        LargeStack
UserTime           00:00:00.078
KernelTime          00:00:00.218
Win32 Start Address msvcrt!_endthreadex (0x77b9b4bc)
Start Address kernel32!BaseThreadStartThunk (0x7c8217ec)
Stack Init b550a000 Current b5509c60 Base b550a000 Limit b5507000 Call 0
Priority 8 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr Args to Child
b5509c78 80833465 86dcf3a8 86dcf450 00000003 nt!KiSwapContext+0x26
b5509ca4 80829a62 00000000 b5509d14 00000000 nt!KiSwapThread+0x2e5
b5509cec 80938d0c 8708b888 00000006 00000001 nt!KeWaitForSingleObject+0x346
b5509d50 8088978c 00000960 00000000 b5509d14 nt!NtWaitForSingleObject+0x9a
b5509d50 7c9485ec 00000960 00000000 b5509d14 nt!KiFastCallEntry+0xfc

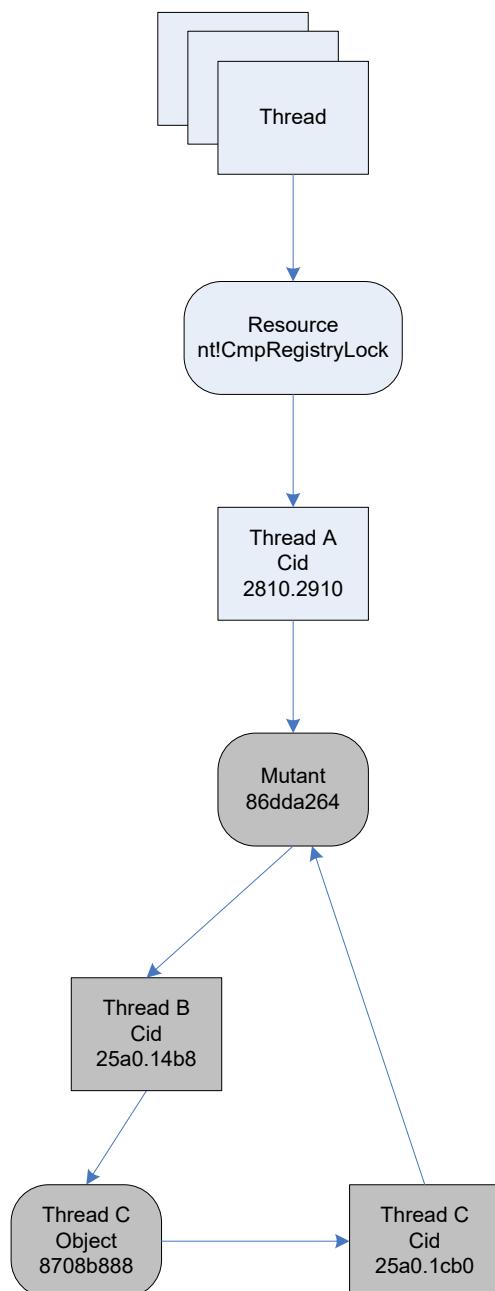
```

```
WARNING: Stack unwind information not available. Following frames may be wrong.  
0454f3cc 00000000 00000000 00000000 ntdll!KiFastSystemCallRet
```

We see that it is waiting on 8708b888 object which is a thread itself, and it is waiting for the same mutant 86dda264 owned by the thread 86dcf3a8 (Cid 25a0.14b8):

```
3: kd> !thread 8708b888  
THREAD 8708b888 Cid 25a0.1cb0 Teb: 7ffa6000 Win32Thread: bc3ecb20 WAIT: (Unknown) KernelMode Non-  
Alertable  
86dda264 Mutant - owning thread 86dcf3a8  
Not impersonating  
DeviceMap da534618  
Owning Process 87272d88 Image: ApplicationB.exe  
Wait Start TickCount 1070470 Ticks: 55612 (0:00:14:28.937)  
Context Switch Count 11 LargeStack  
UserTime 00:00:00.000  
KernelTime 00:00:00.000  
Win32 Start Address dll!_beginthread (0x1b1122a9)  
Start Address kernel32!BaseThreadStartThunk (0x7c8217ec)  
Stack Init b4d12000 Current b4d117fc Base b4d12000 Limit b4d0f000 Call 0  
Priority 9 BasePriority 8 PriorityDecrement 0  
ChildEBP RetAddr Args to Child  
b4d11814 80833465 8708b888 8708b930 00000003 nt!KiSwapContext+0x26  
b4d11840 80829a62 0000096c b4d118c4 b91e8f08 nt!KiSwapThread+0x2e5  
b4d11888 b91f4c08 86dda264 00000006 00000000 nt!KeWaitForSingleObject+0x346  
WARNING: Stack unwind information not available. Following frames may be wrong.  
b4d118ac b91ee818 86dda260 b4d11d64 86dda000 DriverA+0xec08  
b4d11d04 b91e8f58 000025a0 0000096c b4d11d64 DriverA+0x8818  
b4d11d58 8088978c 0000096c 0567f974 7c9485ec DriverA+0x2f58  
b4d11d58 7c9485ec 0000096c 0567f974 7c9485ec nt!KiFastCallEntry+0xfc  
0567f974 30cba6ad 0000096c 00000000 00000003 ntdll!KiFastSystemCallRet
```

We can summarize our findings in the following **Wait Chain** diagram (page 1104):



Looking from the component-object relationship perspective, it is *DriverA.sys* that is waiting on the mutant 86dda264 although both blocked threads B and C belong to *ApplicationB* process.

## User Space

This is another variant of **Deadlock** pattern when we have mixed synchronization objects, for example, events, and critical sections. An event may be used to signal the availability of some work item for processing it, the fact that the queue is not empty and a critical section may be used to protect some shared data.

The typical deadlock scenario here is when one thread resets an event by calling *WaitForSingleObject* and tries to acquire a critical section. In the mean time the second thread has already acquired that critical section and now is waiting for the event to be set:

<u>Thread A</u>	<u>Thread B</u>
..	..
reset Event	..
..	acquire CS
wait for CS	..
	wait for Event

The classical fix to this bug is to acquire the critical section and wait for the event in the same order in both threads.

In our example crash dump, we can easily identify the second thread that acquired the critical section and is waiting for the event 0x480:

```
0:000> !locks

CritSec ntdll!LdrpLoaderLock+0 at 7c889d94
WaiterWoken      No
LockCount        9
RecursionCount   1
OwningThread     2038
EntryCount       0
ContentionCount  164
*** Locked

13  Id: 590.2038 Suspend: 1 Teb: 7ffaa000 Unfrozen
ChildEBP RetAddr  Args to Child
0483fd5c 7c822124 77e6bad8 00000480 00000000 ntdll!KiFastSystemCallRet
0483fd60 77e6bad8 00000480 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
0483fd60 77e6ba42 00000480 ffffffff 00000000 kernel32!WaitForSingleObjectEx+0xac
0483fd64 776cfb30 00000480 ffffffff 777904f8 kernel32!WaitForSingleObject+0x12
0483fe00 776adfaa 00000480 00000000 00000080 ole32!CDllHost::ClientCleanupFinish+0x2a
0483fe2c 776adfc1a 00000000 0483fe7c 77790828 ole32!D1lHostProcessUninitialize+0x80
0483fe4c 776b063f 00000000 00000000 0c9ece0 ole32!ApartmentUninitialize+0xf8
0483fe64 776b06e3 0483fe7c 00000000 00000001 ole32!wCoUninitialize+0x48
0483fe80 776e43f5 00000001 77670000 776afef0 ole32!CoUninitialize+0x65
0483fe8c 776afef0 0483feb4 776b5cb8 77670000 ole32!DoThreadSpecificCleanup+0x63
0483fe94 776b5cb8 77670000 00000003 00000000 ole32!ThreadNotification+0x37
0483feb4 776b5c1b 77670000 00000003 00000000 ole32!D1lMain+0x176
0483fed4 7c82257a 77670000 00000003 00000000 ole32!_D1lMainCRTStartup+0x52
0483fef4 7c83c195 776b5bd3 77670000 00000003 ntdll!LdrpCallInitRoutine+0x14
0483ffa8 77e661d6 00000000 00000000 0483ffec ntdll!LdrShutdownThread+0xd2
```

```
0483ffb8 77e66090 00000000 00000000 00000000 kernel32!ExitThread+0x2f
0483ffec 00000000 77c5de6d 0ab24f68 00000000 kernel32!BaseThreadStart+0x39

0:000> !handle 480 ff
Handle 00000480
Type      Event
Attributes 0
GrantedAccess 0x1f0003:
    Delete,ReadControl,WriteDac,WriteOwner,Synch
    QueryState,ModifyState
HandleCount 2
PointerCount 4
Name      <none>
No object specific information available
```

It is difficult to find the first thread, the one which has reset the event and is waiting for the critical section. In our dump we have 9 such threads from **!locks** command output:

LockCount	9
-----------	---

Event as a synchronization primitive doesn't have an owner. Despite this, we can try to find 0x480 and *WaitForSingleObject* address nearby on some other thread raw stack if that information wasn't overwritten. Let's do a virtual memory search:

```
0:000> s -d 0 L40000000 00000480
000726ec 00000480 00000022 000004a4 00000056
008512a0 00000480 00000480 00000000 00000000
008512a4 00000480 00000000 00000000 01014220
0085ab68 00000480 00000480 00000092 00000000
0085ab6c 00000480 00000092 00000000 01014234
00eb12a0 00000480 00000480 00000000 00000000
00eb12a4 00000480 00000000 00000000 0101e614
00ebbeb68 00000480 00000480 00000323 00000000
00ebbeb6c 00000480 00000323 00000000 0101e644
03ffb4fc 00000480 d772c13b ce753966 00fa840f
040212a0 00000480 00000480 00000000 00000000
040212a4 00000480 00000000 00000000 01063afc
0402ab68 00000480 00000480 00000fb6 00000000
0402ab6c 00000480 00000fb6 00000000 01063b5c
041312a0 00000480 00000480 00000000 00000000
041312a4 00000480 00000000 00000000 01065b28
0413eb68 00000480 00000480 00001007 00000000
0413eb6c 00000480 00001007 00000000 01065b7c
043412a0 00000480 00000480 00000000 00000000
043412a4 00000480 00000000 00000000 01066b44
0434ab68 00000480 00000480 00001033 00000000
0434ab6c 00000480 00001033 00000000 01066b9c
0483fd68 00000480 00000000 00000000 00000000
0483ffd8 00000480 ffffffff 00000000 0483fe00
0483fdec 00000480 ffffffff 777904f8 77790738
0483fe08 00000480 00000000 00000080 776b0070
0483fe20 00000480 00000000 00000000 0483fe4c
05296f58 00000480 ffffffff ffffffff ffffffff
```

```

05297eb0 00000480 00000494 000004a4 000004c0
0557cf9c 00000480 00000000 00000000 00000000
05580adc 00000480 00000000 00000000 00000000
0558715c 00000480 00000000 00000000 00000000
0558d3cc 00000480 00000000 00000000 00000000
0559363c 00000480 00000000 00000000 00000000
0559ee0c 00000480 00000000 00000000 00000000
055a507c 00000480 00000000 00000000 00000000
056768ec 00000480 00000000 00000000 00000000
0568ef14 00000480 00000000 00000000 00000000
0581ff88 00000480 07ca7ee0 0581ff98 776cf2a3
05ed1260 00000480 00000480 00000000 00000000
05ed1264 00000480 00000000 00000000 01276efc
05ed8b68 00000480 00000480 00005c18 00000000
05ed8b6c 00000480 00005c18 00000000 01276f74
08f112a0 00000480 00000480 00000000 00000000
08f112a4 00000480 00000000 00000000 00000000
08f1ab68 00000480 00000480 00007732 00000000
08f1ab6c 00000480 00007732 00000000 01352db0

```

In bold we highlighted the thread #13 raw stack occurrences and in italics bold, we highlighted memory locations that belong to another thread raw stack. In fact, these are the only memory locations from search results that make any sense from the code perspective. The only meaningful stack traces can be found in memory locations highlighted in bold above.

This can be seen if we feed search results to WinDbg **dds** command:

```

0:000> .foreach (place { s-[1]d 0 L40000000 00000480 }) { dds place -30; .printf "\n" }
000726bc 00000390
000726c0 00000022
000726c4 000003b4
000726c8 00000056
000726cc 00000004
000726d0 6dc3f6fd
000726d4 0000040c
000726d8 0000001e
000726dc 0000042c
000726e0 00000052
000726e4 00000004
000726e8 eacb0f6d
000726ec 00000480
000726f0 00000022
000726f4 000004a4
000726f8 00000056
000726fc 00000004
00072700 62b796d2
00072704 000004fc
00072708 0000001e
0007270c 0000051c
00072710 00000052
00072714 00000004
00072718 2a615cff
0007271c 00000570
00072720 00000024

```

00072724	00000598
00072728	00000058
0007272c	00000004
00072730	51913e59
00072734	000005f0
00072738	00000016
...	
...	
...	
0568eee4	05680008 xpsp2res+0x1b0008
0568eee8	01200000
0568eec	00001010
0568eef0	00200001
0568eef4	00000468
0568eef8	00000121
0568eefc	00000000
0568ef00	00000028
0568ef04	00000030
0568ef08	00000060
0568ef0c	00040001
0568ef10	00000000
0568ef14	00000480
0568ef18	00000000
0568ef1c	00000000
0568ef20	00000000
0568ef24	00000000
0568ef28	00000000
0568ef2c	00800000
0568ef30	00008000
0568ef34	00808000
0568ef38	00000080
0568ef3c	00800080
0568ef40	00008080
0568ef44	00808080
0568ef48	00c0c0c0
0568ef4c	00ff0000
0568ef50	0000ff00
0568ef54	00ffff00
0568ef58	000000ff
0568ef5c	00ff00ff
0568ef60	0000ffff
0581ff58	0581ff70
0581ff5c	776b063f ole32!wCoUninitialize+0x48
0581ff60	00000001
0581ff64	00007530
0581ff68	77790438 ole32!gATHost
0581ff6c	00000000
0581ff70	0581ff90
0581ff74	776cf370 ole32!CDllHost::WorkerThread+0xdd
0581ff78	0581ff8c
0581ff7c	00000001
0581ff80	77e6ba50 kernel32!WaitForSingleObjectEx
0581ff84	0657cfe8
0581ff88	<u>00000480</u>
0581ff8c	07ca7ee0

0581ff90	0581ff98
0581ff94	776cf2a3 ole32!DLLHostThreadEntry+0xd
0581ff98	0581ffb8
<b>0581ff9c</b>	<b>776b2307 ole32!CRpcThread::WorkerLoop+0x1e</b>
0581ffa0	77790438 ole32!gATHost
0581ffa4	00000000
0581ffa8	0657cfe8
0581ffac	77670000 ole32!_imp__InstallApplication <PERF> (ole32+0x0)
0581ffb0	776b2374 ole32!CRpcThreadCache::RpcWorkerThreadEntry+0x20
0581ffb4	00000000
0581ffb8	0581ffec
0581ffbc	77e6608b kernel32!BaseThreadStart+0x34
0581ffc0	0657cfe8
0581ffc4	00000000
0581ffc8	00000000
0581ffcc	0657cfe8
0581ffd0	3cfb5963
0581ffd4	0581ffc4
05ed1230	0101f070
05ed1234	05ed1274
05ed1238	05ed1174
05ed123c	05ed0000
05ed1240	05ed1280
05ed1244	00000000
05ed1248	00000000
05ed124c	00000000
05ed1250	05ed8b80
05ed1254	05ed8000
05ed1258	00002000
05ed125c	00001000
05ed1260	00000480
05ed1264	00000480
05ed1268	00000000
05ed126c	00000000
05ed1270	01276efc
05ed1274	05ed12b4
05ed1278	05ed1234
05ed127c	05ed0000
05ed1280	05ed2d00
05ed1284	05ed1240
05ed1288	05ed1400
05ed128c	00000000
05ed1290	05edad0
05ed1294	05eda000
05ed1298	00002000
05ed129c	00001000
05ed12a0	00000220
05ed12a4	00000220
05ed12a8	00000000
05ed12ac	00000000
...	
...	
...	
08f1ab3c	00000000
08f1ab40	00000000

08f1ab44	00000000
08f1ab48	00000000
08f1ab4c	00000000
08f1ab50	00000000
08f1ab54	00000000
08f1ab58	00000000
08f1ab5c	00000000
08f1ab60	abcdbbbb
08f1ab64	08f11000
08f1ab68	00000480
08f1ab6c	00000480
08f1ab70	00007732
08f1ab74	00000000
08f1ab78	01352db0
08f1ab7c	dcbabbbb
08f1ab80	ffffffff
08f1ab84	c0c00ac1
08f1ab88	00000000
08f1ab8c	c0c0c0c0
08f1ab90	c0c0c0c0
08f1ab94	c0c0c0c0
08f1ab98	c0c0c0c0
08f1ab9c	c0c0c0c0
08f1aba0	c0c0c0c0
08f1aba4	ffffffff
08f1aba8	c0c00ac1
08f1abac	00000000
08f1abb0	c0c0c0c0
08f1abb4	c0c0c0c0
08f1abb8	c0c0c0c0

We see that the address 0581ff88 is the most meaningful and it also has *WaitForSingleObjectEx* nearby. This address belongs to the raw stack of the following thread #16:

```

16  Id: 590.1a00 Suspend: 1 Teb: 7ffa9000 Unfrozen
ChildEBP RetAddr
0581fc98 7c822124 nt!KiFastSystemCallRet
0581fc9c 7c83970f nt!NtWaitForSingleObject+0xc
0581fcda 7c839620 nt!RtlpWaitOnCriticalSection+0x19c
0581fcf8 7c83a023 nt!RtlEnterCriticalSection+0xa8
0581fe00 77e67bcd nt!LdrUnloadDll+0x35
0581fe14 776b46fb kernel32!FreeLibrary+0x41
0581fe20 776b470f ole32!CClassCache::CDllPathEntry::CFinishObject::Finish+0x2f
0581fe34 776b44a0 ole32!CClassCache::CFinishComposite::Finish+0x1d
0581ff0c 776b0bfd ole32!CClassCache::CleanUpDllsForApartment+0x1d0
0581ff38 776b0b1f ole32!FinishShutdown+0xd7
0581ff58 776b063f ole32!ApartmentUninitialize+0x94
0581ff70 776cf370 ole32!wCoUninitialize+0x48
0581ff90 776cf2a3 ole32!CDllHost::WorkerThread+0xdd
0581ff98 776b2307 ole32!DLLHostThreadEntry+0xd
0581ffac 776b2374 ole32!CRpcThread::WorkerLoop+0x1e
0581ffb8 77e6608b ole32!CRpcThreadCache::RpcWorkerThreadEntry+0x20
0581ffec 00000000 kernel32!BaseThreadStart+0x34

```

If we disassemble *ole32!CRpcThread::WorkerLoop* function which is found below *WaitForSingleObjectEx* function on both stack trace and raw stack data from search results, we see that the former function calls the latter function indeed:

```
0:000> uf ole32!CRpcThread::WorkerLoop
ole32!CRpcThread::WorkerLoop:
776b22e9 mov     edi,edi
776b22eb push    esi
776b22ec mov     esi,ecx
776b22ee cmp     dword ptr [esi+4],0
776b22f2 jne     ole32!CRpcThread::WorkerLoop+0x67 (776b234d)

ole32!CRpcThread::WorkerLoop+0xb:
776b22f4 push    ebx
776b22f5 push    edi
776b22f6 mov     edi,dword ptr [ole32!_imp__WaitForSingleObjectEx (77671304)]
776b22fc mov     ebx,7530h

ole32!CRpcThread::WorkerLoop+0x18:
776b2301 push    dword ptr [esi+0Ch]
776b2304 call    dword ptr [esi+8]
776b2307 call    dword ptr [ole32!_imp__GetCurrentThread (7767130c)]
776b230d push    eax

776b230e call    dword ptr [ole32!_imp__RtlCheckForOrphanedCriticalSection (77671564)]
776b2314 xor     eax,eax
776b2316 cmp     dword ptr [esi],eax
776b2318 mov     dword ptr [esi+8],eax
776b231b mov     dword ptr [esi+0Ch],eax
776b231e je     ole32!CRpcThread::WorkerLoop+0x65 (776b234b)

ole32!CRpcThread::WorkerLoop+0x37:
776b2320 push    esi
776b2321 mov     ecx,offset ole32!gRpcThreadCache (7778fc28)
776b2326 call    ole32!CRpcThreadCache::AddToFreeList (776de78d)

ole32!CRpcThread::WorkerLoop+0x55:
776b232b push    0
776b232d push    ebx
776b232e push    dword ptr [esi]
776b2330 call    edi
776b2332 test    eax,eax
776b2334 je     ole32!CRpcThread::WorkerLoop+0x60 (776cf3be)

ole32!CRpcThread::WorkerLoop+0x44:
776b233a push    esi
776b233b mov     ecx,offset ole32!gRpcThreadCache (7778fc28)
776b2340 call    ole32!CRpcThreadCache::RemoveFromFreeList (776e42de)
776b2345 cmp     dword ptr [esi+4],0
776b2349 je     ole32!CRpcThread::WorkerLoop+0x55 (776b232b)
```

```

ole32!CRpcThread::WorkerLoop+0x65:
776b234b pop     edi
776b234c pop     ebx

ole32!CRpcThread::WorkerLoop+0x67:
776b234d pop     esi
776b234e ret

ole32!CRpcThread::WorkerLoop+0x60:
776cf3be cmp     dword ptr [esi+4],eax
776cf3c1 je      ole32!CRpcThread::WorkerLoop+0x18 (776b2301)

ole32!CRpcThread::WorkerLoop+0x69:
776cf3c7 jmp     ole32!CRpcThread::WorkerLoop+0x65 (776b234b)

```

Therefore, we have possibly identified the thread #16 that resets the event by calling *WaitForSingleObjectEx* and tries to acquire the critical section. We also know the second thread #13 that has already acquired that critical section and now is waiting for the event to be signaled.

## Comments

---

Regarding thread#16 we can see the module it was trying to free if we dump parameters for *FreeLibrary*:

```
0581fe14 776b46fb kernel32!FreeLibrary+0x41
```

In another memory dump, we had this:

```

16 Id: 13d0.2780 Suspend: 1 Peb: 7ffa8000 Unfrozen
ChildEBP RetAddr Args to Child
0210fc98 7c942124 7c95970f 00000100 00000000 ntdll!KiFastSystemCallRet
0210fc9c 7c95970f 00000100 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
0210fcdb 7c959620 00000000 00000004 00000000 ntdll!RtlpWaitOnCriticalSection+0x19c
0210fcf8 7c95a023 7c9a9d94 0210fe78 0210fea8 ntdll!RtlEnterCriticalSection+0xa8
0210fe00 7c827bcd 02a10000 0210fea8 0210fee4 ntdll!LdrUnloadDll+0x35
0210fe14 775346eb 02a10000 0210ff0c 775346ff kernel32!FreeLibrary+0x41
[...]
0:000> lm
start end module name
[...]
02a10000 02a19000 DLL_A DLL_A.dll

```

## Self

This is a variation of **Deadlock** pattern (page 194) where a thread that owns a resource (either in shared or exclusive mode) attempts to acquire it exclusively again. This results in a self-deadlock:

```
Resource @ 0x85d9c018      Shared 1 owning threads
Contention Count = 2
NumberOfExclusiveWaiters = 2
Threads: 85db0030-02<*>
Threads Waiting On Exclusive Access:
    85f07d78      85db0030
```

## Comments

One of the users of Software Diagnostics Library commented that this could happen only if the resource had been first acquired as shared and then exclusively. According to WDK, resources support recursive locking. The only scenario which is not supported (and which may cause deadlocks) is acquiring a resource as shared and then exclusively.

## Debugger Bug

When doing software behavior artifact collection, live debugging, or postmortem memory dump analysis we must also take into consideration the possibility of **Debugger Bugs**. We classify them into hard and soft bugs. The former are those software defects and behavioral problems that result in further abnormal software behavior incidents like crashes and hangs. One example is this Microsoft KB article about DebugDiag<sup>35</sup>. Soft debugger bugs usually manifest themselves as glitches in data output, nonsense or false positive diagnostics, for example, the following excessive non-paged pool usage message in the output from !vm WinDbg command (see the corresponding MS KB article<sup>36</sup>):

```
1: kd> !vm

*** Virtual Memory Usage ***
Physical Memory: 1031581 ( 4126324 Kb)
Page File: \??\C:\pagefile.sys
Current: 4433524 Kb Free Space: 4433520 Kb
Minimum: 4433524 Kb Maximum: 12378972 Kb
Unimplemented error for MiSystemVaTypeCount
Available Pages: 817652 ( 3270608 Kb)
ResAvail Pages: 965229 ( 3860916 Kb)
Locked IO Pages: 0 ( 0 Kb)
Free System PTEs: 33555714 ( 134222856 Kb)
Modified Pages: 15794 ( 63176 Kb)
Modified PF Pages: 15793 ( 63172 Kb)
NonPagedPool Usage: 88079121 ( 352316484 Kb)
NonPagedPoolNx Usage: 12885 ( 51540 Kb)
NonPagedPool Max: 764094 ( 3056376 Kb)
***** Excessive NonPaged Pool Usage *****
PagedPool 0 Usage: 35435 ( 141740 Kb)
PagedPool 1 Usage: 3620 ( 14480 Kb)
PagedPool 2 Usage: 573 ( 2292 Kb)
PagedPool 3 Usage: 535 ( 2140 Kb)
PagedPool 4 Usage: 538 ( 2152 Kb)
PagedPool Usage: 40701 ( 162804 Kb)
PagedPool Maximum: 33554432 ( 134217728 Kb)
Session Commit: 9309 ( 37236 Kb)
Shared Commit: 6460 ( 25840 Kb)
Special Pool: 0 ( 0 Kb)
Shared Process: 5760 ( 23040 Kb)
PagedPool Commit: 40765 ( 163060 Kb)
Driver Commit: 2805 ( 11220 Kb)
Committed pages: 212472 ( 849888 Kb)
Commit limit: 2139487 ( 8557948 Kb)
```

<sup>35</sup> <http://support.microsoft.com/kb/970107/>

<sup>36</sup> <http://support.microsoft.com/kb/2509968/>

## Debugger Omission

If some false positives can be considered soft **Debugger Bugs** (page 219), false negatives can have more severe impact on software behavior analysis, especially in the malware analysis. A typical example here is the current `.imgscan` command which according to the documentation should by default scan virtual process space for MZ/PE signatures. Unfortunately it doesn't detect such signatures in resource pages (we haven't checked stack regions yet):

```
0000000000fd0000 image base

SECTION HEADER #4
.rsrc name
6430 virtual size
4000 virtual address
6600 size of raw data
1600 file pointer to raw data
[...]
40000040 flags
Initialized Data
(no align specified)
Read Only

0:000> .imgscan /r 00000000`00fd4000 L200

0:000> s -[12]sa 00000000`00fd4000 1200
00000000`00fd40b0  "MZ"
00000000`00fd40fd  "!This program cannot be run in D"
00000000`00fd411d  "OS mode."
00000000`00fd4188  "Rich"
00000000`00fd4198  "PE"

0:000> !dh 00000000`00fd40b0

File Type: DLL
FILE HEADER VALUES
14C machine (i386)
3 number of sections
time date stamp Fri Jan 18 21:27:25 2013

0 file pointer to symbol table
0 number of symbols
E0 size of optional header
2102 characteristics
Executable
32 bit word machine
DLL
[...]
```

Other analysis scenarios include `!analyze -v` that shows us a breakpoint instead of an exception violation from a parallel thread.

## Design Value

The pattern called **Small Value** (page 873) deals with easily recognizable values such as handles, timeouts, mouse pointer coordinates, enumeration values, and window messages. There is another kind of values, for example, 256 (+/- 1) or some other round value. Here we can also add some regular patterns in hex representation such as window handles or flags, for example, 0x10008000. Such designed values may fall into some module range, the so-called **Coincidental Symbolic Information** (page 134) pattern. They may not necessarily be stack trace parameters (which can also be **False Function Parameters**, page 390). If we see a design value in the output of WinDbg commands, especially related to abnormal behavior patterns, then it might point to some reached design limitations. For example, **Blocked ALPC Queue** (page 77) may have a limitation on I/O completion port<sup>37</sup>. We observed that when we had ALPC **Wait Chains** (page 1097) in one unresponsive system:

```
0: kd> !alpc /p <port_address>
[...]
512 thread(s) are registered with port IO completion object:
[...]
```

---

<sup>37</sup> Understanding I/O Completion Ports, Memory Dump Analysis Anthology, Volume 1, page 653

## Deviant Module

When looking at the module list (**!lmv**), searching for modules (**.imgscan**) or examining the particular module (**!address**, **!dh**) we may notice one of them as deviant. The deviation may be in (but not limited to) as anything is possible:

- suspicious module name
- suspicious protection
- suspicious module load address

```
0:005> .imgscan
MZ at 00040000, prot 00000040, type 00020000 - size 1d000
MZ at 00340000, prot 00000002, type 01000000 - size 9c000
Name: iexplore.exe
MZ at 02250000, prot 00000002, type 00040000 - size 2000
MZ at 023b0000, prot 00000002, type 01000000 - size b000
Name: msimtf.dll
MZ at 03f80000, prot 00000002, type 00040000 - size 2000
MZ at 10000000, prot 00000004, type 00020000 - size 5000
Name: screens_dll.dll
MZ at 16080000, prot 00000002, type 01000000 - size 25000
Name: mdnsNSP.dll
MZ at 6ab50000, prot 00000002, type 01000000 - size 26000
Name: DSSENH.dll
MZ at 6b030000, prot 00000002, type 01000000 - size 5b0000
Name: MSHTML.dll
MZ at 6ba10000, prot 00000002, type 01000000 - size b4000
Name: JSCRIPT.dll
MZ at 6cec0000, prot 00000002, type 01000000 - size 1b000
Name: CRYPTNET.dll
MZ at 6d260000, prot 00000002, type 01000000 - size e000
Name: PNGFILTER.DLL
MZ at 6d2f0000, prot 00000002, type 01000000 - size 29000
Name: msls31.dll
MZ at 6d700000, prot 00000002, type 01000000 - size 30000
Name: MLANG.dll
MZ at 6d740000, prot 00000002, type 01000000 - size 4d000
Name: SSV.DLL
MZ at 6d7b0000, prot 00000002, type 01000000 - size c000
Name: ImgUtil.dll
MZ at 6ddb0000, prot 00000002, type 01000000 - size 2f000
Name: iepeers.DLL
MZ at 6df20000, prot 00000002, type 01000000 - size 33000
Name: IESHims.dll
MZ at 6eb80000, prot 00000002, type 01000000 - size a94000
Name: IEFRAFME.dll
MZ at 703b0000, prot 00000002, type 01000000 - size 53000
Name: SWEETPRX.dll
MZ at 70740000, prot 00000002, type 01000000 - size 40000
Name: SWEETPRX.dll
MZ at 725a0000, prot 00000002, type 01000000 - size 12000
Name: PNRPNSP.dll
MZ at 725d0000, prot 00000002, type 01000000 - size 8000
```

```
Name: WINRNR.dll
MZ at 725e0000, prot 00000002, type 01000000 - size 136000
Name: MSXML3.dll
MZ at 72720000, prot 00000002, type 01000000 - size c000
Name: wshbth.dll
MZ at 72730000, prot 00000002, type 01000000 - size f000
Name: NAPINSP.dll
MZ at 72890000, prot 00000002, type 01000000 - size 6000
Name: SensApi.dll
MZ at 72ec0000, prot 00000002, type 01000000 - size 42000
Name: WINSPOOL.DRV
MZ at 734b0000, prot 00000002, type 01000000 - size 6000
Name: rasadhlp.dll
MZ at 736b0000, prot 00000002, type 01000000 - size 85000
Name: COMCTL32.dll
MZ at 73ac0000, prot 00000002, type 01000000 - size 7000
Name: MIDIMAP.dll
MZ at 73ae0000, prot 00000002, type 01000000 - size 14000
Name: MSACM32.dll
MZ at 73b00000, prot 00000002, type 01000000 - size 66000
Name: audioeng.dll
MZ at 73c30000, prot 00000002, type 01000000 - size 9000
Name: MSACM32.DRV
MZ at 73c60000, prot 00000002, type 01000000 - size 21000
Name: AudioSes.DLL
MZ at 73c90000, prot 00000002, type 01000000 - size 2f000
Name: WINMMDRV.dll
MZ at 74290000, prot 00000002, type 01000000 - size bb000
Name: PROPSYS.dll
MZ at 74390000, prot 00000002, type 01000000 - size f000
Name: nlaapi.dll
MZ at 743a0000, prot 00000002, type 01000000 - size 4000
Name: ksuser.dll
MZ at 74430000, prot 00000002, type 01000000 - size 15000
Name: Cabinet.dll
MZ at 74450000, prot 00000002, type 01000000 - size 3d000
Name: OLEACC.dll
MZ at 74490000, prot 00000002, type 01000000 - size 1ab000
Name: gdiplus.dll
MZ at 74640000, prot 00000002, type 01000000 - size 28000
Name: MMDevAPI.DLL
MZ at 74670000, prot 00000002, type 01000000 - size 32000
Name: WINMM.dll
MZ at 746b0000, prot 00000002, type 01000000 - size 31000
Name: TAPI32.dll
MZ at 749e0000, prot 00000002, type 01000000 - size 19e000
Name: COMCTL32.dll
MZ at 74b80000, prot 00000002, type 01000000 - size 7000
Name: AVRT.dll
MZ at 74ba0000, prot 00000002, type 01000000 - size 4a000
Name: RASAPI32.dll
MZ at 74ce0000, prot 00000002, type 01000000 - size 3f000
Name: UxTheme.dll
MZ at 74de0000, prot 00000002, type 01000000 - size 2d000
Name: WINTRUST.dll
MZ at 74ea0000, prot 00000002, type 01000000 - size 14000
```

```
Name: rasman.dll
MZ at 74f70000, prot 00000002, type 01000000 - size c000
Name: rtutils.dll
MZ at 74f80000, prot 00000002, type 01000000 - size 5000
Name: WSHTCPIP.dll
MZ at 74fb0000, prot 00000002, type 01000000 - size 21000
Name: NTMARTA.dll
MZ at 75010000, prot 00000002, type 01000000 - size 3b000
Name: RSAENH.dll
MZ at 75050000, prot 00000002, type 01000000 - size 5000
Name: MSIMG32.dll
MZ at 75060000, prot 00000002, type 01000000 - size 15000
Name: GPAPI.dll
MZ at 750a0000, prot 00000002, type 01000000 - size 46000
Name: SCHANNEL.dll
MZ at 752b0000, prot 00000002, type 01000000 - size 3b000
Name: MSWSOCK.dll
MZ at 75370000, prot 00000002, type 01000000 - size 45000
Name: bcrypt.dll
MZ at 753f0000, prot 00000002, type 01000000 - size 5000
Name: WSHIP6.dll
MZ at 75400000, prot 00000002, type 01000000 - size 8000
Name: VERSION.dll
MZ at 75420000, prot 00000002, type 01000000 - size 7000
Name: CREDSSP.dll
MZ at 75430000, prot 00000002, type 01000000 - size 35000
Name: ncrypt.dll
MZ at 75480000, prot 00000002, type 01000000 - size 22000
Name: dhcpcsvc6.DLL
MZ at 754b0000, prot 00000002, type 01000000 - size 7000
Name: WINNSI.DLL
MZ at 754c0000, prot 00000002, type 01000000 - size 35000
Name: dhcpcsvc.DLL
MZ at 75500000, prot 00000002, type 01000000 - size 19000
Name: IPHLAPI.DLL
MZ at 75590000, prot 00000002, type 01000000 - size 3a000
Name: slc.dll
MZ at 755d0000, prot 00000002, type 01000000 - size f2000
Name: CRYPT32.dll
MZ at 75740000, prot 00000002, type 01000000 - size 12000
Name: MSASN1.dll
MZ at 75760000, prot 00000002, type 01000000 - size 11000
Name: SAMLIB.dll
MZ at 75780000, prot 00000002, type 01000000 - size 76000
Name: NETAPI32.dll
MZ at 75800000, prot 00000002, type 01000000 - size 2c000
Name: DNSAPI.dll
MZ at 75a70000, prot 00000002, type 01000000 - size 5f000
Name: sxs.dll
MZ at 75ad0000, prot 00000002, type 01000000 - size 2c000
Name: apphelp.dll
MZ at 75b30000, prot 00000002, type 01000000 - size 14000
Name: Secur32.dll
MZ at 75b50000, prot 00000002, type 01000000 - size 1e000
Name: USERENV.dll
MZ at 75c90000, prot 00000002, type 01000000 - size 7000
```

Name: PSAPI.DLL  
MZ at 75ca0000, prot 00000002, type 01000000 - size c3000  
Name: RPCRT4.dll  
MZ at 75d70000, prot 00000002, type 01000000 - size 73000  
Name: COMDLG32.dll  
MZ at 75df0000, prot 00000002, type 01000000 - size 9000  
Name: LPK.dll  
MZ at 75e00000, prot 00000002, type 01000000 - size dc000  
Name: KERNEL32.dll  
MZ at 75ee0000, prot 00000002, type 01000000 - size aa000  
Name: msvcrt.dll  
MZ at 75f90000, prot 00000002, type 01000000 - size 1e8000  
Name: iertutil.dll  
MZ at 76180000, prot 00000002, type 01000000 - size 29000  
Name: imagehlp.dll  
MZ at 761b0000, prot 00000002, type 01000000 - size 6000  
Name: NSI.dll  
MZ at 761c0000, prot 00000002, type 01000000 - size 84000  
Name: CLBCatQ.DLL  
MZ at 76250000, prot 00000002, type 01000000 - size 49000  
Name: WLDAP32.dll  
MZ at 762a0000, prot 00000002, type 01000000 - size c6000  
Name: ADVAPI32.dll  
MZ at 76370000, prot 00000002, type 01000000 - size 4b000  
Name: GDI32.dll  
MZ at 763c0000, prot 00000002, type 01000000 - size 59000  
Name: SHLWAPI.dll  
MZ at 76420000, prot 00000002, type 01000000 - size e6000  
Name: WININET.dll  
MZ at 76510000, prot 00000002, type 01000000 - size b10000  
Name: SHELL32.dll  
MZ at 77020000, prot 00000002, type 01000000 - size 145000  
Name: ole32.dll  
MZ at 77170000, prot 00000002, type 01000000 - size 7d000  
Name: USP10.dll  
MZ at 771f0000, prot 00000002, type 01000000 - size 8d000  
Name: OLEAUT32.dll  
MZ at 77280000, prot 00000002, type 01000000 - size 18a000  
Name: SETUPAPI.dll  
MZ at 77410000, prot 00000002, type 01000000 - size 9d000  
Name: USER32.dll  
MZ at 774b0000, prot 00000002, type 01000000 - size 133000  
Name: urlmon.dll  
MZ at 775f0000, prot 00000002, type 01000000 - size 127000  
Name: ntdll.dll  
MZ at 77720000, prot 00000002, type 01000000 - size 3000  
Name: Normaliz.dll  
MZ at 77730000, prot 00000002, type 01000000 - size 2d000  
Name: WS2\_32.dll  
MZ at 77760000, prot 00000002, type 01000000 - size 1e000  
Name: IMM32.dll  
MZ at 77780000, prot 00000002, type 01000000 - size c8000  
Name: MSCTF.dll  
MZ at 7c340000, prot 00000002, type 01000000 - size 56000  
Name: MSVCR71.dll

```
0:005> !address 00040000
Usage: <unclassified>
Allocation Base: 00040000
Base Address: 00040000
End Address: 0005d000
Region Size: 0001d000
Type: 00020000 MEM_PRIVATE
State: 00001000 MEM_COMMIT
Protect: 00000040 PAGE_EXECUTE_READWRITE

0:005> !address 10000000
Usage: <unclassified>
Allocation Base: 10000000
Base Address: 10000000
End Address: 10001000
Region Size: 00001000
Type: 00020000 MEM_PRIVATE
State: 00001000 MEM_COMMIT
Protect: 00000004 PAGE_READWRITE
```

- suspicious text inside<sup>38</sup>
- suspicious import table (for example, screen grabbing) or its absence (dynamic imports)

```
0:005> !dh 10000000
[...]
2330 [ 50] address [size] of Export Directory
20E0 [ 78] address [size] of Import Directory
0 [ 0] address [size] of Resource Directory
0 [ 0] address [size] of Exception Directory
0 [ 0] address [size] of Security Directory
4000 [ 34] address [size] of Base Relocation Directory
2060 [ 1C] address [size] of Debug Directory
0 [ 0] address [size] of Description Directory
0 [ 0] address [size] of Special Directory
0 [ 0] address [size] of Thread Storage Directory
0 [ 0] address [size] of Load Configuration Directory
0 [ 0] address [size] of Bound Import Directory
2000 [ 58] address [size] of Import Address Table Directory
0 [ 0] address [size] of Delay Import Directory
0 [ 0] address [size] of COR20 Header Directory
0 [ 0] address [size] of Reserved Directory
[...]
```

<sup>38</sup> Crash Dump Analysis of Defective Malware, Memory Dump Analysis Anthology, Volume 5, page 406

```
0:005> dps 10000000+2000 10000000+2000+58
10002000 76376101 gdi32!CreateCompatibleDC
10002004 763793d6 gdi32!StretchBlt
10002008 76377461 gdi32!CreateDIBSection
1000200c 763762a0 gdi32!SelectObject
10002010 00000000
10002014 75e4a411 kernel32!lstrcmpW
10002018 75e440aa kernel32!VirtualFree
1000201c 75e4ad55 kernel32!VirtualAlloc
10002020 00000000
10002024 77429ced user32!ReleaseDC
10002028 77423ba7 user32!NtUserGetWindowDC
1000202c 77430e21 user32!GetWindowRect
10002030 00000000
10002034 744a75e9 GdiPlus!GdipPlusStartup
10002038 744976dd GdiPlus!GdipSaveImageToStream
1000203c 744cdd38 GdiPlus!GdipGetImageEncodersSize
10002040 744971cf GdiPlus!GdipDisposeImage
10002044 744a8591 GdiPlus!GdipCreateBitmapFromHBITMAP
10002048 744cd8ae GdiPlus!GdipGetImageEncoders
1000204c 00000000
10002050 7707d51b ole32!CreateStreamOnHGlobal
10002054 00000000
10002058 00000000

0:000> !dh 012a0000
[...]
0 [      0] address [size] of Export Directory
0 [      0] address [size] of Import Directory
0 [      0] address [size] of Resource Directory
0 [      0] address [size] of Exception Directory
0 [      0] address [size] of Security Directory
8000 [      FC] address [size] of Base Relocation Directory
4000 [      1C] address [size] of Debug Directory
0 [      0] address [size] of Description Directory
0 [      0] address [size] of Special Directory
0 [      0] address [size] of Thread Storage Directory
0 [      0] address [size] of Load Configuration Directory
0 [      0] address [size] of Bound Import Directory
0 [      0] address [size] of Import Address Table Directory
0 [      0] address [size] of Delay Import Directory
0 [      0] address [size] of COR20 Header Directory
0 [      0] address [size] of Reserved Directory
[...]
```

- suspicious path names

```
Age: 7, Pdb: d:\work\BekConnekt\Client_src_code_New\Release\Blackjoe_new.pdb
```

```
Debug Directories(1)
Type Size Address Pointer
cv 46 2094 894 Format: RSDS, guid, 1, C:\MyWork\screens_dll\Release\screens_dll.pdb
```

- suspicious image path (although could be just dynamic code generation for .NET assemblies)
- uninitialized image resources

```
0:002> lm v m C6DC
start    end      module name
012a0000 012a9000  C6DC      C (no symbols)
Loaded symbol image file: C6DC.tmp
Image path: C:\Users\User\AppData\Local\Temp\C6DC.tmp
Image name: C6DC.tmp
Timestamp:      Sun May 30 20:18:32 2010 (4C02BA08)
CheckSum:        00000000
ImageSize:       00009000
File version:   0.0.0.0
Product version: 0.0.0.0
File flags:      0 (Mask 0)
File OS:         0 Unknown Base
File type:       0.0 Unknown
File date:       00000000.00000000
Translations:    0000.04b0 0000.04e4 0409.04b0 0409.04e4
```

## Comments

.imgscan might not be able to find all hidden modules (**Debugger Omission** pattern, page 220).

Timestamps might be suspicious too.

## Deviant Token

Sometimes we need to check what security principal or group we run a process under, or what privileges it has or whether it has impersonating threads. We may find an unexpected token with a different security identifier, for example, *Network Service* instead of *Local System* (SID: S-1-5-18):

```
PROCESS 8f218d88 SessionId: 0 Cid: 09c4 Peb: 7ffdf000 ParentCid: 0240
DirBase: bffd4260 ObjectTable: e10eae90 HandleCount: 93.
Image: ServiceA.exe
VadRoot 8f1f70e8 Vads 141 Clone 0 Private 477. Modified 2. Locked 0.
DeviceMap e10038d8
Token e10ff5d8
[...]

0: kd> !token e10ff5d8
_TOKEN e10ff5d8
TS Session ID: 0
User: S-1-5-20
[...]
```

Well-known SIDs can be found in MS KB article 243330<sup>39</sup>.

---

<sup>39</sup> <http://support.microsoft.com/kb/243330>

## Diachronic Module

When we have a performance issue, we may request a set of consecutive memory dump saved after some interval. In such memory dumps we may see the same thread(s) having similar stack trace(s). In this simple diagnostic scenario we may diagnose several patterns based on the stack traces: **Active Threads** (page 66) that can be **Spiking Threads** (page 885) with **Spike Intervals** (page 884) or stable, not changing, **Wait Chains** (for example, critical sections, page 1086). Here we may easily identify **Top** (page 1012) active and **Blocking** (page 96) modules based on **Module Wait Chain** (page 1103).

The more complex case arises when we have different **Active Threads** and/or **Wait Chains** with different thread IDs at different times. However, if their **Top Module** is the same, we may have found it as a performance root cause component especially in the case of **Active Threads** since it is statistically probable that such threads were active for considerable time deltas around the snapshot times (since threads are usually waiting). Such hypothesis may also be confirmed by inter-correlational analysis (**Inter-Correlation**<sup>40</sup>) with software traces and logs where we can see **Thread of Activity**<sup>41</sup> **Discontinuities**<sup>42</sup> and **Time Deltas**<sup>43</sup>.

We call this analysis pattern **Diachronic Module** since we see the module component appears in different thread stack traces diachronically (at different times). The typical simplified scenario is illustrated in this diagram:

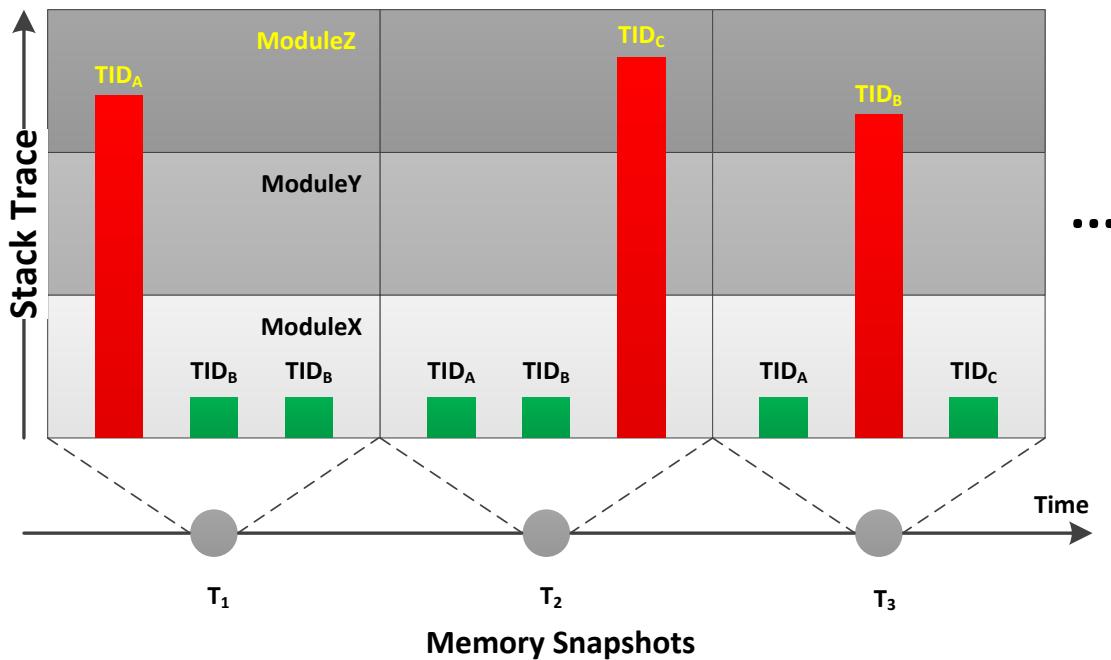
---

<sup>40</sup> Memory Dump Analysis Anthology, Volume 4, page 350

<sup>41</sup> Ibid., page 339

<sup>42</sup> Ibid., page 341

<sup>43</sup> Memory Dump Analysis Anthology, Volume 5, page 282



This analysis pattern is different from synchronous module case (the module component appears in different thread stack traces at the same time) which was named **Ubiquitous Component** (page 1020).

## Dialog Box

Similar to **Message Box** (page 660) and **String Parameter** (page 962) patterns we also have **Dialog Box** pattern where we can see dialog window caption and contents when we examine function parameters. Although in the examples below we know the dialog purpose from the friendly call stack function names, for many 3rd-party applications we either don't have symbols or such helper functions, but we want to know what was on the screen when screenshots were not collected.

The first two examples are from *Notepad* application, and the third is from IE:

```
0:000> kv
ChildEBP RetAddr Args to Child
0017f5c4 777b073f 777c3c9f 000d023c 00000001 ntdll!KiFastSystemCallRet
0017f5c8 777c3c9f 000d023c 00000001 00000000 user32!NtUserWaitMessage+0xc
0017f5fc 777c2dc0 00310778 000d023c 00000001 user32!DialogBox2+0x202
0017f624 777c2eec 76460000 02a6bc60 000d023c user32!InternalDialogBox+0xd0
0017f644 76489a65 76460000 02a6bc60 000d023c user32!DialogBoxIndirectParamAorW+0x37
0017f680 76489ccf 0017f68c 00000001 0017f6d4 comdlg32!ChooseFontX+0x1ba
0017f6bc 006741c7 0017f6d4 00000111 00000000 comdlg32!ChooseFontW+0x2e
0017f734 0067164a 000d023c 00000021 00000000 notepad!NPCommand+0x4c7
0017f758 777af72 000d023c 00000111 00000021 notepad!NPWndProc+0x4cf
0017f784 777afe4a 0067146c 000d023c 00000111 user32!InternalCallWinProc+0x23
0017f7fc 777b018d 00000000 0067146c 000d023c user32!UserCallWinProcCheckWow+0x14b
0017f860 777b022b 0067146c 00000000 0017f8a4 user32!DispatchMessageWorker+0x322
0017f870 00671465 0017f888 00000000 0067a21c user32!DispatchMessageW+0xf
0017f8a4 0067195d 00670000 00000000 00231cfa notepad!WinMain+0xe3
0017f934 7652d0e9 7ffd9000 0017f980 77b019bb notepad!_initterm_e+0x1a1
0017f940 77b019bb 7ffd9000 78f7b908 00000000 kernel32!BaseThreadInitThunk+0xe
0017f980 77b0198e 006731ed 7ffd9000 00000000 ntdll!__RtlUserThreadStart+0x23
0017f998 00000000 006731ed 7ffd9000 00000000 ntdll!_RtlUserThreadStart+0x1b

0:000> dc 02a6bc60 150
02a6bc60 80c800c4 00000000 000d0014 011f0036 .....6...
02a6bc70 000000c4 00460000 006e006f 00000074 .....F.o.n.t...
02a6bc80 004d0008 00200053 00680053 006c0065 ..M.S. .S.h.e.1.
02a6bc90 0020006c 006c0044 00000067 50020000 1. .D.1.g.....P
02a6bc9a0 00000000 00070007 00090028 ffff0440 .....(@...
02a6bc9b0 00260082 006f0046 0074006e 0000003a ..&.F.o.n.t. .....
02a6bcc0 00000000 50210b51 00000000 00100007 ....Q.!P. .....
02a6bcd0 004c0062 ffff0470 00000085 00000000 b.L.p. .....
02a6bcce0 50020000 00000000 0007006e 0009002c ...P....n.,...
02a6bcf0 ffff0441 00460082 006e006f 00200074 A.....F.o.n.t. .
02a6bd00 00740073 00790026 0065006c 0000003a s.t.&.y.l.e. .....
02a6bd10 00000000 50210041 00000000 0010006e ....A.!P....n...
02a6bd20 004c004a ffff0471 00000085 00000000 J.L.q. .....
02a6bd30 50020000 00000000 000700bd 0009001e ...P. .....
02a6bd40 ffff0442 00260082 00690053 0065007a B.....&.S.i.z.e.
02a6bd50 0000003a 00000000 50210b51 00000000 :.....Q.!P....
02a6bd60 001000be 004c0024 ffff0472 00000085 ....$.L.r. .....
02a6bd70 00000000 50020007 00000000 00610007 .....P. .....
02a6bd80 00480062 ffff0430 00450080 00660066 b.H.0.....E.f.f.
02a6bd90 00630065 00730074 00000000 50010003 e.c.t.s. .....
```

0:000> kv  
ChildEBP RetAddr Args to Child  
0017f5a8 777b073f 777c3c9f 000d023c 00000001 ntdll!KiFastSystemCallRet  
0017f5ac 777c3c9f 000d023c 00000001 00000000 user32!NtUserWaitMessage+0xc  
0017f5e0 777c2dc0 0044034a 000d023c 00000001 user32!DialogBox2+0x202  
0017f608 777c2eec 768a0000 **029030bc** 000d023c user32!InternalDialogBox+0xd0  
0017f628 777c10ef 768a0000 029030bc 000d023c user32!DialogBoxIndirectParamAorW+0x37  
0017f64c 7695d877 768a0000 00003810 000d023c user32!DialogBoxParamW+0x3f  
0017f670 76a744dc 768a0000 00003810 000d023c shell32!SHFusionDialogBoxParam+0x32  
0017f6b0 00674416 000d023c 002530dc 00672fc4 shell32!ShellAboutW+0x4d  
0017f734 0067164a 000d023c 00000041 00000000 notepad!NPCommand+0x718  
0017f758 777af72 000d023c 00000111 00000041 notepad!NPWndProc+0x4cf  
0017f784 777afe4a 0067146c 000d023c 00000111 user32!InternalCallWinProc+0x23  
0017f7fc 777b018d 00000000 0067146c 000d023c user32!UserCallWinProcCheckWow+0x14b  
0017f860 777b022b 0067146c 00000000 0017f8a4 user32!DispatchMessageWorker+0x322  
0017f870 00671465 0017f888 00000000 0067a21c user32!DispatchMessageW+0xf  
0017f8a4 0067195d 00670000 00000000 00231cfa notepad!WinMain+0xe3  
0017f934 7652d0e9 7ffd9000 0017f980 77b019bb notepad!\_initterm\_e+0x1a1  
0017f940 77b019bb 7ffd9000 78f7b908 00000000 kernel32!BaseThreadInitThunk+0xe  
0017f980 77b0198e 006731ed 7ffd9000 00000000 ntdll!\_RtlUserThreadStart+0x23  
0017f998 00000000 006731ed 7ffd9000 00000000 ntdll!\_RtlUserThreadStart+0x1b

0:000> dc 029030bc 150  
029030bc fffff0001 00000000 00000000 80c800cc .....  
029030cc 0014000c 01130014 000000ee 00410000 .....A.  
029030dc 006f0062 00740075 00250020 00000073 b.o.u.t. %.s...  
029030ec 00000008 004d0000 00200053 00680053 .....M.S. .S.h.  
029030fc 006c0065 0020006c 006c0044 00000067 e.l.l. .D.l.g...  
0290310c 00000000 00000000 50000043 00370007 .....C..P..7.  
0290311c 00140015 00003009 0082ffff 0000ffff .....0.....  
0290312c 00000000 00000000 0000008c 5000008c .....P  
0290313c 00370023 000a00c8 00003500 0082ffff #.7.....5.....  
0290314c 00000000 00000000 00000000 5000008c .....P  
0290315c 00410023 000a00eb 0000350b 0082ffff #.A.....5.....  
0290316c 00000000 00000000 00000000 50000080 .....P  
0290317c 004b0023 000a00d2 0000350a 0082ffff #.K.....5.....  
0290318c 00000000 00000000 00000000 50000080 .....P  
0290319c 00550023 002800d2 00003513 0082ffff #.U....(.5.....  
029031ac 00680054 00200065 00570025 004e0049 T.h.e. %.W.I.N.  
029031bc 004f0044 00530057 004c005f 004e004f D.O.W.S.\_.L.O.N.  
029031cc 00250047 006f0020 00650070 00610072 G.% .o.p.e.r.a.  
029031dc 00690074 0067006e 00730020 00730079 t.i.n.g. .s.y.s.  
029031ec 00650074 0020006d 006e0061 00200064 t.e.m. .a.n.d. .

16 Id: 10fc.124c Suspend: 0 Teb: 7ffd7000 Unfrozen  
ChildEBP RetAddr Args to Child  
053f8098 777b073f 777c3c9f 003d0650 00000001 ntdll!KiFastSystemCallRet  
053f809c 777c3c9f 003d0650 00000001 00000000 user32!NtUserWaitMessage+0xc  
053f80d0 777c2dc0 002e0378 003d0650 00000001 user32!DialogBox2+0x202  
053f80f8 777c2eec 6f270000 **03387bd4** 003d0650 user32!InternalDialogBox+0xd0  
053f8118 777c10ef 6f270000 03387bd4 003d0650 user32!DialogBoxIndirectParamAorW+0x37  
053f813c 6f2c5548 6f270000 00005398 003d0650 user32!DialogBoxParamW+0x3f  
053f8164 6f2c5743 6f270000 00005398 003d0650 ieframe!Detour\_DialogBoxParamW+0x47  
053f8188 6f2c56f5 6f270000 00005398 001905ea ieframe!SHFusionDialogBoxParam+0x32  
053f9228 6f2c5378 001905ea 053fb540 00000104 ieframe!DoAddToFavDlgEx+0xcf  
053fb5b5c 6f2c58f9 001905ea 0e69a0c0 053fbff0 ieframe!AddToFavoritesEx+0x349

```

053fdbdb8 6f2c57ee 00000000 053fbff0 00000000 ieframe!CBaseBrowser2::_AddToFavorites+0xe9
053fc0f4 6f2c3e5e 00000000 00000000 00000001 ieframe!CBaseBrowser2::_ExecAddToFavorites+0x123
053fc124 6f39ca4e 6f39c524 00000008 00000001 ieframe!CBaseBrowser2::_ExecExplorer+0xbe
053fc14c 6f39cee8 114ea39c 6f39c524 00000008 ieframe!CBaseBrowser2::Exec+0x12d
053fc17c 6f39cf17 6f39c524 00000008 00000001 ieframe!CShellBrowser2::_Exec_CCommonBrowser+0x80
053fc414 6f498284 114ea39c 6f39c524 00000008 ieframe!CShellBrowser2::Exec+0x626
053fc43c 6f49e5cd 0000a173 00000000 ffffff71 ieframe!CShellBrowser2::_FavoriteOnCommand+0x75
053fc458 6f3c5ea8 0000a173 00000000 00000111 ieframe!CShellBrowser2::_OnDefault+0x3e
053fd6f0 6f394194 0000a173 00000000 0000031a ieframe!CShellBrowser2::v_OnCommand+0xa7b
053fd70c 6f39898d 001905ea 00000111 0000a173 ieframe!CBaseBrowser2::v_WndProc+0x247
053fd770 6f3988db 001905ea 00000111 0000a173 ieframe!CShellBrowser2::v_WndProc+0x3fe
053fd794 777af72 001905ea 00000111 0000a173 ieframe!CShellBrowser2::s_WndProc+0xfb
053fd7c0 777afe4a 6f39887a 001905ea 00000111 user32!InternalCallWinProc+0x23
053fd838 777b0943 00000000 6f39887a 001905ea user32!UserCallWinProcCheckWow+0x14b
053fd878 777b0b36 00252838 01223dc0 0000a173 user32!SendMessageWorker+0x4b7
053fd898 6f3cf032 001905ea 00000111 0000a173 user32!SendMessageW+0x7c
053fd8d0 6f396ead 0056049c 00000111 0000a173 ieframe!CIInternetToolbarHost::v_WndProc+0xf8
053fd8f4 777af72 0056049c 00000111 0000a173 ieframe!CImpWndProc::s_WndProc+0x65
053fd920 777afe4a 6f396e6e 0056049c 00000111 user32!InternalCallWinProc+0x23
053fd998 777b018d 00000000 6f396e6e 0056049c user32!UserCallWinProcCheckWow+0x14b
053fd9fc 777b022b 6f396e6e 00000000 053ffb14 user32!DispatchMessageWorker+0x322
053fd9fc 6f39c1f5 053fd9fc 00000000 10eec4c0 user32!DispatchMessageW+0xf
053ffb14 6f34337f 0e7c3708 00000000 11bd8dc8 ieframe!CTabWindow::_TabWindowThreadProc+0x54c
053ffbcc 77525179 10eec4c0 00000000 053ffbe8 ieframe!LCIETab_ThreadProc+0x2c1
053ffbdcc 7652d0e9 11bd8dc8 053ffc28 77b019bb iertutil!CIsoScope::RegisterThread+0xab
053ffbe8 77b019bb 11bd8dc8 7dd62326 00000000 kernel32!BaseThreadInitThunk+0xe
053ffc28 77b0198e 7752516b 11bd8dc8 00000000 ntdll!__RtlUserThreadStart+0x23
053ffc40 00000000 7752516b 11bd8dc8 00000000 ntdll!__RtlUserThreadStart+0x1b

0:000> dc 03387bd4 150
03387bd4 ffff0001 00000000 00000000 80c808c0 ..... .
03387be4 0000000a 011f0000 00000064 00410000 .....d.....A.
03387bf4 00640064 00610020 00460020 00760061 d.d. .a. .F.a.v.
03387c04 0072006f 00740069 00000065 00000008 o.r.i.t.e.....
03387c14 004d0000 00200053 00680053 006c0065 ..M.S. .S.h.e.l.
03387c24 0020006c 006c0044 00000067 00000000 1. .D.l.g.....
03387c34 00000000 50000003 0007000f 00140015 .....P.....
03387c44 00009760 0082ffff 00bfffff 00000000 `.....
03387c54 00000000 00000000 50020000 00070035 .....P5...
03387c64 000800db 000003f4 0082ffff 00640041 .....A.d.
03387c74 00200064 00200061 00610046 006f0076 d. .a. .F.a.v.o.
03387c84 00690072 00650074 00000000 00000000 r.i.t.e.....
03387c94 00000000 50020000 00110035 001000db .....P5.....
03387ca4 000003f5 0082ffff 00640041 00200064 .....A.d.d. .
03387cb4 00680074 00730069 00770020 00620065 t.h.i.s. .w.e.b.
03387cc4 00610070 00650067 00610020 00200073 p.a.g.e. .a.s. .
03387cd4 00200061 00610066 006f0076 00690072 a. .f.a.v.o.r.i.
03387ce4 00650074 0020002e 006f0054 00610020 t.e... .T.o. .a.
03387cf4 00630063 00730065 00200073 006f0079 c.c.e.s.s. .y.o.

```

Stack traces with *DialogBoxIndirectParam* call and x64 platform complicate the picture a bit. Please also note that a user may not see the dialog box that you see in a stack trace due to many reasons like terminal session problems or a process running in a non-interactive session.

## Directing Module

In certain software behavior scenarios such as **Memory Leak** (page 650) when we see **Top Modules** (page 1012) calling OS API functions, we may suspect them having defects. However, this might not be the case when these modules were used from a directing module keeping references or handles preventing top modules from freeing memory or releasing resources.

For example, a memory dump from a process had two growing heap segments, and one of them had this recurrent stack trace saved in a user mode **Stack Trace Database** (page 919):

```
38D2CE78: 02ba8 . 02ba8 [07] - busy (2b90), tail fill
Stack trace (38101) at 83e390:
7d6568be: ntdll!RtlAllocateHeapSlowly+0x00000041
7d62b846: ntdll!RtlAllocateHeap+0x00000E9F
337d0572: ModuleA!XHeapAlloc+0x00000115
[...]
338809e2: ModuleA!Execute+0x000002CD
488b3fc1: ModuleB!Execute+0x000000D3
679b8c64: ModuleC!ExecuteByHandle+0x00000074
[...]
67d241cb: ModuleD!Query+0x0000016B
67ba2ed4: ModuleE!Browse+0x000000E4
[...]
667122c6: ModuleF!Check+0x00000126
65e73826: ModuleG!Enum+0x00000406
[...]
```

Initially, we suspected *ModuleA* but found a different recurrent stack trace corresponding to another growing segment:

```
40C81688: 000c8 . 00058 [07] - busy (40), tail fill
Stack trace (38136) at 83f6a4:
7d6568be: ntdll!RtlAllocateHeapSlowly+0x00000041
7d62b846: ntdll!RtlAllocateHeap+0x00000E9F
7c3416b3: msocr71!_heap_alloc+0x000000E0
7c3416db: msocr71!_nh_malloc+0x00000010
67745875: ModuleX!BufAllocate+0x00000015
6775085e: ModuleY!QueryAttribute+0x0000008E
[...]
677502b5: ModuleY!Query+0x00000015
67ba2f19: ModuleE!Browser+0x00000129
[...]
667122c6: ModuleF!Check+0x00000126
65e73826: ModuleG!Enum+0x00000406
[...]
```

From the common stack trace fragment (highlighted in bold italics) we transferred our investigation to *ModuleE*, and indeed, the similar software incident (as per the latter stack trace) was found in a troubleshooting database.

## Disconnected Network Adapter

Sometimes we need to check network adapters (miniports) to see whether they are up, down, connected or disconnected. This can be done using **ndiskd** WinDbg extension and its commands. Here is an example from a kernel memory dump:

```
1: kd> !ndiskd.miniports
raspptp.sys, v0.0
88453360 NetLuidIndex 1, IfIndex 3, WAN Miniport (PPTP)
rasppoe.sys, v0.0
884860e8 NetLuidIndex 0, IfIndex 4, WAN Miniport (PPPOE)
ndiswan.sys, v0.0
8842f0e8 NetLuidIndex 0, IfIndex 5, WAN Miniport (IPv6)
8842e0e8 NetLuidIndex 3, IfIndex 6, WAN Miniport (IP)
rasl2tp.sys, v0.0
8842b0e8 NetLuidIndex 0, IfIndex 2, WAN Miniport (L2TP)
E1G60I32.sys, v8.1
84b730e8 NetLuidIndex 4, IfIndex 8, Intel(R) PRO/1000 MT Network Connection
tunnel.sys, v1.0
84b370e8 NetLuidIndex 2, IfIndex 9, isatap.{0DC6D9AD-70DC-41CE-9798-F71D1A8C899F}

1: kd> !ndiskd.miniport 84b730e8

MINIPORT

Intel(R) PRO/1000 MT Network Connection

Ndis Handle      84b730e8
Ndis API Version v6.0
Adapter Context  88460008
Miniport Driver   84b44938 - E1G60I32.sys  v8.1
Ndis Verifier     [No flags set]

Media Type        802.3
Physical Medium   802.3
Device Path       \??\PCI#VEN_8086&DEV_100F&SUBSYS_075015AD&REV_01#4&b70f118&0&0888#\{ad498944-762f-
11d0-8dc8-00c04fc3358c}\{0DC6D9AD-70DC-41CE-9798-F71D1A8C899F}
Device Object     84b73030
MAC Address       00-0c-29-b1-7d-39

STATE

Miniport          Running
Device PnP         Started
Datapath        00000002      ← DIVERTED_BECAUSE_MEDIA_DISCONNECTED
NBL Status       NDIS_STATUS_MEDIA_DISCONNECTED
Operational status DOWN
Operational flags 00000002      ← DOWN_NOT_CONNECTED
Admin status       ADMIN_UP
Media             MediaDisconnected
Power             D0
References        6
```

```
User Handles      0
Total Resets     0
Pending OID      None
Flags            0c452218
    ↑ BUS_MASTER, 64BIT_DMA, SG_DMA, DEFAULT_PORT_ACTIVATED,
    SUPPORTS_MEDIA_SENSE, DOES_NOT_DO_LOOPBACK, NOT_MEDIA_CONNECTED
PnPFlags         00210021
    ↑ PM_SUPPORTED, DEVICE_POWER_ENABLED, RECEIVED_START, HARDWARE_DEVICE
```

**BINDINGS**

Filter List	Filter	Filter Driver	Context	-
QoS Packet Scheduler-0000				
	88e453d8	88e18938	88e1ed60	
Open List	Open	Protocol	Context	-
RSPNDR	8bcbb470	8bd23ac8	8bcbb820	
LLTDIO	8bcb8c00	8bd15980	8bd153f8	
TCPIP6	88e528e8	88e02350	88e52c98	
TCPIP	88e1c078	88e02aa8	88e1e6a8	

**MORE INFORMATION**

- Driver handlers
- Task offloads
- Power management
- Pending OIDs
- Timers
- Receive Side Throttling
- Wake-on-LAN (WoL)
- Packet filter
- NDIS ports

Another example from a different complete memory dump:

**STATE**

```
Device PnP        Started
Datapath          00000002      ← DIVERTED_BECAUSE_MEDIA_DISCONNECTED
Packet Status     NDIS_STATUS_NO_CABLE
Media             Not Connected
[...]
```

## Disk Packet Buildup

This is similar to **Network Packet Buildup** (page 733) pattern. It can be detectable either through SCSI WinDbg extension or using **IRP Object Distribution Anomaly** (page 756) pattern:

```
0: kd> .load scsikd

0: kd> !scsikd.classext
Storage class devices:

* !classext ffffffa80026395b0 [1,2] SAMSUNG HS082HB Paging Disk

Usage: !classext <class device> <level [0-2]>

0: kd> !scsikd.classext ffffffa80026395b0
Storage class device ffffffa80026395b0 with extension at ffffffa8002639700

Classpnp Internal Information at ffffffa8002648010

-- dt classpnp!_CLASS_PRIVATE_FDO_DATA ffffffa8002648010 --

Classpnp External Information at ffffffa8002639700

SAMSUNG HS082HB NL100-01 S140JR0SA00025

Minidriver information at ffffffa8002639bc0
Attached device object at ffffffa80017ecda0
Physical device object at ffffffa80024ab060

Media Geometry:

Bytes in a Sector = 512
Sectors per Track = 63
Tracks / Cylinder = 255
Media Length      = 80026361856 bytes = ~74 GB

-- dt classpnp!_FUNCTIONAL_DEVICE_EXTENSION ffffffa8002639700 --
```

This is a normal case:

```
0: kd> !scsikd.classext ffffffa80026395b0 2
Storage class device ffffffa80026395b0 with extension at ffffffa8002639700

Classpnp Internal Information at ffffffa8002648010

Transfer Packet Engine:

Packet      Status   DL Irp          Opcode  Sector/ListId   UL Irp
-----  -----  -----  -----  -----  -----  -----
```

```

fffffa8002648e80 Free ffffffa800249eac0
fffffa8002644220 Free ffffffa80024aa4f0
fffffa80026898a0 Free ffffffa80019d2b30
fffffa800267ad40 Free ffffffa8001801b90
fffffa800267aa60 Free ffffffa8001835e10
fffffa8002679010 Free ffffffa80019fac40
fffffa8002679770 Free ffffffa8002679500
fffffa80027659a0 Free ffffffa8002764e10
fffffa8002790e80 Free ffffffa800267a6a0
fffffa800278f5e0 Free ffffffa80019d53c0
fffffa8002599410 Free ffffffa8002785600
fffffa80027f7490 Free ffffffa800278ea00
fffffa80027f6e80 Free ffffffa80027f6c80
fffffa80027f69a0 Free ffffffa80027f67a0
fffffa80027f64c0 Free ffffffa80027f62c0
fffffa8002dd4440 Free ffffffa80027fc600
fffffa8002dc30 Free ffffffa8002dceb30
fffffa8002dce850 Free ffffffa8002ddc010
fffffa8002ddc530 Free ffffffa8002ddc330
fffffa8002de2d30 Free ffffffa8002de2b30
fffffa8002de2850 Free ffffffa8002de2650
fffffa8002de2370 Free ffffffa8002de2170
fffffa8002ddbe80 Free ffffffa8002ddbc80
fffffa8002ddb9a0 Free ffffffa8002ddb7a0
fffffa8002dda010 Free ffffffa8002ddae10
[...]

```

This is not:

```

0: kd> !scsikd.classext ffffffa80026395b0 2

Storage class device ffffffa80026395b0 with extension at ffffffa8002639700

Classpnp Internal Information at ffffffa8002648010

```

Transfer Packet Engine:

Packet	Status	DL Irp	Opcode	Sector	UL Irp
fffffa80c71d9560	Queued	fffffa80c71d9360	2a	03ccb948	fffffa80c4f269d0 \FileName
fffffa80c77a3360	Queued	fffffa80c77a3160	2a	0400f0a8	fffffa80c59c1010 \FileName
fffffa80c6cefef0	Queued	fffffa80c6cefef0	2a	0400f128	fffffa80c59c1010 \FileName
fffffa80c6e92260	Queued	fffffa80c4f80010	2a	0400f1e8	fffffa80c59c1010 \FileName
fffffa80c79dbca0	Queued	fffffa80c79dbaa0	2a	0400c4e8	fffffa80c59c1010 \FileName
fffffa80c83f2d90	Queued	fffffa80c3b23bc0	2a	0400f168	fffffa80c59c1010 \FileName
fffffa80c4a94640	Queued	fffffa80c4a94440	2a	0400d5e8	fffffa80c59c1010 \FileName
fffffa80c7984010	Queued	fffffa80c7984210	2a	0400d328	fffffa80c59c1010 \FileName
fffffa80c6e52be0	Queued	fffffa80c6e529e0	2a	0400f1a8	fffffa80c59c1010 \FileName
fffffa80c7afada0	Queued	fffffa80c7afaba0	2a	04010268	fffffa80c59c1010 \FileName
fffffa80c7c19d90	Queued	fffffa80ca2c5e10	2a	0400c628	fffffa80c59c1010 \FileName
fffffa80c6182d60	Queued	fffffa80c6182b60	2a	0400f9a8	fffffa80c59c1010 \FileName
fffffa80c8695ba0	Queued	fffffa80c86959a0	2a	0400d128	fffffa80c59c1010 \FileName
fffffa80c6b42b40	Queued	fffffa80c6b42940	2a	0400ed28	fffffa80c59c1010 \FileName
fffffa80c5e1ab00	Queued	fffffa80c5e1a900	2a	0400eee8	fffffa80c59c1010 \FileName

fffffa80c5d80a30	Queued	fffffa80c4841e10	2a	0400fba8	fffffa80c59c1010 \FileName
fffffa80c48255d0	Queued	fffffa80c48253d0	2a	040119e8	fffffa80c59c1010 \FileName
fffffa80c718a270	Queued	fffffa80c47d0010	2a	0400d1e8	fffffa80c59c1010 \FileName
fffffa80c51a94b0	Queued	fffffa80c51a92b0	2a	0400bd28	fffffa80c59c1010 \FileName
fffffa80ca280990	Queued	fffffa80c52b2930	2a	0400d268	fffffa80c59c1010 \FileName
fffffa80c586f280	Queued	fffffa80c551fe10	2a	0400f068	fffffa80c59c1010 \FileName
fffffa80c8413540	Queued	fffffa80c544ae10	2a	04011a68	fffffa80c59c1010 \FileName
fffffa80c544ac60	Queued	fffffa80c535ba90	2a	0400e7e8	fffffa80c59c1010 \FileName
fffffa80c4678010	Queued	fffffa80c4678230	2a	04011168	fffffa80c59c1010 \FileName
fffffa80c9d94be0	Queued	fffffa80c59205e0	2a	0400d4a8	fffffa80c59c1010 \FileName
fffffa80c5920430	Queued	fffffa80c59248f0	2a	0400ea68	fffffa80c59c1010 \FileName
fffffa80c737e8f0	Queued	fffffa80c737e6f0	2a	0400fee8	fffffa80c59c1010 \FileName
fffffa80c4797c60	Queued	fffffa80c5d31800	2a	0400f328	fffffa80c59c1010 \FileName
fffffa80c711d270	Queued	fffffa80c76ee390	2a	0400eaa8	fffffa80c59c1010 \FileName
fffffa80c872dba0	Queued	fffffa80c872d9a0	2a	0400eb28	fffffa80c59c1010 \FileName
fffffa80c9e67d10	Queued	fffffa80c9e67b10	2a	04012168	fffffa80c59c1010 \FileName
fffffa80ca3bb350	Queued	fffffa80c66e4370	2a	0400c928	fffffa80c59c1010 \FileName
fffffa80c5894ab0	Queued	fffffa80c58948b0	2a	0400c368	fffffa80c59c1010 \FileName
fffffa80c305fe60	Queued	fffffa80c305fc60	2a	04013168	fffffa80c59c1010 \FileName
fffffa80c496cce0	Queued	fffffa80c496cae0	2a	0400d168	fffffa80c59c1010 \FileName
fffffa80c5e78c60	Queued	fffffa80c905c7f0	2a	0400f8a8	fffffa80c59c1010 \FileName
fffffa80c905c640	Queued	fffffa80c5c1c410	2a	0400f428	fffffa80c59c1010 \FileName
fffffa80c68ffc40	Queued	fffffa80c68ffa40	2a	0400f468	fffffa80c59c1010 \FileName
fffffa80c3aa3e60	Queued	fffffa80c3aa3c60	2a	0400c7a8	fffffa80c59c1010 \FileName
fffffa80c5e8dc60	Queued	fffffa80c8852cf0	2a	0400f4a8	fffffa80c59c1010 \FileName
fffffa80c90082b0	Queued	fffffa80c7907440	2a	04013428	fffffa80c59c1010 \FileName
fffffa80c7907290	Queued	fffffa80c67aea80	2a	0400fe68	fffffa80c59c1010 \FileName
fffffa80c67ae8d0	Queued	fffffa80c9383cf0	2a	0400f3a8	fffffa80c59c1010 \FileName
fffffa80c8497010	Queued	fffffa80c8497270	2a	0400c5e8	fffffa80c59c1010 \FileName
fffffa80c78c7480	Queued	fffffa80c78c7280	2a	0400c3e8	fffffa80c59c1010 \FileName
fffffa80c7f37d90	Queued	fffffa80c618b480	2a	0400cce8	fffffa80c59c1010 \FileName
fffffa80c618b2d0	Queued	fffffa80ca2e9e10	2a	0400ee28	fffffa80c59c1010 \FileName
fffffa80ca2e9c60	Queued	fffffa80c5e783f0	2a	0400d8e8	fffffa80c59c1010 \FileName
fffffa80c64e1650	Queued	fffffa80c64e1450	2a	0400d0e8	fffffa80c59c1010 \FileName
fffffa80c684dd60	Queued	fffffa80c684db60	2a	0400c6a8	fffffa80c59c1010 \FileName
fffffa80c3b2bac0	Queued	fffffa80c3b2b8c0	2a	040127a8	fffffa80c59c1010 \FileName
fffffa80c5ff64d0	Queued	fffffa80c5ff62d0	2a	0400de68	fffffa80c59c1010 \FileName
fffffa80c99a84b0	Queued	fffffa80c99a82b0	2a	0400fce8	fffffa80c59c1010 \FileName
fffffa80ca300510	Queued	fffffa80ca300310	2a	0400c168	fffffa80c59c1010 \FileName
[...]					

## Comments

For Windows 8 and higher there is **!storagekd.storclass 2** extension command.

## Dispatch Level Spin

**Spiking Thread** pattern (page 900) includes normal threads running at PASSIVE\_LEVEL or APC\_LEVEL IRQL that can be preempted by any other higher priority thread. Therefore, **Spiking Threads** are not necessarily ones that were in a RUNNING state when the memory dump was saved. They consumed much CPU, and this is reflected in their *User* and *Kernel* time values. The pattern also includes threads running at DISPATCH\_LEVEL and higher IRQL. These threads cannot be preempted by another thread, so they usually remain in the RUNNING state all the time unless they lower their IRQL. Some of them can be trying to acquire a spinlock, and we need a more specialized pattern for them. We would see it when a spinlock for some data structure wasn't released or was corrupt, and some thread tries to acquire it and enters endless spinning loop unless interrupted by a higher IRQL interrupt. These infinite loops can also happen due to software defects in code running at dispatch level or higher IRQL.

Let's look at one example. The following running thread was interrupted by a keyboard interrupt apparently to save **Manual Dump** (page 625). We see that it spent almost 11 minutes in the kernel:

```
0: kd> !thread
THREAD 830c07c0 Cid 0588.0528 Peb: 7ffa3000 Win32Thread: e29546a8 RUNNING on processor 0
Not impersonating
DeviceMap          e257b7c8
Owning Process    831ec608      Image:      MyApp.EXE
Wait Start TickCount 122850      Ticks: 40796 (0:00:10:37.437)
Context Switch Count 191           LargeStack
UserTime           00:00:00.000
KernelTime         00:10:37.406
Win32 Start Address MyApp!ThreadImpersonation (0x35f76821)
Start Address kernel32!BaseThreadStartThunk (0x7c810659)
Stack Init a07bf000 Current a07beca0 Base a07bf000 Limit a07bb000 Call 0
Priority 11 BasePriority 8 PriorityDecrement 2 DecrementCount 16
ChildEBP RetAddr
a07be0f8 f77777fa nt!KeBugCheckEx+0x1b
a07be114 f7777032 i8042prt!I8xProcessCrashDump+0x237
a07be15c 805448e5 i8042prt!I8042KeyboardInterruptService+0x21c
a07be15c 806e4a37 nt!KiInterruptDispatch+0x45 (FPO: [0,2] TrapFrame @ a07be180)
a07be220 a1342755 hal!KeAcquireInStackQueuedSpinLock+0x47
a07be220 a1342755 MyDriver!RcvData+0x98
```

To see the code and context we switch to the trap frame<sup>44</sup> and disassemble the interrupted function:

```
1: kd> .trap a07be180
ErrCode = 00000000
eax=a07be200 ebx=a07be228 ecx=831dabf5 edx=a07beb94 esi=831d02a8 edi=831dabd8
eip=806e4a37 esp=a07be1f4 ebp=a07be220 iopl=0 nv up ei pl nz na po nc
cs=0008 ss=0010 ds=0000 es=0000 fs=0000 gs=0000 ef1=00000202
hal!KeAcquireInStackQueuedSpinLock+0x47:
806e4a37 ebf3      jmp     hal!KeAcquireInStackQueuedSpinLock+0x3c (806e4a2c)
```

<sup>44</sup> Interrupt Frames and Stack Reconstruction, Memory Dump Analysis Anthology, Volume 1, page 83

```

1: kd> uf hal!KeAcquireInStackQueuedSpinLock
hal!KeAcquireInStackQueuedSpinLock:
806e49f0 mov     eax,dword ptr ds:[FFFE0080h]
806e49f5 shr     eax,4
806e49f8 mov     al,byte ptr hal!HalpVectorToIRQL (806ef218)[eax]
806e49fe mov     dword ptr ds:[0FFFFE0080h],41h
806e4a08 mov     byte ptr [edx+8],al
806e4a0b mov     dword ptr [edx+4],ecx
806e4a0e mov     dword ptr [edx],0
806e4a14 mov     eax,edx
806e4a16 xchg    edx,dword ptr [ecx]
806e4a18 cmp     edx,0
806e4a1b jne     hal!KeAcquireInStackQueuedSpinLock+0x34 (806e4a24)

[...]

hal!KeAcquireInStackQueuedSpinLock+0x34:
806e4a24 or      ecx,1
806e4a27 mov     dword ptr [eax+4],ecx
806e4a2a mov     dword ptr [edx],eax

hal!KeAcquireInStackQueuedSpinLock+0x3c:
806e4a2c test    dword ptr [eax+4],1
806e4a33 je     hal!KeAcquireInStackQueuedSpinLock+0x33 (806e4a23)

hal!KeAcquireInStackQueuedSpinLock+0x45:
806e4a35 pause
806e4a37 jmp    hal!KeAcquireInStackQueuedSpinLock+0x3c (806e4a2c)

```

JMP instruction transfers execution to the code that tests the first bit at [EAX+4] address. If it isn't set, it falls through to the same JMP instruction. We know the value of EAX from the trap frame so we can dereference that address:

```

1: kd> dyd eax+4 11
      3          2          1          0
      10987654 32109876 54321098 76543210
      -----
a07be204 10000011 00011101 10101011 111101011 831dabf5

```

The value is odd: the first leftmost bit is set. Therefore, the code loops indefinitely unless a different thread running on another processor clears that bit. However the second processor is idle:

```

0: kd> ~0s
0: kd> k
ChildEBP RetAddr
f794cd54 00000000 nt!KiIdleLoop+0x14

```

Seems we have a problem. We need to examine *MyDriver.sys* code to understand how it uses queued spinlocks.

**Note:** In addition to user-defined there are internal system queued spinlocks we can check by using **!qlocks** WinDbg command.

## Distributed Exception

### Managed Code

Managed code **Nested Exceptions** (page 723) give us process virtual space bound stack traces. However, exception objects may be marshaled across processes and even computers. The remote stack trace return addresses don't have the same validity in different process contexts. Fortunately, there is a `_remoteStackTraceString` field in exception objects, and it contains the original stack trace. Default analysis command sometimes uses it:

```
0:013> !analyze -v

[...]

EXCEPTION_OBJECT: !pe 25203b0
Exception object: 0000000025203b0
Exception type: System.Reflection.TargetInvocationException
Message: Exception has been thrown by the target of an invocation.
InnerException: System.Management.Instrumentation.WmiProviderInstallationException, Use !PrintException
000000002522cf0 to see more.
StackTrace (generated):
SP IP Function
000000001D39E720 0000000000000001 Component!Proxy.Start()+0x20
000000001D39E720 000007FEF503D0B6
mscorlib_ni!System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object, Boolean)+0x286
000000001D39E880 000007FEF503CE1A
mscorlib_ni!System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object, Boolean)+0xa
000000001D39E8B0 000007FEF503CDD8
mscorlib_ni!System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)+0x58
000000001D39E900 000007FEF4FB0302 mscorlib_ni!System.Threading.ThreadHelper.ThreadStart()+0x52

[...]

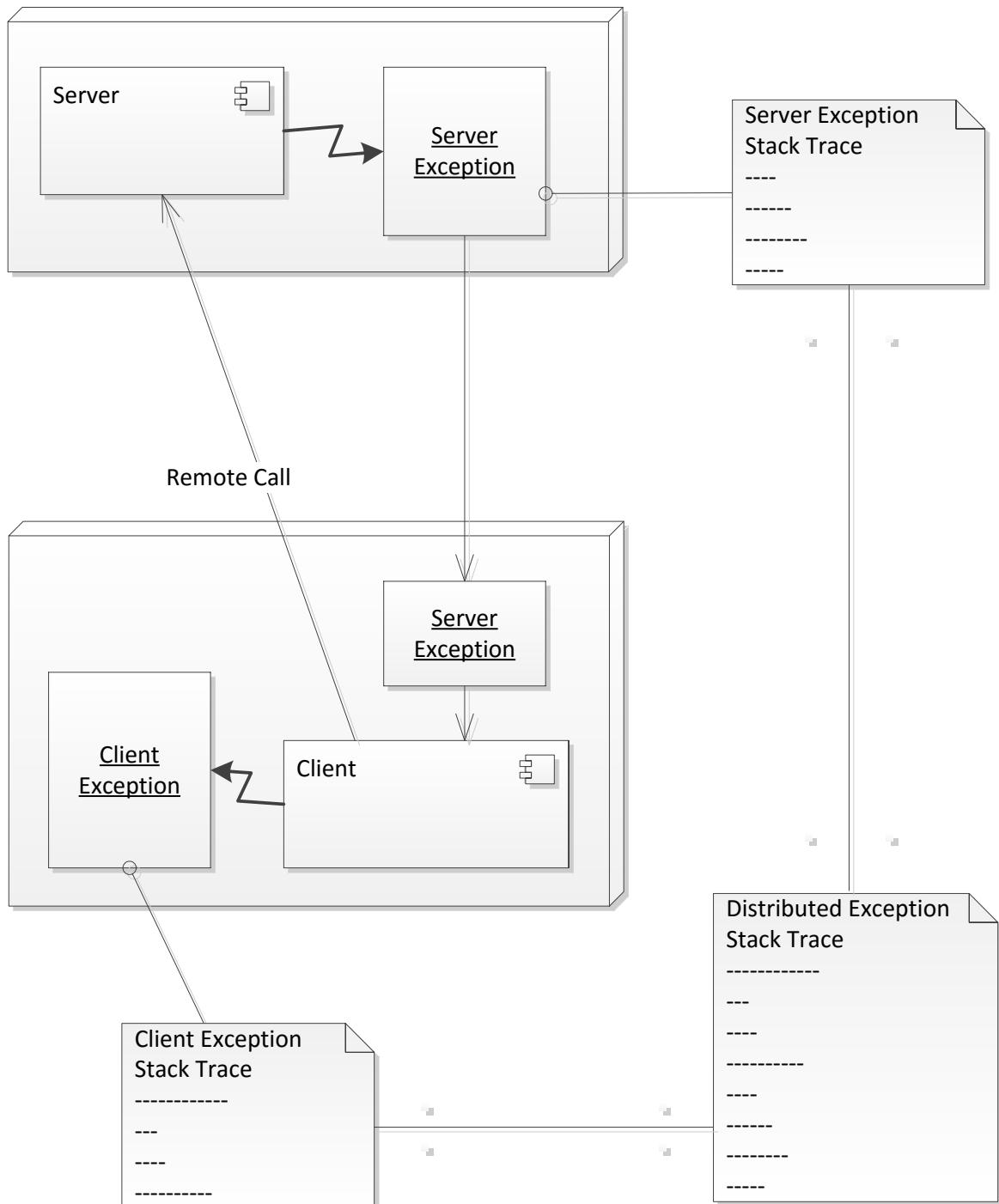
MANAGED_STACK_COMMAND: ** Check field _remoteStackTraceString **;!do 2522cf0;!do 2521900

[...]

0:013> !DumpObj 2522cf0
[...]
000007fef51b77f0 4000054 2c System.String 0 instance 2521900 _remoteStackTraceString
[...]

0:013> !DumpObj 2521900
Name: System.String
[...]
String: at System.Management.Instrumentation.InstrumentationManager.RegisterType(Type managementType)
at Component.Provider..ctor()
at Component.Start()
```

Checking this field may also be necessary for exceptions of interest from managed space **Execution Residue** (page 369). We call this pattern **Distributed Exception**. The basic idea is illustrated in the following diagram using the borrowed UML notation (not limited to just two computers):



## Distributed Spike

Abnormal CPU consumption detection usually goes at a process level when we detect it using Task Manager, for example. Sometimes that process has only one **Spiking Thread** (page 888) among many, but there are cases when CPU consumption is spread among many threads. We call this pattern **Distributed Spike**. Such behavior could be a consequence of weakly **Coupled Processes** (page 151), for example, in these two services (where, for simplicity, we highlight in bold threads with more than 1 second CPU time spent in user mode):

```
0:000> !runaway
User Mode Time
 Thread      Time
120:4e518 0 days 0:05:09.937
126:531bc 0 days 0:03:56.546
44:334c    0 days 0:03:40.765
133:4fe1c 0 days 0:03:31.156
45:42b4    0 days 0:03:27.328
107:25ae0 0 days 0:03:19.921
49:627c    0 days 0:02:48.250
147:6b90c 0 days 0:02:33.046
136:6620c 0 days 0:02:05.109
127:4f2d0 0 days 0:02:04.046
129:5bc30 0 days 0:02:02.171
48:623c    0 days 0:02:01.796
119:41f00 0 days 0:02:00.562
74:cd18   0 days 0:01:59.453
51:7a4c    0 days 0:01:54.234
35:21d4   0 days 0:01:47.390
148:326dc 0 days 0:01:32.640
123:43c8c 0 days 0:01:32.515
135:67b08 0 days 0:01:32.296
11:aa8     0 days 0:01:30.906
118:42f8c 0 days 0:01:20.265
42:3a3c   0 days 0:01:20.000
77:d024   0 days 0:01:19.734
115:3a840 0 days 0:01:15.625
89:145f4 0 days 0:01:10.500
157:4e310 0 days 0:01:07.625
80:d07c   0 days 0:01:07.468
33:1ab0   0 days 0:01:00.593
117:10bd4 0 days 0:00:59.421
151:1aaa0 0 days 0:00:59.015
28:17bc   0 days 0:00:58.796
83:f3a4   0 days 0:00:55.828
122:41964 0 days 0:00:55.578
149:4101c 0 days 0:00:55.234
10:aa4    0 days 0:00:52.453
106:21b80 0 days 0:00:51.187
132:62e5c 0 days 0:00:49.437
160:3a3a8 0 days 0:00:48.875
137:6bf90 0 days 0:00:48.687
145:6f594 0 days 0:00:47.968
143:58d60 0 days 0:00:45.703
72:ba64   0 days 0:00:44.515
```

41:19b0	0 days 0:00:44.000
130:5d480	0 days 0:00:43.750
139:6d090	0 days 0:00:42.062
138:6d578	0 days 0:00:40.406
91:17974	0 days 0:00:40.359
152:37f80	0 days 0:00:39.781
81:de68	0 days 0:00:39.265
150:65b2c	0 days 0:00:36.625
162:1f340	0 days 0:00:35.125
85:10650	0 days 0:00:33.546
131:614e8	0 days 0:00:33.093
128:2eddc	0 days 0:00:33.000
146:6f690	0 days 0:00:32.015
161:3c4b4	0 days 0:00:30.421
167:3cde4	0 days 0:00:29.390
171:3979c	0 days 0:00:28.515
166:3cd40	0 days 0:00:28.312
168:68ef0	0 days 0:00:27.781
65:aad0	0 days 0:00:26.593
109:267f4	0 days 0:00:26.390
88:13624	0 days 0:00:26.000
173:5282c	0 days 0:00:24.640
153:71e14	0 days 0:00:23.390
112:322b4	0 days 0:00:22.812
110:9578	0 days 0:00:22.125
175:20230	0 days 0:00:20.250
79:b458	0 days 0:00:20.218
66:61b8	0 days 0:00:19.875
62:9498	0 days 0:00:19.562
156:d900	0 days 0:00:19.015
121:5106c	0 days 0:00:18.687
142:6bb28	0 days 0:00:18.562
46:2cbc	0 days 0:00:17.796
169:d920	0 days 0:00:16.875
154:720b4	0 days 0:00:16.484
170:4ac8c	0 days 0:00:15.968
73:b010	0 days 0:00:13.609
39:3224	0 days 0:00:13.406
172:722e4	0 days 0:00:12.375
63:9780	0 days 0:00:12.203
177:8464	0 days 0:00:11.906
184:22908	0 days 0:00:10.234
140:5765c	0 days 0:00:09.750
174:2f484	0 days 0:00:08.390
50:7230	0 days 0:00:07.125
187:3c324	0 days 0:00:06.765
125:46cf0	0 days 0:00:06.296
178:3a424	0 days 0:00:05.125
114:33d20	0 days 0:00:03.734
165:3ca74	0 days 0:00:01.203
189:3c358	0 days 0:00:01.000
164:3124c	0 days 0:00:00.578
25:be4	0 days 0:00:00.515
17:ba8	0 days 0:00:00.125
104:5cf8	0 days 0:00:00.109
26:e4c	0 days 0:00:00.109

96:5d44	0 days 0:00:00.093
99:5b18	0 days 0:00:00.078
56:8a6c	0 days 0:00:00.078
55:8a68	0 days 0:00:00.078
6:a08	0 days 0:00:00.078
4:a00	0 days 0:00:00.062
103:5cf0	0 days 0:00:00.046
100:5ab8	0 days 0:00:00.046
68:bf34	0 days 0:00:00.046
37:29d4	0 days 0:00:00.046
101:5ab4	0 days 0:00:00.031
98:5b44	0 days 0:00:00.031
97:5d40	0 days 0:00:00.031
57:8a70	0 days 0:00:00.031
53:8a60	0 days 0:00:00.031
36:29c0	0 days 0:00:00.031
16:ac4	0 days 0:00:00.031
1:9e4	0 days 0:00:00.031
60:880c	0 days 0:00:00.015
58:8a5c	0 days 0:00:00.015
24:be0	0 days 0:00:00.015
15:abc	0 days 0:00:00.015
188:13044	0 days 0:00:00.000
186:6530	0 days 0:00:00.000
185:2013c	0 days 0:00:00.000
183:6047c	0 days 0:00:00.000
182:65400	0 days 0:00:00.000
181:61560	0 days 0:00:00.000
180:2b7a4	0 days 0:00:00.000
179:56294	0 days 0:00:00.000
176:20300	0 days 0:00:00.000
163:2ab1c	0 days 0:00:00.000
159:276cc	0 days 0:00:00.000
158:72134	0 days 0:00:00.000
155:6a078	0 days 0:00:00.000
144:6ce98	0 days 0:00:00.000
141:5404	0 days 0:00:00.000
134:65718	0 days 0:00:00.000
124:4bed4	0 days 0:00:00.000
116:3c770	0 days 0:00:00.000
113:b08	0 days 0:00:00.000
111:28e54	0 days 0:00:00.000
108:25fbc	0 days 0:00:00.000
105:20504	0 days 0:00:00.000
102:5cf4	0 days 0:00:00.000
95:5c70	0 days 0:00:00.000
94:5ed4	0 days 0:00:00.000
93:18c2c	0 days 0:00:00.000
92:19fd8	0 days 0:00:00.000
90:c870	0 days 0:00:00.000
87:7994	0 days 0:00:00.000
86:124cc	0 days 0:00:00.000
84:eab8	0 days 0:00:00.000
82:f2a4	0 days 0:00:00.000
78:d5c0	0 days 0:00:00.000
76:cf0	0 days 0:00:00.000

75:cf64	0 days 0:00:00.000
71:b4f8	0 days 0:00:00.000
70:c628	0 days 0:00:00.000
69:c484	0 days 0:00:00.000
67:be84	0 days 0:00:00.000
64:aa00	0 days 0:00:00.000
61:93f0	0 days 0:00:00.000
59:89e4	0 days 0:00:00.000
54:8a64	0 days 0:00:00.000
52:89a8	0 days 0:00:00.000
47:4c64	0 days 0:00:00.000
43:3fa0	0 days 0:00:00.000
40:2c88	0 days 0:00:00.000
38:2a28	0 days 0:00:00.000
34:1928	0 days 0:00:00.000
32:1668	0 days 0:00:00.000
31:8dc	0 days 0:00:00.000
30:15d4	0 days 0:00:00.000
29:1044	0 days 0:00:00.000
27:fb4	0 days 0:00:00.000
23:bd8	0 days 0:00:00.000
22:bd4	0 days 0:00:00.000
21:bd0	0 days 0:00:00.000
20:bc8	0 days 0:00:00.000
19:bc4	0 days 0:00:00.000
18:bc0	0 days 0:00:00.000
14:ab8	0 days 0:00:00.000
13:ab4	0 days 0:00:00.000
12:ab0	0 days 0:00:00.000
9:aa0	0 days 0:00:00.000
8:a9c	0 days 0:00:00.000
7:a98	0 days 0:00:00.000
5:a04	0 days 0:00:00.000
3:9f4	0 days 0:00:00.000
2:9f0	0 days 0:00:00.000
0:994	0 days 0:00:00.000

This is a real spike in the first service process as can be confirmed by a random non-waiting thread:

```
0:000> ~143k
ChildEBP RetAddr
050dfc68 7c82d6a4 nt!RtlEnterCriticalSection+0x1d
050dfc84 77c7bc50 nt!RtlInitializeCriticalSectionAndSpinCount+0x92
050dfc98 77c7bc7c rpcrt4!_MUTEX::CommonConstructor+0x1b
050dfcac 77c7c000 rpcrt4!_MUTEX::_MUTEX+0x13
050dfcc8 77c6ff47 rpcrt4!BINDING_HANDLE::BINDING_HANDLE+0x2d
050dfcd8 77c6ff1f rpcrt4!SVR_BINDING_HANDLE::SVR_BINDING_HANDLE+0x10
050dfcfc 77c6d338 rpcrt4!RPC_ADDRESS::InquireBinding+0x8a
050dfd0c 77c6fd1d rpcrt4!LRPC_SCALL::ToStringBinding+0x16
050dfd1c 76554c83 rpcrt4!RpcBindingToStringBindingW+0x4d
050dfd5c 77c7c42a ServiceA!RpcSecurityCallback+0x1e
050fdb4 77c7c4b0 rpcrt4!RPC_INTERFACE::CheckSecurityIfNecessary+0x6f
050fdcc 77c7c46c rpcrt4!LRPC_SBINDING::CheckSecurity+0x4f
050fdfc 77c812f0 rpcrt4!LRPC_SCALL::DealWithRequestMessage+0x2bb
050dfe20 77c88678 rpcrt4!RPC_ADDRESS::DealWithLRPCRequest+0x127
050dff84 77c88792 rpcrt4!RPC_ADDRESS::ReceiveLotsaCalls+0x430
```

```

050dff8c 77c8872d rpcrt4!RecvLotsaCallsWrapper+0xd
050dffac 77c7b110 rpcrt4!BaseCachedThreadRoutine+0x9d
050dffb8 77e64829 rpcrt4!ThreadStartRoutine+0x1b
050dffec 00000000 kernel32!BaseThreadStart+0x34

0:000> ~143r
eax=00000000 ebx=00000000 ecx=7c887784 edx=7c887780 esi=7c887784 edi=00163fb0
eip=7c81a37d esp=050dfc5c ebp=050dfc68 iopl=0 nv up ei ng nz na pe cy
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000287
ntdll!RtlEnterCriticalSection+0x1d:
7c81a37d 0f92c0    setb    al

0:000> .asm no_code_bytes
Assembly options: no_code_bytes

0:000> u 7c81a37d
ntdll!RtlEnterCriticalSection+0x1d:
7c81a37d setb    al
7c81a380 test   al,al
7c81a382 je     ntdll!RtlEnterCriticalSection+0x28 (7c82b096)
7c81a388 mov    ecx,dword ptr fs:[18h]
7c81a38f mov    eax,dword ptr [ecx+24h]
7c81a392 pop    edi
7c81a393 mov    dword ptr [edx+0Ch],eax
7c81a396 mov    dword ptr [edx+8],1

0:000> ub 7c81a37d
ntdll!RtlEnterCriticalSection+0x6:
7c81a366 mov    edx,dword ptr [ebp+8]
7c81a369 push   esi
7c81a36a lea    esi,[edx+4]
7c81a36d push   edi
7c81a36e mov    dword ptr [ebp-4],esi
7c81a371 mov    eax,0
7c81a376 mov    ecx,dword ptr [ebp-4]
7c81a379 lock btr dword ptr [ecx],eax

```

The second service is weakly (waiting for event notifications) coupled to the first service above:

User Mode Time	Time
5:dbec	0 days 0:01:50.031
8:46008	0 days 0:01:46.062
11:ad0c	0 days 0:01:13.921
17:932c	0 days 0:01:03.234
14:45d78	0 days 0:00:58.109
15:6d4d0	0 days 0:00:00.015
2:725a4	0 days 0:00:00.015
0:6101c	0 days 0:00:00.015
18:d1c4	0 days 0:00:00.000
16:76bc	0 days 0:00:00.000
13:456a8	0 days 0:00:00.000
12:459e4	0 days 0:00:00.000

```

10:3c768      0 days 0:00:00.000
9:12d20      0 days 0:00:00.000
7:46010      0 days 0:00:00.000
6:4600c      0 days 0:00:00.000
4:dbf0       0 days 0:00:00.000
3:17ed4      0 days 0:00:00.000
1:61024      0 days 0:00:00.000

0:000> ~11k
ChildEBP RetAddr
0223fa68 7c82787b ntdll!KiFastSystemCallRet
0223fa6c 77c80a6e ntdll!NtRequestWaitReplyPort+0xc
0223fab8 77c7fcf0 rpct4!LRPC_CCALL::SendReceive+0x230
0223fac4 77c80673 rpct4!I_RpcSendReceive+0x24
0223fad8 77ce315a rpct4!NdrSendReceive+0x2b
0223fec0 771f4fb0 rpct4!NdrClientCall12+0x22e
0223fed8 771f4f60 ServiceB!RpcWaitEvent+0x1c
[...]

0:000> ~17k
ChildEBP RetAddr
0283fa68 7c82787b ntdll!KiFastSystemCallRet
0283fa6c 77c80a6e ntdll!NtRequestWaitReplyPort+0xc
0283fab8 77c7fcf0 rpct4!LRPC_CCALL::SendReceive+0x230
0283fac4 77c80673 rpct4!I_RpcSendReceive+0x24
0283fad8 77ce315a rpct4!NdrSendReceive+0x2b
0283fec0 771f4fb0 rpct4!NdrClientCall12+0x22e
0283fed8 771f4f60 ServiceB!RpcWaitEvent+0x1c
[...]

```

Sometimes, semantically **Coupled Processes** (page 148) result in distributed spikes, and most often it is possible to predict another spiking process in such cases. In our example above, both spiking processes were semantically coupled with another service, and it was confirmed that it was spiking too:

```

0:000> !runaway
User Mode Time
 Thread      Time
 89:10d4      0 days 0:03:03.500
 28:a94       0 days 0:00:39.562
 73:c10       0 days 0:00:37.531
 54:b88       0 days 0:00:37.140
 29:a98       0 days 0:00:35.906
 27:a90       0 days 0:00:35.500
 75:c2c       0 days 0:00:28.812
 90:10d8      0 days 0:00:27.000
 93:10e4      0 days 0:00:24.265
 32:aa4       0 days 0:00:12.906
 41:ac8       0 days 0:00:11.890
 35:ab0       0 days 0:00:11.875
 58:bc4       0 days 0:00:10.218
 42:acc       0 days 0:00:09.546
 85:e74       0 days 0:00:08.859
 36:ab4       0 days 0:00:08.578
 72:c0c       0 days 0:00:05.890

```

70:c04	0 days 0:00:05.687
33:aa8	0 days 0:00:05.046
74:c14	0 days 0:00:04.953
40:ac4	0 days 0:00:04.953
38:abc	0 days 0:00:04.359
39:ac0	0 days 0:00:04.312
34:aac	0 days 0:00:04.140
64:bec	0 days 0:00:03.812
88:10d0	0 days 0:00:03.187
30:a9c	0 days 0:00:02.859
9:a10	0 days 0:00:01.968
37:ab8	0 days 0:00:01.953
92:10e0	0 days 0:00:01.718
83:d00	0 days 0:00:01.125
94:1150	0 days 0:00:01.031
77:c54	0 days 0:00:00.890
98:f2c0	0 days 0:00:00.265
97:eb1c	0 days 0:00:00.265
76:c50	0 days 0:00:00.265
21:a48	0 days 0:00:00.187
22:a4c	0 days 0:00:00.140
63:be8	0 days 0:00:00.093
23:a50	0 days 0:00:00.093
53:af8	0 days 0:00:00.078
24:a54	0 days 0:00:00.046
71:c08	0 days 0:00:00.031
65:bf0	0 days 0:00:00.031
87:e8c	0 days 0:00:00.015
57:bc0	0 days 0:00:00.015
104:6454c	0 days 0:00:00.000
103:63fb4	0 days 0:00:00.000
102:3c5ec	0 days 0:00:00.000
101:65178	0 days 0:00:00.000
100:5d0e4	0 days 0:00:00.000
99:5bae4	0 days 0:00:00.000
96:574	0 days 0:00:00.000
95:b84	0 days 0:00:00.000
91:10dc	0 days 0:00:00.000
86:e88	0 days 0:00:00.000
84:e70	0 days 0:00:00.000
82:c84	0 days 0:00:00.000
81:c68	0 days 0:00:00.000
80:c64	0 days 0:00:00.000
79:c60	0 days 0:00:00.000
78:c5c	0 days 0:00:00.000
69:c00	0 days 0:00:00.000
68:bfc	0 days 0:00:00.000
67:bf8	0 days 0:00:00.000
66:bf4	0 days 0:00:00.000
62:bd8	0 days 0:00:00.000
61:bd4	0 days 0:00:00.000
60:bd0	0 days 0:00:00.000
59:bcc	0 days 0:00:00.000
56:bbc	0 days 0:00:00.000
55:bb8	0 days 0:00:00.000
52:af4	0 days 0:00:00.000

51:af0	0 days 0:00:00.000
50:aec	0 days 0:00:00.000
49:ae8	0 days 0:00:00.000
48:ae4	0 days 0:00:00.000
47:ae0	0 days 0:00:00.000
46:adc	0 days 0:00:00.000
45:ad8	0 days 0:00:00.000
44:ad4	0 days 0:00:00.000
43:ad0	0 days 0:00:00.000
31:aa0	0 days 0:00:00.000
26:a8c	0 days 0:00:00.000
25:a64	0 days 0:00:00.000
20:a44	0 days 0:00:00.000
19:a40	0 days 0:00:00.000
18:a34	0 days 0:00:00.000
17:a30	0 days 0:00:00.000
16:a2c	0 days 0:00:00.000
15:a28	0 days 0:00:00.000
14:a24	0 days 0:00:00.000
13:a20	0 days 0:00:00.000
12:a1c	0 days 0:00:00.000
11:a18	0 days 0:00:00.000
10:a14	0 days 0:00:00.000
8:a0c	0 days 0:00:00.000
7:a08	0 days 0:00:00.000
6:a04	0 days 0:00:00.000
5:a00	0 days 0:00:00.000
4:9fc	0 days 0:00:00.000
3:9f8	0 days 0:00:00.000
2:9f4	0 days 0:00:00.000
1:9f0	0 days 0:00:00.000
0:9e4	0 days 0:00:00.000

---

## Comments

If timing information is not available, but the random process memory snapshots or snapshots by procdump tool reveal different spiking threads, then we may also have this pattern.

## Distributed Wait Chain

Most **Wait Chain** patterns (page 1082) are about single wait chains. However, it is often a case when there are many different **Wait Chains** in a memory dump especially in terminal services environments. There can be **ALPC** (page 1097) and **Critical Section Wait Chains** (page 1086) at the same time. They can be related or completely disjoint. **Distributed Wait Chain** pattern covers a special case of several **Wait Chains** having the same structure (and possibly pointing in one direction). One such example we put below. In **Stack Trace Collection** (page 943) from a complete memory dump from a hanging system we found several *explorer.exe* processes with **Critical Section Wait Chains** having the same structure and endpoint of **Top** (page 1012) and **Blocking** (page 96) *ModuleA*:

```

THREAD ffffffa80137cf060 Cid 4884.4f9c Teb: 000007fffffaa000 Win32Thread: fffff900c0fb98b0 WAIT:
(UserRequest) UserMode Non-Alertable
    ffffffa8013570dc0 SynchronizationEvent
Not impersonating
DeviceMap          fffff8a014e21d90
Owning Process    ffffffa80131a75d0      Image:      explorer.exe
Attached Process   N/A           Image:      N/A
Wait Start TickCount 274752       Ticks: 212448 (0:00:55:19.500)
Context Switch Count 9889        LargeStack
UserTime           00:00:00.093
KernelTime         00:00:00.171
Win32 Start Address SHLWAPI!WrapperThreadProc (0x000007fefdafc608)
Stack Init fffff88013c25db0 Current fffff88013c25900
Base fffff88013c26000 Limit fffff88013c1b000 Call 0
Priority 11 BasePriority 9 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
Kernel stack not resident.
Child-SP          RetAddr        Call Site
ffffff880`13c25940 ffffff800`01873652 nt!KiSwapContext+0x7a
ffffff880`13c25a80 ffffff800`01884a9f nt!KiCommitThreadWait+0x1d2
ffffff880`13c25b10 ffffff800`01b7768e nt!KeWaitForSingleObject+0x19f
ffffff880`13c25bb0 ffffff800`0187ced3 nt!NtWaitForSingleObject+0xde
ffffff880`13c25c20 00000000`76d8135a nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ fffff880`13c25c20)
00000000`0489e518 00000000`76d7e4e8 ntdll!ZwWaitForSingleObject+0xa
00000000`0489e520 00000000`76d7e3db ntdll!RtlpWaitOnCriticalSection+0xe8
00000000`0489e5d0 000007fe`fdf8ff50 ntdll!RtlEnterCriticalSection+0xd1
00000000`0489e600 000007fe`fdf8fdb3 SHELL32!CFSFolder::GetIconOf+0x24b
00000000`0489f3a0 000007fe`fdf903d3 SHELL32!SHGetIconIndexFromPIDL+0x3f
00000000`0489f3d0 00000000`ff900328 SHELL32!SHMapIDListToSystemImageListIndexAsync+0x73
00000000`0489f470 00000000`ff8fff4b Explorer!SFTBarHost::AddImageForItem+0x9c
00000000`0489f4d0 00000000`ff8fd2f1 Explorer!SFTBarHost::_InternalRepopulateList+0x4ad
00000000`0489f5d0 00000000`ff8fd0b4 Explorer!SFTBarHost::_RepopulateList+0x1f3
00000000`0489f600 00000000`ff8fcccd Explorer!SFTBarHost::_OnBackgroundEnumDone+0xc1
00000000`0489f630 00000000`ff8fc9e2 Explorer!SFTBarHost::_WndProc+0x451
00000000`0489f680 00000000`76669bd1 Explorer!SFTBarHost::_WndProc_ProgramsMFU+0x1b
00000000`0489f6b0 00000000`766698da USER32!UserCallWinProcCheckWow+0x1ad
00000000`0489f770 00000000`ff8f1177 USER32!DispatchMessageWorker+0x3b5
00000000`0489f7f0 00000000`ff9130e9 Explorer!CTray::_MessageLoop+0x446
00000000`0489f880 000007fe`fdafcc71e Explorer!CTray::MainThreadProc+0x8a
00000000`0489f8b0 00000000`76c2652d SHLWAPI!WrapperThreadProc+0x19b
00000000`0489f9b0 00000000`76d5c521 kernel32!BaseThreadInitThunk+0xd
00000000`0489f9e0 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

```

```

0: kd> .process /r /p ffffffa80131a75d0
Implicit process is now ffffffa80`131a75d0
Loading User Symbols

0: kd> !cs -l -o -s
-----
DebugInfo      = 0x00000000000499d90
CriticalSection = 0x000007fefe3d5900 (SHELL32!g_csIconCache+0x0)
LOCKED
LockCount      = 0x2
WaiterWoken    = No
OwningThread   = 0x0000000000002b34
RecursionCount = 0x1
LockSemaphore  = 0x7F8
SpinCount       = 0x0000000000000000
OwningThread   = .thread ffffffa8013dc3b00

THREAD ffffffa8013dc3b00 Cid 4884.2b34 Teb: 000007fffffac000 Win32Thread: fffff900c2bc1010 WAIT:
(Executive) KernelMode Non-Alertable
    fffff88011c03600 SynchronizationEvent
IRP List:
    ffffffa800f8fc790: (0006,0430) Flags: 00000404 Mdl: 00000000
Not impersonating
DeviceMap        ffffff8a014e21d90
Owning Process   ffffffa80131a75d0 Image: explorer.exe
Attached Process N/A Image: N/A
Wait Start TickCount 170052 Ticks: 317148 (0:01:22:35.437)
Context Switch Count 2 LargeStack
UserTime         00:00:00.000
KernelTime       00:00:00.000
Win32 Start Address SHELL32!ShutdownThreadProc (0x000007fefe13ef54)
Stack Init fffff88011c03db0 Current fffff88011c03320
Base fffff88011c04000 Limit fffff88011bf000 Call 0
Priority 11 BasePriority 8 UnusualBoost 0 ForegroundBoost 2 IoPriority 2 PagePriority 5
Child-SP          RetAddr           Call Site
ffffff880`11c03360 ffffff800`01873652 nt!KiSwapContext+0x7a
ffffff880`11c034a0 ffffff800`01884a9f nt!KiCommitThreadWait+0x1d2
ffffff880`11c03530 ffffff880`05c12383 nt!KeWaitForSingleObject+0x19f
ffffff880`11c035d0 ffffff880`012b9288 ModuleA+0x12468
ffffff880`11c03750 ffffff880`012b7d1b fltmgr!FltpPerformPostCallbacks+0x368
ffffff880`11c03820 ffffff880`012b66df fltmgr!FltpLegacyProcessingAfterPreCallbacksCompleted+0x39b
ffffff880`11c038b0 ffffff880`01b895ff fltmgr!FltpDispatch+0xcf
ffffff880`11c03a30 ffffff800`01b783b4 nt!IopCloseFile+0x11f
ffffff880`11c03ac0 ffffff800`01b78171 nt!ObpDecrementHandleCount+0xb4
ffffff880`11c03b40 ffffff800`01b78734 nt!ObpCloseHandleTableEntry+0xb1
ffffff880`11c03bd0 ffffff800`0187ced3 nt!ObpCloseHandle+0x94
ffffff880`11c03c20 00000000`76d8140a nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ fffff880`11c03c20)
00000000`0754f348 000007fe`fd341873 ntdll!NtClose+0xa
00000000`0754f350 00000000`76c32f51 KERNELBASE!CloseHandle+0x13
00000000`0754f380 000007fe`fdaf9690 kernel32!CloseHandleImplementation+0x3d
00000000`0754f490 000007fe`fe191d7f SHLWAPI!CFileStream::Release+0x84
00000000`0754f4c0 000007fe`fe13ed57 SHELL32!IconCacheSave+0x2b7
00000000`0754f780 000007fe`fe13f06f SHELL32!CommonRestart+0x2f
00000000`0754f7f0 00000000`76c2652d SHELL32!ShutdownThreadProc+0x172
00000000`0754f820 00000000`76d5c521 kernel32!BaseThreadInitThunk+0xd
00000000`0754f850 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

```

## Divide by Zero

### Kernel Mode

This is a kernel mode counterpart of **Divide by Zero** pattern in user mode (page 259). It manifests under different bugchecks, for example:

```
1: kd> !analyze -v

[...]

UNEXPECTED_KERNEL_MODE_TRAP (7f)
This means a trap occurred in kernel mode, and it's a trap of a kind that the kernel isn't allowed to
have/catch (bound trap) or that is always instant death (double fault). The first number in the bugcheck
params is the number of the trap (8 = double fault, etc) Consult an Intel x86 family manual to learn more
about what these traps are. Here is a *portion* of those codes:
If kv shows a taskGate
    use .tss on the part before the colon, then kv.
Else if kv shows a trapframe
    use .trap on that value
Else
    .trap on the appropriate frame will show where the trap was taken
        (on x86, this will be the ebp that goes with the procedure KiTrap)
Endif
kb will then show the corrected stack.
Arguments:
Arg1: 00000000, EXCEPTION_DIVIDED_BY_ZERO
Arg2: 00000000
Arg3: 00000000
Arg4: 00000000

[...]

TRAP_FRAME: a8954c8c -- (.trap 0xfffffffffa8954c8c)
ErrCode = 00000000
eax=ffffffff ebx=00000000 ecx=00000005 edx=00000000 esi=00000000 edi=00000000
eip=975c42cd esp=a8954d00 ebp=a8954d4c iopl=0 nv up ei pl zr na pe nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010246
win32k!NtGdiEnumObjects+0xc6:
975c42cd f7f6 div eax,esi
Resetting default scope

PROCESS_NAME: Application.EXE

[...]
```

```

STACK_TEXT:
a8954c2c 81ac2b76 0000007f 5317512a 975c42cd nt!KeBugCheck+0x14
a8954c80 81899808 a8954c8c a8954d4c 975c42cd nt!Ki386CheckDivideByZeroTrap+0x44
a8954c80 975c42cd a8954c8c a8954d4c 975c42cd nt!KiTrap00+0x88
a8954d4c 81898a7a 062102ce 00000001 00000000 Driver!EnumObjects+0xc6
a8954d4c 77a59a94 062102ce 00000001 00000000 nt!KiFastCallEntry+0x12a
WARNING: Frame IP not in any known module. Following frames may be wrong.
0012ca70 00000000 00000000 00000000 00000000 0x77a59a94

0: kd> !analyze -v

[...]

SYSTEM_SERVICE_EXCEPTION (3b)
An exception happened while executing a system service routine.
Arguments:
Arg1: 00000000c0000094, Exception code that caused the bugcheck
Arg2: fffff9600025ba6d, Address of the exception record for the exception that caused the bugcheck
Arg3: fffff8800ac361d0, Address of the context record for the exception that caused the bugcheck
Arg4: 0000000000000000, zero.

[...]

EXCEPTION_CODE: (NTSTATUS) 0xc0000094 - {EXCEPTION} Integer division by zero.

FAULTING_IP:
Driver!EnumObjects+e9
fffff960`0025ba6d f7f6 div eax,esi

CONTEXT: fffff8800ac361d0 -- (.cxr 0xfffff8800ac361d0)
rax=0000000000000000 rbx=0000000000000000 rcx=0000000000000000
rdx=0000000000000000 rsi=0000000000000000 rdi=0000000000000000
rip=fffff9600025ba6d rsp=fffff8800ac36ba0 rbp=fffff8800ac36ca0
r8=0000000000000000 r9=0000000000000000 r10=0000000005892f18
r11=fffff900c28379e0 r12=0000000000000000 r13=0000000000000002
r14=0000000000000001 r15=0000000000000000
iopl=0 nv up ei ng nz na po nc
cs=0010 ss=0018 ds=002b es=002b fs=0053 gs=002b efl=00010286
Driver!EnumObjects+0xe9:
fffff960`0025ba6d f7f6 div eax,esi
Resetting default scope

[...]

STACK_TEXT:
fffff880`0ac36ba0 fffff800`01682993 Driver!EnumObjects+0xe9
fffff880`0ac36c20 00000000`748a1b3a nt!KiSystemServiceCopyEnd+0x13
00000000`001cdf08 00000000`00000000 0x748a1b3a

```

## User Mode

### Linux

This is a Linux variant of **Divide by Zero** (user mode) pattern previously described for Mac OS X (page 258) and Windows (page 259) platforms:

```
GNU gdb (GDB)
[...]
Program terminated with signal 8, Arithmetic exception.
#0 0x00000000040056f in procD ()

(gdb) x/i $rip
=> 0x40056f <procD+18>: idivl -0x8(%rbp)

(gdb) info r $rax
rax 0x1 1

(gdb) x/w $rbp-0x8
0x7f0f6806bd28: 0x00000000
```

## Mac OS X

This is a Mac OS X / GDB counterpart to **Divide by Zero** (user mode) described for Windows platforms (page 259):

```
(gdb) bt
#0 0x000000010d3ebe9e in bar (a=1, b=0)
#1 0x000000010d3ebec3 in foo ()
#2 0x000000010d3ebeeb in main (argc=1, argv=0x7fff6cfeab18)

(gdb) x/i 0x000000010d3ebe9e
0x10d3ebe9e : idiv %esi

(gdb) info r rsi
rsi 0x0 0
```

The modeling application source code:

```
int bar(int a, int b)
{
    return a/b;
}

int foo()
{
    return bar(1,0);
}

int main(int argc, const char * argv[])
{
    return foo();
}
```

## Windows

Integer division by zero is one of the most frequent exceptions<sup>45</sup>. It is easily recognizable in process crash dumps by the processor instruction that caused this exception type (DIV or IDIV):

```
FAULTING_IP:
DLL!FindHighestID+278
1b2713c4 f775e4 div dword ptr [ebp-0x1c]

EXCEPTION_RECORD: ffffffff -- (.exr ffffffffffffc0000094)
ExceptionAddress: 1b2713c4 (DLL!FindHighestID+0x00000278)
ExceptionCode: c0000094 (Integer divide-by-zero)
ExceptionFlags: 00000000
NumberParameters: 0
```

or

```
FAULTING_IP:
Application+263d8
004263d8 f7fe idiv eax,esi

EXCEPTION_RECORD: ffffffff -- (.exr 0xfffffffffffffc0000094)
ExceptionAddress: 004263d8 (Application+0x000263d8)
ExceptionCode: c0000094 (Integer divide-by-zero)
ExceptionFlags: 00000000
NumberParameters: 0

ERROR_CODE: (NTSTATUS) 0xc0000094 - {EXCEPTION} Integer division by zero.
```

---

<sup>45</sup> Win32 Exception Frequencies, Memory Dump Analysis Anthology, Volume 2, page 427

## Double Free

### Kernel Pool

In contrast to **Double Free** pattern (page 267) in a user mode process heap double free in a kernel mode pool results in an immediate bugcheck in order to identify the driver causing the problem (BAD\_POOL\_CALLER bugcheck with Arg1 == 7):

```
2: kd> !analyze -v
...
...
BAD_POOL_CALLER (c2)
The current thread is making a bad pool request. Typically this is at a bad IRQL level or double freeing
the same allocation, etc.
Arguments:
Arg1: 00000007, Attempt to free pool which was already freed
Arg2: 0000121a, (reserved)
Arg3: 02140001, Memory contents of the pool block
Arg4: 89ba74f0, Address of the block of pool being deallocated
```

If we look at the block being deallocated we would see that it was marked as "Free" block:

```
2: kd> !pool 89ba74f0
Pool page 89ba74f0 region is Nonpaged pool
89ba7000 size: 270 previous size: 0 (Allocated) Thre (Protected)
89ba7270 size: 8 previous size: 270 (Free) ....
89ba7278 size: 18 previous size: 8 (Allocated) ReEv
89ba7290 size: 80 previous size: 18 (Allocated) Mdl
89ba7310 size: 80 previous size: 80 (Allocated) Mdl
89ba7390 size: 30 previous size: 80 (Allocated) Vad
89ba73c0 size: 98 previous size: 30 (Allocated) File (Protected)
89ba7458 size: 8 previous size: 98 (Free) Wait
89ba7460 size: 28 previous size: 8 (Allocated) FSfm
89ba74a0 size: 40 previous size: 18 (Allocated) Ntfr
89ba74e0 size: 8 previous size: 40 (Free) File
*89ba74e8 size: a0 previous size: 8 (Free ) *ABCD
Owning component : Unknown (update pooltag.txt)
89ba7588 size: 38 previous size: a0 (Allocated) Sema (Protected)
89ba75c0 size: 38 previous size: 38 (Allocated) Sema (Protected)
89ba75f8 size: 10 previous size: 38 (Free) Nbt1
89ba7608 size: 98 previous size: 10 (Allocated) File (Protected)
89ba76a0 size: 28 previous size: 98 (Allocated) Ntfn
89ba76c8 size: 40 previous size: 28 (Allocated) Ntfr
89ba7708 size: 28 previous size: 40 (Allocated) NtFs
89ba7730 size: 40 previous size: 28 (Allocated) Ntfr
89ba7770 size: 40 previous size: 40 (Allocated) Ntfr
89ba7a10 size: 270 previous size: 260 (Allocated) Thre (Protected)
89ba7c80 size: 20 previous size: 270 (Allocated) VadS
```

The pool tag is a 4-byte character sequence used to associate drivers with pool blocks and is useful to identify a driver allocated or freed a block. In our case, the pool tag is ABCD, and it is associated with the

driver that previously freed the block. All known pool tags corresponding to kernel components can be found in *pooltag.txt* located in triage subfolder where WinDbg is installed. However, our ABCD tag is not listed there. We can try to find the driver corresponding to ABCD tag using **findstr** CMD command:

```
C:\Windows\System32\drivers>findstr /m /l hABCD *.sys
```

The results of the search will help us to identify the driver which freed the block first. The driver that double freed the same block can be found on the call stack, and it may be the same driver or a different driver:

```
2: kd> k
ChildEBP RetAddr
f78be910 8089c8f4 nt!KeBugCheckEx+0x1b
f78be978 8089c622 nt!ExFreePoolWithTag+0x477
f78be988 f503968b nt!ExFreePool+0xf
WARNING: Stack unwind information not available. Following frames may be wrong.
f78be990 f5024a6e driver+0x1768b
f78be9a0 f50249e7 driver+0x2a6e
f78be9a4 84b430e0 driver+0x29e7
```

Because we don't have symbol files for *driver.sys* WinDbg warns us that it was unable to identify the correct stack trace and *driver.sys* might not have called *ExFreePool* or *ExFreePoolWithTag* function. To verify that *driver.sys* called *ExFreePool* function indeed we disassemble backward the return address of it:

```
2: kd> ub f503968b
driver+0x1767b:
f503967b 90          nop
f503967c 90          nop
f503967d 90          nop
f503967e 90          nop
f503967f 90          nop
f5039680 8b442404    mov     eax,dword ptr [esp+4]
f5039684 50          push    eax
f5039685 ff15202302f5  call    dword ptr [driver+0x320 (f5022320)]
```

Finally, we can get some info from the driver:

```
2: kd> lmv m driver
start   end     module name
f5022000 f503e400  driver  (no symbols)
Loaded symbol image file: driver.sys
Image path: \SystemRoot\System32\drivers\driver.sys
Image name: driver.sys
Timestamp: Tue Aug 12 11:32:16 2007
```

If the company name developed the driver is absent, we could try techniques outlined in **Unknown Component** pattern (page 1035).

If we have symbols it is very easy to identify the code as can be seen from this 64-bit crash dump:

```
BAD_POOL_CALLER (c2)
The current thread is making a bad pool request. Typically this is at a bad IRQL level or double freeing
the same allocation, etc.
Arguments:
Arg1: 0000000000000007, Attempt to free pool which was already freed
Arg2: 00000000000121a, (reserved)
Arg3: 0000000000000080, Memory contents of the pool block
Arg4: fffffade6d54e270, Address of the block of pool being deallocated

0: kd> kL
fffffade`45517b08 fffff800`011ad905 nt!KeBugCheckEx
fffffade`45517b10 fffffade`5f5991ac nt!ExFreePoolWithTag+0x401
fffffade`45517bd0 fffffade`5f59a0b0 driver64!ProcessDataItem+0x198
fffffade`45517c70 fffffade`5f5885a6 driver64!OnDataArrival+0x2b4
fffffade`45517cd0 fffff800`01299cae driver64!ReaderThread+0x15a
fffffade`45517d70 fffff800`0102bbe6 nt!PspSystemThreadStartup+0x3e
fffffade`45517dd0 00000000`00000000 nt!KiStartSystemThread+0x16
```

## Comments

One of the asked questions was: what happens if the block being deallocated can not be analyzed:

```
4: kd> !pool a4ef7920
Pool page a4ef7920 region is Nonpaged pool
a4ef7000 size: e0 previous size: 0 (Allocated) MmCi
a4ef70e0 size: 68 previous size: e0 (Allocated) TCIZ
a4ef7148 size: e0 previous size: 68 (Allocated) MmCi
a4ef7228 size: 98 previous size: e0 (Allocated) File (Protected)
a4ef72c0 size: 98 previous size: 98 (Allocated) File (Protected)
a4ef7358 size: 100 previous size: 98 (Allocated) MmCi
a4ef7458 size: 28 previous size: 100 (Allocated) NtFs
a4ef7480 size: 40 previous size: 28 (Allocated) Ntfr
a4ef74c0 size: 98 previous size: 40 (Allocated) File (Protected)
a4ef7558 size: 8 previous size: 98 (Free) CcPL
a4ef7560 size: 40 previous size: 8 (Allocated) SevE
a4ef75a0 size: 40 previous size: 40 (Allocated) Ntfr
a4ef75e0 size: 10 previous size: 40 (Free) TCI1
a4ef75f0 size: 180 previous size: 10 (Allocated) MmCi
a4ef7770 size: 98 previous size: 180 (Allocated) File (Protected)
a4ef7808 size: 88 previous size: 98 (Allocated) Adap (Protected)
a4ef7890 size: 88 previous size: 88 (Allocated) NEtd
a4ef7918 is not a valid large pool allocation, checking large session pool...
a4ef7918 is freed (or corrupt) pool
Bad allocation size @a4ef7918, zero is invalid

***
*** An error (or corruption) in the pool was detected;
*** Attempting to diagnose the problem.
***
*** Use !poolval a4ef7000 for more details.
***
```

```
Pool page [ a4ef7000 ] is __inVALID.

Analyzing linked list...
[ a4ef7890 -> a4ef7a10 (size = 0x180 bytes)]: Corrupt region

Scanning for single bit errors...

None found
```

The problem block start address can be calculated:

```
a4ef7890+88 = a4ef7918
```

We would try to see its contents, perhaps **dds** and **dps** would point to some symbolic data. Also, search for this address in kernel space might point to some other blocks as well. If we suspect some driver, we may want to enable Driver Verifier special pool.

Another asked question: the special pool is enabled, and it shows both free operation happening through the same thread and the stacks are exactly same:

```
2: kd> .bugcheck
Bugcheck code 000000C2
Arguments 00000000`00000007 00000000`00001097 00000000`00210007 ffffff8a0`04b98e00

2: kd> !pool ffffff8a0`04b98e00 2
Pool page ffffff8a004b98e00 region is Paged pool
*fffff8a004b98df0 size: 210 previous size: 70 (Free) *MmSt
Pooltag MmSt : Mm section object prototype ptes, Binary : nt!mm
```

```
2: kd> !verifier 0x80 ffffff8a0`04b98e00
```

Log of recent kernel pool Allocate and Free operations:

There are up to 0x10000 entries in the log.

Parsing 0x0000000000010000 log entries, searching for address 0xfffff8a004b98e00.

```
=====
Pool block ffffff8a004b98df0, Size 000000000000210, Thread ffffffa80122674f0
fffff80001b0bc9a nt!VfFreePoolNotification+0x4a
fffff800017a367c nt!ExDeferredFreePool+0x126d
fffff8000165b880 nt!MiDeleteSegmentPages+0x35c
fffff8000195cf2f nt!MiSegmentDelete+0x7b
fffff80001637e07 nt!MiCleanSection+0x2f7
fffff80001676754 nt!ObfDereferenceObject+0xd4
fffff80001661170 nt!CcDeleteSharedCacheMap+0x1bc
fffff80001699880 nt!CcUninitializeCacheMap+0x2f0
fffff880030ecfa6 ModuleA!ProcC+0x4b6
fffff880030ec840 ModuleA!ProcB+0x2a8
fffff880030ec994 ModuleA!ProcA+0x38
```

```
fffff80001b16750 nt!IovCallDriver+0xa0
fffff800019824bf nt!IopCloseFile+0x11f
=====
Pool block fffff8a004b98df0, Size 0000000000000210, Thread fffffa80122674f0
fffff80001b0bc9a nt!VfFreePoolNotification+0x4a
fffff800017a367c nt!ExDeferredFreePool+0x126d
fffff8000165b880 nt!MiDeleteSegmentPages+0x35c
fffff8000195cf2f nt!MiSegmentDelete+0x7b
fffff80001637e07 nt!MiCleanSection+0x2f7
fffff80001676754 nt!ObfDereferenceObject+0xd4
fffff80001661170 nt!CcDeleteSharedCacheMap+0x1bc
fffff80001699880 nt!CcUninitializeCacheMap+0x2f0
fffff880030ecfa6 ModuleA!ProcC+0x4b6
fffff880030ec840 ModuleA!ProcB+0x2a8
fffff880030ec994 ModuleA!ProcA+0x38
fffff80001b16750 nt!IovCallDriver+0xa0
fffff800019824bf nt!IopCloseFile+0x11f
```

But current thread has no sign of *ModuleA.sys*:

```
2: kd> k
Child-SP RetAddr Call Site
fffff880`02378b28 fffff800`017a360e nt!KeBugCheckEx
fffff880`02378b30 fffff800`0178a53e nt!ExDeferredFreePool+0x11eb
fffff880`02378be0 fffff800`01798a0a nt!MiDeleteCachedSubsection+0x10ae
fffff880`02378c90 fffff800`01798b43 nt!MiRemoveUnusedSegments+0x8a
fffff880`02378cc0 fffff800`01910726 nt!MiDereferenceSegmentThread+0x103
fffff880`02378d40 fffff800`0164fae6 nt!PspSystemThreadStartup+0x5a
fffff880`02378d80 00000000`00000000 nt!KiStartSystemThread+0x16
```

Two logged free operations clearly show **Double Free**. But it was detected at a later time by a different thread.

Recently we found that search needs to be done without h prefix:

```
findstr /m /l ABCD
```

If we see some leaking pool tag we can find its entries using this command **!poolfind ABCD** and then dump their memory contents.

Another example from a Software Diagnostics Library user:

```
Windows Server 2003 Kernel Version 3790 (Service Pack 2) MP (8 procs) Free x86 compatible
Product: Server, suite: Enterprise TerminalServer SingleUserTS
Built by: 3790.srv03_sp2_rtm.070216-1710
Kernel base = 0x80800000 PsLoadedModuleList = 0x808a6ea8
Debug session time: Wed May 21 23:55:16.743 2008 (GMT+8)
System Uptime: 1 days 5:48:24.125
Loading Kernel Symbols
Loading User Symbols
Loading unloaded module list
*
* Bugcheck Analysis
*
```

Use !analyze -v to get detailed debugging information.

BugCheck C2, {7, 121a, 0, 8b6e6d00}

Probably caused by : Fs\_Rec.SYS ( Fs\_Rec!UdfsRecFsControl+63 )

Followup: MachineOwner

---

3: kd> !analyze -v

\*

\* Bugcheck Analysis

\*

BAD\_POOL\_CALLER (c2)

The current thread is making a bad pool request. Typically this is at a bad IRQL level or double freeing the same allocation, etc.

Arguments:

Arg1: 00000007, Attempt to free pool which was already freed

Arg2: 0000121a, (reserved)

Arg3: 00000000, Memory contents of the pool block

Arg4: 8b6e6d00, Address of the block of pool being deallocated

Debugging Details:

---

POOL\_ADDRESS: 8b6e6d00

FREED\_POOL\_TAG: Thre

BUGCHECK\_STR: 0xc2\_7\_Thre

CUSTOMER\_CRASH\_COUNT: 1

DEFAULT\_BUCKET\_ID: DRIVER\_FAULT\_SERVER\_MINIDUMP

PROCESS\_NAME: Rtvscan.exe

CURRENT\_IRQL: 0

LAST\_CONTROL\_TRANSFER: from 808927bb to 80827c63

STACK\_TEXT:

b86e18c0 808927bb 000000c2 00000007 0000121a nt!KeBugCheckEx+0x1b  
b86e1928 8081e1b6 8b6e6d00 00000000 8b6e6af8 nt!ExFreePoolWithTag+0x477  
b86e1954 f78037a1 8c8c8af8 8b95d030 b86e1988 nt!IopfCompleteRequest+0x180  
b86e1964 f780309e 8c8c8af8 8a3aab8 8c897730 Fs\_Rec!UdfsRecFsControl+0x63  
b86e1974 8081df65 8c8c8af8 8a3aab8 Fs\_Rec!FsRecFsControl+0x5a  
b86e1988 808f785c 80a5a4d0 8b95d030 80a5a540 nt!IofCallDriver+0x45  
b86e19d8 808220a4 8c8c8af8 b86e1c00 00000000 nt!IopMountVolume+0x1b4  
b86e1a04 808f8910 b86e1c38 8b95d000 b86e1b40 nt!IopCheckVpbMounted+0x5c  
b86e1afc 80937942 8b95d030 00000000 8a8d01e0 nt!IopParseDevice+0x3d4

```
b86e1b7c 80933a76 00000000 b86e1bbc 00000040 nt!ObpLookupObjectName+0x5b0
b86e1bd0 808ec76b 00000000 00000000 b86e1c01 nt!ObOpenObjectByName+0xea
b86e1d54 8088978c 05e7e2b4 05e7e28c 05e7e2d4 nt!NtQueryAttributesFile+0x11d
b86e1d54 7c8285ec 05e7e2b4 05e7e28c 05e7e2d4 nt!KiFastCallEntry+0xfc
WARNING: Frame IP not in any known module. Following frames may be wrong.
05e7e2d4 00000000 00000000 00000000 00000000 0x7c8285ec
```

STACK\_COMMAND: kb

FOLLOWUP\_IP:

```
Fs_Rec!UdfsRecFsControl+63
f78037a1 8bc6 mov eax,esi
```

SYMBOL\_STACK\_INDEX: 3

SYMBOL\_NAME: Fs\_Rec!UdfsRecFsControl+63

FOLLOWUP\_NAME: MachineOwner

MODULE\_NAME: Fs\_Rec

IMAGE\_NAME: Fs\_Rec.SYS

DEBUG\_FLR\_IMAGE\_TIMESTAMP: 3e800074

FAILURE\_BUCKET\_ID: 0xc2\_7\_Thre\_Fs\_Rec!UdfsRecFsControl+63

BUCKET\_ID: 0xc2\_7\_Thre\_Fs\_Rec!UdfsRecFsControl+63

Followup: MachineOwner

An example involving special pool:

```
DRIVER_PAGE_FAULT_IN_FREED_SPECIAL_POOL (d5)
Memory was referenced after it was freed.
This cannot be protected by try-except.
When possible, the guilty driver's name (Unicode string) is printed on
the bugcheck screen and saved in KiBugCheckDriver.
```

## Process Heap

### Windows

Double-free bugs lead to **Dynamic Memory Corruption** pattern (page 307). The reason why **Double Free** deserves its own pattern name is the fact that either debug runtime libraries or even OS itself detect such bugs and save crash dumps immediately.

For some heap implementations, the double free operation doesn't lead to an immediate heap corruption and subsequent crash. For example, if we allocate 3 blocks in a row and then free the middle one twice there will be no crash as the second free call is able to detect that the block was already freed and does nothing. The following program loops forever and never crashes:

```
#include "stdafx.h"
#include <windows.h>

int _tmain(int argc, _TCHAR* argv[])
{
    while (true)
    {
        puts("Allocate: p1");
        void *p1 = malloc(100);
        puts("Allocate: p2");
        void *p2 = malloc(100);
        puts("Allocate: p3");
        void *p3 = malloc(100);

        puts("Free: p2");
        free(p2);
        puts("Double-Free: p2");
        free(p2);
        puts("Free: p1");
        free(p1);
        puts("Free: p3");
        free(p3);

        Sleep(100);
    }

    return 0;
}
```

The output of the program:

```
...
...
...
Allocate: p1
Allocate: p2
Allocate: p3
Free: p2
Double-Free: p2
Free: p1
Free: p3
Allocate: p1
Allocate: p2
Allocate: p3
Free: p2
Double-Free: p2
Free: p1
Free: p3
Allocate: p1
Allocate: p2
Allocate: p3
Free: p2
Double-Free: p2
...
...
...
```

However, if a free call triggered heap coalescence (adjacent free blocks form the bigger free block) then we have a heap corruption crash on the next double-free call because the coalescence triggered by the previous free call erased the free block information:

```
#include "stdafx.h"
#include <windows.h>

int _tmain(int argc, _TCHAR* argv[])
{
    while (true)
    {
        puts("Allocate: p1");
        void *p1 = malloc(100);
        puts("Allocate: p2");
        void *p2 = malloc(100);
        puts("Allocate: p3");
        void *p3 = malloc(100);

        puts("Free: p3");
        free(p3);
        puts("Free: p1");
        free(p1);
        puts("Free: p2");
        free(p2);
        puts("Double-Free: p2");
        free(p2);
```

```

        Sleep(100);
    }

    return 0;
}

```

The output of the program:

```

Allocate: p1
Allocate: p2
Allocate: p3
Free: p3
Free: p1
Free: p2
Double-Free: p2
Crash!

```

If we open a crash dump we will see the following **Stack Trace** (page 926):

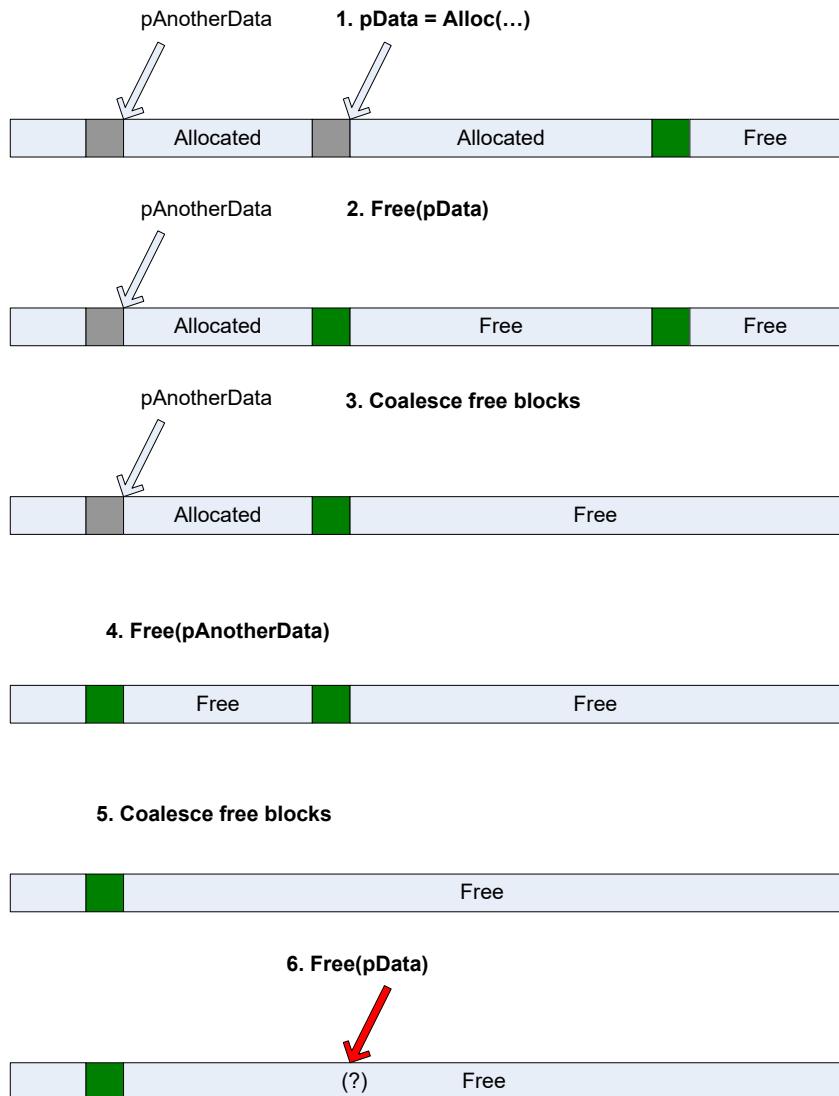
```

0:000> r
eax=00922130 ebx=00920000 ecx=10101010 edx=10101010 esi=00922128 edi=00921fc8
eip=76ee1ad5 esp=0012fd6c ebp=0012fd94 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010246
ntdll!RtlpCoalesceFreeBlocks+0x6ef:
76ee1ad5 8b4904          mov     ecx,dword ptr [ecx+4] ds:0023:10101014=?????????

0:000> kL
ChildEBP RetAddr
0012fd94 76ee1d37 ntdll!RtlpCoalesceFreeBlocks+0x6ef
0012fe8c 76ee1c21 ntdll!RtlpFreeHeap+0x1e2
0012fea8 758d7a7e ntdll!RtlFreeHeap+0x14e
0012feb0 6cff4c39 kernel32!HeapFree+0x14
0012ff08 0040107b msavr80!free+0xcd
0012ff5c 004011f1 DoubleFree!wmain+0x7b
0012ffa0 758d3833 DoubleFree!__tmainCRTStartup+0x10f
0012ffac 76eba9bd kernel32!BaseThreadInitThunk+0xe
0012ffec 00000000 ntdll!_RtlUserThreadStart+0x23

```

This is illustrated in the following picture where free calls result in heap coalescence, and the subsequent double-free call corrupts the heap:



The problem here is that heap coalescence can be triggered sometime after the double free, so we need some solution to diagnose double-free bugs earlier, ideally at the first double-free call. For example, the following program crashes during the normal free operation long after the first double-free happened:

```
#include "stdafx.h"
#include <windows.h>

int _tmain(int argc, _TCHAR* argv[])
{
    while (true)
    {
        puts("Allocate: p1");
        void *p1 = malloc(100);
        puts("Allocate: p2");
        void *p2 = malloc(100);
```

```
puts("Allocate: p3");
void *p3 = malloc(100);

puts("Free: p1");
free(p1);
puts("Free: p2");
free(p2);
puts("Double-Free: p2");
free(p2);
puts("Double-Free: p3");
free(p3);

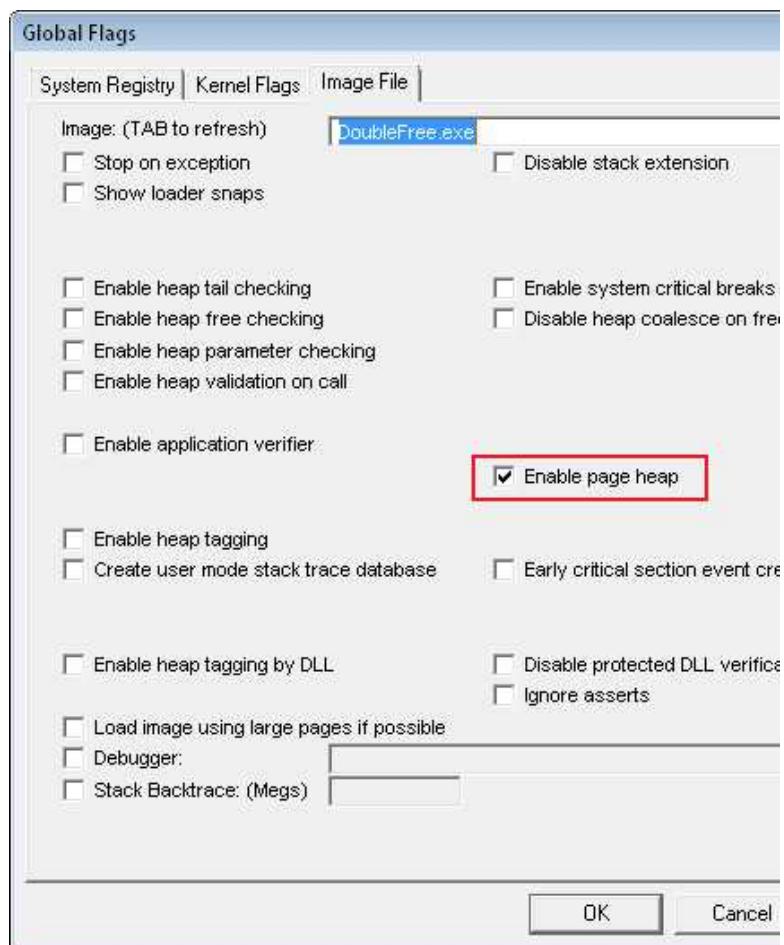
Sleep(100);
}

return 0;
}
```

The output of the program:

```
Allocate: p1
Allocate: p2
Allocate: p3
Free: p1
Free: p2
Double-Free: p2
Free: p3
Allocate: p1
Allocate: p2
Allocate: p3
Free: p1
Free: p2
Double-Free: p2
Free: p3
Allocate: p1
Allocate: p2
Allocate: p3
Free: p1
Free: p2
Double-Free: p2
Free: p3
Allocate: p1
Allocate: p2
Allocate: p3
Free: p1
Free: p2
Double-Free: p2
Free: p3
Crash!
```

If we enable full page heap using *gflags.exe* from Debugging Tools for Windows the program crashes immediately on the double free call:



```
Allocate: p1
Allocate: p2
Allocate: p3
Free: p1
Free: p2
Double-Free: p2
Crash!
```

The crash dump shows the following **Stack Trace** (page 926):

```
0:000> kL
ChildEBP RetAddr
0012f810 71aa4ced ntdll!DbgBreakPoint+0x1
0012f834 71aa9fc2 verifier!VerifierStopMessage+0x1fd
0012f890 71aaa4da verifier!AVrfpDphReportCorruptedBlock+0x102
0012f8a4 71ab2c98 verifier!AVrfpDphCheckNormalHeapBlock+0x18a
0012f8b8 71ab2a0e verifier!_EH4_CallFilterFunc+0x12
```

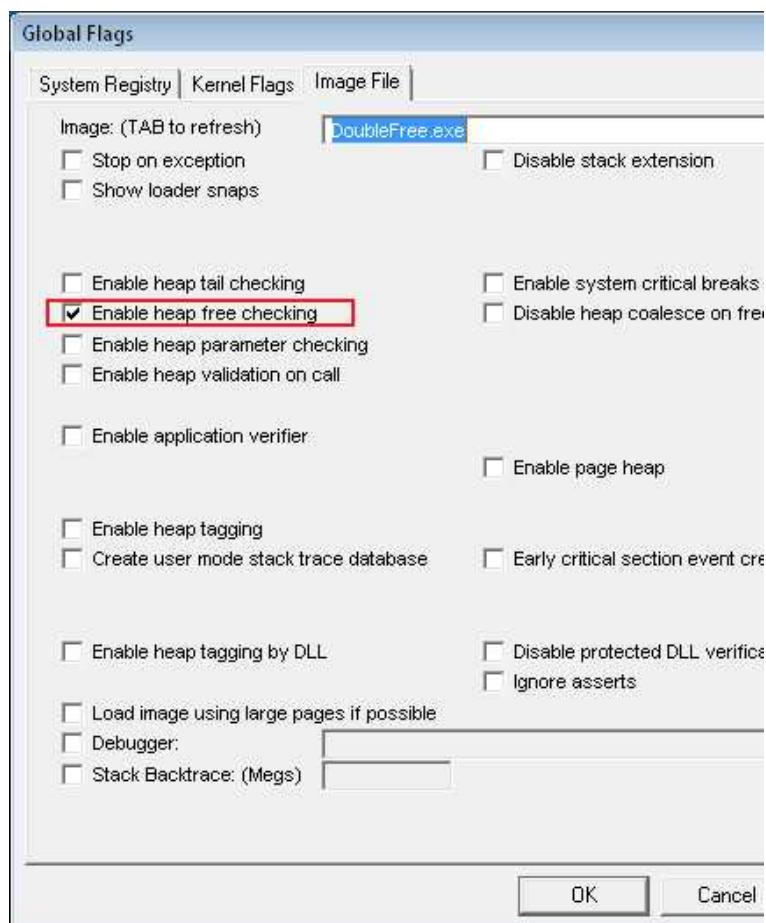
```

0012f8e0 76ee1039 verifier!_except_handler4+0x8e
0012f904 76ee100b ntdll!ExecuteHandler2+0x26
0012f9ac 76ee0e97 ntdll!ExecuteHandler+0x24
0012f9ac 71aaa3ad ntdll!KiUserExceptionDispatcher+0xf
0012fcf0 71aaa920 verifier!AVrfpDphCheckNormalHeapBlock+0x5d
0012fd0c 71aa879b verifier!AVrfpDphNormalHeapFree+0x20
0012fd60 76f31c8f verifier!AVrfDebugPageHeapFree+0x1cb
0012fda8 76efd9fa ntdll!RtlDebugFreeHeap+0x2f
0012fe9c 76ee1c21 ntdll!RtlpFreeHeap+0x5f
0012feb8 758d7a7e ntdll!RtlFreeHeap+0x14e
0012fecf 6cff4c39 kernel32!HeapFree+0x14
0012ff18 0040105f msrv80!free+0xcd
0012ff5c 004011f1 DoubleFree!wmain+0x5f
0012ffa0 758d3833 DoubleFree!__tmainCRTStartup+0x10f
0012ffac 76eba9bd kernel32!BaseThreadInitThunk+0xe

0:000> !gflag
Current NtGlobalFlag contents: 0x02000000
  hpa - Place heap allocations at ends of pages

```

If we enable heap free checking instead of page heap we get our crash on the first double free call immediately too:



```
Allocate: p1
Allocate: p2
Allocate: p3
Free: p1
Free: p2
Double-Free: p2
Crash!
```

The crash dump shows the following **Stack Trace** (page 926):

```
0:000> r
eax=feeefeee ebx=001b2040 ecx=001b0000 edx=001b2040 esi=d4476047 edi=001b2038
eip=76ee2086 esp=0012fe68 ebp=0012fe9c iopl=0 nv up ei ng nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010286
ntdll!RtlpLowFragHeapFree+0x31:
76ee2086 8b4604          mov     eax,dword ptr [esi+4] ds:0023:d447604b=????????
```

```
0:000> kL
ChildEBP RetAddr
0012fe9c 76ee18c3 ntdll!RtlpLowFragHeapFree+0x31
0012feb0 758d7a7e ntdll!RtlFreeHeap+0x101
0012fec4 6cff4c39 kernel32!HeapFree+0x14
0012ff10 0040106d msocr80!free+0xcd
0012ff5c 004011f1 DoubleFree!wmain+0x6d
0012ffa0 758d3833 DoubleFree!_tmainCRTStartup+0x10f
0012ffac 76eba9bd kernel32!BaseThreadInitThunk+0xe
0012ffec 00000000 ntdll!_RtlUserThreadStart+0x23

0:000> !gflag
Current NtGlobalFlag contents: 0x00000020
    hfc - Enable heap free checking
```

## Comments

An example of **Double Free** detected in Windows 7:

```
0:048> k
ChildEBP RetAddr
206dee98 777f8567 ntdll!ZwWaitForSingleObject+0x15
206def1c 777f8695 ntdll!RtlReportExceptionEx+0x14b
206def74 7781e6e6 ntdll!RtlReportException+0x86
206def88 7781e763 ntdll!RtlpTerminateFailureFilter+0x14
206def94 777c73dc ntdll!RtlReportCriticalFailure+0x67
206defa8 777c7281 ntdll!_EH4_CallFilterFunc+0x12
206defd0 777ab499 ntdll!_except_handler4+0x8e
206deff4 777ab46b ntdll!ExecuteHandler2+0x26
206df018 777ab40e ntdll!ExecuteHandler+0x24
206df0a4 77760133 ntdll!RtlDispatchException+0x127
206df0a4 7781e753 ntdll!KiUserExceptionDispatcher+0xf
206df5e8 7781f659 ntdll!RtlReportCriticalFailure+0x57
206df5f8 7781f739 ntdll!RtlpReportHeapFailure+0x21
206df62c 777ce045 ntdll!RtlpLogHeapFailure+0xa1
206df65c 76aa6e6a ntdll!RtlFreeHeap+0x64
206df670 58110076 ole32!CRetailMalloc_Free+0x1c [d:\w7rtm\com\ole32\com\class\memapi.cxx @ 687]
WARNING: Stack unwind information not available. Following frames may be wrong.
206df6ac 581100e9 OUTLMIME!MimeOleInetDateToFileTime+0xd562
206df6b8 5811051d OUTLMIME!MimeOleInetDateToFileTime+0xd5d5
206df6e0 771562fa OUTLMIME!MimeOleInetDateToFileTime+0xda09
206df70c 77156d3a user32!InternalCallWinProc+0x23
206df784 771577c4 user32!UserCallWinProcCheckWow+0x109
206df7e4 77157bca user32!DispatchMessageWorker+0x3bc
206df7f4 581d74e6 user32!DispatchMessageA+0xf
206df830 581e04a3 OUTLPH!D11GetClassObject+0x5616
206df84c 581df9ac OUTLPH!D11GetClassObject+0xe5d3
206df880 6e558488 OUTLPH!D11GetClassObject+0xdadc
206df8a4 650fa17d OLEAPI32!HrCreateAsyncArgSet+0x479
206df8e8 650f8221 MSO!Ordinal381+0x48d
206df908 650f80a9 MSO!Ordinal9712+0x237
206df924 650f32d6 MSO!Ordinal9712+0xbf
206df958 650efe05 MSO!Ordinal5368+0x382
206df9b4 7725338a MSO!MsoFInitOffice+0x363
206df9c0 77789f72 kernel32!BaseThreadInitThunk+0xe
206dfa00 77789f45 ntdll!__RtlUserThreadStart+0x70
206dfa18 00000000 ntdll!_RtlUserThreadStart+0x1b
```

Because we have the stack unwind warning, we double check the return address to verify that *OUTLMIME* module called heap free function. The call involves a triple indirection of 58149f04 pointer address:

```

0:048> ub 58110076
OUTLMIME!MimeOleInetDateToFileTime+0xd550:
58110064 8b0f      mov     ecx,dword ptr [edi]
58110066 3bc9      cmp     ecx,ebx
58110068 740e      je      OUTLMIME!MimeOleInetDateToFileTime+0xd564 (58110078)
5811006a a1049f1458 mov     eax,dword ptr [OUTLMIME!HrGetMIMESTreamForMAPIMsg+0xe528 (58149f04)]
5811006f 8b10      mov     edx,dword ptr [eax]
58110071 51        push    ecx
58110072 50        push    eax
58110073 ff5214    call    dword ptr [edx+14h]

0:048> dps poi(poi(58149f04))+14 L1
76b97264 76aa6e4e ole32!CRetailMalloc_Free [d:\w7rtm\com\ole32\com\class\memapi.cxx @ 680]

0:048> !heap -s
*****
*          *
*          HEAP ERROR DETECTED
*          *
*****
```

Details:

Heap address: 00280000  
**Error address:** 1cecd3e8  
Error type: HEAP\_FAILURE\_BLOCK\_NOT\_BUSY  
Details: The caller performed an operation (such as a free or a size check) that is **illegal on a free block**.  
Follow-up: Check the error's stack trace to find the culprit.

Stack trace:

```

777ce045: ntdll!RtlFreeHeap+0x00000064
76aa6e6a: ole32!CRetailMalloc_Free+0x0000001c
58110076: OUTLMIME!MimeOleInetDateToFileTime+0x0000d562
581100e9: OUTLMIME!MimeOleInetDateToFileTime+0x0000d5d5
5811051d: OUTLMIME!MimeOleInetDateToFileTime+0x0000da09
771562fa: user32!InternalCallWinProc+0x00000023
77156d3a: user32!UserCallWinProcCheckWow+0x000000109
771577c4: user32!DispatchMessageWorker+0x0000003bc
77157bca: user32!DispatchMessageA+0x0000000f
581d74e6: OUTLPH!DllGetClassObject+0x000005616
581e04a3: OUTLPH!DllGetClassObject+0x00000e5d3
581df9ac: OUTLPH!DllGetClassObject+0x00000dac
6e558488: OLEAPI32!HrCreateAsyncArgSet+0x00000479
650fa17d: MSO!Ordinal381+0x0000048d
650f8221: MSO!Ordinal9712+0x00000237
650f80a9: MSO!Ordinal9712+0x000000bf
```

[...]

```

0:048> !heap -x 1cecd3e8
Entry      User      Heap      Segment      Size   PrevSize  Unused   Flags
-----
```

Entry	User	Heap	Segment	Size	PrevSize	Unused	Flags
1cecd3e8	1cecd3f0	00280000	0f945f18	20	-	0	LFH;free

## Mac OS X

This is a Mac OS X / GDB counterpart to **Double Free** pattern:

```
(gdb) bt
#0 0x000007fff8479582a in __kill ()
#1 0x000007fff8e0e0a9c in abort ()
#2 0x000007fff8e13f84c in free ()
#3 0x00000001035a8ef4 in main (argc=1, argv=0x7fff631a7b20)

(gdb) x/2i 0x00000001035a8ef4-8
0x1035a8eec : mov -0x20(%rbp),%edi
0x1035a8eef : callq 0x1035a8f06

(gdb) frame 3
#3 0x00000001035a8ef4 in main (argc=1, argv=0x7fff631a7b20)
at .../DoubleFree/main.c:23
23 free(p2);
Current language: auto; currently minimal

(gdb) x/g $rbp-0x20
0x7fff631a7ae0: 0x000007fe6a8801400

(gdb) x/2w 0x000007fe6a8801400
0x7fe6a8801400: 0x00000000 0xb0000000
```

Here's the source code of the modeling application:

```
int main(int argc, const char * argv[])
{
    char *p1 = (char *) malloc (1024);
    printf("p1 = %p\n", p1);

    char *p2 = (char *) malloc (1024);
    printf("p2 = %p\n", p2);

    free(p2);
    free(p1);
    free(p2);

    return 0;
}
```

## Double IRP Completion

Similar to **Double Free** (process heap, page 267) and **Double Free** (kernel pool, page 260) that might be detected through **Instrumentation Information** (page 516) such as *gflags* and Driver Verifier there is also **Double IRP Completion** variant implemented through **Self-Diagnosis** (kernel mode, page 844). Here's a typical example:

```
0: kd> !analyze -v

[...]

MULTIPLE_IRP_COMPLETE_REQUESTS (44)
A driver has requested that an IRP be completed (IoCompleteRequest()), but the packet has already been completed. This is a tough bug to find because the easiest case, a driver actually attempted to complete its own packet twice, is generally not what happened. Rather, two separate drivers each believe that they own the packet, and each attempts to complete it. The first actually works, and the second fails. Tracking down which drivers in the system actually did this is difficult, generally because the trails of the first driver have been covered by the second. However, the driver stack for the current request can be found by examining the DeviceObject fields in each of the stack locations.

Arguments:
Arg1: ffffffa80104aa010, Address of the IRP
Arg2: 0000000000000eaе
Arg3: 0000000000000000
Arg4: 0000000000000000

STACK_TEXT:
fffff880`0e322428 fffff800`01666224 : 0000000`00000044 fffffa80`104aa010 0000000`00000eaе 0000000`0000000 :
nt!KeBugCheckEx
fffff880`0e322430 fffff880`03dd121f : fffffa80`0dc12c50 fffffa80`107750c8 fffffa80`104aa010 fffff880`0e322580 : nt! ?? ::FNODOBFM:: string`+0x3eb3d
fffff880`0e322520 fffff880`03def17f : fffffa80`0dc12c50 fffffa80`104aa010 fffffa80`0cacb610 0000000`0000001 :
DriverA!DriverA::Create+0x3bf
[...]
fffff880`0e322740 fffff800`01972ba4 : fffffa80`0dc129f0 0000000`0000000 fffffa80`0fe7a010 0000000`0000001 :
nt!IopParseDevice+0x5a7
fffff880`0e3228d0 fffff880`01977b7d : fffffa80`0fe7a010 fffff880`0e322a30 fffffa80`00000040 fffffa80`0cae5080 :
nt!ObpLookupObjectName+0x585
fffff880`0e3229d0 fffff880`0197e647 : 0000000`000007ff 0000000`0000003 fffff8a0`05716d01 0000000`0000000 :
nt!ObOpenObjectByName+0x1cd
fffff880`0e322a80 fffff800`01988398 : 0000000`03f3e510 fffff8a0`c010000 fffff8a0`0c26fe50 0000000`03f3e118 :
nt!IopCreateFile+0x2b7
fffff880`0e322b20 fffff800`0167b813 : fffffa80`0e10db30 0000000`0000001 fffffa80`1002b060 fffff800`0198f294 :
nt!NtCreateFile+0x78
fffff880`0e322bb0 0000000`772efc0a : 000007fe`f62c358f 0000000`03f3e1b0 0000000`7719fd72 000007fe`f62c6490 :
nt!KiSystemServiceCopyEnd+0x13
0000000`03f3e068 000007fe`f62c358f : 0000000`03f3e1b0 0000000`7719fd72 000007fe`f62c6490 0000000`0000005 :
ntdll!NtCreateFile+0xa

[...]

0: kd> !irp fffffa80104aa010
Irp is active with 1 stacks 3 is current (= 0xfffffa80104aa170)
No Mdl: No System Buffer: Thread fffffa801002b060: Irp is completed. Pending has been returned
cmd flg cl Device File Completion-Context
[ 0, 0 ] 0 2 fffffa800dc129f0 00000000 00000000-00000000
\Driver\DriverA
Args: 00000000 00000000 00000000 ffffffc00a0006
```

## Driver Device Collection

This pattern can be used to compare the current list of device and driver objects with some saved reference list to find out any changes. This listing can be done by using **!object** command:

```
0: kd> !object \Driver  
[...]
```

```
0: kd> !object \FileSystem  
[...]
```

```
0: kd> !object \Device  
[...]
```

Note that the collection is called **Driver Device** and not Device Driver.

## Dry Weight

Sometimes what looks like a memory leak when we install a new product version is not really a leak. With the previous version, we had 400 MB typical memory usage, but suddenly we get twice as more. We should not panic but collect a process memory dump to inspect it calmly offline. We may see **Dry Weight** increase: the size of all module images. For some products, the new release may mean complete redesign with a new more powerful framework or incorporation of the significant number of new 3rd-party components (**Module Variety**, page 703). Additional sign against the memory leak hypothesis is simultaneous memory usage increase for many product processes. Although, this may be some shared module with leaking code. For example, in the example below 50% of all committed memory was image memory:

```
0:000> !address -summary

--- Usage Summary ----- RgnCount ----- Total Size ----- %ofBusy %ofTotal
[...]
Image                      1806      0`19031000 ( 402.535 Mb)  4.29%  0.00%
Heap                         72        0`02865000 (   40.395 Mb)  0.44%  0.00%
[...]

--- Type Summary (for busy) ----- RgnCount ----- Total Size ----- %ofBusy %ofTotal
[...]
MEM_IMAGE                  2281      0`19AA8000 ( 413.000 Mb)  4.40%  0.00%
[...]

--- State Summary ----- RgnCount ----- Total Size ----- %ofBusy %ofTotal
[...]
MEM_COMMIT                 2477      0`326e8000 ( 806.906 Mb)  8.76%  0.00%
[...]
```

WinDbg **!mt** command shows almost 50 new .NET components.

## Dual Stack Trace

This is the kernel mode and space counterpart to a user mode and space stack trace and vice versa, for example:

```

25 Id: e8c.f20 Suspend: 1 Teb: 7ff9c000 Unfrozen
ChildEBP RetAddr
086acac4 7c90df5a nt!KiFastSystemCallRet
086acac8 7c8025db nt!ZwWaitForSingleObject+0xc
086acb2c 7c802542 kernel32!WaitForSingleObjectEx+0xa8
086acb40 00fbba3a kernel32!WaitForSingleObject+0x12
WARNING: Stack unwind information not available. Following frames may be wrong.
086ad3c8 00fbcb139 ModuleA!DllCanUnloadNow+0x638b4a
[...]
086afffb4 7c80b729 ModuleA!DllCanUnloadNow+0xc65c0
086affec 00000000 kernel32!BaseThreadStart+0x37

0: kd> !thread 88ec9020 1f
THREAD 88ec9020 Cid 17a0.2034 Teb: 7ffad000 Win32Thread: bc28c6e8 WAIT: (Unknown) UserMode Non-Alertable
89095f48 Semaphore Limit 0x10000
IRP List:
 89a5a370: (0006,0094) Flags: 00000900 Mdl: 00000000
Not impersonating
DeviceMap          d6c30c48
Owning Process    88ffffd88      Image: iexplore.exe
Attached Process   N/A           Image: N/A
Wait Start TickCount 5632994 Ticks: 2980 (0:00:00:46.562)
Context Switch Count 2269        LargeStack
UserTime           00:00:00.000
KernelTime         00:00:00.000
Win32 Start Address 0x00a262d0
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init b204c000 Current b204bc60 Base b204c000 Limit b2048000 Call 0
Priority 8 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr
b204bc78 80833ec5 nt!KiSwapContext+0x26
b204bca4 80829c14 nt!KiSwapThread+0x2e5
b204bcec 8093b174 nt!KeWaitForSingleObject+0x346
b204bd50 8088b41c nt!NtWaitForSingleObject+0x9a
b204bd50 7c82860c nt!KiFastCallEntry+0xfc (TrapFrame @ b204bd64)
058fcabc 7c827d29 nt!KiFastSystemCallRet
058fcac0 77e61d1e nt!ZwWaitForSingleObject+0xc
058fcb30 77e61c8d kernel32!WaitForSingleObjectEx+0xac
058fcb44 00f98b4a kernel32!WaitForSingleObject+0x12
WARNING: Stack unwind information not available. Following frames may be wrong.
058fd3cc 00f99249 ModuleA+0x638b4a
[...]
058ffffb8 77e6482f ModuleA+0xc65c0
058ffffec 00000000 kernel32!BaseThreadStart+0x34

```

This pattern is helpful when we have both process user space memory dumps and kernel and complete memory dumps and want to match stack traces of interest between them. See also patterns **Stack Trace** (page 926) and **Stack Trace Collection** (page 943).

## Duplicate Extension

This pattern is **Duplicate Module** (page 287) equivalent for a debugger that uses loaded modules to extend its functionality. For example, in the case of WinDbg, there is a possibility that two different **Version-Specific Extensions** (page 1058) are loaded wreaking havoc on debugging process (Debugger DLL Hell). For example, we loaded a specific version of SOS extension and successfully got a stack trace:

```
0:000> lmv m mscorewks
start end module name
79e70000 7a3ff000 mscorewks (deferred)
Image path:      C:\Windows\Microsoft.NET\Framework\v2.0.50727\mscorewks.dll
Image name:     mscorewks.dll
Timestamp:      Wed Oct 24 08:41:29 2007 (471EF729)
CheckSum:       00597AA8
ImageSize:      0058F000
File version:   2.0.50727.1433
Product version: 2.0.50727.1433
File flags:     0 (Mask 3F)
File OS:        4 Unknown Win32
File type:      2.0 Dll
File date:      00000000.00000000
Translations:   0409.04b0
CompanyName:   Microsoft Corporation
ProductName:    Microsoft® .NET Framework
InternalName:   mscorewks.dll
OriginalFilename: mscorewks.dll
ProductVersion: 2.0.50727.1433
FileVersion:    2.0.50727.1433 (REDBITS.050727-1400)
FileDescription: Microsoft .NET Runtime Common Language Runtime - WorkStation
LegalCopyright: © Microsoft Corporation. All rights reserved.
Comments: Flavor=Retail

0:000> .chain
Extension DLL search Path:
[...]
Extension DLL chain:
dbghelp: image 6.12.0002.633, API 6.1.6, built Mon Feb 01 20:08:26 2010
[path: C:\Program Files (x86)\Debugging Tools for Windows (x86)\dbghelp.dll]
ext: image 6.12.0002.633, API 1.0.0, built Mon Feb 01 20:08:31 2010
[path: C:\Program Files (x86)\Debugging Tools for Windows (x86)\winext\ext.dll]
exts: image 6.12.0002.633, API 1.0.0, built Mon Feb 01 20:08:24 2010
[path: C:\Program Files (x86)\Debugging Tools for Windows (x86)\WINXP\exts.dll]
uext: image 6.12.0002.633, API 1.0.0, built Mon Feb 01 20:08:23 2010
[path: C:\Program Files (x86)\Debugging Tools for Windows (x86)\winext\uext.dll]
ntsdexts: image 6.1.7650.0, API 1.0.0, built Mon Feb 01 20:08:08 2010
[path: C:\Program Files (x86)\Debugging Tools for Windows (x86)\WINXP\ntsdexts.dll]
```

```

0:000> .load .load C:\Frameworks\32-bit\Framework.Updates\Microsoft.NET\Framework\v2.0.50727\sos

0:000> .chain
Extension DLL search Path:
[...]
Extension DLL chain:
C:\Frameworks\32-bit\Framework.Updates\Microsoft.NET\Framework\v2.0.50727\sos: image 2.0.50727.1433, API
1.0.0, built Wed Oct 24 04:41:30 2007
[path: C:\Frameworks\32-bit\Framework.Updates\Microsoft.NET\Framework\v2.0.50727\sos.dll]
dbghelp: image 6.12.0002.633, API 6.1.6, built Mon Feb 01 20:08:26 2010
[path: C:\Program Files (x86)\Debugging Tools for Windows (x86)\dbghelp.dll]
ext: image 6.12.0002.633, API 1.0.0, built Mon Feb 01 20:08:31 2010
[path: C:\Program Files (x86)\Debugging Tools for Windows (x86)\winext\ext.dll]
exts: image 6.12.0002.633, API 1.0.0, built Mon Feb 01 20:08:24 2010
[path: C:\Program Files (x86)\Debugging Tools for Windows (x86)\WINXP\exts.dll]
uext: image 6.12.0002.633, API 1.0.0, built Mon Feb 01 20:08:23 2010
[path: C:\Program Files (x86)\Debugging Tools for Windows (x86)\winext\uext.dll]
ntsdexts: image 6.1.7650.0, API 1.0.0, built Mon Feb 01 20:08:08 2010
[path: C:\Program Files (x86)\Debugging Tools for Windows (x86)\WINXP\ntsdexts.dll]

0:000> !CLRStack
OS Thread Id: 0xdd0 (0)
ESP      EIP
002eeaa8 77c40f34 [InlinedCallFrame: 002eeaa8] System.Windows.Forms.UnsafeNativeMethods.WaitMessage()
002eeaa4 7b08374f System.Windows.Forms.Application+ComponentManager.System.Windows.Forms.
UnsafeNativeMethods.IMsoComponentManager.FPushMessageLoop(Int32, Int32, Int32)
002eef44 7b0831a5 System.Windows.Forms.Application+ThreadContext.RunMessageLoopInner(Int32,
System.Windows.Forms.ApplicationContext)
002eefbc 7b082fe3 System.Windows.Forms.Application+ThreadContext.RunMessageLoop(Int32,
System.Windows.Forms.ApplicationContext)
002eefec 7b0692c2 System.Windows.Forms.Application.Run(System.Windows.Forms.Form)
002eefc 00833264 LINQPad.Program.Run(System.String, Boolean, System.String, Boolean, Boolean,
System.String)
002eec50 008311dc LINQPad.Program.Go(System.String[])
002eedac 00830545 LINQPad.Program.Start(System.String[])
002eede0 00830362 LINQPad.ProgramStarter.Run(System.String[])
002eede8 008300e3 LINQPad.Loader.Main(System.String[])
002ef00c 79e7c74b [GCFrame: 002ef00c]

```

Then we tried the default analysis command `!analyze -v -hang` and continued using SOS commands. Unfortunately, they no longer worked correctly:

```

0:000> !CLRStack
OS Thread Id: 0xdd0 (0)
ESP EIP
002eeaa8 77c40f34 [InlinedCallFrame: 002eeaa8]
002eeaa4 7b08374f
002eef44 7b0831a5
002eefbc 7b082fe3
002eefec 7b0692c2
002eefc 00833264
002eec50 008311dc
002eedac 00830545
002eede0 00830362

```

```
002eede8 008300e3
002ef00c 79e7c74b [GCFrame: 002ef00c]
```

Looking at loaded extensions list we see that an additional wrong version of SOS.DLL was loaded and that one gets all SOS commands:

```
0:000> .chain
Extension DLL search Path:
[...]
Extension DLL chain:
C:\Windows\Microsoft.NET\Framework\v2.0.50727\sos: image 2.0.50727.4963, API 1.0.0, built Thu Jul 07
03:08:08 2011
[path: C:\Windows\Microsoft.NET\Framework\v2.0.50727\sos.dll]
C:\Frameworks\32-bit\Framework.Updates\Microsoft.NET\Framework\v2.0.50727\sos: image 2.0.50727.1433, API
1.0.0, built Wed Oct 24 04:41:30 2007
[path: C:\Frameworks\32-bit\Framework.Updates\Microsoft.NET\Framework\v2.0.50727\sos.dll]
dbghelp: image 6.12.0002.633, API 6.1.6, built Mon Feb 01 20:08:26 2010
[path: C:\Program Files (x86)\Debugging Tools for Windows (x86)\dbghelp.dll]
ext: image 6.12.0002.633, API 1.0.0, built Mon Feb 01 20:08:31 2010
[path: C:\Program Files (x86)\Debugging Tools for Windows (x86)\winext\ext.dll]
exts: image 6.12.0002.633, API 1.0.0, built Mon Feb 01 20:08:24 2010
[path: C:\Program Files (x86)\Debugging Tools for Windows (x86)\WINXP\exts.dll]
uext: image 6.12.0002.633, API 1.0.0, built Mon Feb 01 20:08:23 2010
[path: C:\Program Files (x86)\Debugging Tools for Windows (x86)\winext\uext.dll]
ntsdexts: image 6.1.7650.0, API 1.0.0, built Mon Feb 01 20:08:08 2010
[path: C:\Program Files (x86)\Debugging Tools for Windows (x86)\WINXP\ntsdexts.dll]
```

If we specify the full path to the correct extension we get the right stack trace:

```
0:000> !C:\Frameworks\32-bit\Framework.Updates\Microsoft.NET\Framework\v2.0.50727\sos.CLRStack
OS Thread Id: 0xdd0 (0)
ESP      EIP
002eeaa8 77c40f34 [InlinedCallFrame: 002eeaa8] System.Windows.Forms.UnsafeNativeMethods.WaitMessage()
002eeaa4 7b08374f System.Windows.Forms.Application+ComponentManager.System.Windows.Forms.
UnsafeNativeMethods.IMsoComponentManager.FPushMessageLoop(Int32, Int32, Int32)
002eeb44 7b0831a5 System.Windows.Forms.Application+ThreadContext.RunMessageLoopInner(Int32,
System.Windows.Forms.ApplicationContext)
002eebbc 7b082fe3 System.Windows.Forms.Application+ThreadContext.RunMessageLoop(Int32,
System.Windows.Forms.ApplicationContext)
002eebec 7b0692c2 System.Windows.Forms.Application.Run(System.Windows.Forms.Form)
002eebfc 00833264 LINQPad.Program.Run(System.String, Boolean, System.String, Boolean, Boolean,
System.String)
002eec50 008311dc LINQPad.Program.Go(System.String[])
002eedac 00830545 LINQPad.Program.Start(System.String[])
002eede0 00830362 LINQPad.ProgramStarter.Run(System.String[])
002eede8 008300e3 LINQPad.Loader.Main(System.String[])
002ef00c 79e7c74b [GCFrame: 002ef00c]
```

To avoid confusion we unload the last loaded extension:

```
0:000> .unload C:\Windows\Microsoft.NET\Framework\v2.0.50727\sos
Unloading C:\Windows\Microsoft.NET\Framework\v2.0.50727\sos extension DLL
```

```
0:000> !CLRStack
OS Thread Id: 0xdd0 (0)
ESP      EIP
002eaa8 77c40f34 [InlinedCallFrame: 002eaa8] System.Windows.Forms.UnsafeNativeMethods.WaitMessage()
002eaa4 7b08374f System.Windows.Forms.Application+ComponentManager.System.Windows.Forms.
UnsafeNativeMethods.IMsoComponentManager.FPushMessageLoop(Int32, Int32, Int32)
002eb44 7b0831a5 System.Windows.Forms.Application+ThreadContext.RunMessageLoopInner(Int32,
System.Windows.Forms.ApplicationContext)
002eebbc 7b082fe3 System.Windows.Forms.Application+ThreadContext.RunMessageLoop(Int32,
System.Windows.Forms.ApplicationContext)
002eebec 7b0692c2 System.Windows.Forms.Application.Run(System.Windows.Forms.Form)
002eefbc 00833264 LINQPad.Program.Run(System.String, Boolean, System.String, Boolean, Boolean,
System.String)
002eec50 008311dc LINQPad.Program.Go(System.String[])
002eedac 00830545 LINQPad.Program.Start(System.String[])
002eede0 00830362 LINQPad.ProgramStarter.Run(System.String[])
002eede8 008300e3 LINQPad.Loader.Main(System.String[])
002ef00c 79e7c74b [GCFrame: 002ef00c]
```

## Comments

---

There were a few questions asked:

**Q.** I thought the version of *sos.dll* that resides in the same folder as *mscorwks.dll* is always the correct one. It looks like, from your folder path names, that *mscorwks.dll* was updated, but *sos.dll* was not. If so, how did you get the correct version of *sos.dll*?

**A.** The dump came from another machine where *mscorwks* and *sos* were the same versions, of course. On the analysis machine, we have a different version of the framework installed with a different *sos*. So we have a discrepancy between the version of the *mscorwks* in the dump and *sos* on the analysis machine. We copy the correct version of the framework from the machine the dump came from. Please also check **Version-Specific Extension** pattern (page 1058).

**Q.** Yes, managed debugging requires that the analysis machine uses the same bitness and framework as the “dump” machine. When you say you “copy” the framework, what exactly do you copy? A subset of the binaries or all of them?

**A.** We usually request copy the whole folder just in case if there are any extra dependencies. However, we found out that *mscorwks* or *clr*, *mscordacwks*, and *sos* DLLs are sufficient.

## Duplicated Module

In addition to **Module Variety** (page 703), this is another DLL Hell pattern. Here the same module is loaded at least twice, and we can detect this when we see the module load address appended to its name in the output of **!m** commands (this is done to make the name of the module unique):

```
0:000> !m
start    end      module name
00b20000 0147f000  MSO_b20000
30000000 309a7000  EXCEL
30c90000 31848000  mso
71c20000 71c32000  tsappcmp
745e0000 7489e000  msip
76290000 762ad000  imm32
76b70000 76b7b000  psapi
76f50000 76f63000  secur32
77380000 77411000  user32
77670000 777a9000  ole32
77ba0000 77bfa000  msvcrt
77c00000 77c48000  gdi32
77c50000 77cef000  rpcrt4
77da0000 77df2000  shlwapi
77e40000 77f42000  kernel32
77f50000 77feb000  advapi32
7c800000 7c8c0000  ntdll
```

Usually, this happens when the DLL is loaded from different locations. It can also be exactly the same DLL version. The problems usually surface when there are different DLL versions, and the new code loads the old version of the DLL and uses it. This may result in interface incompatibility issues and ultimately in application fault like an access violation.

In order to provide a dump to play with I created a small toy program called **2DLLS** to model the worst case scenario similar to the one that I encountered in a production environment. The program periodically loads **MyDLL** module to call one of its functions. Unfortunately, in one place, it uses hard-coded relative path:

```
HMODULE hLib = LoadLibrary(L".\\DLL\\MyDLL.dll");
```

and in another place, it relies on DLL search order<sup>46</sup>:

```
hLib = LoadLibrary(L".\\MyDLL.dll");
```

PATH variable directories would be used for search if this DLL was not found in other locations specified by DLL search order. We see that the problem can happen when another application is installed which uses the old version of that DLL and modifies the PATH variable to point to its location. To model interface incompatibility,

---

<sup>46</sup> <http://msdn.microsoft.com/en-us/library/ms682586.aspx>

we compiled the version of MyDLL that causes NULL pointer access violation when the same function is called from it. The DLL was placed into a separate folder, and the PATH variable was modified to reference that folder:

```
C:\>set PATH=C:\OLD;%PATH%
```

The application crashes and the installed default postmortem debugger<sup>47</sup> saves its crash dump. If we open it, we see that it crashed in *MyDLL\_1e60000* module which should trigger suspicion:

```
0:000> r
rax=0000000001e61010 rbx=0000000000000000 rcx=0000775dcac00000
rdx=0000000000000000 rsi=0000000000000006 rdi=0000000000001770
rip=0000000001e61010 rsp=00000000012fed8 rbp=0000000000000000
r8=0000000000000000 r9=00000000012fd58 r10=0000000000000001
r11=000000000012fcc0 r12=0000000000000000 r13=0000000000000002
r14=0000000000000000 r15=0000000000000000
iopl=0 nv up ei pl nz na pe nc
cs=0033 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010200
MyDLL_1e60000!fnMyDLL:
00000000`01e61010 c70425000000000000000000 mov dword ptr [0],0 ds:00000000`00000000=????????

0:000> kL
Child-SP      RetAddr          Call Site
00000000`00012fed8 00000001`40001093 MyDLL_1e60000!fnMyDLL
00000000`00012fee0 00000001`40001344 2DLLs+0x1093
00000000`00012ff10 00000000`773acdcd 2DLLs+0x1344
00000000`00012ff60 00000000`774fc6e1 kernel32!BaseThreadInitThunk+0xd
00000000`00012ff90 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

Looking at the list of modules we see two versions of MyDLL loaded from two different folders:

```
0:000> lm
start          end            module name
00000000`01e60000 00000000`01e71000  MyDLL_1e60000
00000000`772a0000 00000000`7736a000  user32
00000000`77370000 00000000`774a1000  kernel32
00000000`774b0000 00000000`7762a000  ntdll
00000001`40000000 00000001`40010000  2DLLs
00000001`80000000 00000001`80011000  MyDLL
000007fe`fc9e0000 000007fe`fca32000  uxtheme
000007fe`fe870000 000007fe`fe9a9000  rpcrt4
000007fe`fe9b0000 000007fe`fe9bc000  lpk
000007fe`fea10000 000007fe`feaee8000 oleaut32
000007fe`fec00000 000007fe`fed6a000  usp10
000007fe`fed00000 000007fe`fefb0000  ole32
000007fe`fefb0000 000007fe`ff0af000  advapi32
000007fe`ff0d0000 000007fe`ff131000  gdi32
000007fe`ff2e0000 000007fe`ff381000  msvcrt
000007fe`ff390000 000007fe`ff3b8000  imm32
000007fe`ff4b0000 000007fe`ff5b4000  msctf
```

<sup>47</sup> Custom Postmortem Debuggers in Vista, Memory Dump Analysis Anthology, Volume 1, page 618

```
0:000> lmv m MyDLL_1e60000
start           end             module name
00000000`01e60000 00000000`01e71000  MyDLL_1e60000
    Loaded symbol image file: MyDLL.dll
    Image path: C:\OLD\MyDLL.dll
    Image name: MyDLL.dll
    Timestamp:      Wed Jun 18 14:49:13 2008 (48591259)
    ...
0:000> lmv m MyDLL
start           end             module name
00000001`80000000 00000001`80011000  MyDLL
    Image path: C:\2DLLs\DLL\MyDLL.dll
    Image name: MyDLL.dll
    Timestamp:      Wed Jun 18 14:50:56 2008 (485912C0)
    ...

```

We can also see that the old version of MyDLL was the last loaded DLL:

```
0:000> !dlls -l

0x002c2680: C:\2DLLs\2DLLs.exe
    Base   0x1400000000  EntryPoint  0x1400013b0  Size       0x00010000
    Flags  0x00004000  LoadCount   0x0000ffff  TlsIndex  0x00000000
        LDRP_ENTRY_PROCESSED

    ...
0x002ea9b0: C:\2DLLs\DLL\MyDLL.dll
    Base   0x1800000000  EntryPoint  0x1800013d0  Size       0x00011000
    Flags  0x00084004  LoadCount   0x00000001  TlsIndex  0x00000000
        LDRP_IMAGE_DLL
        LDRP_ENTRY_PROCESSED
        LDRP_PROCESS_ATTACH_CALLED

    ...
0x002ec430: C:\OLD\MyDLL.dll
    Base   0x01e60000  EntryPoint  0x01e613e0  Size       0x00011000
    Flags  0x00284004  LoadCount   0x00000001  TlsIndex  0x00000000
        LDRP_IMAGE_DLL
        LDRP_ENTRY_PROCESSED
        LDRP_PROCESS_ATTACH_CALLED
        LDRP_IMAGE_NOT_AT_BASE

```

We can also see that the PATH variable points to its location, and this might explain why it was loaded:

```
0:000> !peb
PEB at 000007fffffd6000
...
Path=C:\OLD;C:\Windows\system32;C:\Windows;...
...
```

We might think that the module having an address in its name was loaded the last, but this is not true. If we save another copy of the dump from the existing one using **.dump** command and load the new dump file we would see that order of the module names is reversed:

```
0:000> kL
Child-SP          RetAddr          Call Site
00000000`0012fed8 00000001`40001093 MyDLL!fnMyDLL
00000000`0012fee0 00000001`40001344 2DLLs+0x1093
00000000`0012ff10 00000000`773acdcd 2DLLs+0x1344
00000000`0012ff60 00000000`774fc6e1 kernel32!BaseThreadInitThunk+0xd
00000000`0012ff90 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

0:000> lm
start          end            module name
00000000`01e60000 00000000`01e71000  MyDLL
00000000`772a0000 00000000`7736a000  user32
00000000`77370000 00000000`774a1000  kernel32
00000000`774b0000 00000000`7762a000  ntdll
00000001`40000000 00000001`40010000  2DLLs
00000001`80000000 00000001`80011000  MyDLL_180000000
000007fe`fc9e0000 000007fe`fca32000  uxtheme
000007fe`fe870000 000007fe`fe9a9000  rpcrt4
000007fe`fe9b0000 000007fe`fe9bc000  lpk
000007fe`fea10000 000007fe`feae8000  oleaut32
000007fe`fecd0000 000007fe`fed6a000  usp10
000007fe`fedd0000 000007fe`fefb0000  ole32
000007fe`fefb0000 000007fe`ff0af000  advapi32
000007fe`ff0d0000 000007fe`ff131000  gdi32
000007fe`ff2e0000 000007fe`ff381000  msvcrt
000007fe`ff390000 000007fe`ff3b8000  imm32
000007fe`ff4b0000 000007fe`ff5b4000  msctf

0:000> !dlls -1

...
0x002ec430: C:\OLD\MyDLL.dll
    Base   0x01e60000  EntryPoint  0x01e613e0  Size        0x00011000
    Flags  0x00284004  LoadCount   0x00000001  TlsIndex   0x00000000
        LDRP_IMAGE_DLL
        LDRP_ENTRY_PROCESSED
        LDRP_PROCESS_ATTACH_CALLED
        LDRP_IMAGE_NOT_AT_BASE
```

The post-processed dump file used for this example can be downloaded from FTP to play with<sup>48</sup>.

<sup>48</sup> <ftp://dumpanalysis.org/pub/CDAPatternDuplicatedModule.zip>

## Comments

There were a few questions:

**Q.** We can also see that the old version of *MyDLL* was the last loaded DLL. That statement contradicts the following output where it's clear that the old DLL was loaded first:

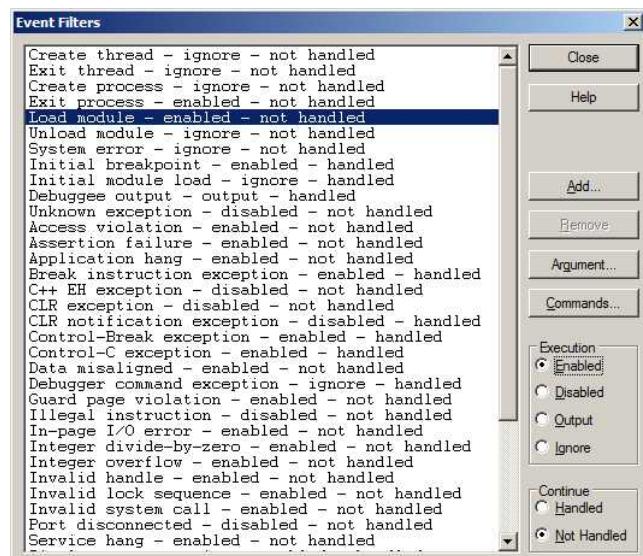
```
0:000> lmv m MyDLL_1e60000
start end module name
00000000 01e60000 00000000`01e71000 MyDLL_1e60000
Loaded symbol image file: MyDLL.dll
Image path: C:\OLD\MyDLL.dll
Image name: MyDLL.dll
Timestamp: Wed Jun 18 14:49:13 2008 (48591259)
[...]

0:000> lmv m MyDLL
start end module name
00000001`80000000 00000001`80011000 MyDLL
Image path: C:\2DLLs\DLL\MyDLL.dll
Image name: MyDLL.dll
Timestamp: Wed Jun 18 14:50:56 2008 (485912C0)
[...]
```

**A.** We don't see the contradiction here: *Timestamp* is for the link time not the load time, and the loader is free to choose the base address.

**Q.** From the list of modules we are able to see two modules are duplicated. Is there any way to find out, who called the module to load into memory. We would like to see the calling function (the originator of duplication).

**A.** We suggest searching for module start address on thread raw stacks. Also, consider live debugging with enabled load modules event handling in WinDbg.



## Dynamic Memory Corruption

### Kernel Pool

If kernel pools are corrupt then calls that allocate or free memory result in bugchecks C2 or 19 and in other fewer frequent bugchecks (from Google stats<sup>49</sup>):

BugCheck C2: BAD_POOL_CALLER	1600
BugCheck 19: BAD_POOL_HEADER	434
BugCheck C5: DRIVER_CORRUPTED_EXPOOL	207
BugCheck DE: POOL_CORRUPTION_IN_FILE_AREA	106
BugCheck D0: DRIVER_CORRUPTED_MMPOOL	8
BugCheck D6: DRIVER_PAGE_FAULT_BEYOND_END_OF_ALLOCATION	3
BugCheck CD: PAGE_FAULT_BEYOND_END_OF_ALLOCATION	2
BugCheck C6: DRIVER_CAUGHT MODIFYING_FREED_POOL	0

Bugchecks 0xC2 and 0x19 have parameters in their arguments that tell the type of detected pool corruption. We should refer to WinDbg help for details or use the variant of **!analyze** command where we can supply optional bugcheck arguments:

```
1: kd> !analyze -show c2
BAD_POOL_CALLER (c2)
The current thread is making a bad pool request. Typically this is at a bad IRQL level or double freeing
the same allocation, etc.
Arguments:
Arg1: 00000000, The caller is requesting a zero byte pool allocation.
Arg2: 00000000, zero.
Arg3: 00000000, the pool type being allocated.
Arg4: 00000000, the pool tag being used.
```

---

<sup>49</sup> Bugcheck Frequencies, Memory Dump Analysis Anthology, Volume 2, page 429

```

1: kd> !analyze -show 19 2 1 1 1
BAD_POOL_HEADER (19)
The pool is already corrupt at the time of the current request.
This may or may not be due to the caller.
The internal pool links must be walked to figure out a possible cause of
the problem, and then special pool applied to the suspect tags or the driver
verifier to a suspect driver.
Arguments:
Arg1: 00000002, the verifier pool pattern check failed. The owner has likely corrupted the pool block
Arg2: 00000001, the pool entry being checked.
Arg3: 00000001, size of the block.
Arg4: 00000001, 0.

```

If we enable special pool on suspected drivers, we may get these bugchecks too with the following Google frequency:

BugCheck C1: SPECIAL_POOL_DETECTED_MEMORY_CORRUPTION	59
BugCheck D5: DRIVER_PAGE_FAULT_IN_FREED_SPECIAL_POOL	5
BugCheck CC: PAGE_FAULT_IN_FREED_SPECIAL_POOL	1

Here is one example of nonpaged pool corruption detected during *free* operation with the following **!analyze -v** output:

```

BAD_POOL_HEADER (19)
The pool is already corrupt at the time of the current request.
This may or may not be due to the caller.
The internal pool links must be walked to figure out a possible cause of
the problem, and then special pool applied to the suspect tags or the driver
verifier to a suspect driver.
Arguments:
Arg1: 00000020, a pool block header size is corrupt.
Arg2: a34583b8, The pool entry we were looking for within the page.
Arg3: a34584f0, The next pool entry.
Arg4: 0a270001, (reserved)

POOL_ADDRESS: a34583b8 Nonpaged pool

PROCESS_NAME: process.exe

CURRENT_IRQL: 2

STACK_TEXT:
b80a60cc 808927bb nt!KeBugCheckEx+0x1b
b80a6134 80892b6f nt!ExFreePoolWithTag+0x477
b80a6144 b9591400 nt!ExFreePool+0xf
WARNING: Stack unwind information not available. Following frames may be wrong.
b80a615c b957b954 driver+0x38400
b80a617c b957d482 driver+0x22954
b80a61c0 b957abf4 driver+0x24482
b80a6260 b957cceff driver+0x21bf4
b80a62a8 8081df65 driver+0x23cef
b80a62bc f721ac45 nt!IoCallDriver+0x45
b80a62e4 8081df65 fltp!FltpDispatch+0x6f

```

```
b80a62f8 b99de70b nt!IoCallDriver+0x45
b80a6308 b99da6ee filter!Dispatch+0xfb
b80a6318 8081df65 filter!dispatch+0x6e
b80a632c b9bddebfe nt!IoCallDriver+0x45
b80a6334 8081df65 2ndfilter!Redirect+0x7ea
b80a6348 b9bd1756 nt!IoCallDriver+0x45
b80a6374 b9bd1860 3rdfilter!PassThrough+0x136
b80a6384 8081df65 3rdfilter!Dispatch+0x80
b80a6398 808f5437 nt!IoCallDriver+0x45
b80a63ac 808ef963 nt!IoSyncrhousServiceTail+0x10b
b80a63d0 8088978c nt!NtQueryDirectoryFile+0x5d
b80a63d0 7c8285ec nt!KiFastCallEntry+0xfc
00139524 7c8274eb ntdll!KiFastSystemCallRet
00139528 77e6ba40 ntdll!NtQueryDirectoryFile+0xc
00139830 77e6bb5f kernel32!FindFirstFileExW+0x3d5
00139850 6002665e kernel32!FindFirstFileW+0x16
00139e74 60026363 process+0x2665e
0013a328 60027852 process+0x26363
0013a33c 60035b58 process+0x27852
0013b104 600385ff process+0x35b58
0013b224 612cb643 process+0x385ff
0013b988 612cc109 d11!FileDialog+0xc53
0013bba0 612cb47b d11!FileDialog+0x1719
0013c2c0 7739b6e3 d11!FileDialog+0xa8b
0013c2ec 77395f82 USER32!InternalCallWinProc+0x28
0013c368 77395e22 USER32!UserCallDlgProcCheckWow+0x147
0013c3b0 7739c9c6 USER32!DefDlgProcWorker+0xa8
0013c3d8 7c828536 USER32!__fnDWORD+0x24
0013c3d8 808308f4 ntdll!KiUserCallbackDispatcher+0x2e
b80a66b8 8091d6d1 nt!KiCallUserMode+0x4
b80a6710 bf8a2622 nt!KeUserModeCallback+0x8f
b80a6794 bf8a2517 win32k!SfnDWORD+0xb4
b80a67dc bf8a13d9 win32k!xxxSendMessageToClient+0x133
b80a6828 bf85ae67 win32k!xxxSendMessageTimeout+0x1a6
b80a684c bf8847a1 win32k!xxxWrapSendMessage+0x1b
b80a6868 bf8c1459 win32k!NtUserfnNCDESTROY+0x27
b80a68a0 8088978c win32k!NtUserMessageCall+0xc0
b80a68a0 7c8285ec nt!KiFastCallEntry+0xfc
0013c3d8 7c828536 ntdll!KiFastSystemCallRet
0013c3d8 808308f4 ntdll!KiUserCallbackDispatcher+0x2e
b80a6b7c 8091d6d1 nt!KiCallUserMode+0x4
b80a6bd4 bf8a2622 nt!KeUserModeCallback+0x8f
b80a6c58 bf8a23a0 win32k!SfnDWORD+0xb4
b80a6ca0 bf8a13d9 win32k!xxxSendMessageToClient+0x118
b80a6cec bf85ae67 win32k!xxxSendMessageTimeout+0x1a6
b80a6d10 bf8c148c win32k!xxxWrapSendMessage+0x1b
b80a6d40 8088978c win32k!NtUserMessageCall+0x9d
b80a6d40 7c8285ec nt!KiFastCallEntry+0xfc
0013f474 7c828536 ntdll!KiFastSystemCallRet
0013f4a0 7739d1ec ntdll!KiUserCallbackDispatcher+0x2e
0013f4dc 7738cf29 USER32!NtUserMessageCall+0xc
0013f4fc 612d3276 USER32!SendMessageA+0x7f
0013f63c 611add41 d11!SubWindow+0x3dc6
0013f658 7739b6e3 d11!SetWindowText+0x37a1
0013f684 7739b874 USER32!InternalCallWinProc+0x28
0013f6fc 7739ba92 USER32!UserCallWinProcCheckWow+0x151
```

```
0013f764 7739bad0 USER32!DispatchMessageWorker+0x327
0013f774 61221ca8 USER32!DispatchMessageW+0xf
0013f7e0 0040156d dll!MainLoop+0x2c8
0013ff24 00401dfa process+0x156d
0013ffc0 77e6f23b process+0x1dfa
0013fff0 00000000 kernel32!BaseProcessStart+0x23
```

MODULE\_NAME: driver

IMAGE\_NAME: driver.sys

We see that WinDbg pointed to *driver.sys* by using a procedure described in Component Identification article<sup>50</sup>.

Any OS component could corrupt the pool prior to the detection as the bugcheck description says: "The pool is already corrupt at the time of the current request.". What other evidence can reinforce our belief in *driver.sys*? Let's look at our pool entry tag first:

```
1: kd> !pool a34583b8
Pool page a34583b8 region is Nonpaged pool
a3458000 size: 270 previous size: 0 (Allocated) Thre (Protected)
a3458270 size: 10 previous size: 270 (Free) RxIr
a3458280 size: 40 previous size: 10 (Allocated) Vadl
a34582c0 size: 98 previous size: 40 (Allocated) File (Protected)
a3458358 size: 8 previous size: 98 (Free) Vadl
a3458360 size: 50 previous size: 8 (Allocated) Gsem
a34583b0 size: 8 previous size: 50 (Free) CcSc
*a34583b8 size: 138 previous size: 8 (Allocated) *DRIV
Owning component : Unknown (update pooltag.txt)
a34584f0 is not a valid large pool allocation, checking large session pool...
a34584f0 is freed (or corrupt) pool
Bad allocation size @a34584f0, zero is invalid

*** 
*** An error (or corruption) in the pool was detected;
*** Attempting to diagnose the problem.
*** 
*** Use !poolval a3458000 for more details.
*** 

Pool page [ a3458000 ] is __inVALID.

Analyzing linked list...
[ a34583b8 --> a34583d8 (size = 0x20 bytes)]: Corrupt region
[ a34583f8 --> a34585e8 (size = 0x1f0 bytes)]: Corrupt region

Scanning for single bit errors...

None found
```

---

<sup>50</sup> Component Identification, Memory Dump Analysis Anthology, Volume 1, page 46

We see that the tag is DRIV, and we know either from the association or from similar problems in the past that it belongs to *driver.sys*. Let's dump our pool entry contents to see if there are any symbolic hints in it:

```
1: kd> dps a34583b8
a34583b8 0a270001
a34583bc 5346574e
a34583c0 00000000
a34583c4 00000000
a34583c8 b958f532 driver+0x36532
a34583cc a3471010
a34583d0 0000012e
a34583d4 00000001
a34583d8 00041457
a34583dc 05af0026
a34583e0 00068002
a34583e4 7b9ec6f5
a34583e8 ffffff00
a34583ec 73650cff
a34583f0 7461445c
a34583f4 97a10061
a34583f8 ff340004
a34583fc c437862a
a3458400 6a000394
a3458404 00000038
a3458408 00000000
a345840c bf000000
a3458410 bf0741b5
a3458414 f70741b5
a3458418 00000000
a345841c 00000000
a3458420 00000000
a3458424 00000000
a3458428 05000000
a345842c 34303220
a3458430 31323332
a3458434 ff322d36
```

Indeed we see that the possible code pointer *driver+0x36532* and the code around this address look normal:

```
3: kd> .asm no_code_bytes
Assembly options: no_code_bytes

3: kd> u b958f532
driver+0x36532:
b958f532 push    2Ch
b958f534 push    offset driver+0x68d08 (b95c1d08)
b958f539 call    driver+0x65c50 (b95bec50)
b958f53e mov     byte ptr [ebp-19h],0
b958f542 and    dword ptr [ebp-24h],0
b958f546 call    dword ptr [driver+0x65f5c (b95bef5c)]
b958f54c mov     ecx,dword ptr [ebp+0Ch]
b958f54f cmp    eax,ecx
```

```
3: kd> ub b958f532
driver+0x36528:
b958f528 leave
b958f529 ret     18h
b958f52c int     3
b958f52d int     3
b958f52e int     3
b958f52f int     3
b958f530 int     3
b958f531 int     3
```

## Comments

---

Here is one of the asked questions.

**Q.** I have the same problem with BAD\_POOL\_HEADER; I see the pool header e173c1d8 is corrupt and marked as \*Ppen. Could you help me to know what is that?

```
BAD_POOL_HEADER (19)
The pool is already corrupt at the time of the current request.
This may or may not be due to the caller.
The internal pool links must be walked to figure out a possible cause of
the problem, and then special pool applied to the suspect tags or the driver
verifier to a suspect driver.
Arguments:
Arg1: 00000020, a pool block header size is corrupt.
Arg2: e173c1d8, The pool entry we were looking for within the page.
Arg3: e173c1f8, The next pool entry.
Arg4: 0c040404, (reserved)
```

### Debugging Details:

---

BUGCHECK\_STR: 0x19\_20

POOL\_ADDRESS: e173c1d8

CUSTOMER\_CRASH\_COUNT: 6

DEFAULT\_BUCKET\_ID: COMMON\_SYSTEM\_FAULT

PROCESS\_NAME: System

LOCK\_ADDRESS: 8055b4e0 - (!locks 8055b4e0)

Resource @ nt!PiEngineLock (0x8055b4e0) Available

WARNING: SystemResourcesList->Flink chain invalid. Resource may be corrupted, or already deleted.

WARNING: SystemResourcesList->Blink chain invalid. Resource may be corrupted, or already deleted.

1 total locks

PNP\_TRIAGE:

Lock address : 0x8055b4e0  
Thread Count : 0  
Thread address: 0x00000000  
Thread wait : 0x0

LAST\_CONTROL\_TRANSFER: from 8054b583 to 804f9f33

STACK\_TEXT:

f79f392c 8054b583 00000019 00000020 e173c1d8 nt!KeBugCheckEx+0x1b  
f79f397c 8058fc05 e173c1e0 00000000 00000000 nt!ExFreePoolWithTag+0x2a3  
f79f39dc 8059161d 85804dd0 e10f29a8 f79f3a48 nt!PipMakeGloballyUniqueId+0x3a9  
f79f3ad0 8059222b 8593a7c8 861d33e8 8593a7c8 nt!PipProcessNewDeviceNode+0x185  
f79f3d24 805927fa 8593a7c8 00000001 00000000 nt!PipProcessDevNodeTree+0x16b  
f79f3d54 804f698e 00000003 8055b5c0 8056485c nt!PiRestartDevice+0x80  
f79f3d7c 8053876d 00000000 00000000 863c1da8 nt!PipDeviceActionWorker+0x168  
f79f3dac 805cff64 00000000 00000000 00000000 nt!ExpWorkerThread+0xef  
f79f3ddc 805460de 8053867e 00000001 00000000 nt!PspSystemThreadStartup+0x34  
00000000 00000000 00000000 00000000 nt!KiThreadStartup+0x16

STACK\_COMMAND: kb

FOLLOWUP\_IP:

nt!ExFreePoolWithTag+2a3  
8054b583 8b45f8 mov eax,dword ptr [ebp-8]

SYMBOL\_STACK\_INDEX: 1

SYMBOL\_NAME: nt!ExFreePoolWithTag+2a3

FOLLOWUP\_NAME: MachineOwner

MODULE\_NAME: nt

IMAGE\_NAME: ntkrpamp.exe

DEBUG\_FLR\_IMAGE\_TIMESTAMP: 4802516a

FAILURE\_BUCKET\_ID: 0x19\_20\_nt!ExFreePoolWithTag+2a3

BUCKET\_ID: 0x19\_20\_nt!ExFreePoolWithTag+2a3

Followup: MachineOwner

```
0: kd> !pool e173c1d8
Pool page e173c1d8 region is Unknown
e173c000 size: 28 previous size: 0 (Allocated) CMVa
e173c028 size: 8 previous size: 28 (Free) 0...
e173c030 size: 10 previous size: 8 (Allocated) MmSt
e173c040 size: 48 previous size: 10 (Allocated) ScSh
e173c088 size: 68 previous size: 48 (Allocated) ScNc
e173c0f0 size: 8 previous size: 68 (Free) Ntf0
e173c0f8 size: 10 previous size: 8 (Allocated) ObDi
e173c108 size: 28 previous size: 10 (Allocated) CMVa
e173c130 size: 80 previous size: 28 (Allocated) IoNm
e173c1b0 size: 8 previous size: 80 (Free) Sect
e173c1b8 size: 20 previous size: 8 (Allocated) CMVa
*e173c1d8 size: 20 previous size: 20 (Allocated) *Ppen
Pooltag Ppen : routines to perform device enumeration, Binary : nt!pnp
GetULONGFromAddress: unable to read from 80565d50
e173c1f8 is not a valid small pool allocation, checking large pool...
unable to get pool big page table - either wrong symbols or pool tagging is disabled
e173c1f8 is freed (or corrupt) pool
Bad previous allocation size @e173c1f8, last size was 4
```

```
***  
*** An error (or corruption) in the pool was detected;  
*** Pool Region unknown (0xFFFFFFFFE173C1F8)  
***  
*** Use !poolval e173c000 for more details.  
***
```

```
0: kd> !poolval e173c000
Pool page e173c000 region is Unknown

Validating Pool headers for pool page: e173c000
```

```
Pool page [ e173c000 ] is __inVALID.
```

```
Analyzing linked list...
[ e173c1d8 -> e173c2c0 (size = 0xe8 bytes)]: Corrupt region
```

```
Scanning for single bit errors...
```

```
None found
```

```
0: kd> dps e173c1d8
e173c1d8 0c040404
e173c1dc 6e657050
e173c1e0 00260032
e173c1e4 00370061
e173c1e8 00360035
e173c1ec 00370035
e173c1f0 00260032
e173c1f4 004e0030
e173c1f8 00520054
e173c1fc 004c004f
e173c200 002e0053
```

```
[...]
e173c214 00360032
e173c218 00300030
e173c21c 0c12040a
e173c220 e24e4d43
e173c224 00010001
e173c228 7224c689
e173c22c e17b53a4
e173c230 23230077
e173c234 4448233f
[...]
e173c24c 44264431
e173c250 375f5645
e173c254 26353036
```

#### A. Pooltag Ppen : routines to perform device enumeration, Binary : nt!pnp

Ppen is from PnP manager

If we search for this stack trace frame:

```
PipMakeGloballyUniqueId+0x3a9
```

we find the discussion of a similar BSOD and perhaps a solution<sup>51</sup>.

Also, please note that the content of the address e173c1f8 and the content of nearby addresses are covered by UNICODE Regular Data (page 833). We may want to check these String Hints (page 960).

Pool corruption may also cause access violations in pool management with general bugchecks such as KMODE\_EXCEPTION\_NOT\_HANDLED (1e):

```
0: kd> k
# Child-SP RetAddr Call Site
00 fffff802`d2afe568 fffff802`d103db86 nt!KeBugCheckEx
01 fffff802`d2afe570 fffff802`d0fc652d nt!KiFatalExceptionHandler+0x22
02 fffff802`d2afe5b0 fffff802`d0e83139 nt!RtlpExecuteHandlerForException+0xd
03 fffff802`d2afe5e0 fffff802`d0e815a8 nt!RtlDispatchException+0x429
04 fffff802`d2afece0 fffff802`d0fcbb0c2 nt!KiDispatchException+0x144
05 fffff802`d2aff3c0 fffff802`d0fc957d nt!KiExceptionDispatch+0xc2
06 fffff802`d2aff5a0 fffff802`d10af119 nt!KiGeneralProtectionFault+0xfd
*** ERROR: Module load completed but symbols could not be loaded for vmci.sys
07 fffff802`d2aff730 fffff800`84456159 nt!ExAllocatePoolWithTag+0x7c9
08 fffff802`d2aff810 fffff800`84457131 vmci+0x6159
09 fffff802`d2aff840 fffff800`8445398b vmci+0x7131
0a fffff802`d2aff890 fffff802`d0eec3c0 vmci+0x398b
0b fffff802`d2aff8c0 fffff802`d0eebad9 nt!KiExecuteAllDpcs+0x270
0c fffff802`d2affa10 fffff802`d0fc323a nt!KiRetireDpcList+0xe9
0d fffff802`d2affc60 00000000`00000000 nt!KiIdleLoop+0x5a
```

<sup>51</sup> <http://social.microsoft.com/Forums/en/whssoftware/thread/98b381be-edff-43fb-b1d9-5307e6204733>

## Managed Heap

Here's **Managed Heap** counterpart to **Process Heap** (page 307) and **Kernel Pool** (page 292) **Dynamic Memory Corruption** patterns. It is usually detected by CLR during GC phases. Here is a typical stack from CLR 2 (CLR 4 is similar):

```
0:000> kL
ChildEBP RetAddr
002baae0 779b06a0 ntdll!KiFastSystemCallRet
002baae4 772c77d4 ntdll!NtWaitForSingleObject+0xc
002bab54 772c7742 kernel32!WaitForSingleObjectEx+0xbe
002bab68 7a0c0a43 kernel32!WaitForSingleObject+0x12
002bab98 7a0c0e89 mscorewks!ClrWaitForSingleObject+0x24
002bb054 7a0c2bfd mscorewks!RunWatson+0x1df
002bb798 7a0c3171 mscorewks!DoFaultReportWorker+0xb62
002bb7d4 7a106b2d mscorewks!DoFaultReport+0xc3
002bb7fc 7a1061ac mscorewks!WatsonLastChance+0x43
002bbc8 7a10624f mscorewks!EEPolicy::LogFatalError+0x3ae
002bbcd0 79ffee2f mscorewks!EEPolicy::HandleFatalError+0x36
002bbcf4 79f04f1f mscorewks!CLRVectoredExceptionHandlerPhase3+0xc1
002bbd28 79f04e98 mscorewks!CLRVectoredExceptionHandlerPhase2+0x20
002bbd5c 79f9149e mscorewks!CLRVectoredExceptionHandler+0x10a
002bbd70 779b1039 mscorewks!CLRVectoredExceptionHandlerShimX86+0x27
002bbd94 779b100b ntdll!ExecuteHandler2+0x26
002bbe3c 779b0e97 ntdll!ExecuteHandler+0x24
002bbe3c 79f69360 ntdll!KiUserExceptionDispatcher+0xf
002bc13c 79f663f1 mscorewks!SVR::heap_segment_next_rw+0xf
002bc228 79f65d63 mscorewks!WKS::gc_heap::plan_phase+0x37c
002bc248 79f6614c mscorewks!WKS::gc_heap::gc1+0x6e
002bc25c 79f65f5d mscorewks!WKS::gc_heap::garbage_collect+0x261
002bc288 79f663c2 mscorewks!WKS::GCHeap::GarbageCollectGeneration+0x1a9
002bc314 79ef1566 mscorewks!WKS::gc_heap::try_allocate_more_space+0x12e
002bc328 79ef1801 mscorewks!WKS::gc_heap::allocate_more_space+0x11
002bc348 79e7510e mscorewks!WKS::GCHeap::Alloc+0x3b
002bc364 79e86713 mscorewks!Alloc+0x60
002bc3a0 79e86753 mscorewks!SlowAllocateString+0x29
002bc3ac 79eb4efb mscorewks!UnframedAllocateString+0xc
002bc3e0 79e91f58 mscorewks!AllocateStringObject+0x2e
002bc424 79e82892 mscorewks!GlobalStringLiteralMap::AddStringLiteral+0x3f
002bc438 79e82810 mscorewks!GlobalStringLiteralMap::GetStringLiteral+0x43
002bc47c 79e82956 mscorewks!AppDomainStringLiteralMap::GetStringLiteral+0x72
002bc494 79e81b6f mscorewks!BaseDomain::GetStringObjRefPtrFromUnicodeString+0x31
002bc4cc 79ef4704 mscorewks!Module::ResolveStringRef+0x88
002bc4e4 79f23132 mscorewks!ConstructStringLiteral+0x39
002bc558 7908c351 mscorewks!CEEInfo::constructStringLiteral+0x108
002bc57c 7906276d mscorjit!Compiler::fgMorphConst+0xa3
002bc598 79065ea0 mscorjit!Compiler::fgMorphTree+0x63
002bc610 79062bb5 mscorjit!Compiler::fgMorphArgs+0x86
002bc63c 7906311f mscorjit!Compiler::fgMorphCall+0x2c1
002bc658 79065ea0 mscorjit!Compiler::fgMorphTree+0xa3
002bc6d0 79062bb5 mscorjit!Compiler::fgMorphArgs+0x86
002bc6fc 7906311f mscorjit!Compiler::fgMorphCall+0x2c1
002bc718 790650fa mscorjit!Compiler::fgMorphTree+0xa3
002bc738 79065026 mscorjit!Compiler::fgMorphStmts+0x63
```

```

002bc774 79064f9f mscorjit!Compiler::fgMorphBlocks+0x79
002bc788 79064e63 mscorjit!Compiler::fgMorph+0x60
002bc798 790614e6 mscorjit!Compiler::compCompile+0x5f
002bc7f0 79061236 mscorjit!Compiler::compCompile+0x2df
002bc884 7906118c mscorjit!jitNativeCode+0xb8
002bc8bc 79f0f9cf mscorjit!CILJit::compileMethod+0x3d
002bc928 79f0f945 mscorwks!invokeCompileMethodHelper+0x72
002bc96c 79f0f8da mscorwks!invokeCompileMethod+0x31
002bc9c4 79f0ea33 mscorwks!CallCompileMethodWithSEHWrapper+0x84
002bcd7c 79f0e795 mscorwks!UnsafeJitFunction+0x230
002bce20 79e87f52 mscorwks!MethodDesc::MakeJitWorker+0x1c1
002bce78 79e8809e mscorwks!MethodDesc::DoPrestub+0x486
002bcec8 00330836 mscorwks!PreStubWorker+0xeb
WARNING: Frame IP not in any known module. Following frames may be wrong.
002bcee0 79e7c74b 0x330836
002bcf10 79e7c6cc mscorwks!CallDescrWorker+0x33
002bcf90 79e7c8e1 mscorwks!CallDescrWorkerWithHandler+0xa3
002bd0d0 79e7c783 mscorwks!MethodDesc::CallDescr+0x19c
002bd0ec 79e7c90d mscorwks!MethodDesc::CallTargetWorker+0x1f
002bd100 79e8b983 mscorwks!MethodDescCallSite::Call_RetArgSlot+0x18
002bd1d8 79e8b8e6 mscorwks!MethodTable::RunClassInitWorker+0x8b
002bd260 79e8b7fa mscorwks!MethodTable::RunClassInitEx+0x11e
002bd724 79ebcee6 mscorwks!MethodTable::DoRunClassInitThrowing+0x2f0
002bd79c 79fc49db mscorwks!MethodTable::CheckRunClassInitNT+0x8c
002bd82c 790a2801 mscorwks!CEEInfo::initClass+0x19b
002bddcc 79062cdc mscorjit!Compiler::impExpandInline+0x2aaa
002bde24 79062b7c mscorjit!Compiler::fgMorphCallInline+0xf8
002bde50 7906311f mscorjit!Compiler::fgMorphCall+0x27b
002bde6c 790650fa mscorjit!Compiler::fgMorphTree+0xa3
002bde8c 79065026 mscorjit!Compiler::fgMorphStmts+0x63
002bdec8 79064f9f mscorjit!Compiler::fgMorphBlocks+0x79
002bdedc 79064e63 mscorjit!Compiler::fgMorph+0x60
002bdeec 790614e6 mscorjit!Compiler::compCompile+0x5f
002bdf44 79061236 mscorjit!Compiler::compCompile+0x2df
002bdfd8 7906118c mscorjit!jitNativeCode+0xb8
002be010 79f0f9cf mscorjit!CILJit::compileMethod+0x3d
002be07c 79f0f945 mscorwks!invokeCompileMethodHelper+0x72
002be0c0 79f0f8da mscorwks!invokeCompileMethod+0x31
002be118 79f0ea33 mscorwks!CallCompileMethodWithSEHWrapper+0x84
002be4d0 79f0e795 mscorwks!UnsafeJitFunction+0x230
002be574 79e87f52 mscorwks!MethodDesc::MakeJitWorker+0x1c1
002be5cc 79e8809e mscorwks!MethodDesc::DoPrestub+0x486
002be61c 00330836 mscorwks!PreStubWorker+0xeb
002be634 0570c859 0x330836
002be69c 0595bcc1 0x570c859
002be700 0595b954 0x595bcc1
002be704 099b66e0 0x595b954
002be708 002be728 0x99b66e0
002be70c 09589c90 0x2be728
002be728 099b67b8 0x9589c90
002be72c 00000000 0x99b67b8

```

Usually **!VerifyHeap** SOS WinDbg extension command helps to find the first invalid object on managed heap and shows the last valid one. Sometimes the corruption can deeply affect heap or when a crash happens during traversal GC state might not be valid for analysis:

```
0:000> !VerifyHeap
-verify will only produce output if there are errors in the heap
The garbage collector data structures are not in a valid state for traversal.
It is either in the "plan phase," where objects are being moved around, or
we are at the initialization or shutdown of the gc heap. Commands related to
displaying, finding or traversing objects as well as gc heap segments may not
work properly. !dumpheap and !verifyheap may incorrectly complain of heap
consistency errors.
Error requesting heap segment 80018001
Failed to retrieve segments for gc heap
Unable to build snapshot of the garbage collector state

0:000> !DumpHeap
The garbage collector data structures are not in a valid state for traversal.
It is either in the "plan phase," where objects are being moved around, or
we are at the initialization or shutdown of the gc heap. Commands related to
displaying, finding or traversing objects as well as gc heap segments may not
work properly. !dumpheap and !verifyheap may incorrectly complain of heap
consistency errors.
Error requesting heap segment 80018001
Failed to retrieve segments for gc heap
Unable to build snapshot of the garbage collector state
```

In such cases it is recommended to collect several dumps to catch more consistent heap state:

```
0:000> !VerifyHeap
-verify will only produce output if there are errors in the heap
object 0981f024: does not have valid MT
curr_object: 0981f024
Last good object: 0981f010
```

Then we can use **!DumpObj (!do)** command to check objects and **d\*** WinDbg command variations to inspect raw memory.

## Process Heap

### Linux

This is a Linux variant of **Dynamic Memory Corruption** (process heap) pattern previously described for Mac OS X (page 305) and Windows (page 307) platforms.

The corruption may be internal to heap structures with subsequent memory access violation:

```
(gdb) bt
#0 0x000000000041482e in _int_malloc ()
#1 0x0000000000416d88 in malloc ()
#2 0x00000000004005dc in proc ()
#3 0x00000000004006ee in bar_three ()
#4 0x00000000004006fe in foo_three ()
#5 0x0000000000400716 in thread_three ()
#6 0x0000000000401760 in start_thread (arg=<optimized out>
at pthread_create.c:304
#7 0x0000000000432609 in clone ()
#8 0x0000000000000000 in ?? ()

(gdb) x/i $rip
=> 0x41482e <_int_malloc+622>: mov    %rbx,0x10(%r12)

(gdb) x $r12+0x10
0x21687371: Cannot access memory at address 0x21687371

(gdb) p (char[4])0x21687371
$1 = "qsh!"
```

Or it may be detected with a diagnostic message (similar to double free):

```
(gdb) bt
#0 0x000000000043ef65 in raise ()
#1 0x0000000000409fc0 in abort ()
#2 0x000000000040bf5b in __libc_message ()
#3 0x0000000000412042 in malloc_printerr ()
#4 0x0000000000416c27 in free ()
#5 0x0000000000400586 in proc ()
#6 0x000000000040067e in bar_four ()
#7 0x000000000040068e in foo_four ()
#8 0x00000000004006a6 in thread_four ()
#9 0x00000000004016c0 in start_thread (arg=<optimized out>
at pthread_create.c:304
#10 0x0000000000432589 in clone ()
#11 0x0000000000000000 in ?? ()
```

## Mac OS X

This is a Mac OS X / GDB counterpart to **Dynamic Memory Corruption** (process heap) pattern (page 307):

```
(gdb) bt
#0 0x00007fff8479582a in __kill ()
#1 0x00007fff8e0e0a9c in abort ()
#2 0x00007fff8e1024ac in szone_error ()
#3 0x00007fff8e1024e8 in free_list_checksum_botch ()
#4 0x00007fff8e102a7b in small_free_list_remove_ptr ()
#5 0x00007fff8e106bf7 in szone_free_definite_size ()
#6 0x00007fff8e13f789 in free ()
#7 0x000000010afafe23 in main (argc=1, argv=0x7fff6abaeb08)
```

Here's the source code of the modeling application:

```
int main(int argc, const char * argv[])
{
    char *p1 = (char *) malloc (1024);
    printf("p1 = %p\n", p1);

    char *p2 = (char *) malloc (1024);
    printf("p2 = %p\n", p2);

    char *p3 = (char *) malloc (1024);
    printf("p3 = %p\n", p3);

    char *p4 = (char *) malloc (1024);
    printf("p4 = %p\n", p4);

    char *p5 = (char *) malloc (1024);
    printf("p5 = %p\n", p5);

    char *p6 = (char *) malloc (1024);
    printf("p6 = %p\n", p6);

    char *p7 = (char *) malloc (1024);
    printf("p7 = %p\n", p7);

    free(p6);
    free(p4);
    free(p2);

    printf("Hello Crash!\n");
    strcpy(p2, "Hello Crash!");
    strcpy(p4, "Hello Crash!");
    strcpy(p6, "Hello Crash!");

    p2 = (char *) malloc (512);
    printf("p2 = %p\n", p2);
    p4 = (char *) malloc (1024);
    printf("p4 = %p\n", p4);

    6 = (char *) malloc (512);
```

```
printf("p6 = %p\n", p6);

free (p7);
free (p6);
free (p5);
free (p4);
free (p3);
free (p2);
free (p1);

return 0;
}
```

## Windows

This pattern is ubiquitous, its manifestations are random, and usually crashes happen far away from the original corruption point. In our user mode and space part of exception threads (we should not forget about **Multiple Exceptions** pattern, page 714) you would see something like this:

```
ntdll!RtlpCoalesceFreeBlocks+0x10c
ntdll!RtlFreeHeap+0x142
MSVCRT!free+0xda
componentA!xxx
```

or this stack trace fragment:

```
ntdll!RtlpCoalesceFreeBlocks+0x10c
ntdll!RtlpExtendHeap+0x1c1
ntdll!RtlAllocateHeap+0x3b6
componentA!xxx
```

or any similar variants and we need to know exact component that corrupted the application heap (which usually is not the same as *componentA.dll* we see in the crashed thread stack).

For this **common recurrent problem** we have a **general solution**: enable heap checking. This general solution has many variants applied in a **specific context**:

- parameter value checking for heap functions
- user space software heap checks before or after certain checkpoints (like “malloc”/“new” and/or “free”/“delete” calls): usually implemented by checking various fill patterns, etc.
- hardware/OS supported heap checks (like using guard and nonaccessible pages to trap buffer overruns)

The latter variant is the mostly used according to our experience and mainly due to the fact that usually heap corruptions originate from buffer overflows. And it is easier to rely on instant MMU support than on checking fill patterns. Debugging TV episode 0x26 describes how we can enable full page heap<sup>52</sup>. There is an article on how to check in a user dump that full page heap was enabled<sup>53</sup>.

---

<sup>52</sup> [http://www.debugging.tv/Frames/0x26/DebuggingTV\\_Frame\\_0x26.pdf](http://www.debugging.tv/Frames/0x26/DebuggingTV_Frame_0x26.pdf) (<https://www.youtube.com/watch?v=F4cCxHkJVCQ>)

<sup>53</sup> <http://support.citrix.com/article/CTX105955>

The following Microsoft article discusses various heap related checks and tools: How to use Pageheap.exe in Windows XP, Windows 2000, and Windows Server 2003<sup>54</sup>.

The Windows kernel analog to user mode and space heap corruption is called page and nonpaged pool corruption. If we consider Windows kernel pools as variants of the heap, then exactly the same techniques are applicable there, for example, the so-called special pool enabled by Driver Verifier is implemented by nonaccessible pages. Please refer to the following Microsoft article for further details: How to use the special pool feature to isolate pool damage<sup>55</sup>.

## Comments

**!heap -s -v** WinDbg extension command verifies heap blocks:

```
0:001> !heap -s -v
*****
* HEAP ERROR DETECTED *
*
*****
Details:
Error address: 00740f28
Heap handle: 00740000
Error type heap_failure_multiple_entries_corruption (4)
Last known valid blocks: before - 007409e8, after - 007416b8
Stack trace:
77b6fc76: ntdll!RtlpAnalyzeHeapFailure+0x0000025b
77b29ef1: ntdll!RtlpCoalesceFreeBlocks+0x00000060
77ad2d07: ntdll!RtlpFreeHeap+0x000001f4
77ad2bf2: ntdll!RtlFreeHeap+0x00000142
752914d1: kernel32!HeapFree+0x000000014
010b11f0: Application+0x000011f0
010b1274: Application+0x00001274
010b1310: Application+0x00001310
75293677: kernel32!BaseThreadInitThunk+0x0000000e
77ad9f02: ntdll!__RtlUserThreadStart+0x00000070
77ad9ed5: ntdll!_RtlUserThreadStart+0x0000001b
LFH Key : 0x7c150f40
Termination on corruption : DISABLED
Heap Flags Reserv Commit Virt Free List UCR Virt Lock Fast
(k) (k) (k) (k) length blocks cont. heap
-----
.004c0000 00000002 1024 104 104 2 1 1 0 0 LFH
.ERROR: Block 007416b8 previous size f2 does not match previous block size 44
HEAP 00740000 (Seg 00740000) At 007416b8 Error: invalid block Previous
```

<sup>54</sup> <http://support.microsoft.com/kb/286470>

<sup>55</sup> <http://support.microsoft.com/kb/188831>

```
00740000 00001002 64 12 64 3 2 1 0 0
```

Sometimes we may have a buffer underflow, and full page heap which places allocations at the end of pages will not catch the moment of corruption. Here we need to use backward full page heap:

```
gflags /p /enable ImageFile /full /backwards
```

Debugging TV frames episode 0x26 has full recording for such an example<sup>56</sup>.

The new WinDbg 6.2.9200.20512 **!analyze -v** command detects invalid heap calls in case “heap termination on corruption” is not enabled (by default on legacy 32-bit apps). In the past, it was possible to see that only with the heap verification command such as **!heap -s -v** or via **dps ntdll!RtlpHeapFailureInfo**.

Visual C++ 2012 enables heap termination on corruption by default even for 32-bit targets according to [SDL guidelines<sup>57</sup>](#):

```
0:001> !heap -s -v
[...]
Heap address: 00580000
Error address: 005c1a2a
Error type: HEAP_FAILURE_INVALID_ARGUMENT
Details: The caller tried to a free a block at an invalid
(unaligned) address.
Follow-up: Check the error's stack trace to find the culprit.

Stack trace:
7799dff5: ntdll!RtlFreeHeap+0x00000064
767514dd: kernel32!HeapFree+0x00000014
0138140f: AppD8!free+0x0000001a
0138134d: AppD8!StartModeling+0x0000001d
0138121a: AppD8!WndProc+0x0000007a
76f162fa: USER32!InternalCallWinProc+0x00000023
76f16d3a: USER32!UserCallWinProcCheckWow+0x00000109
76f177c4: USER32!DispatchMessageWorker+0x000003bc
76f1788a: USER32!DispatchMessageW+0x0000000f
0138109d: AppD8!wWinMain+0x0000009d
0138152a: AppD8!__tmainCRTStartup+0x000000fd
767533aa: kernel32!BaseThreadInitThunk+0x00000000
77959ef2: ntdll!__RtlUserThreadStart+0x00000070
77959ec5: ntdll!_RtlUserThreadStart+0x0000001b

LFH Key : 0x3d43a3cb
Termination on corruption : DISABLED
```

<sup>56</sup> <http://www.debugging.tv>

<sup>57</sup> <http://msdn.microsoft.com/en-us/library/windows/desktop/cc307399.aspx>

```
0:001> !analyze -v
[...]
BUGCHECK_STR: APPLICATION_FAULT_ACTIONABLE_HEAP_CORRUPTION_heap_failure_invalid_argument
[...]
STACK_TEXT:
77a242a0 7799dff5 ntdll!RtlFreeHeap+0x64
77a242a4 767514dd kernel32!HeapFree+0x14
77a242a8 0138140f appd8!free+0x1a
77a242ac 0138134d appd8!StartModeling+0x1d
77a242b0 0138121a appd8!WndProc+0x7a
77a242b4 76f162fa user32!InternalCallWinProc+0x23
77a242b8 76f16d3a user32!UserCallWinProcCheckWow+0x109
77a242bc 76f177c4 user32!DispatchMessageWorker+0x3bc
77a242c0 76f1788a user32!DispatchMessageW+0xf
77a242c4 0138109d appd8!wWinMain+0x9d
77a242c8 0138152a appd8!_tmainCRTStartup+0xfd
77a242cc 767533aa kernel32!BaseThreadInitThunk+0xe
77a242d0 77959ef2 ntdll!__RtlUserThreadStart+0x70
77a242d4 77959ec5 ntdll!__RtlUserThreadStart+0x1b
[...]

0:001> dps ntdll!RtlpHeapFailureInfo
77a24268 00000000
77a2426c 00000000
77a24270 00000009
77a24274 00580000
77a24278 005c1a2a
77a2427c 00000000
77a24280 00000000
77a24284 00000000
77a24288 00000000
77a2428c 00000000
77a24290 00000000
77a24294 00000000
77a24298 00000000
77a2429c 00000000
77a242a0 7799dff5 ntdll!RtlFreeHeap+0x64
77a242a4 767514dd kernel32!HeapFree+0x14
77a242a8 0138140f AppD8!free+0x1a
77a242ac 0138134d AppD8!StartModeling+0x1d
77a242b0 0138121a AppD8!WndProc+0x7a
77a242b4 76f162fa USER32!InternalCallWinProc+0x23
77a242b8 76f16d3a USER32!UserCallWinProcCheckWow+0x109
77a242bc 76f177c4 USER32!DispatchMessageWorker+0x3bc
77a242c0 76f1788a USER32!DispatchMessageW+0xf
77a242c4 0138109d AppD8!wWinMain+0x9d
77a242c8 0138152a AppD8!_tmainCRTStartup+0xfd
77a242cc 767533aa kernel32!BaseThreadInitThunk+0xe
77a242d0 77959ef2 ntdll!__RtlUserThreadStart+0x70
77a242d4 77959ec5 ntdll!__RtlUserThreadStart+0x1b
77a242d8 00000000
77a242dc 00000000
77a242e0 00000000
77a242e4 00000000

0:001> ~*k
```

```
0 Id: 1d74.fd4 Suspend: 1 Teb: 7efdd000 Unfrozen
ChildEBP RetAddr
001cfa3c 76f1790d USER32!NtUserGetMessage+0x15
001cfa58 0138106f USER32!GetMessageW+0x33
001cfa90 0138152a AppD8!wWinMain+0x6f
001cfadc 767533aa AppD8!__tmainCRTStartup+0xfd
001cfae8 77959ef2 kernel32!BaseThreadInitThunk+0xe
001cfb28 77959ec5 ntdll!__RtlUserThreadStart+0x70
001cfb40 00000000 ntdll!_RtlUserThreadStart+0x1b

# 1 Id: 1d74.e98 Suspend: 1 Teb: 7efda000 Unfrozen
ChildEBP RetAddr
0118fbdb4 779bf896 ntdll!DbgBreakPoint
0118fbe4 767533aa ntdll!DbgUiRemoteBreakin+0x3c
0118fbf0 77959ef2 kernel32!BaseThreadInitThunk+0xe
0118fc30 77959ec5 ntdll!__RtlUserThreadStart+0x70
0118fc48 00000000 ntdll!_RtlUserThreadStart+0x1b
```

**E**

## Early Crash Dump

Some bugs are fixed using a brute-force approach via putting an exception handler to catch access violations and other exceptions. A long time ago I saw one such “incredible fix” when the image processing application was crashing after approximately N<sup>th</sup> heap free runtime call. To ignore crashes, an SEH handler was put in place, but the application started to crash in different places. Therefore, the additional fix was to skip free calls when approaching N and resume afterward. The application started to crash less frequently.

Here getting **Early Crash Dump** when a first-chance exception happens can help in the component identification before corruption starts spreading across data. Recall that when an access violation happens in a process thread in user mode the system generates the first-chance exception which can be caught by an attached debugger and if there is no such debugger the system tries to find an exception handler and if that exception handler catches and dismisses the exception the thread resumes its normal execution path. If there are no such handlers found the system generates the so-called second-chance exception with the same exception context to notify the attached debugger and if it is not attached a default thread exception handler usually saves a postmortem user dump.

We can get first-chance exception memory dumps with:

- DebugDiag<sup>58</sup>
- ADPlus in crash mode from Debugging Tools for Windows
- Exception Monitor from User Mode Process Dumper package<sup>59</sup>
- ProcDump<sup>60</sup>

Here is an example configuration rule for crashes in one of the previous Debug Diagnostic tool versions for TestDefaultDebugger<sup>61</sup> process (Unconfigured First Chance Exceptions option is set to Full Userdump):

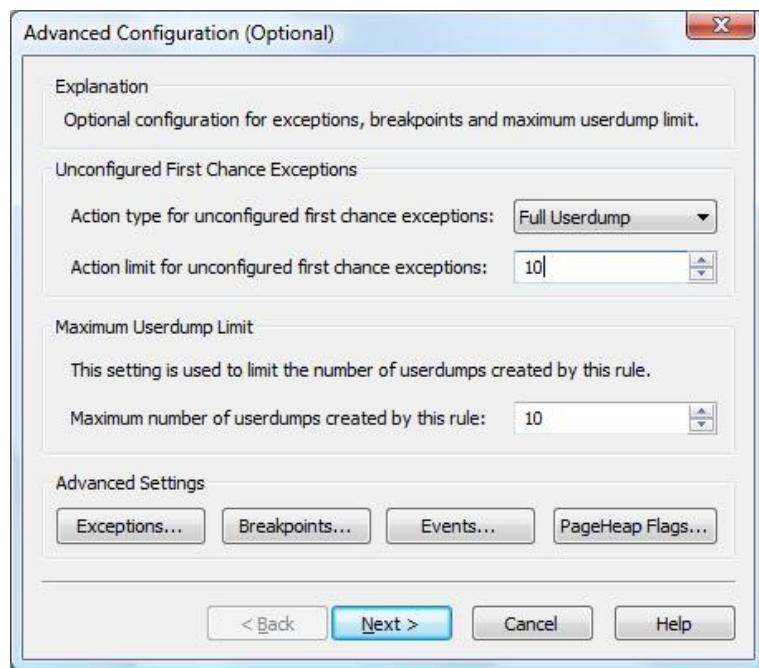
---

<sup>58</sup> <http://blogs.msdn.com/b/debugdiag/archive/2013/10/03/debugdiag-2-0-is-now-rtw.aspx>

<sup>59</sup> <http://www.microsoft.com/downloads/details.aspx?FamilyID=e089ca41-6a87-40c8-bf69-28ac08570b7e&DisplayLang=en>

<sup>60</sup> <https://technet.microsoft.com/en-us/sysinternals/dd996900.aspx>

<sup>61</sup> TestDefaultDebugger, Memory Dump Analysis Anthology, Volume 1, page 641



When we push the big crash button in TestDefaultDebugger dialog box, two crash dumps are saved, with the first and second-chance exceptions pointing to the same code:

```

Loading Dump File [C:\Program Files (x86)\DebugDiag\Logs\Crash rule for all instances of
TestDefaultDebugger.exe\TestDefaultDebugger_PID_4316_Date_11_21_2007_Time_04_28_27PM_2_First
chance exception 0XC0000005.dmp]
User Mini Dump File with Full Memory: Only application data is available

Comment: 'Dump created by DbgHost. First chance exception 0XC0000005'
Symbol search path is: srv*c:\ms*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows Vista Version 6000 MP (2 procs) Free x86 compatible
Product: WinNt, suite: SingleUserTS
Debug session time: Wed Nov 21 16:28:27.000 2007 (GMT+0)
System Uptime: 0 days 23:45:34.711
Process Uptime: 0 days 0:01:09.000

This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
(10dc.590): Access violation - code c0000005 (first/second chance not available)
eax=00000000 ebx=00000001 ecx=0017fe70 edx=00000000 esi=00425ae8 edi=0017fe70
eip=004014f0 esp=0017f898 ebp=0017f8a4 iopl=0 nv up ei ng nz ac pe cy
cs=0023 ss=002b ds=002b es=0053 gs=002b efl=00010297
TestDefaultDebugger!CTestDefaultDebuggerDlg::OnBnClickedButton1:
004014f0 c7050000000000000000000000000000 mov dword ptr ds:[0],0 ds:002b:00000000=?????????

```

```
Loading Dump File [C:\Program Files (x86)\DebugDiag\Logs\Crash rule for all instances of
TestDefaultDebugger.exe\TestDefaultDebugger_PID_4316_Date_11_21_2007_Time_04_28_34PM_693_
Second_Chance_Exception_C0000005.dmp]
User Mini Dump File with Full Memory: Only application data is available
```

```
Comment: 'Dump created by DbgHost. Second_Chance_Exception_C0000005'
Symbol search path is: srv*c:\mss*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows Vista Version 6000 MP (2 procs) Free x86 compatible
Product: WinNt, suite: SingleUserTS
Debug session time: Wed Nov 21 16:28:34.000 2007 (GMT+0)
System Uptime: 0 days 23:45:39.313
Process Uptime: 0 days 0:01:16.000
```

```
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
(10dc.590): Access violation - code c0000005 (first/second chance not available)
eax=00000000 ebx=00000001 ecx=0017fe70 edx=00000000 esi=00425ae8 edi=0017fe70
eip=004014f0 esp=0017f898 ebp=0017f8a4 iopl=0 nv up ei ng nz ac pe cy
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010297
TestDefaultDebugger!CTestDefaultDebuggerDlg::OnBnClickedButton1:
004014f0 c7050000000000000000000000000000 mov dword ptr ds:[0],0  ds:002b:00000000=????????
```

## Effect Component

Some modules like drivers or runtime DLLs are always present after some action has happened. We call them **Effect Components**. It is the last thing to assume them to be the “Cause” components” or “Root Cause” or the so-called “culprit” components. Typical example is dump disk driver symbolic references found in **Execution Residue** (page 371) on the raw stack of a running bugchecking thread:

```
0: kd> !thread
THREAD ffffffa8002bdebb0 Cid 03c4.03f0 Teb: 000007fffffd000 Win32Thread: fffff900c20f9810 RUNNING on
processor 0
IRP List:
  ffffffa8002b986f0: (0006,0118) Flags: 00060000 Mdl: 00000000
Not impersonating
DeviceMap          ffffff88005346920
Owning Process     ffffffa80035bec10      Image:       Application.exe
Attached Process   N/A           Image:       N/A
Wait Start TickCount 35246        Ticks: 7 (0:00:00.00.109)
Context Switch Count 1595         LargeStack
UserTime            00:00:00.000
KernelTime          00:00:00.031
Win32 Start Address Application (0x0000000140002708)
Stack Init ffffffa600495ddb0 Current ffffffa600495d720
Base ffffffa600495e000 Limit ffffffa6004955000 Call 0
Priority 11 BasePriority 8 PriorityDecrement 1 IoPriority 2 PagePriority 5
Child-SP           RetAddr       : Call Site
ffffffa60`0495d558 ffffff800`0186e3ee : nt!KeBugCheckEx
ffffffa60`0495d560 ffffff800`0186d2cb : nt!KiBugCheckDispatch+0x6e
ffffffa60`0495d6a0 ffffffa60`03d5917a : nt!KiPageFault+0x20b (TrapFrame @ ffffffa60`0495d6a0)
[...]

0: kd> dps ffffffa6004955000 ffffffa600495e000
ffffffa60`04955000 00d4d0c8`00d4d0c8
ffffffa60`04955008 00d4d0c8`00d4d0c8
ffffffa60`04955010 00d4d0c8`00d4d0c8
[...]
ffffffa60`0495c7e0 00000000`00000001
ffffffa60`0495c7e8 ffffffa60`02877f6f dump SATA Driver!RecordExecutionHistory+0xcf
ffffffa60`0495c7f0 ffffffa80`024c05a8
ffffffa60`0495c7f8 ffffffa60`02869ad4 dump_dumpata!IdeDumpNotification+0x1a4
ffffffa60`0495c800 ffffffa60`0495cb00
ffffffa60`0495c808 ffffff800`0182ff34 nt!output_l+0x6c0
ffffffa60`0495c810 ffffffa60`02860110 crashdump!StrBeginningDump
ffffffa60`0495c818 ffffffa60`0495cb00
ffffffa60`0495c820 00000000`00000000
ffffffa60`0495c828 ffffffa60`02869b18 dump_dumpata!IdeDumpNotification+0x1e8
ffffffa60`0495c830 00000000`00000000
ffffffa60`0495c838 ffffffa60`0495c8c0
ffffffa60`0495c840 00000000`00000000
ffffffa60`0495c848 ffffffa60`00000024
ffffffa60`0495c850 00000000`ffffffff
ffffffa60`0495c858 00000000`00000000
ffffffa60`0495c860 00000000`00000000
ffffffa60`0495c868 ffffffa60`0495cb00
ffffffa60`0495c870 ffffffa80`00000000
```

```

fffffa60`0495c878 00000000`00000000
fffffa60`0495c880 00000000`00000101
fffffa60`0495c888 fffffa60`02877f6f dump SATA Driver!RecordExecutionHistory+0xcf
fffffa60`0495c890 fffffa60`0495cb0f
fffffa60`0495c898 fffff800`0182ff34 nt!output_l+0x6c0
fffffa60`0495c8a0 fffffa60`0495cb0f
fffffa60`0495c8a8 fffffa60`0495cb90
fffffa60`0495c8b0 00000000`00000040
fffffa60`0495c8b8 fffffa60`02877f6f dump SATA Driver!RecordExecutionHistory+0xcf
fffffa60`0495c8c0 fffff80`024c0728
fffffa60`0495c8c8 fffffa80`024c0728
fffffa60`0495c8d0 00000001`00000000
fffffa60`0495c8d8 fffffa60`00000026
fffffa60`0495c8e0 00000000`ffffffff
fffffa60`0495c8e8 00000000`00000000
fffffa60`0495c8f0 fffffa80`00000000
fffffa60`0495c8f8 fffffa60`0495cb90
fffffa60`0495c900 00000000`00000000
fffffa60`0495c908 fffffa60`02877f6f dump SATA Driver!RecordExecutionHistory+0xcf
fffffa60`0495c910 00000000`00000000
fffffa60`0495c918 fffffa60`02877f6f dump SATA Driver!RecordExecutionHistory+0xcf
fffffa60`0495c920 fffff880`05311010
fffffa60`0495c928 00000000`00000002
fffffa60`0495c930 fffffa60`02875094 dump SATA Driver!AhciAdapterControl
fffffa60`0495c938 fffffa80`024c6018
fffffa60`0495c940 fffffa80`024c0728
fffffa60`0495c948 fffffa60`02877f6f dump SATA Driver!RecordExecutionHistory+0xcf
fffffa60`0495c950 fffffa80`024c0728
fffffa60`0495c958 00000000`00000000
fffffa60`0495c960 fffffa60`0495ca18
fffffa60`0495c968 00000000`00000000
fffffa60`0495c970 fffffa80`024c0728
fffffa60`0495c978 fffffa60`02876427 dump SATA Driver!AhciHwInitialize+0x337
fffffa60`0495c980 fffffa80`024c0be6
fffffa60`0495c988 fffffa60`0286a459 dump_dumpata!IdeDumpWaitOnRequest+0x79
fffffa60`0495c990 00000000`00000000
fffffa60`0495c998 00000000`0000023a
fffffa60`0495c9a0 20474e55`534d4153
fffffa60`0495c9a8 204a4831`36314448
fffffa60`0495c9b0 20202020`20202020
fffffa60`0495c9b8 20202020`20202020
fffffa60`0495c9c0 fffffa80`024c05a8
fffffa60`0495c9c8 fffffa60`02869b18 dump_dumpata!IdeDumpNotification+0x1e8
fffffa60`0495c9d0 00000000`00000000
fffffa60`0495c9d8 fffffa60`0495ca60
fffffa60`0495c9e0 00000000`00000001
fffffa60`0495c9e8 fffffa60`02869396 dump_dumpata!IdeDumpMiniportChannelInitialize+0x236
fffffa60`0495c9f0 fffffa80`024c05a8
fffffa60`0495c9f8 fffffa60`02869ad4 dump_dumpata!IdeDumpNotification+0x1a4
fffffa60`0495ca00 00000000`00000000
fffffa60`0495ca08 fffffa60`0495ca90
fffffa60`0495ca10 00000000`00000001
fffffa60`0495ca18 00000001`00000038
fffffa60`0495ca20 00000000`10010000
fffffa60`0495ca28 00000000`00000003
fffffa60`0495ca30 fffffa80`024c05a8

```

```

fffffa60`0495ca38  fffffa60`0286a954 dump_dumpata!AtaPortGetPhysicalAddress+0x2c
fffffa60`0495ca40  fffffa80`024c0728
fffffa60`0495ca48  fffffa60`02877f6f dump SATA Driver!RecordExecutionHistory+0xcf
fffffa60`0495ca50  00000000`00000001
fffffa60`0495ca58  0000003f`022a8856
fffffa60`0495ca60  fffffa80`0000000c
fffffa60`0495ca68  fffffa80`024c0728
fffffa60`0495ca70  00000000`00000200
fffffa60`0495ca78  fffffa60`02877f6f dump SATA Driver!RecordExecutionHistory+0xcf
fffffa60`0495ca80  fffffa80`024c0728
fffffa60`0495ca88  ffff6226`4f5f3eb8
fffffa60`0495ca90  00000000`00000010
fffffa60`0495ca98  fffffa60`02860370 crashdump!Context+0x30
fffffa60`0495caa0  fffffa80`024c05a8
fffffa60`0495caa8  fffffa60`02875a0d dump SATA Driver!AhciHwStartIo+0x69d
fffffa60`0495cab0  fffffa80`024c0728
fffffa60`0495cab8  00000000`00000000
fffffa60`0495cac0  00000000`00000001
fffffa60`0495cac8  fffff800`018f3dfc nt!DisplayCharacter+0x5c
fffffa60`0495cad0  00000000`00000000
fffffa60`0495cad8  fffffa60`02877f6f dump SATA Driver!RecordExecutionHistory+0xcf
fffffa60`0495cae0  00000000`00010000
fffffa60`0495cae8  00000000`00000000
fffffa60`0495caf0  fffffa60`0495cd10
fffffa60`0495caf8  fffffa60`0495cc00
fffffa60`0495cb00  fffffa80`024c01c0
fffffa60`0495cb08  fffffa60`02875c3f dump SATA Driver!AhciHwInterrupt+0x2b
fffffa60`0495cb10  fffffa80`024c05a8
fffffa60`0495cb18  00000000`00000000
fffffa60`0495cb20  00000000`00000000
fffffa60`0495cb28  fffff800`01d406c9 hal!KeStallExecutionProcessor+0x25
fffffa60`0495cb30  00000000`00010000
fffffa60`0495cb38  00000000`00000000
fffffa60`0495cb40  fffffa60`0495cd10
fffffa60`0495cb48  fffffa60`0495cc00
fffffa60`0495cb50  00000000`00000000
fffffa60`0495cb58  fffffa60`0286a429 dump_dumpata!IdeDumpWaitOnRequest+0x49
fffffa60`0495cb60  fffffa60`02860370 crashdump!Context+0x30
fffffa60`0495cb68  00000000`d8bda325
fffffa60`0495cb70  00000000`00000000
fffffa60`0495cb78  00000000`0000033e
fffffa60`0495cb80  00000000`00000000
fffffa60`0495cb88  fffffa60`028694d2 dump_dumpata!IdeDumpWritePending+0xee
fffffa60`0495cb90  fffffa80`024c0000
fffffa60`0495cb98  fffffa80`024c01c0
fffffa60`0495cba0  00000000`00000000
fffffa60`0495cba8  00000000`00000000
fffffa60`0495cbb0  fffffa80`024c01c0
fffffa60`0495cbb8  fffffa80`01e3c740
fffffa60`0495cbc0  00000000`00010000
fffffa60`0495cbc8  00000000`00000000
fffffa60`0495cbd0  00000000`0c01f000
fffffa60`0495cbd8  fffffa60`0285bca9 crashdump!WritePageSpanToDisk+0x181
fffffa60`0495cbe0  00000000`83d81000
fffffa60`0495cbe8  00000000`00000000
fffffa60`0495cbf0  fffffa60`02860370 crashdump!Context+0x30

```

```

fffffa60`0495cbf8 00000000`00000002
[...]
fffffa60`0495cc0d 00000000`0000c01d
fffffa60`0495ccd8 fffffa60`02860370 crashdump!Context+0x30
fffffa60`0495cce0 00000000`0000bf80
fffffa60`0495cce8 00000000`00000001
fffffa60`0495ccf0 00000000`00000000
fffffa60`0495ccf8 fffffa80`01e353d0
fffffa60`0495cd00 fffffa80`01e353f8
fffffa60`0495cd08 fffffa60`0285bacc crashdump!WriteFullDump+0x70
fffffa60`0495cd10 00000002`3a3d8000
fffffa60`0495cd18 00000000`0000c080
fffffa60`0495cd20 fffffa80`00000000
fffffa60`0495cd28 fffffa60`0285c9c0 crashdump!CrashdumpWriteRoutine
fffffa60`0495cd30 fffff880`05311010
fffffa60`0495cd38 00000000`00000002
fffffa60`0495cd40 fffffa60`0495cf70
fffffa60`0495cd48 00000000`00000000
fffffa60`0495cd50 fffffa60`02860370 crashdump!Context+0x30
fffffa60`0495cd58 fffffa60`0285b835 crashdump!DumpWrite+0xc5
fffffa60`0495cd60 00000000`00000000
fffffa60`0495cd68 00000000`0000000f
fffffa60`0495cd70 00000000`00000001
fffffa60`0495cd78 fffffa60`00000001
fffffa60`0495cd80 fffffa80`02bdeb0
fffffa60`0495cd88 fffffa60`0285b153 crashdump!CrashdumpWrite+0x57
fffffa60`0495cd90 00000000`00000000
fffffa60`0495cd98 fffffa60`028602f0 crashdump!StrInitPortDriver
fffffa60`0495cda0 00000000`00000000
fffffa60`0495cda8 fffffa60`02860a00 crashdump!ContextCopy
fffffa60`0495cdb0 00000000`00000000
fffffa60`0495cdb8 fffff800`01902764 nt!IoWriteCrashDump+0x3f4
fffffa60`0495cdc0 fffffa60`0495ce00
fffffa60`0495cdc8 00000028`00000025
fffffa60`0495cd00 fffff800`018af4d0 nt! ?? ::FNOD0BFM::`string'
fffffa60`0495cd88 00000000`000000d1
fffffa60`0495cde0 fffff880`05311010
fffffa60`0495cde8 00000000`00000002
fffffa60`0495cdf0 00000000`00000000
fffffa60`0495cdf8 fffffa60`03d5917a
fffffa60`0495ce00 202a2a2a`0a0d0a0d
fffffa60`0495ce08 7830203a`504f5453
fffffa60`0495ce10 31443030`30303030
fffffa60`0495ce18 46464646`78302820
fffffa60`0495ce20 31333530`30383846
fffffa60`0495ce28 fffff800`018f5f83 nt!VidDisplayString+0x143
fffffa60`0495ce30 30303030`30300030
fffffa60`0495ce38 2c323030`30303030
fffffa60`0495ce40 30303030`30307830
fffffa60`0495ce48 30303030`30303030
fffffa60`0495ce50 46464678`302c3030
fffffa60`0495ce58 fffff800`018fe040 nt!KiInvokeBugCheckEntryCallbacks+0x80
fffffa60`0495ce60 fffffa80`02bdeb0
fffffa60`0495ce68 fffff800`01921d52 nt!InbvDisplayString+0x72
fffffa60`0495ce70 fffff880`05311000
fffffa60`0495ce78 fffff800`01d406c9 hal!KeStallExecutionProcessor+0x25

```

```
fffffa60`0495ce80 00000000`00000001
fffffa60`0495ce88 00000000`0000000a
fffffa60`0495ce90 fffffa60`03d5917a
fffffa60`0495ce98 00000000`40000082
fffffa60`0495cea0 00000000`00000001
fffffa60`0495cea8 ffffff800`01922c3e nt!KeBugCheck2+0x92e
fffffa60`0495ceb0 ffffff800`000000d1
fffffa60`0495ceb8 00000000`000004d0
fffffa60`0495cec0 ffffff800`01a43640 nt!KiProcessorBlock
fffffa60`0495cec8 00000000`0000000a
fffffa60`0495ced0 fffffa60`03d5917a
fffffa60`0495ced8 fffffa60`0495cf70
fffffa60`0495cee0 fffffa80`02bdeb0
fffffa60`0495cee8 00000000`00000000
fffffa60`0495cef0 00000000`00000000
fffffa60`0495cef8 fffffa80`02bdeb0
fffffa60`0495cf00 00000000`c21a6d00
fffffa60`0495cf08 00000000`00000000
fffffa60`0495cf10 ffffff800`0198e7a0 nt!KiInitialPCR+0x2a0
fffffa60`0495cf18 ffffff800`0198e680 nt!KiInitialPCR+0x180
fffffa60`0495cf20 fffffa80`02bb7320
fffffa60`0495cf28 00000000`00000000
fffffa60`0495cf30 00000000`00000000
fffffa60`0495cf38 ffffff960`00000003
[...]
fffffa60`0495d050 00000000`00000001
fffffa60`0495d058 00000000`83360018
fffffa60`0495d060 fffffa80`02b3ee40
fffffa60`0495d068 ffffff800`0186e650 nt!KeBugCheckEx
fffffa60`0495d070 00000000`00000000
fffffa60`0495d078 00000000`00000000
fffffa60`0495d080 00000000`00000000
fffffa60`0495d088 00000000`00000000
fffffa60`0495d090 00000000`00000000
fffffa60`0495d098 00000000`00000000
fffffa60`0495d0a0 00000000`00000000
[...]
```

If BSOD was reported after installing new drivers, we shouldn't suspect *SATA\_Driver* package here because its components would almost always be present on any bugcheck thread as referenced after a bugcheck cause. Their presence is the "effect". This example might seem trivial and pointless, but we've seen some memory dump analysis conclusions based on the reversal of causes and effects.

## Embedded Comments

Such comments in dump files are useful to record external information like the reason for saving a memory dump, a tool used to do that, and some pre-analysis and monitoring data that might help or guide in the future analysis. Comments are not widely used, but some examples include **Manual Dump** (page 630), **False Positive Dump** (page 393) patterns, and process and thread CPU consumption comments in dump files saved by Sysinternals ProcDump<sup>62</sup> tool. Such comments may not be necessarily saved by *IDebugClient2 :: WriteDumpFile2* function but any buffer saved in memory that is accessible later from a dump file will do as was easily demonstrated by the Citrix SystemDump<sup>63</sup> tool that allowed embedding comments of arbitrary complexity.

---

<sup>62</sup> <http://technet.microsoft.com/en-us/sysinternals/dd996900.aspx>

<sup>63</sup> SystemDump, Memory Dump Analysis Anthology, Volume 1, page 646

## Empty Stack Trace

Here we might need to do manual stack trace reconstruction<sup>64</sup> like shown in the following example:

```
0:002> ~2s
eax=00000070 ebx=0110fb94 ecx=00000010 edx=005725d8 esi=0110fe58 edi=00000d80
eip=7c82847c esp=0110efe0 ebp=0110eff0 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
ntdll!KiFastSystemCallRet:
7c82847c c3 ret

0:002> kL
ChildEBP RetAddr
0110efdc 00000000 ntdll!KiFastSystemCallRet

0:002> !teb
TEB at 7ffdc000
ExceptionList: 0110f980
StackBase: 01110000
StackLimit: 0110d000
SubSystemTib: 00000000
FiberData: 00001e00
ArbitraryUserPointer: 00000000
Self: 7ffdc000
EnvironmentPointer: 00000000
ClientId: 00000b04 . 00000bd0
RpcHandle: 00000000
Tls Storage: 00000000
PEB Address: 7ffdा000
LastErrorValue: 87
LastStatusValue: c000000d
Count Owned Locks: 0
HardErrorMode: 0

0:002> dps 0110d000 01110000
0110d000 00000000
0110d004 00000000
[...]
0110f640 0110f64c
0110f644 02b91ea8
0110f648 00001000
0110f64c 00000004
0110f650 0110f6f0
0110f654 0374669d DbgHelp!WriteFullMemory+0x3cd
0110f658 ffffffff
0110f65c 0110d000
0110f660 00000000
[...]
```

---

<sup>64</sup> Manual Stack Trace Reconstruction, Memory Dump Analysis Anthology, Volume 1, page 157

```
0110f6ac 00040004
0110f6b0 7ffe0000 SharedUserData
0110f6b4 00000000
0110f6b8 00001000
0110f6bc 00000000
0110f6c0 0480f5c0
0110f6c4 00000000
0110f6c8 04c4a000
0110f6cc 00000000
0110f6d0 000003c7
0110f6d4 00000000
0110f6d8 00023b17
0110f6dc 00000000
0110f6e0 01110000
0110f6e4 00000000
0110f6e8 0099f000
0110f6ec 00000000
0110f6f0 0110f704
0110f6f4 037469d6 DbgHelp!WriteDumpData+0x206
0110f6f8 0110f738
0110f6fc 0110f7b0
0110f700 00000000
0110f704 0110f868
0110f708 03747449 DbgHelp!MiniDumpProvideDump+0x359
0110f70c 0110f738
[...]
0110ff24 0000000a
0110ff28 33017f51 ModuleA!Run+0xde
0110ff2c 00000001
0110ff30 0110ff74
0110ff34 00f08898
0110ff38 00000000
0110ff3c 00f082a8
0110ff40 00000000
0110ff44 00000001
0110ff48 33017e33 ModuleA!ThreadProc+0x2c
0110ff4c a9b21e1e
0110ff50 00000000
0110ff54 00000000
0110ff58 00f08898
0110ff5c 0110ff4c
0110ff60 0110ffac
0110ff64 0110ff9c
0110ff68 33054245
0110ff6c 9ba52ad2
0110ff70 00000000
0110ff74 0110ffac
0110ff78 78543433 msrvcr90!_endthreadex+0x44
0110ff7c 00f082a8
0110ff80 a9b2b0d3
0110ff84 00000000
0110ff88 00000000
0110ff8c 00f08898
0110ff90 0110ff80
0110ff94 0110ff80
0110ff98 0110ffdc
```

```

0110ff9c 0110ffdc
0110ffa0 7858cf5e msrvcr90!_except_handler4
0110ffa4 d0f887df
0110ffa8 00000000
0110ffac 0110ffb8
0110ffb0 785434c7 msrvcr90!_endthreadex+0xd8
0110ffb4 00000000
0110ffb8 0110ffec
0110ffbc 77e6482f kernel32!BaseThreadStart+0x34
0110ffc0 00f08898
0110ffc4 00000000
0110ffc8 00000000
0110ffcc 00f08898
0110ffd0 00000000
0110ffd4 0110ffc4
0110ffd8 80833bcc
0110ffdc ffffffff
0110ffe0 77e61a60 kernel32!_except_handler3
0110ffe4 77e64838 kernel32!`string'+0x98
0110ffe8 00000000
0110ffec 00000000
0110fff0 00000000
0110fff4 7854345e msrvcr90!_endthreadex+0x6f
0110fff8 00f08898
0110ffc 00000000
01110000 00000130

```

```

0:002> k L=0110f650 0110f650 0110f650
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
0110f650 0374669d 0x110f650
0110f6f0 037469d6 DbgHelp!WriteFullMemory+0x3cd
0110f704 03747449 DbgHelp!WriteDumpData+0x206
0110f868 03747662 DbgHelp!MiniDumpProvideDump+0x359
0110f8dc 33050dd9 DbgHelp!MiniDumpWriteDump+0x1b2
[...]
0110fdfc 33031726 ModuleA!WriteExceptionMiniDump+0x50
0110fea0 33018c81 ModuleA!ThreadHung+0x6c
[...]
0110ff44 33017e33 ModuleA!Run+0xde
00000000 00000000 ModuleA!ThreadProc+0x2c

```

## Comments

---

One of the asked questions:

**Q.** Why does “k” even need the instruction pointer to walk the stack? I’ve always seen it set to EBP and ESP like this which doesn’t make sense to me.

**A.** We think it needs EIP to show the top stack frame correctly pointing to the currently executing instruction. In our pattern example, it doesn’t matter.

## Environment Hint

This pattern is useful for inconsistent dumps or incomplete supporting information. It provides information about environment variables for troubleshooting suggestions such as product elimination for testing purposes and necessary upgrade, for example:

```
0: kd> !peb
PEB at 7ffd7000
InheritedAddressSpace: No
ReadImageFileExecOptions: Yes
BeingDebugged: No
ImageBaseAddress: 01000000
Ldr 7c8897e0
Ldr.Initialized: Yes
Ldr.InInitializationOrderModuleList: 00081f18 . 000f9e88
Ldr.InLoadOrderModuleList: 00081eb0 . 000f9e78
Ldr.InMemoryOrderModuleList: 00081eb8 . 000f9e80
Base TimeStamp Module
10000000 45d6a03c Feb 17 06:27:08 2007 C:\WINNT\system32\svchost.exe
7c800000 49900d60 Feb 09 11:02:56 2009 C:\WINNT\system32\ntdll.dll
[...]
SubSystemData: 00000000
ProcessHeap: 00080000
ProcessParameters: 00020000
WindowTitle: 'C:\WINNT\system32\svchost.exe'
ImageFile: 'C:\WINNT\system32\svchost.exe'
CommandLine: 'C:\WINNT\system32\svchost.exe -k rpcss'
DllPath: [...]
Environment: 00010000
ALLUSERSPROFILE=C:\Documents and Settings\All Users
[...]
PROTECTIONDIR=C:\Documents and Settings\All Users\Application Data\3rdPartyAntivirus\Protection
[...]
Path= [...]
```

## Comments

We can get environment hints from all processes in a complete memory dump by using this command:

```
!for_each_process ".process /r /p @#Process; !peb"
```

## Error Reporting Fault

This pattern is about faults in error reporting infrastructure. The latter should be guarded against such faults to avoid recursion. Here is a summary example of such a pattern on Windows platforms that involve Windows Error Reporting (WER).

In a complete memory dump, we notice thousands of *WerFault.exe* processes:

```
0: kd> !process 0 0
[...]
PROCESS ffffffa8058010380
SessionId: 2 Cid: 488f0 Peb: 7efdf000 ParentCid: 27cb8
DirBase: 25640c000 ObjectTable: fffff8a06cd2ac50 HandleCount: 54.
Image: WerFault.exe

PROCESS ffffffa805bbd5970
SessionId: 2 Cid: 4801c Peb: 7efdf000 ParentCid: 27cb8
DirBase: 2c3f69000 ObjectTable: fffff8a040563af0 HandleCount: 54.
Image: WerFault.exe

PROCESS ffffffa8078aec060
SessionId: 2 Cid: 3feac Peb: 7efdf000 ParentCid: 488f0
DirBase: abd200000 ObjectTable: fffff8a07851a0a0 HandleCount: 59.
Image: WerFault.exe

PROCESS ffffffa805bbe9a10
SessionId: 2 Cid: 3d8b8 Peb: 7efdf000 ParentCid: 4801c
DirBase: 261f91000 ObjectTable: fffff8a02d864d40 HandleCount: 56.
Image: WerFault.exe

PROCESS ffffffa805bd29060
SessionId: 2 Cid: 1142c Peb: 7efdf000 ParentCid: 3feac
DirBase: 429fb3000 ObjectTable: fffff8a0355b42e0 HandleCount: 58.
Image: WerFault.exe

PROCESS ffffffa8053d853d0
SessionId: 2 Cid: 1fc4c Peb: 7efdf000 ParentCid: 3d8b8
DirBase: 714371000 ObjectTable: fffff8a01cb6bba0 HandleCount: 58.
Image: WerFault.exe
[...]
```

Each process has only one thread running through WOW64 modules (**Virtualized Process** pattern, page 1068), so we get its 32-bit stack trace:

```
0: kd> !process ffffffa8075c21b30 3f
[...]
THREAD ffffffa807c183b60 Cid 2d3c8.4334c Peb: 000000007efdb000 Win32Thread: fffff900c3f71010 WAIT:
(UserRequest) UserMode Non-Alertable
[...]
```

```

0: kd> .load wow64exts

0: kd> .process /r /p ffffffa8075c21b30
Implicit process is now ffffffa80`75c21b30
Loading User Symbols
Loading Wow64 Symbols

0: kd> .thread /w ffffffa807c183b60
Implicit thread is now ffffffa80`7c183b60
x86 context set

0: kd:x86> k
*** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr
000bf474 77080bbd ntdll!ZwWaitForMultipleObjects+0x15
000bf510 76bb1a2c KERNELBASE!WaitForMultipleObjectsEx+0x100
000bf558 76bb4208 kernel32!WaitForMultipleObjectsExImplementation+0xe0
000bf574 76bd80a4 kernel32!WaitForMultipleObjects+0x18
000bf5e0 76bd7f63 kernel32!WerReportFaultInternal+0x186
000bf5f4 76bd7858 kernel32!WerReportFault+0x70
000bf604 76bd77d7 kernel32!BasepReportFault+0x20
000bf690 776674df kernel32!UnhandledExceptionFilter+0x1af
000bf698 776673bc ntdll!_RtlUserThreadStart+0x62
000bf6ac 77667261 ntdll!_EH4_CallFilterFunc+0x12
000bf6d4 7764b459 ntdll!_except_handler4+0x8e
000bf6f8 7764b42b ntdll!ExecuteHandler2+0x26
000bf71c 7764b3ce ntdll!ExecuteHandler+0x24
000bf7a8 77600133 ntdll!RtlDispatchException+0x127
000bf7b4 000bf7c0 ntdll!KiUserExceptionDispatcher+0xf
WARNING: Frame IP not in any known module. Following frames may be wrong.
000fbf00 77629ef2 0xbff7c0
[...]

```

We find exception processing (**Exception Stack Trace** pattern, page 363) and the binary value in the stack trace (return address belongs to the stack region range). This thread is waiting for another process, and it is *WerFault.exe* too:

```

0: kd:x86> .effmach AMD64

0: kd> !process ffffffa8075c21b30 3f
[...]
THREAD ffffffa807c183b60 Cid 2d3c8.4334c Peb: 000000007efdb000 Win32Thread: fffff900c3f71010 WAIT:
(UserRequest) UserMode Non-Alertable
fffffa80809c44e0 ProcessObject
[...]

0: kd> !process ffffffa80809c44e0
PROCESS ffffffa80809c44e0
SessionId: 2 Cid: 33844 Peb: 7efdf000 ParentCid: 2d3c8
DirBase: 9c53f000 ObjectTable: fffff8a0423d4170 HandleCount: 978.
Image: WerFault.exe
[...]

```

We go back to our original *WerFault* process, and in its PEB data we find it was called to report a fault from another process with PID 0n189240:

```
0: kd> !process ffffffa8075c21b30 3f
[...]
CommandLine: 'C:\Windows\SysWOW64\WerFault.exe -u -p 189240 -s 3888'
[...]
```

And it is *WerFault.exe* too:

```
0: kd> !process 0n189240
Searching for Process with Cid == 2e338

PROCESS ffffffa8078b659e0
SessionId: 2 Cid: 2e338 Peb: 7efdf000 ParentCid: 47608
DirBase: 201796000 ObjectTable: fffff8a02e664380 HandleCount: 974.
Image: WerFault.exe
[...]
```

So we see a chain of *WerFault.exe* processes each processing a fault in the previous one. So there should be the first fault somewhere which we can find in **Stack Trace Collection** (page 943, including 32-bit stack traces for this example<sup>65</sup>) unless that exception stack trace data was **Paged Out** (page 778) due to insufficient memory occupied by *WerFault.exe* processes.

---

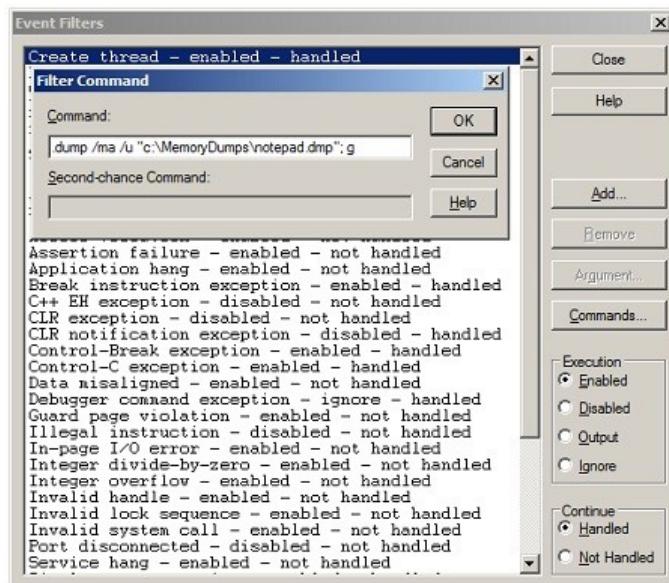
<sup>65</sup> Complete Stack Traces from x64 System, Memory Dump Analysis Anthology, Volume 5, page 30

## Evental Dumps

One of the customers of Software Diagnostics Services<sup>66</sup> submitted memory dumps saved by DebugDiag<sup>67</sup> to accompany software logs for the analysis of sudden process exit. We didn't request such memory dumps and initially dismissed them. However, during software log analysis we decided to look at **Adjoint Spaces**<sup>68</sup> to see whether there was some additional information in stack traces. We found out that those dumps were saved on each thread exit event. Since other threads were either waiting or **Active Threads** (page 63), their analysis gave clues of process behavior before process exit. For example, we found ALPC **Wait Chain** (page 1097) to **Coupled Process** (page 149). The latter prompted us to analyze **Coupled Activities**<sup>69</sup> in the software log and diagnose the possible problem there. Since saving memory dumps on thread creation and exit can be a useful technique we decided to add **Evental Dumps** memory analysis pattern to our pattern catalog.

To illustrate this pattern we show **Stack Trace Collection** (page 943) from *notepad.exe*. This process usually has just one thread. But, if we try to open a Print dialog the number of threads increases up to 12.

We attach WinDbg to *notepad.exe* process and set up debugging event filter (Debug \ Event Filters... menu) for Create Thread event with a command line as shown in this picture:



<sup>66</sup> <http://www.PatternDiagnostics.com/>

<sup>67</sup> <http://blogs.msdn.com/b/debugdiag/>

<sup>68</sup> Memory Dump Analysis Anthology, Volume 8b, page 67

<sup>69</sup> Memory Dump Analysis Anthology, Volume 9a, page 92

We then resume execution using **g** command and switch to Notepad. There we first open File \ Page Setup... dialog. We observe that a memory dump is saved. Then we open File \ Print... dialog and notice the creation 11 more process memory dumps:

notepad_09a8_2015-10-25_18-17-20-150_1344.dmp	25/10/2015 18:18	154,645 KB
notepad_09a8_2015-10-25_18-18-00-630_1344.dmp	25/10/2015 18:18	134,886 KB
notepad_09a8_2015-10-25_18-16-58-514_1344.dmp	25/10/2015 18:17	95,159 KB
notepad_09a8_2015-10-25_18-16-45-712_1344.dmp	25/10/2015 18:17	73,716 KB
notepad_09a8_2015-10-25_18-16-41-612_1344.dmp	25/10/2015 18:16	70,303 KB
notepad_09a8_2015-10-25_18-16-44-042_1344.dmp	25/10/2015 18:16	70,345 KB
notepad_09a8_2015-10-25_18-16-45-602_1344.dmp	25/10/2015 18:16	70,960 KB
notepad_09a8_2015-10-25_18-16-41-162_1344.dmp	25/10/2015 18:16	68,832 KB
notepad_09a8_2015-10-25_18-16-41-302_1344.dmp	25/10/2015 18:16	70,127 KB
notepad_09a8_2015-10-25_18-16-41-402_1344.dmp	25/10/2015 18:16	70,127 KB
notepad_09a8_2015-10-25_18-16-41-502_1344.dmp	25/10/2015 18:16	70,225 KB
notepad_09a8_2015-10-25_18-16-11-760_1344.dmp	25/10/2015 18:16	51,364 KB

We now show stack traces from these dumps where we use **\*\*kc** WinDbg command to minimize the amount of inessential for our purposes output:

```
// 1st dump
```

```
. 0 Id: 1344.1ca4 Suspend: 1 Teb: 000007ff`ffffdd000 Unfrozen
# Call Site
00 ntdll!NtAlertThread
01 ntdll!TpWaiterEnqueueTransition
02 ntdll!TpWaitpSet
03 ntdll!TpSetWait
04 ntdll!TpTimerpInitTimerQueueQueue
05 ntdll!TpTimerpAllocTimerQueue
06 ntdll!TpTimerpAcquirePoolTimerQueue
07 ntdll!TpTimerAlloc
08 ntdll!TpAllocTimer
09 KERNELBASE!CreateThreadpoolTimer
0a rpcrt4!RPC_THREAD_POOL::CreateTimer
0b rpcrt4!GarbageCollectionNeeded
0c rpcrt4!LRPC_CASSOCIATION::RemoveReference
0d rpcrt4!LRPC_CCALL::`vector deleting destructor'
0e rpcrt4!LRPC_CCALL::FreeBuffer
0f rpcrt4!Ndr64pClientFinally
10 rpcrt4!NdrpClientCall3
11 rpcrt4!NdrClientCall3
12 sechost!LsaLookupClose
13 sechost!LookupAccountNameInternal
14 sechost!LookupAccountNameLocalW
15 rpcrt4!RpcpLookupAccountNameDirect
16 rpcrt4!RpcpLookupAccountName
17 rpcrt4!LRPC_BASE_BINDING_HANDLE::SetAuthInformation
18 rpcrt4!LRPC_BINDING_HANDLE::SetAuthInformation
19 rpcrt4!RpcBindingSetAuthInfoExW
1a winspool!STRING_HANDLE_bind
```

```

1b rpcrt4!GenericHandleMgr
1c rpcrt4!ExplicitBindHandleMgr
1d rpcrt4!Ndr64pClientSetupTransferSyntax
1e rpcrt4!NdrpClientCall3
1f rpcrt4!NdrClientCall3
20 winspool!RpcSplOpenPrinter
21 winspool!OpenPrinterRPC
22 winspool!OpenPrinter2W
23 comdlg32!PrintOpenPrinter
24 comdlg32!PrintDlgX
25 comdlg32!PageSetupDlgX
26 comdlg32!PageSetupDlgW
27 notepad!NPCommand
28 notepad!NPWndProc
29 user32!UserCallWinProcCheckWow
2a user32!DispatchMessageWorker
2b notepad!WinMain
2c notepad!DisplayNonGenuineDlgWorker
2d kernel32!BaseThreadInitThunk
2e ntdll!RtlUserThreadStart

1 Id: 1344.1ab0 Suspend: 1 Teb: 000007ff`ffffdb000 Unfrozen
# Call Site
00 ntdll!RtlUserThreadStart

```

// 2nd dump

```

. 0 Id: 1344.1ca4 Suspend: 1 Teb: 000007ff`ffffdd000 Unfrozen
# Call Site
00 ntdll!ZwOpenKeyEx
01 kernel32!LocalBaseRegOpenKey
02 kernel32!RegOpenKeyExInternalW
03 kernel32!RegOpenKeyExW
04 rpcrt4!Server2003NegotiateDisable
05 rpcrt4!IsBindTimeFeatureNegotiationDisabled
06 rpcrt4!OSF_CCONNECTION::SendBindPacket
07 rpcrt4!OSF_CCONNECTION::ActuallyDoBinding
08 rpcrt4!OSF_CCONNECTION::OpenConnectionAndBind
09 rpcrt4!OSF_CCALL::BindToServer
0a rpcrt4!OSF_BINDING_HANDLE::InitCCallWithAssociation
0b rpcrt4!OSF_BINDING_HANDLE::AllocateCCall
0c rpcrt4!OSF_BINDING_HANDLE::NegotiateTransferSyntax
0d rpcrt4!I_RpcNegotiateTransferSyntax
0e rpcrt4!Ndr64pClientSetupTransferSyntax
0f rpcrt4!NdrpClientCall3
10 rpcrt4!NdrClientCall3
11 srvcli!NetShareEnum
12 ntshruui!CShareCache::RefreshNoCritSec
13 ntshruui!CShareCache::Refresh
14 ntshruui!DllMain
15 ntdll!RtlRunOnceExecuteOnce
16 kernel32!InitOnceExecuteOnce
17 ntshruui!DllGetClassObject
18 ole32!CClassCache::CDllPathEntry::DllGetClassObject
19 ole32!CClassCache::CDllFnPtrMoniker::BindToObjectNoSwitch

```

```
1a ole32!CClassCache::GetClassObject
1b ole32!CServerContextActivator::CreateInstance
1c ole32!ActivationPropertiesIn::DelegateCreateInstance
1d ole32!CApartmentActivator::CreateInstance
1e ole32!CProcessActivator::CCICallback
1f ole32!CProcessActivator::AttemptActivation
20 ole32!CProcessActivator::ActivateByContext
21 ole32!CProcessActivator::CreateInstance
22 ole32!ActivationPropertiesIn::DelegateCreateInstance
23 ole32!CClientContextActivator::CreateInstance
24 ole32!ActivationPropertiesIn::DelegateCreateInstance
25 ole32!ICoCreateInstanceEx
26 ole32!CoCreateInstance
27 shell32!_SHCoCreateInstance
28 shell32!SHExtCoCreateInstance
29 shell32!DCA_SHExtCoCreateInstance
2a shell32!CFSIconOverlayManager::_s_LoadIconOverlayIdentifiers
2b shell32!CFSIconOverlayManager::CreateInstance
2c shell32!IconOverlayManagerInit
2d shell32!GetIconOverlayManager
2e shell32!FileIconInit
2f shell32!Shell_GetImageLists
30 comdlg32!CPrintDialog::CPrintDialog
31 comdlg32!Print_GeneralDlgProc
32 user32!UserCallDlgProcCheckWow
33 user32!DefDlgProcWorker
34 user32!InternalCreateDialog
35 user32!CreateDialogIndirectParamAorW
36 user32!CreateDialogIndirectParamW
37 comctl32!_CreatePageDialog
38 comctl32!_CreatePage
39 comctl32!PageChange
3a comctl32!InitPropSheetDlg
3b comctl32!PropSheetDlgProc
3c user32!UserCallDlgProcCheckWow
3d user32!DefDlgProcWorker
3e user32!InternalCreateDialog
3f user32!CreateDialogIndirectParamAorW
40 user32!CreateDialogIndirectParamW
41 comctl32!_RealPropertySheet
42 comctl32!_PropertySheet
43 comdlg32!Print_InvokePropertySheets
44 comdlg32!PrintDlgExX
45 comdlg32!PrintDlgExW
46 notepad!GetPrinterDCviaDialog
47 notepad!PrintIt
48 notepad!NPCommand
49 notepad!NPWndProc
4a user32!UserCallWinProcCheckWow
4b user32!DispatchMessageWorker
4c notepad!WinMain
4d notepad!DisplayNonGenuineDlgWorker
4e kernel32!BaseThreadInitThunk
4f ntdll!RtlUserThreadStart
```

```

1 Id: 1344.1ab0 Suspend: 1 Teb: 000007ff`ffffdb000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 ntdll!TpWaiterPThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

2 Id: 1344.1638 Suspend: 1 Teb: 000007ff`ffffd9000 Unfrozen
# Call Site
00 ntdll!RtlUserThreadStart

```

// 3rd dump: we see 2 threads start at the same time

```

. 0 Id: 1344.1ca4 Suspend: 1 Teb: 000007ff`ffffdd000 Unfrozen
# Call Site
00 ntdll!RtlCompareMemoryUlong
01 ntdll!RtlpAllocateHeap
02 ntdll!RtlAllocateHeap
03 ntdll!RtlDebugAllocateHeap
04 ntdll! ?? ::FNODOBFM::`string'
05 ntdll!RtlAllocateHeap
06 ole32!CRpcResolver::GetThreadWinstaDesktop
07 ole32!CRpcResolver::GetConnection
08 ole32!CoInitializeSecurity
09 ole32!InitializeSecurity
0a ole32!ChannelProcessInitialize
0b ole32!CComApartment::InitRemoting
0c ole32!CGIPTable::RegisterInterfaceInGlobalHlp
0d ole32!CGIPTable::RegisterInterfaceInGlobal
0e shell32!MarshalToGIT
0f shell32!CBrowserProgressAggregator::BeginSession
10 shell32!IUnknown_BeginBrowserProgressSession
11 shell32!CDefView::CreateViewWindow3
12 shell32!CExplorerBrowser::_CreateViewWindow
13 shell32!CExplorerBrowser::_SwitchView
14 shell32!CExplorerBrowser::_BrowseToView
15 shell32!CExplorerBrowser::_BrowseObjectInternal
16 shell32!CExplorerBrowser::_OnBrowseObject
17 shell32!CExplorerBrowser::BrowseObject
18 comdlg32!CPrintDialog::CreatePrintBrowser
19 comdlg32!CPrintDialog::OnInitDialog
1a comdlg32!Print_GeneralDlgProc
1b user32!UserCallDlgProcCheckWow
1c user32!DefDlgProcWorker
1d user32!InternalCreateDialog
1e user32!CreateDialogIndirectParamAorW
1f user32!CreateDialogIndirectParamW
20 comctl32!_CreatePageDialog
21 comctl32!_CreatePage
22 comctl32!PageChange
23 comctl32!InitPropSheetDlg
24 comctl32!PropSheetDlgProc
25 user32!UserCallDlgProcCheckWow
26 user32!DefDlgProcWorker
27 user32!InternalCreateDialog

```

```
28 user32!CreateDialogIndirectParamAorW
29 user32!CreateDialogIndirectParamW
2a comctl32!_RealPropertySheet
2b comctl32!_PropertySheet
2c comdlg32!Print_InvokePropertySheets
2d comdlg32!PrintDlgExX
2e comdlg32!PrintDlgExW
2f notepad!GetPrinterDCviaDialog
30 notepad!PrintIt
31 notepad!NPCommand
32 notepad!NPWndProc
33 user32!UserCallWinProcCheckWow
34 user32!DispatchMessageWorker
35 notepad!WinMain
36 notepad!DisplayNonGenuineDlgWorker
37 kernel32!BaseThreadInitThunk
38 ntdll!RtlUserThreadStart

1 Id: 1344.1ab0 Suspend: 1 Teb: 000007ff`ffffdb000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 ntdll!TppWaiterpThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

2 Id: 1344.1638 Suspend: 1 Teb: 000007ff`ffffd9000 Unfrozen
# Call Site
00 ntdll!ZwAlpcSendWaitReceivePort
01 rpcrt4!LRPC_CCALL::SendReceive
02 rpcrt4!NdrpClientCall3
03 rpcrt4!NdrClientCall3
04 sechost!LsaLookupOpenLocalPolicy
05 sechost!LookupAccountNameInternal
06 sechost!LookupAccountNameLocalW
07 rpcrt4!RpcpLookupAccountNameDirect
08 rpcrt4!RpcpLookupAccountName
09 rpcrt4!LRPC_BASE_BINDING_HANDLE::SetAuthInformation
0a rpcrt4!LRPC_BINDING_HANDLE::SetAuthInformation
0b rpcrt4!RpcBindingSetAuthInfoExW
0c winspool!STRING_HANDLE_bind
0d rpcrt4!GenericHandleMgr
0e rpcrt4!ExplicitBindHandleMgr
0f rpcrt4!Ndr64pClientSetupTransferSyntax
10 rpcrt4!NdrpClientCall3
11 rpcrt4!NdrClientCall3
12 winspool!RpcSplOpenPrinter
13 winspool!OpenPrinterRPC
14 winspool!OpenPrinter2W
15 prncache!PrintCache::Listeners::Listener::Start
16 prncache!PrintCache::Listeners::Listener::StartCB
17 ntdll!TppWorkpExecuteCallback
18 ntdll!TppWorkerThread
19 kernel32!BaseThreadInitThunk
1a ntdll!RtlUserThreadStart
```

```
3 Id: 1344.830 Suspend: 1 Teb: 000007ff`ffffd7000 Unfrozen
# Call Site
00 ntdll!RtlUserThreadStart

4 Id: 1344.1edc Suspend: 0 Teb: 000007ff`ffffd5000 Unfrozen
# Call Site
00 ntdll!RtlUserThreadStart
```

// 4th dump

```
. 0 Id: 1344.1ca4 Suspend: 1 Teb: 000007ff`ffffdd000 Unfrozen
# Call Site
00 ntdll!RtlCompareMemoryUlong
01 ntdll!RtlpAllocateHeap
02 ntdll!RtlAllocateHeap
03 ntdll!RtlDebugAllocateHeap
04 ntdll! ?? ::FNODOBFM::`string'
05 ntdll!RtlAllocateHeap
06 ole32!CRpcResolver::GetThreadWinstaDesktop
07 ole32!CRpcResolver::GetConnection
08 ole32!CoInitializeSecurity
09 ole32!InitializeSecurity
0a ole32!ChannelProcessInitialize
0b ole32!CComApartment::InitRemoting
0c ole32!CGIPTable::RegisterInterfaceInGlobalHlp
0d ole32!CGIPTable::RegisterInterfaceInGlobal
0e shell32!MarshalToGIT
0f shell32!CBrowserProgressAggregator::BeginSession
10 shell32!IUnknown_BeginBrowserProgressSession
11 shell32!CDefView::CreateViewWindow3
12 shell32!CExplorerBrowser::_CreateViewWindow
13 shell32!CExplorerBrowser::_SwitchView
14 shell32!CExplorerBrowser::_BrowseToView
15 shell32!CExplorerBrowser::_BrowseObjectInternal
16 shell32!CExplorerBrowser::_OnBrowseObject
17 shell32!CExplorerBrowser::BrowseObject
18 comdlg32!CPrintDialog::CreatePrintBrowser
19 comdlg32!CPrintDialog::OnInitDialog
1a comdlg32!Print_GeneralDlgProc
1b user32!UserCallDlgProcCheckWow
1c user32!DefDlgProcWorker
1d user32!InternalCreateDialog
1e user32!CreateDialogIndirectParamAorW
1f user32!CreateDialogIndirectParamW
20 comctl32!_CreatePageDialog
21 comctl32!_CreatePage
22 comctl32!PageChange
23 comctl32!InitPropSheetDlg
24 comctl32!PropSheetDlgProc
25 user32!UserCallDlgProcCheckWow
26 user32!DefDlgProcWorker
27 user32!InternalCreateDialog
28 user32!CreateDialogIndirectParamAorW
29 user32!CreateDialogIndirectParamW
2a comctl32!_RealPropertySheet
```

```
2b comctl32!_PropertySheet
2c comdlg32!Print_InvokePropertySheets
2d comdlg32!PrintDlgExX
2e comdlg32!PrintDlgExW
2f notepad!GetPrinterDCviaDialog
30 notepad!PrintIt
31 notepad!NPCommand
32 notepad!NPWndProc
33 user32!UserCallWinProcCheckWow
34 user32!DispatchMessageWorker
35 notepad!WinMain
36 notepad!DisplayNonGenuineDlgWorker
37 kernel32!BaseThreadInitThunk
38 ntdll!RtlUserThreadStart

1 Id: 1344.1ab0 Suspend: 1 Teb: 000007ff`ffffdb000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 ntdll!TpWaiterpThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

2 Id: 1344.1638 Suspend: 1 Teb: 000007ff`ffffd9000 Unfrozen
# Call Site
00 ntdll!ZwAlpcSendWaitReceivePort
01 rpcrt4!LRPC_CCALL::SendReceive
02 rpcrt4!NdrpClientCall3
03 rpcrt4!NdrClientCall3
04 sechost!LsaLookupOpenLocalPolicy
05 sechost!LookupAccountNameInternal
06 sechost!LookupAccountNameLocalW
07 rpcrt4!RpcpLookupAccountNameDirect
08 rpcrt4!RpcpLookupAccountName
09 rpcrt4!LRPC_BASE_BINDING_HANDLE::SetAuthInformation
0a rpcrt4!RPC_BINDING_HANDLE::SetAuthInformation
0b rpcrt4!RpcBindingSetAuthInfoExW
0c winspool!STRING_HANDLE_bind
0d rpcrt4!GenericHandleMgr
0e rpcrt4!ExplicitBindHandleMgr
0f rpcrt4!Ndr64pClientSetupTransferSyntax
10 rpcrt4!NdrpClientCall3
11 rpcrt4!NdrClientCall3
12 winspool!RpcSplOpenPrinter
13 winspool!OpenPrinterRPC
14 winspool!OpenPrinter2W
15 prncache!PrintCache::Listeners::Listener::Start
16 prncache!PrintCache::Listeners::Listener::StartCB
17 ntdll!TppWorkpExecuteCallback
18 ntdll!TppWorkerThread
19 kernel32!BaseThreadInitThunk
1a ntdll!RtlUserThreadStart
```

```

3 Id: 1344.830 Suspend: 1 Teb: 000007ff`ffffd7000 Unfrozen
# Call Site
00 ntdll!LdrpInitializeThread
01 ntdll!LdrpInitialize
02 ntdll!LdrInitializeThunk

4 Id: 1344.1edc Suspend: 1 Teb: 000007ff`ffffd5000 Unfrozen
# Call Site
00 ntdll!RtlUserThreadStart

```

// 5th dump

```

. 0 Id: 1344.1ca4 Suspend: 1 Teb: 000007ff`ffffdd000 Unfrozen
# Call Site
00 ntdll!ZwAlpcSendWaitReceivePort
01 rpcrt4!LRPC_CCALL::SendReceive
02 rpcrt4!NdrpClientCall3
03 rpcrt4!NdrClientCall3
04 sechost!LsaLookupClose
05 sechost!LookupAccountNameInternal
06 sechost!LookupAccountNameLocalW
07 rpcrt4!RcpLookupAccountNameDirect
08 rpcrt4!RcpLookupAccountName
09 rpcrt4!LRPC_BASE_BINDING_HANDLE::SetAuthInformation
0a rpcrt4!LRPC_FAST_BINDING_HANDLE::SetAuthInformation
0b rpcrt4!LRPC_FAST_BINDING_HANDLE::LRPC_FAST_BINDING_HANDLE
0c rpcrt4!LrpCreateFastBindingHandle
0d rpcrt4!RpcBindingCreateW
0e ole32!CFastBH::CreateFromBindingString
0f ole32!CFastBH::GetOrCreate
10 ole32!CRpcResolver::GetConnection
11 ole32!CoInitializeSecurity
12 ole32!InitializeSecurity
13 ole32!ChannelProcessInitialize
14 ole32!CComApartment::InitRemoting
15 ole32!CGIPTable::RegisterInterfaceInGlobalHlp
16 ole32!CGIPTable::RegisterInterfaceInGlobal
17 shell32!MarshalToGIT
18 shell32!CBrowserProgressAggregator::BeginSession
19 shell32!IUnknown_BeginBrowserProgressSession
1a shell32!CDefView::CreateViewWindow3
1b shell32!CExplorerBrowser::_CreateViewWindow
1c shell32!CExplorerBrowser::_SwitchView
1d shell32!CExplorerBrowser::_BrowseToView
1e shell32!CExplorerBrowser::_BrowseObjectInternal
1f shell32!CExplorerBrowser::_OnBrowseObject
20 shell32!CExplorerBrowser::BrowseObject
21 comdlg32!CPrintDialog::CreatePrintBrowser
22 comdlg32!CPrintDialog::OnInitDialog
23 comdlg32!Print_GeneralDlgProc
24 user32!UserCallDlgProcCheckWow
25 user32!DefDlgProcWorker
26 user32!InternalCreateDialog
27 user32!CreateDialogIndirectParamAorW
28 user32!CreateDialogIndirectParamW

```

```
29 comctl32!_CreatePageDialog
2a comctl32!_CreatePage
2b comctl32!PageChange
2c comctl32!InitPropSheetDlg
2d comctl32!PropSheetDlgProc
2e user32!UserCallDlgProcCheckWow
2f user32!DefDlgProcWorker
30 user32!InternalCreateDialog
31 user32!CreateDialogIndirectParamAorW
32 user32!CreateDialogIndirectParamW
33 comctl32!_RealPropertySheet
34 comctl32!_PropertySheet
35 comdlg32!Print_InvokePropertySheets
36 comdlg32!PrintDlgExX
37 comdlg32!PrintDlgExW
38 notepad!GetPrinterDCviaDialog
39 notepad!PrintIt
3a notepad!NPCommand
3b notepad!NPWndProc
3c user32!UserCallWinProcCheckWow
3d user32!DispatchMessageWorker
3e notepad!WinMain
3f notepad!DisplayNonGenuineDlgWorker
40 kernel32!BaseThreadInitThunk
41 ntdll!RtlUserThreadStart

1 Id: 1344.1ab0 Suspend: 1 Teb: 000007ff`ffffdb000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 ntdll!TppWaiterpThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

2 Id: 1344.1638 Suspend: 1 Teb: 000007ff`ffffd9000 Unfrozen
# Call Site
00 ntdll!ZwAlpcSendWaitReceivePort
01 rpcrt4!LRPC_CCALL::SendReceive
02 rpcrt4!NdrpClientCall3
03 rpcrt4!NdrClientCall3
04 winspool!RpcSplOpenPrinter
05 winspool!OpenPrinterRPC
06 winspool!OpenPrinter2W
07 prncache!PrintCache::Listeners::Listener::Start
08 prncache!PrintCache::Listeners::Listener::StartCB
09 ntdll!TppWorkpExecuteCallback
0a ntdll!TppWorkerThread
0b kernel32!BaseThreadInitThunk
0c ntdll!RtlUserThreadStart

3 Id: 1344.830 Suspend: 1 Teb: 000007ff`ffffd7000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 KERNELBASE!WaitForMultipleObjectsEx
02 kernel32!WaitForMultipleObjects
03 prncache!PrintCache::Store::CacheStore::RegistryMonitor::MonitorRegistryChanges
04 prncache!PrintCache::Store::CacheStore::RegistryMonitor::MonitorRegistryChangesThreadProc
```

```

05 kernel32!BaseThreadInitThunk
06 ntdll!RtlUserThreadStart

4 Id: 1344.1edc Suspend: 1 Teb: 000007ff`ffffd5000 Unfrozen
# Call Site
00 ntdll!ZwAlpcSendWaitReceivePort
01 rpcrt4!LRPC_CCALL::SendReceive
02 rpcrt4!NdrpClientCall3
03 rpcrt4!NdrClientCall3
04 winspool!RpcEnumPrinters
05 winspool!EnumPrintersW
06 prncache!PrintCache::Listeners::ConnectionListener::EnumConnectionsAndRegister
07 prncache!PrintCache::Listeners::ConnectionListener::UpdateCB
08 ntdll!TppWorkpExecuteCallback
09 ntdll!TppWorkerThread
0a kernel32!BaseThreadInitThunk
0b ntdll!RtlUserThreadStart

5 Id: 1344.1b44 Suspend: 1 Teb: 000007ff`ffffd3000 Unfrozen
# Call Site
00 ntdll!RtlUserThreadStart

```

// 6th dump

```

. 0 Id: 1344.1ca4 Suspend: 1 Teb: 000007ff`ffffdd000 Unfrozen
# Call Site
00 ntdll!NtAlpcConnectPort
01 rpcrt4!LRPC_CASSOCIATION::AlpcConnect
02 rpcrt4!LRPC_CASSOCIATION::Connect
03 rpcrt4!LRPC_BASE_BINDING_HANDLE::DriveStateForward
04 rpcrt4!LRPC_FAST_BINDING_HANDLE::Bind
05 rpcrt4!RpcBindingBind
06 ole32!CFastBH::CreateFromBindingString
07 ole32!CFastBH::GetOrCreate
08 ole32!CRpcResolver::GetConnection
09 ole32!CoInitializeSecurity
0a ole32!InitializeSecurity
0b ole32!ChannelProcessInitialize
0c ole32!CComApartment::InitRemoting
0d ole32!CGIPTable::RegisterInterfaceInGlobalHlp
0e ole32!CGIPTable::RegisterInterfaceInGlobal
0f shell32!MarshalToGIT
10 shell32!CBrowserProgressAggregator::BeginSession
11 shell32!IUnknown_BeginBrowserProgressSession
12 shell32!CDefView::CreateViewWindow3
13 shell32!CExplorerBrowser::_CreateViewWindow
14 shell32!CExplorerBrowser::_SwitchView
15 shell32!CExplorerBrowser::_BrowseToView
16 shell32!CExplorerBrowser::_BrowseObjectInternal
17 shell32!CExplorerBrowser::_OnBrowseObject
18 shell32!CExplorerBrowser::BrowseObject
19 comdlg32!CPrintDialog::CreatePrintBrowser
1a comdlg32!CPrintDialog::OnInitDialog
1b comdlg32!Print_GeneralDlgProc
1c user32!UserCallDlgProcCheckWow

```

```
1d user32!DefDlgProcWorker
1e user32!InternalCreateDialog
1f user32!CreateDialogIndirectParamAorW
20 user32!CreateDialogIndirectParamW
21 comctl32!_CreatePageDialog
22 comctl32!_CreatePage
23 comctl32!PageChange
24 comctl32!InitPropSheetDlg
25 comctl32!PropSheetDlgProc
26 user32!UserCallDlgProcCheckWow
27 user32!DefDlgProcWorker
28 user32!InternalCreateDialog
29 user32!CreateDialogIndirectParamAorW
2a user32!CreateDialogIndirectParamW
2b comctl32!_RealPropertySheet
2c comctl32!_PropertySheet
2d comdlg32!Print_InvokePropertySheets
2e comdlg32!PrintDlgExX
2f comdlg32!PrintDlgExW
30 notepad!GetPrinterDCviaDialog
31 notepad!PrintIt
32 notepad!NPCommand
33 notepad!NPWndProc
34 user32!UserCallWinProcCheckWow
35 user32!DispatchMessageWorker
36 notepad!WinMain
37 notepad!DisplayNonGenuineDlgWorker
38 kernel32!BaseThreadInitThunk
39 ntdll!RtlUserThreadStart

1 Id: 1344.1ab0 Suspend: 1 Teb: 000007ff`ffffdb000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 ntdll!TppWaiterpThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

2 Id: 1344.1638 Suspend: 1 Teb: 000007ff`ffffd9000 Unfrozen
# Call Site
00 ntdll!RtlEnterCriticalSection
01 ntdll!RtlDebugAllocateHeap
02 ntdll! ?? ::FNODOBFM::`string'
03 ntdll!RtlAllocateHeap
04 rpcrt4!AllocWrapper
05 rpcrt4!SID_CACHE::Query
06 rpcrt4!RpcLookupAccountName
07 rpcrt4!LRPC_BASE_BINDING_HANDLE::SetAuthInformation
08 rpcrt4!LRPC_BINDING_HANDLE::SetAuthInformation
09 rpcrt4!RpcBindingSetAuthInfoExW
0a winspool!STRING_HANDLE_bind
0b rpcrt4!GenericHandleMgr
0c rpcrt4!ExplicitBindHandleMgr
0d rpcrt4!Ndr64pClientSetupTransferSyntax
0e rpcrt4!NdrpClientCall13
0f rpcrt4!NdrClientCall13
10 winspool!RpcSplOpenPrinter
```

```

11 winspool!OpenPrinterRPC
12 winspool!OpenPrinter2W
13 prncache!PrintCache::Listeners::Listener::Start
14 prncache!PrintCache::Listeners::Listener::StartCB
15 ntdll!TppWorkpExecuteCallback
16 ntdll!TppWorkerThread
17 kernel32!BaseThreadInitThunk
18 ntdll!RtlUserThreadStart

3 Id: 1344.830 Suspend: 1 Teb: 000007ff`ffffd7000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 KERNELBASE!WaitForMultipleObjectsEx
02 kernel32!WaitForMultipleObjects
03 prncache!PrintCache::Store::CacheStore::RegistryMonitor::MonitorRegistryChanges
04 prncache!PrintCache::Store::CacheStore::RegistryMonitor::MonitorRegistryChangesThreadProc
05 kernel32!BaseThreadInitThunk
06 ntdll!RtlUserThreadStart

4 Id: 1344.1edc Suspend: 1 Teb: 000007ff`ffffd5000 Unfrozen
# Call Site
00 user32!NtUserAttachThreadInput
01 shell32!CWaitTask::s_WaitBeforeCursing
02 ntdll!RtlpTpWaitCallback
03 ntdll!TppWaitpExecuteCallback
04 ntdll!TppWorkerThread
05 kernel32!BaseThreadInitThunk
06 ntdll!RtlUserThreadStart

5 Id: 1344.1b44 Suspend: 1 Teb: 000007ff`ffffd3000 Unfrozen
# Call Site
00 ntdll!NtWaitForWorkViaWorkerFactory
01 ntdll!TppWorkerThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

6 Id: 1344.1d9c Suspend: 1 Teb: 000007ff`ffffae000 Unfrozen
# Call Site
00 ntdll!RtlUserThreadStart

```

// 7th dump

```
.
. 0 Id: 1344.1ca4 Suspend: 1 Teb: 000007ff`ffffdd000 Unfrozen
# Call Site
00 rpcrt4!RpcStringFreeW
01 ole32!CFastBH::CreateFromBindingString
02 ole32!CFastBH::GetOrCreate
03 ole32!CRpcResolver::GetConnection
04 ole32!CoInitializeSecurity
05 ole32!InitializeSecurity
06 ole32!ChannelProcessInitialize
07 ole32!CComApartment::InitRemoting
08 ole32!CGIPTable::RegisterInterfaceInGlobalHlp
09 ole32!CGIPTable::RegisterInterfaceInGlobal
```

```

0a shell32!MarshalToGIT
0b shell32!CBrowserProgressAggregator::BeginSession
0c shell32!IUnknown_BeginBrowserProgressSession
0d shell32!CDefView::CreateViewWindow3
0e shell32!CExplorerBrowser::_CreateViewWindow
0f shell32!CExplorerBrowser::_SwitchView
10 shell32!CExplorerBrowser::_BrowseToView
11 shell32!CExplorerBrowser::_BrowseObjectInternal
12 shell32!CExplorerBrowser::_OnBrowseObject
13 shell32!CExplorerBrowser::BrowseObject
14 comdlg32!CPrintDialog::CreatePrintBrowser
15 comdlg32!CPrintDialog::OnInitDialog
16 comdlg32!Print_GeneralDlgProc
17 user32!UserCallDlgProcCheckWow
18 user32!DefDlgProcWorker
19 user32!InternalCreateDialog
1a user32!CreateDialogIndirectParamAorW
1b user32!CreateDialogIndirectParamW
1c comctl32!_CreatePageDialog
1d comctl32!_CreatePage
1e comctl32!PageChange
1f comctl32!InitPropSheetDlg
20 comctl32!PropSheetDlgProc
21 user32!UserCallDlgProcCheckWow
22 user32!DefDlgProcWorker
23 user32!InternalCreateDialog
24 user32!CreateDialogIndirectParamAorW
25 user32!CreateDialogIndirectParamW
26 comctl32!_RealPropertySheet
27 comctl32!_PropertySheet
28 comdlg32!Print_InvokePropertySheets
29 comdlg32!PrintDlgExX
2a comdlg32!PrintDlgExW
2b notepad!GetPrinterDCviaDialog
2c notepad!PrintIt
2d notepad!NPCommand
2e notepad!NPWndProc
2f user32!UserCallWinProcCheckWow
30 user32!DispatchMessageWorker
31 notepad!WinMain
32 notepad!DisplayNonGenuineDlgWorker
33 kernel32!BaseThreadInitThunk
34 ntdll!RtlUserThreadStart

1 Id: 1344.1ab0 Suspend: 1 Teb: 000007ff`ffffdb000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 ntdll!TpWaiterpThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

2 Id: 1344.1638 Suspend: 1 Teb: 000007ff`ffffd9000 Unfrozen
# Call Site
00 ntdll!ZwAlpcSendWaitReceivePort
01 rpcrt4!LRPC_CCALL::SendReceive
02 rpcrt4!NdrpClientCall3

```

```
03 rpcrt4!NdrClientCall3
04 winspool!RpcSplOpenPrinter
05 winspool!OpenPrinterRPC
06 winspool!OpenPrinter2W
07 prncache!PrintCache::Listeners::Listener::Start
08 prncache!PrintCache::Listeners::Listener::StartCB
09 ntdll!TpWorkpExecuteCallback
0a ntdll!TpWorkerThread
0b kernel32!BaseThreadInitThunk
0c ntdll!RtlUserThreadStart

3 Id: 1344.830 Suspend: 1 Teb: 000007ff`ffffd7000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 KERNELBASE!WaitForMultipleObjectsEx
02 kernel32!WaitForMultipleObjects
03 prncache!PrintCache::Store::CacheStore::RegistryMonitor::MonitorRegistryChanges
04 prncache!PrintCache::Store::CacheStore::RegistryMonitor::MonitorRegistryChangesThreadProc
05 kernel32!BaseThreadInitThunk
06 ntdll!RtlUserThreadStart

4 Id: 1344.1edc Suspend: 1 Teb: 000007ff`ffffd5000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 KERNELBASE!WaitForMultipleObjectsEx
02 kernel32!WaitForMultipleObjectsExImplementation
03 user32!RealMsgWaitForMultipleObjectsEx
04 user32!MsgWaitForMultipleObjectsEx
05 user32!MsgWaitForMultipleObjects
06 shell32!SHProcessMessagesUntilEventsEx
07 shell32!CWaitTask::s_WaitBeforeCursing
08 ntdll!RtlpTpWaitCallback
09 ntdll!TpWorkpExecuteCallback
0a ntdll!TpWorkerThread
0b kernel32!BaseThreadInitThunk
0c ntdll!RtlUserThreadStart

5 Id: 1344.1b44 Suspend: 1 Teb: 000007ff`ffffd3000 Unfrozen
# Call Site
00 ntdll!NtWaitForWorkViaWorkerFactory
01 ntdll!TpWorkerThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

6 Id: 1344.1d9c Suspend: 1 Teb: 000007ff`ffffae000 Unfrozen
# Call Site
00 ntdll!NtWaitForWorkViaWorkerFactory
01 ntdll!TpWorkerThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

7 Id: 1344.139c Suspend: 1 Teb: 000007ff`ffffac000 Unfrozen
# Call Site
00 ntdll!RtlUserThreadStart
```

// 8th dump

```
. 0 Id: 1344.1ca4 Suspend: 1 Teb: 000007ff`ffffdd000 Unfrozen
# Call Site
00 user32!NtUserSetWindowLongPtr
01 user32!SetWindowLongPtr
02 ole32!OXIDEntry::StartServer
03 ole32!CGIPTable::RegisterInterfaceInGlobalHlp
04 ole32!CGIPTable::RegisterInterfaceInGlobal
05 shell32!MarshalToGIT
06 shell32!CBrowserProgressAggregator::BeginSession
07 shell32!IUnknown_BeginBrowserProgressSession
08 shell32!CDefView::CreateViewWindow3
09 shell32!CExplorerBrowser::_CreateViewWindow
0a shell32!CExplorerBrowser::_SwitchView
0b shell32!CExplorerBrowser::_BrowseToView
0c shell32!CExplorerBrowser::_BrowseObjectInternal
0d shell32!CExplorerBrowser::_OnBrowseObject
0e shell32!CExplorerBrowser::BrowseObject
0f comdlg32!CPrintDialog::CreatePrintBrowser
10 comdlg32!CPrintDialog::OnInitDialog
11 comdlg32!Print_GeneralDlgProc
12 user32!UserCallDlgProcCheckWow
13 user32!DefDlgProcWorker
14 user32!InternalCreateDialog
15 user32!CreateDialogIndirectParamAorW
16 user32!CreateDialogIndirectParamW
17 comctl32!_CreatePageDialog
18 comctl32!_CreatePage
19 comctl32!PageChange
1a comctl32!InitPropSheetDlg
1b comctl32!PropSheetDlgProc
1c user32!UserCallDlgProcCheckWow
1d user32!DefDlgProcWorker
1e user32!InternalCreateDialog
1f user32!CreateDialogIndirectParamAorW
20 user32!CreateDialogIndirectParamW
21 comctl32!_RealPropertySheet
22 comctl32!_PropertySheet
23 comdlg32!Print_InvokePropertySheets
24 comdlg32!PrintDlgExX
25 comdlg32!PrintDlgExW
26 notepad!GetPrinterDCviaDialog
27 notepad!PrintIt
28 notepad!NPCommand
29 notepad!NPWndProc
2a user32!UserCallWinProcCheckWow
2b user32!DispatchMessageWorker
2c notepad!WinMain
2d notepad!DisplayNonGenuineDlgWorker
2e kernel32!BaseThreadInitThunk
2f ntdll!RtlUserThreadStart
```

```
1 Id: 1344.1ab0 Suspend: 1 Teb: 000007ff`ffffdb000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 ntdll!TppWaiterPThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

2 Id: 1344.1638 Suspend: 1 Teb: 000007ff`ffffd9000 Unfrozen
# Call Site
00 ntdll!NtWaitForWorkViaWorkerFactory
01 ntdll!TppWorkerThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

3 Id: 1344.830 Suspend: 1 Teb: 000007ff`ffffd7000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 KERNELBASE!WaitForMultipleObjectsEx
02 kernel32!WaitForMultipleObjects
03 prncache!PrintCache::Store::CacheStore::RegistryMonitor::MonitorRegistryChanges
04 prncache!PrintCache::Store::CacheStore::RegistryMonitor::MonitorRegistryChangesThreadProc
05 kernel32!BaseThreadInitThunk
06 ntdll!RtlUserThreadStart

4 Id: 1344.1edc Suspend: 1 Teb: 000007ff`ffffd5000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 KERNELBASE!WaitForMultipleObjectsEx
02 kernel32!WaitForMultipleObjectsExImplementation
03 user32!RealMsgWaitForMultipleObjectsEx
04 user32!MsgWaitForMultipleObjectsEx
05 user32!MsgWaitForMultipleObjects
06 shell32!SHProcessMessagesUntilEventsEx
07 shell32!CWaitTask::s_WaitBeforeCursing
08 ntdll!RtlpTpWaitCallback
09 ntdll!TppWaitpExecuteCallback
0a ntdll!TppWorkerThread
0b kernel32!BaseThreadInitThunk
0c ntdll!RtlUserThreadStart

5 Id: 1344.1b44 Suspend: 1 Teb: 000007ff`ffffd3000 Unfrozen
# Call Site
00 ntdll!NtWaitForWorkViaWorkerFactory
01 ntdll!TppWorkerThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

6 Id: 1344.1d9c Suspend: 1 Teb: 000007ff`ffffae000 Unfrozen
# Call Site
00 ntdll!NtWaitForWorkViaWorkerFactory
01 ntdll!TppWorkerThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart
```

```
7 Id: 1344.139c Suspend: 1 Teb: 000007ff`ffffac000 Unfrozen
# Call Site
00 ntdll!NtWaitForWorkViaWorkerFactory
01 ntdll!TpWorkerThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

8 Id: 1344.1b80 Suspend: 1 Teb: 000007ff`ffffaa000 Unfrozen
# Call Site
00 ntdll!RtlUserThreadStart
```

// 9th dump

```
. 0 Id: 1344.1ca4 Suspend: 1 Teb: 000007ff`ffffdd000 Unfrozen
# Call Site
00 user32!NtUserPeekMessage
01 user32!PeekMessageW
02 shell32!PeekMessageWithWakeMask
03 shell32!SHProcessMessagesUntilEventsEx
04 shell32!CDefView::_SetItemCollection
05 shell32!CDefView::_CreateNewCollection
06 shell32!CDefView::CreateViewWindow3
07 shell32!CExplorerBrowser::_CreateViewWindow
08 shell32!CExplorerBrowser::_SwitchView
09 shell32!CExplorerBrowser::_BrowseToView
0a shell32!CExplorerBrowser::_BrowseObjectInternal
0b shell32!CExplorerBrowser::_OnBrowseObject
0c shell32!CExplorerBrowser::BrowseObject
0d comdlg32!CPrintDialog::CreatePrintBrowser
0e comdlg32!CPrintDialog::OnInitDialog
0f comdlg32!Print_GeneralDlgProc
10 user32!UserCallDlgProcCheckWow
11 user32!DefDlgProcWorker
12 user32!InternalCreateDialog
13 user32!CreateDialogIndirectParamAorW
14 user32!CreateDialogIndirectParamW
15 comctl32!_CreatePageDialog
16 comctl32!_CreatePage
17 comctl32!PageChange
18 comctl32!InitPropSheetDlg
19 comctl32!PropSheetDlgProc
1a user32!UserCallDlgProcCheckWow
1b user32!DefDlgProcWorker
1c user32!InternalCreateDialog
1d user32!CreateDialogIndirectParamAorW
1e user32!CreateDialogIndirectParamW
1f comctl32!_RealPropertySheet
20 comctl32!_PropertySheet
21 comdlg32!Print_InvokePropertySheets
22 comdlg32!PrintDlgExX
23 comdlg32!PrintDlgExW
24 notepad!GetPrinterDCviaDialog
25 notepad!PrintIt
26 notepad!NPCommand
27 notepad!NPWndProc
```

```
28 user32!UserCallWinProcCheckWow
29 user32!DispatchMessageWorker
2a notepad!WinMain
2b notepad!DisplayNonGenuineDlgWorker
2c kernel32!BaseThreadInitThunk
2d ntdll!RtlUserThreadStart

1 Id: 1344.1ab0 Suspend: 1 Teb: 000007ff`ffffdb000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 ntdll!TppWaiterpThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

2 Id: 1344.1638 Suspend: 1 Teb: 000007ff`ffffd9000 Unfrozen
# Call Site
00 ntdll!NtWaitForSingleObject
01 KERNELBASE!WaitForSingleObjectEx
02 shlwapi!CreateThreadWorker
03 shlwapi!SHCreateThread
04 shell32!CEnumThread::Run
05 shell32!CEnumTask::_StartEnumThread
06 shell32!CEnumTask::_IncrEnumFolder
07 shell32!CEnumTask::InternalResumeRT
08 shell32!CRunnableTask::Run
09 shell32!CShellTask::TT_Run
0a shell32!CShellTaskThread::ThreadProc
0b shell32!CShellTaskThread::s_ThreadProc
0c shlwapi!ExecuteWorkItemThreadProc
0d ntdll!RtlpTpWorkCallback
0e ntdll!TppWorkerThread
0f kernel32!BaseThreadInitThunk
10 ntdll!RtlUserThreadStart

3 Id: 1344.830 Suspend: 1 Teb: 000007ff`ffffd7000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 KERNELBASE!WaitForMultipleObjectsEx
02 kernel32!WaitForMultipleObjects
03 prncache!PrintCache::Store::CacheStore::RegistryMonitor::MonitorRegistryChanges
04 prncache!PrintCache::Store::CacheStore::RegistryMonitor::MonitorRegistryChangesThreadProc
05 kernel32!BaseThreadInitThunk
06 ntdll!RtlUserThreadStart

4 Id: 1344.1edc Suspend: 1 Teb: 000007ff`ffffd5000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 KERNELBASE!WaitForMultipleObjectsEx
02 kernel32!WaitForMultipleObjectsExImplementation
03 user32!RealMsgWaitForMultipleObjectsEx
04 user32!MsgWaitForMultipleObjectsEx
05 user32!MsgWaitForMultipleObjects
06 shell32!SHProcessMessagesUntilEventsEx
07 shell32!CWaitTask::s_WaitBeforeCursing
08 ntdll!RtlpTpWaitCallback
```

```

09 ntdll!TppWaitpExecuteCallback
0a ntdll!TppWorkerThread
0b kernel32!BaseThreadInitThunk
0c ntdll!RtlUserThreadStart

5 Id: 1344.1b44 Suspend: 1 Teb: 000007ff`ffffd3000 Unfrozen
# Call Site
00 ntdll!NtWaitForWorkViaWorkerFactory
01 ntdll!TppWorkerThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

6 Id: 1344.1d9c Suspend: 1 Teb: 000007ff`ffffae000 Unfrozen
# Call Site
00 ntdll!NtWaitForWorkViaWorkerFactory
01 ntdll!TppWorkerThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

7 Id: 1344.139c Suspend: 1 Teb: 000007ff`ffffac000 Unfrozen
# Call Site
00 ntdll!NtWaitForWorkViaWorkerFactory
01 ntdll!TppWorkerThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

8 Id: 1344.1b80 Suspend: 1 Teb: 000007ff`ffffaa000 Unfrozen
# Call Site
00 ntdll!ZwDelayExecution
01 KERNELBASE!SleepEx
02 ole32!CROIDTable::WorkerThreadLoop
03 ole32!CRpcThread::WorkerLoop
04 ole32!CRpcThreadCache::RpcWorkerThreadEntry
05 kernel32!BaseThreadInitThunk
06 ntdll!RtlUserThreadStart

9 Id: 1344.1310 Suspend: 1 Teb: 000007ff`ffffa8000 Unfrozen
# Call Site
00 ntdll!RtlUserThreadStart

```

// 10th dump

```

. 0 Id: 1344.1ca4 Suspend: 1 Teb: 000007ff`ffffdd000 Unfrozen
# Call Site
00 kernel32!TlsGetValue
01 usp10!UspFreeMem
02 usp10!ScriptApplyDigitSubstitution
03 lpk!LpkCharsetDraw
04 lpk!LpkDrawTextEx
05 user32!DT_GetExtentMinusPrefixes
06 user32!NeedsEndEllipsis
07 user32!AddEllipsisAndDrawLine
08 user32!DrawTextExWorker
09 user32!DrawTextW

```

```

0a comctl32!CLVView::_ComputeLabelSizeWorker
0b comctl32!CLVView::v_RecomputeLabelSize
0c comctl32!CLVListView::v_DrawItem
0d comctl32!CLVDrawItemManager::DrawItem
0e comctl32!CLVDrawManager::PaintItems
0f comctl32!CLVDrawManager::PaintWorkArea
10 comctl32!CLVDrawManager::OnPaintWorkAreas
11 comctl32!CLVDrawManager::OnPaint
12 comctl32!CListView::WndProc
13 comctl32!CListView::s_WndProc
14 user32!UserCallWinProcCheckWow
15 user32!CallWindowProcAorW
16 user32!CallWindowProcW
17 comctl32!CallOriginalWndProc
18 comctl32!CallNextSubclassProc
19 comctl32!DefSubclassProc
1a shell32!DefSubclassProc
1b shell32!CListViewHost::s_ListViewSubclassWndProc
1c comctl32!CallNextSubclassProc
1d comctl32!MasterSubclassProc
1e user32!UserCallWinProcCheckWow
1f user32!DispatchClientMessage
20 user32!_fnDWORD
21 ntdll!KiUserCallbackDispatcherContinue
22 user32!NtUserDispatchMessage
23 user32!DispatchMessageWorker
24 user32!IsDialogMessageW
25 comctl32!Prop_IsDialogMessage
26 comctl32!_RealPropertySheet
27 comctl32!_PropertySheet
28 comdlg32!Print_InvokePropertySheets
29 comdlg32!PrintDlgExX
2a comdlg32!PrintDlgExW
2b notepad!GetPrinterDCviaDialog
2c notepad!PrintIt
2d notepad!NPCommand
2e notepad!NPWndProc
2f user32!UserCallWinProcCheckWow
30 user32!DispatchMessageWorker
31 notepad!WinMain
32 notepad!DisplayNonGenuineDlgWorker
33 kernel32!BaseThreadInitThunk
34 ntdll!RtlUserThreadStart

1 Id: 1344.1ab0 Suspend: 1 Teb: 000007ff`ffffdb000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 ntdll!TpWaiterpThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

2 Id: 1344.1638 Suspend: 1 Teb: 000007ff`ffffd9000 Unfrozen
# Call Site
00 ntdll!ZwCreateSection
01 KERNELBASE!BasepLoadLibraryAsDataFileInternal
02 KERNELBASE!LoadLibraryExW

```

```
03 user32!PrivateExtractIconsW
04 shell32!SHPrivateExtractIcons
05 shell32!SHDefExtractIconW
06 shell32!CExtractIcon::_ExtractW
07 shell32!CExtractIconBase::Extract
08 shell32!IExtractIcon_Extract
09 shell32!_GetILIndexGivenPXiIcon
0a shell32!_GetILIndexFromItem
0b shell32!SHGetIconIndexFromPIDL
0c shell32!MapIDListToIconILIndex
0d shell32!CLoadSystemIconTask::InternalResumeRT
0e shell32!CRunnableTask::Run
0f shell32!CShellTask::TT_Run
10 shell32!CShellTaskThread::ThreadProc
11 shell32!CShellTaskThread::s_ThreadProc
12 shlwapi!ExecuteWorkItemThreadProc
13 ntdll!RtlpTpWorkCallback
14 ntdll!TppWorkerThread
15 kernel32!BaseThreadInitThunk
16 ntdll!RtlUserThreadStart

3 Id: 1344.830 Suspend: 1 Teb: 000007ff`ffffd7000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 KERNELBASE!WaitForMultipleObjectsEx
02 kernel32!WaitForMultipleObjects
03 prncache!PrintCache::Store::CacheStore::RegistryMonitor::MonitorRegistryChanges
04 prncache!PrintCache::Store::CacheStore::RegistryMonitor::MonitorRegistryChangesThreadProc
05 kernel32!BaseThreadInitThunk
06 ntdll!RtlUserThreadStart

4 Id: 1344.1edc Suspend: 1 Teb: 000007ff`ffffd5000 Unfrozen
# Call Site
00 ntdll!ZwCreateSection
01 KERNELBASE!BasepLoadLibraryAsDataFileInternal
02 KERNELBASE!LoadLibraryExW
03 user32!PrivateExtractIconsW
04 shell32!SHPrivateExtractIcons
05 shell32!SHDefExtractIconW
06 shell32!CExtractIcon::_ExtractW
07 shell32!CExtractIconBase::Extract
08 shell32!IExtractIcon_Extract
09 shell32!_GetILIndexGivenPXiIcon
0a shell32!_GetILIndexFromItem
0b shell32!SHGetIconIndexFromPIDL
0c shell32!MapIDListToIconILIndex
0d shell32!CLoadSystemIconTask::InternalResumeRT
0e shell32!CRunnableTask::Run
0f shell32!CShellTask::TT_Run
10 shell32!CShellTaskThread::ThreadProc
11 shell32!CShellTaskThread::s_ThreadProc
12 shlwapi!ExecuteWorkItemThreadProc
13 ntdll!RtlpTpWorkCallback
14 ntdll!TppWorkerThread
15 kernel32!BaseThreadInitThunk
16 ntdll!RtlUserThreadStart
```

```
5 Id: 1344.1b44 Suspend: 1 Teb: 000007ff`ffffd3000 Unfrozen
# Call Site
00 ntdll!ZwCreateSection
01 KERNELBASE!BasepLoadLibraryAsDataFileInternal
02 KERNELBASE!LoadLibraryExW
03 user32!PrivateExtractIconsW
04 shell32!SHPrivateExtractIcons
05 shell32!SHDefExtractIconW
06 shell32!CExtractIcon::_ExtractW
07 shell32!CExtractIconBase::Extract
08 shell32!IExtractIcon_Extract
09 shell32!_GetILIndexGivenPXIcon
0a shell32!_GetILIndexFromItem
0b shell32!SHGetIconIndexFromPIDL
0c shell32!MapIDListToIconILIndex
0d shell32!CLoadSystemIconTask::InternalResumeRT
0e shell32!CRunnableTask::Run
0f shell32!CShellTask::TT_Run
10 shell32!CShellTaskThread::ThreadProc
11 shell32!CShellTaskThread::s_ThreadProc
12 shlwapi!ExecuteWorkItemThreadProc
13 ntdll!RtlpTpWorkCallback
14 ntdll!TppWorkerThread
15 kernel32!BaseThreadInitThunk
16 ntdll!RtlUserThreadStart

6 Id: 1344.1d9c Suspend: 1 Teb: 000007ff`ffffae000 Unfrozen
# Call Site
00 ntdll!ZwCreateSection
01 KERNELBASE!BasepLoadLibraryAsDataFileInternal
02 KERNELBASE!LoadLibraryExW
03 user32!PrivateExtractIconsW
04 shell32!SHPrivateExtractIcons
05 shell32!SHDefExtractIconW
06 shell32!CExtractIcon::_ExtractW
07 shell32!CExtractIconBase::Extract
08 shell32!IExtractIcon_Extract
09 shell32!_GetILIndexGivenPXIcon
0a shell32!_GetILIndexFromItem
0b shell32!SHGetIconIndexFromPIDL
0c shell32!MapIDListToIconILIndex
0d shell32!CLoadSystemIconTask::InternalResumeRT
0e shell32!CRunnableTask::Run
0f shell32!CShellTask::TT_Run
10 shell32!CShellTaskThread::ThreadProc
11 shell32!CShellTaskThread::s_ThreadProc
12 shlwapi!ExecuteWorkItemThreadProc
13 ntdll!RtlpTpWorkCallback
14 ntdll!TppWorkerThread
15 kernel32!BaseThreadInitThunk
16 ntdll!RtlUserThreadStart

7 Id: 1344.139c Suspend: 1 Teb: 000007ff`ffffac000 Unfrozen
# Call Site
00 ntdll!NtQueryKey
01 kernel32!BaseRegGetKeySemantics
```

```

02 kernel32!BaseReg GetUserAndMachineClass
03 kernel32!LocalBaseRegQueryValue
04 kernel32!RegQueryValueExW
05 shlwapi!SHRegQueryValueW
06 shlwapi!SHRegGetValueW
07 shlwapi!SHQueryValueExW
08 shell32!_GetServerInfo
09 shell32!_SHCoCreateInstance
0a shell32!CRegFolder:::_CreateCachedRegFolder
0b shell32!CRegFolder:::_BindToItem
0c shell32!CRegFolder::BindToObject
0d shell32!CRegFolder:::_BindToItem
0e shell32!CRegFolder::BindToObject
0f shell32!SHBindToObject
10 shell32!CIconOverlayTask::InternalResumeRT
11 shell32!CRunnableTask::Run
12 shell32!CShellTask::TT_Run
13 shell32!CShellTaskThread::ThreadProc
14 shell32!CShellTaskThread::s_ThreadProc
15 shlwapi!ExecuteWorkItemThreadProc
16 ntdll!RtlpTpWorkCallback
17 ntdll!TppWorkerThread
18 kernel32!BaseThreadInitThunk
19 ntdll!RtlUserThreadStart

8 Id: 1344.1b80 Suspend: 1 Teb: 000007ff`ffffaa000 Unfrozen
# Call Site
00 ntdll!ZwDelayExecution
01 KERNELBASE!SleepEx
02 ole32!CROIDTable::WorkerThreadLoop
03 ole32!CRpcThread::WorkerLoop
04 ole32!CRpcThreadCache::RpcWorkerThreadEntry
05 kernel32!BaseThreadInitThunk
06 ntdll!RtlUserThreadStart

9 Id: 1344.eb8 Suspend: 1 Teb: 000007ff`ffffa8000 Unfrozen
# Call Site
00 ntdll!RtlUserThreadStart

```

// 11th dump

```

. . 0 Id: 1344.1ca4 Suspend: 1 Teb: 000007ff`ffffdd000 Unfrozen
# Call Site
00 usp10!GenericEngineGetGlyphs
01 usp10!ShlShape
02 usp10!ScriptShape
03 usp10!RenderItemNoFallback
04 usp10!RenderItemWithFallback
05 usp10!RenderItem
06 usp10!ScriptStringAnalyzeGlyphs
07 usp10!ScriptStringAnalyse
08 lpk!LpkCharsetDraw
09 lpk!LpkDrawTextEx
0a user32!DT_GetExtentMinusPrefixes
0b user32!NeedsEndEllipsis

```

```
0c user32!AddEllipsisAndDrawLine
0d user32!DrawTextExWorker
0e user32!DrawTextW
0f comctl32!CLVView::_ComputeLabelSizeWorker
10 comctl32!CLVView::v_RecomputeLabelSize
11 comctl32!CLVListView::v_DrawItem
12 comctl32!CLVDrawItemManager::DrawItem
13 comctl32!CLVDrawManager::_PaintItems
14 comctl32!CLVDrawManager::_PaintWorkArea
15 comctl32!CLVDrawManager::_OnPaintWorkAreas
16 comctl32!CLVDrawManager::_OnPaint
17 comctl32!CListView::WndProc
18 comctl32!CListView::s_WndProc
19 user32!UserCallWinProcCheckWow
1a user32!CallWindowProcAorW
1b user32!CallWindowProcW
1c comctl32!CallOriginalWndProc
1d comctl32!CallNextSubclassProc
1e comctl32!DefSubclassProc
1f shell32!DefSubclassProc
20 shell32!CListViewHost::s_ListViewSubclassWndProc
21 comctl32!CallNextSubclassProc
22 comctl32!MasterSubclassProc
23 user32!UserCallWinProcCheckWow
24 user32!DispatchClientMessage
25 user32!_fnDWORD
26 ntdll!KiUserCallbackDispatcherContinue
27 user32!NtUserDispatchMessage
28 user32!DispatchMessageWorker
29 user32!IsDialogMessageW
2a comctl32!Prop_IsDialogMessage
2b comctl32!_RealPropertySheet
2c comctl32!_PropertySheet
2d comdlg32!Print_InvokePropertySheets
2e comdlg32!PrintDlgExX
2f comdlg32!PrintDlgExW
30 notepad!GetPrinterDCviaDialog
31 notepad!PrintIt
32 notepad!NPCommand
33 notepad!NPWndProc
34 user32!UserCallWinProcCheckWow
35 user32!DispatchMessageWorker
36 notepad!WinMain
37 notepad!DisplayNonGenuineDlgWorker
38 kernel32!BaseThreadInitThunk
39 ntdll!RtlUserThreadStart

1 Id: 1344.1ab0 Suspend: 1 Teb: 000007ff`ffffdb000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 ntdll!TppWaiterpThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart
```

2 Id: 1344.1638 Suspend: 1 Teb: 000007ff`ffffd9000 Unfrozen  
# Call Site  
00 ntdll!NtWaitForSingleObject  
01 ntdll!RtlpWaitOnCriticalSection  
02 ntdll!RtlEnterCriticalSection  
03 user32!BitmapFromDIB  
04 user32!ConvertDIBBitmap  
05 user32!ConvertDIBIcon  
06 user32!CreateIconFromResourceEx  
07 user32!PrivateEnumProc  
08 kernel32!EnumResourceNamesInternal  
09 kernel32!EnumResourceNamesExW  
0a user32!PrivateExtractIconsW  
0b shell32!SHPrivateExtractIcons  
0c shell32!SHDefExtractIconW  
0d shell32!CExtractIcon::\_ExtractW  
0e shell32!CExtractIconBase::Extract  
0f shell32!IExtractIcon\_Extract  
10 shell32!\_GetILIndexGivenPXIcon  
11 shell32!\_GetILIndexFromItem  
12 shell32!SHGetIconIndexFromPIDL  
13 shell32!MapIDListToIconILIndex  
14 shell32!CLoadSystemIconTask::InternalResumeRT  
15 shell32!CRunnableTask::Run  
16 shell32!CShellTask::TT\_Run  
17 shell32!CShellTaskThread::ThreadProc  
18 shell32!CShellTaskThread::s\_ThreadProc  
19 shlwapi!ExecuteWorkItemThreadProc  
1a ntdll!RtlpTpWorkCallback  
1b ntdll!TppWorkerThread  
1c kernel32!BaseThreadInitThunk  
1d ntdll!RtlUserThreadStart

3 Id: 1344.830 Suspend: 1 Teb: 000007ff`ffffd7000 Unfrozen  
# Call Site  
00 ntdll!NtWaitForMultipleObjects  
01 KERNELBASE!WaitForMultipleObjectsEx  
02 kernel32!WaitForMultipleObjects  
03 prncache!PrintCache::Store::CacheStore::RegistryMonitor::MonitorRegistryChanges  
04 prncache!PrintCache::Store::CacheStore::RegistryMonitor::MonitorRegistryChangesThreadProc  
05 kernel32!BaseThreadInitThunk  
06 ntdll!RtlUserThreadStart

4 Id: 1344.1edc Suspend: 1 Teb: 000007ff`ffffd5000 Unfrozen  
# Call Site  
00 ntdll!NtWaitForSingleObject  
01 ntdll!RtlpWaitOnCriticalSection  
02 ntdll!RtlEnterCriticalSection  
03 user32!BitmapFromDIB  
04 user32!ConvertDIBBitmap  
05 user32!ConvertDIBIcon  
06 user32!CreateIconFromResourceEx  
07 user32!PrivateEnumProc  
08 kernel32!EnumResourceNamesInternal  
09 kernel32!EnumResourceNamesExW  
0a user32!PrivateExtractIconsW

```

0b shell32!SHPrivateExtractIcons
0c shell32!SHDefExtractIconW
0d shell32!CExtractIcon::_ExtractW
0e shell32!CExtractIconBase::Extract
0f shell32!IExtractIcon_Extract
10 shell32!_GetILIndexGivenPXIcon
11 shell32!_GetILIndexFromItem
12 shell32!SHGetIconIndexFromPIDL
13 shell32!MapIDListToIconILIndex
14 shell32!CLoadSystemIconTask::InternalResumeRT
15 shell32!CRunnableTask::Run
16 shell32!CShellTask::TT_Run
17 shell32!CShellTaskThread::ThreadProc
18 shell32!CShellTaskThread::s_ThreadProc
19 shlwapi!ExecuteWorkItemThreadProc
1a ntdll!RtlpTpWorkCallback
1b ntdll!TppWorkerThread
1c kernel32!BaseThreadInitThunk
1d ntdll!RtlUserThreadStart

5 Id: 1344.1b44 Suspend: 1 Teb: 000007ff`ffffd3000 Unfrozen
# Call Site
00 user32!NtUserGetIconInfo
01 user32!CopyIcoCur
02 user32!InternalCopyImage
03 user32!CopyImage
04 comctl32!CImageList::_ReplaceIcon
05 comctl32!CImageList::ReplaceIcon
06 comctl32!CParseImageList::ReplaceIcon
07 shell32!CIconCache::AddToBackIconTable
08 shell32!AddToBackIconTable
09 shell32!SHAddIconsToCache
0a shell32!_GetILIndexGivenPXIcon
0b shell32!_GetILIndexFromItem
0c shell32!SHGetIconIndexFromPIDL
0d shell32!MapIDListToIconILIndex
0e shell32!CLoadSystemIconTask::InternalResumeRT
0f shell32!CRunnableTask::Run
10 shell32!CShellTask::TT_Run
11 shell32!CShellTaskThread::ThreadProc
12 shell32!CShellTaskThread::s_ThreadProc
13 shlwapi!ExecuteWorkItemThreadProc
14 ntdll!RtlpTpWorkCallback
15 ntdll!TppWorkerThread
16 kernel32!BaseThreadInitThunk
17 ntdll!RtlUserThreadStart

6 Id: 1344.1d9c Suspend: 1 Teb: 000007ff`ffffae000 Unfrozen
# Call Site
00 gdi32!NtUserSelectPalette
01 gdi32!SelectPalette
02 gdi32!SetDIBits
03 user32!BitmapFromDIB
04 user32!ConvertDIBBitmap
05 user32!ConvertDIBIcon
06 user32!CreateIconFromResourceEx

```

```
07 user32!PrivateEnumProc
08 kernel32!EnumResourceNamesInternal
09 kernel32!EnumResourceNamesExW
0a user32!PrivateExtractIconsW
0b shell32!SHPrivateExtractIcons
0c shell32!SHDefExtractIconW
0d shell32!CExtractIcon::_ExtractW
0e shell32!CExtractIconBase::Extract
0f shell32!IExtractIcon_Extract
10 shell32!_GetILIndexGivenPXIcon
11 shell32!_GetILIndexFromItem
12 shell32!SHGetIconIndexFromPIDL
13 shell32!MapIDListToIconILIndex
14 shell32!CLoadSystemIconTask::InternalResumeRT
15 shell32!CRunnableTask::Run
16 shell32!CShellTask::TT_Run
17 shell32!CShellTaskThread::ThreadProc
18 shell32!CShellTaskThread::s_ThreadProc
19 shlwapi!ExecuteWorkItemThreadProc
1a ntdll!RtlpTpWorkCallback
1b ntdll!TppWorkerThread
1c kernel32!BaseThreadInitThunk
1d ntdll!RtlUserThreadStart

7 Id: 1344.139c Suspend: 1 Teb: 000007ff`ffffac000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 KERNELBASE!WaitForMultipleObjectsEx
02 kernel32!WaitForMultipleObjectsExImplementation
03 user32!RealMsgWaitForMultipleObjectsEx
04 user32!MsgWaitForMultipleObjectsEx
05 user32!MsgWaitForMultipleObjects
06 shell32!CShellTaskScheduler::_TT_MsgWaitForMultipleObjects
07 shell32!CShellTaskScheduler::TT_TransitionThreadToRunningOrTerminating
08 shell32!CShellTaskThread::ThreadProc
09 shell32!CShellTaskThread::s_ThreadProc
0a shlwapi!ExecuteWorkItemThreadProc
0b ntdll!RtlpTpWorkCallback
0c ntdll!TppWorkerThread
0d kernel32!BaseThreadInitThunk
0e ntdll!RtlUserThreadStart

8 Id: 1344.1b80 Suspend: 1 Teb: 000007ff`ffffaa000 Unfrozen
# Call Site
00 ntdll!ZwDelayExecution
01 KERNELBASE!SleepEx
02 ole32!CROIDTable::WorkerThreadLoop
03 ole32!CRpcThread::WorkerLoop
04 ole32!CRpcThreadCache::RpcWorkerThreadEntry
05 kernel32!BaseThreadInitThunk
06 ntdll!RtlUserThreadStart

9 Id: 1344.eb8 Suspend: 1 Teb: 000007ff`ffffa8000 Unfrozen
# Call Site
00 ntdll!ZwAlpcSendWaitReceivePort
01 rpcrt4!LRPC_CCALL::SendReceive
```

```

02 rpcrt4!NdrpClientCall13
03 rpcrt4!NdrClientCall13
04 winspool!RpcFindNextPrinterChangeNotification
05 winspool!FindNextPrinterChangeNotification
06 prncache!PrintCache::Listeners::Listener::ProcessWait
07 prncache!PrintCache::Listeners::Listener::ProcessWaitCB
08 ntdll!TppWaitpExecuteCallback
09 ntdll!TppWorkerThread
0a kernel32!BaseThreadInitThunk
0b ntdll!RtlUserThreadStart

```

```

10 Id: 1344.f98 Suspend: 1 Teb: 000007ff`ffffa6000 Unfrozen
# Call Site
00 ntdll!RtlUserThreadStart

```

// 12th dump

```

. 0 Id: 1344.1ca4 Suspend: 1 Teb: 000007ff`ffffdd000 Unfrozen
# Call Site
00 usp10!otlChainingLookup::apply
01 usp10!ApplyLookup
02 usp10!ApplyFeatures
03 usp10!SubstituteOtlGlyphs
04 usp10!GenericEngineGetGlyphs
05 usp10!ShlShape
06 usp10!ScriptShape
07 usp10!RenderItemNoFallback
08 usp10!RenderItemWithFallback
09 usp10!RenderItem
0a usp10!ScriptStringAnalyzeGlyphs
0b usp10!ScriptStringAnalyse
0c lpk!LpkCharsetDraw
0d lpk!LpkDrawTextEx
0e user32!DT_DrawStr
0f user32!DT_DrawJustifiedLine
10 user32!DrawTextExWorker
11 user32!DrawTextW
12 comctl32!CLVView::_ComputeLabelSizeWorker
13 comctl32!CLVView::v_RecomputeLabelSize
14 comctl32!CLVList View::v_DrawItem
15 comctl32!CLVDrawItemManager::DrawItem
16 comctl32!CLVDrawManager::_PaintItems
17 comctl32!CLVDrawManager::_PaintWorkArea
18 comctl32!CLVDrawManager::_OnPaintWorkAreas
19 comctl32!CLVDrawManager::_OnPaint
1a comctl32!CListView::WndProc
1b comctl32!CListView::s_WndProc
1c user32!UserCallWinProcCheckWow
1d user32!CallWindowProcAorW
1e user32!CallWindowProcW
1f comctl32!CallOriginalWndProc
20 comctl32!CallNextSubclassProc
21 comctl32!DefSubclassProc
22 shell32!DefSubclassProc
23 shell32!CListViewHost::s_ListViewSubclassWndProc

```

```
24 comctl32!CallNextSubclassProc
25 comctl32!MasterSubclassProc
26 user32!UserCallWinProcCheckWow
27 user32!DispatchClientMessage
28 user32!_fnDWORD
29 ntdll!KiUserCallbackDispatcherContinue
2a user32!NtUserDispatchMessage
2b user32!DispatchMessageWorker
2c user32!IsDialogMessageW
2d comctl32!Prop_IsDialogMessage
2e comctl32!_RealPropertySheet
2f comctl32!_PropertySheet
30 comdlg32!Print_InvokePropertySheets
31 comdlg32!PrintDlgExX
32 comdlg32!PrintDlgExW
33 notepad!GetPrinterDCviaDialog
34 notepad!PrintIt
35 notepad!NPCommand
36 notepad!NPWndProc
37 user32!UserCallWinProcCheckWow
38 user32!DispatchMessageWorker
39 notepad!WinMain
3a notepad!DisplayNonGenuineDlgWorker
3b kernel32!BaseThreadInitThunk
3c ntdll!RtlUserThreadStart

1 Id: 1344.1ab0 Suspend: 1 Teb: 000007ff`ffffdb000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 ntdll!TppWaiterpThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

2 Id: 1344.1638 Suspend: 1 Teb: 000007ff`ffffd9000 Unfrozen
# Call Site
00 ntdll!NtUnmapViewOfSection
01 KERNELBASE!FreeLibrary
02 user32!PrivateExtractIconsW
03 shell32!SHPrivateExtractIcons
04 shell32!SHDefExtractIconW
05 shell32!CEExtractIcon::_ExtractW
06 shell32!CEExtractIconBase::Extract
07 shell32!IExtractIcon_Extract
08 shell32!_GetILIndexGivenPXIcon
09 shell32!_GetILIndexFromItem
0a shell32!SHGetIconIndexFromPIDL
0b shell32!MapIDListToIconILIndex
0c shell32!CLoadSystemIconTask::InternalResumeRT
0d shell32!CRunnableTask::Run
0e shell32!CShellTask::TT_Run
0f shell32!CShellTaskThread::ThreadProc
10 shell32!CShellTaskThread::s_ThreadProc
11 shlwapi!ExecuteWorkItemThreadProc
12 ntdll!RtlpTpWorkCallback
13 ntdll!TppWorkerThread
```

```
14 kernel32!BaseThreadInitThunk
15 ntdll!RtlUserThreadStart

3 Id: 1344.830 Suspend: 1 Teb: 000007ff`ffffd7000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 KERNELBASE!WaitForMultipleObjectsEx
02 kernel32!WaitForMultipleObjects
03 prncache!PrintCache::Store::CacheStore::RegistryMonitor::MonitorRegistryChanges
04 prncache!PrintCache::Store::CacheStore::RegistryMonitor::MonitorRegistryChangesThreadProc
05 kernel32!BaseThreadInitThunk
06 ntdll!RtlUserThreadStart

4 Id: 1344.1edc Suspend: 1 Teb: 000007ff`ffffd5000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 KERNELBASE!WaitForMultipleObjectsEx
02 kernel32!WaitForMultipleObjectsExImplementation
03 user32!RealMsgWaitForMultipleObjectsEx
04 user32!MsgWaitForMultipleObjectsEx
05 user32!MsgWaitForMultipleObjects
06 shell32!CShellTaskScheduler::_TT_MsgWaitForMultipleObjects
07 shell32!CShellTaskScheduler::TT_TransitionThreadToRunningOrTerminating
08 shell32!CShellTaskThread::ThreadProc
09 shell32!CShellTaskThread::s_ThreadProc
0a shlwapi!ExecuteWorkItemThreadProc
0b ntdll!RtlpTpWorkCallback
0c ntdll!TppWorkerThread
0d kernel32!BaseThreadInitThunk
0e ntdll!RtlUserThreadStart

5 Id: 1344.1b44 Suspend: 1 Teb: 000007ff`ffffd3000 Unfrozen
# Call Site
00 ntdll!NtWaitForMultipleObjects
01 KERNELBASE!WaitForMultipleObjectsEx
02 kernel32!WaitForMultipleObjectsExImplementation
03 user32!RealMsgWaitForMultipleObjectsEx
04 user32!MsgWaitForMultipleObjectsEx
05 user32!MsgWaitForMultipleObjects
06 shell32!CShellTaskScheduler::_TT_MsgWaitForMultipleObjects
07 shell32!CShellTaskScheduler::TT_TransitionThreadToRunningOrTerminating
08 shell32!CShellTaskThread::ThreadProc
09 shell32!CShellTaskThread::s_ThreadProc
0a shlwapi!ExecuteWorkItemThreadProc
0b ntdll!RtlpTpWorkCallback
0c ntdll!TppWorkerThread
0d kernel32!BaseThreadInitThunk
0e ntdll!RtlUserThreadStart

6 Id: 1344.1d9c Suspend: 1 Teb: 000007ff`ffffae000 Unfrozen
# Call Site
00 gdi32!ZwGdiSetDIBitsToDeviceInternal
01 gdi32!SetDIBitsToDevice
02 gdi32!SetDIBits
03 user32!BitmapFromDIB
```

```
04 user32!ConvertDIBBitmap
05 user32!ConvertDIBIcon
06 user32!CreateIconFromResourceEx
07 user32!PrivateEnumProc
08 kernel32!EnumResourceNamesInternal
09 kernel32!EnumResourceNamesExW
0a user32!PrivateExtractIconsW
0b shell32!SHPrivateExtractIcons
0c shell32!SHDefExtractIconW
0d shell32!CExtractIcon::_ExtractW
0e shell32!CExtractIconBase::Extract
0f shell32!IExtractIcon_Extract
10 shell32!_GetILIndexGivenPXIcon
11 shell32!_GetILIndexFromItem
12 shell32!SHGetIconIndexFromPIDL
13 shell32!MapIDListToIconILIndex
14 shell32!CLoadSystemIconTask::InternalResumeRT
15 shell32!CRunnableTask::Run
16 shell32!CShellTask::TT_Run
17 shell32!CShellTaskThread::ThreadProc
18 shell32!CShellTaskThread::s_ThreadProc
19 shlwapi!ExecuteWorkItemThreadProc
1a ntdll!RtlpTpWorkCallback
1b ntdll!TppWorkerThread
1c kernel32!BaseThreadInitThunk
1d ntdll!RtlUserThreadStart
```

```
7 Id: 1344.139c Suspend: 1 Teb: 000007ff`ffffac000 Unfrozen
# Call Site
00 ntdll!ZwMapViewOfSection
01 KERNELBASE!BasepLoadLibraryAsDataFileInternal
02 KERNELBASE!LoadLibraryExW
03 user32!PrivateExtractIconsW
04 shell32!SHPrivateExtractIcons
05 shell32!SHDefExtractIconW
06 shell32!CExtractIcon::_ExtractW
07 shell32!CExtractIconBase::Extract
08 shell32!IExtractIcon_Extract
09 shell32!_GetILIndexGivenPXIcon
0a shell32!_GetILIndexFromItem
0b shell32!SHGetIconIndexFromPIDL
0c shell32!MapIDListToIconILIndex
0d shell32!CLoadSystemIconTask::InternalResumeRT
0e shell32!CRunnableTask::Run
0f shell32!CShellTask::TT_Run
10 shell32!CShellTaskThread::ThreadProc
11 shell32!CShellTaskThread::s_ThreadProc
12 shlwapi!ExecuteWorkItemThreadProc
13 ntdll!RtlpTpWorkCallback
14 ntdll!TppWorkerThread
15 kernel32!BaseThreadInitThunk
16 ntdll!RtlUserThreadStart
```

```
8 Id: 1344.1b80 Suspend: 1 Teb: 000007ff`ffffaa000 Unfrozen
# Call Site
00 ntdll!ZwAlpcSendWaitReceivePort
01 rpcrt4!LRPC_CASSOCIATION::AlpcSendWaitReceivePort
02 rpcrt4!LRPC_BASE_CCALL::DoSendReceive
03 rpcrt4!LRPC_BASE_CCALL::SendReceive
04 rpcrt4!NdrClientCall2
05 rpcrt4!NdrClientCall2
06 ole32!CRpcResolver::BulkUpdateOIDs
07 ole32!CROIDTable::ClientBulkUpdateOIDWithPingServer
08 ole32!CROIDTable::WorkerThreadLoop
09 ole32!CRpcThread::WorkerLoop
0a ole32!CRpcThreadCache::RpcWorkerThreadEntry
0b kernel32!BaseThreadInitThunk
0c ntdll!RtlUserThreadStart

9 Id: 1344.eb8 Suspend: 1 Teb: 000007ff`ffffa8000 Unfrozen
# Call Site
00 ntdll!NtWaitForWorkViaWorkerFactory
01 ntdll!TpWorkerThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

10 Id: 1344.f98 Suspend: 1 Teb: 000007ff`ffffa6000 Unfrozen
# Call Site
00 ntdll!NtWaitForWorkViaWorkerFactory
01 ntdll!TpWorkerThread
02 kernel32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

11 Id: 1344.1fb0 Suspend: 1 Teb: 000007ff`ffffa4000 Unfrozen
# Call Site
00 ntdll!RtlUserThreadStart
```

## Exception Module

It is a module or component where the actual exception happened, for example, *ModuleA* from this **Exception Stack Trace** (page 363):

```

9 Id: 1df4.a08 Suspend: -1 Peb: 7fff4000 Unfrozen
ChildEBP RetAddr
1022f5a8 7c90df4a ntdll!KiFastSystemCallRet
1022f5ac 7c8648a2 ntdll!ZwWaitForMultipleObjects+0xc
1022f900 7c83ab50 kernel32!UnhandledExceptionFilter+0x8b9
1022f908 7c839b39 kernel32!BaseThreadStart+0x4d
1022f930 7c9032a8 kernel32!_except_handler3+0x61
1022f954 7c90327a ntdll!ExecuteHandler2+0x26
1022fa04 7c90e48a ntdll!ExecuteHandler+0x24
1022fa04 7c812afb ntdll!KiUserExceptionDispatcher+0xe
1022fd5c 0b82e680 kernel32!RaiseException+0x53
WARNING: Stack unwind information not available. Following frames may be wrong.
1022fd94 0b82d2f2 ModuleA+0x21e640
1022fde8 7753004f ModuleA+0x21d4f2
1022fdfc 7753032f ole32!CClassCache::CDllPathEntry::CanUnload_r1+0x3b
1022ff3c 7753028b ole32!CClassCache::FreeUnused+0x70
1022ff4c 775300b5 ole32!CoFreeUnusedLibrariesEx+0x36
1022ff58 77596af5 ole32!CoFreeUnusedLibraries+0x9
1022ff6c 77566fff ole32!CDllHost::MTAWorkerLoop+0x25
1022ff8c 7752687c ole32!CDllHost::WorkerThread+0xc1
1022ff94 774fe3ee ole32!DLLHostThreadEntry+0xd
1022ffa8 774fe456 ole32!CRpcThread::WorkerLoop+0x1e
1022ffb4 7c80b729 ole32!CRpcThreadCache::RpcWorkerThreadEntry+0x1b
1022ffec 00000000 kernel32!BaseThreadStart+0x37

```

Because we have **Software Exception** (page 875), we can use backward disassembly (**ub** WinDbg command) to check stack trace correctness in the case of stack unwind warnings (like in **Coincidental Symbolic Information** pattern, page 137). Here's another example, for recent MS Paint crash we observed, with *msvcrt* **Exception Module**. However, if we skip it as **Well-Tested Module** (page 1147), the next **Exception Module** candidate is *mspaint*.

```

0:000> kc
Call Site
ntdll!NtWaitForMultipleObjects
KERNELBASE!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjectsExImplementation
kernel32!WerFaultInternal
kernel32!WerReportFault
kernel32!BaseFaultReport
kernel32!UnhandledExceptionFilter
ntdll! ?? ::FNODOBFM::`string'
ntdll! _C_specific_handler
ntdll!RtlpExecuteHandlerForException
ntdll!RtlDispatchException
ntdll!KiUserExceptionDispatch
msvcrt!memcpy
mspaint!CImgWnd::CmdCrop

```

```
mspaint!CPBView::OnImageCrop
mfc42u!_AfxDispatchCmdMsg
mfc42u!CCmdTarget::OnCmdMsg
mfc42u!CView::OnCmdMsg
mspaint!CPBView::OnCmdMsg
mfc42u!CFrameWnd::OnCmdMsg
mspaint!CGenericCommandSite::XGenericCommandSiteCommandHandler::Execute
UIRibbon!CControlUser::_ExecuteOnHandler
UIRibbon!CGenericControlUser::SetValueImpl
UIRibbon!CGenericDataSource::SetValue
UIRibbon!OfficeSpace::DataSource::SetValue
UIRibbon!OfficeSpace::FSControl::SetValue
UIRibbon!NetUI::DeferCycle::ProcessDataBindingPropertyChangeRecords
UIRibbon!NetUI::DeferCycle::HrAddDataBindingPropertyChangeRecord
UIRibbon!NetUI::Binding::SetDataSourceValue
UIRibbon!NetUI::Bindings::OnBindingPropertyChanged
UIRibbon!NetUI::Node::OnPropertyChanged
UIRibbon!FlexUI::Concept::OnPropertyChanged
UIRibbon!NetUI::Node::FExecuteCommand
UIRibbon!FlexUI::ExecuteAction::OnCommand
UIRibbon!NetUI::Node::FExecuteCommand
UIRibbon!NetUI::SimpleButton::OnEvent
UIRibbon!NetUI::Element::_DisplayNodeCallback
UIRibbon!GPCB::xwInvokeDirect
UIRibbon!GPCB::xwInvokeFull
UIRibbon!DUserSendEvent
UIRibbon!NetUI::Element::FireEvent
UIRibbon!NetUI::_FireClickEvent
UIRibbon!NetUI::SimpleButton::OnInput
UIRibbon!NetUI::Element::_DisplayNodeCallback
UIRibbon!GPCB::xwInvokeDirect
UIRibbon!GPCB::xwInvokeFull
UIRibbon!BaseMsgQ::xwProcessNL
UIRibbon!DelayedMsgQ::xwProcessDelayedNL
UIRibbon!ContextLock::~ContextLock
UIRibbon!HWndContainer::xdHandleMessage
UIRibbon!ExtraInfoWndProc
user32!UserCallWinProcCheckWow
user32!DispatchMessageWorker
mfc42u!CWinThread::PumpMessage
mfc42u!CWinThread::Run
mfc42u!AfxWinMain
mspaint!LDunscale
kernel32!BaseThreadInitThunk
ntdll!RtlUserThreadStart
```

## Exception Stack Trace

This is a pattern that we see in many pattern interaction case studies published in Memory Dump Analysis Anthology volumes. We can also call it **Exception Thread**. It is **Stack Trace** (page 926) that has exception processing functions, for example:

```

9 Id: 1df4.a08 Suspend: -1 Teb: 7fff4000 Unfrozen
ChildEBP RetAddr
1022f5a8 7c90df4a ntdll!KiFastSystemCallRet
1022f5ac 7c8648a2 ntdll!ZwWaitForMultipleObjects+0xc
1022f900 7c83ab50 kernel32!UnhandledExceptionFilter+0x8b9
1022f908 7c839b39 kernel32!BaseThreadStart+0x4d
1022f930 7c9032a8 kernel32!_except_handler3+0x61
1022f954 7c90327a ntdll!ExecuteHandler2+0x26
1022fa04 7c90e48a ntdll!ExecuteHandler+0x24
1022fa04 7c812afb ntdll!KiUserExceptionDispatcher+0xe
1022fd5c 0b82e680 kernel32!RaiseException+0x53
1022fd94 0b82d2f2 D11A+0x21e640
1022fde8 7753004f D11A+0x21d4f2
1022fdfc 7753032f ole32!CClassCache::CDllPathEntry::CanUnload_r1+0x3b
1022ff3c 7753028b ole32!CClassCache::FreeUnused+0x70
1022ff4c 775300b5 ole32!CoFreeUnusedLibrariesEx+0x36
1022ff58 77596af5 ole32!CoFreeUnusedLibraries+0x9
1022ff6c 77566ff9 ole32!CDllHost::MTAWorkerLoop+0x25
1022ff8c 7752687c ole32!CDllHost::WorkerThread+0xc1
1022ff94 774fe3ee ole32!DLLHostThreadEntry+0xd
1022ffa8 774fe456 ole32!CRpcThread::WorkerLoop+0x1e
1022ffb4 7c80b729 ole32!CRpcThreadCache::RpcWorkerThreadEntry+0x1b
1022ffec 00000000 kernel32!BaseThreadStart+0x37

```

Such exceptions can be detected by the default analysis command (for example, `!analyze -v` WinDbg command) or by inspecting **Stack Trace Collection** (page 943). However, if we don't see any exception thread, it doesn't mean there were no exceptions. There could be **Hidden Exceptions** (page 455) on raw stack data.

In our case we can get the exception information by looking at parameters to a unhandled exception filter:

```

0:009> kv 3
ChildEBP RetAddr Args to Child
1022f5a8 7c90df4a 7c8648a2 00000002 1022f730 ntdll!KiFastSystemCallRet
1022f5ac 7c8648a2 00000002 1022f730 00000001 ntdll!ZwWaitForMultipleObjects+0xc
1022f900 7c83ab50 1022f928 7c839b39 1022f930 kernel32!UnhandledExceptionFilter+0x8b9

```

```

0:009> .exptr 1022f928

----- Exception record at 1022fa1c:
ExceptionAddress: 7c812afb (kernel32!RaiseException+0x00000053)
ExceptionCode: e06d7363 (C++ EH exception)
ExceptionFlags: 00000001
NumberParameters: 3
Parameter[0]: 19930520
Parameter[1]: 1022fda4
Parameter[2]: 0b985074
pExceptionObject: 1022fda4
_s_ThrowInfo : 0b985074

----- Context record at 1022fa3c:
eax=1022fd0c ebx=00000001 ecx=00000000 edx=1022fda4 esi=1022fd94 edi=77606068
eip=7c812afb esp=1022fd08 ebp=1022fd5c iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
kernel32!RaiseException+0x53:
7c812afb 5e pop esi

```

## Comments

---

We can use Raymond Chen's technique to find out more about the type of C++ exception<sup>70</sup>.

This pattern example is taken from Software Diagnostics Services case study<sup>71</sup>.

---

<sup>70</sup> <http://blogs.msdn.com/b/oldnewthing/archive/2010/07/30/10044061.aspx>

<sup>71</sup> <http://www.patterndiagnostics.com/case-study>

## Execution Residue

### Linux

This is a Linux variant of **Execution Residue** pattern previously described for Mac OS X (page 367) and Windows (page 371) platforms. This is symbolic information left in a stack region including ASCII and UNICODE fragments or pointers to them, for example, return addresses from past function calls:

```
(gdb) bt
#0  0x00000000004431f1 in nanosleep ()
#1  0x00000000004430c0 in sleep ()
#2  0x0000000000400771 in procNE() ()
#3  0x00000000004007aa in bar_two() ()
#4  0x00000000004007b5 in foo_two() ()
#5  0x00000000004007c8 in thread_two(void*) ()
#6  0x00000000004140f0 in start_thread (arg=<optimized out>
at pthread_create.c:304
#7  0x0000000000445879 in clone ()
#8  0x0000000000000000 in ?? ()
```

```
(gdb) x/512a $rsp-2000
0x7f4cacc42360: 0x0 0x0
0x7f4cacc42370: 0x0 0x0
0x7f4cacc42380: 0x0 0x0
0x7f4cacc42390: 0x0 0x0
[...]
0x7f4cacc42830: 0x0 0x0
0x7f4cacc42840: 0x0 0x0
0x7f4cacc42850: 0x0 0x0
0x7f4cacc42860: 0x7f4cacc42870 0x4005af <_Z6work_8v+9>
0x7f4cacc42870: 0x7f4cacc42880 0x4005ba <_Z6work_7v+9>
0x7f4cacc42880: 0x7f4cacc42890 0x4005c5 <_Z6work_6v+9>
0x7f4cacc42890: 0x7f4cacc428a0 0x4005d0 <_Z6work_5v+9>
0x7f4cacc428a0: 0x7f4cacc428b0 0x4005db <_Z6work_4v+9>
0x7f4cacc428b0: 0x7f4cacc428c0 0x4005e6 <_Z6work_3v+9>
0x7f4cacc428c0: 0x7f4cacc428d0 0x4005f1 <_Z6work_2v+9>
0x7f4cacc428d0: 0x7f4cacc428e0 0x4005fc <_Z6work_1v+9>
0x7f4cacc428e0: 0x7f4cacc42cf0 0x40060e <_Z4workv+16>
0x7f4cacc428f0: 0x0 0x0
0x7f4cacc42900: 0x0 0x0
0x7f4cacc42910: 0x0 0x0
[...]
0x7f4cacc42af0: 0x0 0x0
0x7f4cacc42b00: 0x0 0x0
0x7f4cacc42b10: 0x0 0x0
0x7f4cacc42b20: 0x0 0x4431e6 <nanosleep+38>
0x7f4cacc42b30: 0x0 0x4430c0 <sleep+224>
0x7f4cacc42b40: 0x0 0x0
0x7f4cacc42b50: 0x0 0x0
0x7f4cacc42b60: 0x0 0x0
0x7f4cacc42b70: 0x0 0x0
[...]
0x7f4cacc42cb0: 0x0 0x0
```

```
0x7f4cacc42cc0: 0x0 0x0
0x7f4cacc42cd0: 0x0 0x0
0x7f4cacc42ce0: 0xfffffed2 0x3ad3affa
0x7f4cacc42cf0: 0x7f4cacc42d00 0x0
0x7f4cacc42d00: 0x7f4cacc42d20 0x49c740 <default_attr>
0x7f4cacc42d10: 0x7f4cacc439c0 0x400771 <_Z6procNEv+19>
0x7f4cacc42d20: 0x7f4cacc42d30 0x4007aa <_Z7bar_twov+9>
0x7f4cacc42d30: 0x7f4cacc42d40 0x4007b5 <_Z7foo_twov+9>
0x7f4cacc42d40: 0x7f4cacc42d60 0x4007c8 <_Z10thread_twopv+17>
0x7f4cacc42d50: 0x0 0x0
0x7f4cacc42d60: 0x0 0x4140f0 <start_thread+208>
0x7f4cacc42d70: 0x0 0x7f4cacc43700
0x7f4cacc42d80: 0x0 0x0
0x7f4cacc42d90: 0x0 0x0
[...]
```

However, supposed return addresses need to be checked for **Coincidental Symbolic Information** (page 134) pattern.

## Mac OS X

This is a Mac OS X / GDB counterpart to **Execution Residue** pattern on Windows platforms (page 371):

```
(gdb) bt
#0 0x000007fff8616e82a in __kill ()
#1 0x000007fff8fab9a9c in abort ()
#2 0x0000000010269dc29 in bar_5 ()
#3 0x0000000010269dc39 in bar_4 ()
#4 0x0000000010269dc49 in bar_3 ()
#5 0x0000000010269dc59 in bar_2 ()
#6 0x0000000010269dc69 in bar_1 ()
#7 0x0000000010269dc79 in bar ()
#8 0x0000000010269dca0 in main (argc=1, argv=0x7fff6229cb00)

(gdb) x $rsp
0x7fff6229ca38: 0x8fab9a9c

(gdb) x/1000a 0x7fff6229c000
0x7fff6229c000: 0x7fff8947b000 0x7fff8947b570
0x7fff6229c010: 0x4f3ee10c 0x7fff90cb0000
0x7fff6229c020: 0x7fff90cb04d0 0x4e938b16
[...]
0x7fff6229c5f0: 0x7fff622d8d80 0x10269d640
0x7fff6229c600: 0x7fff6229cad0 0x7fff622a460b
0x7fff6229c610: 0x100000000 0x269d000
0x7fff6229c620: 0x7fff6229c630 0x10269db59 <foo_8+9>
0x7fff6229c630: 0x7fff6229c640 0x10269db69 <foo_7+9>
0x7fff6229c640: 0x7fff6229c650 0x10269db79 <foo_6+9>
0x7fff6229c650: 0x7fff6229c660 0x10269db89 <foo_5+9>
0x7fff6229c660: 0x7fff6229c670 0x10269db99 <foo_4+9>
0x7fff6229c670: 0x7fff6229c680 0x10269dba9 <foo_3+9>
0x7fff6229c680: 0x7fff6229c690 0x10269dbb9 <foo_2+9>
0x7fff6229c690: 0x7fff6229c6a0 0x10269dbc9 <foo_1+9>
0x7fff6229c6a0: 0x7fff6229cac0 0x10269dbe <foo+30>
0x7fff6229c6b0: 0x0 0x0
0x7fff6229c6c0: 0x0 0x0
[...]
0x7fff6229c8d0: 0x7fff6229c960 0x7fff622b49cd
0x7fff6229c8e0: 0x10269f05c 0x0
0x7fff6229c8f0: 0x7fff622c465c 0x7fff8a31e5c0 <_Z21dyldGlobalLockReleasev>
0x7fff6229c900: 0x7fff8fab99eb <abort> 0x10269f05c
0x7fff6229c910: 0x1010000000000000 0x7fff622d2110
0x7fff6229c920: 0x7fff622d8d80 0x10269f078
0x7fff6229c930: 0x7fff622daac8 0x18
0x7fff6229c940: 0x0 0x0
0x7fff6229c950: 0x10269e030 0x0
0x7fff6229c960: 0x7fff6229c980 0x7fff622a1922
0x7fff6229c970: 0x0 0x0
0x7fff6229c980: 0x7fff6229ca50 0x7fff8a31e716 <dyld_stub_binder_+13>
0x7fff6229c990: 0x1 0x7fff6229cb00
0x7fff6229c9a0: 0x7fff6229cb10 0xe223ea612ddc10b7
0x7fff6229c9b0: 0x8 0x0
0x7fff6229c9c0: 0xe223ea612ddc10b7 0x0
```

```

0x7fff6229c9d0: 0x0 0x0
0x7fff6229c9e0: 0x585f5f00474e414c 0x20435058005f4350
0x7fff6229c9f0: 0x0 0x0
0x7fff6229ca00: 0x0 0x0
0x7fff6229ca10: 0x0 0x0
0x7fff6229ca20: 0x0 0x0
0x7fff6229ca30: 0x7fff6229ca60 0x7fff8fab9a9c <abort+177>
0x7fff6229ca40: 0x0 0x0
0x7fff6229ca50: 0x7fffffffdf 0x0
0x7fff6229ca60: 0x7fff6229ca70 0x10269dc29 <bar_5+9>
0x7fff6229ca70: 0x7fff6229ca80 0x10269dc39 <bar_4+9>
0x7fff6229ca80: 0x7fff6229ca90 0x10269dc49 <bar_3+9>
0x7fff6229ca90: 0x7fff6229caa0 0x10269dc59 <bar_2+9>
0x7fff6229caa0: 0x7fff6229cab0 0x10269dc69 <bar_1+9>
0x7fff6229cab0: 0x7fff6229cac0 0x10269dc79 <bar+9>
0x7fff6229cac0: 0x7fff6229cae0 0x10269dca0 <main+32>
0x7fff6229cad0: 0x7fff6229cb00 0x1
0x7fff6229cae0: 0x7fff6229caf0 0x10269db34 <start+52>
0x7fff6229caf0: 0x0 0x1
0x7fff6229cb00: 0x7fff6229cc48 0x0
0x7fff6229cb10: 0x7fff6229ccae 0x7fff6229ccca
[...]

```

Here's the source code of the modeling application:

```

#define def_call(name,x,y) void name##_##x() { name##_##y(); }
#define def_final(name,x) void name##_##x() { }
#define def_final_abort(name,x) void name##_##x() { abort(); }
#define def_init(name,y) void name() { name##_##y(); }
#define def_init_alloc(name,y,size) void name() { int arr[size]; name##_##y(); *arr=0; }

def_final(foo,9)
def_call(foo,8,9)
def_call(foo,7,8)
def_call(foo,6,7)
def_call(foo,5,6)
def_call(foo,4,5)
def_call(foo,3,4)
def_call(foo,2,3)
def_call(foo,1,2)
def_init_alloc(foo,1,256)
def_final_abort(bar,5)
def_call(bar,4,5)
def_call(bar,3,4)
def_call(bar,2,3)
def_call(bar,1,2)
def_init(bar,1)

int main(int argc, const char * argv[])
{
    foo();
    bar();
}

```

## Windows

### Managed Space

This is a .NET counterpart to unmanaged and native code **Execution Residue** pattern. Here we can use SOS extension **!DumpStack** command for call level execution residue (see **Caller-n-Callee** pattern example, page 111) and **!DumpStackObjects** (**!dso**) for managed object references found on a raw stack:

```
0:011> !DumpStackObjects
OS Thread Id: 0x8e0 (11)
ESP/REG Object Name
09efe4b8 0a2571bc System.Threading.Thread
09efe538 0a1ffddc System.Threading.Thread
09efe844 0a1ffb8 UserQuery
09efe974 0a1ffc0 System.Signature
09fea20 0a1ffd10 System.RuntimeTypeHandle[]
09feae8 08985e14 System.Object[] (System.Reflection.AssemblyName[])
09feaec 0a1ffa78 System.Diagnostics.Stopwatch
09feaf0 0a1ffa6c LINQPad.Extensibility.DataContext.QueryExecutionManager
09feafc 0a1ffb8 UserQuery
09feb00 0a1ffa58 System.RuntimeType
09feb04 08995474 LINQPad.ObjectGraph.Formatters.XhtmlWriter
09feb08 08985dfc System.Reflection.Assembly
09feb0c 08985dc8 LINQPad.ExecutionModel.ResultData
09feb10 08984548 LINQPad.ExecutionModel.Server
09febdc 0a1ffbe8 System.Reflection.RuntimeMethodInfo
09febe0 0a1fcfc4 LINQPad.ExecutionModel.ConsoleTextReader
09febe4 0a1fcddc System.IO.StreamReader+NullStreamReader
09febe8 0899544c System.IO.TextWriter+SyncTextWriter
09febec 08985efc System.Reflection.AssemblyName
09febfb0 08985d4c System.String C:\Users\Training\AppData\Local\Temp\LINQPad\fcamvgpa
09fec30 08984548 LINQPad.ExecutionModel.Server
09feedc 08985910 System.Threading.ThreadStart

0:011> !DumpObj 0a2571bc
Name: System.Threading.Thread
MethodTable: 790fe704
EEClass: 790fe694
Size: 56(0x38) bytes
(C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
Fields:
MT Field Offset Type VT Attr Value Name
7910a5c4 4000634 4 ...Contexts.Context 0 instance 08980ee4 m_Context
79104de8 4000635 8 ...ExecutionContext 0 instance 00000000 m_ExecutionContext
790fd8c4 4000636 c System.String 0 instance 00000000 m_Name
790fe3b0 4000637 10 System.Delegate 0 instance 00000000 m_Delegate
79130084 4000638 14 System.Object[][] 0 instance 00000000 m_ThreadStaticsBuckets
7912d7c0 4000639 18 System.Int32[] 0 instance 00000000 m_ThreadStaticsBits
791028f4 400063a 1c ...ation.CultureInfo 0 instance 00000000 m_CurrentCulture
791028f4 400063b 20 ...ation.CultureInfo 0 instance 00000000 m_CurrentUICulture
790fd0f0 400063c 24 System.Object 0 instance 00000000 m_ThreadStartArg
791016bc 400063d 28 System.IntPtr 1 instance 8f69280 DONT_USE_InternalThread
79102290 400063e 2c System.Int32 1 instance 2 m_Priority
```

```
79102290 400063f 30 System.Int32 1 instance 11 m_ManagedThreadId  
7910a7a8 4000640 168 ...LocalDataStoreMgr 0 shared static s_LocalDataStoreMgr  
>> Domain:Value 000710a8:06c42ef4 08e65d48:00000000 <<  
790fd0f0 4000641 16c System.Object 0 shared static s_SyncObject  
>> Domain:Value 000710a8:017b25d8 08e65d48:0898381c <<
```

Although unmanaged, CLR and JIT-code residue is useful for analysis, for example, as shown in [Handled Exception](#) (page 428) pattern examples.

---

## Comments

Sometimes, if no exceptions are found on raw stack we can search all runtime types, for example:

```
0:000> !DumpRuntimeTypes  
[...]  
098b93e8 05179888 05622254 CustomException  
[...]  
09bcd368 ? 6969470c System.NullReferenceException  
[...]
```

## Unmanaged Space

For the pattern about **NULL Code Pointer** (page 750), we created a simple program that crashes when we pass a NULL thread procedure pointer to *CreateThread* function. We might expect to see little in the raw stack data<sup>72</sup> because there was no user-supplied thread code. In reality, if we dump it we will see a lot of symbolic information for code and data including ASCII and UNICODE fragments that we call **Execution Residue** patterns, and one of them is **Exception Handling Residue**<sup>73</sup> we can use to check for **Hidden Exceptions** (page 457) and differentiate between 1st and 2nd chance exceptions<sup>74</sup>. Code residues are very powerful in reconstructing stack traces manually<sup>75</sup> or looking for partial stack traces and **Historical Information** (page 483).

To show typical execution residues we created another small program with two additional threads based on Visual Studio Win32 project. After we dismiss *About* box we create the first thread, and then we crash the process when creating the second thread because of the NULL thread procedure:

```
typedef DWORD (WINAPI *THREADPROC)(PVOID);

DWORD WINAPI ThreadProc(PVOID pvParam)
{
    for (unsigned int i = 0xFFFFFFFF; i; --i);
    return 0;
}

// Message handler for about box.
INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    UNREFERENCED_PARAMETER(lParam);
    switch (message)
    {
        case WM_INITDIALOG:
            return (INT_PTR)TRUE;

        case WM_COMMAND:
            if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
            {
                EndDialog(hDlg, LOWORD(wParam));
                THREADPROC thProc = ThreadProc;
                HANDLE hThread = CreateThread(NULL, 0, ThreadProc, 0, 0, NULL);
                CloseHandle(hThread);
                Sleep(1000);
                hThread = CreateThread(NULL, 0, NULL, 0, 0, NULL);
            }
    }
}
```

<sup>72</sup> Raw Stack Dump of All Threads (Process Dump), Memory Dump Analysis Anthology, Volume 1, page 231

<sup>73</sup> This is a pattern we may add in the future

<sup>74</sup> How to Distinguish Between 1st and 2nd Chances, Memory Dump Analysis Anthology, Volume 1, page 109

<sup>75</sup> Manual Stack Trace Reconstruction, Memory Dump Analysis Anthology, Volume 1, page 157

```

        CloseHandle(hThread);
        return (INT_PTR)TRUE;
    }
    break;
}
return (INT_PTR)FALSE;
}

```

When we open the crash dump we see these threads:

```

0:002> ~*kL

0  Id: cb0.9ac Suspend: 1 Teb: 7efdd000 Unfrozen
ChildEBP RetAddr
0012fdf4 00411554 user32!NtUserGetMessage+0x15
0012ff08 00412329 NullThread!wWinMain+0xa4
0012ffb8 0041208d NullThread!_tmainCRTStartup+0x289
0012ffc0 7d4e7d2a NullThread!wWinMainCRTStartup+0xd
0012fff0 00000000 kernel32!BaseProcessStart+0x28

1  Id: cb0.8b4 Suspend: 1 Teb: 7efda000 Unfrozen
ChildEBP RetAddr
01eafea4 7d63f501 ntdll!NtWaitForMultipleObjects+0x15
01eaff48 7d63f988 ntdll!EtwpWaitForMultipleObjectsEx+0xf7
01eaffb8 7d4dfe21 ntdll!EtwpEventPump+0x27f
01eaffec 00000000 kernel32!BaseThreadStart+0x34

2  Id: cb0.ca8 Suspend: 1 Teb: 7efd7000 Unfrozen
ChildEBP RetAddr
0222ffb8 7d4dfe21 NullThread!ThreadProc+0x34
0222ffec 00000000 kernel32!BaseThreadStart+0x34

# 3  Id: cb0.5bc Suspend: 1 Teb: 7efaf000 Unfrozen
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
0236ffb8 7d4dfe21 0x0
0236ffec 00000000 kernel32!BaseThreadStart+0x34

4  Id: cb0.468 Suspend: -1 Teb: 7efac000 Unfrozen
ChildEBP RetAddr
01f7ffb4 7d674807 ntdll!NtTerminateThread+0x12
01f7ffc4 7d66509f ntdll!RtlExitUserThread+0x26
01f7fff4 00000000 ntdll!DbgUiRemoteBreakin+0x41

```

We see our first created thread looping:

```

0:003> ~2s
eax=cbcfc04b5 ebx=00000000 ecx=00000000 edx=00000000 esi=00000000 edi=0222ffb8
eip=00411aa4 esp=0222fee0 ebp=0222ffb8 iopl=0 nv up ei ng nz na po nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000282
NullThread!ThreadProc+0x34:
00411aa4 7402 je NullThread!ThreadProc+0x38 (00411aa8) [br=0]

```

```
0:002> u
NullThread!ThreadProc+0x34:
00411aa4 je      NullThread!ThreadProc+0x38 (00411aa8)
00411aa6 jmp    NullThread!ThreadProc+0x27 (00411a97)
00411aa8 xor    eax,eax
00411aaa pop   edi
00411aab pop   esi
00411aac pop   ebx
00411aad mov    esp,ebp
00411aaaf pop   ebp
```

We might expect it is having very little in its raw stack data but what we see when we dump its stack range from **!teb** command is **Thread Startup Residue** (this is a pattern we may add in the future) where some symbolic information might be **Coincidental** too (**Coincidental Symbolic Information**, page 137):

<pre>0:002&gt; dds 0222f000 02230000 0222f000 00000000 0222f004 00000000 0222f008 00000000 ... 0222f104 00000000 0222f108 00000000 0222f10c 00000000 0222f110 7d621954 ntdll!RtlImageNtHeaderEx+0xee 0222f114 7fdfde000 0222f118 00000000 0222f11c 00000001 0222f120 000000e8 0222f124 004000e8 NullThread!_enc\$textbss\$begin &lt;PERF&gt; (NullThread+0xe8) 0222f128 00000000 0222f12c 0222f114 0222f130 00000000 0222f134 0222fca0 0222f138 7d61f1f8 ntdll!_except_handler3 0222f13c 7d621958 ntdll!RtlpRunTable+0x4a0 0222f140 ffffffff 0222f144 7d621954 ntdll!RtlImageNtHeaderEx+0xee 0222f148 7d6218ab ntdll!RtlImageNtHeader+0x1b 0222f14c 00000001 0222f150 00400000 NullThread!_enc\$textbss\$begin &lt;PERF&gt; (NullThread+0x0) 0222f154 00000000 0222f158 00000000 0222f15c 0222f160 0222f160 004000e8 NullThread!_enc\$textbss\$begin &lt;PERF&gt; (NullThread+0xe8) 0222f164 0222f7bc 0222f168 7d4dffea3 kernel32!ConsoleApp+0xe 0222f16c 00400000 NullThread!_enc\$textbss\$begin &lt;PERF&gt; (NullThread+0x0) 0222f170 7d4dfc77 kernel32!ConDllInitialize+0x1f5 0222f174 00000000 0222f178 7d4dfc8c kernel32!ConDllInitialize+0x20a 0222f17c 00000000 0222f180 00000000 ... 0222f290 00000000 0222f294 0222f2b0 0222f298 7d6256e8 ntdll!bsearch+0x42 0222f29c 00180144 0222f2a0 0222f2b4 0222f2a4 7d625992 ntdll!ARRAY_FITS+0x29 0222f2a8 00000a8c 0222f2ac 00001f1c 0222f2b0 0222f2c0 0222f2b4 0222f2f4 0222f2b8 7d625944 ntdll!RtlpLocateActivationContextSection+0x1da</pre>	<pre>0222f2bc 00001f1c 0222f2c0 000029a8 ... 0222f2e0 536cd652 0222f2e4 0222f334 0222f2e8 7d625b62 ntdll!RtlpFindUnicodeStringInSection+0x7b 0222f2ec 0222f418 0222f2f0 00000000 0222f2f4 0222f324 0222f2f8 7d6257f1 ntdll!RtlpFindNextActivationContextSection+0x64 0222f2fc 00181f1c 0222f300 c0150008 ... 0222f320 7efd7000 0222f324 0222f344 0222f328 7d625cd2 ntdll!RtlFindNextActivationContextSection+0x46 0222f32c 0222f368 0222f330 0222f3a0 0222f334 0222f38c 0222f338 0222f340 0222f33c 00181f1c 0222f340 00000000 0222f344 0222f390 0222f348 7d625ad8 ntdll!RtlFindActivationContextSectionString+0xe1 0222f34c 0222f368 0222f350 0222f3a0 ... 0222f38c 00000a8c 0222f390 0222f454 0222f394 7d626381 ntdll!CsrCaptureMessageMultiUnicodeStringsInPlace+0xa57 0222f398 00000003 0222f39c 00000000 0222f3a0 00181f1c 0222f3a4 0222f418 0222f3a8 0222f3b4 0222f3ac 7d6a0340 ntdll!LdrApiDefaultExtension 0222f3b0 7d6263df ntdll!CsrCaptureMessageMultiUnicodeStringsInPlace+0xb73 0222f3b4 00000040 0222f3b8 00000000 ... 0222f420 00000000 0222f424 0222f458 0222f428 7d625f9a ntdll!CsrCaptureMessageMultiUnicodeStringsInPlace+0x4c1 0222f42c 00020000 0222f430 0222f44c 0222f434 0222f44c 0222f438 0222f44c 0222f43c 00000002</pre>
--	---

0222f440 00000002	0222f6a0 0222f684
0222f444 7d625f9a	0222f6a4 7efd7c00
ntdll!CsrCaptureMessageMultiUnicodeStringsInPlace+0x4c1	0222f6a8 0222fb20
0222f448 00020000	0222f6ac 7d4d89c4 kernel32!_except_handler3
0222f44c 00000000	0222f6b0 7d4df28 kernel32!`string'+0x18
0222f450 00003cfb	0222f6b4 ffffffff
0222f454 0222f5bc	0222f6b8 7d4dff1e kernel32!GetModuleHandleForUnicodeString+0x97
0222f458 0222f4f4	0222f6bc 7d4e001f kernel32!BasepGetModuleHandleExW+0x17f
0222f45c 0222f5bc	0222f6c0 7d4e009f kernel32!BasepGetModuleHandleExW+0x23c
0222f460 7d626290	0222f6c4 00000000
ntdll!RtlDosApplyFileIsolationRedirection_Ustr+0x346	0222f6c8 0222fc08
0222f464 0222f490	0222f6cc 00000001
0222f468 00000000	0222f6d0 ffffffff
0222f46c 0222f69c	0222f6d4 001a0018
0222f470 7d6262f5	0222f6d8 7efd7c00
ntdll!RtlDosApplyFileIsolationRedirection_Ustr+0x3de	0222f6dc 0222fb50
0222f474 0222f510	0222f6e0 00000000
0222f478 7d6a0340 ntdll!LdrApiDefaultExtension	0222f6e4 00000000
0222f47c 7d626290	0222f6e8 00000000
ntdll!RtlDosApplyFileIsolationRedirection_Ustr+0x346	0222f6ec 02080000 oleaut32!_PictSaveEnhMetaFile+0x76
0222f480 00000000	0222f6f0 0222f90c
0222f484 00800000	0222f6f4 02080000 oleaut32!_PictSaveEnhMetaFile+0x76
...	0222f6f8 0222f704
0222f544 00000000	0222f6fc 00000000
0222f548 00000001	0222f700 7d4c0000 kernel32!_imp__NtFsControlFile <PERF>(kernel32+0x0)
0222f54c 7d6a0290 ntdll!LdrpHashTable+0x50	0222f704 00000000
0222f550 00000000	0222f708 02080000 oleaut32!_PictSaveEnhMetaFile+0x76
0222f554 00500000	0222f70c 0222f928
...	0222f710 02080000 oleaut32!_PictSaveEnhMetaFile+0x76
0222f59c 00000000	0222f714 0222f720
0222f5a0 0222f5d4	0222f718 00000000
0222f5a4 7d6251d0 ntdll!LdrUnlockLoaderLock+0x84	0222f71c 7d4c0000 kernel32!_imp__NtFsControlFile <PERF>(kernel32+0x0)
0222f5a8 7d6251d7 ntdll!LdrUnlockLoaderLock+0xad	0222f720 00000000
0222f5ac 00000000	0222f724 00000000
0222f5b0 0222f69c	...
0222f5b4 00000000	0222f7b8 0000f949
0222f5b8 00003cfb	0222f7bc 0222fbf4
0222f5bc 0222f5ac	0222f7c0 7d4dffdd kernel32!_BaseDllInitialize+0x6b
0222f5c0 7d626de0 ntdll!LdrGetD1lHandleEx+0xbe	0222f7c4 00000002
0222f5c4 0222f640	0222f7c8 00000000
0222f5c8 7d61f1f8 ntdll!_except_handler3	0222f7cc 00000000
0222f5cc 7d6251e0 ntdll!`string'+0x74	0222f7d0 7d4dfde4 kernel32!_BaseDllInitialize+0x495
0222f5d0 ffffffff	0222f7d4 00000000
0222f5d4 7d6251d7 ntdll!LdrUnlockLoaderLock+0xad	0222f7d8 7efde000
0222f5d8 7d626fb3 ntdll!LdrGetD1lHandleEx+0x368	0222f7dc 7d4c0000 kernel32!_imp__NtFsControlFile <PERF>(kernel32+0x0)
0222f5dc 00000001	0222f7e0 00000000
0222f5e0 0ca80042	0222f7e4 00000000
0222f5e4 7d626f76 ntdll!LdrGetD1lHandleEx+0x329	...
0222f5e8 00000000	0222f894 01c58ae0
0222f5ec 7d626d0b ntdll!LdrGetD1lHandle	0222f898 0222fac0
0222f5f0 00000002	0222f89c 7d62155b ntdll!RtlAllocateHeap+0x460
0222f5f4 001a0018	0222f8a0 7d61f78c ntdll!RtlAllocateHeap+0xee7
...	0222f8a4 00000000
0222f640 0222f6a8	0222f8a8 0222fc08
0222f644 7d61f1f8 ntdll!_except_handler3	...
0222f648 7d626e60 ntdll!`string'+0xb4	0222f8d8 00000000
0222f64c ffffffff	0222f8dc 7d621954 ntdll!RtlImageNtHeaderEx+0xee
0222f650 7d626f76 ntdll!LdrGetD1lHandleEx+0x329	0222f8e0 0222f9a4
0222f654 7d626d23 ntdll!LdrGetD1lHandle+0x18	0222f8e4 7d614c88 ntdll!\$VProc_ImageExportDirectory+0x2c48
0222f658 00000001	0222f8e8 0222f9a6
...	0222f8ec 7d612040 ntdll!\$VProc_ImageExportDirectory
0222f66c 0222f6b8	0222f8f0 00000221
0222f670 7d4dff0e kernel32!GetModuleHandleForUnicodeString+0x20	0222f8f4 0222f944
0222f674 00000001	0222f8f8 7d627405 ntdll!LdrpSnapThunk+0xc0
0222f678 00000000	0222f8fc 0222f9a6
0222f67c 0222f6d4	0222f900 00000584
0222f680 7d4dff1e kernel32!GetModuleHandleForUnicodeString+0x97	0222f904 7d600000 ntdll!RtlDosPathSeparatorsString <PERF>(ntdll+0x0)
0222f684 00000000	0222f908 7d613678 ntdll!\$VProc_ImageExportDirectory+0x1638
0222f688 7efd7c00	0222f90c 7d614c88 ntdll!\$VProc_ImageExportDirectory+0x2c48
0222f68c 00000002	0222f910 0222f9a4
0222f690 00000001	0222f914 00000001
0222f694 00000000	
0222f698 0222f6f0	
0222f69c 7d4c0000 kernel32!_imp__NtFsControlFile <PERF>(kernel32+0x0)	

0222f918 0222f9a4	0222fa5c 00000000
0222f91c 00000000	0222fa60 0022faa8
0222f920 0222f990	0222fa64 0222fab0
0222f924 7d6000f0 ntdll!RtlDosPathSeparatorsString <PERF>	0222fa68 0222fb0c
(ntdll+0xf0)	0222fa6c 7d627607 ntdll!LdrpGetProcedureAddress+0x274
0222f928 0222f968	0222fa70 7d6a0180 ntdll!LdrpLoaderLock
0222f92c 00000001	0222fa74 7d6275b2 ntdll!LdrpGetProcedureAddress+0xb3
0222f930 0222f9a4	0222fa78 102ce1ac msvcr80d!`string'
0222f934 7d6000f0 ntdll!RtlDosPathSeparatorsString <PERF>	0222fa7c 0222fc08
(ntdll+0xf0)	0222fa80 0000ffff
0222f938 0222f954	0222fa84 0022fb80
0222f93c 00000000	0222fa88 0022fb8a0
0222f940 00000000	0222fa8c 00000003
0222f944 0222fa00	0222fa90 0222fbd4
0222f948 7d62757a ntdll!LdrpGetProcedureAddress+0x189	0222fa94 020215fc oleaut32!DllMain+0x2c
0222f94c 0222f95c	0222fa98 02020000 oleaut32!_imp__RegFlushKey <PERF>
0222f950 00000098	(oleaut32+0x0)
0222f954 00000005	0222fa9c 00000002
0222f958 01c44f48	0222faa0 00000000
0222f95c 0222fb84	0222faa4 00000000
0222f960 7d62155b ntdll!RtlAllocateHeap+0x460	0222faa8 00000002
0222f964 7d61f78c ntdll!RtlAllocateHeap+0xee7	0222faac 0202162d oleaut32!DllMain+0x203
0222f968 00000000	0222fab0 65440000
0222f96c 0000008c	0222fab4 02020000 oleaut32!_imp__RegFlushKey <PERF>
0222f970 00000000	(oleaut32+0x0)
0222f974 7d4d8472 kernel32!\$VProc_ImageExportDirectory+0x6d4e	0222fab8 00000001
0222f978 0222fa1c	0222fabc 00726574
0222f97c 7d627607 ntdll!LdrpGetProcedureAddress+0x274	0222fac0 0222facc
0222f980 7d612040 ntdll!\$VProc_ImageExportDirectory	0222fac4 7d627c2e ntdll!RtlDecodePointer
0222f984 002324f8	0222fac8 00000000
0222f988 7d600000 ntdll!RtlDosPathSeparatorsString <PERF>	0222facc 65440000
(ntdll+0x0)	0222fad0 00000000
0222f98c 0222faa8	0222fad4 00000000
0222f990 000007bb	0222fad8 00726574
0222f994 00221f08	0222fadc 00000005
0222f998 0222f9a4	0222fae0 00000000
0222f99c 7d627c2e ntdll!RtlDecodePointer	0222fae4 1021af95 msvcr80d!_heap_alloc_dbg+0x375
0222f9a0 00000000	0222fae8 002322f0
0222f9a4 74520000	0222faec 00000000
0222f9a8 6365446c	0222faf0 01c40238
0222f9ac 5065646f	0222faf4 0222fa78
0222f9b0 746e696f	0222faf8 7efd7bf8
0222f9b4 00007265	0222fafc 00000020
0222f9b8 7d627c2e ntdll!RtlDecodePointer	0222fb00 7d61f1f8 ntdll!_except_handler3
0222f9bc 00000000	0222fb04 7d6275b8 ntdll!`string'+0xc
...	0222fb08 ffffffff
0222f9f8 01c40640	0222fb0c 7d6275b2 ntdll!LdrpGetProcedureAddress+0xb3
0222f9fc 00000000	0222fb10 00000000
0222fa00 7d6275b2 ntdll!LdrpGetProcedureAddress+0xb3	0222fb14 00000000
0222fa04 7d627772 ntdll!LdrpSnapThunk+0x31c	0222fb18 0222fb48
0222fa08 7d600000 ntdll!RtlDosPathSeparatorsString <PERF>	0222fb1c 00000000
(ntdll+0xe0)	0222fb20 01000000
0222fa0c 0222fa44	0222fb24 00000001
0222fa10 00000000	0222fb28 0222fb50
0222fa14 0222faa8	0222fb2c 7d4dac3a kernel32!GetProcAddress+0x44
0222fa18 00000000	0222fb30 0222fb50
0222fa1c 0222fab0	0222fb34 7d4dac4c kernel32!GetProcAddress+0x5c
0222fa20 00000001	0222fb38 0222fc08
0222fa24 00000001	0222fb3c 00000013
0222fa28 00000000	0222fb40 00000000
0222fa2c 0222fa9c	0222fb44 01c44f40
0222fa30 7d4c00e8 kernel32!_imp__NtFsControlFile <PERF>	0222fb48 01c4015c
(kernel32+0xe8)	0222fb4c 00000098
0222fa34 01c44fe0	0222fb50 01c44f40
0222fa38 00000001	0222fb54 01c44f48
0222fa3c 01c401a0	0222fb58 01c40238
0222fa40 7d4c00e8 kernel32!_imp__NtFsControlFile <PERF>	0222fb5c 10204f9f msvcr80d!_initptd+0x10f
(kernel32+0xe8)	0222fb60 00000098
0222fa44 00110010	0222fb64 00000000
0222fa48 7d4d8478 kernel32!\$VProc_ImageExportDirectory+0x6d54	0222fb68 01c40000
0222fa4c 00000000	0222fb6c 0222f968
0222fa50 0222fb0c	0222fb70 7d4c0000 kernel32!_imp__NtFsControlFile <PERF>
0222fa54 7d62757a ntdll!LdrpGetProcedureAddress+0x189	(kernel32+0x0)
0222fa58 7d600000 ntdll!RtlDosPathSeparatorsString <PERF>	0222fb74 000000ca8
(ntdll+0x0)	0222fb78 4b405064 msctf!g_tlistim

0222fb7c 0222fbb8	0222fc84 00000000
0222fb80 4b3c384f msctf!CTimList::Leave+0x6	0222fc88 0022f8a0
0222fb84 4b3c14d7 msctf!CTimList::IsThreadId+0x5a	0222fc8c 0202155c oleaut32!_DllMainCRTStartup
0222fb88 00000ca8	0222fc90 7efde000
0222fb8c 4b405064 msctf!g_timlist	0222fc94 7d6a01f4 ntdll!PebLdr+0x14
0222fb90 4b3c0000 msctf!_imp__CheckTokenMembership <PERF>	0222fc98 0222fc2c
(msctf+0x0)	0222fc9c 00000000
0222fb94 01c70000	0222fca0 0222fcfc
0222fb98 00000000	0222fca4 7d61f1f8 ntdll!_except_handler3
0222fb9c 4b405064 msctf!g_timlist	0222fca8 7d628148 ntdll!`string'+0xac
0222fbba 0222fb88	0222fcac ffffffff
0222fba4 7d4dfd40 kernel32!FlsSetValue+0xc7	0222fcb0 7d62821c ntdll!LdrpInitializeThread+0x18f
0222fbba 0222fca0	0222fcb4 7d61e299 ntdll!ZwTestAlert+0x15
0222fbac 4b401bdb msctf!_except_handler3	0222fcb8 7d628088 ntdll!_LdrpInitialize+0x1de
0222fbba 4b3c14e0 msctf!`string'+0x78	0222fcbc 0222fd20
0222fb4 0222fbd4	0222fcc0 00000000
0222fb8 0022f8a0	...
0222fbcc 00000001	0222fcfc 0222ffec
0222fb0 00000000	0222fd00 7d61f1f8 ntdll!_except_handler3
0222fb4 00000000	0222fd04 7d628090 ntdll!`string'+0xfc
0222fb8 0222fc80	0222fd08 ffffffff
0222fbcc 0022f8a0	0222fd0c 7d628088 ntdll!_LdrpInitialize+0x1de
0222fb0 0000156f	0222fd10 7d61ce0d ntdll!NtContinue+0x12
0222fb4 0222fbf4	0222fd14 7d61e9b2 ntdll!KiUserApcDispatcher+0x3a
0222fb8 002115a4 oleaut32!_DllMainCRTStartup+0x52	0222fd18 0222fd20
0222fbdc 002020000 oleaut32!_imp__RegFlushKey <PERF>	0222fd1c 00000001
(oleaut32+0x0)	0222fd20 0001002f
0222fbe0 00000002	...
0222fbe4 00000000	0222fdc8 00000000
0222fbe8 00000000	0222fdcc 00000000
0222fbec 0222fc08	0222fd0 00411032 NullThread!ILT+45(?ThreadProcYGKPAXZ)
0222fbf0 00000001	0222fd4 00000000
0222fbf4 0222fc14	0222fd8 7d4d1504 kernel32!BaseThreadStartThunk
0222fbf8 7d610024 ntdll!LdrpCallInitRoutine+0x14	0222fdc 00000023
0222fbfc 00200000 oleaut32!_imp__RegFlushKey <PERF>	0222fde0 00000202
(oleaut32+0x0)	...
0222fc00 00000001	0222ffb4 cccccccc
0222fc04 00000000	0222ffb8 0222ffec
0222fc08 00000001	0222ffbcc 7d4dfe21 kernel32!BaseThreadStart+0x34
0222fc0c 00000000	0222ffc0 00000000
0222fc10 0022f8a0	0222ffc4 00000000
0222fc14 00000001	0222ffc8 00000000
0222fc18 00000000	0222ffcc 00000000
0222fc1c 0222fcbb	0222ffd0 00000000
0222fc20 7d62822e ntdll!LdrpInitializeThread+0x1a5	0222ffd4 0222fc4
0222fc24 7d6a0180 ntdll!LdrpLoaderLock	0222ffd8 00000000
0222fc28 7d62821c ntdll!LdrpInitializeThread+0x18f	0222ffdc ffffffff
0222fc2c 00000000	0222ffe0 7d4d89c4 kernel32!_except_handler3
0222fc30 7efde000	0222ffe4 7d4dfe28 kernel32!`string'+0x18
0222fc34 00000000	0222ffe8 00000000
...	0222ffec 00000000
0222fc6c 00000070	0222fff0 00000000
0222fc70 ffffffff	0222fff4 00411032 NullThread!ILT+45(?ThreadProcYGKPAXZ)
0222fc74 ffffffff	0222fff8 00000000
0222fc78 7d6281c7 ntdll!LdrpInitializeThread+0xd8	0222fffc 00000000
0222fc7c 7d6280d6 ntdll!LdrpInitializeThread+0x12c	02230000 ????????
0222fc80 00000000	

The second crashed thread has much more symbolic information in it overwriting previous thread startup residue. It is mostly **Exception Handling Residue**<sup>76</sup> because exception handling consumes stack space as explained in the article<sup>77</sup>:

<sup>76</sup> We may add this pattern in the future.

<sup>77</sup> Who Calls the Postmortem Debugger?, Memory Dump Analysis Anthology, Volume 1, page 113

0:003> dds 0236a000 02370000	0236a4ac 00230b98
0236a000 00000000	0236a4b0 0236a590
...	0236a4b4 7d61f5d1 ntdll!RtlFreeHeap+0x20e
0236a060 00000000	0236a4b8 00221378
0236a064 0236a074	0236a4bc 7d61f5ed ntdll!RtlFreeHeap+0x70f
0236a068 00220000	0236a4c0 00000000
0236a06c 7d61f7b4 ntdll!RtlpAllocateFromHeapLookaside+0x13	0236a4c4 7d61f4ab ntdll!RtlFreeHeap
0236a070 00221378	0236a4c8 00000000
0236a074 0236a29c	0236a4cc 00000000
0236a078 7d61f748 ntdll!RtlAllocateHeap+0x1dd	...
0236a07c 7d61f78c ntdll!RtlAllocateHeap+0xee7	0236a538 00000000
0236a080 0236a5f4	0236a53c 0236a678
0236a084 00000000	0236a540 7d61f1f8 ntdll!_except_handler3
...	0236a544 7d624ba8 ntdll!`string'+0x1c
0236a1b4 0236a300	0236a548 ffffffff
0236a1b8 0236a1dc	0236a54c 7d624ba1
0236a1bc 7d624267 ntdll!RtlIsDosDeviceName_Ustr+0x2f	ntdll!RtlpDosPathNameToRelativeNtPathName_Ustr+0x3cb
0236a1c0 0236a21c	0236a550 7d624c43
0236a1c4 7d624274 ntdll!RtlpDosSlashCONDevice	ntdll!RtlpDosPathNameToRelativeNtPathName_U+0x55
0236a1c8 00000001	0236a554 00000001
0236a1cc 0236a317	0236a558 0236a56c
0236a1d0 00000000	...
0236a1d4 0236a324	0236a590 0236a5c0
0236a1d8 0236a290	0236a594 7d620304 ntdll!RtlNtStatusToDosError+0x38
0236a1dc 7d6248af ntdll!RtlGetFullPathName_Ustr+0x80b	0236a598 7d620309 ntdll!RtlNtStatusToDosError+0x3d
0236a1e0 7d6a00e0 ntdll!FastPebLock	0236a59c 7d61c828 ntdll!ZwWaitForSingleObject+0x15
0236a1e4 7d62489d ntdll!RtlGetFullPathName_Ustr+0x15b	0236a5a0 7d4d8c82 kernel32!WaitForSingleObjectEx+0xac
0236a1e8 0236a5f4	0236a5a4 00000124
0236a1ec 00000208	0236a5a8 00000000
...	0236a5ac 7d4d8c7 kernel32!WaitForSingleObjectEx+0xdc
0236a224 00000000	0236a5b0 00000124
0236a228 00000038	0236a5b4 7d61f49c ntdll!RtlGetLastWin32Error
0236a22c 02080038 oleaut32!_PictSaveMetaFile+0x33	0236a5b8 80070000
0236a230 00000000	0236a5bc 00000024
...	...
0236a27c 00000000	0236a5f8 00000000
0236a280 0236a53c	0236a5fc 0236a678
0236a284 7d61f1f8 ntdll!_except_handler3	0236a600 7d4d89c4 kernel32!_except_handler3
0236a288 7d6245f0 ntdll!`string'+0x5c	0236a604 7d4d8cb0 kernel32!`string'+0x68
0236a28c ffffffff	0236a608 ffffffff
0236a290 7d62489d ntdll!RtlGetFullPathName_Ustr+0x15b	0236a60c 7d4d8ca7 kernel32!WaitForSingleObjectEx+0xdc
0236a294 0236a5c8	0236a610 7d4d8bf1 kernel32!WaitForSingleObject+0x12
0236a298 00000008	0236a614 7d61f49c ntdll!RtlGetLastWin32Error
0236a29c 00000000	0236a618 7d61c92d ntdll!NtClose+0x12
0236a2a0 0236a54c	0236a61c 7d4d8e4f kernel32!CloseHandle+0x59
0236a2a4 7d624bcf	0236a620 00000124
ntdll!RtlpDosPathNameToRelativeNtPathName_Ustr+0x3d8	0236a624 0236a688
0236a2a8 7d6a00e0 ntdll!FastPebLock	0236a628 69511753 <Unloaded_faultrep.dll>+0x11753
0236a2ac 7d624ba1	0236a62c 6951175b <Unloaded_faultrep.dll>+0x1175b
ntdll!RtlpDosPathNameToRelativeNtPathName_Ustr+0x3cb	0236a630 0236c6d0
0236a2b0 00000000	...
0236a2b4 0236e6d0	0236a668 00000120
...	0236a66c 00000000
0236a2e0 000a0008	0236a670 0236a630
0236a2e4 7d624be8 ntdll!`string'	0236a674 7d94a2e9 user32!GetSystemMetrics+0x62
0236a2e8 00000000	0236a678 0236f920
0236a2ec 003a0038	0236a67c 69510078 <Unloaded_faultrep.dll>+0x10078
...	0236a680 69503d10 <Unloaded_faultrep.dll>+0x3d10
0236a330 00650070	0236a684 ffffffff
0236a334 0050005c	0236a688 6951175b <Unloaded_faultrep.dll>+0x1175b
0236a338 00480043 advapi32!LsaGetQuotasForAccount+0x25	0236a68c 69506136 <Unloaded_faultrep.dll>+0x6136
0236a33c 00610046	0236a690 0236e6d0
0236a340 006c0075	0236a694 0236c6d0
0236a344 00520074	0236a698 0000009c
0236a348 00700065	0236a69c 0236a6d0
0236a34c 00780045	0236a6a0 00002000
0236a350 00630065	0236a6a4 0236eae4
0236a354 00690050	0236a6a8 695061ff <Unloaded_faultrep.dll>+0x61ff
0236a358 00650070	0236a6ac 00000000
0236a35c 00000000	0236a6b0 00000001
0236a360 00000000	0236a6b4 0236f742
...	0236a6b8 69506210 <Unloaded_faultrep.dll>+0x6210
0236a4a0 0236a4b0	0236a6bc 00000028
0236a4a4 00000001	0236a6c0 0236c76c
0236a4a8 7d61f645 ntdll!RtlpFreeToHeapLookaside+0x22	...

0236e6e0 00500005c	0236e834 00000103
0236e6e4 00480043 advapi32!LsaGetQuotasForAccount+0x25	0236e838 00000000
0236e6e8 00610046	0236e83c 0236f712
...	0236e840 7efaf000
0236e718 002204d8	0236e844 002316f0
0236e71c 0236e890	0236e848 0000009c
0236e720 77b940bb <Unloaded_VERSION.dll>+0x40bb	0236e84c 7efaf000
0236e724 77b91798 <Unloaded_VERSION.dll>+0x1798	0236e850 00000004
0236e728 ffffffff	0236e854 002314b4
0236e72c 77b9178e <Unloaded_VERSION.dll>+0x178e	0236e858 0000ea13
0236e730 69512587 <Unloaded_faultrep.dll>+0x12587	0236e85c 0236e894
0236e734 0236e744	0236e860 00456b0d advapi32!RegQueryValueExW+0x96
0236e738 00220000	0236e864 00000128
0236e73c 7d61f7b4 ntdll!RtlpAllocateFromHeapLookaside+0x13	0236e868 0236e888
0236e740 00221378	0236e86c 0236e8ac
0236e744 0236e96c	0236e870 0236e8c8
0236e748 7d61f748 ntdll!RtlAllocateHeap+0x1dd	0236e874 0236e8a4
0236e74c 7d61f78c ntdll!RtlAllocateHeap+0xee7	0236e878 0236e89c
0236e750 0236eca4	0236e87c 0236e88c
0236e754 00000000	0236e880 7d635dc4 ntdll!iswdigit+0xf
0236e758 0236ec94	0236e884 00000064
0236e75c 7d620309 ntdll!RtlNtStatusToDosError+0x3d	0236e888 00000004
0236e760 0236e7c8	0236e88c 7d624d81 ntdll!RtlpValidateCurrentDirectory+0xf6
0236e764 7d61c9db ntdll!NtQueryValueKey	0236e890 7d635d4e ntdll!RtlIsDosDeviceName_Ustr+0x1c0
0236e768 0236e888	0236e894 00000064
0236e76c 0236e760	0236e898 0236e9d0
0236e770 7d61c9ed ntdll!NtQueryValueKey+0x12	0236e89c 0236e9e7
0236e774 0236f920	0236e8a0 00000000
0236e778 7d61f1f8 ntdll!_except_handler3	0236e8a4 0236e9f4
0236e77c 7d62b310 ntdll!RtlpRunTable+0x490	0236e8a8 0236e960
0236e780 0236e790	0236e8ac 7d6248af ntdll!RtlGetFullPathName_Ustr+0x80b
0236e784 00220000	0236e8b0 7d6a00e0 ntdll!FastPebLock
0236e788 7d61f7b4 ntdll!RtlpAllocateFromHeapLookaside+0x13	0236e8b4 7d62489d ntdll!RtlGetFullPathName_Ustr+0x15b
0236e78c 00221378	0236e8b8 0236eca4
0236e790 0236e9b8	0236e8bc 00000208
0236e794 7d61f748 ntdll!RtlAllocateHeap+0x1dd	0236e8c0 0236ec94
0236e798 7d61f78c ntdll!RtlAllocateHeap+0xee7	0236e8c4 00000000
0236e79c 0236ef18	0236e8c8 00220178
0236e7a0 00000000	0236e8cc 00000004
0236e7a4 00000000	0236e8d0 0236eb3c
0236e7a8 00220000	0236e8d4 0236e8c8
0236e7ac 0236e89c	0236e8d8 7d624d81 ntdll!RtlpValidateCurrentDirectory+0xf6
0236e7b0 00000000	0236e8dc 0236e8f8
0236e7b4 00000128	0236e8e0 7d6246c1 ntdll!RtlIsDosDeviceName_Ustr+0x14
0236e7b8 00000000	0236e8e4 0236ea1c
0236e7bc 0236e8c8	0236e8e8 0236ea33
0236e7c0 0236e7c8	0236e8ec 00000000
0236e7c4 c0000034	0236e8f0 0236ea40
0236e7c8 0236e814	0236e8f4 0236e9ac
0236e7cc 7d61f1f8 ntdll!_except_handler3	0236e8f8 7d6248af ntdll!RtlGetFullPathName_Ustr+0x80b
0236e7d0 7d61f5f0 ntdll!CheckHeapFillPattern+0x64	0236e8f8c 7d6a00e0 ntdll!FastPebLock
0236e7d4 ffffffff	0236e900 7d62489d ntdll!RtlGetFullPathName_Ustr+0x15b
0236e7d8 7d61f5ed ntdll!RtlFreeHeap+0x70f	0236e904 0236ef18
0236e7dc 7d4ded95 kernel32!FindClose+0x9b	0236e908 00000208
0236e7e0 00220000	...
0236e7e4 00000000	0236e934 00000022
0236e7e8 00220000	0236e938 00460044 advapi32!GetPerflibKeyValue+0x19e
0236e7ec 00000000	0236e93c 0236ecd0
0236e7f0 002314b4	0236e940 00000000
0236e7f4 7d61ca1d ntdll!NtQueryInformationProcess+0x12	0236e944 00000044
0236e7f8 7d4da465 kernel32!GetErrorMode+0x18	0236e948 02080044 oleaut32!_PictSaveMetaFile+0x3f
0236e7fc ffffffff	0236e94c 00000000
0236e800 0000000c	0236e950 4336ec0c
0236e804 7d61ca65 ntdll!ZwSetInformationProcess+0x12	...
0236e808 7d4da441 kernel32!SetErrorMode+0x37	0236e9a8 0236ebd0
0236e80c ffffffff	0236e9ac 7d62155b ntdll!RtlAllocateHeap+0x460
0236e810 0000000c	0236e9b0 7d61f78c ntdll!RtlAllocateHeap+0xee7
0236e814 0236e820	0236e9b4 00000000
0236e818 00000004	0236e9b8 000003ee
0236e81c 00000000	0236e9bc 0236ed2c
0236e820 00000005	0236e9c0 7d624bcf
0236e824 0236eae8	ntdll!RtlpDosPathNameToRelativeNtPathName_Ustr+0x3d8
0236e828 7d4e445f kernel32!GetLongPathNameW+0x38f	0236e9c4 7d6a00e0 ntdll!FastPebLock
0236e82c 7d4e4472 kernel32!GetLongPathNameW+0x3a2	0236e9c8 00000ab0
0236e830 00000001	0236e9cc 00000381

0236e9d0 00233950	0236ebc8 00233948
0236e9d4 0236ebfc	0236ebcc 00233950
0236e9d8 7d62155b ntdll!RtlAllocateHeap+0x460	0236ebd0 00220178
0236e9dc 7d61f78c ntdll!RtlAllocateHeap+0xee7	0236ebd4 00220000
0236e9e0 00000003	0236ebd8 00000ab0
0236e9e4 ffffffc	0236ebdc 00220178
0236e9e8 00000aa4	0236ebe0 00000000
0236e9ec 00230ba0	0236ebe4 00233950
0236e9f0 00000004	0236ebe8 7d4dde8 kernel32!`string'+0x50
0236e9f4 003a0043	0236ebec 00000000
0236e9f8 00000000	0236ebf0 00233950
0236e9fc 000a0008	0236ebf4 00220178
0236ea00 7d624be8 ntdll!`string'	0236ebf8 00000aa4
0236ea04 00000000	0236ebfc 00000000
0236ea08 00460044 advapi32!GetPerflibKeyValue+0x19e	0236ec00 0236ec54
0236ea0c 0236ecd0	0236ec04 7d63668a ntdll!RtlCreateProcessParameters+0x375
0236ea10 00233948	0236ec08 7d63668f ntdll!RtlCreateProcessParameters+0x37a
...	0236ec0c 7d6369e9 ntdll!RtlCreateProcessParameters+0x35f
0236ea44 00220640	0236ec10 00000000
0236ea48 7d62273d ntdll!RtlIntegerToUnicode+0x126	...
0236ea4c 0000000c	0236ec4c 0000007f
...	0236ec50 0236ef4c
0236eb4b 0236f79c	0236ec54 7d61f1f8 ntdll!_except_handler3
0236ebab 7d61f1f8 ntdll!_except_handler3	0236ec58 7d61f5f0 ntdll!CheckHeapFillPattern+0x64
0236eabc 7d622758 ntdll!RtlpIntegerWChars+0x54	0236ec5c ffffffff
0236eac0 00220178	0236ec60 7d61f5ed ntdll!RtlFreeHeap+0x70f
0236eac4 0236ed3c	0236ec64 7d6365e2 ntdll!RtlDestroyProcessParameters+0x1b
0236eac8 00000005	0236ec68 00220000
0236eacc 0236ed00	0236ec6c 00000000
0236ead0 7d622660 ntdll!RtlConvertSidToString+0x1cb	0236ec70 00233950
0236ead4 00220178	0236ec74 0236ef5c
0236ead8 0236eaef0	0236ec78 7d4ec4bc kernel32!BasePushProcessParameters+0x806
0236eadc 0236eaec	0236ec7c 00233950
0236eae0 00000001	0236ec80 7d4ec478 kernel32!BasePushProcessParameters+0x7c5
0236eae4 7d61f645 ntdll!RtlpFreeToHeapLookaside+0x22	0236ec84 7efde000
0236eae8 00223620	0236ec88 0236f748
0236eaec 00220178	0236ec8c 00000000
0236eaf0 7d61f5d1 ntdll!RtlFreeHeap+0x20e	0236ec90 0236ed92
0236eaf4 002217f8	0236ec94 00000000
0236eaf8 7d61f5ed ntdll!RtlFreeHeap+0x70f	0236ec98 00000000
0236eafc 00000000	0236ec9c 01060104
0236eb00 00220178	0236eca0 0236f814
...	0236eca4 0020001e
0236eb48 0236eb58	0236eca8 7d535b50 kernel32!`string'
0236eb4c 7d635dc4 ntdll!iswdigit+0xf	0236ecac 00780076
0236eb50 00220178	0236ecb0 002314e0
0236eb54 00000381	0236ecb4 00780076
0236eb58 002343f8	0236ecb8 0236ed2c
0236eb5c 0236eb78	0236ecbc 00020000
0236eb60 7d620deb ntdll!RtlCoalesceFreeBlocks+0x383	0236ecc0 7d4dde4 kernel32!`string'
0236eb64 00000381	0236ecc4 0236efec
0236eb68 002343f8	...
0236eb6c 00220000	0236ed3c 006d0061
0236eb70 00233948	0236ed40 00460020 advapi32!GetPerflibKeyValue+0x17a
0236eb74 00220000	0236ed44 006c0069
0236eb78 00000000	0236ed48 00730065
0236eb7c 00220000	0236ed4c 00280020
0236eb80 0236ec60	0236ed50 00380078
0236eb84 7d620fbe ntdll!RtlFreeHeap+0x6b0	0236ed54 00290036
0236eb88 00220608	0236ed58 0044005c advapi32!CryptDuplicateHash+0x3
0236eb8c 7d61f5ed ntdll!RtlFreeHeap+0x70f	0236ed5c 00620065
0236eb90 000000e8	0236ed60 00670075
0236eb94 7d61cd23 ntdll!ZwWriteVirtualMemory	...
0236eb98 7efde000	0236ee7c 0236ee8c
0236eb9c 000000e8	0236ee80 00000001
0236eba0 00233948	0236ee84 7d61f645 ntdll!RtlpFreeToHeapLookaside+0x22
0236eba4 7efde000	0236ee88 00230dc0
0236eba8 000002e8	0236ee8c 0236ef6c
0236ebac 0000005d	0236ee90 0236eea0
0236ebb0 00220178	0236ee94 00000001
0236ebb4 00000156	0236ee98 7d61f645 ntdll!RtlpFreeToHeapLookaside+0x22
0236ebb8 0236eb94	0236ee9c 00223908
0236ebbc 00233948	0236eea0 0236ef80
0236ebc0 7d61f1f8 ntdll!_except_handler3	0236eea4 7d61f5d1 ntdll!RtlFreeHeap+0x20e
0236ebc4 00000ab0	0236eea8 00221d38

0236eeac	7d61f5ed ntdll!RtlFreeHeap+0x7f	0236f0d8	0236f204
0236eeb0	7d61f4ab ntdll!RtlFreeHeap	0236f0dc	00000020
0236eeb4	7d61c91b ntdll!NtClose	...	
0236eeb8	00000000	0236f190	000002a8
...		0236f194	7d625b62 ntdll!RtlpFindUnicodeStringInSection+0x7b
0236ef08	00000000	0236f198	00000001
0236ef0c	7d621954 ntdll!RtlImageNtHeaderEx+0xee	0236f19c	00000000
0236ef10	7efde000	0236f1a0	0236f1d0
0236ef14	00001000	0236f1a4	7d6257f1
0236ef18	00000000	ntdll!RtlpFindNextActivationContextSection+0x64	
0236ef1c	000000e8	0236f1a8	00181f1c
0236ef20	004000e8 NullThread!_enc\$textbss\$begin <PERF>	...	
(NullThread+0xe8)		0236f1f0	7efaf000
0236ef24	00000000	0236f1f4	7d625ad8
0236ef28	0236ef10	ntdll!RtlFindActivationContextSectionString+0xe1	
0236ef2c	00000000	0236f1f8	0236f214
0236ef30	0236f79c	0236f1fc	0236f24c
0236ef34	7d61f1f8 ntdll!_except_handler3	0236f200	00000000
0236ef38	7d621954 ntdll!RtlImageNtHeaderEx+0xee	0236f204	7d6256e8 ntdll!bsearch+0x42
0236ef3c	00220000	0236f208	00180144
...		...	
0236ef68	0236eeb0	0236f24c	00000200
0236ef6c	7d61f5ed ntdll!RtlFreeHeap+0x7f	0236f250	00000734
0236ef70	0236f79c	0236f254	7d625b62 ntdll!RtlpFindUnicodeStringInSection+0x7b
0236ef74	7d61f1f8 ntdll!_except_handler3	0236f258	0236f384
0236ef78	7d61f5f0 ntdll!CheckHeapFillPattern+0x64	...	
0236ef7c	fffffff	0236f3f0	00000000
0236ef80	7d61f5ed ntdll!RtlFreeHeap+0x70f	0236f3f4	00000000
0236ef84	7d4ea183 kernel32!CreateProcessInternalW+0x21f5	0236f3f8	01034236
0236ef88	00220000	0236f3fc	00000000
0236ef8c	00000000	0236f400	7d4d1510 kernel32!BaseProcessStartThunk
0236ef90	00223910	0236f404	00000018
0236ef94	7d4ebc0b kernel32!CreateProcessInternalW+0x1f26	0236f408	00003000
0236ef98	00000000	...	
0236ef9c	00000096	0236f62c	0236f63c
0236ef9d	0236f814	0236f630	00000001
0236ef9e	00000103	0236f634	7d61f645 ntdll!RtlpFreeToHeapLookaside+0x22
0236ef9f	7efde000	0236f638	00231088
0236efac	00000001	0236f63c	0236f71c
0236efb0	0236effc	...	
0236efb4	00000200	0236f70c	002333b8
0236efb8	00000cb0	0236f710	0236f720
0236efbc	0236f90c	0236f714	00000001
0236efc0	0236efdc	0236f718	7d61f645 ntdll!RtlpFreeToHeapLookaside+0x22
0236efc4	7d6256e8 ntdll!bsearch+0x42	0236f71c	00228fb0
0236efc8	00180144	0236f720	0236f800
0236efcc	0236efe0	0236f724	7d61f5d1 ntdll!RtlFreeHeap+0x20e
0236efd0	7d625992 ntdll!ARRAY_FITS+0x29	0236f728	00221318
0236efd4	00000a8c	0236f72c	7d61f5ed ntdll!RtlFreeHeap+0x70f
0236efd8	00000000	0236f730	00000000
0236efdc	00000000	0236f734	00000096
0236efe0	00080000	0236f738	0236f814
0236efe4	00070000	0236f73c	00220608
0236efe8	00040000	0236f740	7d61f5ed ntdll!RtlFreeHeap+0x70f
0236fec	00000044	0236f744	0236f904
0236eff0	00000000	0236f748	008e0000
0236eff4	7d535b50 kernel32!`string'	0236f74c	002334c2
0236eff8	00000000	...	
0236effc	00000000	0236f784	0236f7bc
...		0236f788	7d63d275 ntdll!_vsnwprintf+0x30
0236f070	00000001	0236f78c	0236f79c
0236f074	7d625ad8	0236f790	0000f949
ntdll!RtlFindActivationContextSectionString+0xe1		0236f794	0236ef98
0236f078	004000e8 NullThread!_enc\$textbss\$begin <PERF>	0236f798	00000095
(NullThread+0xe8)		0236f79c	0236fb7c
0236f07c	0236f0cc	0236f7a0	7d4d89c4 kernel32!_except_handler3
0236f080	00000000	0236f7a4	7d4ed1d0 kernel32!`string'+0xc
0236f084	7d6256e8 ntdll!bsearch+0x42	0236f7a8	ffffffffff
0236f088	00180144	0236f7ac	7d4ebc0b kernel32!CreateProcessInternalW+0x1f26
0236f08c	0236f9a0	0236f7b0	7d4d14a2 kernel32!CreateProcessW+0x2c
0236f090	7d625992 ntdll!ARRAY_FITS+0x29	0236f7b4	00000000
0236f094	00000a8c	...	
...		0236f7f0	0236fb7c
0236f0d0	0236f120	0236f7f4	7d61f1f8 ntdll!_except_handler3
0236f0d4	7d625b62 ntdll!RtlpFindUnicodeStringInSection+0x7b	0236f7f8	7d61d051 ntdll!NtWaitForMultipleObjects+0x15

0236f7fc	7d61c92d ntdll!NtClose+0x12	0236f9ec	00000000
0236f800	7d4d8e4f kernel32!CloseHandle+0x59	0236f9f0	0000000c
0236f804	00000108	0236f9f4	00000000
0236f808	0236fb8c	0236f9f8	00000001
0236f80c	7d535b07 kernel32!UnhandledExceptionFilter+0x815	0236f9fc	00000118
0236f810	00000108	0236fa00	000000e8
0236f814	00430022 advapi32!_imp__OutputDebugStringW <PERF>	0236fa04	c0000005
(advapi32+0x22)		0236fa08	00000000
0236f818	005c003a	0236fa0c	00000008
0236f81c	00720050	0236fa10	00000000
...		0236fa14	00000110
0236f8ec	00550005c	0236fa18	0236f814
0236f8f0	00650073	0236fa1c	6950878a <Unloaded_faultrep.dll>+0x878a
0236f8f4	00440072 advapi32!CryptDuplicateHash+0x19	0236fa20	00120010
0236f8f8	006d0075	0236fa24	7d51c5e4 kernel32!`string'
0236f8fc	00730070	0236fa28	00000003
0236f900	006e005c	0236fa2c	05bc0047
0236f904	00770065	...	
0236f908	0064002e	0236fa74	00570005c
0236f90c	0070006d	0236fa78	004b0032 advapi32!szPerflibSectionName <PERF>
0236f910	0020003b	(advapi32+0x80032)	
0236f914	00220071	0236fa7c	005c0033
0236f918	00000000	0236fa80	00790073
0236f91c	00000096	...	
0236f920	7d4ddaa7 kernel32!DuplicateHandle+0xd0	0236fac8	0000002b
0236f924	7d4ddaa7 kernel32!DuplicateHandle+0xd0	0236facc	00000000
0236f928	0236fb8c	0236fad0	7d61e3e6 ntdll!ZwWow64CsrNewThread+0x12
0236f92c	7d5358cb kernel32!UnhandledExceptionFilter+0x5f1	0236fad4	00000000
0236f930	0236f9f0	...	
0236f934	00000001	0236fb44	00000000
0236f938	00000000	0236fb48	00000000
0236f93c	7d535b43 kernel32!UnhandledExceptionFilter+0x851	0236fb4c	7d61cb0d ntdll!ZwQueryVirtualMemory+0x12
0236f940	00000000	0236fb50	7d54eeb8 kernel32!_ValidateEH3RN+0xb6
0236f944	00000000	0236fb54	ffffffffff
0236f948	00000000	0236fb58	7d4df28 kernel32!`string'+0x18
0236f94c	0236f95c	0236fb5c	00000000
0236f950	00000098	0236fb60	0236fb78
0236f954	000001a2	0236fb64	0000001c
0236f958	01c423b0	0236fb68	0000000f
0236f95c	0236fb84	0236fb6c	7d4df28 kernel32!`string'+0x18
0236f960	7d62155b ntdll!RtlAllocateHeap+0x460	0236fb70	0000f949
0236f964	7d61f78c ntdll!RtlAllocateHeap+0xee7	0236fb74	0236f814
0236f968	00000000	0236fb78	7d4df000 kernel32!CheckForSameCurdir+0x39
0236f96c	0000008c	0236fb7c	0236fb4
0236f970	00000000	0236fb80	7d4d89c4 kernel32!_except_handler3
0236f974	7d4d8472 kernel32!\$VProc_ImageExportDirectory+0x6d4e	0236fb84	7d535be0 kernel32!`string'+0xc
0236f978	0236fa1c	0236fb88	ffffffffff
0236f97c	00000044	0236fb8c	7d535b43 kernel32!UnhandledExceptionFilter+0x851
0236f980	00000000	0236fb90	7d508f4e kernel32!BaseThreadStart+0x4a
0236f984	7d535b50 kernel32!`string'	0236fb94	0236fb4
0236f988	00000000	0236fb98	7d4d8a25 kernel32!_except_handler3+0x61
0236f98c	00000000	0236fba0	0236fbcc
0236f990	00000000	0236fba4	0236fbcc
0236f994	00000000	0236fba8	00000000
0236f998	00000000	0236fbac	00000000
0236f99c	00000000	0236fb0	00000000
0236f9a0	00000000	0236fb4	0236fc0
0236f9a4	00000000	0236fb8	0236fc0
0236f9a8	00000000	0236fb0	0236fb0
0236f9ac	00000000	0236fb4	0236fc0
0236f9b0	00000000	0236fb8	7d61ec2a ntdll!ExecuteHandler2+0x26
0236f9b4	00000000	0236fbc4	0236fc0
0236f9b8	00000000	0236fb8	0236fdc
0236f9bc	00000000	0236fbcc	0236fcf0
0236f9c0	0010000e	0236fb0	0236fc7c
0236f9c4	7ffe0030 SharedUserData+0x30	0236fb4	0236fdc
0236f9c8	000000e8	0236fb8	7d61ec3e ntdll!ExecuteHandler2+0x3a
0236f9cc	00000108	0236fbdc	0236fdc
0236f9d0	00000200	0236fbe0	0236fc88
0236f9d4	00000734	0236fbef	7d61ebfb ntdll!ExecuteHandler+0x24
0236f9d8	00000018	0236fbe8	0236fc0
0236f9dc	00000000	0236fbec	0236fdc
0236f9e0	7d5621d0 kernel32!ProgramFilesEnvironment+0x74	0236fb0	00000000
0236f9e4	00000040	0236fb4	0236fc7c
0236f9e8	00000000	0236fb8	7d4d89c4 kernel32!_except_handler3

0236fbfc 00000000	0236fd44 00000000
0236fc00 0036fc00	0236fd48 00000000
0236fc04 0236fc18	0236fd4c 00000000
0236fc08 7d640ca6 ntdll!RtlCallVectoredContinueHandlers+0x15	0236fd50 00000000
0236fc0c 0236fc00	0236fd54 00000000
0236fc10 0236fc00	0236fd58 00000000
0236fc14 7d6a0608 ntdll!RtlpCallbackEntryList	0236fd5c 00000000
0236fc18 0236fc88	0236fd60 00000000
0236fc1c 7d6354c9 ntdll!RtlDispatchException+0x11f	0236fd64 00000000
0236fc20 0236fc00	0236fd68 00000000
0236fc24 0236fc00	0236fd6c 00000000
0236fc28 00000000	0236fd70 00000000
0236fc2c 00000000	0236fd74 00000000
...	0236fd78 00000000
0236fc88 0236ffec	0236fd7c 0000002b
0236fc8c 7d61dd26 ntdll!NtRaiseException+0x12	0236fd80 00000053
0236fc90 7d61ea51 ntdll!KiUserExceptionDispatcher+0x29	0236fd84 0000002b
0236fc94 0236fc00	0236fd88 0000002b
0236fc98 0236fc00	0236fd8c 00000000
0236fc9c 00000000	0236fd90 00000000
0236fc00 c0000005	0236fd94 00000000
0236fc04 00000000	0236fd98 00000000
0236fc08 00000000	0236fd9c 47f30000
0236fcac 00000000	0236fda0 00000000
0236fcb0 00000002	0236fda4 0236ffec
0236fcb4 00000008	0236fda8 00000000
0236fcb8 00000000	0236fdac 00000023
0236fcbc 00000000	0236fdb0 0010246
0236fcc0 00000000	0236fdb4 0236ffbc
0236fcc4 6b021fa0	0236fdb8 0000002b
0236fcc8 78b83980	0236fdbc 0000027f
0236fccc 00000000	0236fdc0 00000000
0236fc00 00000000	0236fdc4 00000000
0236fc04 00000000	0236fdc8 00000000
0236fc08 7efad000	0236fdcc 00000000
0236fc0c 023af000	0236fdde 00000000
0236fc0e 023af110	0236fdd4 00001f80
0236fc04 78b83980	0236fdd8 00000000
0236fc08 010402e1	0236fddc 00000000
0236fc0c 00000000	...
0236fc0f 0001003f	0236ffb4 00000000
0236fc04 00000000	0236ffb8 00000000
0236fc08 00000000	0236ffb0 7d4dfe21 kernel32!BaseThreadStart+0x34
0236fc0c 00000000	0236ffc0 00000000
0236fd00 00000000	0236ffc4 00000000
0236fd04 00000000	0236ffc8 00000000
0236fd08 00000000	0236ffcc 00000000
0236fd0c 0000027f	0236ffd0 c0000005
0236fd10 00000000	0236ffd4 0236fc4
0236fd14 0000ffff	0236ffd8 0236fb04
0236fd18 00000000	0236ffdc ffffffff
0236fd1c 00000000	0236ffe0 7d4d89c4 kernel32!_except_handler3
0236fd20 00000000	0236ffe4 7d4dfe28 kernel32!`string'+0x18
0236fd24 00000000	0236ffe8 00000000
0236fd28 00000000	0236ffec 00000000
0236fd2c 00000000	0236fff0 00000000
0236fd30 00000000	0236fff4 00000000
0236fd34 00000000	0236fff8 00000000
0236fd38 00000000	0236fffc 00000000
0236fd3c 00000000	02370000 ???????
0236fd40 00000000	

## Comments

**!DumpStack** and **!EEStack** SOS commands provide a summary of “call type” execution residue from the raw stack.

**System Objects** can shed extra light on past software behavior (page 971).

We can also use **dpS** WinDbg command on a range to get all symbolic references only.

The undocumented **!ddstack** WinDbg extension (ext) command may save some time for listing **Execution Residue** symbols. It is equivalent to **!teb**, then **dpS**, but also gives stack addresses for mapped symbols like **dps** command but with less output (the command may give incorrect results for WOW64 process memory dumps saved as x64 memory dumps):

```
0:001> !ddstack
Range: 0000000002ecf000->0000000002ee0000
0x00000000`02edf4b0 0x00000000`775d0000 ntdll!RtlDeactivateActivationContext <PERF>
(ntdll+0x0)+0000000000000000
0x00000000`02edf4c8 0x00000000`775fc454 ntdll!LdrpInitialize+000000000000000a4
0x00000000`02edf538 0x00000000`775fc358 ntdll!LdrInitializeThunk+0000000000000018
0x00000000`02edf5e0 0x00000000`776d4578 ntdll!`string'+0000000000000000
0x00000000`02edf5e8 0x00000000`776d44e0 ntdll!`string'+0000000000000000
0x00000000`02edf5f8 0x00000000`775f0f3b ntdll!TpPostTask+0000000000000019b
0x00000000`02edf658 0x00000000`775e6199 ntdll!TppWorkPost+0000000000000089
0x00000000`02edf688 0x00000000`775fc520 ntdll!RtlUserThreadStart+0000000000000000
0x00000000`02edf698 0x00000000`775e6b4d ntdll!TppWaitComplete+000000000000003d
0x00000000`02edf6a8 0x00000000`775e6b4d ntdll!TppWaitComplete+000000000000003d
0x00000000`02edf6c8 0x00000000`775eb828 ntdll!TppWaiterpDoTransitions+0000000000000154
0x00000000`02edf6e8 0x00000000`775e6abe ntdll!TppWaiterpCompleteWait+000000000000004e
0x00000000`02edf6f8 0x00000000`775d7858 ntdll!TppWaiterpWaitTimerExpired+0000000000000038
0x00000000`02edf720 0x00000000`776d4578 ntdll!`string'+0000000000000000
0x00000000`02edf728 0x00000000`776d44e0 ntdll!`string'+0000000000000000
0x00000000`02edf738 0x00000000`776d4500 ntdll!`string'+0000000000000000
0x00000000`02edf748 0x00000000`775eb037 ntdll!TppWaiterpThread+000000000000014d
0x00000000`02edf768 0x00000000`776d4550 ntdll!`string'+0000000000000000
0x00000000`02edf9e8 0x00000000`773c59ed kernel32!BaseThreadInitThunk+000000000000000d
0x00000000`02edfa18 0x00000000`775fc541 ntdll!RtlUserThreadStart+000000000000001d

0:001> !teb
TEB at 000007fffffdb000
ExceptionList: 0000000000000000
StackBase: 0000000002ee0000
StackLimit: 0000000002ecf000
SubSystemTib: 0000000000000000
FiberData: 0000000000001e00
ArbitraryUserPointer: 0000000000000000
Self: 000007fffffdb000
EnvironmentPointer: 0000000000000000
ClientId: 000000000001344 . 0000000000001ab0
RpcHandle: 0000000000000000
Tls Storage: 0000000000000000
PEB Address: 000007fffffdf000
LastErrorValue: 0
LastStatusValue: 0
Count Owned Locks: 0
HardErrorMode: 0

0:001> dpS 0000000002ecf000 0000000002ee0000
00000000`775d0000 ntdll!RtlDeactivateActivationContext <PERF> (ntdll+0x0)
00000000`775fc454 ntdll!LdrpInitialize+0xa4
00000000`775fc358 ntdll!LdrInitializeThunk+0x18
00000000`776d4578 ntdll!`string'
00000000`776d44e0 ntdll!`string'
00000000`775f0f3b ntdll!TpPostTask+0x19b
```

```
00000000`775e6199 ntdll!TppWorkPost+0x89
00000000`775fc520 ntdll!RtlUserThreadStart
00000000`775e6b4d ntdll!TppWaitComplete+0x3d
00000000`775e6b4d ntdll!TppWaitComplete+0x3d
00000000`775eb828 ntdll!TppWaiterpDoTransitions+0x154
00000000`775e6abe ntdll!TppWaiterpCompleteWait+0x4e
00000000`775d7858 ntdll!TppWaiterpWaitTimerExpired+0x38
00000000`776d4578 ntdll!`string'
00000000`776d44e0 ntdll!`string'
00000000`776d4500 ntdll!`string'
00000000`775eb037 ntdll!TppWaiterpThread+0x14d
00000000`776d4550 ntdll!`string'
00000000`773c59ed kernel32!BaseThreadInitThunk+0xd
00000000`775fc541 ntdll!RtlUserThreadStart+0x1d
```

**F****Fake Module**

In **Fake Module** pattern, one of the loaded modules masquerades as a legitimate system DLL or a widely known value adding DLL from some popular 3<sup>rd</sup>-party product. To illustrate this pattern we modeled it as Victimware<sup>78</sup>: a process crashed after loading a malware module:

```
0:000> k
*** Stack trace for last set context - .thread/.cxr resets it
Child-SP      RetAddr          Call Site
00000000`0026f978 00000001`3f89103a 0x0
00000000`0026f980 00000001`3f8911c4 FakeModule!wmain+0x3a
00000000`0026f9c0 00000000`76e3652d FakeModule!__tmainCRTStartup+0x144
00000000`0026fa00 00000000`7752c521 kernel32!BaseThreadInitThunk+0xd
00000000`0026fa30 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

When we inspected loaded modules we didn't find anything suspicious:

```
0:000> lmp
start          end            module name
00000000`76e20000 00000000`76f3f000  kernel32 <none>
00000000`77500000 00000000`776a9000  ntdll   <none>
00000001`3f890000 00000001`3f8a6000  FakeModule <none>
000007fe`f8cb0000 000007fe`f8cc7000  winspool <none>
000007fe`fdb30000 000007fe`fdb9c000  KERNELBASE <none>
```

However, when checking module images for any modifications we find that *winspool* module was not compared with the corresponding existing binary from Microsoft symbol server:

```
0:000> !for_each_module "!chkimg -v -d #@ModuleName"
Searching for module with expression: kernel32
Will apply relocation fixups to file used for comparison
Will ignore NOP/LOCK errors
Will ignore patched instructions
Image specific ignores will be applied
Comparison image path: C:\WSDK8\Debuggers\x64\sym\kernel32.dll\503285C111f000\kernel32.dll
No range specified

Scanning section:    .text
Size: 633485
Range to scan: 76e21000-76ebba8d
Total bytes compared: 633485(100%)
Number of errors: 0
0 errors : kernel32
```

---

<sup>78</sup> <http://www.dumpanalysis.org/victimware-book>

```

Searching for module with expression: ntdll
Will apply relocation fixups to file used for comparison
Will ignore NOP/LOCK errors
Will ignore patched instructions
Image specific ignores will be applied
Comparison image path: C:\WSDK8\Debuggers\x64\sym\ntdll.dll\4EC4AA8E1a9000\ntdll.dll
No range specified

Scanning section: .text
Size: 1049210
Range to scan: 77501000-7760127a
Total bytes compared: 1049210(100%)
Number of errors: 0

Scanning section: RT
Size: 474
Range to scan: 77602000-776021da
Total bytes compared: 474(100%)
Number of errors: 0
0 errors : ntdll
Searching for module with expression: FakeModule
Error for FakeModule: Could not find image file for the module. Make sure binaries are included in the symbol path.
Searching for module with expression: winspool
Error for winspool: Could not find image file for the module. Make sure binaries are included in the symbol path.
Searching for module with expression: KERNELBASE
Will apply relocation fixups to file used for comparison
Will ignore NOP/LOCK errors
Will ignore patched instructions
Image specific ignores will be applied
Comparison image path: C:\WSDK8\Debuggers\x64\sym\KERNELBASE.dll\503285C26c000\KERNELBASE.dll
No range specified

Scanning section: .text
Size: 302047
Range to scan: 7fefdb31000-7fefdb7abdf
Total bytes compared: 302047(100%)
Number of errors: 0
0 errors : KERNELBASE

```

Checking module data reveals that it was loaded not from *System32* folder and also doesn't have any version information:

```

0:000> lmv m winspool
start           end             module name
000007fe`f8cb0000 000007fe`f8cc7000  winspool  (deferred)
Image path: C:\Work\AWMA\FakeModule\x64\Release\winspool.drv
Image name: winspool.drv
Timestamp:      Fri Dec 28 22:22:42 2012 (50DE1BB2)
CheckSum:        00000000
ImageSize:       00017000
File version:   0.0.0.0
Product version: 0.0.0.0

```

```
File flags:      0 (Mask 0)
File OS:        0 Unknown Base
File type:      0.0 Unknown
File date:      00000000.00000000
Translations:   0000.04b0 0000.04e4 0409.04b0 0409.04e4
```

We could see that path by running the following command as well:

```
0:000> !for_each_module
00: 000000076e20000 000000076f3f000      kernel32
C:\Windows\System32\kernel32.dll           kernel32.dll
01: 000000077500000 00000000776a9000      ntdll
C:\Windows\System32\ntdll.dll             ntdll.dll
02: 000000013f890000 000000013f8a6000      FakeModule
C:\Work\AWMA\FakeModule\x64\Release\FakeModule.exe  FakeModule.exe
03: 000007fef8cb0000 000007fef8cc7000      winspool C:\Work\AWMA\FakeModule\x64\Release\winspool.drv
04: 000007feffdb30000 000007fefdb9c000      KERNELBASE
C:\Windows\System32\KERNELBASE.dll        KERNELBASE.dll
```

Or from PEB:

```
0:000> !peb
PEB at 000007fffffdf000
[...]
7fef8cb0000 50de1bb2 Dec 28 22:22:42 2012 C:\Work\AWMA\FakeModule\x64\Release\winspool.drv
[...]
```

Another sign is the module size in memory which is much smaller than the real *winspool.drv*:

```
0:000> ? 000007fe`f8cc7000 - 000007fe`f8cb0000
Evaluate expression: 94208 = 0000000`0001700
```

Module size could help if legitimate module from the well-known folder was replaced. Module debug directory, and the size of export and import directories are also different with the original one revealing the development folder:

```
0:000> !dh 000007fe`f8cb0000
[...]
0 [     0] address [size] of Export Directory
[...]
9000 [    208] address [size] of Import Address Table Directory
[...]
Debug Directories(2)
Type      Size      Address  Pointer
cv         49       e2c0      cac0 Format: RSFS, guid, 1,
C:\Work\AWMA\FakeModule\x64\Release\winspool.pdb
```

This can also be seen from the output of **!lmi** command:

```
0:000> !lmi 7fef8cb0000
Loaded Module Info: [7fef8cb0000]
Module: winspool
Base Address: 000007fef8cb0000
Image Name: winspool.drv
Machine Type: 34404 (X64)
Time Stamp: 50de1bb2 Fri Dec 28 22:22:42 2012
Size: 17000
CheckSum: 0
Characteristics: 2022
Debug Data Dirs: Type Size VA Pointer
CODEVIEW 49, e2c0, cac0 RSDS - GUID: {29D85193-1C9D-4997-95BA-DD190FA3C1BF}
Age: 1, Pdb: C:\Work\AWMA\FakeModule\x64\Release\winspool.pdb
?: 10, e30c, cb0c [Data not mapped]
Symbol Type: DEFERRED - No error - symbol load deferred
Load Report: no symbols loaded
```

## False Effective Address

When calculating effective addresses such as [r10+10h] or [rax+rcx\*12h+40h] to show their value in the output of some commands such as **.trap** or **.cxr** a debugger uses CPU register values from a saved trap frame or context structure. If such information is invalid, the reported effective address doesn't correspond to the real one during code execution. This analysis pattern is similar to **False Function Parameters** (page 390). Therefore, if a fault address is saved during bugcheck or exception processing, it may not correspond to the output of some commands where such calculation is necessary. For example, in a bugcheck parameter we have this referenced memory address:

```
Arg1: ffffffadda17d001d, memory referenced
```

But the output of **.trap** command shows **NULL Pointer** (page 752) address:

```
NOTE: The trap frame does not contain all registers.  
Some register values may be zeroed or incorrect.  
rax=0000000000000000 rbx=0000000000000000 rcx=0000000000000000  
[...]  
movzx eax,word ptr [rax+10h] 0010=????
```

Usually, we are lucky, and an effective address is correct despite a warning such as in a pattern example on page 169 and a pattern interaction case study<sup>79</sup>.

---

<sup>79</sup> “NULL Data Pointer, Stack Trace, Inline Function Optimization and Platformorphic Fault”, Memory Dump Analysis Anthology, Volume 4, page 201

## False Function Parameters

Beginner users of WinDbg sometimes confuse the first 3 parameters (or 4 for x64) displayed by **kb** or **kv** commands with real function parameters:

```
0:000> kbnL
# ChildEBP RetAddr  Args to Child
00 002df5f4 0041167b 002df97c 00000000 7efdf000 ntdll!DbgBreakPoint
01 002df6d4 004115c9 00000000 40000000 00000001 CallingConventions!A::thiscallFunction+0x2b
02 002df97c 004114f9 00000001 40001000 00000002 CallingConventions!fastcallFunction+0x69
03 002dfbf8 0041142b 00000000 40000000 00000001 CallingConventions!cdeclFunction+0x59
04 002dfe7c 004116e8 00000000 40000000 00000001 CallingConventions!stdcallFunction+0x5b
05 002dff68 00411c76 00000001 005a2820 005a28c8 CallingConventions!wmain+0x38
06 002dffb8 00411abd 002dff0 7d4e7d2a 00000000 CallingConventions!__tmainCRTStartup+0x1a6
07 002dffc0 7d4e7d2a 00000000 00000000 7efdf000 CallingConventions!wmainCRTStartup+0xd
08 002dff0 00000000 00411082 00000000 000000c8 kernel32!BaseProcessStart+0x28
```

The calling sequence for it is:

```
stdcallFunction(0, 0x40000000, 1, 0x40001000, 2, 0x40002000) ->
cdeclFunction(0, 0x40000000, 1, 0x40001000, 2, 0x40002000) ->
fastcallFunction(0, 0x40000000, 1, 0x40001000, 2, 0x40002000) ->
A::thiscallFunction(0, 0x40000000, 1, 0x40001000, 2, 0x40002000)
```

and we see that only in the case of **fastcall** calling convention we have a discrepancy due to the fact that the first two parameters are passed not via the stack but through ECX and EDX:

```
0:000> ub 004114f9
CallingConventions!cdeclFunction+0x45
004114e5 push    ecx
004114e6 mov     edx,dword ptr [ebp+14h]
004114e9 push    edx
004114ea mov     eax,dword ptr [ebp+10h]
004114ed push    eax
004114ee mov     edx,dword ptr [ebp+0Ch]
004114f1 mov     ecx,dword ptr [ebp+8]
004114f4 call    CallingConventions!ILT+475(?fastcallFunctionYIXHHHHHZ) (004111e0)
```

However, if we have full symbols we can see all parameters:

```
0:000> .frame 2
02 002df97c 004114f9 CallingConventions!fastcallFunction+0x69

0:000> dv /i /V
prv param 002df974 @ebp-0x08      a = 0
prv param 002df968 @ebp-0x14      b = 1073741824
prv param 002df984 @ebp+0x08      c = 1
prv param 002df988 @ebp+0x0c      d = 1073745920
prv param 002df98c @ebp+0x10      e = 2
prv param 002df990 @ebp+0x14      f = 1073750016
prv local 002df7c7 @ebp-0x1b5    obj = class A
prv local 002df7d0 @ebp-0x1ac    dummy = int [100]
```

How does **dv** command know about values in ECX and EDX which were definitely overwritten by later code? This is because the called function prolog saved them as local variables which you can notice as negative offsets for EBP register in **dv** output above:

```
0:000> uf CallingConventions!fastcallFunction
CallingConventions!fastcallFunction
 32 00411560 push    ebp
 32 00411561 mov     ebp,esp
 32 00411563 sub    esp,27Ch
 32 00411569 push    ebx
 32 0041156a push    esi
 32 0041156b push    edi
 32 0041156c push    ecx
 32 0041156d lea     edi,[ebp-27Ch]
 32 00411573 mov     ecx,9Fh
 32 00411578 mov     eax,0CCCCCCCCh
 32 0041157d rep stos dword ptr es:[edi]
 32 0041157f pop     ecx
 32 00411580 mov     dword ptr [ebp-14h],edx
 32 00411583 mov     dword ptr [ebp-8],ecx
...
...
...
```

In order to spot the occurrences of this pattern, double checks and knowledge of calling conventions are required. Sometimes this pattern is a consequence of **Optimized Code** pattern (page 767).

x64 stack traces don't show any discrepancies except the fact that **thiscall** function parameters are shifted to the right:

```
0:000> kbl
RetAddr      : Args to Child                      : Call Site
00000001`40001397 : cccccccc`ccccccccc`ccccccccc`ccccccccc`ccccccccc`ccccccccc`ccccccccc`ccccccccc : ntdll!DbgBreakPoint
00000001`40001233 : 00000000`0012fa94 cccccccc`00000000 cccccccc`40000000 cccccccc`00000001 : CallingConventions!A::thiscallFunction+0x37
00000001`40001177 : cccccccc`00000000 cccccccc`40000000 cccccccc`00000001 cccccccc`40001000 : CallingConventions!fastcallFunction+0x93
00000001`400010c7 : cccccccc`00000000 cccccccc`40000000 cccccccc`00000001 cccccccc`40001000 : CallingConventions!cdeclFunction+0x87
00000001`400012ae : cccccccc`00000000 cccccccc`40000000 cccccccc`00000001 cccccccc`40001000 : CallingConventions!stdcallFunction+0x87
00000001`400018ec : 00000001`00000000 00000000`00481a80 00000000`00000000 00000001`400026ee : CallingConventions!wmain+0x4e
00000001`4000173c : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : CallingConventions!_tmainCRTStartup+0x19c
00000000`77d5964c : 00000000`77d59620 00000000`00000000 00000000`00000000 00000000`0012ffa8 : CallingConventions!wmainCRTStartup+0xe
00000000`00000000 : 00000001`40001730 00000000`00000000 00000000`00000000 00000000`00000000 : kernel32!BaseProcessStart+0x29
```

How can this happen if the standard x64 calling convention passes the first 4 parameters via ECX, EDX, R8, and R9? This is because the called function prolog saved them on the stack (this might not be true in the case of optimized code):

```
0:000> uf CallingConventions!fastcallFunction
CallingConventions!fastcallFunction
 32 00000001`400011a0 44894c2420      mov     dword ptr [rsp+20h],r9d
 32 00000001`400011a5 4489442418      mov     dword ptr [rsp+18h],r8d
 32 00000001`400011aa 89542410      mov     dword ptr [rsp+10h],edx
 32 00000001`400011ae 894c2408      mov     dword ptr [rsp+8],ecx
...
...
...
```

*A::thiscallFunction* function passes **this** pointer via ECX too, and this explains the right shift of parameters.

Here is the C++ code we used for experimentation:

```
#include "stdafx.h"
#include <windows.h>

void __stdcall stdcallFunction (int, int, int, int, int, int);
void __cdecl cdeclFunction (int, int, int, int, int, int);
void __fastcall fastcallFunction (int, int, int, int, int, int);

class A
{
public:
    void thiscallFunction (int, int, int, int, int, int) { DebugBreak(); };

void __stdcall stdcallFunction (int a, int b, int c, int d, int e, int f)
{
    int dummy[100] = {0};

    cdeclFunction (a, b, c, d, e, f);
}

void __cdecl cdeclFunction (int a, int b, int c, int d, int e, int f)
{
    int dummy[100] = {0};

    fastcallFunction (a, b, c, d, e, f);
}

void __fastcall fastcallFunction (int a, int b, int c, int d, int e, int f)
{
    int dummy[100] = {0};

    A obj;

    obj.thiscallFunction (a, b, c, d, e, f);
}

int _tmain(int argc, _TCHAR* argv[])
{
    stdcallFunction (0, 0x40000000, 1, 0x40001000, 2, 0x40002000);

    return 0;
}
```

## False Positive Dump

Here we get crash dump files pointing to a wrong direction or not useful for analysis. This usually happens when a wrong tool was selected, or the right one was not properly configured for capturing crash dumps. Here is one example investigated in detail.

The customer experienced frequent spooler crashes on Windows Server 2003. The dump was sent for investigation to find an offending component. Usually, it is a printer driver. WinDbg revealed the following exception thread stack:

```
KERNEL32!RaiseException+0x56
KERNEL32!OutputDebugStringA+0x55
KERNEL32!OutputDebugStringW+0x39
PRINTER!ConvertTicket+0x3c90
PRINTER!D11GetClassObject+0x5d9b
PRINTER!D11GetClassObject+0x11bb
```

The immediate response is to point to *PRINTER.DLL*, but if we look at parameters to *OutputDebugStringA* we see that the string passed to it is a valid NULL-terminated string:

```
0:010> da 000d0040
000d0040  ".Lower DWORD of elapsed time = 3"
000d0060  "750000."
```

If we disassemble *OutputDebugStringA* up to *RaiseException* call we see:

```
0:010> u KERNEL32!OutputDebugStringA
KERNEL32!OutputDebugStringA+0x55
KERNEL32!OutputDebugStringA:
push    ebp
mov     ebp,esp
push    0FFFFFFFh
push    offset KERNEL32!'string'+0x10
push    offset KERNEL32!_except_handler3
mov     eax,dword ptr fs:[00000000h]
push    eax
mov     dword ptr fs:[0],esp
push    ecx
push    ecx
sub    esp,228h
push    ebx
push    esi
push    edi
mov     dword ptr [ebp-18h],esp
and    dword ptr [ebp-4],0
mov     edx,dword ptr [ebp+8]
mov     edi,edx
or     ecx,0FFFFFFFh
xor    eax,eax
repne scas byte ptr es:[edi]
not    ecx
```

```
mov    dword ptr [ebp-20h],ecx
mov    dword ptr [ebp-1Ch],edx
lea    eax,[ebp-20h]
push   eax
push   2
push   0
push   40010006h
call   KERNEL32!RaiseException
```

There are no jumps in the code prior to *RaiseException* call, and this means that raising an exception is expected. Also, MSDN documentation says:

*"If the application has no debugger, the system debugger displays the string. If the application has no debugger and the system debugger is not active, OutputDebugString does nothing."*

So *spoolsv.exe* might have been monitored by a debugger which caught that exception and instead of dismissing it dumped the spooler process.

If we look at **!analyze -v** output we see the following:

```
Comment: 'Userdump generated complete user-mode minidump
with Exception Monitor function on WS002E00-01-MFP'ERROR_CODE: (NTSTATUS) 0x40010006 -
Debugger printed exception on control C.
```

## Fat Process Dump

During repeated execution either on one computer or in parallel on many computers with a uniform software and hardware the given process VM size tends to cluster around some value range, for example, 40 - 60 MB. If we get a collection of user process memory dumps taken from several production servers, say 20 files, we can either employ scripts to process all of them<sup>80</sup> or compare their file size and look for bigger ones for a starter, for example, 85 or 110 Mb. For certain processes, for example, a print spooler, after a software problem the process size tends to increase compared to normal execution. For other processes, certain error processing modules might be loaded increasing VM size, or in the case of incoming requests for a hung process, certain memory regions like heap could increase as well contributing to a dump file size increase. If we have fat and thin clients, we should also have thin and fat process dumps as well.

---

<sup>80</sup> Hundreds of Crash Dumps, Memory Dump Analysis Anthology, Volume 1, page 227

## Fault Context

In the case of multiple different faults like bugchecks and different crash points, stack traces, and modules we can look at what is common among them. It could be their process context, which can easily be seen from the default analysis command:

```
1: kd> !analyze -v  
[...]  
PROCESS_NAME: Application.exe
```

Then we can check whether an application is resource consumption intensive (could implicate hardware faults) like games and simulators or uses its own drivers (implicates latent corruption). In a production environment, it can also be removed if it is functionally non-critical and can be avoided or replaced.

## First Fault Stack Trace

The case of **Error Reporting Fault** chain (page 325) led us to this pattern that corresponds to First Fault software diagnostics pattern proper<sup>81</sup>. Here the term *first fault* is used for an exception that was either ignored by surrounding code or led to other exceptions or error message boxes with stack traces that masked the first one. Typical examples where it is sometimes possible to get a first exception stack trace include but not limited to:

- Fault in error reporting (page 325) started as a fault in some other process
- **Hidden Exception** in user (page 457) or kernel (page 455) space
- Double fault (**Stack Overflow**, page 900)
- **Nested Exception** in unmanaged (page 726) and managed (page 723) code
- **Nested Offender** (page 730)

It is also sometimes possible unless a stack region was **Paged Out** (page 778) to get partial stack traces from **Execution Residue** (page 371) when the sequence of return addresses was partially overwritten by subsequently executed code.

---

<sup>81</sup> Patterns of Software Diagnostics, First Fault, Memory Dump Analysis Anthology, Volume 7, page 406

## Foreign Module Frame

Visio was freezing after saving a diagram as a picture after we tried to close it. It eventually crashed with WER saving a crash dump file in *LocalDumps* folder<sup>82</sup>. After a few such incidents, Visio suggested to disable a 3rd-party plugin. We did that, and double checked in Options \ Add-Ins dialog. Unfortunately, the same abnormal behavior continued. When we looked at the crash dump stack trace we noticed **Foreign Module Frame**:

```
0:000> k
# ChildEBP RetAddr
00 0019cbac 746b1556 ntdll!NtWaitForMultipleObjects+0xc
01 0019cd40 746b1408 KERNELBASE!WaitForMultipleObjectsEx+0x136
02 0019cd5c 747ea02a KERNELBASE!WaitForMultipleObjects+0x18
03 0019d198 747e9ac6 kernel32!WerReportFaultInternal+0x545
04 0019d1a8 747ccf09 kernel32!WerReportFault+0x7a
05 0019d1b0 746c9f53 kernel32!BasepReportFault+0x19
06 0019d244 76fc2de5 KERNELBASE!UnhandledExceptionFilter+0x1b3
07 0019d2e8 76f8acd6 ntdll!LdrpLogFatalUserCallbackException+0x4d
08 0019d2f4 76f9d572 ntdll!KiUserCallbackExceptionHandler+0x26
09 0019d318 76f9d544 ntdll!ExecuteHandler2+0x26
0a 0019d3e0 76f8ad8f ntdll!ExecuteHandler+0x24
0b 0019d3e0 55403000 ntdll!KiUserExceptionDispatcher+0xf
WARNING: Stack unwind information not available. Following frames may be wrong.
0c 0019d8d0 55402faa VISLIB!Ordinal1+0x24f3b
0d 0019d914 5b85c67e VISLIB!Ordinal1+0x24ee5
0e 0019d940 5b85c638 MSO!Ordinal12138+0x10a
0f 0019d950 5b8e7620 MSO!Ordinal12138+0xc4
10 0019d964 5b8e7602 MSO!Ordinal19998+0x3bc
11 0019d97c 5bc938a6 MSO!Ordinal19998+0x39e
12 0019dbb0 5c240add MSO!Ordinal17238+0x25bef
13 0019ddec 65598ed1 MSO!Ordinal2007+0x1766
14 0019de78 655c5eaa VisioPlugin!DLLRegisterServer+0x43bf1
15 0019dfbc 555601db VisioPlugin!DLLRegisterServer+0x70bca
16 0019dfe8 5555fe61 VISLIB!Ordinal1+0x182116
17 0019e028 55421b7c VISLIB!Ordinal1+0x181d9c
18 0019e070 5549f1a9 VISLIB!Ordinal1+0x43ab7
19 0019e090 5549ebba VISLIB!Ordinal1+0xc10e4
1a 0019e0c0 5540dd14 VISLIB!Ordinal1+0xc0af5
1b 0019e110 55426168 VISLIB!Ordinal1+0x2fc4f
1c 0019e134 55425446 VISLIB!Ordinal1+0x480a3
1d 0019e20c 5549eace VISLIB!Ordinal1+0x47381
1e 0019e264 5549e90e VISLIB!Ordinal1+0xc0a09
1f 0019e28c 6571fb03 VISLIB!Ordinal1+0xc0849
20 0019e334 6571f6cc mfc90u!CWnd::OnWndMsg+0x410
21 0019e354 553ef572 mfc90u!CWnd::WindowProc+0x24
22 0019e370 6571e2f2 VISLIB!Ordinal1+0x114ad
23 0019e3d8 6571e57e mfc90u!AfxCallWndProc+0xa3
24 0019e3fc 553ef518 mfc90u!AfxWndProc+0x37
25 0019e440 553ef4d9 VISLIB!Ordinal1+0x11453
```

<sup>82</sup> <https://msdn.microsoft.com/en-us/library/bb787181.aspx>

```
26 0019e458 553ef49e VISLIB!Ordinal1+0x11414
27 0019e480 553ef338 VISLIB!Ordinal1+0x113d9
28 0019e49c 553ef2d6 VISLIB!Ordinal1+0x11273
29 0019e4c4 553ef107 VISLIB!Ordinal1+0x11211
2a 0019e528 75864923 VISLIB!Ordinal1+0x11042
2b 0019e554 75844790 user32!_InternalCallWinProc+0x2b
2c 0019e5fc 75844527 user32!UserCallWinProcCheckWow+0x1f0
2d 0019e638 71db7d40 user32!CallWindowProcW+0x97
2e 0019e6b8 71db7996 comctl32!CallNextSubclassProc+0x140
2f 0019e6d8 5b84d95a comctl32!DefSubclassProc+0x56
30 0019e720 5b84d7ad MSO!Ordinal1+0x25e
31 0019e74c 71db7db8 MSO!Ordinal1+0xb1
32 0019e7d0 71db7b61 comctl32!CallNextSubclassProc+0x1b8
33 0019e82c 75864923 comctl32!MasterSubclassProc+0xa1
34 0019e858 75844790 user32!_InternalCallWinProc+0x2b
35 0019e900 75844370 user32!UserCallWinProcCheckWow+0x1f0
36 0019e960 7584b179 user32!DispatchClientMessage+0xf0
37 0019e9a0 76f8ad66 user32!_fnDWORD+0x49
38 0019e9d8 75864dac ntdll!KiUserCallbackDispatcher+0x36
39 0019e9dc 75842ce8 user32!NtUserMessageCall+0xc
3a 0019ea68 758423ba user32!RealDefWindowProcWorker+0x148
3b 0019ea80 71f882ee user32!RealDefWindowProcW+0x5a
3c 0019eaa0 71f88145 uxtheme!DoMsgDefault+0x3a
3d 0019ebab 71f87bba uxtheme!OnDwpSysCommand+0x35
3e 0019eb1c 71f868d8 uxtheme!_ThemeDefWindowProc+0x6ca
3f 0019eb30 75842b66 uxtheme!ThemeDefWindowProcW+0x18
40 0019eb80 758415ee user32!DefWindowProcW+0x176
41 0019eb98 75851e3b user32!DefWindowProcWorker+0x2e
42 0019ec1c 758aa09b user32!DefFrameProcWorker+0xb7
43 0019ec34 55718ac5 user32!DefFrameProcW+0x1b
44 0019ec58 55708027 VISLIB!Ordinal1+0x33aa00
45 0019ec70 6571e3c1 VISLIB!Ordinal1+0x329f62
46 0019ec84 65725604 mfc90u!CWnd::Default+0x30
47 0019ec94 5549e617 mfc90u!CFrameWnd::OnSysCommand+0x50
48 0019ecb4 6571fd15 VISLIB!Ordinal1+0xc0552
49 0019ed64 6571f6cc mfc90u!CWnd::OnWndMsg+0x622
4a 0019ed84 553ef572 mfc90u!CWnd::WindowProc+0x24
4b 0019eda0 6571e2f2 VISLIB!Ordinal1+0x114ad
4c 0019ee08 6571e57e mfc90u!AfxCallWndProc+0xa3
4d 0019ee2c 553ef518 mfc90u!AfxWndProc+0x37
4e 0019ee70 553ef4d9 VISLIB!Ordinal1+0x11453
4f 0019ee88 553ef49e VISLIB!Ordinal1+0x11414
50 0019eeb0 553ef338 VISLIB!Ordinal1+0x113d9
51 0019eec5 553ef2d6 VISLIB!Ordinal1+0x11273
52 0019eef4 553ef107 VISLIB!Ordinal1+0x11211
53 0019ef58 75864923 VISLIB!Ordinal1+0x11042
54 0019ef84 75844790 user32!_InternalCallWinProc+0x2b
55 0019f02c 75844527 user32!UserCallWinProcCheckWow+0x1f0
56 0019f068 71db7d40 user32!CallWindowProcW+0x97
57 0019f0e8 71db7996 comctl32!CallNextSubclassProc+0x140
58 0019f108 5b84d95a comctl32!DefSubclassProc+0x56
59 0019f150 5b84d7ad MSO!Ordinal1+0x25e
5a 0019f17c 71db7db8 MSO!Ordinal1+0xb1
5b 0019f200 71db7b61 comctl32!CallNextSubclassProc+0x1b8
5c 0019f25c 75864923 comctl32!MasterSubclassProc+0xa1
5d 0019f288 75844790 user32!_InternalCallWinProc+0x2b
```

```
5e 0019f330 75844370 user32!UserCallWinProcCheckWow+0x1f0
5f 0019f390 7584b179 user32!DispatchClientMessage+0xf0
60 0019f3d0 76f8ad66 user32!__fnDWORD+0x49
61 0019f408 00000000 ntdll!KiUserCallbackDispatcher+0x36
```

Next, we applied **!mv** WinDbg command to the module name and followed its image path to rename it. After that, the problem disappeared. We call such modules **Foreign** because they were created not by the OS or the main process module vendors. Most likely these modules are either value-adding plugins or exposed **Message Hooks** (page 663).

## FPU Exception

This pattern sometimes happens where we least expect it. Here's an extract from one crash dump raw stack analysis showing exception context and record, and the usage of `r` WinDbg command variant to display FPU registers:

```
0:002> dps 056c1000 057c0000
[...]
057bdee0 00000008
057bdee4 00000000
057bdee8 057bed6c
057bdeec 0d6e3130
057bdef0 057c0000
057bdef4 057b9000
057bdef8 006e3138
057bdefc 057be200
057bdf00 7c90e48a ntdll!KiUserExceptionDispatcher+0xe
057bdf04 057bed6c
057bdf08 057bdf2c
057bdf0c 057bdf14
057bdf10 057bdf2c
057bdf14 c0000090
057bdf18 00000010
057bdf1c 00000000
057bdf20 79098cc0 mscorejit!Compiler::FlatFPIsSameAsFloat+0xd
057bdf24 00000001
057bdf28 00000000
057bdf2c 0001003f
057bdf30 00000000
057bdf34 00000000
057bdf38 00000000
057bdf3c 00000000
057bdf40 00000000
057bdf44 00000000
057bdf48 ffff1372
057bdf4c fffffda1
057bdf50 fffffbfff
[...]

0:002> .cxr 057bdf2c
eax=c0000090 ebx=00000000 ecx=c0000090 edx=00000000 esi=057be244 edi=001d4388
eip=79f5236b esp=057be1f8 ebp=057be200 iopl=0 nv up ei ng nz ac pe cy
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010297
mscorwks!SO TolerantBoundaryFilter+0x22:
79f5236b d9059823f579 fld dword ptr [mscorwks!_real (79f52398)] ds:0023:79f52398=40800000

0:002> .exr 057bdf14
ExceptionAddress: 79098cc0 (mscorejit!Compiler::FlatFPIsSameAsFloat+0x0000000d)
ExceptionCode: c0000090
ExceptionFlags: 00000010
NumberParameters: 1
Parameter[0]: 00000000
```

```
0:002> !error c0000090
Error code: (NTSTATUS) 0xc0000090 (3221225616) - {EXCEPTION} Floating-point invalid operation.

0:002> rMF
Last set context:
eax=c0000090 ebx=00000000 ecx=c0000090 edx=00000000 esi=057be244 edi=001d4388
eip=79f5236b esp=057be1f8 ebp=057be200 iopl=0 nv up ei ng nz ac pe cy
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010297
fpcw=1372: rn 64 pu-d- fpsw=FDA1: top=7 cc=1101 b-p--i fptw=BFFF
opcode=045D fpip=001b:79098cc0 fpdp=0023:057bea7c
st0=-1.#IND0000000000000000e+0000 st1= 0.006980626232475338220e-4916
st2= 6.543831490564206840810e-4932 st3=-0.003025663186207448300e+2614
st4= 2.00000000000000000000000000000e+0000 st5= 6.2914560000000000000000e+0006
st6= 1.00000000000000000000000000000e+0000 st7= 2.5000000000000000000000e-0001
mscorwks!SOTolerantBoundaryFilter+0x22:
79f5236b d9059823f579 fld dword ptr [mscorwks!_real (79f52398)] ds:0023:79f52398=40800000
```

## Frame Pointer Omission

This pattern is the most visible compiler optimization technique, and we can notice it in verbose stack traces:

```
0:000> kv
ChildEBP RetAddr
0012ee10 004737a7 application!Memcpy+0x17 (FPO: [3,0,2])
0012ef0c 35878c5b application!ProcessData+0x97 (FPO: [Uses EBP] [3,59,4])
WARNING: Frame IP not in any known module. Following frames may be wrong.
0012ef1c 72a0015b 0x35878c5b
0012ef20 a625e1b0 0x72a0015b
0012ef24 d938bcfe 0xa625e1b0
0012ef28 d4f91bb4 0xd938bcfe
0012ef2c c1c035ce 0xd4f91bb4
...
...
...
```

To recall, FPO is a compiler optimization where ESP register is used to address local variables and parameters instead of EBP. EBP may or may not be used for other purposes. When it is used, we notice

### FPO: [Uses EBP]

as in the trace above. For a description of other FPO number triplets, please see Debugging Tools for Windows help section “k, kb, kd, kp, kP, kv (Display Stack Backtrace)”.

Running the analysis command (**!analyze -v**) points to possible stack corruption:

```
PRIMARY_PROBLEM_CLASS: STACK_CORRUPTION

BUGCHECK_STR: APPLICATION_FAULT_STACK_CORRUPTION

FAULTING_IP:
application!Memcpy+17
00438637 f3a5 rep movs dword ptr es:[edi],dword ptr [esi]
```

Looking at EBP and ESP shows that they are mismatched:

```
0:000> r
eax=00000100 ebx=00a027f3 ecx=00000040 edx=0012ee58 esi=d938bcfe edi=0012ee58
eip=00438637 esp=0012ee0c ebp=00a02910 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
application!Memcpy+0x17:
00438637 f3a5 rep movs dword ptr es:[edi],dword ptr [esi] es:0023:0012ee58=00000010
ds:0023:d938bcfe=????????
```

We might think about **Local Buffer Overflow** pattern (page 611) here, but two top stack trace lines are in accordance with each other:

```
0:000> ub 004737a7
application!ProcessData+0x80:
00473790 cmp     eax,edi
00473792 jb      application!ProcessData+0x72 (00473782)
00473794 mov     ecx,dword ptr [esp+104h]
0047379b push    esi
0047379c lea     edx,[esp+38h]
004737a0 push    ecx
004737a1 push    edx
004737a2 call    application!Memcpy (00438620)
```

So perhaps EBP value differs greatly from ESP due to its usage as general purpose register and, in fact, there was no any stack corruption. Despite using public symbols, we have the instance of **Incorrect Stack Trace** pattern (page 499), and we might want to reconstruct it manually. Let's search for EBP value on the raw stack below the crash point:

```
0:000> !teb
TEB at 7ffd000
ExceptionList:          0012ffb0
StackBase:              00130000
StackLimit:             00126000
SubSystemTib:           00000000
FiberData:              00001e00
ArbitraryUserPointer:   00000000
Self:                   7ffd000
EnvironmentPointer:    00000000
ClientId:               0000660c . 00005890
RpcHandle:              00000000
Tls Storage:            00000000
PEB Address:            7ffd9000
LastErrorValue:         0
LastStatusValue:        0
Count Owned Locks:     0
HardErrorMode:          0

0:000> dds 00126000 00130000
00126000 00000000
00126004 00000000
00126008 00000000
...
...
...
0012eb0c 00a02910
0012eb10 7c90e25e ntdll!NtRaiseException+0xc
0012eb14 7c90eb15 ntdll!KiUserExceptionDispatcher+0x29
0012eb18 0012eb24
0012eb1c 0012eb40
0012eb20 00000000
0012eb24 c0000005
0012eb28 00000000
0012eb2c 00000000
0012eb30 00438637 application!Memcpy+0x17
0012eb34 00000002
0012eb38 00000000
0012eb3c d938bcfe
```

```
...
...
...
0012ebf4 00a02910
0012ebf8 00438637 application!MemCopy+0x17
0012ebfc 0000001b
...
...
...
0012f134 00436323 application!ConstructInfo+0x113
0012f138 00a02910
0012f13c 00000011c
...
...
...
```

Let's see what functions *ConstructInfo* calls:

```
0:000> ub 00436323
application!ConstructInfo+0x103
00436313 lea      edx,[esp+10h]
00436317 push     edx
00436318 push     eax
00436319 push     30h
0043631b push     ecx
0043631c push     ebx
0043631d push     ebp
0043631e call    application!EnvelopeData (00438bf0)
```

We notice EBP was pushed prior to calling *EnvelopeData* function. If we disassemble this function, we see that it calls *ProcessData* function from our partial stack trace:

```
0:000> uf 00438bf0
application!EnvelopeData:
00438bf0 sub      esp,1F4h
00438bf6 push     ebx
00438bf7 mov      ebx,dword ptr [esp+20Ch]
00438bfe test    ebx,ebx
00438c00 push     ebp
...
...
...
00438c76 rep stos byte ptr es:[edi]
00438c78 lea      eax,[esp+14h]
00438c7c push     eax
00438c7d push     ebp
00438c7e call    application!ProcessData (00473710)
00438c83 pop      edi
00438c84 pop      esi
```

Let's try the elaborate form of **k** command and supply it with custom ESP and EBP values pointing to

```
0012f134 00436323 application!ConstructInfo+0x113
```

and also EIP of the fault:

```
0:000> k L=0012f134 0012f134 00438637
ChildEBP RetAddr
0012f1cc 00435a65 application!Memcpy+0x17
0012f28c 0043532e application!ClientHandleServerRequest+0x395
0012f344 00434fcd application!Accept+0x23
0012f374 0042e4f3 application!DataArrival+0x17d
0012f43c 0041aea9 application!ProcessInput+0x98
0012ff0c 0045b278 application!AppMain+0xda
0012ff24 0041900e application!WinMain+0x78
0012ffc0 7c816fd7 application!WinMainCRTStartup+0x134
0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

We see that although it misses some initial frames after the *Memcpy* function, we aided WinDbg to walk to the bottom of the stack and reconstruct the plausible stack trace for us.

## Frozen Process

Looks like Windows 8 reuses the debugging concept of a frozen thread for the so-called a “deeply frozen” process:

```
0: kd> !sprocess 2
Dumping Session 2
[...]
PROCESS ffffffa8002cb2940
SessionId: 2 Cid: 0c80 Peb: 7f6c41dd000 ParentCid: 0288
DeepFreeze
DirBase: 2ef45000 ObjectTable: fffff8a002f215c0 HandleCount: <Data Not Accessible>
Image: iexplore.exe
[...]

0: kd> dt nt!_KPROCESS ffffffa8002cb2940
+0x000 Header : _DISPATCHER_HEADER
+0x018 ProfileListHead : _LIST_ENTRY [ 0xffffffa80`02cb2958 - 0xffffffa80`02cb2958 ]
+0x028 DirectoryTableBase : 0x2ef45000
+0x030 ThreadListHead : _LIST_ENTRY [ 0xffffffa80`01e4edf8 - 0xffffffa80`01f5bbf8 ]
+0x040 ProcessLock : 0
+0x044 Spare0 : 0
+0x048 Affinity : _KAFFINITY_EX
+0x0f0 ReadyListHead : _LIST_ENTRY [ 0xffffffa80`02cb2a30 - 0xffffffa80`02cb2a30 ]
+0x100 SwapListEntry : _SINGLE_LIST_ENTRY
+0x108 ActiveProcessors : _KAFFINITY_EX
+0x1b0 AutoAlignment : 0y0
+0x1b0 DisableBoost : 0y0
+0x1b0 DisableQuantum : 0y0
+0x1b0 AffinitySet : 0y0
+0x1b0 DeepFreeze : 0y1
+0x1b0 TimerVirtualization : 0y1
+0x1b0 ActiveGroupsMask : 0y000000000000000000000001 (0x1)
+0x1b0 ReservedFlags : 0y000000 (0)
+0x1b0 ProcessFlags : 0n112
+0x1b4 BasePriority : 8 "
+0x1b5 QuantumReset : 6 "
+0x1b6 Visited : 0 "
+0x1b7 Flags : _KEXECUTE_OPTIONS
+0x1b8 ThreadSeed : [20] 0
+0x208 IdealNode : [20] 0
+0x230 IdealGlobalNode : 0
+0x232 Spare1 : 0
+0x234 StackCount : _KSTACK_COUNT
+0x238 ProcessListEntry : _LIST_ENTRY [ 0xffffffa80`03816b78 - 0xffffffa80`02cc2b78 ]
+0x248 CycleTime : 0x225078
+0x250 ContextSwitches : 0x22
+0x258 SchedulingGroup : (null)
+0x260 FreezeCount : 0
+0x264 KernelTime : 0
+0x268 UserTime : 0
+0x26c LdtFreeSelectorHint : 0
+0x26e LdtTableLength : 0
+0x270 LdtSystemDescriptor : _KGDTENTRY64
```

```
+0x280 LdtBaseAddress : (null)
+0x288 LdtProcessLock : _FAST_MUTEX
+0x2c0 InstrumentationCallback : (null)
```

We also see that all its threads have a freeze count 1:

```
0: kd> !process ffffffa8002cb2940 2
[...]
THREAD ffffffa8001e4eb00 Cid 0c80.0514 Peb: 000007f6c41de000 Win32Thread: fffff901000e5b90 WAIT:
(Suspended) KernelMode Non-Alertable
FreezeCount 1
fffffa8001e4ede0 NotificationEvent
[...]

THREAD ffffffa800219c080 Cid 0c80.0d88 Peb: 000007f6c41db000 Win32Thread: fffff90103f206e0 WAIT:
(Suspended) KernelMode Non-Alertable
FreezeCount 1
fffffa800219c360 NotificationEvent
[...]

0: kd> dt _KTHREAD ffffffa800219c080
nt!_KTHREAD
+0x000 Header : _DISPATCHER_HEADER
+0x018 SListFaultAddress : (null)
+0x020 QuantumTarget : 0x18c26200
+0x028 InitialStack : 0xfffff880`1548ddd0 Void
+0x030 StackLimit : 0xfffff880`15488000 Void
+0x038 StackBase : 0xfffff880`1548e000 Void
+0x040 ThreadLock : 0
+0x048 CycleTime : 0x15ca97c8
+0x050 CurrentRunTime : 0
+0x054 ExpectedRunTime : 0xd77e
+0x058 KernelStack : 0xfffff880`1548d430 Void
+0x060 StateSaveArea : 0xfffff880`1548de00 _XSAVE_FORMAT
+0x068 SchedulingGroup : (null)
+0x070 WaitRegister : _KWAIT_STATUS_REGISTER
+0x071 Running : 0 ''
+0x072 Alerted : [2] ""
+0x074 KernelStackResident : 0y1
+0x074 ReadyTransition : 0y0
+0x074 ProcessReadyQueue : 0y0
+0x074 WaitNext : 0y0
+0x074 SystemAffinityActive : 0y0
+0x074 Alertable : 0y0
+0x074 CodePatchInProgress : 0y0
+0x074 UserStackWalkActive : 0y0
+0x074 ApcInterruptRequest : 0y0
+0x074 QuantumEndMigrate : 0y0
+0x074 UmsDirectedSwitchEnable : 0y0
+0x074 TimerActive : 0y0
+0x074 SystemThread : 0y0
+0x074 ProcessDetachActive : 0y0
+0x074 CalloutActive : 0y0
+0x074 ScbReadyQueue : 0y0
+0x074 ApcQueueable : 0y1
+0x074 ReservedStackInUse : 0y0
```

```
+0x074 UmsPerformingSyscall : 0y0
+0x074 Reserved          : 0y00000000000000 (0)
+0x074 MiscFlags         : 0n65537
+0x078 AutoAlignment     : 0y0
+0x078 DisableBoost      : 0y0
+0x078 UserAffinitySet   : 0y0
+0x078 AlertedByThreadId: 0y0
+0x078 QuantumDonation   : 0y0
+0x078 EnableStackSwap   : 0y1
+0x078 GuiThread         : 0y1
+0x078 DisableQuantum    : 0y0
+0x078 ChargeOnlyGroup   : 0y0
+0x078 DeferPreemption   : 0y0
+0x078 QueueDeferPreemption: 0y0
+0x078 ForceDeferSchedule: 0y0
+0x078 ExplicitIdealProcessor: 0y0
+0x078 FreezeCount       : 0y1
+0x078 EtwStackTraceApcInserted: 0y00000000 (0)
+0x078 ReservedFlags     : 0y0000000000 (0)
+0x078 ThreadFlags        : 0n8288
+0x07c Spare0            : 0
+0x080 SystemCallNumber   : 0x87
+0x084 Spare1            : 0
+0x088 FirstArgument      : 0x00000000`0000017c Void
+0x090 TrapFrame          : (null)
+0x098 ApcState           : _KAPC_STATE
+0x098 ApcStateFill       : [43] "???""
+0x0c3 Priority           : 8 "
+0x0c4 UserIdealProcessor  : 1
+0x0c8 WaitStatus          : 0n256
+0x0d0 WaitBlockList       : 0xfffffa80`0219c1c0 _KWAIT_BLOCK
+0x0d8 WaitListEntry       : _LIST_ENTRY [ 0xfffffa80`0418a458 - 0xfffff880`009eb300 ]
+0x0d8 SwapListEntry       : _SINGLE_LIST_ENTRY
+0x0e8 Queue               : 0xfffffa80`03da4bc0 _KQUEUE
+0x0f0 Teb                 : 0x000007f6`c41db000 Void
+0x0f8 RelativeTimerBias   : 0x00000001`8b165f54
+0x100 Timer               : _KTIMER
+0x140 WaitBlock           : [4] _KWAIT_BLOCK
+0x140 WaitBlockFill14     : [20] "h???""
+0x154 ContextSwitches    : 0x1817
+0x140 WaitBlockFill15     : [68] "h???""
+0x184 State               : 0x5 "
+0x185 NpxState            : 1 "
+0x186 WaitIrql            : 0 "
+0x187 WaitMode             : 0 "
+0x140 WaitBlockFill16     : [116] "h???""
+0x1b4 WaitTime             : 0xf0172e
+0x140 WaitBlockFill17     : [164] "h???""
+0x1e4 KernelApcDisable    : 0n0
+0x1e6 SpecialApcDisable   : 0n0
+0x1e4 CombinedApcDisable  : 0
+0x140 WaitBlockFill18     : [40] "h???""
+0x168 ThreadCounters      : (null)
+0x140 WaitBlockFill19     : [88] "h???""
+0x198 XStateSave          : (null)
+0x140 WaitBlockFill10     : [136] "h???""
```

```
+0x1c8 Win32Thread      : 0xfffff901`03f206e0 Void
+0x140 WaitBlockFill11  : [176]  "h???"  

+0x1f0 Ucb              : (null)
+0x1f8 Uch              : (null)
+0x200 TebMappedLowVa   : (null)
+0x208 QueueListEntry   : _LIST_ENTRY [ 0xfffffa80`02ccf408 - 0xfffffa80`03da4bf0 ]
+0x218 NextProcessor    : 0
+0x21c DeferredProcessor: 1
+0x220 Process          : 0xfffffa80`02cb2940 _KPROCESS
+0x228 UserAffinity     : _GROUP_AFFINITY
+0x228 UserAffinityFill: [10]  "???"  

+0x232 PreviousMode    : 1 "
+0x233 BasePriority     : 8 "
+0x234 PriorityDecrement: 0 "
+0x234 ForegroundBoost  : 0y0000
+0x234 UnusualBoost    : 0y0000
+0x235 Preempted        : 0 "
+0x236 AdjustReason     : 0 "
+0x237 AdjustIncrement  : 0 "
+0x238 Affinity         : _GROUP_AFFINITY
+0x238 AffinityFill     : [10]  "???"  

+0x242 ApcStateIndex    : 0 "
+0x243 WaitBlockCount   : 0x1 "
+0x244 IdealProcessor   : 1
+0x248 ApcStatePointer  : [2] 0xfffffa80`0219c118 _KAPC_STATE
+0x258 SavedApcState   : _KAPC_STATE
+0x258 SavedApcStateFill: [43]  "???"  

+0x283 WaitReason       : 0x5 "
+0x284 SuspendCount     : 0 "
+0x285 Saturation        : 0 "
+0x286 SListFaultCount  : 0
+0x288 SchedulerApc     : _KAPC
+0x288 SchedulerApcFill0: [1]  "??????"  

+0x289 ResourceIndex    : 0x1 "
+0x288 SchedulerApcFill1: [3]  "???"  

+0x28b QuantumReset     : 0x6 "
+0x288 SchedulerApcFill2: [4]  "???"  

+0x28c KernelTime        : 7
+0x288 SchedulerApcFill3: [64]  "???"  

+0x2c8 WaitPrcb          : (null)
+0x288 SchedulerApcFill4: [72]  "???"  

+0x2d0 LegoData          : (null)
[...]
```

This is different when a process is under a debugger, and all its threads are frozen except the one that communicates to the debugger like in the case study<sup>83</sup>. In Windows 8 this happens, for example, when we switch to a desktop from IE launched from the start page. Then we would see shortly that iexplore.exe process changes from *Running* to *Suspended* in Task Manager *Details* page. This pattern covers both the new feature and a debugged process case.

---

<sup>83</sup> "Stack Trace Collection, Suspended Threads, Not My Version, Special Process, Main Thread and Blocked LPC Chain Threads", Memory Dump Analysis Anthology, Volume 4, page 204

**G**

## Ghost Thread

Sometimes **Wait Chains** (page 1092) such as involving critical sections (page 1086) may have **Missing Thread** (page 683) endpoint. However, in some cases, we might see **Ghost Thread** whose TID was reused by subsequent thread creation in a different process. For example, critical section structure may refer to such TID as in the example below.

```
// Critical section from LSASS process

THREAD ffffffa803431cb50 Cid 03e8.2718 Teb: 000007fffff80000 Win32Thread: 0000000000000000 WAIT:
(UserRequest) UserMode Non-Alertable
ffffffa80330e0500 SynchronizationEvent
Impersonation token: ffffff8a00b807060 (Level Impersonation)
Owning Process      ffffffa8032354c40   Image: lsass.exe
Attached Process    N/A                  Image: N/A
Wait Start TickCount 107175            Ticks: 19677 (0:00:05:06.963)
Context Switch Count 2303              IdealProcessor: 1
UserTime             00:00:00.218
KernelTime           00:00:00.109
Win32 Start Address ntdll!TppWorkerThread (0x0000000076e1f2e0)
Stack Init ffffff88008e5fdb0 Current ffffff88008e5f900
Base ffffff88008e60000 Limit ffffff88008e5a000 Call 0
Priority 10 BasePriority 10 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
Kernel stack not resident.
Child-SP RetAddr Call Site
ffffff880`08e5f940 ffffff800`01c7cf72 nt!KiSwapContext+0x7a
ffffff880`08e5fa80 ffffff800`01c8e39f nt!KiCommitThreadWait+0x1d2
ffffff880`08e5fb10 ffffff800`01f7fe3e nt!KeWaitForSingleObject+0x19f
ffffff880`08e5fb00 ffffff800`01c867d3 nt!NtWaitForSingleObject+0xde
ffffff880`08e5fc20 00000000`76e5067a nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ ffffff880`08e5fc20)
00000000`0427cca8 00000000`76e4d808 ntdll!NtWaitForSingleObject+0xa
00000000`0427ccb0 00000000`76e4d6fb ntdll!RtlpWaitOnCriticalSection+0xe8
00000000`0427cd50 000007fe f46a4afe ntdll!RtlEnterCriticalSection+0xd1
[...]

1: kd> .process /r /p ffffffa8032354c40
Implicit process is now ffffffa80`32353b30
Loading User Symbols
```

```

1: kd> !cs -l -o -s
-----
DebugInfo      = 0x0000000003475220
Critical section = 0x0000000003377740 (+0x3377740)
LOCKED
LockCount      = 0x10
WaiterWoken     = No
OwningThread    = 0x000000000000004e4
RecursionCount   = 0x0
LockSemaphore   = 0x0
SpinCount       = 0x0000000000000000
OwningThread    = .thread ffffffa80344e4c00
[...]

// The "owner" thread is from winlogon.exe

1: kd> !thread ffffffa80344e4c00 1f
THREAD ffffffa80344e4c00 Cid 21d0.14e4 Teb: 000007fffffae000 Win32Thread: fffff900c0998c20 WAIT:
(WrUserRequest) UserMode Non-Alertable
ffffffa80355817d0 SynchronizationEvent
Not impersonating
DeviceMap        fffff8a0000088f0
Owning Process   ffffffa8034ff77c0      Image: winLogon.exe
[...]

```

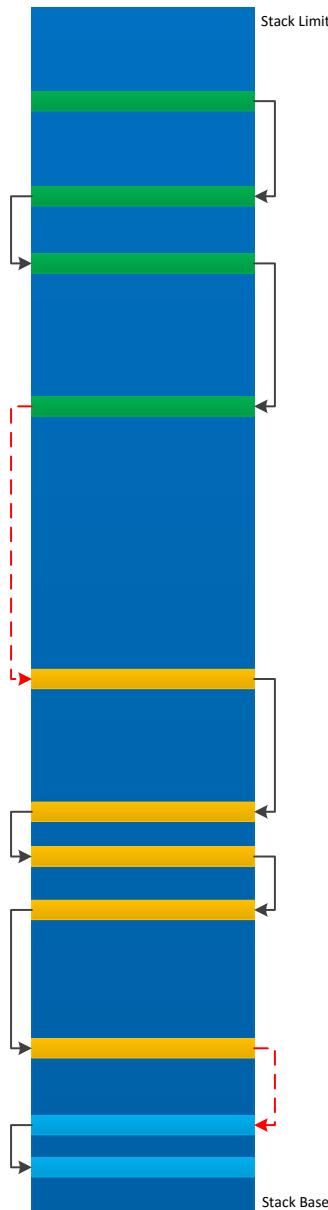
A PML (Process Monitor) log was recorded before the complete memory dump was forced, and it clearly shows **Glued Activity** trace analysis pattern<sup>84</sup>. LSASS owned the thread but then the thread exited, and 2 other processes subsequently reused its TID.

---

<sup>84</sup> Glued Activity, Memory Dump Analysis Anthology, Volume 6, page 250

## Glued Stack Trace

Sometimes we have **Truncated Stack Trace** (page 1015) and need to perform manual stack trace reconstruction<sup>85</sup> of the missing part to get approximate full stack trace. Often we are only able to reconstruct some parts and glue them together perhaps with some missing intermediate frames:



<sup>85</sup> Manual Stack Trace Reconstruction, Memory Dump Analysis Anthology, Volume 1, page 157

For example, we have this truncated stack trace due to the lack of symbols:

```
1: kd> k
ChildEBP RetAddr
97543b6c 85adf579 nt!KiTrap0E+0x2ac
WARNING: Stack unwind information not available. Following frames may be wrong.
97543be8 85adf770 myfault+0x579
97543bf4 85adf7fc myfault+0x770
97543c2c 81827ecf myfault+0x7fc
97543c44 81988f65 nt!IoCallDriver+0x63
97543c64 81989f25 nt!IopSynchronousServiceTail+0x1e0
97543d00 8198ee8d nt!IopXxxControlFile+0x6b7
97543d34 8188c96a nt!NtDeviceIoControlFile+0x2a
97543d34 77510f34 nt!KiFastCallEntry+0x12a
0012f9a0 7750f850 ntdll!KiFastSystemCallRet
0012f9a4 77417c92 ntdll!NtDeviceIoControlFile+0xc
0012fa04 00401a5b kernel32!DeviceIoControl+0x14a
0012fa94 7700becf NotMyfault+0x1a5b
0012facc 00000000 USER32!xxxDrawButton+0xc1
```

Manual stack reconstruction brings this fragment:

```
1: kd> k L=0012fb94 0012fb94 0012fb94
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
0012fb94 77001ae8 0x12fb94
0012fc0c 7700286a USER32!UserCallWinProcCheckWow+0x14b
0012fc4c 77002bba USER32!SendMessageWorker+0x4b7
0012fc6c 7700c6b4 USER32!SendMessageW+0x7c
0012fc84 7700c7c9 USER32!xxxButtonNotifyParent+0x41
0012fcfa0 7700c7e8 USER32!xxxBNReleaseCapture+0xf7
0012fd24 7701632e USER32!ButtonWndProcWorker+0x910
0012fd44 77001a10 USER32!ButtonWndProcA+0x4c
0012fd70 77001ae8 USER32!InternalCallWinProc+0x23
0012fde8 77002a47 USER32!UserCallWinProcCheckWow+0x14b
0012fe4c 77002a98 USER32!DispatchMessageWorker+0x322
0012fe5c 76ff11fc USER32!DispatchMessageW+0xf
0012fe80 76fe98d2 USER32!IsDialogMessageW+0x586
0012fea0 00401cc9 USER32!IsDialogMessageA+0xff
0012ff10 004022ec NotMyfault+0x1cc9
00000000 00000000 NotMyfault+0x22ec
```

And finally we get the 3rd usual thread start fragment:

```
1: kd> k L=0012ffa0 0012ffa0 0012ffa0
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
0012ffa0 77413833 0x12ffa0
0012ffac 774ea9bd kernel32!BaseThreadInitThunk+0xe
0012ffec 00000000 ntdll!_RtlUserThreadStart+0x23
```

Gluing them together, we get this approx. stack trace:

```
97543b6c 85adf579 nt!KiTrap0E+0x2ac
WARNING: Stack unwind information not available. Following frames may be wrong.
97543be8 85adf770 myfault+0x579
97543bf4 85adf7fc myfault+0x770
97543c2c 81827ecf myfault+0x7fc
97543c44 81988f65 nt!IoCallDriver+0x63
97543c64 81989f25 nt!IoPynchronousServiceTail+0x1e0
97543d00 8198ee8d nt!IoPxxxControlFile+0x6b7
97543d34 8188c96a nt!NtDeviceIoControlFile+0x2a
97543d34 77510f34 nt!KiFastCallEntry+0x12a
0012f9a0 7750f850 ntdll!KiFastSystemCallRet
0012f9a4 77417c92 ntdll!NtDeviceIoControlFile+0xc
0012fa04 00401a5b kernel32!DeviceIoControl+0x14a
0012fa94 7700becf NotMyfault+0x1a5b
0012fc0c 7700286a USER32!UserCallWinProcCheckWow+0x14b
0012fc4c 77002bba USER32!SendMessageWorker+0x4b7
0012fc6c 7700c6b4 USER32!SendMessageW+0x7c
0012fc84 7700c7c9 USER32!xxxButtonNotifyParent+0x41
0012fc a0 7700c7e8 USER32!xxxBNReleaseCapture+0xf7
0012fd24 7701632e USER32!ButtonWndProcWorker+0x910
0012fd44 77001a10 USER32!ButtonWndProcA+0x4c
0012fd70 77001ae8 USER32!InternalCallWinProc+0x23
0012fde8 77002a47 USER32!UserCallWinProcCheckWow+0x14b
0012fe4c 77002a98 USER32!DispatchMessageWorker+0x322
0012fe5c 76ff11fc USER32!DispatchMessageW+0xf
0012fe80 76fe98d2 USER32!IsDialogMessageW+0x586
0012fea0 00401cc9 USER32!IsDialogMessageA+0xff
0012ff10 004022ec NotMyfault+0x1cc9
0012ffac 774ea9bd kernel32!BaseThreadInitThunk+0xe
0012ffec 00000000 ntdll!_RtlUserThreadStart+0x23
```

# H

## Handle Leak

Although **Handle Leak** may lead to **Insufficient Memory** (page 526), it is not always the case especially if pool structures are small such as events. So **Handle Leak** pattern covers high memory usage (including fat structures), high handle counts and also abnormal differences in allocations and deallocations. As an example of the latter here is a nonpaged pool leak of Event objects and correlated pooltag ABCD. Although memory usage footprint is small compared with other nonleaking pooltags we see that the difference between *Allocs* and *Frees* is surely abnormal and correlating with high handle counts:

```
0: kd> !poolused 3
Sorting by NonPaged Pool Consumed

Pool Used:
NonPaged          Paged
Tag    Allocs   Frees    Diff    Used    Allocs   Frees    Diff    Used
[...]
ABCD  1778517  1704538  73979  4734656      0        0        0        0 UNKNOWN pooltag 'ABCD',
please update pooltag.txt
Even   6129633  6063728  65905  4224528      0        0        0        0 Event objects
[...]

0: kd> !process 0 0

[...]

PROCESS d2b85360 SessionId: 2 Cid: 1bf4 Peb: 7ffdf000 ParentCid: 1688
DirBase: 7d778dc0 ObjectTable: e53dda08 HandleCount: 18539.
Image: AppA.exe

PROCESS b2fcfd670 SessionId: 2 Cid: 0818 Peb: 7ffd4000 ParentCid: 1688
DirBase: 7d778400 ObjectTable: b3ffd8c0 HandleCount: 36252.
Image: AppB.exe

[...]
```

## Handle Limit

### GDI

#### Kernel Space

Among various memory leaks leading to **Insufficient Memory** pattern (page 523), there is so-called session pool leak briefly touched in the pattern about kernel pool leaks (page 535). It also involves GDI handles and structures allocated per user session that has the limit on how many of them can be created and this pattern should rather be called **Handle Limit**. Such leaks can result in poor visual application behavior after some time when drawing requests are not satisfied anymore. In severe cases, when the same bugs are present in a display driver, it can result in bugchecks like

```
BugCheck AB: SESSION_HAS_VALID_POOL_ON_EXIT
```

or, if a handle allocation request was not satisfied, it may result in a NULL pointer stored somewhere with the subsequent **Invalid Pointer** access (page 589):

```
SYSTEM_THREAD_EXCEPTION_NOT_HANDLED (7e)
```

```
CONTEXT: b791e010 -- (.cxr 0xfffffffffb791e010)
eax=00000000 ebx=bc43d004 ecx=a233add8 edx=00000000 esi=bc430fff edi=00000000
eip=bfe7d380 esp=b791e3dc ebp=b791e480 iopl=0 nv up ei pl zr na pe nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010246
DisplayDriver+0x3e380:
bfe7d380 8a4702    mov     al,byte ptr [edi+2]        ds:0023:00000002=??
```

We can write three Win32 applications in Visual C++ that simulate GDI leaks. All of them create GDI objects in a loop and select them into their current graphics device context (DC) on Windows Server 2003 x64 SP2. Before running them, we get the following session paged pool statistics:

```
1kd> !poolused c

Sorting by Session Paged Pool Consumed

Pool Used:
      NonPaged          Paged
  Tag   Allocs   Used   Allocs   Used
NV_x       0       0      5 14024704 UNKNOWN pooltag 'NV_x', please update pooltag.txt
BIG        0       0     257  3629056 Large session pool allocations (ntos\ex\pool.c) , Binary:
nt!mm
NV        0       0     203  1347648 nVidia video driver
Ttfdf      0       0     233  1053152 TrueType Font driver
Gh05       0       0     391  1050400 Gdi Handle manager specific object types: defined in
w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
Gla1       0       0     348  785088 Gdi handle manager specific object types allocated from
lookaside memory: defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
Gcac       0       0      25  640880 Gdi glyph cache
Gla5       0       0     631  323072 Gdi handle manager specific object types allocated from
```

```

lookaside memory: defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
Gdrs      0      0      33  172288 Gdi GDITAG_DRVSUP
Gla:      0      0      212  139072 Gdi handle manager specific object types allocated from
Lookaside memory: defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
Gla4      0      0      487  116880 Gdi handle manager specific object types allocated from
lookaside memory: defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
Usti      0      0      148  97088 THREADINFO , Binary: win32k!AllocateW32Thread
Gla8      0      0      383  91920 Gdi handle manager specific object types allocated from
lookaside memory: defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
Gla@     0      0      339  70512 Gdi handle manager specific object types allocated from
Lookaside memory: defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
Gbaf      0      0      48   67584 UNKNOWN pooltag 'Gbaf', please update pooltag.txt
Knlf      0      0      20   66496 UNKNOWN pooltag 'knlf', please update pooltag.txt
GDev      0      0      7    57344 Gdi pdev
Usqu      0      0      152  53504 Q , Binary: win32k!InitQEntryLookaside
Uscu      0      0      334  53440 CURSOR , Binary: win32k!_CreateEmptyCursorObject
Bmfd      0      0      21   50224 Font related stuff
Uspi      0      0      153  40000 PROCESSINFO , Binary: win32k!MapDesktop
Gfnt      0      0      47   39856 UNKNOWN pooltag 'Gfnt', please update pooltag.txt
Ggb       0      0      34   39088 Gdi glyph bits
Gh08      0      0      33   38656 Gdi Handle manager specific object types: defined in
w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
Ghab      0      0      228  32832 Gdi Handle manager specific object types: defined in
w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
Ovfl      0      0      1    32768 The internal pool tag table has overflowed - usually this is a
result of nontagged allocations being made
Gpff      0      0      88   27712 Gdi physical font file
Gpfe      0      0      88   27600 UNKNOWN pooltag 'Gpfe', please update pooltag.txt
Thdd      0      0      1    20480 DirectDraw/3D handle manager table
Gebr      0      0      17   19776 Gdi ENGBRUSH
Gh0@     0      0      86  19264 Gdi Handle manager specific object types: defined in
w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
Gsp       0      0      79   18960 Gdi sprite
HT40      0      0      2    16384 UNKNOWN pooltag 'HT40', please update pooltag.txt
Gpat      0      0      4    16192 UNKNOWN pooltag 'Gpat', please update pooltag.txt
Ggls      0      0      169  12944 Gdi glyphset
Glnk      0      0      371  11872 Gdi PFILELINK
Gldv      0      0      9    11248 Gdi Ldev
Gffv      0      0      84   9408 Gdi FONTPFILEVIEW
Gfsb      0      0      1    8192 Gdi font substitution list
Uskt      0      0      2    7824 KBDTABLE , Binary: win32k!ReadLayoutFile
Gh04      0      0      7    5856 Gdi Handle manager specific object types: defined in
w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
Gdcf      0      0      51   5712 UNKNOWN pooltag 'Gdcf', please update pooltag.txt
Gh0<     0      0      88   5632 Gdi Handle manager specific object types: defined in
w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
Gglb      0      0      1    4096 Gdi temp buffer
Ustm      0      0      30   3360 TIMER , Binary: win32k!InternalSetTimer
Gspm      0      0      39   3120 UNKNOWN pooltag 'Gspm', please update pooltag.txt
Usac      0      0      16   3056 ACCEL , Binary: win32k!_CreateAcceleratorTable
Usqm      0      0      25   2800 QMSG , Binary: win32k!InitQEntryLookaside
Ghas      0      0      3    2592 Gdi Handle manager specific object types: defined in
w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
Uscl      0      0      20   2128 CLASS , Binary: win32k!ClassAlloc
Uswl      0      0      1    2032 WINDOWLIST , Binary: win32k!BuildHwndList
Gmul      0      0      19   1520 UNKNOWN pooltag 'Gmul', please update pooltag.txt

```

Dddp	0	0	8	1472 UNKNOWN pooltag 'Dddp', please update pooltag.txt
Ggdv	0	0	8	1472 Gdi GDITAG_GDEVICE
UsDI	0	0	4	1408 DEVICEINFO , Binary: win32k!CreateDeviceInfo
Vtfd	0	0	4	1312 Font file/context
Ushk	0	0	20	1280 HOTKEY , Binary: win32k!_RegisterHotKey
Gspr	0	0	3	1264 Gdi sprite grow range
Gtmw	0	0	13	1248 Gdi TMW_INTERNAL
Gxlt	0	0	8	1152 Gdi Xlate
Gpft	0	0	2	944 Gdi font table
Uspp	0	0	5	944 PNP , Binary: win32k!AllocateAndLinkHidTLCInf
Ussm	0	0	7	896 SMS , Binary: win32k!InitSMSLookaside
Gdbr	0	0	10	800 Gdi driver brush realization
Usdc	0	0	8	768 DCE , Binary: win32k!CreateCacheDC
Usct	0	0	12	768 CHECKPT , Binary: win32k!CkptRestore
Usim	0	0	2	736 IME , Binary: win32k!CreateInputContext
Usci	0	0	3	720 CLIENTTHREADINFO , Binary: win32k!InitSystemThread
Gh09	0	0	1	640 Gdi Handle manager specific object types: defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
Ussy	1	80	4	608 SYSTEM , Binary: win32k!xxxDesktopThread
Urdr	0	0	9	576 REDIRECT , Binary: win32k!SetRedirectionBitmap
Uswd	0	0	2	576 WINDOW , Binary: win32k!xxxCreateWindowEx
Uscb	0	0	3	544 CLIPBOARD , Binary: win32k!_ConvertMemHandle
Gcsl	0	0	1	496 Gdi string resource script names
Ustx	0	0	10	496 TEXT , Binary: win32k!NtUserDrawCaptionTemp
Ussw	0	0	1	496 SWP , Binary: win32k!_BeginDeferWindowPos
Gdev	0	0	2	480 Gdi GDITAG_DEVMODE
Usih	0	0	10	480 IMEHOTKEY , Binary: win32k!SetImeHotKey
Gdrv	0	0	1	368 UNKNOWN pooltag 'Gdrv', please update pooltag.txt
GVdv	0	0	1	320 UNKNOWN pooltag 'GVdv', please update pooltag.txt
Gmap	0	0	1	320 Gdi font map signature table
Uskb	0	0	2	288 KBDLAYOUT , Binary: win32k!xxxLoadKeyboardLayoutEx
Uskf	0	0	2	288 KBDFILE , Binary: win32k!LoadKeyboardLayoutFile
Uswe	0	0	2	224 WINEVENT , Binary: win32k!_SetWinEventHook
Gddf	0	0	2	224 Gdi ddraw driver heaps
Gddv	0	0	2	192 Gdi ddraw driver video memory list
GFil	0	0	2	192 Gdi engine descriptor list
Gwdw	0	0	2	96 Gdi watchdog support objects , Binary: win32k.sys
Usd9	0	0	1	80 DDE9 , Binary: win32k!xxxCsDdeInitialize
Gvds	0	0	1	64 UNKNOWN pooltag 'Gvds', please update pooltag.txt
GreA	0	0	1	64 UNKNOWN pooltag 'GreA', please update pooltag.txt
Usse	0	0	1	48 SECURITY , Binary: win32k!SetDisconnectDesktopSecu
Usvl	0	0	1	48 VWPL , Binary: win32k!VWPLAdd
Mdxg	1	112	0	0 UNKNOWN pooltag 'Mdxg', please update pooltag.txt
Gini	3	128	0	0 Gdi fast mutex
Usev	1	64	0	0 EVENT , Binary: win32k!xxxPollAndWaitForSingle0
Gdde	3	240	0	0 Gdi ddraw event
TOTAL	9	624	6256	24408704

The first application leaks fonts:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)
    {
        case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps);
            while (true)
            {
                HFONT hf = CreateFont(10, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, L"Arial");
                SelectObject(ps.hdc, hf);
            }
            EndPaint(hWnd, &ps);
            break;
    }
}
```

We clearly see the leak as the greatly increased the number of allocations for "Gla:" pool tag:

Pool Used:					
	NonPaged		Paged		
Tag	Allocs	Used	Allocs	Used	
NV_X	0	0	5	14024704	UNKNOWN pooltag 'NV_X', please update pooltag.txt
<b>Gla:</b>	<b>0</b>	<b>0</b>	<b>10194</b>	<b>6687264</b>	<b>Gdi handle manager specific object types allocated from</b>
<i>Lookaside memory: defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys</i>					
BIG	0	0	248	3690496	Large session pool allocations (ntos\ex\pool.c) , Binary: nt!mm
NV	0	0	203	1347648	nVidia video driver
Gh05	0	0	396	1057888	Gdi Handle manager specific object types: defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
Ttfd	0	0	226	1043264	TrueType Font driver

The second application leaks fonts and pens:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)
    {
        case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps);
            while (true)
            {
                HFONT hf = CreateFont(10, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, L"Arial");
                HPEN hp = CreatePen(0, 10, RGB(10, 20, 30));
                SelectObject(ps.hdc, hf);
                SelectObject(ps.hdc, hp);
            }
    }
}
```

```
EndPaint(hWnd, &ps);
break;
```

We see that roughly the same number of allocations is split between "Gla:" and "Gh0@" pool tags:

Pool Used:				
	NonPaged		Paged	
Tag	Allocs	Used	Allocs	Used
NV_x	0	0	5	14024704 UNKNOWN pooltag 'NV_x', please update pooltag.txt
BIG	0	0	262	3874816 Large session pool allocations (ntos\ex\pool.c) , Binary: nt!mm
<b>Gla:</b>	<b>0</b>	<b>0</b>	<b>5203</b>	<b>3413168</b> Gdi handle manager specific object types allocated from lookaside memory: defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
NV	0	0	203	1347648 nVidia video driver
<b>Gh0@</b>	<b>0</b>	<b>0</b>	<b>5077</b>	<b>1137248</b> Gdi Handle manager specific object types: defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
Ttfd	0	0	233	1053152 TrueType Font driver

The third program leaks fonts, pens, and brushes:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)
    {
        case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps);
            while (true)
            {
                HFONT hf = CreateFont(10, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, L"Arial");
                HPEN hp = CreatePen(0, 10, RGB(10, 20, 30));
                HBRUSH hb = CreateSolidBrush(RGB(10, 20, 30));
                SelectObject(ps.hdc, hf);
                SelectObject(ps.hdc, hp);
                SelectObject(ps.hdc, hb);
            }
            EndPaint(hWnd, &ps);
            break;
    }
}
```

Now we see that the same number of allocations is almost equally split between "Gla:", "Gh0@", and "Gla@" pool tags:

```
Pool Used:
      NonPaged          Paged
    Tag   Allocs   Used   Allocs   Used
NV_x       0       0      5 14024704 UNKNOWN pooltag 'NV_x', please update pooltag.txt
BIG        0       0     262 3874816 Large session pool allocations (ntos\ex\pool.c) , Binary:
nt!mm
Gla:       0       0     3539 2321584 Gdi handle manager specific object types allocated from
Lookaside memory: defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
NV         0       0     203 1347648 nVidia video driver
Ttfdf      0       0     233 1053152 TrueType Font driver
Gh05       0       0     392 1052768 Gdi Handle manager specific object types: defined in
w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
Gla1       0       0     353 796368 Gdi handle manager specific object types allocated from
lookaside memory: defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
Gh0@       0       0     3414 764736 Gdi Handle manager specific object types: defined in
w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
Gla@       0       0     3665 762320 Gdi handle manager specific object types allocated from
Lookaside memory: defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
Gcac       0       0      25 640880 Gdi glyph cache
```

When the certain amount of handles is reached, all subsequent GDI *Create* calls fail, and other applications start showing various visual defects. Print screen operation also fails with insufficient memory message.

## User Space

Windows imposes a restriction on the number of GDI handles per process, and by default, it is approx. 10,000. If this number is reached, we can have abnormal software behavior symptoms such as hangs, glitches in visual appearance, and out-of-memory exceptions resulted in error messages and crashes. We already documented this analysis pattern for kernel and complete memory dumps that we called **Handle Limit (GDI, Kernel Space)**, page 417). However, one of Software Diagnostics Services' training customers reported an out-of-memory exception with trace analysis diagnostics pointing to 10,000 leaked GDI objects. The process memory dump was saved, and the customer asked whether it was possible to analyze it, or similar memory dumps to find out from the dump itself whether we have GDI leak and what GDI objects were involved.

We recreated one of the applications from the kernel pattern variant (the one that leaks fonts) with one modification that it just stops after 10,000 font creation attempts. After launch, we tried to open *About* dialog box, but the whole application became unresponsive, and no dialog box was visible. We save the process memory dump and found out that its **Main Thread** (page 614) was inside **Dialog Box** (page 232) processing:

```
0:000> kc
# Call Site
00 user32!NtUserWaitMessage
01 user32!DialogBox2
02 user32!InternalDialogBox
03 user32!DialogBoxIndirectParamAorW
04 user32!DialogBoxParamW
05 GUIHandleLeak!WndProc
06 user32!UserCallWinProcCheckWow
07 user32!DispatchMessageWorker
08 GUIHandleLeak!wWinMain
09 GUIHandleLeak!invoke_main
0a GUIHandleLeak!__scrt_common_main_seh
0b kernel32!BaseThreadInitThunk
0c ntdll!RtlUserThreadStart
```

In order to look at GDI handle table we studied the relevant chapters in Feng Yuan's book "Windows Graphics Programming" and the post<sup>86</sup> which has all necessary structure definitions.

We get the current process GDI table address from the disassembly:

```
0:000> .asm no_code_bytes
Assembly options: no_code_bytes

0:000> uf gdi32!GdiQueryTable
gdi32!GdiQueryTable:
00007ffc`7f172610 sub      rsp,38h
00007ffc`7f172614 or     qword ptr [rsp+20h],0xFFFFFFFFFFFFFFFh
```

---

<sup>86</sup> <http://stackoverflow.com/questions/13905661/how-to-get-list-of-gdi-handles>

```

00007ffc`7f17261a lea      rdx,[rsp+20h]
00007ffc`7f17261f mov      ecx,0Eh
00007ffc`7f172624 mov      byte ptr [rsp+28h],0
00007ffc`7f172629 call    qword ptr [gdi32!_imp_NtVdmControl (00007ffc`7f1ba5a8)]
00007ffc`7f17262f test    eax,eax
00007ffc`7f172631 js      gdi32!GdiQueryTable+0x33 (00007ffc`7f172643) Branch

gdi32!GdiQueryTable+0x23:
00007ffc`7f172633 cmp      byte ptr [rsp+28h],0
00007ffc`7f172638 je      gdi32!GdiQueryTable+0x33 (00007ffc`7f172643) Branch

gdi32!GdiQueryTable+0x2a:
00007ffc`7f17263a mov      rax,qword ptr [gdi32!pGdiSharedHandleTable (00007ffc`7f2541b8)]
00007ffc`7f172641 jmp      gdi32!GdiQueryTable+0x35 (00007ffc`7f172645) Branch

gdi32!GdiQueryTable+0x33:
00007ffc`7f172643 xor      eax,eax

gdi32!GdiQueryTable+0x35:
00007ffc`7f172645 add      rsp,38h
00007ffc`7f172649 ret

0:000> dp 00007ffc`7f2541b8 L1
00007ffc`7f2541b8 000000db`56a50000

```

We dump the first 0x1000 qword values:

```

0:000> dq 000000db`56a50000 L1000
000000db`56a50000 00000000`00000000 40000000`00000000
000000db`56a50010 00000000`00000000 00000000`00000000
000000db`56a50020 40000000`00000000 00000000`00000000
000000db`56a50030 00000000`00000000 00000000`00000000
000000db`56a50040 00000000`00000000 00000000`00000000
000000db`56a50050 00000000`00000000 00000000`00000000
000000db`56a50060 00000000`00000000 00000000`00000000
000000db`56a50070 00000000`00000000 00000000`00000000
000000db`56a50080 00000000`00000000 00000000`00000000
000000db`56a50090 00000000`00000000 00000000`00000000
000000db`56a500a0 00000000`00000000 00000000`00000000
000000db`56a500b0 00000000`00000000 00000000`00000000
000000db`56a500c0 00000000`00000000 00000000`00000000
000000db`56a500d0 00000000`00000000 00000000`00000000
000000db`56a500e0 00000000`00000000 00000000`00000000
000000db`56a500f0 ffffff901`40000e60 40040104`00000000
000000db`56a50100 00000000`00000000 ffffff901`40000d60
000000db`56a50110 40080188`00000000 00000000`00000000
000000db`56a50120 ffffff901`400008b0 40080108`00000000
000000db`56a50130 00000000`00000000 ffffff901`400007c0
000000db`56a50140 40080108`00000000 00000000`00000000
000000db`56a50150 ffffff901`400006d0 40080108`00000000
[...]
000000db`56a57ce0 ffffff901`429d24f0 400aee0a`00002c30
000000db`56a57cf0 000000db`564f3b20 ffffff901`42910570
000000db`56a57d00 400a360a`00002c30 000000db`564e57b0

```

```

000000db`56a57d10 fffff901`40700420 40105310`000002b4
000000db`56a57d20 00000089`39410fc0 fffff901`407ec4a0
000000db`56a57d30 400a010a`000002b4 00000089`3900ae70
000000db`56a57d40 fffff901`407036d0 400a010a`000002b4
000000db`56a57d50 00000089`3900ae60 fffff901`440b56e0
000000db`56a57d60 400a030a`00002c30 000000db`564f0e90
000000db`56a57d70 fffff901`43e7fd50 40040104`00000000
000000db`56a57d80 00000000`00000000 fffff901`42c0f010
000000db`56a57d90 400a4b0a`00003190 0000003a`1a30b670
000000db`56a57da0 fffff901`440deaf0 400a6d0a`00002c30
000000db`56a57db0 000000db`564f3b00 fffff901`407f2010
000000db`56a57dc0 40100510`00001704 000000d8`d6230f60
000000db`56a57dd0 fffff901`40714180 40100210`00001704
000000db`56a57de0 000000d8`d6230f48 fffff901`4009d840
000000db`56a57df0 40100210`00001704 000000d8`d6230f78
000000db`56a57e00 fffff901`43e50950 40100230`00001704
000000db`56a57e10 00000000`00000000 fffff901`43e30010
000000db`56a57e20 40100230`00001704 00000000`00000000
000000db`56a57e30 fffff901`44f1d010 4005a105`0000168c
000000db`56a57e40 00000000`00000000 fffff901`440ded80
000000db`56a57e50 400a2e0a`00002c30 000000db`564f3b10
000000db`56a57e60 fffff901`4070b3b0 40050405`00000000
000000db`56a57e70 00000000`00000000 fffff901`42a0a010
000000db`56a57e80 400a870a`00002c30 000000db`564e7160
000000db`56a57e90 fffff901`407a7450 4008cd08`00000000
000000db`56a57ea0 00000000`00000000 fffff901`400c49c0
000000db`56a57eb0 40046904`000002b4 00000089`39410f90
000000db`56a57ec0 fffff901`41fb8010 4005c705`00000000
000000db`56a57ed0 00000000`00000000 fffff901`423dc790
000000db`56a57ee0 40086708`00000000 00000000`00000000
000000db`56a57ef0 fffff901`40699620 40010301`0000168c
000000db`56a57f00 00000000`01100000 fffff901`43e54510
000000db`56a57f10 40050305`0000168c 00000000`00000000
000000db`56a57f20 fffff901`407164c0 40100610`00001448
000000db`56a57f30 000000fe`4d8d0cf0 fffff901`407eee50
000000db`56a57f40 40100410`00001448 000000fe`4d8d0cd8
000000db`56a57f50 fffff901`43e2abb0 40080508`00000000
000000db`56a57f60 00000000`00000000 fffff901`40715010
000000db`56a57f70 40050305`0000168c 00000000`00000000
000000db`56a57f80 fffff901`42872b80 40084e08`00000000
000000db`56a57f90 00000000`00000000 fffff901`407175a0
000000db`56a57fa0 410f080f`00000000 00000000`00000000
000000db`56a57fb0 fffff901`407f4000 40050605`00000000
000000db`56a57fc0 00000000`00000000 fffff901`406d6bb0
000000db`56a57fd0 40080508`00000000 00000000`00000000
000000db`56a57fe0 fffff901`43e3a4c0 40120812`00000000
000000db`56a57ff0 00000000`00000000 fffff901`44fe1010

```

We see that typical cell value has 3 qwords (8-byte or 4-word sized values for both x64 and **Virtualized Process**, page 1068). The non-zeroed data starts from **000000db`56a500f0** address. Clearly, some entries have *wProcessId* equal to the PID from our dump:

```

0:000> ~
. 0  Id: 2c30.292c Suspend: 0 Teb: 00007ff7`1bf3e000 Unfrozen

```

Let's look at one of such entries (the first and the last qword values are pointers):

```
0:000> dq 000000db`56a57da0 L3
000000db`56a57da0  ffffff901`440deaf0 400a6d0a`00002c30
000000db`56a57db0  000000db`564f3b00

0:000> dw 000000db`56a57da0 L3*4
000000db`56a57da0  ea0f 440d f901 ffff 2c30 0000 6d0a 400a
000000db`56a57db0  3b00 564f 00db 0000
```

Applying 7f to wType word 400a gives us 0xa or 10 which is a font handle:

```
0:000> ? 400a & 7f
Evaluate expression: 10 = 00000000`0000000a
```

Since there are entries from other processes from the same session in this table assessing the handle leak visually is difficult so we wrote a WinDbg script that goes from the first non-zero *pKernelAddress* (\$t0) to the first zero entry and for the given *wProcessId* (\$tpid) counts the number of entries (\$t1) and the number of entries (\$t3) for the given *wType* (\$t2). The script also counts the total entries till the first zero one (\$t4):

```
.while (qwo(@$t0)) { .if (wo(@$t0+8) == @$tpid) {r $t1 = @$t1+1; .if (((qwo(@$t0+8) >> 0n48) & 7f) ==
@$t2) {r $t3 = @$t3+1} }; r $t0 = @$t0+3*8; r $t4 = @$t4+1}
```

To execute it we prepare the variables:

```
0:000> r $t0=000000db`56a500f0

0:000> r $t1=0

0:000> r $t2=a

0:000> r $t3=0

0:000> r $t4=0

0:000> .while (qwo(@$t0)) { .if (wo(@$t0+8) == @$tpid) {r $t1 = @$t1+1; .if (((qwo(@$t0+8) >> 0n48) & 7f) ==
@$t2) {r $t3 = @$t3+1} }; r $t0 = @$t0+3*8; r $t4 = @$t4+1}
```

After execution we get the modified variables that show us that the total consecutive non-zero handle table entries is 21464, the total number of entries for the current process is 9990, and the total number of fonts is 9982:

```
0:000> ? $t0
Evaluate expression: 942052007216 = 000000db`56acdd30

0:000> ? $t1
Evaluate expression: 9990 = 00000000`00002706
```

```
0:000> ? $t2
Evaluate expression: 10 = 00000000`0000000a

0:000> ? $t3
Evaluate expression: 9982 = 00000000`000026fe

0:000> ? $t4
Evaluate expression: 21464 = 00000000`000053d8
```

If we repeat the same script for device contexts (*wType* is 1) we get only 2 entries for our PID:

```
0:000> r $t0=000000db`56a500f0

0:000> r $t1=0

0:000> r $t2=1

0:000> r $t3=0

0:000> r $t4=0

0:000> .while (qwo(@$t0)) { .if (wo(@$t0+8) == @$tpid) {r $t1 = @$t1+1; .if (((qwo(@$t0+8) >> 0n48) &
7f) == @$t2) {r $t3 = @$t3+1} }; r $t0 = @$t0+3*8; r $t4 = @$t4+1}

0:000> ? $t3
Evaluate expression: 2 = 00000000`00000002
```

Of course, this script may be further improved, for example, to process all possible *wType* values and print their statistics. It can also be made as a textual WinDbg script procedure with arguments.

## Handled Exception

### .NET CLR

Similar to unmanaged user space **Handled Exceptions** (page 434) residue we can see a similar one on raw stacks of .NET CLR threads. Here are some typical fragments (x86, CLR 4 has similar residue):

```
[...]
09c8e1e0 79ef2dee mscorewks!ExInfo::Init+0x41
09c8e1e4 00004000
09c8e1e8 79f088cc mscorewks!`string'
09c8e1ec 79f088c2 mscorewks!ExInfo::UnwindExInfo+0x14d
09c8e1f0 08f68728
09c8e1f4 95f5b898
09c8e1f8 09c8e1a4
09c8e1fc 09c8e92c
09c8e200 7a34d0d8 mscorewks!GetManagedNameForTypeInfo+0x22b02
09c8e204 79f091ee mscorewks!COMPlusCheckForAbort+0x15
09c8e208 00000000
09c8e20c 0aada664
09c8e210 0aaabff4
09c8e214 00000000
09c8e218 09c8eec
09c8e21c 074c1f23
09c8e220 09c8ef0c
09c8e224 79f091cb mscorewks!JIT_EndCatch+0x16
09c8e228 09c8ef0c
09c8e22c 09c8eec
09c8e230 074c1f23
09c8e234 09c8e25c
09c8e238 0009c108
09c8e23c 09c8e460
09c8e240 09c8e5c4
09c8e244 00071d88
09c8e248 08f68728
09c8e24c 79e734c4 mscorewks!ClrFlsSetValue+0x57
09c8e250 95f5b8e4
09c8e254 0aada634
09c8e258 08f68728
09c8e25c 0aada90c
09c8e260 0aaabff4
09c8e264 00000002
09c8e268 09c8e304
09c8e26c 0aada664
09c8e270 00000000
09c8e274 09c8ef0c
09c8e278 09c8e234
09c8e27c 074c1f13
09c8e280 00000000
09c8e284 08f688a0
09c8e288 09c8e234
09c8e28c 79f00c0b mscorewks!Thread::ReturnToContext+0x4e2
09c8e290 0aada90c
09c8e294 09c8eef4
```

```
09c8e298 09c8e2bc
09c8e29c 79f08eb8 mscorewks!EEJitManager::ResumeAtJitEH+0x28
09c8e2a0 09c8e460
09c8e2a4 074c1ed8
09c8e2a8 074b41a8
09c8e2ac 00000000
09c8e2b0 08f68728
09c8e2b4 00000000
09c8e2b8 09c8e410
09c8e2bc 09c8e3c8
09c8e2c0 79f08df5 mscorewks!COMPlusUnwindCallback+0x7c3
09c8e2c4 09c8e460
09c8e2c8 074b41a8
09c8e2cc 00000000
09c8e2d0 08f68728
09c8e2d4 00000000
09c8e2d8 0009c108
09c8e2dc 09c8e410
09c8e2e0 09c8e5c4
09c8e2e4 074b41a8
09c8e2e8 09c8e3a4
09c8e2ec 79e734c4 mscorewks!ClrFlsSetValue+0x57
09c8e2f0 95f5b984
09c8e2f4 0009c128
09c8e2f8 09c8e3a4
09c8e2fc 00000000
09c8e300 00000000
09c8e304 00000002
[...]
09c8e4e4 00000000
09c8e4e8 79f09160 mscorewks!_CT??_R0H+0x34b4
09c8e4ec ffffffff
09c8e4f0 73792e2f msvcr80!_getptd+0x6
09c8e4f4 ffffffff
09c8e4f8 737b7a78 msvcr80!__FrameUnwindToState+0xd9
09c8e4fc 737b7a5e msvcr80!__FrameUnwindToState+0xbf
09c8e500 95f5bc05
09c8e504 e06d7363
09c8e508 1fffffff
09c8e50c 19930522
09c8e510 ffffffff
09c8e514 ffffffff
09c8e518 09c8e500
09c8e51c 09c8e554
09c8e520 09c8e5a8
09c8e524 73798cd9 msvcr80!_except_handler4
09c8e528 efb0d3d
09c8e52c ffffffff
09c8e530 737b7a5e msvcr80!__FrameUnwindToState+0xbf
09c8e534 737b89cb msvcr80!__InternalCxxFrameHandler+0x6d
09c8e538 09c8eab0
09c8e53c 09c8e6a0
09c8e540 79f09160 mscorewks!_CT??_R0H+0x34b4
09c8e544 ffffffff
09c8e548 00000000
09c8e54c 00000000
```

```
09c8e550 00000000
09c8e554 09c8e590
09c8e558 737b8af1 msvcr80!__CxxFrameHandler3+0x26
09c8e55c 09c8e600
09c8e560 09c8eab0
09c8e564 01010101
09c8e568 09000000
09c8e56c 09c8f160
09c8e570 07540c00
09c8e574 00071d88
09c8e578 08e65d48
09c8e57c 09c8e5ec
09c8e580 074c1ec8
09c8e584 00000024
09c8e588 00000001
09c8e58c 0009c108
09c8e590 08f68728
09c8e594 00000000
09c8e598 00000000
09c8e59c 09c8eb38
09c8e5a0 00000000
09c8e5a4 09c8e6a0
09c8e5a8 09c8f15c
09c8e5ac 09c8f15c
09c8e5b0 09c8eb38
09c8e5b4 95f5bf28
09c8e5b8 09c8e8f4
09c8e5bc 79e84bf2 mscorewks!Thread::StackWalkFrames+0xb8
09c8e5c0 08f68728
09c8e5c4 09c8ea40
09c8e5c8 79e84bf2 mscorewks!Thread::StackWalkFrames+0xb8
09c8e5cc 09c8e5ec
09c8e5d0 79f07d64 mscorewks!COMPlusUnwindCallback
09c8e5d4 09c8ea40
09c8e5d8 00000005
09c8e5dc 00000000
09c8e5e0 08f68728
09c8e5e4 08f688a0
09c8e5e8 08f68728
09c8e5ec 09c8ec20
09c8e5f0 00000000
09c8e5f4 09c8ecbc
09c8e5f8 09c8ecc0
09c8e5fc 09c8ecc4
09c8e600 09c8ecc8
09c8e604 09c8eccc
09c8e608 09c8ecd0
09c8e60c 09c8ecd4
09c8e610 09c8eeeec
09c8e614 09c8ecd8
09c8e618 09c8ecd8
09c8e61c 00000024
09c8e620 00000000
09c8e624 0009c108
09c8e628 08f68728
09c8e62c 00000000
```

```
09c8e630 00000000
09c8e634 79e71ba4 mscorewks!Thread::CatchAtSafePoint
09c8e638 00000000
09c8e63c 79e71ba4 mscorewks!Thread::CatchAtSafePoint
09c8e640 09c8f15c
09c8e644 09c8f15c
09c8e648 00000000
09c8e64c 95f5bcc0
09c8e650 09c8e988
09c8e654 79e84bf2 mscorewks!Thread::StackWalkFrames+0xb8
09c8e658 09c8ea40
09c8e65c 79e84bf2 mscorewks!Thread::StackWalkFrames+0xb8
09c8e660 09c8e680
09c8e664 79f07957 mscorewks!COMPlusThrowCallback
09c8e668 09c8ea40
09c8e66c 00000000
09c8e670 00000000
09c8e674 0aada90c
09c8e678 09c8ea40
09c8e67c 79e84bff mscorewks!Thread::StackWalkFrames+0xc5
09c8e680 09c8ec20
09c8e684 00000000
09c8e688 09c8ecbc
09c8e68c 09c8ecc0
09c8e690 09c8ecc4
09c8e694 09c8ecc8
[...]
09c8e8f0 95f5b264
09c8e8f4 09c8e914
09c8e8f8 79f07d5e mscorewks!UnwindFrames+0x62
09c8e8fc 79f07d64 mscorewks!COMPlusUnwindCallback
09c8e900 09c8ea40
09c8e904 00000005
09c8e908 00000000
09c8e90c 09c8ef6c
09c8e910 08f68728
09c8e914 09c8e9a4
09c8e918 79f089cc mscorewks!COMPlusAfterUnwind+0x97
09c8e91c 08f68728
09c8e920 09c8ea40
09c8e924 00000001
09c8e928 00000000
09c8e92c 09c8ef6c
09c8e930 79f0a3d9 mscorewks!COMPlusNestedExceptionHandler
09c8e934 09c8f160
09c8e938 00000000
09c8e93c 00000000
09c8e940 cccccccc
[...]
```

Sometimes we can see ‘*ExecuteHandler*’ calls if they were not overwritten:

```
[...]
09d2e6e0 00000000
09d2e6e4 00000720
09d2e6e8 77c41039 ntdll!ExecuteHandler2+0x26
09d2e6ec 09d2e7c8
09d2e6f0 09d2eb60
09d2e6f4 09d2e7e4
09d2e6f8 09d2e7a4
09d2e6fc 09d2eb60
09d2e700 77c4104d ntdll!ExecuteHandler2+0x3a
09d2e704 09d2eb60
09d2e708 09d2e7b0
09d2e70c 77c4100b ntdll!ExecuteHandler+0x24
09d2e710 09d2e7c8
09d2e714 00000001
09d2e718 09d2e6b0
09d2e71c 09d2e7a4
09d2e720 09d2e784
09d2e724 76545ac9 kernel32!_except_handler4
[...]
```

If there are such traces they can be visible as **Caller-n-Callee** (page 111) pattern:

```
0:011> !DumpStack
OS Thread Id: 0x3cc (11)
Current frame: ntdll!KiFastSystemCallRet
ChildEBP RetAddr Caller, Callee
09d2e690 77c40690 ntdll!ZwWaitForMultipleObjects+0xc
09d2e694 76577e09 kernel32!WaitForMultipleObjectsEx+0x11d, calling ntdll!NtWaitForMultipleObjects
09d2e6d8 76578101 kernel32!WaitForMultipleObjectsEx+0x33, calling ntdll! 09d2e6e4 77c41039 ntdll!ExecuteHandler2+0x26
09d2e708 77c4100b ntdll!ExecuteHandler+0x24, calling ntdll!ExecuteHandler2
09d2e730 6baa516a clr!WaitForMultipleObjectsEx_SO_TOLERANT+0x56, calling RtlActivateActivationContextUnsafeFast
kernel32!WaitForMultipleObjectsEx
09d2e794 6baa4f98 clr!Thread::DoAppropriateAptStateWait+0x4d, calling clr!WaitForMultipleObjectsEx_SO_TOLERANT
09d2e7b4 6baa4dd8 clr!Thread::DoAppropriateWaitWorker+0x17d, calling clr!Thread::DoAppropriateAptStateWait
09d2e848 6baa4e99 clr!Thread::DoAppropriateWait+0x60, calling clr!Thread::DoAppropriateWaitWorker
09d2e8b4 6baa4f17 clr!CLREvent::WaitEx+0x106, calling clr!Thread::DoAppropriateWait
09d2e8e0 6baa484b clr!CLRGetTickCount64+0x6b, calling clr!_allmul
09d2e908 6ba4d409 clr!CLREvent::Wait+0x19, calling clr!CLREvent::WaitEx
[...]
```

## Kernel Space

This is a variant of **Handled Exception** pattern in kernel space similar to the user (page 434) and managed spaces (page 428). The crash dump was the same as in **Hidden Exception** in kernel space pattern (page 455):

```
fffff880`0a83d910 00000000`00000000
fffff880`0a83d918 fffff6fc`40054fd8
fffff880`0a83d920 fffff880`0a83dca0
fffff880`0a83d928 fffff800`016bcc1c nt!_C_specific_handler+0xcc
fffff880`0a83d930 00000000`00000000
fffff880`0a83d938 00000000`00000000
fffff880`0a83d940 00000000`00000000
fffff880`0a83d948 00000000`00000000
fffff880`0a83d950 fffff800`0189ee38 nt!BBTBuffer <PERF> (nt+0x280e38)
fffff880`0a83d958 fffff880`0a83e940
fffff880`0a83d960 fffff800`016ad767 nt!IopCompleteRequest+0x147
fffff880`0a83d968 fffff880`0a83de40
fffff880`0a83d970 fffff800`01665e40 nt!_GSHandlerCheck_SEH
fffff880`0a83d978 fffff800`017e5338 nt!_imp_NtOpenSymbolicLinkObject+0xfe30
fffff880`0a83d980 fffff880`0a83e310
fffff880`0a83d988 00000000`00000000
fffff880`0a83d990 00000000`00000000
fffff880`0a83d998 fffff800`016b42dd nt!RtlpExecuteHandlerForException+0xd
fffff880`0a83d9a0 fffff800`017d7d0c nt!_imp_NtOpenSymbolicLinkObject+0x2804
fffff880`0a83d9a8 fffff880`0a83eab0
fffff880`0a83d9b0 00000000`00000000

0: kd> ub fffff800`016b42dd
nt!RtlpExceptionHandler+0x24:
fffff800`016b42c4 cc          int 3
fffff800`016b42c5 cc          int 3
fffff800`016b42c6 cc          int 3
fffff800`016b42c7 cc          int 3
fffff800`016b42c8 0f1f840000000000 nop dword ptr [rax+rax]
nt!RtlpExecuteHandlerForException:
fffff800`016b42d0 4883ec28    sub rsp,28h
fffff800`016b42d4 4c894c2420  mov qword ptr [rsp+20h],r9
fffff800`016b42d9 41ff5130    call qword ptr [r9+30h]
```

## User Space

If we do not see exception codes when we inspect raw stack data we, nevertheless, in some cases may see execution residue left after calling exception handlers. For example, we can see that when we launch TestWER<sup>87</sup> tool and select '*Handled Exception*' checkbox:



If we then click on a button and then save a process memory dump using Task Manager we find the following traces on a raw stack:

```
0:000> !teb
TEB at 7efdd000
ExceptionList:      0018fe20
StackBase:          00190000
StackLimit:         0018d000
SubSystemTib:       00000000
FiberData:          00001e00
ArbitraryUserPointer: 00000000
Self:               7efdd000
EnvironmentPointer: 00000000
ClientId:           00000b38 . 00000f98
RpcHandle:          00000000
Tls Storage:        7efdd02c
PEB Address:        7efde000
LastErrorValue:     0
LastStatusValue:    c0000034
Count Owned Locks:  0
HardErrorMode:      0
```

<sup>87</sup> TestWER Tool to Test Windows Error Reporting, Memory Dump Analysis Anthology, Volume 6, page 280

```

0:000> dps 0018d000 00190000
[...]
0018f414 00000000
0018f418 0018f840
0018f41c 0018f4cc
0018f420 77726a9b ntdll!ExecuteHandler+0x24
0018f424 0018f4e4
0018f428 0018f840
0018f42c 0018f534
0018f430 0018f4b8
0018f434 00412600 TestWER!_except_handler4
0018f438 00000001
0018f43c 00000000
[...]

0:000> ub 77726a9b
ntdll!ExecuteHandler+0x7:
77726a7e 33f6 xor esi,esi
77726a80 33ff xor edi,edi
77726a82 ff742420 push dword ptr [esp+20h]
77726a86 ff742420 push dword ptr [esp+20h]
77726a8a ff742420 push dword ptr [esp+20h]
77726a8e ff742420 push dword ptr [esp+20h]
77726a92 ff742420 push dword ptr [esp+20h]
77726a96 e808000000 call ntdll!ExecuteHandler2 (77726aa3)

```

If we compare the output above with the raw stack fragment from the second chance exception memory dump (after we relaunch TestWER, don't select '*Handled Exception*' checkbox and click on the big lightning button) we would see the similar call fragment:

```

[...]
0018f3f4 00dd0aa7
0018f3f8 0018f41c
0018f3fc 77726ac9 ntdll!ExecuteHandler2+0x26
0018f400 fffffffe
0018f404 0018ffc4
0018f408 0018f534
0018f40c 0018f4b8
0018f410 0018f840
0018f414 77726add ntdll!ExecuteHandler2+0x3a
0018f418 0018ffc4
0018f41c 0018f4cc
0018f420 77726a9b ntdll!ExecuteHandler+0x24
0018f424 0018f4e4
0018f428 0018ffc4
0018f42c 0018f534
0018f430 0018f4b8
0018f434 77750ae5 ntdll!_except_handler4
0018f438 00000000
0018f43c 0018f4e4
0018f440 0018ffc4
0018f444 77726a3d ntdll!RtlDispatchException+0x127
0018f448 0018f4e4
0018f44c 0018ffc4
0018f450 0018f534

```

```
0018f454 0018f4b8
0018f458 77750ae5 ntdll!_except_handler4
0018f45c 00000111
0018f460 0018f4e4
[...]
```

Sometimes, we can also see “*Unwind*”, “*StackWalk*”, “*WalkFrames*”, “*EH*”, “*Catch*” functions too. Sometimes we don’t see anything because such residue was overwritten by subsequent function calls after **Handled Exceptions** happened sometime in the past.

---

## Comments

If we get several same first chance exception dumps for the same process, it means that it was handled. Otherwise, we would not see the subsequent dumps.

## Hardware Activity

Sometimes, when a high number of interrupts is reported, but there are no signs of an interrupt storm<sup>88</sup> or pending DPCs in a memory dump file it is useful to search for this pattern in running and / or suspected threads. This can be done by examining execution residue left on a thread raw stack. Although the found driver activity may not be related to reported problems, it can be a useful start for a driver elimination procedure for a general recommendation to check suspected drivers for any updates. Here is an example of a thread raw stack with a network card doing “Scatter-Gather” DMA:

```
1: kd> !thread
THREAD f7732090 Cid 0000.0000 Peb: 00000000 Win32Thread: 00000000 RUNNING on processor 1
Not impersonating
Owning Process 8089db40 Image: Idle
Attached Process N/A Image: N/A
Wait Start TickCount 0 Ticks: 24437545 (4:10:03:56.640)
Context Switch Count 75624870
UserTime 00:00:00.000
KernelTime 4 Days 08:56:05.125
Stack Init f78b3000 Current f78b2d4c Base f78b3000 Limit f78b0000 Call 0
Priority 0 BasePriority 0 PriorityDecrement 0
ChildEBP RetAddr Args to Child
f3b30c5c 00000000 00000000 00000000 00000000 LiveKdD+0x1c07

1: kd> dds f78b0000 f78b3000
f78b0000 00000000
f78b0004 00000000
f78b0008 00000000
f78b000c 00000000
f78b0010 00000000
[...]
f78b2870 8b3de0d0
f78b2874 80887b75 nt!KiFlushTargetSingleTb+0xd
f78b2878 8b49032c
f78b287c 00000000
f78b2880 2d003202
f78b2884 00000000
f78b2888 00000000
f78b288c 2d003202
f78b2890 8b490302
f78b2894 f78b28a4
f78b2898 80a61456 hal!KfLowerIrql+0x62
f78b289c 2d00320a
f78b28a0 00000000
f78b28a4 8b3de0d0
f78b28a8 8b3e3730
f78b28ac 00341eb0
f78b28b0 f78b2918
```

<sup>88</sup> [http://msdn.microsoft.com/en-us/library/ff540586\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ff540586(VS.85).aspx)

```
f78b28b4 f63fbf78 NetworkAdapterA!SendWithScatterGather+0x318
f78b28b8 8b3de0d0
f78b28bc 8b341eb0
f78b28c0 f78b28d4
f78b28c4 00000000
f78b28c8 80a5f3c0 hal!KfAcquireSpinLock
f78b28cc 00000000
f78b28d0 8b3de0d0
f78b28d4 00000000
f78b28d8 8b3de0d0
f78b28dc 8b3eb730
f78b28e0 005a7340
f78b28e4 f78b294c
f78b28e8 f63fbf78 NetworkAdapterA!SendWithScatterGather+0x318
f78b28ec 8b3de0d0
f78b28f0 8a5a7340
f78b28f4 f78b2908
f78b28f8 00000000
f78b28fc 8b3de0d0
f78b2900 8b0f5158
f78b2904 001e2340
f78b2908 f78b2970
f78b290c f63fbf78 NetworkAdapterA!SendWithScatterGather+0x318
f78b2910 8b3de0d0
f78b2914 8b1e2340
f78b2918 f78b292c
f78b291c 00000000
f78b2920 80a5f3c0 hal!KfAcquireSpinLock
f78b2924 00000000
f78b2928 8b3de0d0
f78b292c 00000000
f78b2930 8b3eb700
f78b2934 00000000
f78b2938 00000000
f78b293c 00000000
f78b2940 00000000
f78b2944 00000000
f78b2948 00000000
f78b294c 0a446aa2
f78b2950 f78b29b8
f78b2954 8b0f5158
f78b2958 8b01ce10
f78b295c 00000001
f78b2960 8b3de0d0
f78b2964 80a5f302 hal!HalpPerfInterrupt+0x32
f78b2968 00000001
f78b296c 8b3de0d0
f78b2970 80a5f302 hal!HalpPerfInterrupt+0x32
f78b2974 8b3de302
f78b2978 f78b2988
f78b297c 80a61456 hal!KfLowerIrql+0x62
f78b2980 80a5f3c0 hal!KfAcquireSpinLock
f78b2984 8b3de302
f78b2988 f78b29a4
f78b298c 80a5f44b hal!KfReleaseSpinLock+0xb
f78b2990 f63fbfff NetworkAdapterA!SendPackets+0x1b3
```

```
f78b2994 8a446a90
f78b2998 8b0e8ab0
f78b299c 00000000
f78b29a0 008b29d0
f78b29a4 f78b29bc
f78b29a8 f7163790 NDIS!ndisMProcessSGList+0x90
f78b29ac 8b3de388
f78b29b0 f78b29d0
f78b29b4 00000001
f78b29b8 00000000
f78b29bc f78b29e8
f78b29c0 80a60147 hal!HalBuildScatterGatherList+0x1c7
f78b29c4 8b0e89b0
f78b29c8 00000000
f78b29cc 8a44cde8
f78b29d0 8b1e2340
f78b29d4 8a446aa2
f78b29d8 8b026ca0
f78b29dc 8b1e2340
f78b29e0 8b0e8ab0
f78b29e4 8b0e8ab0
f78b29e8 f78b2a44
f78b29ec f716369f NDIS!ndisMAlocateSGList+0xda
f78b29f0 8a44cde8
f78b29f4 8b0e89b0
f78b29f8 8a446a70
f78b29fc 00000000
f78b2a00 00000036
f78b2a04 f7163730 NDIS!ndisMProcessSGList
f78b2a08 8b1e2340
f78b2a0c 00000000
f78b2a10 8a44cde8
f78b2a14 00000218
f78b2a18 8b1e2308
[...]
f78b2a40 029a9e02
f78b2a44 f78b2a60
f78b2a48 f71402ff NDIS!ndisMSendX+0x1dd
f78b2a4c 8b490310
f78b2a50 8b1e2340
f78b2a54 8a446a70
f78b2a58 8a9a9e02
f78b2a5c 8a9a9e02
f78b2a60 f78b2a88
f78b2a64 f546c923 tcpip!ARPSendData+0x1a9
f78b2a68 8b3e76c8
f78b2a6c 8b1e2340
f78b2a70 8a9a9ea8
f78b2a74 8b490310
f78b2a78 80888b00 nt!RtlBackoff+0x68
f78b2a7c 8a446a70
f78b2a80 8a446aa2
f78b2a84 8a446a70
f78b2a88 f78b2ab4
f78b2a8c f546ba5d tcpip!ARPTransmit+0x112
f78b2a90 8b490310
```

```
f78b2a94 8b1e2340
f78b2a98 8a9a9ea8
f78b2a9c 00000103
f78b2aa0 8a4446a70
f78b2aa4 00000000
f78b2aa8 8b342398
f78b2aac 8a47e1f8
f78b2ab0 8b1e2340
f78b2ab4 f78b2bf0
f78b2ab8 f546c4fc tcpip!_IPTtransmit+0x866
f78b2abc 8a9a9ebc
f78b2ac0 f78b2b02
f78b2ac4 00000001
[...]
```

We also do a sanity check for **Coincidental Symbolic Information** (page 134):

```
1: kd> ub f63fbf78
NetworkAdapterA!SendWithScatterGather+0x304:
f63fbf64 push    eax
f63fbf65 push    edi
f63fbf66 push    esi
f63fbf67 mov     dword ptr [ebp-44h],ecx
f63fbf6a mov     dword ptr [ebp-3Ch],ecx
f63fbf6d mov     dword ptr [ebp-34h],ecx
f63fbf70 mov     dword ptr [ebp-2Ch],ecx
f63fbf73 call    NetworkAdapterA!PacketRetrieveNicActions (f63facd2)

1: kd> ub f63fbfff
NetworkAdapterA!SendPackets+0x190:
f63fbb9c cmp     dword ptr [esi+0Ch],2
f63fbb9a0 jl      NetworkAdapterA!SendPackets+0x19e (f63fbbaa)
f63fbb9a2 mov     dword ptr [ecx+3818h],eax
f63fbb9a8 jmp    NetworkAdapterA!SendPackets+0x1a4 (f63fbbb0)
f63fbb9a9 mov     dword ptr [ecx+438h],eax
f63fbb9a9 mov     dl,byte ptr [esi+2BCh]
f63fbb9a9 mov     ecx,dword ptr [ebp+8]
f63fbb9a9 call   dword ptr [NetworkAdapterA!_imp_KfReleaseSpinLock (f640ca18)]

1: kd> ub 80a60147
hal!HalBuildScatterGatherList+0x1b0:
80a60130 je      hal!HalBuildScatterGatherList+0x1b9 (80a60139)
80a60132 mov    dword ptr [eax+4],1
80a60139 push   dword ptr [ebp+20h]
80a6013c push   eax
80a6013d mov    eax,dword ptr [ebp+0Ch]
80a60140 push   dword ptr [eax+14h]
80a60143 push   eax
80a60144 call   dword ptr [ebp+1Ch]
```

## Hardware Error

This pattern occurs frequently. It can be internal CPU malfunction due to overheating, RAM or hard disk I/O problem that usually results in the appropriate bugcheck. The most frequent one is the 6th from the top of bugcheck frequency table<sup>89</sup>:

- BUGCHECK 9C: MACHINE\_CHECK\_EXCEPTION

Other relevant bugchecks include:

- BUGCHECK 7B: INACCESSIBLE\_BOOT\_DEVICE
- BUGCHECK 77: KERNEL\_STACK\_INPAGE\_ERROR
- BUGCHECK 7A: KERNEL\_DATA\_INPAGE\_ERROR

Another bugcheck from this category can be triggered on purpose to get a crash dump of a hanging or slow system<sup>90</sup>:

- BUGCHECK 80: NMI\_HARDWARE\_FAILURE

Please also note that other popular bugchecks like

- BUGCHECK 7F: UNEXPECTED\_KERNEL\_MODE\_TRAP
- BUGCHECK 50: PAGE\_FAULT\_IN\_NONPAGED\_AREA

can result from RAM problems but we should try to find a software cause first.

Sometimes the following bugchecks like

- BUGCHECK 7E: SYSTEM\_THREAD\_EXCEPTION\_NOT\_HANDLED

report EXCEPTION\_DOESNOT\_MATCH\_CODE, where read or write address doesn't correspond to faulted instruction at EIP:

---

<sup>89</sup> Bugcheck Frequencies, Memory Dump Analysis Anthology, Volume 2, page 429

<sup>90</sup> NMI\_HARDWARE\_FAILURE, Memory Dump Analysis Anthology, Volume 1, page 135

```
SYSTEM_THREAD_EXCEPTION_NOT_HANDLED (7e)
This is a very common bugcheck. Usually the exception address pinpoints
the driver/function that caused the problem. Always note this address
as well as the link date of the driver/image that contains this address.

Arguments:
Arg1: c0000005, The exception code that was not handled
Arg2: bf802671, The address that the exception occurred at
Arg3: f10b8c74, Exception Record Address
Arg4: f10b88c4, Context Record Address

FAULTING_IP:
driver!AcquireSemaphoreShared+4
bf802671 90 nop

EXCEPTION_RECORD: f10b8c74 -- (.exr ffffffff10b8c74)
ExceptionAddress: bf802671 (driver!AcquireSemaphoreShared+0x00000004)
ExceptionCode: c0000005 (Access violation)
ExceptionFlags: 00000000
NumberParameters: 2
Parameter[0]: 00000001
Parameter[1]: 0000000c
Attempt to write to address 0000000c

CONTEXT: f10b88c4 -- (.cxr ffffffff10b88c4)
eax=884d2d01 ebx=0000000c ecx=00000000 edx=80010031 esi=8851ef60 edi=bc3846d4
eip=bf802671 esp=f10b8d3c ebp=f10b8d70 iopl=0 nv up ei pl nz na po nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010206
driver!AcquireSemaphoreShared+0x4:
bf802671 90 nop
Resetting default scope

WRITE_ADDRESS: 0000000c

EXCEPTION_DOESNOT_MATCH_CODE: This indicates a hardware error.
Instruction at bf802671 does not read/write to 0000000c
```

Code mismatch can also happen in user mode but from my experience it usually results from improper **Hooked Function** (page 488) or similar corruption:

```
EXCEPTION_RECORD: ffffffff -- (.exr 0xfffffffffffffff)
ExceptionAddress: 7c848768 (ntdll!_LdrpInitialize+0x00000184)
ExceptionCode: c0000005 (Access violation)
ExceptionFlags: 00000001
NumberParameters: 0

DEFAULT_BUCKET_ID: CODE_ADDRESS_MISMATCH

WRITE_ADDRESS: f774f120
```

```

FAULTING_IP:
ntdll!_LdrpInitialize+184
7c848768 cc int 3

EXCEPTION_DOESNOT_MATCH_CODE: This indicates a hardware error.
Instruction at 7c848768 does not read/write to f774f120

STACK_TEXT:
0012fd14 7c8284c5 0012fd28 7c800000 00000000 ntdll!_LdrpInitialize+0x184
00000000 00000000 00000000 00000000 ntdll!KiUserApcDispatcher+0x25

```

In such cases, EIP might point to the middle of the expected instruction (see also **Wild Code**, page 1148):

```

FAULTING_IP:
+59c3659
059c3659 86990508f09b xchg bl,byte ptr [ecx-640FF7FBh]

```

Here is another example of the real hardware error (note the concatenated error code for bugcheck 0x9C):

```

MACHINE_CHECK_EXCEPTION (9c)
A fatal Machine Check Exception has occurred.
KeBugCheckEx parameters;
x86 Processors
    If the processor has ONLY MCE feature available (For example Intel
    Pentium), the parameters are:
        1 - Low 32 bits of P5_MC_TYPE MSR
        2 - Address of MCA_EXCEPTION structure
        3 - High 32 bits of P5_MC_ADDR MSR
        4 - Low 32 bits of P5_MC_ADDR MSR
    If the processor also has MCA feature available (For example Intel
    Pentium Pro), the parameters are:
        1 - Bank number
        2 - Address of MCA_EXCEPTION structure
        3 - High 32 bits of MCI_STATUS MSR for the MCA bank that had the error
        4 - Low 32 bits of MCI_STATUS MSR for the MCA bank that had the error
IA64 Processors
    1 - Bugcheck Type
        1 - MCA_ASSERT
        2 - MCA_GET_STATEINFO
            SAL returned an error for SAL_GET_STATEINFO while processing MCA.
        3 - MCA_CLEAR_STATEINFO
            SAL returned an error for SAL_CLEAR_STATEINFO while processing MCA.
        4 - MCA_FATAL
            FW reported a fatal MCA.
        5 - MCA_NONFATAL
            SAL reported a recoverable MCA and we don't support currently
            support recovery or SAL generated an MCA and then couldn't
            produce an error record.
    0xB - INIT_ASSERT
    0xC - INIT_GET_STATEINFO
        SAL returned an error for SAL_GET_STATEINFO while processing INIT event.
    0xD - INIT_CLEAR_STATEINFO
        SAL returned an error for SAL_CLEAR_STATEINFO while processing INIT event.
    0xE - INIT_FATAL
        Not used.

```

- 2 - Address of log
- 3 - Size of log
- 4 - Error code in the case of x GET STATEINFO or x CLEAR STATEINFO

AMD64 Processors

- 1 - Bank number
  - 2 - Address of MCA\_EXCEPTION structure
  - 3 - High 32 bits of Mci\_STATUS MSR for the MCA bank that had the error
  - 4 - Low 32 bits of Mci\_STATUS MSR for the MCA bank that had the error

## Arguments:

Arg1: 00000000

Arg2: 808a07a0

Arg3: be000300

Arg4: 1008081f

## Debugging Details:

NOTE: This is a hardware error. This error was reported by the CPU via Interrupt 18. This analysis will provide more information about the specific error. Please contact the manufacturer for additional information about this error and troubleshooting assistance.

This error is documented in the following publication:

- IA-32 Intel(r) Architecture Software Developer's Manual  
Volume 3: System Programming Guide

Bit Mask:

MA	Model Specific	MCA	
O ID	Other Information	Error Code	Error Code
0 SDP	_____ _____	_____ _____	_____ _____
AEUECRC			
LRCNVVC			
^^^^^^^			

VAL - MCi STATUS register is valid

IA32\_MCI\_STATUS register is valid  
Indicates that the information contained within the IA32\_MCI\_STATUS register is valid. When this flag is set, the processor follows the rules given for the OVER flag in the IA32\_MCI\_STATUS register when overwriting previously valid entries. The processor sets the VAL flag and software is responsible for clearing it.

UC - Error Uncorrected

Indicates that the processor did not or was not able to correct the error condition. When clear, this flag indicates that the processor was able to correct the error condition.

EN - Error Enabled

Indicates that the error was enabled by the associated EEj bit of the IA32\_MCI\_CTL register.

MISCV - IA32\_MCi\_MISC Register Valid  
 Indicates that the IA32\_MCi\_MISC register contains additional information regarding the error. When clear, this flag indicates that the IA32\_MCi\_MISC register is either not implemented or does not contain additional information regarding the error.

ADDRV - IA32\_MCi\_ADDR register valid  
 Indicates that the IA32\_MCi\_ADDR register contains the address where the error occurred.

PCC - Processor Context Corrupt  
 Indicates that the state of the processor might have been corrupted by the error condition detected and that reliable restarting of the processor may not be possible.

BUSCONNERR - Bus and Interconnect Error    BUS{LL}\_{PP}\_{RRRR}\_{II}\_{T}\_err  
 These errors match the format 0000 1PPT RRRR IILL

**Concatenated Error Code:**  
 -----  
~~\_VAL\_UC\_EN\_MISCV\_ADDRV\_PCC\_BUSCONNERR\_1F~~

*This error code can be reported back to the manufacturer.  
 They may be able to provide additional information based upon this error. All questions regarding STOP 0x9C should be directed to the hardware manufacturer.*

BUGCHECK\_STR: 0x9C\_IA32\_GenuineIntel

DEFAULT\_BUCKET\_ID: DRIVER\_FAULT

PROCESS\_NAME: Idle

CURRENT\_IRQL: 2

LAST\_CONTROL\_TRANSFER: from 80a7fb08 to 8087b6be

STACK\_TEXT:  
 f773d280 80a7fb08 0000009c 00000000 f773d2b0 nt!KeBugCheckEx+0x1b  
 f773d3b4 80a7786f f7737fe0 00000000 00000000 hal!HalpMcaExceptionHandler+0x11e  
 f773d3b4 f75a9ca2 f7737fe0 00000000 00000000 hal!HalpMcaExceptionHandlerWrapper+0x77  
 f78c6d50 8083abf2 00000000 0000000e 00000000 intelppm!AcpiC1Idle+0x12  
 f78c6d54 00000000 0000000e 00000000 00000000 nt!KiIdleLoop+0xa

## Comments

Another possibility of a hardware error: frequent multiple unrelated bugchecks and bugchecks in memory dumps with valid instructions at faulting IP. Beware also about misaligned IP that can look like a valid instruction.

Additional examples:

```
WHEA_UNCORRECTABLE_ERROR (124)
A fatal hardware error has occurred. Parameter 1 identifies the type of error
source that reported the error. Parameter 2 holds the address of the
WHEA_ERROR_RECORD structure that describes the error condition.
Arguments:
Arg1: 0000000000000000, Machine Check Exception
Arg2: ffffffa8004b46748, Address of the WHEA_ERROR_RECORD structure.
Arg3: 0000000000000000, High order 32-bits of the Mci_STATUS value.
Arg4: 0000000000000000, Low order 32-bits of the Mci_STATUS value.

1: kd> dt -r _WHEA_ERROR_RECORD ffffffa8004b46748
hal!_WHEA_ERROR_RECORD
+0x000 Header : _WHEA_ERROR_RECORD_HEADER
  +0x000 Signature : 0x52455043
  +0x004 Revision : _WHEA_REVISION
    +0x000 MinorRevision : 0x10 ''
    +0x001 MajorRevision : 0x2 ''
    +0x000 AsUSHORT : 0x210
  +0x006 SignatureEnd : 0xffffffff
  +0x00a SectionCount : 3
  +0x00c Severity : 1 ( WheaErrSevFatal )
  +0x010 ValidBits : _WHEA_ERROR_RECORD_HEADER_VALIDBITS
    +0x000 PlatformId : 0y0
    +0x000 Timestamp : 0y1
    +0x000 PartitionId : 0y0
    +0x000 Reserved : 0y00000000000000000000000000000000 (0)
    +0x000 AsULONG : 2
  +0x014 Length : 0x3a0
  +0x018 Timestamp : _WHEA_TIMESTAMP
    +0x000 Seconds : 0y00100010 (0x22)
    +0x000 Minutes : 0y00101011 (0x2b)
    +0x000 Hours : 0y00001100 (0xc)
    +0x000 Precise : 0y0
    +0x000 Reserved : 0y00000000 (0)
    +0x000 Day : 0y00010110 (0x16)
    +0x000 Month : 0y00000100 (0x4)
    +0x000 Year : 0y00001010 (0xa)
    +0x000 Century : 0y00010100 (0x14)
    +0x000 AsLARGE_INTEGER : _LARGE_INTEGER 0x140a0416`000c2b22
  +0x020 PlatformId : _GUID {00000000-0000-0000-0000-000000000000}
    +0x000 Data1 : 0
    +0x004 Data2 : 0
    +0x006 Data3 : 0
    +0x008 Data4 : [8] ""
  +0x030 PartitionId : _GUID {00000000-0000-0000-0000-000000000000}
    +0x000 Data1 : 0
    +0x004 Data2 : 0
    +0x006 Data3 : 0
    +0x008 Data4 : [8] ""
  +0x040 CreatorId : _GUID {cf07c4bd-b789-4e18-b3c4-1f732cb57131}
    +0x000 Data1 : 0xcf07c4bd
    +0x004 Data2 : 0xb789
    +0x006 Data3 : 0x4e18
    +0x008 Data4 : [8] "???""
  +0x050 NotifyType : _GUID {e8f56ffe-919c-4cc5-ba88-65abe14913bb}
```

```

+0x000 Data1 : 0xe8f56ffe
+0x004 Data2 : 0x919c
+0x006 Data3 : 0x4cc5
+0x008 Data4 : [8] "???"
+0x060 RecordId : 0x01cae219`673474d3
+0x068 Flags : _WHEA_ERROR_RECORD_HEADER_FLAGS
    +0x000 Recovered : 0y0
    +0x000 PreviousError : 0y1
    +0x000 Simulated : 0y0
    +0x000 Reserved : 0y00000000000000000000000000000000 (0)
    +0x000 AsULONG : 2
+0x06c PersistenceInfo : _WHEA_PERSISTENCE_INFO
    +0x000 Signature : 0y0000000000000000 (0)
    +0x000 Length : 0y00000000000000000000000000000000 (0)
    +0x000 Identifier : 0y0000000000000000 (0)
    +0x000 Attributes : 0y0
    +0x000 DoNotLog : 0y0
    +0x000 Reserved : 0y00000 (0)
    +0x000 AsULONGLONG : 0
+0x074 Reserved : [12] ""
+0x080 SectionDescriptor : [1] _WHEA_ERROR_RECORD_SECTION_DESCRIPTOR
    +0x000 SectionOffset : 0x158
    +0x004 SectionLength : 0xc0
    +0x008 Revision : _WHEA_REVISION
        +0x000 MinorRevision : 0x1 ''
        +0x001 MajorRevision : 0x2 ''
        +0x000 AsUSHORT : 0x201
    +0x00a ValidBits : _WHEA_ERROR_RECORD_SECTION_DESCRIPTOR_VALIDBITS
        +0x000 FRUID : 0y0
        +0x000 FRUText : 0y0
        +0x000 Reserved : 0y000000 (0)
        +0x000 AsUCHAR : 0 ''
    +0x00b Reserved : 0 ''
+0x00c Flags : _WHEA_ERROR_RECORD_SECTION_DESCRIPTOR_FLAGS
    +0x000 Primary : 0y1
    +0x000 ContainmentWarning : 0y0
    +0x000 Reset : 0y0
    +0x000 ThresholdExceeded : 0y0
    +0x000 ResourceNotAvailable : 0y0
    +0x000 LatentError : 0y0
    +0x000 Reserved : 0y00000000000000000000000000000000 (0)
    +0x000 AsULONG : 1
+0x010 SectionType : _GUID {9876ccad-47b4-4bdb-b65e-16f193c4f3db}
    +0x000 Data1 : 0x9876ccad
    +0x004 Data2 : 0x47b4
    +0x006 Data3 : 0x4bdb
    +0x008 Data4 : [8] "???"
+0x020 FRUID : _GUID {00000000-0000-0000-0000-000000000000}
    +0x000 Data1 : 0
    +0x004 Data2 : 0
    +0x006 Data3 : 0
    +0x008 Data4 : [8] ""
+0x030 SectionSeverity : 1 ( WheaErrSevFatal )
+0x034 FRUText : [20] ""

```

**KERNEL\_STACK\_INPAGE\_ERROR (77)**  
The requested page of kernel data could not be read in. Caused by bad block in paging file or disk controller error.  
In the case when the first arguments is 0 or 1, the stack signature in the kernel stack was not found. Again, bad hardware.  
An I/O status of c000009c (STATUS\_DEVICE\_DATA\_ERROR) or C000016AL (STATUS\_DISK\_OPERATION\_FAILED) normally indicates the data could not be read from the disk due to a bad block. Upon reboot autocheck will run and attempt to map out the bad sector. If the status is C0000185 (STATUS\_IO\_DEVICE\_ERROR) and the paging file is on a SCSI disk device, then the cabling and termination should be checked. See the knowledge base article on SCSI termination.  
Arguments:  
Arg1: 0000000000000001, (page was retrieved from disk)  
Arg2: fffffa800818e870, value found in stack where signature should be  
Arg3: 0000000000000000, 0  
Arg4: fffff8800c6e5e80, address of signature on kernel stack

```
2: kd> k
Child-SP RetAddr Call Site
fffff880`0371da18 fffff800`03110b01 nt!KeBugCheckEx
fffff880`0371da20 fffff800`030c8c54 nt! ?? ::FNODOBFM::`string'+0x51e31
fffff880`0371db30 fffff800`030c8bef nt!MmInPageKernelStack+0x40
fffff880`0371db90 fffff800`030c8928 nt!KiInSwapKernelStacks+0x1f
fffff880`0371dbc0 fffff800`0332be5a nt!KeSwapProcessOrStack+0x84
fffff880`0371dc00 fffff800`03085d26 nt!PspSystemThreadStartup+0x5a
fffff880`0371dc40 00000000`00000000 nt!KiStartSystemThread+0x16
```

For WHEA\_UNCORRECTABLE\_ERROR (124) we have additional WinDbg commands **!whea**, **!errrec**, and **!errpkt**:

```
2: kd> !whea
Error Source Table @ fffff8004bbd4a90
4 Error Sources
Error Source 0 @ fffffe00014376bd0
Notify Type : {14374010-e000-ffff-984a-bd4b00f8ffff}
Type : 0x0 (MCE)
Error Count : 1
Record Count : 4
Record Length : 728
Error Records : wrapper @ fffffe000110e0000 record @ fffffe000110e0028
: wrapper @ fffffe000110e0728 record @ fffffe000110e0750
: wrapper @ fffffe000110e0e50 record @ fffffe000110e0e78
: wrapper @ fffffe000110e1578 record @ fffffe000110e15a0
Descriptor : @ fffffe00014376c29
Length : 3cc
Max Raw Data Length : 141
Num Records To Preallocate : 4
Max Sections Per Record : 4
Error Source ID : 0
Flags : 00000000
[...]
```

```
2: kd> !errrec fffffe000110e0028
=====
Common Platform Error Record @ fffffe000110e0028

Record Id : 01d21a1a7e5ffffd1
Severity : Fatal (1)
Length : 928
```

Creator : Microsoft  
Notify Type : Machine Check Exception  
Timestamp : 9/30/2016 9:05:50 (UTC)  
Flags : 0x00000000

=====

Section 0 : Processor Generic

=====

Descriptor @ fffffe000110e00a8

Section @ fffffe000110e0180

Offset : 344

Length : 192

Flags : 0x00000001 Primary

Severity : Fatal

Proc. Type : x86/x64

Instr. Set : x64

Error Type : Micro-Architectural Error

Flags : 0x00

CPU Version : 0x00000000000306a9

Processor ID : 0x0000000000000002

=====

Section 1 : x86/x64 Processor Specific

=====

Descriptor @ fffffe000110e00f0

Section @ fffffe000110e0240

Offset : 536

Length : 128

Flags : 0x00000000

Severity : Fatal

Local APIC Id : 0x0000000000000002

CPU Id : a9 06 03 00 00 08 10 02 - bf e3 ba 7f ff fb eb bf  
00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  
00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00

Proc. Info 0 @ fffffe000110e0240

=====

Section 2 : x86/x64 MCA

=====

Descriptor @ fffffe000110e0138

Section @ fffffe000110e02c0

Offset : 664

Length : 264

Flags : 0x00000000

Severity : Fatal

Error : Internal unclassified (Proc 2 Bank 4)

Status : 0xb200000000100402

## Hidden Call

Sometimes, due to optimization or indeterminate stack trace reconstruction, we may not see all stack trace frames. In some cases, it is possible to reconstruct such **Hidden Calls**. For example, we have the following unmanaged **Stack Trace** (page 926) of **CLR Thread** (page 124):

```
0:000> k
ChildEBP RetAddr
0011d6b8 66fdee7c mscorwks!JIT_IsInstanceOfClass+0xd
0011d6cc 67578500 PresentationCore_ni!`string'+0x4a2bc
0011d6e0 67578527 PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x778500)
0011d6f4 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x778527)
0011d708 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d71c 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d730 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d744 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d758 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d76c 67578527 PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d780 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x778527)
0011d794 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d7a8 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d7bc 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d7d0 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d7e4 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d7f8 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d80c 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d820 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d834 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d848 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d85c 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d870 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d884 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d898 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d8ac 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d8c0 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d8d4 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d8e8 67578527 PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d8fc 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x778527)
0011d910 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d924 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d938 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d94c 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d960 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d974 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d988 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d99c 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d9b0 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d9c4 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d9d8 67578527 PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d9ec 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x778527)
0011da00 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011da14 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011da28 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011da3c 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
```

```

0011da50 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011da64 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011da78 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011da8c 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011daa0 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dab4 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dac8 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dad0 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011daf0 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011db04 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011db18 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011db2c 67578527 PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011db40 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x778527)
0011db54 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011db68 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011db7c 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011db90 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dba4 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dbb8 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dbcc 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dbe0 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dbf4 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dc08 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dc1c 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dc30 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dc44 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dc58 66fc3282 PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
*** WARNING: Unable to verify checksum for PresentationFramework.ni.dll
0011dd28 662a75e6 PresentationCore_ni!`string'+0x2e6c2
0011de08 662190a0 PresentationFramework_ni+0x2675e6
0011dff0 66fc35e2 PresentationFramework_ni+0x1d90a0
0011e0ec 66fd9dad PresentationCore_ni!`string'+0x2ea22
0011e214 66fe0459 PresentationCore_ni!`string'+0x451ed
0011e238 66fdfd40 PresentationCore_ni!`string'+0x4b899
0011e284 66fdfc9b PresentationCore_ni!`string'+0x4b180
*** WARNING: Unable to verify checksum for WindowsBase.ni.dll
0011e2b0 723ca31a PresentationCore_ni!`string'+0x4b0db
0011e2cc 723ca20a WindowsBase_ni+0x9a31a
0011e30c 723c8384 WindowsBase_ni+0x9a20a
0011e330 723cd26d WindowsBase_ni+0x98384
0011e368 723cd1f8 WindowsBase_ni+0x9d26d
0011e380 72841b4c WindowsBase_ni+0x9d1f8
0011e390 728589ec mscorwks!CallDescrWorker+0x33
0011e410 72865acc mscorwks!CallDescrWorkerWithHandler+0xa3
0011e54c 72865aff mscorwks!MethodDesc::CallDescr+0x19c
0011e568 72865b1d mscorwks!MethodDesc::CallTargetWorker+0x1f
0011e580 728bd9c8 mscorwks!MethodDescCallSite::CallWithValueTypes+0x1a
0011e74c 728bdb1e mscorwks!ExecuteCodeWithGuaranteedCleanupHelper+0x9f
*** WARNING: Unable to verify checksum for mscorlib.ni.dll
0011e7fc 68395887 mscorwks!ReflectionInvocation::ExecuteCodeWithGuaranteedCleanup+0x10f
0011e818 683804b5 mscorlib_ni+0x235887
0011e830 723cd133 mscorlib_ni+0x2204b5
0011e86c 723c7a27 WindowsBase_ni+0x9d133
0011e948 723c7d13 WindowsBase_ni+0x97a27
0011e984 723ca4fe WindowsBase_ni+0x97d13
0011e9d0 723ca42a WindowsBase_ni+0x9a4fe

```

```

0011e9f0 723ca31a WindowsBase_ni+0x9a42a
0011ea0c 723ca20a WindowsBase_ni+0x9a31a
0011ea4c 723c8384 WindowsBase_ni+0x9a20a
0011ea70 723c74e1 WindowsBase_ni+0x98384
0011eaac 723c7430 WindowsBase_ni+0x974e1
0011eadc 723c9b6c WindowsBase_ni+0x97430
0011eb2c 757462fa WindowsBase_ni+0x99b6c
0011eb58 75746d3a user32!InternalCallWinProc+0x23
0011ebd0 757477c4 user32!UserCallWinProcCheckWow+0x109
0011ec30 7574788a user32!DispatchMessageWorker+0x3bc
0011ec40 0577304e user32!DispatchMessageW+0xf
WARNING: Frame IP not in any known module. Following frames may be wrong.
0011ec5c 723c7b24 0x577304e
0011eccc 723c71f9 WindowsBase_ni+0x97b24
0011ecd8 723c719c WindowsBase_ni+0x971f9
0011ece4 6620f07e WindowsBase_ni+0x9719c
0011ecf0 6620e37f PresentationFramework_ni+0x1cf07e
0011ed14 661f56d6 PresentationFramework_ni+0x1ce37f
0011ed24 661f5699 PresentationFramework_ni+0x1b56d6
0011ed80 72841b4c PresentationFramework_ni+0x1b5699
0011eda0 72841b4c mscorewks!CallDescrWorker+0x33
0011edb0 728589ec mscorewks!CallDescrWorker+0x33
0011ee30 72865acc mscorewks!CallDescrWorkerWithHandler+0xa3
0011ef6c 72865aff mscorewks!MethodDesc::CallDescr+0x19c
0011ef88 72865b1d mscorewks!MethodDesc::CallTargetWorker+0x1f
0011efa0 728fef01 mscorewks!MethodDescCallSite::CallWithValueTypes+0x1a
0011f104 728fee21 mscorewks!ClassLoader::RunMain+0x223
0011f36c 728ff33e mscorewks!Assembly::ExecuteMainMethod+0xa6
0011f83c 728ff528 mscorewks!SystemDomain::ExecuteMainMethod+0x45e
0011f88c 728ff458 mscorewks!ExecuteEXE+0x59
0011f8d4 70aef4f3 mscorewks!_CorExeMain+0x15c
0011f90c 70b77efd mscoreei!_CorExeMain+0x10a
0011f924 70b74de3 mscoreei!ShellShim__CorExeMain+0x7d
0011f92c 754c338a mscoreei!_CorExeMain_Exported+0x8
0011f938 77659f72 kernel32!BaseThreadInitThunk+0xe
0011f978 77659f45 ntdll!__RtlUserThreadStart+0x70
0011f990 00000000 ntdll!_RtlUserThreadStart+0x1b

```

Its **Managed Stack Trace** (page 624) is the following:

```

0:000> !CLRStack
OS Thread Id: 0x1520 (0)
ESP      EIP
0011e7a0 728493a4 [HelperMethodFrame_PROTECTOBJ: 0011e7a0]
System.Runtime.CompilerServices.RuntimeHelpers.ExecuteCodeWithGuaranteedCleanup(TryCode, CleanupCode, System.Object)
0011e808 68395887 System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
0011e824 683804b5 System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
0011e83c 723cd133 System.Windows.Threading.DispatcherOperation.Invoke()
0011e874 723c7a27 System.Windows.Threading.Dispatcher.ProcessQueue()
0011e950 723c7d13 System.Windows.Threading.Dispatcher.WndProcHook(IntPtr, Int32, IntPtr, IntPtr, Boolean ByRef)
0011e99c 723ca4fe MS.Win32.HwndWrapper.WndProc(IntPtr, Int32, IntPtr, IntPtr, Boolean ByRef)
0011e9e8 723ca42a MS.Win32.HwndSubclass.DispatcherCallbackOperation(System.Object)
0011e9f8 723ca31a System.Windows.Threading.ExceptionWrapper.InternalRealCall(System.Delegate, System.Object, Boolean)
0011ea1c 723ca20a System.Windows.Threading.ExceptionWrapper.TryCatchWhen(System.Object, System.Delegate, System.Object,
Boolean, System.Delegate)
0011ea64 723c8384 System.Windows.Threading.Dispatcher.WrappedInvoke(System.Delegate, System.Object, Boolean,

```

```

System.Delegate)
0011ea84 723c74e1 System.Windows.Threading.Dispatcher.InvokeImpl(System.Windows.Threading.DispatcherPriority,
System.TimeSpan, System.Delegate, System.Object, Boolean)
0011eac8 723c7430 System.Windows.Threading.Dispatcher.Invoke(System.Windows.Threading.DispatcherPriority,
System.Delegate, System.Object)
0011eaec 723c9b6c MS.Win32.HwndSubclass.SubclassWndProc(IntPtr, Int32, IntPtr, IntPtr)
0011ec74 00270b04 [NDirectMethodFrameStandalone: 0011ec74]
MS.Win32.UnsafeNativeMethods.DispatchMessage(System.Windows.Interop.MSG ByRef)
0011ec84 723c7b24 System.Windows.Threading.Dispatcher.PushFrameImpl(System.Windows.Threading.DispatcherFrame)
0011ecd4 723c71f9 System.Windows.Threading.Dispatcher.PushFrame(System.Windows.Threading.DispatcherFrame)
0011ece0 723c719c System.Windows.Threading.Dispatcher.Run()
0011ec6c 6620f07e System.Windows.Application.RunDispatcher(System.Object)
0011ecf8 6620e37f System.Windows.Application.RunInternal(System.Windows.Window)
0011ed1c 661f56d6 System.Windows.Application.Run(System.Windows.Window)
0011ed2c 661f5699 System.Windows.Application.Run()
[...]

```

**Caller-n-Callee** (page 111) traces also don't reveal anything more:

```

Thread 0
Current frame: mscorwks!JIT_IsInstanceOfClass+0xd
ChildEBP RetAddr Caller,Callee
0011d6b8 66fdee7c (MethodDesc 0x66ee2954 +0x3c
MS.Internal.DeferredElementTreeState.GetLogicalParent(System.Windows.DependencyObject,
MS.Internal.DeferredElementTreeState)), calling mscorwks!JIT_IsInstanceOfClass
0011d6cc 67578500 (MethodDesc 0x66ee1270 +0x110
MS.Internal.UIElementHelper.InvalidateAutomationAncestors(System.Windows.DependencyObject)), calling (MethodDesc
0x66ee2954 +0 MS.Internal.DeferredElementTreeState.GetLogicalParent(System.Windows.DependencyObject,
MS.Internal.DeferredElementTreeState))
0011d6e0 67578527 (MethodDesc 0x66ee1270 +0x137
MS.Internal.UIElementHelper.InvalidateAutomationAncestors(System.Windows.DependencyObject)), calling (MethodDesc
0x66ee1270 +0 MS.Internal.UIElementHelper.InvalidateAutomationAncestors(System.Windows.DependencyObject))
0011d6f4 6757850d (MethodDesc 0x66ee1270 +0x11d
MS.Internal.UIElementHelper.InvalidateAutomationAncestors(System.Windows.DependencyObject)), calling (MethodDesc
0x66ee1270 +0 MS.Internal.UIElementHelper.InvalidateAutomationAncestors(System.Windows.DependencyObject))
0011d708 6757850d (MethodDesc 0x66ee1270 +0x11d
MS.Internal.UIElementHelper.InvalidateAutomationAncestors(System.Windows.DependencyObject)), calling (MethodDesc
0x66ee1270 +0 MS.Internal.UIElementHelper.InvalidateAutomationAncestors(System.Windows.DependencyObject))
0011d71c 6757850d (MethodDesc 0x66ee1270 +0x11d
MS.Internal.UIElementHelper.InvalidateAutomationAncestors(System.Windows.DependencyObject)), calling (MethodDesc
0x66ee1270 +0 MS.Internal.UIElementHelper.InvalidateAutomationAncestors(System.Windows.DependencyObject))
[...]

```

However, if we check the return address for **Top Module** (page 1012) *mscorwks* (66fdee7c) we will see a call possibly related to 3D processing:

```

0:000> k
ChildEBP RetAddr
0011d6b8 66fdee7c mscorwks!JIT_IsInstanceOfClass+0xd
0011d6cc 67578500 PresentationCore_ni!`string'+0x4a2bc
0011d6e0 67578527 PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x778500)
0011d6f4 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x778527)
[...]

```

```
0:000> ub 66fdee7c
PresentationCore_ni!`string'+0x4a2a2:
66fdee62 740c      je     PresentationCore_ni!`string'+0x4a2b0 (66fdee70)
66fdee64 8bc8      mov    ecx,eax
66fdee66 8b01      mov    eax,dword ptr [ecx]
66fdee68 ff90d8030000 call   dword ptr [eax+3D8h]
66fdee6e 8bf0      mov    esi,eax
66fdee70 8bd7      mov    edx,edi
66fdee72 b998670467 mov    ecx,offset PresentationCore_ni!`string'+0xb1bd8 (67046798)
66fdee77 e82c7afaff call   PresentationCore_ni!?System.Windows.Media.Media3D.Viewport3DVisual.
PrecomputeContent@@200001+0x3c (66f868a8)
```

The call structure seems to be valid when we check the next return address from the stack trace (67578500):

```
0:000> ub 67578500
PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x7784e7):
675784e7 e8f4a2a0ff call   PresentationCore_ni!?System.Windows.Media.Media3D.ScaleTransform3D.
UpdateResource@@2002011280M802+0x108 (66f827e0)
PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x7784ec):
675784ec eb05      jmp   PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x7784f3)
(675784f3)
PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x7784ee):
675784ee b801000000 mov    eax,1
PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x7784f3):
675784f3 85c0      test   eax,eax
PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x7784f5):
675784f5 74b1      je    PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x7784a8)
(675784a8)
PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x7784f7):
675784f7 8bcb      mov    ecx,ebx
PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x7784f9):
675784f9 33d2      xor    edx,edx
PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x7784fb):
675784fb e84069a6ff call   PresentationCore_ni!`string'+0x4a280 (66fdee40)
```

## Hidden Exception

### Kernel Space

This is an example of **Hidden Exception** (page 457) pattern in kernel space:

```
0: kd> !thread
THREAD ffffffa800d4bf9c0 Cid 0e88.56e0 Peb: 000007fffffd8000 Win32Thread: 0000000000000000 RUNNING on
processor 0
Not impersonating
DeviceMap          ffffff8a001e91950
Owning Process    ffffffa800b33cb30      Image:        svchost.exe
Attached Process   N/A           Image:        N/A
Wait Start TickCount 13154529      Ticks: 0
Context Switch Count 1426
UserTime           00:00:00.015
KernelTime         00:00:00.124
Win32 Start Address 0x0000000077728d20
Stack Init fffff8800a83fdb0 Current fffff8800a83eb90
Base fffff8800a840000 Limit fffff8800a83a000 Call 0
Priority 10 BasePriority 10 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
[...]

0: kd> dps fffff8800a83a000 fffff8800a840000
[...]
ffffff880`0a83e180 fffff880`0a83ea10
ffffff880`0a83e188 fffff880`0a83e6d0
ffffff880`0a83e190 fffff880`0a83e968
ffffff880`0a83e198 fffff880`016c88cf nt!KiDispatchException+0x16f
ffffff880`0a83e1a0 fffff880`0a83e968
ffffff880`0a83e1a8 fffff880`0a83e1d0
ffffff880`0a83e1b0 fffff880`00000000
ffffff880`0a83e1b8 00000000`00000000
ffffff880`0a83e1c0 00000000`00000000
ffffff880`0a83e1c8 00000000`00000000
[...]

0: kd> .cxr fffff880`0a83e1d0
rax=0000000000000009 rbx=fffffa800d4c1de0 rcx=0000000000000000
rdx=fffffa800a83ece0 rsi=0000000000000000 rdi=0000000000000000
rip=fffff800016ad74f rsp=fffff8800a83eba0 rbp=00000000a000000c
r8=fffff8800a83ecd8 r9=fffff8800a83ecc0 r10=0000000000000000
r11=fffff8800a83ed58 r12=0000000000000000 r13=0000000000000000
r14=fffffa800d4bf9c0 r15=fffffa800d4c1ea0
iopl=0 nv up ei pl zr na po nc
cs=0010 ss=0018 ds=002b es=002b fs=0053 gs=002b efl=00010246
nt!IoCompleteRequest+0x12f:
fffff800`016ad74f 48894108 mov qword ptr [rcx+8],rax ds:002b:00000000`00000008=????????????????????
```

## Comments

Another example:

```

0: kd> k
# ChildEBP RetAddr
00 8078aefc 8281db8c hal!READ_PORT_USHORT+0x8
01 8078af0c 8281dcf5 hal!HalpCheckPowerButton+0x2e
02 8078af10 8292cdde hal!HaliHaltSystem+0x7
03 8078af5c 8292dc79 nt!KiBugCheckDebugBreak+0x73
04 8078b320 8292cc24 nt!KeBugCheck2+0xa7f
05 8078b340 82a5a49b nt!KeBugCheckEx+0x1e
06 8078bc90 828fe9c9 nt!PspSystemThreadStartup+0xde
07 00000000 00000000 nt!KiThreadStartup+0x19

0: kd> !thread
THREAD 863475f8 Cid 0004.0008 Teb: 00000000 Win32Thread: 00000000 RUNNING on processor 0
Not impersonating
DeviceMap 8d6080c0
Owning Process 863478d0 Image: System
Attached Process N/A Image: N/A
Wait Start TickCount 2624 Ticks: 7 (0:00:00:00.109)
Context Switch Count 1025 IdealProcessor: 0
UserTime 00:00:00.000
KernelTime 00:00:03.962
Win32 Start Address nt!Phase1Initialization (0x829dd53b)
Stack Init 8078bed0 Current 8078b890 Base 8078c000 Limit 80789000 Call 0
Priority 31 BasePriority 8 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
ChildEBP RetAddr Args to Child
8078aefc 8281db8c 00001000 00000000 8078af5c hal!READ_PORT_USHORT+0x8 (FPO: [1,0,0])
8078af0c 8281dcf5 8292cdde 2b2952aa 807c960c hal!HalpCheckPowerButton+0x2e (FPO: [Non-Fpo])
8078af10 8292cdde 2b2952aa 807c960c 00000000 hal!HaliHaltSystem+0x7 (FPO: [0,0,0])
8078af5c 8292dc79 00000004 00000000 00000000 nt!KiBugCheckDebugBreak+0x73
8078b320 8292cc24 0000007e c0000005 8cc14540 nt!KeBugCheck2+0xa7f
8078b340 82a5a49b 0000007e c0000005 8cc14540 nt!KeBugCheckEx+0x1e
8078bc90 828fe9c9 829dd53b 80806cb0 00000000 nt!PspSystemThreadStartup+0xde
00000000 00000000 00000000 00000000 nt!KiThreadStartup+0x19

0: kd> dps 80789000 8078c000
80789000 00000000
80789004 00000000
...
8078b470 8078b880
8078b474 82902277 nt!KiDispatchException+0x17c
8078b478 8078b89c
8078b47c 8078b480
8078b480 00010017
...
0: kd> .cxr 8078b480
eax=00000000 ebx=87428554 ecx=8078b998 edx=00000000 esi=871121d0 edi=0000008c
eip=8cc11340 esp=8078b964 ebp=8078ba28 iopl=0 nv up ei ng nz ac po nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00210292
Driver+0x1340:
8cc11340 ff5000 call dword ptr [eax] ds:0023:00000000=????????
```

## User Space

This pattern occurs frequently. It manifests itself when we run `!analyze -v` command, and we don't see an exception, or we see only a breakpoint hit. In this case, manual analysis is required. Sometimes this happens because of another pattern: **Multiple Exceptions** (page 714). In other cases an exception happens, and it is handled by an exception handler dismissing it, and a process continues its execution slowly accumulating corruption inside its data leading to a new crash or hang. Sometimes we see a process hanging during its termination like the case shown below.

We have a process dump with only one thread:

```
0:000> kv
ChildEBP RetAddr
0096fc0c 7c822124 ntdll!KiFastSystemCallRet
0096fce0 77e6baa8 ntdll!NtWaitForSingleObject+0xc
0096fd50 77e6ba12 kernel32!WaitForSingleObjectEx+0xac
0096fd64 67f016ce kernel32!WaitForSingleObject+0x12
0096fd78 7c82257a component!DllInitialize+0xc2
0096fd98 7c8118b0 ntdll!LdrpCallInitRoutine+0x14
0096fe34 77e52fea ntdll!LdrShutdownProcess+0x130
0096ff20 77e5304d kernel32!_ExitProcess+0x43
0096ff34 77bcade4 kernel32!ExitProcess+0x14
0096ff40 77bcfaefb msrvct!__crtExitProcess+0x32
0096ff70 77bcfa6d msrvct!_cinit+0xd2
0096ff84 77bcbb555 msrvct!_exit+0x11
0096ffb8 77e66063 msrvct!_endthreadex+0xc8
0096ffec 00000000 kernel32!BaseThreadStart+0x34
```

We can look at its raw stack and try to find the symbol address for *KiUserExceptionDispatcher*. This function calls *RtlDispatchException*:

```
0:000> !teb
TEB at 7ffd000
  ExceptionList:      0096fd40
  StackBase:          00970000
  StackLimit:         0096a000
  SubSystemTib:       00000000
  FiberData:          00001e00
  ArbitraryUserPointer: 00000000
  Self:               7ffd000
  EnvironmentPointer: 00000000
  ClientId:           000000858 . 0000008c0
  RpcHandle:          00000000
  Tls Storage:        00000000
  PEB Address:        7ffdd000
  LastErrorValue:     0
  LastStatusValue:    c0000135
  Count Owned Locks:  0
  HardErrorMode:      0
```

```
0:000> dds 0096a000 00970000
...
...
...
0096c770  7c8140cc ntdll!RtlDispatchException+0x91
0096c774  0096c808
0096c778  0096ffa8
0096c77c  0096c824
0096c780  0096c7e4
0096c784  77bc6c74 msvcrt!_except_handler3
0096c788  00000000
0096c78c  0096c808
0096c790  01030064
0096c794  00000000
0096c798  00000000
0096c79c  00000000
0096c7a0  00000000
0096c7a4  00000000
0096c7a8  00000000
0096c7ac  00000000
0096c7b0  00000000
0096c7b4  00000000
0096c7b8  00000000
0096c7bc  00000000
0096c7c0  00000000
0096c7c4  00000000
0096c7c8  00000000
0096c7cc  00000000
0096c7d0  00000000
0096c7d4  00000000
0096c7d8  00000000
0096c7dc  00000000
0096c7e0  00000000
0096c7e4  00000000
0096c7e8  00970000
0096c7ec  00000000
0096c7f0  0096caf0
0096c7f4  7c82ecc6 ntdll!KiUserExceptionDispatcher+0xe
0096c7f8  0096c000
0096c7fc  0096c824 ; a pointer to an exception context
0096c800  0096c808
0096c804  0096c824
0096c808  c0000005
0096c80c  00000000
0096c810  00000000
0096c814  77bd8df3 msvcrt!wcschr+0x15
0096c818  00000002
0096c81c  00000000
0096c820  01031000
0096c824  0001003f
0096c828  00000000
0096c82c  00000000
0096c830  00000000
0096c834  00000000
0096c838  00000000
0096c83c  00000000
```

A second parameter to both functions is a pointer to the so-called exception context (processor state when an exception occurred). We can use `.cxr` command to change thread execution context to what it was at exception time:

```
0:000> .cxr 0096c824
[...]
msvcrt!wcschr+0x15:
77bd8df3 668b08 mov cx, word ptr [eax] ds:0023:01031000=????
```

After changing the context, we can see the thread stack prior to that exception:

```
0:000> kL
ChildEBP RetAddr
0096caf0 67b11808 msvcrt!wcschr+0x15
0096cb10 67b1194d component2!function1+0x50
0096cb24 67b11afb component2!function2+0x1a
0096eb5c 67b11e10 component2!function3+0x39
0096ed94 67b14426 component2!function4+0x155
0096fdc0 67b164b7 component2!function5+0x3b
0096fdcc 00402831 component2!function6+0x5b
0096feec 0096ff14 program!function+0x1d1
0096ffec 00000000 kernel32!BaseThreadStart+0x34
```

We see that the exception happened when *component2* was searching a Unicode string for a character (`wcschr`). Most likely the string was not zero terminated.

To summarize and show the common exception handling path in user space here is another thread stack taken from a different dump:

```
ntdll!KiFastSystemCallRet
ntdll!NtWaitForMultipleObjects+0xc
kernel32!UnhandledExceptionFilter+0x746
kernel32!_except_handler3+0x61
ntdll!ExecuteHandler2+0x26
ntdll!ExecuteHandler+0x24
ntdll!RtlDispatchException+0x91
ntdll!KiUserExceptionDispatcher+0xe
ntdll!RtlpCoalesceFreeBlocks+0x36e ; crash is here
ntdll!RtlFreeHeap+0x38e
msvcrt!free+0xc3
msvcrt!_freefls+0x124
msvcrt!_freeptd+0x27
msvcrt!__CRTL DLL _INIT+0x1da
ntdll!LdrpCallInitRoutine+0x14
ntdll!LdrShutdownThread+0xd2
kernel32!ExitThread+0x2f
kernel32!BaseThreadStart+0x39
```

When *RtlpCoalesceFreeBlocks* (this function compacts a heap, and it is called from *RtlFreeHeap*) does an illegal memory access, then this exception is first processed in kernel, and because it happened in user space and mode the execution is transferred to *RtlDispatchException* which searches for exception handlers, and in this case there is a default one installed: *UnhandledExceptionFilter*. If we see the latter function on a call stack we can manually get exception context, and from it, a thread stack leading to it, like in this example:

```
0:000> ~*kv
...
...
...
. 0 Id: 1568.68c Suspend: 1 Teb: 7ffde000 Unfrozen
ChildEBP RetAddr  Args to Child
...
...
...
0012a984 715206e0 0012a9ac 7800bdb5 0012a9b4 KERNEL32!UnhandledExceptionFilter+0x140 (FPO: [Non-Fpo])
...
...
...
0:000> dt _EXCEPTION_POINTERS 0012a9ac
+0x000 ep_xrecord : 0x12aa78
+0x004 ep_context : 0x12aa94

0:000> .cxr 0012aa94
eax=00000000 ebx=00000000 ecx=00000000 edx=7283e058 esi=0271a60c edi=00000000
eip=35c5f973 esp=0012ad60 ebp=0012ad7c iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=0038 gs=0000 efl=00010246
componentA!InternalFoo+0x21:
35c5f973 8b01 mov eax,dword ptr [ecx] ds:0023:00000000=????????
```

We can also search for exception codes like c0000005 using scripts to dump raw stack data<sup>91</sup>. For example:

```
007cfa40 017d0000
007cfa44 007cf90
007cfa48 7c82855e ntdll!KiUserExceptionDispatcher+0xe
007cfa4c 7c826d9b ntdll!NtContinue+0xc
007cfa50 7c82856c ntdll!KiUserExceptionDispatcher+0x1c
007cfa54 007cfa78
007cfa58 00000000
007cfa5c c0000005
007cfa60 00000000
007cfa64 00000000
007cfa68 0100e076 component!foo+0x1c4
007cfa6c 00000002
007cfa70 00000001
007cfa74 00000000
007cfa78 0001003f
007cfa7c 00000003
```

---

<sup>91</sup> WinDbg Scripts, Memory Dump Analysis Anthology, Volume 1, pages 231-240

```
007cfa80 000000b0
007cfa84 00000001
007cfa88 00000000
[...]

1: kd> .cxr 007cfa78
eax=01073bb0 ebx=7ffd9000 ecx=00000050 edx=01073bb0 esi=000003e5 edi=00000000
eip=0100e076 esp=007cf44 ebp=007cf900 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
component!foo+0x1c4:
001b:0100e076 891a mov dword ptr [edx],ebx ds:0023:01073bb0=????????
```

The presence of unloaded fault handling modules can be the sign of **Hidden Exceptions** too:

```
Unloaded modules:
697b0000 697c7000 faultrep.dll
Timestamp: Fri Mar 25 02:11:44 2005 (42437360)
Checksum: 0001DC38
```

## Comments

---

Sometimes we can spot 0001003f and its address can be the beginning of a context record:

```
[...]
0070f668 00000000
0070f66c 00000000
0070f670 00000000
0070f674 00000000
0070f678 00000000
0070f67c 0001003f
[...]
```

On Windows 10 RSP below KiUserExceptionDispatch can be used as an address for **.cxr** command:

```
[...]
00000023`d432f4e8 00000023`d3e3c190
00000023`d432f4f0 00000000`00000000
00000023`d432f4f8 00007ffa`e5c5577a ntdll!KiUserExceptionDispatch+0x3a
00000023`d432f500 00000000`00000000
00000023`d432f508 00000000`00000810
[...]

0:001> .cxr 00000023`d432f500
rax=0000000000000000 rbx=0000000000000000 rcx=00007ff676f399b0
rdx=0000000000000000 rsi=00000023d3e3c190 rdi=00007ff676f211e0
rip=00007ff676f2120d rsp=00000023d432fc30 rbp=0000000000000000
r8=00006f6d5c4f5ead r9=0000000000000032 r10=0000000000000032
r11=000000023d432f960 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0 nv up ei pl nz na po nc
cs=0033 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010206
Application+0x120d:
00007ff6`76f2120d c70000000000 mov dword ptr [rax],0 ds:00000000`00000000=????????
```

## Hidden IRP

Sometimes we suspect a particular thread is doing I/O, but IRP is missing in the output of **!thread** WinDbg command. The way to proceed is to examine the list of IRPs and associated threads from the output of **!irpfind** command. Here is a synthesized example from a few **Virtualized** (page 1075) **Young System** (page 1156) crash dumps:

```
0: kd> !thread ffffffa8004e2d280

THREAD ffffffa8004e2d280 Cid 0004.0020 Teb: 0000000000000000 Win32Thread: 0000000000000000 WAIT:
(Executive) KernelMode Non-Alertable
fffff880009ec440 NotificationEvent
Not impersonating
[...]

0: kd> !irpfind

Irp [ Thread ] irpStack: (Mj,Mn) DevObj [Driver] MDL Process
[...]
fffffa800424e4e0 [fffffa8004e2d280] irpStack: (3, 0) ffffffa8004ed6d40 [ \Driver\DriverA]
[...]
```

Now we can inspect the found IRP (**!irp** command) and device object (for example, by using **!devobj** and **!devstack** commands). Sometimes we can see the same IRP address as **Execution Residue** (page 371) among “*Args to Child*” values in the output of **!thread** command or **kv** (if the thread is current).

## Hidden Module

Sometimes we look for modules that were loaded and unloaded at some time. **!m** command lists unloaded modules, but some of them could be mapped to address space without using the runtime loader. The latter case is common for drm-type protection tools, rootkits, malware or crimeware which can influence a process execution. In such cases, we can hope that they still remain in virtual memory and search for them. WinDbg **.imgscan** command greatly helps in identifying MZ/PE module headers. The following example illustrates this command without implying that the found module did any harm:

```
0:000> .imgscan
MZ at 000d0000, prot 00000002, type 01000000 - size 6000
  Name: usrcptn.dll
MZ at 00350000, prot 00000002, type 01000000 - size 9b000
  Name: ADVAPI32.dll
MZ at 00400000, prot 00000002, type 01000000 - size 23000
  Name: javaw.exe
MZ at 01df0000, prot 00000002, type 01000000 - size 8b000
  Name: OLEAUT32.dll
MZ at 01e80000, prot 00000002, type 01000000 - size 52000
  Name: SHLWAPI.dll
...
...
```

We don't see *usrcptn* in either loaded or unloaded module lists:

```
0:002> !m
start      end        module name
00350000  003eb000  advapi32
00400000  00423000  javaw
01df0000  01e7b000  oleaut32
01e80000  01ed2000  shlwapi
...
Unloaded modules:
```

Then we can use **Unknown Component** pattern (page 1035) to see the module resources if present in memory:

```
0:002> !dh 000d0000
...
SECTION HEADER #4
.rsrc name
  418 virtual size
  4000 virtual address
   600 size of raw data
  1600 file pointer to raw data
    0 file pointer to relocation table
    0 file pointer to line numbers
    0 number of relocations
    0 number of line numbers
40000040 flags
```

```

Initialized Data
(no align specified)
Read Only

...
0:002> dc 000d0000+4000 L418
...
000d4140 ... n...z.)...F.i.l.
000d4150 ... e.D.e.s.c.r.i.p.
000d4160 ... t.i.o.n....U.s.
000d4170 ... e.r. .D.u.m.p. .
000d4180 ... U.s.e.r. .M.o.d.
000d4190 ... e. .E.x.c.e.p.t.
000d41a0 ... i.o.n. .D.i.s.p.
000d41b0 ... a.t.c.h.e.r....
0:002> du 000d416C
000d416c "User Dump User Mode Exception Di"
000d41ac "spatcher"

```

This component seems to be loaded or mapped only if the userdump package is fully installed: *usrxcptn.dll* is a part of its redistribution package, and the application is added to Process Dumper applet in Control Panel. Although from the memory dump comment we also see that the dump was taken manually using command line *userdump.exe*, we see that the full userdump package was additionally installed and that was probably not necessary<sup>92</sup>:

```

Loading Dump File [javaw.dmp]
User Mini Dump File with Full Memory: Only application data is available

Comment: 'Userdump generated complete user-mode minidump with Standalone function on COMPUTER-NAME'

```

## Comments

---

**.imgscan** may not be able to find all hidden modules (**Debugger Omission** pattern, page 220).

---

<sup>92</sup> Correcting Microsoft Article about *userdump.exe*, Memory Dump Analysis Anthology, Volume 1, page 612

## Hidden Parameter

This pattern is a variant of **Execution Residue** (page 371) and **String Parameter** (page 962) where we have parameters left out from a stack trace due to register calling conventions and compiler optimizations. However, using raw stack analysis of a region around stack frames of interest we find what we are looking for. Here's an example from an x64 system blocked thread waiting for data from a named pipe:

```
0: kd> kL
*** Stack trace for last set context - .thread/.cxr resets it
Child-SP      RetAddr          Call Site
fffffa60`2c3627d0  ffffff800`018b90fa nt!KiSwapContext+0x7f
fffffa60`2c362910  ffffff800`018add3b nt!KiSwapThread+0x13a
fffffa60`2c362980  ffffff800`01b2121f nt!KeWaitForSingleObject+0x2cb
fffffa60`2c362a10  ffffff800`01b319b6 nt!IopXxxControlFile+0xdeb
fffffa60`2c362b40  ffffff800`018b68f3 nt!NtFsControlFile+0x56
fffffa60`2c362bb0  00000000`778d6eaa nt!KiSystemServiceCopyEnd+0x13
00000000`11f4da68  00000000`77767b6e ntdll!ZwFsControlFile+0xa
00000000`11f4da70  000007fe`ff94abc8 kernel32!WaitNamedPipeW+0x22f
00000000`11f4db60  000007fe`ff98a32d RPCRT4!NdrProxyForwardingFunction255+0x814d
00000000`11f4dc30  000007fe`ff98918b RPCRT4!OSF_CCONNECTION::TransOpen+0xcd
00000000`11f4dcc0  000007fe`ff988f9b RPCRT4!OSF_CCONNECTION::OpenConnectionAndBind+0x17b
00000000`11f4dd90  000007fe`ff988ec6 RPCRT4!OSF_CCALL::BindToServer+0xbb
00000000`11f4de40  000007fe`ff983368 RPCRT4!OSF_BINDING_HANDLE::InitCCallWithAssociation+0xa5
00000000`11f4dea0  000007fe`ff983220 RPCRT4!OSF_BINDING_HANDLE::AllocateCCall+0x118
00000000`11f4dfd0  000007fe`ffa1f740 RPCRT4!OSF_BINDING_HANDLE::NegotiateTransferSyntax+0x30
00000000`11f4e020  000007fe`ffa1fecb RPCRT4!Ndr64pClientSetupTransferSyntax+0x200
00000000`11f4e080  000007fe`ffa20281 RPCRT4!NdrpClientCall3+0x6b
00000000`11f4e2d0  000007fe`fe087c8c RPCRT4!NdrClientCall3+0xdd
[...]
```

Even if we disassemble the return address of a caller of *WaitNamedPipeW* function we won't easily find the passed first string parameter (named pipe name) unless we do a substantial reverse engineering and data flow analysis:

```
0: kd> ub 000007fe`ff94abc8
RPCRT4!_imp_load_getaddrinfo+0x7:
000007fe`ff94ab9f jmp RPCRT4!_tailMerge_WS2_32_dll (000007fe`ff94cef8)
000007fe`ff94aba4 call qword ptr [RPCRT4!_imp_GetLastError (000007fe`ffa2d528)]
000007fe`ff94abaa mov r12d,eax
000007fe`ff94abad cmp r12d,0E7h
000007fe`ff94abb4 jne RPCRT4!NdrProxyForwardingFunction255+0x8193 (000007fe`ff99c8fb)
000007fe`ff94abba mov edx,3E8h
000007fe`ff94abbf mov rcx,rsi
000007fe`ff94abc2 call qword ptr [RPCRT4!_imp_WaitNamedPipeW (000007fe`ffa2d3f8)]
```

However, dumping raw stack data around corresponding frames gives us pipe name clue and possible service name to look further:

```
0: kd> dpu 00000000`11f4da70
00000000`11f4da70 00000000`11f4dba8 "\\.\PIPE\ServiceArpc"
00000000`11f4da78 00000000`00000000
00000000`11f4da80 00000000`00000000
00000000`11f4da88 00000000`000003e8
00000000`11f4da90 00000000`11f4db30
00000000`11f4da98 00000000`00110018
00000000`11f4daa0 00000000`0d9001a0
00000000`11f4daa8 00000000`0000001a
00000000`11f4dab0 00000000`00000000
00000000`11f4dab8 00000000`00000000
00000000`11f4dac0 00000000`0020000c
00000000`11f4dac8 00000000`0d9001e2 "ServiceArpc"
00000000`11f4dad0 00000000`00000000
00000000`11f4dad8 00000000`00000000
00000000`11f4dae0 00000000`00240022
```

## Hidden Process

Not all processes are linked into a list that some commands traverse such as **!process 0 0**. A process may unlink itself or be in an initialization stage. However, a process structure is allocated from the nonpaged pool, and such pool can be searched for "Proc" pool tag (unless a process changes that in memory). For example:

```
0: kd> !poolfind Proc

Searching NonPaged pool (83c3c000 : 8bc00000) for Tag: Proc

*87b15000 size: 298 previous size: 0 (Free) Pro.
*87b18370 size: 298 previous size: 98 (Allocated) Proc (Protected)
[...]
*8a35e900 size: 298 previous size: 30 (Allocated) Proc (Protected)
*8a484000 size: 298 previous size: 0 (Allocated) Proc (Protected)
*8a4a2d68 size: 298 previous size: 28 (Allocated) Proc (Protected)
[...]
```

One such structure is missing from the active process linked list (note that it has a parent PID):

```
0: kd> !process 8a484000+20
PROCESS 8a484020 SessionId: 0 Cid: 05a0 Peb: 00000000 ParentCid: 0244
DirBase: bffc2200 ObjectTable: e17e6a78 HandleCount: 0.
Image: AppChild.exe
VadRoot 8a574f80 Vads 4 Clone 0 Private 3. Modified 0. Locked 0.
DeviceMap e1002898
Token e1a36030
Elapsed Time 00:00:00.000
User Time 00:00:00.000
Kernel Time 419 Days 13:24:16.625
QuotaPoolUsage[PagedPool] 7580
QuotaPoolUsage[NonPagedPool] 160
Working Set Sizes (now,min,max) (12, 50, 345) (48KB, 200KB, 1380KB)
PeakWorkingSetSize 12
Virtual Size 1 Mb
PeakVirtual Size 1 Mb
PageFault Count 5
Memory Priority BACKGROUND
Base Priority 8
Commit Charge 156

No active threads
```

We may think that this process is a zombie (note that unlike terminated processes it has a non-zero data such as VAD and object table and zero PEB and elapsed time) but inspection of its parent process thread stacks reveals that it was in the process of creation (note an attached process field):

```
THREAD 8a35dad8 Cid 0244.0248 Teb: 7fffd000 Win32Thread: bc3aa688 WAIT: (Unknown) KernelMode Non-Alertable
ba971608 NotificationEvent
Impersonation token: e2285030 (Level Impersonation)
DeviceMap e1a31a58
Owning Process 8a35e920 Image: AppParent.exe
Attached Process 8a484020 Image: AppChild.exe
Wait Start TickCount 2099 Ticks: 1 (0:00:00:00.015)
Context Switch Count 279 LargeStack
UserTime 00:00:00.046
KernelTime 00:00:00.046
Win32 Start Address AppParent!mainCRTStartup (0x0100d303)
Start Address kernel32!BaseProcessStartThunk (0x77e617f8)
Stack Init ba972000 Current ba971364 Base ba972000 Limit ba96e000 Call 0
Priority 8 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr
ba97137c 80833f2d nt!KiSwapContext+0x26
ba9713a8 80829c72 nt!KiSwapThread+0x2e5
ba9713f0 bad3c9db nt!KeWaitForSingleObject+0x346
[...]
ba971b94 8094cf3 nt!MmCreatePeb+0x2cc
ba971ce4 8094d42d nt!PspCreateProcess+0x5a9
ba971d38 8088b4ac nt!NtCreateProcessEx+0x77
ba971d38 7c82845c nt!KiFastCallEntry+0xfc (TrapFrame @ ba971d64)
0006f498 7c826d09 ntdll!KiFastSystemCall1Ret
0006f49c 77e6cf95 ntdll!ZwCreateProcessEx+0xc
0006fcc0 7d1ec670 kernel32!CreateProcessInternalW+0x15e5
0006fd0c 01008bcf ADVAPI32!CreateProcessAsUserW+0x108
[...]
```

## Hidden Stack Trace

Sometimes, a stack trace from **Stack Trace Collection** (page 943) may look well-formed at first sight, for example, having an expected start frames:

```
0:000> ~*k

[...]

# 19 Id: 16a4.21f4 Suspend: 0 Teb: 7e95b000 Unfrozen
ChildEBP RetAddr
0c2de6b0 74eb112f ntdll!NtWaitForMultipleObjects+0xc
0c2de83c 76ca7b89 KERNELBASE!WaitForMultipleObjectsEx+0xcc
0c2de858 76d007bf kernel32!WaitForMultipleObjects+0x19
0c2dec98 76d00295 kernel32!WerFaultInternal+0x50b
0c2deca8 76ce1709 kernel32!WerReportFault+0x74
0c2decb0 74f5f705 kernel32!BaseReportFault+0x19
0c2ded3c 76fb4f84 KERNELBASE!UnhandledExceptionFilter+0x1f4
0c2ded54 76fb5728 ntdll!TppExceptionFilter+0x30
0c2ded64 76f5c95a ntdll!TppWorkerpInnerExceptionFilter+0xe
0c2df914 76ca7c04 ntdll!TppWorkerThread+0x87f5a
0c2df928 76f1ad1f kernel32!BaseThreadInitThunk+0x24
0c2df970 76f1acea ntdll!_RtlUserThreadStart+0x2f
0c2df980 00000000 ntdll!_RtlUserThreadStart+0x1b

[...]
```

So, we may think something wrong happened in *ntdll!TppWorkerThread* code (although 0x87f5a offset looks suspicious). However, in reality, in this case, due to exception filter logic (or some other reason in different cases), we have **Hidden Stack Trace**. When looking at *UnhandledExceptionFilter* parameters (or raw stack as in the case of **Hidden Exceptions**, page 455), we find an exception context:

```
0:019> kv
ChildEBP RetAddr Args to Child
0c2de6b0 74eb112f 00000003 0c2de880 00000001 ntdll!NtWaitForMultipleObjects+0xc
0c2de83c 76ca7b89 00000003 0c2de880 00000000 KERNELBASE!WaitForMultipleObjectsEx+0xcc
0c2de858 76d007bf 00000003 0c2de880 00000000 kernel32!WaitForMultipleObjects+0x19
0c2dec98 76d00295 00000000 00000001 00000000 kernel32!WerFaultInternal+0x50b
0c2deca8 76ce1709 0c2ded3c 74f5f705 0c2ded94 kernel32!WerReportFault+0x74
0c2decb0 74f5f705 0c2ded94 00000001 a79b7895 kernel32!BaseReportFault+0x19
0c2ded3c 76fb4f84 0c2ded94 00000000 KERNELBASE!UnhandledExceptionFilter+0x1f4
0c2ded54 76fb5728 00000000 00000000 0c2df914 ntdll!TppExceptionFilter+0x30
0c2ded64 76f5c95a 0c2df8d0 76f00a70 0c2df914 ntdll!TppWorkerpInnerExceptionFilter+0xe
0c2df914 76ca7c04 0f79e380 76ca7be0 a5b45024 ntdll!TppWorkerThread+0x87f5a
0c2df928 76f1ad1f 0f79e380 a59141d7 00000000 kernel32!BaseThreadInitThunk+0x24
0c2df970 76f1acea ffffffff 76f00233 00000000 ntdll!_RtlUserThreadStart+0x2f
0c2df980 00000000 76ed4a00 0f79e380 00000000 ntdll!_RtlUserThreadStart+0x1b

0:019> dd 0c2ded94 L2
0c2ded94 0c2deef8 0c2def48
```

```
0:019> .cxr 0c2def48
eax=15f237e5 ebx=15f235e9 ecx=15f237e1 edx=7e95b000 esi=15f237e1 edi=09724b10
eip=76f00fb2 esp=0c2df3ac ebp=0c2df3ac iopl=0 nv up ei pl nz na po nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b ef=00010202
ntdll!RtlEnterCriticalSection+0x12:
76f00fb2 f00fba3000 lock btr dword ptr [eax],0 ds:002b:15f237e5=?????????
```

```
0:019> k
*** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr
0c2df3ac 7407999c ntdll!RtlEnterCriticalSection+0x12
0c2df3cc 7407acd6 ModuleA!DoWork+0x1b
[...]
0c2df73c 76ee3aa7 ModuleA!ThreadPoolWorkCallback+0xa9
0c2df77c 76ee1291 ntdll!TppWorkpExecuteCallback+0x137
0c2df914 76ca7c04 ntdll!TppWorkerThread+0x48e
0c2df928 76f1ad1f kernel32!BaseThreadInitThunk+0x24
0c2df970 76f1acea ntdll!__RtlUserThreadStart+0x2f
0c2df980 00000000 ntdll!_RtlUserThreadStart+0x1b
```

We consider this a different pattern than **Hidden Call** (page 450) because an entire stack (sub)trace is missing between *UnhandledExceptionFilter* and thread start frames:

```
ntdll!NtWaitForMultipleObjects
KERNELBASE!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
kernel32!WerFaultInternal
kernel32!WerReportFault
kernel32!BaseReportFault
KERNELBASE!UnhandledExceptionFilter
[...]
ntdll!TppWorkerThread
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

This pattern is also different from **Past Stack Trace** (page 800) pattern because **Hidden Stack Trace** belongs to PRESENT time zone. Our example is also different from **Hidden Exception** (page 455) analysis pattern and its recovered stack trace because exception processing is not hidden and shows **Exception Stack Trace** (page 363) albeit with a hidden part.

We were also fortunate to have **Stored Exception** (page 959, accessible by **!analyze -v** command):

```
0:019> .exr -1
ExceptionAddress: 76f00fb2 (ntdll!RtlEnterCriticalSection+0x00000012)
ExceptionCode: c0000005 (Access violation)
ExceptionFlags: 00000000
NumberParameters: 2
Parameter[0]: 00000001
Parameter[1]: 15f237e5
Attempt to write to address 15f237e5
```

```
0:019> .ecxr
eax=15f237e5 ebx=15f235e9 ecx=15f237e1 edx=7e95b000 esi=15f237e1 edi=09724b10
eip=76f00fb2 esp=0c2df3ac ebp=0c2df3ac iopl=0 nv up ei pl nz na po nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010202
ntdll!RtlEnterCriticalSection+0x12:
76f00fb2 f00fba3000 lock btr dword ptr [eax],0 ds:002b:15f237e5=????????
```

So this **Hidden Stack Trace** is detected straightforwardly. But in other cases, such as when we have **Multiple Exceptions** (page 714) in a process dump or **Stack Trace Collection** (page 943) from a complete memory dump, we need to pay attention to such a possibility.

## High Contention

### .NET CLR Monitors

This is a high contention pattern variant where the contention is around a monitor object. For example, we have a **Distributed CPU Spike** (page 243) for some threads:

```
0:000> !runaway
User Mode Time
Thread      Time
 9:6ff4    0 days 0:07:39.019
12:6b88    0 days 0:06:19.786
11:6bf0    0 days 0:06:13.889
10:6930    0 days 0:06:09.240
16:3964    0 days 0:05:44.483
17:6854    0 days 0:05:35.326
13:668c    0 days 0:05:35.123
14:5594    0 days 0:05:34.858
15:7248    0 days 0:05:23.111
 2:c54     0 days 0:00:41.215
 4:1080    0 days 0:00:00.349
 7:10f0    0 days 0:00:00.302
 0:c3c     0 days 0:00:00.271
[...]
```

If we look at their stack traces, we find them all blocked trying to enter a monitor<sup>93</sup>, for example:

```
0:000> ~*k
[...]
12 Id: d50.6b88 Suspend: 0 Teb: 000007ff`ffffd8000 Unfrozen
Child-SP RetAddr Call Site
00000000`1a98e798 000007fe`fd0c1420 ntdll!ZwWaitForMultipleObjects+0xa
00000000`1a98e7a0 00000000`76e82cf3 KERNELBASE!WaitForMultipleObjectsEx+0xe8
00000000`1a98e8a0 000007fe`f82e0669 kernel32!WaitForMultipleObjectsExImplementation+0xb3
00000000`1a98e930 000007fe`f82dbe9 mscorwks!WaitForMultipleObjectsEx_SO_TOLERANT+0xc1
00000000`1a98e9d0 000007fe`f82a0569 mscorwks!Thread::DoAppropriateAptStateWait+0x41
00000000`1a98ea30 000007fe`f82beaec mscorwks!Thread::DoAppropriateWaitWorker+0x191
00000000`1a98eb30 000007fe`f81f1b9a mscorwks!Thread::DoAppropriateWait+0x5c
00000000`1a98eba0 000007fe`f82fd3c9 mscorwks!CLREvent::WaitEx+0xbe
00000000`1a98ec50 000007fe`f81ac6be mscorwks!AwareLock::EnterEpilog+0xc9
00000000`1a98ed20 000007fe`f81c7b2b mscorwks!AwareLock::Enter+0x72
00000000`1a98ed50 000007fe`f87946af mscorwks!AwareLock::Contention+0x1fb
00000000`1a98ee20 000007ff`00161528 mscorwks!JITUtil_MonContention+0xdf
00000000`1a98efd0 000007ff`0016140e 0x7ff`00161528
00000000`1a98f040 000007ff`00167271 0x7ff`0016140e
```

<sup>93</sup> [http://en.wikipedia.org/wiki/Monitor\\_\(synchronization\)](http://en.wikipedia.org/wiki/Monitor_(synchronization))

```

00000000`1a98f0a0 000007fe`f74e2bbb 0x7ff`00167271
00000000`1a98f130 000007fe`f753ed76 mscorelib_ni+0x2f2bbb
00000000`1a98f180 000007fe`f8390282 mscorelib_ni+0x34ed76
00000000`1a98f1d0 000007fe`f8274363 mscorewks!CallDescrWorker+0x82
00000000`1a98f220 000007fe`f8274216 mscorewks!CallDescrWorkerWithHandler+0xd3
00000000`1a98f2c0 000007fe`f81c96a7 mscorewks!DispatchCallDebuggerWrapper+0x3e
00000000`1a98f320 000007fe`f830ae42 mscorewks!DispatchCallNoEH+0x5f
00000000`1a98f3a0 000007fe`f81bdc00 mscorewks!AddTimerCallback_Worker+0x92
00000000`1a98f430 000007fe`f82a41a5 mscorewks!ManagedThreadCallState::IsAppDomainEqual+0x4c
00000000`1a98f480 000007fe`f82df199 mscorewks!SVR::gc_heap::make_heap_segment+0x155
00000000`1a98f550 000007fe`f82cececa mscorewks!DoOpenIAssemblyStress::DoOpenIAssemblyStress+0x99
00000000`1a98f590 000007fe`f830c0db mscorewks!AddTimerCallbackEx+0xba
00000000`1a98f650 000007fe`f81ebb37 mscorewks!ThreadPoolMgr::AsyncTimerCallbackCompletion+0x53
00000000`1a98f6b0 000007fe`f81fe92a mscorewks!UnmanagedPerAppDomainTPCount::DispatchWorkItem+0x157
00000000`1a98f750 000007fe`f81bb1fc mscorewks!ThreadPoolMgr::WorkerThreadStart+0x1ba
00000000`1a98f7f0 00000000`76e7652d mscorewks!Thread::intermediateThreadProc+0x78
00000000`1a98fcc0 00000000`76fac521 kernel32!BaseThreadInitThunk+0xd
00000000`1a98fcf0 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

```

[...]

```

15 Id: d50.7248 Suspend: 0 Teb: 000007ff`ffee6000 Unfrozen
Child-SP RetAddr Call Site
00000000`1c16e6f0 000007fe`f87946af mscorewks!AwareLock::Contention+0x13b
00000000`1c16e7c0 000007ff`0016135e mscorewks!JITUtil_MonContention+0xdf
00000000`1c16e970 000007ff`0016726b 0x7ff`0016135e
00000000`1c16e9c0 000007fe`f74e2bbb 0x7ff`0016726b
00000000`1c16ea50 000007fe`f753ed76 mscorelib_ni+0x2f2bbb
00000000`1c16eaa0 000007fe`f8390282 mscorelib_ni+0x34ed76
00000000`1c16eaf0 000007fe`f8274363 mscorewks!CallDescrWorker+0x82
00000000`1c16eb40 000007fe`f8274216 mscorewks!CallDescrWorkerWithHandler+0xd3
00000000`1c16ebe0 000007fe`f81c96a7 mscorewks!DispatchCallDebuggerWrapper+0x3e
00000000`1c16ec40 000007fe`f830ae42 mscorewks!DispatchCallNoEH+0x5f
00000000`1c16ecc0 000007fe`f81bdc00 mscorewks!AddTimerCallback_Worker+0x92
00000000`1c16ed50 000007fe`f82a41a5 mscorewks!ManagedThreadCallState::IsAppDomainEqual+0x4c
00000000`1c16eda0 000007fe`f82df199 mscorewks!SVR::gc_heap::make_heap_segment+0x155
00000000`1c16ee70 000007fe`f82cececa mscorewks!DoOpenIAssemblyStress::DoOpenIAssemblyStress+0x99
00000000`1c16eeb0 000007fe`f830c0db mscorewks!AddTimerCallbackEx+0xba
00000000`1c16ef70 000007fe`f81ebb37 mscorewks!ThreadPoolMgr::AsyncTimerCallbackCompletion+0x53
00000000`1c16efd0 000007fe`f81fe92a mscorewks!UnmanagedPerAppDomainTPCount::DispatchWorkItem+0x157
00000000`1c16f070 000007fe`f81bb1fc mscorewks!ThreadPoolMgr::WorkerThreadStart+0x1ba
00000000`1c16f110 00000000`76e7652d mscorewks!Thread::intermediateThreadProc+0x78
00000000`1c16f9e0 00000000`76fac521 kernel32!BaseThreadInitThunk+0xd
00000000`1c16fa10 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

```

[...]

Thread #15 seems was caught at the time when it was trying to enter and not waiting yet. If we check a monitor object the thread #12 tries to enter we see it has an address 01af0be8:

```
0:000> !u 000007ff`00161528
Normal JIT generated code
[...]
000007ff`00161505 90          nop
000007ff`00161506 48b8f089ae1100000000 mov rax,11AE89F0h
000007ff`00161510 488b00      mov     rax,qword ptr [rax]
000007ff`00161513 48894528      mov     qword ptr [rbp+28h],rax
000007ff`00161517 488b4528      mov     rax,qword ptr [rbp+28h]
000007ff`0016151b 48894518      mov     qword ptr [rbp+18h],rax
000007ff`0016151f 488b4d28      mov     rcx,qword ptr [rbp+28h]
000007ff`00161523 e8b8d422f8    call   msCorwks!JIT_MonEnter (000007fe`f838e9e0)
>>> 000007ff`00161528 90        nop
000007ff`00161529 90        nop
000007ff`0016152a 90        nop
[...]
000007ff`001615d2 4883c430    add    rsp,30h
000007ff`001615d6 5d        pop    rbp
000007ff`001615d7 f3c3       rep    ret

0:000> dps 11AE89F0h 11
00000000`11ae89f0  00000000`01af0be8
```

This object seems to be owned by the thread #17:

```
0:000> !syncblk
Index      SyncBlock MonitorHeld Recursion Owning Thread Info      SyncBlock Owner
1362 000000001ba7b6c0      15      1 000000001c0173b0  6854  17  000000001af0be8 System.Object
[...]
```

This thread seems to be blocked in ALPC:

```
0:017> k
Child-SP RetAddr Call Site
00000000`1d55c9e8 000007fe`fee1a776 ntdll!NtAlpcSendWaitReceivePort+0xa
00000000`1d55c9f0 000007fe`fee14e42 rpcrt4!LRPC_CCALL::SendReceive+0x156
00000000`1d55cab0 000007fe`ff0828c0 rpcrt4!I_RpcSendReceive+0x42
00000000`1d55cae0 000007fe`ff08282f ole32!ThreadSendReceive+0x40
00000000`1d55cb30 000007fe`ff08265b ole32!CRpcChannelBuffer::SwitchAptAndDispatchCall+0xa3
00000000`1d55cbd0 000007fe`fef3daaa ole32!CRpcChannelBuffer::SendReceive2+0x11b
00000000`1d55cd90 000007fe`fef3da0c ole32!CAptRpcChnl::SendReceive+0x52
00000000`1d55ce60 000007fe`ff08205d ole32!CCtxComChnl::SendReceive+0x68
00000000`1d55cf10 000007fe`feebfd61 ole32!NdrExtpProxySendReceive+0x45
00000000`1d55cf40 000007fe`ff07f82f rpcrt4!NdrpClientCall2+0x9ea
00000000`1d55d6b0 000007fe`fef3d8a2 ole32!ObjectStublessClient+0x1ad
00000000`1d55da40 000007fe`fa511ba8 ole32!ObjectStubless+0x42
[...]
```

## Critical Sections

Similar to kernel mode involving executive resources (page 477), **High Contention** pattern can be observed in user space involving critical sections guarding shared regions like serialized process heap or a memory database, for example, in one Windows service process during increased workload:

```
0:000> !locks

CritSec +310608 at 00310608
WaiterWoken      No
LockCount        6
RecursionCount   1
OwningThread   d9c
EntryCount       0
ContentionCount  453093
*** Locked
```

```
CritSec +8f60f78 at 08f60f78
WaiterWoken      No
LockCount        8
RecursionCount   1
OwningThread   d9c
EntryCount       0
ContentionCount  af7f0
*** Locked
```

```
CritSec +53bf8f10 at 53bf8f10
WaiterWoken      No
LockCount        0
RecursionCount   1
OwningThread    1a9c
EntryCount       0
ContentionCount  e
*** Locked
```

```
Scanned 7099 critical sections
```

When looking at the owning thread, we see that the contention involves process heap:

```
0:000> ~~[d9c]kL
ChildEBP RetAddr
0e2ff9d4 7c81e845 nt!RtlpFindAndCommitPages+0x14e
0e2ffa0c 7c81e4ef nt!RtlpExtendHeap+0xa6
0e2ffc38 7c3416b3 nt!RtlAllocateHeap+0x645
0e2ffc78 7c3416db ms!_heap_alloc+0xe0
0e2ffc80 7c3416f8 ms!_nh_malloc+0x10
0e2ffc8c 672e14fd ms!_malloc+0xf
0e2ffc98 0040bc28 nt!MemAlloc+0xd
...
0e2fff84 7c349565 nt!WorkItemThread+0x152
0e2ffb8 77e6608b ms!_endthreadex+0xa0
0e2fffec 00000000 kernel32!BaseThreadStart+0x34
```

However, two critical section addresses belong to the same heap:

```
0:000> !address 00310608
00310000 : 00310000 - 00010000
    Type      00020000 MEM_PRIVATE
    Protect   00000004 PAGE_READWRITE
    State     00001000 MEM_COMMIT
    Usage     RegionUsageHeap
    Handle    00310000

0:000> !address 08f60f78
08f30000 : 08f30000 - 00200000
    Type      00020000 MEM_PRIVATE
    Protect   00000004 PAGE_READWRITE
    State     00001000 MEM_COMMIT
    Usage     RegionUsageHeap
    Handle    00310000
```

Lock contention is confirmed by heap statistics as well:

```
0:000> !heap -s
LFH Key          : 0x07262959
  Heap   Flags  Reserv Commit   Virt   Free  List   UCR   Virt  Lock  Fast
                (k)    (k)    (k)    (k) length      blocks cont. heap
00140000 00000002    8192   2876   3664    631   140     46    0    1e   L
  External fragmentation 21 % (140 free blocks)
00240000 00008000      64     12     12     10     1     1    0    0
Virtual block: 0ea20000 - 0ea20000 (size 00000000)
Virtual block: 0fa30000 - 0fa30000 (size 00000000)
00310000 00001002 1255320 961480 1249548 105378      0 16830      2 453093  L
  Virtual address fragmentation 23 % (16830 uncommitted ranges)
  Lock contention 4534419
003f0000 00001002      64     36     36      0     0     1    0    0   L
00610000 00001002      64     16     16      4     2     1    0    0   L
...
```

## Executive Resources

Some Windows synchronization objects like executive resources and critical sections have a struct member called *ContentionCount*. This is the number of times a resource was accessed or, in other words, it is the accumulated number of threads waiting for an object: when a thread tries to acquire an object and is put into a waiting state the count is incremented. Hence the name of this pattern: **High Contention**.

Here is an example. In a kernel memory dump, we have just one exclusively owned lock, and it seems that no other threads were blocked by it at the time the dump was saved. However, the high contention count reveals CPU spike (**Spiking Thread**, page 888):

```
3: kd> !locks
**** DUMP OF ALL RESOURCE OBJECTS ****
KD: Scanning for held locks...

Resource @ 0x8abc11f0    Exclusively owned
Contention Count = 19648535
Threads: 896395f8-01<*>
KD: Scanning for held locks...

Resource @ 0x896fab88    Shared 1 owning threads
Threads: 88c78608-01<*>
KD: Scanning for held locks...
15464 total locks, 2 locks currently held

3: kd> !thread 896395f8
THREAD 896395f8  Cid 04c0.0138  Teb: 7ffdde000 Win32Thread: bc922d20 RUNNING on processor 1
Not impersonating
DeviceMap          e3d4c008
Owning Process     8a035020      Image:        MyApp.exe
Wait Start TickCount 36969283      Ticks: 0
Context Switch Count 1926423           LargeStack
UserTime            00:00:53.843
KernelTime          00:13:10.703
Win32 Start Address 0x00401478
Start Address 0x77e617f8
Stack Init ba14b000 Current ba14abf8 Base ba14b000 Limit ba146000 Call 0
Priority 11 BasePriority 6 PriorityDecrement 5
ChildEBP RetAddr
ba14ac94 bf8c6505 001544c8 bf995948 000c000a nt!_wcsicmp+0x3a
ba14ace0 bf8c6682 00000000 00000000 00000000 win32k!_FindWindowEx+0xfb
ba14ad48 8088978c 00000000 00000000 0012f8d4 win32k!NtUserFindWindowEx+0xef
ba14ad48 7c8285ec 00000000 00000000 0012f8d4 nt!KiFastCallEntry+0xfc
```

```
3: kd> !process 8a035020
PROCESS 8a035020 SessionId: 9 Cid: 04c0 Peb: 7ffdf000 ParentCid: 10e8
  DirBase: cffaf7a0 ObjectTable: e4ba30a0 HandleCount: 73.
  Image: MyApp.exe
  VadRoot 88bc1bf8 Vads 82 Clone 0 Private 264. Modified 0. Locked 0.
  DeviceMap e3d4c008
  Token e5272028
Elapsed Time 00:14:19.360
  UserTime 00:00:53.843
  KernelTime 00:13:10.703
  QuotaPoolUsage[PagedPool] 40660
  QuotaPoolUsage[NonPagedPool] 3280
  Working Set Sizes (now,min,max) (1139, 50, 345) (4556KB, 200KB, 1380KB)
  PeakWorkingSetSize 1141
  VirtualSize 25 Mb
  PeakVirtualSize 27 Mb
  PageFaultCount 1186
  MemoryPriority BACKGROUND
  BasePriority 6
  CommitCharge 315
```

## Comments

**High Contention** may also be visible from non-locked objects too:

```
0: kd> !locks -v
**** DUMP OF ALL RESOURCE OBJECTS ****

Resource @ nt!ExpFirmwareTableResource (0xfffff80001a315c0) Available

Resource @ nt!PsLoadedModuleResource (0xfffff80001a596a0) Available

Resource @ nt!MmSectionExtendResource (0xfffff80001a59440) Available

Resource @ nt!MmSectionExtendSetResource (0xfffff80001a594c0) Available
Contention Count = 68

Resource @ nt!SepRmDbLock (0xfffff80001a2e340) Available
Contention Count = 1710

Resource @ nt!SepRmDbLock (0xfffff80001a2e3a8) Available
Contention Count = 387

Resource @ nt!SepRmDbLock (0xfffff80001a2e410) Available
Contention Count = 256

Resource @ nt!SepRmDbLock (0xfffff80001a2e478) Available
Contention Count = 212

Resource @ nt!SepRmGlobalSaclLock (0xfffff80001a2e4e0) Available

Resource @ nt!SepLsaAuditQueueInfo (0xfffff80001a2e240) Available

Resource @ nt!SepLsaDeletedLogonQueueInfo (0xfffff80001a2e0e0) Available

Resource @ 0xfffffa804dc09920 Available

Resource @ nt!PnpRegistryDeviceResource (0xfffff80001a8f9e0) Available
Contention Count = 23562

Resource @ nt!PopPolicyLock (0xfffff80001a3b9e0) Available
Contention Count = 19

Resource @ CI!g_StoreLock (0xfffff88000c05cc0) Available

Resource @ nt!CmpRegistryLock (0xfffff80001a10000) Available
Contention Count = 21297917

[...]
```

## Processors

This is a variant of **High Contention** pattern for processors where we have more threads at the same priority than the available processors. All these threads share the same notification event (or any other similar synchronization mechanism) and rush once it is signaled. If this happens often, the system becomes sluggish, or even appears frozen.

```
0: kd> !running

System Processors 3 (affinity mask)
Idle Processors 0

Prcbs Current Next
0 ffdfff120 89a92020      0.....
1 f7737120 89275020      W.....
0: kd> !ready
Processor 0: Ready Threads at priority 8
    THREAD 894a1db0 Cid 1a98.25c0 Teb: 7ffdde000 Win32Thread: bc19cea8 READY
    THREAD 897c4818 Cid 11d8.1c5c Teb: 7ffa2000 Win32Thread: bc2c5ba8 READY
    THREAD 8911fd18 Cid 2730.03f4 Teb: 7ffd9000 Win32Thread: bc305830 READY
Processor 1: Ready Threads at priority 8
    THREAD 89d89db0 Cid 1b10.20ac Teb: 7ffd7000 Win32Thread: bc16e680 READY
    THREAD 891f24a8 Cid 1e2c.20d0 Teb: 7ffda000 Win32Thread: bc1b9ea8 READY
    THREAD 89214db0 Cid 1e2c.24d4 Teb: 7ffd7000 Win32Thread: bc24ed48 READY
    THREAD 89a28020 Cid 1b10.21b4 Teb: 7ffa7000 Win32Thread: bc25b3b8 READY
    THREAD 891e03b0 Cid 1a98.05c4 Teb: 7ffdb000 Win32Thread: bc228bb0 READY
    THREAD 891b0020 Cid 1cd0.0144 Teb: 7ffdde000 Win32Thread: bc205ea8 READY
```

All these threads have the common stack trace (we show only a few threads here):

```
0: kd> !thread 89a92020 1f
THREAD 89a92020 Cid 11d8.27d8 Teb: 7ffd9000 Win32Thread: bc1e6860 RUNNING on processor 0
Not impersonating
DeviceMap          e502b248
Owning Process     89e2a020      Image:       ProcessA.exe
Attached Process   N/A          Image:       N/A
Wait Start TickCount 336581      Ticks: 0
Context Switch Count 61983        LargeStack
UserTime            00:00:00.156
KernelTime          00:00:00.281
Win32 Start Address ntdll!RtlpWorkerThread (0x7c839f2b)
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init f3730000 Current f372f7e0 Base f3730000 Limit f372c000 Call 0
Priority 8 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr
f3cc98e8 f6e21915 DriverA+0x1e4d
[...]
f3cc9ac0 f67f05dc nt!IoCallDriver+0x45
[...]
02e7ff44 7c83aa3b ntdll!RtlpWorkerCallout+0x71
02e7ff64 7c83aab2 ntdll!RtlpExecuteWorkerRequest+0x4f
```

```

0e7ff78 7c839f90 ntdll!RtlpApcCallout+0x11
0e7ffb8 77e6482f ntdll!RtlpWorkerThread+0x61
0e7ffec 00000000 kernel32!BaseThreadStart+0x34

0: kd> !thread 89275020 1f
THREAD 89275020 Cid 1cd0.2510 Teb: 7ffa9000 Win32Thread: bc343180 RUNNING on processor 1
Not impersonating
DeviceMap e1390978
Owning Process 89214708 Image: ProcessB.exe
Attached Process N/A Image: N/A
Wait Start TickCount 336581 Ticks: 0
Context Switch Count 183429 LargeStack
UserTime 00:00:00.171
KernelTime 00:00:00.484
Win32 Start Address ntdll!RtlpWorkerThread (0x7c839f2b)
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init b9f6e000 Current b9f6d7e0 Base b9f6e000 Limit b9f6a000 Call 0
Priority 8 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr
b9f6d87c f6e22d4b nt!KeWaitForSingleObject+0x497
b9f6d8e8 f6e21915 DriverA+0x1e4d
[...]
b9f6dac0 f67f05dc nt!IofCallDriver+0x45
[...]
0507ff44 7c83aa3b ntdll!RtlpWorkerCallout+0x71
0507ff64 7c83aab2 ntdll!RtlpExecuteWorkerRequest+0x4f
0507ff78 7c839f90 ntdll!RtlpApcCallout+0x11
0507ffb8 77e6482f ntdll!RtlpWorkerThread+0x61
0507ffec 00000000 kernel32!BaseThreadStart+0x34

0: kd> !thread 89d89db0 1f
THREAD 89d89db0 Cid 1b10.20ac Teb: 7ffd7000 Win32Thread: bc16e680 READY
Not impersonating
DeviceMap e4e3a0b8
Owning Process 898cb020 Image: ProcessC.exe
Attached Process N/A Image: N/A
Wait Start TickCount 336581 Ticks: 0
Context Switch Count 159844 LargeStack
UserTime 00:00:00.234
KernelTime 00:00:00.484
Win32 Start Address ntdll!RtlpWorkerThread (0x7c839f2b)
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init b9e1e000 Current b9e1d7e0 Base b9e1e000 Limit b9e1a000 Call 0
Priority 8 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr
b9e1d7f8 80831292 nt!KiSwapContext+0x26
b9e1d818 80828c73 nt!KiExitDispatcher+0xf8
b9e1d830 80829c72 nt!KiAdjustQuantumThread+0x109
b9e1d87c f6e22d4b nt!KeWaitForSingleObject+0x536
b9e1d8e8 f6e21915 DriverA+0x1e4d
[...]
b9e1dac0 f67f05dc nt!IofCallDriver+0x45
[...]
014dff44 7c83aa3b ntdll!RtlpWorkerCallout+0x71
014dff64 7c83aab2 ntdll!RtlpExecuteWorkerRequest+0x4f

```

```
014dff78 7c839f90 ntdll!RtlpApcCallout+0x11  
014dffb8 77e6482f ntdll!RtlpWorkerThread+0x61
```

These threads also share the same synchronization object:

```
0: kd> .thread 89275020  
Implicit thread is now 89275020

0: kd> kv 1  
ChildEBP RetAddr  Args to Child  
b9f6d87c f6e22d4b f6e25130 00000006 00000001 nt!KeWaitForSingleObject+0x497

0: kd> .thread 89d89db0  
Implicit thread is now 89d89db0

0: kd> kv 4  
ChildEBP RetAddr  Args to Child  
b9e1d7f8 80831292 f7737120 f7737b50 f7737a7c nt!KiSwapContext+0x26  
b9e1d818 80828c73 00000000 89d89db0 89d89e58 nt!KiExitDispatcher+0xf8  
b9e1d830 80829c72 f7737a7c 00000102 00000001 nt!KiAdjustQuantumThread+0x109  
b9e1d87c f6e22d4b f6e25130 00000006 00000001 nt!KeWaitForSingleObject+0x536

0: kd> dt _DISPATCHER_HEADER f6e25130  
ntdll!_DISPATCHER_HEADER  
+0x000 Type : 0 "  
+0x001 Absolute : 0 "  
+0x001 NpxIrql : 0 "  
+0x002 Size : 0x4 "  
+0x002 Hand : 0x4 "  
+0x003 Inserted : 0 "  
+0x003 DebugActive : 0 "  
+0x000 Lock : 262144  
+0x004 SignalState : 1  
+0x008 WaitListHead : _LIST_ENTRY [ 0xf6e25138 - 0xf6e25138 ]
```

## Historical Information

Although crash dumps are static in nature, they contain **Historical Information** about past system dynamics that might give clues to a problem and help with troubleshooting and debugging.

For example, IRP flow between user processes and drivers is readily available in any kernel or complete memory dump. WinDbg **!irpfnd** command will show the list of currently present I/O request packets. **!irp** command will give individual packet details.

Recent Driver Verifier improvements allow embedding stack traces associated with IRP allocation, completion, and cancellation<sup>94</sup>.

Other information that can be included in the process, kernel, and complete memory dumps may reveal some history of function calls beyond the current snapshot of thread stacks:

- Heap allocation stack traces that are usually used for debugging memory leaks (**Stack Trace Database**, page 919).
- Handle traces that are used to debug handle leaks (**!htrace** command).
- Raw stack data interpreted symbolically. Some examples include dumping stack. data from all process threads and dumping kernel mode stack data.
- LPC and ALPC messages (**!lpc thread** or **!alpc /lpp**).
- **Waiting Thread Time** pattern (page 1137).

## Comments

---

Unloaded module list (**!m**) is another example of **Historical Information** pattern.

Last error values for all threads make another good example of historical info (**Last Error Collection** pattern, page 596).

**!obtrace** command monitors more than **!htrace** command<sup>95</sup>.

Debugging TV Frames episode 0x29 shows an example for *notepad.exe*<sup>96</sup>.

---

<sup>94</sup> [http://msdn.microsoft.com/en-us/Library/windows/hardware/ff545448\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/Library/windows/hardware/ff545448(v=vs.85).aspx)

<sup>95</sup> Windows Internals, 5th edition, p. 156

<sup>96</sup> <http://www.debugging.tv>

## Hooked Functions

### Kernel Space

This is a variation of **Hooked Functions** pattern for kernel space. In addition to trampoline patching, we also see a modified service table:

```
0: kd> !chkimg -lo 50 -d !nt -v
Searching for module with expression: !nt
Will apply relocation fixups to file used for comparison
Will ignore NOP/LOCK errors
Will ignore patched instructions
Image specific ignores will be applied
Comparison image path: c:\symdownstream\ntkrnlmp.exe\4B7A8E6228000\ntkrnlmp.exe
No range specified

Scanning section: .text
Size: 625257
Range to scan: 80801000-80899a69
  808373e3-808373e9 7 bytes - nt!KeAcquireQueuedSpinLockAtDpcLevel+1b
  [ f7 41 04 01 00 00 00:e9 00 0d b2 76 cc cc ]
    8083e6c8-8083e6cb 4 bytes - nt!KiServiceTable+440 (+0x72e5)
  [ 98 4e 98 80:d0 66 e9 f4 ]
    80840605-8084060a 6 bytes - nt!KxFlushEntireTb+9 (+0x1f3d)
  [ ff 15 1c 10 80 80:e9 a5 7a b1 76 cc ]
Total bytes compared: 625257(100%)
Number of errors: 17

Scanning section: MISYSPT
Size: 1906
Range to scan: 8089a000-8089a772
Total bytes compared: 1906(100%)
Number of errors: 0

Scanning section: POOLMI
Size: 7868
Range to scan: 8089b000-8089cebc
Total bytes compared: 7868(100%)
Number of errors: 0

Scanning section: POOLCODE
Size: 7754
Range to scan: 8089d000-8089ee4a
Total bytes compared: 7754(100%)
Number of errors: 0

Scanning section: PAGE
Size: 1097281
Range to scan: 808bc000-809c7e41
Total bytes compared: 1097281(100%)
Number of errors: 0
```

```
Scanning section: PAGELK
Size: 63633
Range to scan: 809c8000-809d7891
Total bytes compared: 63633(100%)
Number of errors: 0
```

```
Scanning section: PAGEWMI
Size: 7095
Range to scan: 809ef000-809f0bb7
Total bytes compared: 7095(100%)
Number of errors: 0
```

```
Scanning section: PAGEKD
Size: 16760
Range to scan: 809f1000-809f5178
Total bytes compared: 16760(100%)
Number of errors: 0
```

```
Scanning section: PAGEHDLS
Size: 7508
Range to scan: 809f7000-809f8d54
Total bytes compared: 7508(100%)
Number of errors: 0
17 errors : !nt (808373e3-8084060a)
```

```
0: kd> dds 8083e6c8
8083e6c8 f4e966d0 DriverA+0x20d8
8083e6cc 80983436 nt!NtUnloadKey2
8083e6d0 809837b5 nt!NtUnloadKeyEx
8083e6d4 8091cec8 nt!NtUnlockFile
8083e6d8 80805d80 nt!NtLockVirtualMemory
8083e6dc 80937630 nt!NtUnmapViewOfSection
8083e6e0 808e7154 nt!NtVdmControl
8083e6e4 809c6ba3 nt!NtWaitForDebugEvent
8083e6e8 8092dc24 nt!NtWaitForMultipleObjects
8083e6ec 8092ccf4 nt!NtWaitForSingleObject
8083e6f0 809c132f nt!NtWaitHighEventPair
8083e6f4 809c12c3 nt!NtWaitLowEventPair
8083e6f8 80925c8d nt!NtWriteFile
8083e6fc 80901790 nt!NtWriteFileGather
8083e700 8091214c nt!NtWriteRequestData
8083e704 8093e63b nt!NtWriteVirtualMemory
8083e708 80822751 nt!NtYieldExecution
8083e70c 808c7c46 nt!NtCreateKeyedEvent
8083e710 8093eee3 nt!NtOpenKeyedEvent
8083e714 809c1ee8 nt!NtReleaseKeyedEvent
8083e718 809c2183 nt!NtWaitForKeyedEvent
8083e71c 809a610b nt!NtQueryPortInformationProcess
8083e720 809a6123 nt!NtGetCurrentProcessorNumber
8083e724 809a1849 nt!NtWaitForMultipleObjects32
8083e728 90909090
8083e72c 1c0d3b90
8083e730 0f8089f1
8083e734 037aaa85
8083e738 00c1f700
```

```
8083e73c 0fffff00  
8083e740 037a9e85  
8083e744 9090c300
```

```
0: kd> u 808373e3  
nt!KeAcquireQueuedSpinLockAtDpcLevel+0x1b:  
808373e3 jmp     DriverB+0x10e8 (f73580e8)  
808373e8 int     3  
808373e9 int     3  
808373ea je      nt!KeAcquireQueuedSpinLockAtDpcLevel+0x12 (808373da)  
808373ec pause  
808373ee jmp     nt!KeAcquireQueuedSpinLockAtDpcLevel+0x1b (808373e3)  
nt!KeReleaseInStackQueuedSpinLockFromDpcLevel:  
808373f0 lea     ecx,[ecx]  
nt!KeReleaseQueuedSpinLockFromDpcLevel:  
808373f2 mov     eax,ecx
```

```
0: kd> u 80840605  
nt!KxFFlushEntireTb+0x9:  
80840605 jmp     DriverB+0x10af (f73580af)  
8084060a int     3  
8084060b mov     byte ptr [ebp-1],al  
8084060e mov     ebx,offset nt!KiTbFlushTimeStamp (808a7100)  
80840613 mov     ecx,dword ptr [nt!KiTbFlushTimeStamp (808a7100)]  
80840619 test    cl,1  
8084061c jne     nt!KxFFlushEntireTb+0x19 (8082cd8d)  
80840622 mov     eax,ecx
```

## Comments

---

Another example:

```
4: kd> !chkimg -lo 50 -d !nt
8083351c-80833520 5 bytes - nt!NtYieldExecution
[ 8b ff 55 8b ec:e9 5c 03 e6 73 ]
808345d0-808345d3 4 bytes - nt!KiServiceTable+440 (+0x10b4)
[ 9c c2 8b 80:5c d7 f1 f4 ]
808eeb1e-808eeb22 5 bytes - nt!NtCreateFile
[ 8b ff 55 8b ec:e9 1c 4d da 73 ]
809233b0-809233b4 5 bytes - nt!NtUnmapViewOfSection (+0x34892)
[ 8b ff 55 8b ec:e9 f2 04 d7 73 ]
8092d3ae-8092d3b4 7 bytes - nt!NtMapViewOfSection (+0x9ffe)
[ 6a 38 68 b8 41 80 80:e9 de 64 d6 73 90 90 ]
80931c90-80931c96 7 bytes - nt!NtProtectVirtualMemory (+0x48e2)
[ 6a 44 68 d8 43 80 80:e9 be 1b d6 73 90 90 ]
8094af32-8094af36 5 bytes - nt!NtCreateProcess (+0x192a2)
[ 8b ff 55 8b ec:e9 32 89 d4 73 ]
8094c714-8094c718 5 bytes - nt!NtTerminateProcess (+0x17e2)
[ 8b ff 55 8b ec:e9 12 71 d4 73 ]
43 errors : !nt (8083351c-8094c718)

4: kd> u 8094af32
nt!NtCreateProcess:
*** ERROR: Symbol file could not be found. Defaulted to export symbols for 3rdPartyAVDriver.sys -
8094af32 e93289d473 jmp 3rdPartyAVDriver+0x13869 (f4693869)
8094af37 33c0 xor eax,eax
8094af39 f6451c01 test byte ptr [ebp+1Ch],1
8094af3d 7401 je nt!NtCreateProcess+0xe (8094af40)
8094af3f 40 inc eax
8094af40 f6452001 test byte ptr [ebp+20h],1
8094af44 7403 je nt!NtCreateProcess+0x17 (8094af49)
8094af46 83c802 or eax,2
```

The simplified version of the command:

```
!chkimg -db -v ModuleName
```

For example:

```
!chkimg -db -v nt
```

This allows to use **!for\_each\_module** command

To include the mismatch summary use this simplified version of the command:

```
!chkimg -db -d -v ModuleName
```

For example:

```
!chkimg -db -d -v ntdll
```

## User Space

Hooking functions using trampoline method is so common on Windows that sometimes we need to check **Hooked Functions** in specific modules and determine which module hooked them for troubleshooting or memory forensic analysis needs. If original unhooked modules are available (via symbol server, for example) this can be done by using **!chkimg** WinDbg extension command:

```
0:002> !chkimg -lo 50 -d !kernel32 -v
Searching for module with expression: !kernel32
Will apply relocation fixups to file used for comparison
Will ignore NOP/LOCK errors
Will ignore patched instructions
Image specific ignores will be applied
Comparison image path: c:\syndownstream\kernel32.dll\44C60F39102000\kernel32.dll
No range specified

Scanning section: .text
Size: 564445
Range to scan: 77e41000-77ecacdd
  77e44004-77e44008 5 bytes - kernel32!GetDateFormatA
[ 8b ff 55 8b ec:e9 f7 bf 08 c0 ]
  77e4412e-77e44132 5 bytes - kernel32!GetTimeFormatA (+0x12a)
[ 8b ff 55 8b ec:e9 cd be 06 c0 ]
  77e4e857-77e4e85b 5 bytes - kernel32!FileTimeToLocalFileTime (+0xa729)
[ 8b ff 55 8b ec:e9 a4 17 00 c0 ]
  77e56b5f-77e56b63 5 bytes - kernel32!GetTimeZoneInformation (+0x8308)
[ 8b ff 55 8b ec:e9 9c 94 00 c0 ]
  77e579a9-77e579ad 5 bytes - kernel32!GetTimeFormatW (+0xe4a)
[ 8b ff 55 8b ec:e9 52 86 06 c0 ]
  77e57fc8-77e57fcc 5 bytes - kernel32!GetDateFormatW (+0x61f)
[ 8b ff 55 8b ec:e9 33 80 08 c0 ]
  77e6f32b-77e6f32f 5 bytes - kernel32!GetLocalTime (+0x17363)
[ 8b ff 55 8b ec:e9 d0 0c 00 c0 ]
  77e6f891-77e6f895 5 bytes - kernel32!LocalFileTimeToFileTime (+0x566)
[ 8b ff 55 8b ec:e9 6a 07 01 c0 ]
  77e83499-77e8349d 5 bytes - kernel32!SetLocalTime (+0x13c08)
[ 8b ff 55 8b ec:e9 62 cb 00 c0 ]
  77e88c32-77e88c36 5 bytes - kernel32!SetTimeZoneInformation (+0x5799)
[ 8b ff 55 8b ec:e9 c9 73 01 c0 ]
Total bytes compared: 564445(100%)
Number of errors: 50
50 errors : !kernel32 (77e44004-77e88c36)

0:002> u 77e44004
kernel32!GetDateFormatA:
77e44004 e9f7bf08c0      jmp    37ed0000
77e44009 81ec18020000    sub    esp,218h
77e4400f a148d1ec77      mov    eax,dword ptr [kernel32!__security_cookie (77ecd148)]
77e44014 53              push   ebx
77e44015 8b5d14          mov    ebx,dword ptr [ebp+14h]
77e44018 56              push   esi
77e44019 8b7518          mov    esi,dword ptr [ebp+18h]
77e4401c 57              push   edi
```

```
0:002> u 37ed0000
*** ERROR: Symbol file could not be found. Defaulted to export symbols for MyDateTimeHooks.dll -
37ed0000 e99b262f2d      jmp     MyDateTimeHooks+0x26a0 (651c26a0)
37ed0005 8bff            mov     edi,edi
37ed0007 55              push    ebp
37ed0008 8bec            mov     ebp,esp
37ed000a e9fa3ff73f     jmp     kernel32!GetDateFormatA+0x5 (77e44009)
37ed000f 0000            add     byte ptr [eax],al
37ed0011 0000            add     byte ptr [eax],al
37ed0013 0000            add     byte ptr [eax],al
```

## Comments

---

The simplified version of the command:

```
!chkimg -db -v ModuleName
```

For example:

```
!chkimg -db -v ntdll
```

To include the mismatch summary use this version:

```
!chkimg -db -d -v ModuleName
```

For example:

```
!chkimg -db -d -v ntdll
```

Sometimes, several different modules from different products may patch different functions from the DLL. So, in general, we need to check all reported hooked functions.

## Hooked Modules

In **Hooked Functions** pattern (page 484) we used **!chkimg** WinDbg command. To check all modules, we can use this simple command:

```
!for_each_module !chkimg -lo 50 -d !${@#ModuleName} -v
```

For example:

```
0:000:x86> !for_each_module !chkimg -lo 50 -d !${@#ModuleName} -v
...
Scanning section: .text
Size: 74627
Range to scan: 71c01000-71c13383
    71c02430-71c02434 5 bytes - WS2_32!WSASend
[ 8b ff 55 8b ec:e9 cb db 1c 0d ]
    71c0279b-71c0279f 5 bytes - WS2_32!select (+0x36b)
[ 6a 14 68 58 28:e9 60 d8 15 0d ]
    71c0290e-71c02912 5 bytes - WS2_32!WSASendTo (+0x173)
[ 8b ff 55 8b ec:e9 ed d6 1b 0d ]
    71c02cb2-71c02cb6 5 bytes - WS2_32!closesocket (+0x3a4)
[ 8b ff 55 8b ec:e9 49 d3 19 0d ]
    71c02e12-71c02e16 5 bytes - WS2_32!WSAIoctl (+0x160)
[ 8b ff 55 8b ec:e9 e9 d1 1e 0d ]
    71c02ec2-71c02ec6 5 bytes - WS2_32!send (+0xb0)
[ 8b ff 55 8b ec:e9 39 d1 14 0d ]
    71c02f7f-71c02f83 5 bytes - WS2_32!recv (+0xbd)
[ 8b ff 55 8b ec:e9 7c d0 17 0d ]
    71c03c04-71c03c08 5 bytes - WS2_32!WSAGetOverlappedResult (+0xc85)
[ 8b ff 55 8b ec:e9 f7 c3 1f 0d ]
    71c03c75-71c03c79 5 bytes - WS2_32!recvfrom (+0x71)
[ 8b ff 55 8b ec:e9 86 c3 16 0d ]
    71c03d14-71c03d18 5 bytes - WS2_32!sendto (+0x9f)
[ 8b ff 55 8b ec:e9 e7 c2 13 0d ]
    71c03da8-71c03dac 5 bytes - WS2_32!WSACleanup (+0x94)
[ 8b ff 55 8b ec:e9 53 c2 25 0d ]
    71c03f38-71c03f3c 5 bytes - WS2_32!WSASocketW (+0x190)
[ 6a 20 68 08 40:e9 c3 c0 11 0d ]
    71c0446a-71c0446e 5 bytes - WS2_32!connect (+0x532)
[ 8b ff 55 8b ec:e9 91 bb 18 0d ]
    71c04f3b-71c04f3f 5 bytes - WS2_32!WSAStartup (+0xad1)
[ 6a 14 68 60 50:e9 c0 b0 29 0d ]
    71c06162-71c06166 5 bytes - WS2_32!shutdown (+0x1227)
[ 8b ff 55 8b ec:e9 99 9e 12 0d ]
    71c069e9-71c069ed 5 bytes - WS2_32!WSALookupServiceBeginW (+0x887)
[ 8b ff 55 8b ec:e9 12 96 0f 0d ]
    71c06c91-71c06c95 5 bytes - WS2_32!WSALookupServiceNextW (+0x2a8)
[ 8b ff 55 8b ec:e9 6a 93 10 0d ]
    71c06ecd-71c06ed1 5 bytes - WS2_32!WSALookupServiceEnd (+0x23c)
[ 8b ff 55 8b ec:e9 2e 91 0e 0d ]
    71c090be-71c090c2 5 bytes - WS2_32!WSAEEventSelect (+0x21f1)
[ 8b ff 55 8b ec:e9 3d 6f 20 0d ]
    71c09129-71c0912d 5 bytes - WS2_32!WSACreateEvent (+0x6b)
```

```
[ 33 c0 50 50 6a:e9 d2 6e 22 0d ]
 71c0938e-71c09392 5 bytes - WS2_32!WSACloseEvent (+0x265)
[ 6a 0c 68 c8 93:e9 6d 6c 24 0d ]
 71c093d9-71c093dd 5 bytes - WS2_32!WSAWaitForMultipleEvents (+0x4b)
[ 8b ff 55 8b ec:e9 22 6c 1a 0d ]
 71c093ea-71c093ee 5 bytes - WS2_32!WSAEnumNetworkEvents (+0x11)
[ 8b ff 55 8b ec:e9 11 6c 21 0d ]
 71c09480-71c09484 5 bytes - WS2_32!WSARecv (+0x96)
[ 8b ff 55 8b ec:e9 7b 6b 1d 0d ]
 71c0eecb-71c0eecf 5 bytes - WS2_32!WSACancelAsyncRequest (+0x5a4b)
[ 8b ff 55 8b ec:e9 30 11 26 0d ]
 71c10d39-71c10d3d 5 bytes - WS2_32!WSAAsyncSelect (+0x1e6e)
[ 8b ff 55 8b ec:e9 c2 f2 26 0d ]
 71c10ee3-71c10ee7 5 bytes - WS2_32!WSAConnect (+0x1aa)
[ 8b ff 55 8b ec:e9 18 f1 22 0d ]
 71c10f9f-71c10fa3 5 bytes - WS2_32!WSAAccept (+0xbc)
[ 8b ff 55 8b ec:e9 5c f0 27 0d ]
Total bytes compared: 74627(100%)
Number of errors: 140
140 errors : !WS2_32 (71c02430-71c10fa3)
...

```

## Comments

We can also use these variants:

```
!for_each_module !chkimg -db -d -v ${@#ModuleName}

!for_each_module !chkimg -db -d -v @#ModuleName
```

## Hooking Level

In addition to **Hooked Functions** (page 484) pattern, we should also pay attention a number of patched functions. Often value-added hookware<sup>97</sup> (see also **Hookware** patterns, page 1175) has configuration options that fine-tune hooking behavior. For example, an application with the less number of patched functions behaved incorrectly, and two process user dumps were saved from the working and non-working environment:

### Problem behavior

```
0:000> !chkimg -lo 50 -d !user32 -v
Searching for module with expression: !user32
Will apply relocation fixups to file used for comparison
Will ignore NOP/LOCK errors
Will ignore patched instructions
Image specific ignores will be applied
Comparison image path: c:\mss\user32.dll\49E0380E9d000\user32.dll
No range specified

Scanning section: .text
Size: 422527
Range to scan: 76e31000-76e9827f
76e3d6f8-76e3d6fc 5 bytes - user32!NtUserSetThreadDesktop
[ b8 30 12 00 00:e9 03 29 13 09 ]
76e3dc2a-76e3dc2e 5 bytes - user32!CreateWindowExA (+0x532)
[ 8b ff 55 8b ec:e9 d1 23 15 09 ]
76e3f8f8-76e3f8fc 5 bytes - user32!PostMessageA (+0x1cce)
[ 8b ff 55 8b ec:e9 03 07 fa 08 ]
76e41305-76e41309 5 bytes - user32!CreateWindowExW (+0x1a0d)
[ 8b ff 55 8b ec:e9 f6 ec 13 09 ]
76e435e3-76e435e7 5 bytes - user32!NtUserSetWindowPos (+0x22de)
[ b8 38 12 00 00:e9 18 ca 11 09 ]
76e48343-76e48347 5 bytes - user32!PeekMessageA (+0x4d60)
[ 8b ff 55 8b ec:e9 b8 7c fb 08 ]
76e48ab3-76e48ab7 5 bytes - user32!GetMessageA (+0x770)
[ 8b ff 55 8b ec:e9 48 75 fd 08 ]
76e4a175-76e4a179 5 bytes - user32!PostMessageW (+0x16c2)
[ 8b ff 55 8b ec:e9 86 5e f8 08 ]
76e4fef7-76e4fefb 5 bytes - user32!GetMessageW (+0x5d82)
[ 8b ff 55 8b ec:e9 04 01 fc 08 ]
76e5045a-76e5045e 5 bytes - user32!PeekMessageW (+0x563)
[ 8b ff 55 8b ec:e9 a1 fb f9 08 ]
76e8d37d-76e8d381 5 bytes - user32!MessageBoxTimeoutW (+0x3cf23)
[ 8b ff 55 8b ec:e9 7e 2c fd 08 ]
76e8d4d9-76e8d4dd 5 bytes - user32!MessageBoxIndirectA (+0x15c)
[ 8b ff 55 8b ec:e9 22 2b ff 08 ]
76e8d5d3-76e8d5d7 5 bytes - user32!MessageBoxIndirectW (+0xfa)
[ 8b ff 55 8b ec:e9 28 2a fe 08 ]
```

<sup>97</sup> Hookware, Memory Dump Analysis Anthology, Volume 2, page 63

```
76e8d65d-76e8d661 5 bytes - user32!MessageBoxExW (+0x8a)
[ 8b ff 55 8b ec:e9 9e 29 00 09 ]
Total bytes compared: 422527(100%)
Number of errors: 70
70 errors : !user32 (76e3d6f8-76e8d661)

0:000> u EnumDisplayDevicesW
user32!EnumDisplayDevicesW:
76e3ba5b 8bff      mov edi,edi
76e3ba5d 55        push ebp
76e3ba5e 8bec      mov ebp,esp
76e3ba60 81ec54030000 sub esp,354h
76e3ba66 a1c090e976 mov eax,dword ptr [user32!__security_cookie (76e990c0)]
76e3ba6b 33c5      xor eax,ebp
76e3ba6d 8945fc    mov dword ptr [ebp-4],eax
76e3ba70 53        push ebx
```

## Expected behavior

```
0:000> !chkimg -lo 50 -d !user32 -v
Searching for module with expression: !user32
Will apply relocation fixups to file used for comparison
Will ignore NOP/LOCK errors
Will ignore patched instructions
Image specific ignores will be applied
Comparison image path: c:\mss\user32.dll\49E0380E9d000\user32.dll
No range specified

Scanning section: .text
Size: 422527
Range to scan: 76e31000-76e9827f
76e39c11-76e39c15 5 bytes - user32!MonitorFromPoint
[ 6a 08 68 50 9c:e9 ea 63 10 09 ]
76e3b8ea-76e3b8ee 5 bytes - user32!GetMonitorInfoA (+0x1cd9)
[ 8b ff 55 8b ec:e9 11 47 12 09 ]
76e3ba5b-76e3ba5f 5 bytes - user32!EnumDisplayDevicesW (+0x171)
[ 8b ff 55 8b ec:e9 a0 45 0b 09 ]
76e3d6f8-76e3d6fa 3 bytes - user32!NtUserSetThreadDesktop (+0x1c9d)
[ b8 30 12:e9 03 29 ]
76e3d6fc - user32!NtUserSetThreadDesktop+4 (+0x04)
[ 00:09 ]
76e3dc2a-76e3dc2e 5 bytes - user32!CreateWindowExA (+0x52e)
[ 8b ff 55 8b ec:e9 d1 23 15 09 ]
76e3e7cd-76e3e7d1 5 bytes - user32!SetWindowLongA (+0xba3)
[ 8b ff 55 8b ec:e9 2e 18 03 09 ]
76e3f8f8-76e3f8fc 5 bytes - user32!PostMessageA (+0x112b)
[ 8b ff 55 8b ec:e9 03 07 e7 08 ]
76e41305-76e41309 5 bytes - user32!CreateWindowExW (+0x1a0d)
[ 8b ff 55 8b ec:e9 f6 ec 13 09 ]
76e413b4-76e413b8 5 bytes - user32!SetWindowLongW (+0xaf)
[ 8b ff 55 8b ec:e9 47 ec 03 09 ]
76e41709-76e4170d 5 bytes - user32!MonitorFromRect (+0x355)
[ 6a 08 68 48 17:e9 f2 e8 0e 09 ]
76e435e3-76e435e7 5 bytes - user32!NtUserSetWindowPos (+0x1eda)
[ b8 38 12 00 00:e9 18 ca fe 08 ]
```

```

76e440c5-76e440c9 5 bytes - user32!EnumDisplaySettingsExW (+0xae2)
[ 8b ff 55 8b ec:e9 36 bf 06 09 ]
76e441a1-76e441a5 5 bytes - user32!EnumDisplaySettingsW (+0xdc)
[ 8b ff 55 8b ec:e9 5a be 08 09 ]
76e46d4a-76e46d4e 5 bytes - user32!EnumDisplayDevicesA (+0x2ba9)
[ 8b ff 55 8b ec:e9 b1 92 0b 09 ]
76e46fe6-76e46fea 5 bytes - user32!EnumDisplaySettingsA (+0x29c)
[ 8b ff 55 8b ec:e9 15 90 09 09 ]
76e47010-76e47014 5 bytes - user32!EnumDisplaySettingsExA (+0x2a)
[ 8b ff 55 8b ec:e9 eb 8f 07 09 ]
76e47d12-76e47d16 5 bytes - user32!GetMonitorInfoW (+0xd02)
[ 8b ff 55 8b ec:e9 e9 82 10 09 ]
76e48343-76e48347 5 bytes - user32!PeekMessageA (+0x631)
[ 8b ff 55 8b ec:e9 b8 7c e8 08 ]
76e4844c-76e48450 5 bytes - user32!NtUserEnumDisplayMonitors (+0x109)
[ b8 81 11 00 00:e9 af 7b 0c 09 ]
76e488d4-76e488d8 5 bytes - user32!MonitorFromWindow (+0x488)
[ 6a 08 68 28 89:e9 27 77 0d 09 ]
76e48ab3-76e48ab7 5 bytes - user32!GetMessageA (+0x1df)
[ 8b ff 55 8b ec:e9 48 75 ea 08 ]
76e49994-76e49998 5 bytes - user32!GetWindowLongA (+0xee1)
[ 6a 08 68 d0 99:e9 67 66 00 09 ]
76e49af1-76e49af5 5 bytes - user32!GetSystemMetrics (+0x15d)
[ 6a 0c 68 58 9b:e9 0a 65 12 09 ]
76e4a175-76e4a179 5 bytes - user32!PostMessageW (+0x684)
[ 8b ff 55 8b ec:e9 86 5e e5 08 ]
76e4f8bf-76e4f8c3 5 bytes - user32!GetWindowLongW (+0x574a)
[ 6a 08 68 00 f9:e9 3c 07 01 09 ]
76e4fef7-76e4fefb 5 bytes - user32!GetMessageW (+0x638)
[ 8b ff 55 8b ec:e9 04 01 e9 08 ]
76e5045a-76e5045e 5 bytes - user32!PeekMessageW (+0x563)
[ 8b ff 55 8b ec:e9 a1 fb e6 08 ]
76e8d37d-76e8d381 5 bytes - user32!MessageBoxTimeoutW (+0x3cf23)
[ 8b ff 55 8b ec:e9 7e 2c ea 08 ]
76e8d4d9-76e8d4dd 5 bytes - user32!MessageBoxIndirectA (+0x15c)
[ 8b ff 55 8b ec:e9 22 2b ec 08 ]
76e8d5d3-76e8d5d7 5 bytes - user32!MessageBoxIndirectW (+0xfa)
[ 8b ff 55 8b ec:e9 28 2a eb 08 ]
76e8d65d-76e8d661 5 bytes - user32!MessageBoxExW (+0x8a)
[ 8b ff 55 8b ec:e9 9e 29 ed 08 ]
Total bytes compared: 422527(100%)
Number of errors: 154
154 errors : !user32 (76e39c11-76e8d661)

```

```

0:000> u EnumDisplayDevicesW
user32!EnumDisplayDevicesW:
76e3ba5b e9a0450b09      jmp 7fef0000
76e3ba60 81ec54030000    sub esp,354h
76e3ba66 a1c090e976      mov eax,dword ptr [user32!__security_cookie (76e990c0)]
76e3ba6b 33c5            xor eax,ebp
76e3ba6d 8945fc          mov dword ptr [ebp-4],eax
76e3ba70 53 push         ebx
76e3ba71 56 push         esi
76e3ba72 8b7510          mov esi,dword ptr [ebp+10h]

```

I

## Incomplete Stack Trace

### GDB

Users of WinDbg debugger accustomed to full thread stack traces will wonder whether a thread starts from *main*:

```
(gdb) where
#0 0x000000010d3b0e90 in bar () at main.c:15
#1 0x000000010d3b0ea9 in foo () at main.c:20
#2 0x000000010d3b0ec4 in main (argc=1,
argv=0x7fff6cfafbf8) at main.c:25
```

Of course, it doesn't, and a stack trace is shown starting from *main* function by default. We can change this behavior by using the following command:

```
(gdb) set backtrace past-main
```

Now we see the additional frame:

```
(gdb) where
#0 0x000000010d3b0e90 in bar () at main.c:15
#1 0x000000010d3b0ea9 in foo () at main.c:20
#2 0x000000010d3b0ec4 in main (argc=1,
argv=0x7fff6cfafbf8) at main.c:25
#3 0x000000010d3b0e74 in start ()
```

For Linux we have:

```
(gdb) set backtrace past-main
(gdb) bt
#0 0x000000000042fed1 in nanosleep ()
#1 0x000000000042fda0 in sleep ()
#2 0x000000000040078a in main ()
#3 0x0000000000405283 in __libc_start_main ()
#4 0x00000000004003e9 in _start ()
```

## Incomplete Session

It is a useful pattern for the analysis of memory dumps from terminal services environments. Normally, session processes include *csrss.exe*, *winlogon.exe*, *wfshell.exe* (in the case of some Citrix products), *explorer.exe* and a few user defined processes such as *winword.exe*, for example:

```
0: kd> !session
Sessions on machine: 6
Valid Sessions: 0 1 3 5 6 8

0: kd> !sprocess 6
Dumping Session 6

_MM_SESSION_SPACE ffffffa6009447000
_MMSESSION ffffffa6009447b40
PROCESS ffffffa800fce630
SessionId: 6 Cid: 1974 Peb: 7fffffd5000 ParentCid: 147c
DirBase: 158baf000 ObjectTable: fffff8801ef13b00 HandleCount: 532.
Image: csrss.exe

PROCESS ffffffa800fc77040
SessionId: 6 Cid: 1ae4 Peb: 7fffffd000 ParentCid: 147c
DirBase: 15d2b4000 ObjectTable: fffff8802084b570 HandleCount: 238.
Image: winlogon.exe

PROCESS ffffffa800fe61040
SessionId: 6 Cid: 1edc Peb: 7efdf000 ParentCid: 1ec8
DirBase: 14df74000 ObjectTable: fffff88020f486e0 HandleCount: 313.
Image: wfshell.exe

PROCESS ffffffa800ff5a660
SessionId: 6 Cid: 2054 Peb: 7fffffd000 ParentCid: 1dbc
DirBase: 201a81000 ObjectTable: fffff88020dd56e0 HandleCount: 447.
Image: explorer.exe

PROCESS ffffffa800fe28040
SessionId: 6 Cid: 1ce4 Peb: 7efdf000 ParentCid: 13a8
DirBase: 11f552000 ObjectTable: fffff8801fe96990 HandleCount: 1842.
Image: WINWORD.EXE

PROCESS ffffffa800f119c10
SessionId: 6 Cid: 2074 Peb: 7efdf000 ParentCid: 2054
DirBase: 2d994f000 ObjectTable: fffff8801e76aec0 HandleCount: 673.
Image: iexplore.exe
```

If we compare with the last session #8 we see that the latter has only two processes:

```
0: kd> !sprocess 8
Dumping Session 8

_MM_SESSION_SPACE ffffffa600bafc000
_MMSESSION ffffffa600bafcb40
PROCESS ffffffa80103a4480
SessionId: 8 Cid: 2858 Peb: 7fffffffdf000 ParentCid: 2660
DirBase: a04bb000 ObjectTable: fffff8801cb926a0 HandleCount: 534.
Image: csrss.exe

PROCESS ffffffa801065b770
SessionId: 8 Cid: 2878 Peb: 7fffffffdf000 ParentCid: 2660
DirBase: 5da40000 ObjectTable: fffff8801ce5e440 HandleCount: 235.
Image: winlogon.exe
```

Such anomalies may point to a disconnected session that failed to terminate due to some unresponsive session process, or a session that is stuck in session initialization process launch sequence due to threads blocked in wait chains. Here process threads need to be analyzed.

## Comments

Terminal session management systems may also preallocate sessions to make logon faster.

In the case of many terminal sessions on Windows we can dump processes sorted by session via **!sprocess -4** to spot **Incomplete Sessions** (page 496).

## Inconsistent Dump

We have to live with tools that produce inconsistent dumps. For example, LiveKd.exe from sysinternals.com which was widely used in the past by Microsoft and Citrix technical support to save complete memory dumps without a server reboot. Here we reproduce a note from one of the past articles<sup>98</sup>:

*LiveKd.exe-generated dumps are always inconsistent and cannot be a reliable source for certain types of dump analysis, for example, looking at resource contention. This is because it takes a considerable amount of time to save a dump on a live system, and the system is being changed during that process. The instantaneous traditional CrashOnCtrlScroll method or SystemDump tool always save a reliable and consistent dump because the system is frozen first (any process or kernel activity is disabled), then a dump is saved to a page file.*

If we look at such inconsistent dump, we will find that many useful kernel structures such as ERESOURCE list (**!locks**) are broken and even circular referenced, and therefore, WinDbg commands display “strange” output.

Easy and painless (for customers) crash dump generation using such “Live” tools means that it is widely used, and we have to analyze memory dumps saved by these tools and sent from customers. This brings us to the next crash dump analysis pattern called **Inconsistent Dump**.

If we have such a memory dump, we should look at it in order to extract maximum useful information that helps in identifying the root cause or give us further directions. Not all information is inconsistent in such dumps. For example, drivers, processes, thread stacks and IRP lists can give us some clues about activities. Even some information not visible in a consistent dump can surface in the inconsistent dump.

For example, we had a LiveKd memory dump from Windows Server 2003 where we looked at process stacks and found that for some processes in addition to their own threads there were additional terminated threads belonging to a completely different process (never seen in the consistent memory dumps).

## Comments

Newer versions of LiveKd pause a VM while saving a memory dump<sup>99</sup>. There are even more options added recently for consistency. Mirror dump example<sup>100</sup>. When using LiveKd for child Hyper-V partitions (-hv) we should use -p option to pause the partition. Please also see **Mirror Dump Set** analysis pattern (page 666).

---

<sup>98</sup> Inconsistent Dump, Memory Dump Analysis Anthology, Volume 1, page 269

<sup>99</sup> <http://technet.microsoft.com/en-ie/sysinternals/bb897415>

<sup>100</sup> <http://blogs.msdn.com/b/ntdebugging/archive/2016/01/22/virtual-machine-managment-hangs-on-windows-server-2012-r2-hyper-v-host.aspx>

## Incorrect Stack Trace

One of the mistakes beginners make is trusting WinDbg `!analyze` or `kv` commands displaying a stack trace. WinDbg is only a tool; sometimes information necessary to get correct stack trace is missing, and, therefore, some critical thought is required to distinguish between correct and incorrect stack traces. We call this pattern **Incorrect Stack Trace**. **Incorrect Stack Traces** usually

- Have a WinDbg warning: “*Following frames may be wrong.*”
- Don’t have the correct bottom frame like `kernel32!BaseThreadStart` (in user-mode)
- Have function calls that don’t make any sense
- Have strange looking disassembled function code or code that doesn’t make any sense from compiler perspective
- Have ChildEBP and RetAddr addresses that don’t make any sense

Consider the following **Stack Trace** (page 926):

```
0:011> k
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
0184e434 7c830b10 0x184e5bf
0184e51c 7c81f832 ntdll!RtlGetFullPathName_Ustr+0x15b
0184e5f8 7c83b1dd ntdll!RtlpLowFragHeapAlloc+0xc6a
00099d30 00000000 ntdll!RtlpLowFragHeapFree+0xa7
```

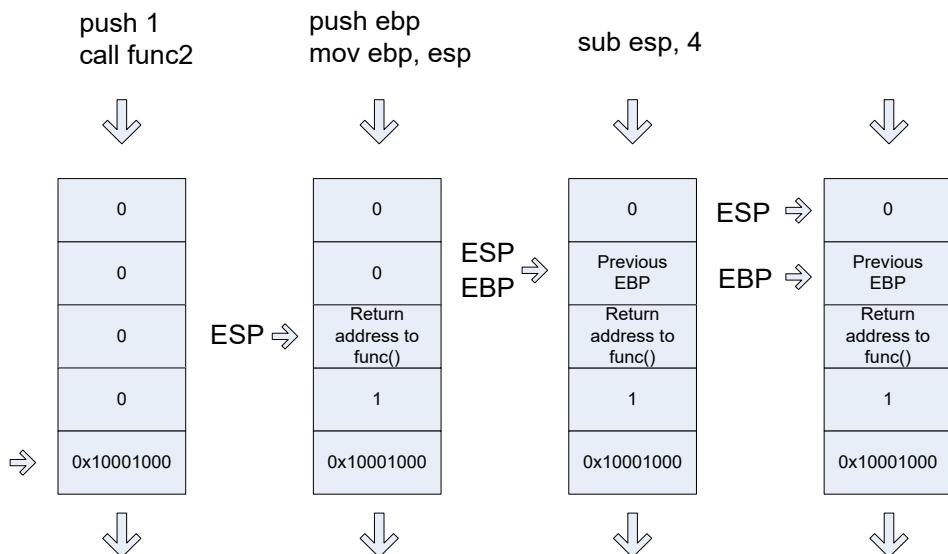
Here we have almost all attributes of a wrong stack trace. At first glance it looks like some heap corruption happened (runtime heap *alloc* and *free* functions are present) but if we give it a second thought we see that the low fragmentation heap *Free* function shouldn’t call low the fragmentation heap *Alloc* function and the latter shouldn’t query the full path name. That doesn’t make any sense.

What should we do here? Look at the raw stack and try to build the correct stack trace ourselves. In our case, this is very easy. We need to traverse stack frames from `BaseThreadStart+0x34` until we don’t find any function call or reach the top. When functions are called (no optimization, most compilers) EBP registers are linked together as explained on the following slide from Practical Foundations of Debugging seminars<sup>101</sup>:

---

<sup>101</sup> Windows Debugging: Practical Foundations, <http://www.dumpanalysis.org/windows-debugging-practical-foundations>

```
func() { func2(1); }    func2(int i) { int var; }
```



```
0:011> !teb
TEB at 7ffd8000
ExceptionList: 0184ebdc
StackBase: 01850000
StackLimit: 01841000
SubSystemTib: 00000000
FiberData: 00001e00
ArbitraryUserPointer: 00000000
Self: 7ffd8000
EnvironmentPointer: 00000000
ClientId: 0000061c . 00001b60
RpcHandle: 00000000
Tls Storage: 00000000
PEB Address: 7ffd0000
LastErrorValue: 0
LastStatusValue: c0000034
Count Owned Locks: 0
HardErrorMode: 0
```

```
0:011> dds 01841000 01850000
01841000 00000000
...
...
0184eef0 0184ef0c
0184eef4 7615dff2 localspl!SplDriverEvent+0x21
0184eef8 00bc3e08
0184eefc 00000003
0184ef00 00000001
0184ef04 00000000
0184ef08 0184efb0
0184ef0c 0184ef30
```

```
0184ef10 7615f9d0 localspl!PrinterDriverEvent+0x46
0184ef14 00bc3e08
0184ef18 00000003
0184ef1c 00000000
0184ef20 0184efb0
0184ef24 00b852a8
0184ef28 00c3ec58
0184ef2c 00bafcc0
0184ef30 0184f3f8
0184ef34 7614a9b4 localspl!SplAddPrinter+0x5f3
0184ef38 00c3ec58
0184ef3c 00000003
0184ef40 00000000
0184ef44 0184efb0
0184ef48 00c117f8
...
...
...
0184ff28 00000000
0184ff2c 00000000
0184ff30 0184ff84
0184ff34 77c75286 RPCRT4!LRPC_ADDRESS::ReceiveLotsaCalls+0x3a
0184ff38 0184ff4c
0184ff3c 77c75296 RPCRT4!LRPC_ADDRESS::ReceiveLotsaCalls+0x4a
0184ff40 7c82f2fc ntdll!RtlLeaveCriticalSection
0184ff44 000de378
0184ff48 00097df0
0184ff4c 4d2fa200
0184ff50 ffffffff
0184ff54 ca5b1700
0184ff58 ffffffff
0184ff5c 8082d821
0184ff60 0184fe38
0184ff64 00097df0
0184ff68 000000aa
0184ff6c 80020000
0184ff70 0184ff54
0184ff74 80020000
0184ff78 000b0c78
0184ff7c 00a50180
0184ff80 0184fe38
0184ff84 0184ff8c
0184ff88 77c5778f RPCRT4!RecvLotsaCallsWrapper+0xd
0184ff8c 0184ffac
0184ff90 77c5f7dd RPCRT4!BaseCachedThreadRoutine+0x9d
0184ff94 0009c410
0184ff98 00000000
0184ff9c 00000000
0184ffa0 00097df0
0184ffa4 00097df0
0184ffa8 00015f90
0184ffac 0184ffb8
0184ffb0 77c5de88 RPCRT4!ThreadStartRoutine+0x1b
0184ffb4 00088258
0184ffb8 0184ffec
0184ffb0 77e6608b kernel32!BaseThreadStart+0x34
```

```

0184ffc0 00097df0
0184ffc4 00000000
0184ffc8 00000000
0184ffcc 00097df0
0184ffd0 8ad84818
0184ffd4 0184ffc4
0184ffd8 8980a700
0184ffdcc fffffff
0184ffe0 77e6b7d0 kernel32!_except_handler3
0184ffe4 77e66098 kernel32!`string'+0x98
0184ffe8 00000000
0184ffec 00000000
0184fff0 00000000
77c5de6d RPCRT4!ThreadStartRoutine
0184fff8 00097df0
0184fffc 00000000
01850000 00000008

```

Next, we need to use custom **k** command and specify the base pointer. In our case, the last found stack address that links EBP pointers is 0184eef0:

```

0:011> k L=0184eef0
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
0184eef0 7615dff2 0x184e5bf
0184eef0c 7615f9d0 localspl!SplDriverEvent+0x21
0184ef30 7614a9b4 localspl!PrinterDriverEvent+0x46
0184f3f8 761482de localspl!SplAddPrinter+0x5f3
0184f424 74067c8f localspl!LocalAddPrinterEx+0x2e
0184f874 74067b76 SPOOLSS!AddPrinterExW+0x151
0184f890 01007e29 SPOOLSS!AddPrinterW+0x17
0184f8ac 01006ec3 spools!YAddPrinter+0x75
0184f8d0 77c70f3b spools!RpcAddPrinter+0x37
0184f8f8 77ce23f7 RPCRT4!Invoke+0x30
0184fcf8 77ce26ed RPCRT4!NdrStubCall2+0x299
0184fd14 77c709be RPCRT4!NdrServerCall2+0x19
0184fd48 77c7093f RPCRT4!DispatchToStubInCNoAvrf+0x38
0184fd9c 77c70865 RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x117
0184fdc0 77c734b1 RPCRT4!RPC_INTERFACE::DispatchToStub+0xa3
0184fdfc 77c71bb3 RPCRT4!LRPC_SCALL::DealWithRequestMessage+0x42c
0184fe20 77c75458 RPCRT4!LRPC_ADDRESS::DealWithLRPCRequest+0x127
0184ff84 77c5778f RPCRT4!LRPC_ADDRESS::ReceiveLotsaCalls+0x430
0184ff8c 77c5f7dd RPCRT4!RecvLotsaCallsWrapper+0xd

```

Stack traces make more sense now, but we don't see *BaseThreadStart+0x34*. By default, WinDbg displays only certain amount of function calls (stack frames), so we need to specify stack frame count, for example, 100:

```
0:011> k L=0184eef0 100
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
0184eef0 7615dff2 0x184e5bf
0184ef0c 7615f9d0 localspl!SplDriverEvent+0x21
0184ef30 7614a9b4 localspl!PrinterDriverEvent+0x46
0184f3f8 761482de localspl!SplAddPrinter+0x5f3
0184f424 74067c8f localspl!LocalAddPrinterEx+0x2e
0184f874 74067b76 SPOOLSS!AddPrinterExW+0x151
0184f890 01007e29 SPOOLSS!AddPrinterW+0x17
0184f8ac 01006ec3 spools!YAddPrinter+0x75
0184f8d0 77c70f3b spools!RpcAddPrinter+0x37
0184f8f8 77ce23f7 RPCRT4!Invoke+0x30
0184fcf8 77ce26ed RPCRT4!NdrStubCall2+0x299
0184fd14 77c709be RPCRT4!NdrServerCall2+0x19
0184fd48 77c7093f RPCRT4!DispatchToStubInCNoAvrf+0x38
0184fd9c 77c70865 RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x117
0184fdc0 77c734b1 RPCRT4!RPC_INTERFACE::DispatchToStub+0xa3
0184fdfe 77c71bb3 RPCRT4!LRPC_SCALL::DealWithRequestMessage+0x42c
0184fe20 77c75458 RPCRT4!LRPC_ADDRESS::DealWithLRPCRequest+0x127
0184ff84 77c5778f RPCRT4!LRPC_ADDRESS::ReceiveLotsaCalls+0x430
0184ff8c 77c5f7dd RPCRT4!RecvLotsaCallsWrapper+0xd
0184ffac 77c5de88 RPCRT4!BaseCachedThreadRoutine+0x9d
0184ffb8 77e6608b RPCRT4!ThreadStartRoutine+0x1b
0184ffec 00000000 kernel32!BaseThreadStart+0x34
```

Now our stack trace looks much better. For another complete example, please see the article<sup>102</sup>.

Sometimes **Incorrect Stack Trace** is reported when symbols were not applied. Non-symbol gaps in stack traces can be the sign of this pattern too:

```
STACK_TEXT:
WARNING: Stack unwind information not available. Following frames may be wrong.
00b2f42c 091607aa mydll!foo+0x8338
00b2f4cc 7c83ab9e mydll!foo+0x8fe3
00b2f4ec 7c832d06 ntdll!RtlFindNextActivationContextSection+0x46
00b2f538 001a5574 ntdll!RtlFindActivationContextSectionString+0xe1
00b2f554 7c8302b3 0x1a5574
00b2f560 7c82f9c1 ntdll!RtlpFreeToHeapLookaside+0x22
00b2f640 7c832b7f ntdll!RtlFreeHeap+0x20e
001dd000 00080040 ntdll!LdrUnlockLoaderLock+0xad
001dd00c 0052005c 0x80040
001dd010 00470045 0x52005c
0052005c 00000000 0x470045
```

<sup>102</sup> Manual Stack Trace Reconstruction, Memory Dump Analysis Anthology, Volume 1, page 157

## Comments

To check the correctness of some frames, we can use the same method as described in **Coincidental Symbolic Information** pattern (page 137). We use backward disassembly on a return address:

```
0286f430 690e6daa mshtml!CBase::PrivateInvokeEx+0x6d
WARNING: Stack unwind information not available. Following frames may be wrong.
0286f494 6915f5c5 jscript9!DllGetClassObject+0x18bb1

0:005> ub 690e6daa
jscript9!DllGetClassObject+0x18b9e:
690e6d97 ff7514 push dword ptr [ebp+14h]
690e6d9a ff7510 push dword ptr [ebp+10h]
690e6d9d 8b06    mov eax,dword ptr [esi]
690e6d9f 53      push ebx
690e6da0 ff75ec push dword ptr [ebp-14h]
690e6da3 ff7508 push dword ptr [ebp+8]
690e6da6 56      push esi
690e6da7 ff5020 call dword ptr [eax+20h]
```

## Incorrect Symbolic Information

Most of the time this pattern is associated with function names and offsets, for example, *module!foo* vs. *module!foo+100*. In some cases, the module name is incorrect itself or absent altogether. This can happen in complete memory dumps when we forget to reload user space symbols after changing the process context, for example:

```
; previous process context of firefox.exe
; switching to winlogon.exe context

kd> .process ffffffadfe718c040
Implicit process is now ffffffadf`e718c040

kd> !process ffffffadfe718c040
PROCESS ffffffadfe718c040
SessionId: 0 Cid: 017c Peb: 7fffffd9000 ParentCid: 0130
DirBase: 01916000 ObjectTable: ffffffa800099a890 HandleCount: 754.
Image: winlogon.exe
VadRoot ffffffadfe75e91f0 Vads 190 Clone 0 Private 2905. Modified 10047. Locked 0.
DeviceMap ffffffa8000004950
Token ffffffa800122a060
Elapsed Time 77 Days 02:14:26.109
UserTime 00:00:04.156
KernelTime 00:00:02.359
QuotaPoolUsage[PagedPool] 143128
QuotaPoolUsage[NonPagedPool] 191072
Working Set Sizes (now,min,max) (541, 50, 345) (2164KB, 200KB, 1380KB)
PeakWorkingSetSize 6323
VirtualSize 108 Mb
PeakVirtualSize 118 Mb
PageFaultCount 212547
MemoryPriority BACKGROUND
BasePriority 13
CommitCharge 3733

[...]
```

```

THREAD ffffffadfe68f2040 Cid 017c.0198 Peb: 000007fffffd7000 Win32Thread: ffffff97ff4a09010 WAIT:
(Unknown) UserMode Non-Alertable
    ffffffadfe7133160 Semaphore Limit 0x7fffffff
    ffffffadfe68f20f8 NotificationTimer
sNot impersonating
DeviceMap          ffffffa8000004950
Owning Process     ffffffadfe718c040      Image:      winlogon.exe
Attached Process   N/A           Image:      N/A
Wait Start TickCount 426298731      Ticks: 51 (0:00:00:00.796)
Context Switch Count 2215076           LargeStack
UserTime           00:00:00.187
KernelTime         00:00:00.468
Start Address 0x00000000077d6b6e0
Stack Init ffffffadfe4481e00 Current ffffffadfe4481860
Base ffffffadfe4482000 Limit ffffffadfe447a000 Call 0
Priority 14 BasePriority 13 PriorityDecrement 0
Child-SP          RetAddr        Call Site
fffffadf`e44818a0 ffffff800`0103b093 nt!KiSwapContext+0x85
fffffadf`e4481a20 ffffff800`0103c433 nt!KiSwapThread+0xc3
fffffadf`e4481a60 ffffff800`012d25ae nt!KeWaitForSingleObject+0x528
fffffadf`e4481af0 ffffff800`0104113d nt!NtReplyWaitReceivePortEx+0x8c8
fffffadf`e4481c00 00000000`77ef0caa nt!KiSystemServiceCopyEnd+0x3 (TrapFrame @ ffffffadf`e4481c70)
00000000`00bcfb98 000007ff`7fd6ff61 ntdll!NtReplyWaitReceivePortEx+0xa
00000000`00bcfbfa0 00000000`000d2340 0x7ff`7fd6ff61
00000000`00bcfbfa8 00000000`00bcfde0 0xd2340
00000000`00bcfb00 00000000`014cd220 0xbpcfde0
00000000`00bcfb08 00000000`000c1d30 0x14cd220
00000000`00bcfb0c 00000000`00bcfce18 0xc1d30
00000000`00bcfb08 0000ffff`00001f80 0xbcfce18
00000000`00bcfb00 00000000`006c0044 0xffff`00001f80
00000000`00bcfb08 00000000`000006ec firefox+0x2c0044
00000000`00bcfbe0 00000000`000007b0 0x6ec
00000000`00bcfbe8 00000000`419b8385 0x7b0
00000000`00bcfbf0 00000000`00000000 0x419b8385

kd> lmu m firefox
start          end            module name
00000000`00400000 00000000`00b67000  firefox  T (no symbols)

```

We have the return address 00000000`006c0044 that is just *firefox+0x2c0044* (00000000`00400000 + 2c0044). We correct it by reloading user space symbols.

```
kd> .reload /user
```

```

kd> !process ffffffadfe718c040
[...]
THREAD ffffffadfe68f2040 Cid 017c.0198 Peb: 000007fffffd7000 Win32Thread: fffff97ff4a09010 WAIT:
(Unknown) UserMode Non-Alertable
    ffffffadfe7133160 Semaphore Limit 0x7fffffff
    ffffffadfe68f20f8 NotificationTimer
Not impersonating
DeviceMap          ffffffa8000004950
Owning Process     ffffffadfe718c040      Image:      winlogon.exe
Attached Process   N/A           Image:      N/A
Wait Start TickCount 426298731      Ticks: 51 (0:00:00:00.796)
Context Switch Count 2215076           LargeStack
UserTime           00:00:00.187
KernelTime         00:00:00.468
Start Address kernel32!BaseThreadStart (0x0000000077d6b6e0)
Stack Init ffffffadfe4481e00 Current ffffffadfe4481860
Base ffffffadfe4482000 Limit ffffffadfe447a000 Call 0
Priority 14 BasePriority 13 PriorityDecrement 0
Child-SP          RetAddr        Call Site
fffffadf`e44818a0 ffffff800`0103b093 nt!KiSwapContext+0x85
fffffadf`e4481a20 ffffff800`0103c433 nt!KiSwapThread+0xc3
fffffadf`e4481a60 ffffff800`012d25ae nt!KeWaitForSingleObject+0x528
fffffadf`e4481af0 ffffff800`0104113d nt!NtReplyWaitReceivePortEx+0x8c8
fffffadf`e4481c00 00000000`77ef0caa nt!KiSystemServiceCopyEnd+0x3 (TrapFrame @ ffffffadf`e4481c70)
00000000`00bcfb98 000007ff`7fd6ff61 ntdll!NtReplyWaitReceivePortEx+0xa
00000000`00bcfba0 000007ff`7fd45369 RPCRT4!LRPC_ADDRESS::ReceiveLotsaCalls+0x2a5
00000000`00bcfeb0 000007ff`7fd65996 RPCRT4!RecvLotsaCallsWrapper+0x9
00000000`00bcfee0 000007ff`7fd65d51 RPCRT4!BaseCachedThreadRoutine+0xde
00000000`00bcff50 00000000`77d6b71a RPCRT4!ThreadStartRoutine+0x21
00000000`00bcff80 00000000`00000000 kernel32!BaseThreadStart+0x3a

```

Commands like **.process /r /p ffffffadfe718c040** or **!process ffffffadfe718c040 3f** do that automatically.

Another case for incorrect module names is malformed unloaded modules information:

```

0:000> lmt
start      end      module name
[...]
7c800000 7c907000  kernel32  Mon Apr 16 16:53:05 2007 (46239BE1)
7c910000 7c9c7000  ntdll    Wed Aug  4 08:57:08 2004 (411096D4)
7c9d0000 7d1ef000  shell32   Tue Dec 19 21:49:37 2006 (45885E71)
7df20000 7dfc0000  urlmon   Wed Aug 22 14:13:03 2007 (46CC365F)
7e360000 7e3f0000  user32   Thu Mar  8 15:36:30 2007 (45F02D7E)
Missing image name, possible paged-out or corrupt data.

```

```

Unloaded modules:
00410053 008a00a3 Unknown_Module_00410053
    Timestamp: Tue Mar 17 20:27:26 1970 (0064002E)
    Checksum: 006C006C
00010755 007407c5 1
    Timestamp: Wed Feb 04 21:26:01 1970 (002E0069)
    Checksum: 006C0064
00000011 411096d2 eme.dll
    Timestamp: Thu Apr 02 01:33:25 1970 (00780055)
    Checksum: 00680054
Missing image name, possible paged-out or corrupt data.
0064002e 00d0009a Unknown_Module_0064002e
    Timestamp: unavailable (00000000)
    Checksum: 00000000

```

Here parts of UNICODE module names appear in checksums and timestamps as well. Such partial module names can appear on thread stacks and raw stack data, for example:

```

0:000> kL
ChildEBP RetAddr
[...]
0015ef3c 0366afc2 ModuleA!Validation+0x5b
WARNING: Frame IP not in any known module. Following frames may be wrong.
0015efcc 79e7c7a6 <Unloaded_ure.dll>+0x366afc1
03dc9b70 00000000 mscorewks!MethodDesc::CallDescr+0x1f

```

Default analysis falls victim too and suggests *ure.dll* that you would try hard to find on your system:

```

MODULE_NAME: ure

IMAGE_NAME: ure.dll

DEBUG_FLR_IMAGE_TIMESTAMP: 750063

FAILURE_BUCKET_ID: ure.dll!Unloaded_c0000005_APPLICATION_FAULT

```

The timestamp is suspiciously UNICODE-like. In such cases, we can even reconstruct the module name:

```

00000011 411096d2 eme.dll
    Timestamp: Thu Apr 02 01:33:25 1970 (00780055)
    Checksum: 00680054

0:000> .formats 00780055
Evaluate expression:
    Hex: 00000000`00780055
    Decimal: 7864405
    Octal: 00000000000036000125
    Binary: 00000000 00000000 00000000 00000000 01111000 00000000 01010101
    Chars: .....x.U
    Time: Thu Apr 02 01:33:25 1970
    Float: low 1.10204e-038 high 0
    Double: 3.88553e-317

```

```
0:000> .formats 00680054
Evaluate expression:
Hex:      00680054
Decimal: 6815828
Octal:   00032000124
Binary:  00000000 01101000 00000000 01010100
Chars:   .h.T
Time:    Fri Mar 20 21:17:08 1970
Float:   low 9.55101e-039 high 0
Double:  3.36747e-317
```

We concatenate UNICODE *Ux* and *Th* with *eme.dll* to get *UxTheme.dll* which is a real DLL name we can find on a system.

## Injected Symbols

This pattern can be used to add missing symbols when we have **Reduced Symbolic Information** (page 829) like it was done previously in this old case study<sup>103</sup>. For example, TestWER<sup>104</sup> module was compiled with static MFC and CRT libraries and its private PDB file contains all necessary symbols including MSG structure. We can load that module into notepad.exe process space and apply symbols:

```
0:000:x86> lm
start          end            module name
00fc0000 00ff0000  notepad    (pdb
symbols)      c:\mss\ntepad.pdb\E325F5195AE94FAEB58D25C9DF8C0CFD2\ntepad.pdb
10000000 10039000  WinCRT     (deferred)
727f0000 7298e000  comctl32   (deferred)
72aa0000 72af1000  winspool   (deferred)
72b10000 72b19000  version    (deferred)
72e40000 72e48000  wow64cpu   (deferred)
72e50000 72eac000  wow64win   (pdb
symbols)      c:\mss\wow64win.pdb\B2D08CC152D64E71B79167DC0A0A53E91\wow64win.pdb
72eb0000 72eef000  wow64     (deferred)
733d0000 733e3000  dwmapi    (deferred)
735b0000 73606000  uxtheme   (deferred)
746f0000 746fc000  CRYPTBASE  (deferred)
74700000 74760000  sspicli   (deferred)
747c0000 74817000  shlwapi   (deferred)
74830000 7547a000  shell32   (deferred)
755d0000 7564b000  comdlg32  (deferred)
75650000 7567e000  imm32    (deferred)
75770000 75810000  advapi32  (deferred)
75810000 75920000  kernel32  (pdb
symbols)      c:\mss\kernel32.pdb\1C690A8592304467BB15A09CEA7180FA2\kernel32.pdb
75920000 759b0000  gdi32    (deferred)
759b0000 759f7000  KERNELBASE (deferred)
75a00000 75b00000  user32   (pdb
symbols)      c:\mss\user32.pdb\0FCE9CC301ED4567A819705B2718E1D62\user32.pdb
75b00000 75b8f000  oleaut32  (deferred)
[...]
76e40000 76fe9000  ntdll    (deferred)
77020000 771a0000  ntdll_77020000 (pdb
symbols)      c:\mss\ntdll.pdb\0D74F79EB1F8D4A45ABCD2F476CCABACC2\ntdll.pdb

0:000:x86> .sympath+ C:\DebuggingTV\TestWER\x86
Symbol search path is: srv*C:\DebuggingTV\TestWER\x86
Expanded Symbol search path is:
SRV*c:\mss*http://msdl.microsoft.com/download/symbols;c:\debuggingtv\testwer\x86
```

<sup>103</sup> Coping with Missing Symbolic Information, Memory Dump Analysis Anthology, Volume 1, page 199

<sup>104</sup> <http://support.citrix.com/article/CTX111901>

```
0:000:x86> .reload /f /i C:\DebuggingTV\TestWER\x86\TestWER.exe=10000000

0:000:x86> lm
start          end            module name
00fc0000 00ff0000  notepad    (pdb symbols)      c:\mss\notepad.pdb\325F5195AE94FAEB58D25C9DF8C0CFD2\notepad.pdb
10000000 10039000  TestWER   (private pdb symbols) c:\debuggingtv\testwer\x86\TestWER.pdb
727f0000 7298e000  comctl32  (deferred)
72aa0000 72af1000  winspool  (deferred)
72b10000 72b19000  version   (deferred)
72e40000 72e48000  wow64cpu (deferred)
72e50000 72eac000  wow64win (pdb symbols)      c:\mss\wow64win.pdb\B2D08CC152D64E71B79167DC0A0A53E91\wow64win.pdb
72eb0000 72eeff000 wow64     (deferred)
733d0000 733e3000  dwmapi   (deferred)
735b0000 73606000  uxtheme   (deferred)
746f0000 746fc000  CRYPTBASE (deferred)
74700000 74760000  sspicli  (deferred)
747c0000 74817000  shlwapi   (deferred)
74830000 7547a000  shell32  (deferred)
755d0000 7564b000  comdlg32 (deferred)
75650000 7567e000  imm32   (deferred)
75770000 75810000  advapi32 (deferred)
75810000 75920000  kernel32 (pdb symbols)      c:\mss\wkernel32.pdb\1C690A8592304467BB15A09CEA7180FA2\wkernel32.pdb
75920000 759b0000  gdi32   (deferred)
759b0000 759f7000  KERNELBASE (deferred)
75a00000 75b00000  user32   (pdb symbols)      c:\mss\wuser32.pdb\0FCE9CC301ED4567A819705B2718E1D62\wuser32.pdb
75b00000 75b8f000  oleaut32 (deferred)
[...]
76e40000 76fe9000  ntdll   (deferred)
77020000 771a0000  ntdll_77020000 (pdb symbols) c:\mss\wntdll.pdb\74F79EB1F8D4A45ABCD2F476CCABACC2\wntdll.pdb

0:000:x86> kv
ChildEBP RetAddr  Args to Child
0013fe34 75a1790d 0013fe74 00000000 00000000 user32!NtUserGetMessage+0x15
0013fe50 00fc148a 0013fe74 00000000 00000000 user32!GetMessageW+0x33
0013fe90 00fc16ec 00fc0000 00000000 00354082 notepad!WinMain+0xe6
0013ff20 758233aa 7efde000 0013ff6c 77059ef2 notepad!_initterm_e+0x1a1
0013ff2c 77059ef2 7efde000 00000000 kernel32!BaseThreadInitThunk+0xe
0013ff6c 77059ec5 00fc3689 7efde000 00000000 ntdll_77020000!_RtlUserThreadStart+0x70
0013ff84 00000000 00fc3689 7efde000 00000000 ntdll_77020000!_RtlUserThreadStart+0x1b

0:000:x86> dt -r MSG 0013fe74
TestWER!MSG
+0x000 hwnd           : 0x0007149c HWND__
+0x000 unused        : ???
+0x004 message       : 0x113
+0x008 wParam         : 0x38a508
+0x00c lParam         : 0n1921500630
+0x010 time           : 0x2079a177
+0x014 pt             : tagPOINT
+0x000 x              : 0n1337
+0x004 y              : 0n448
```

## Inline Function Optimization

### Managed Code

In addition to **Inline Function Optimization** (page 514) of unmanaged and native code we can see similar approach to JIT-compiled code:

```
public class ClassMain
{
    public bool time2stop = false;

    public static void Main(string[] args)
    {
        new ClassMain().Main();
    }

    public void Main()
    {
        while (!time2stop)
        {
            DoWork();
        }
    }

    volatile int inSensor, outSensor;

    void DoWork()
    {
        outSensor ^= inSensor;
    }
}

0:000> kL
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
001fef0a0 79e7c6cc 0x3200a4
001ff020 79e7c8e1 mscorewks!CallDescrWorkerWithHandler+0xa3
001ff160 79e7c783 mscorewks!MethodDesc::CallDescr+0x19c
001ff17c 79e7c90d mscorewks!MethodDesc::CallTargetWorker+0x1f
001ff190 79eefb9e mscorewks!MethodDescCallSite::Call_RetArgSlot+0x18
001ff2f4 79eef830 mscorewks!ClassLoader::RunMain+0x263
001ff55c 79ef01da mscorewks!Assembly::ExecuteMainMethod+0xa6
001ffa2c 79fb9793 mscorewks!SystemDomain::ExecuteMainMethod+0x43f
001ffa7c 79fb96df mscorewks!ExecuteEXE+0x59
001ffac4 736455ab mscorewks!_CorExeMain+0x15c
001ffad0 73747f16 mscoreei!_CorExeMain+0x38
001ffae0 73744de3 mscoreei!ShellShim__CorExeMain+0x99
001ffae8 76573833 mscoreei!_CorExeMain_Exported+0x8
001ffaf4 77c1a9bd kernel32!BaseThreadInitThunk+0xe
001ffb34 00000000 ntdll!_RtlUserThreadStart+0x23
```

```
0:000> r
eax=00000000 ebx=001fefbc ecx=015316e0 edx=0037a238 esi=0037a238 edi=00000000
eip=003200a4 esp=001fef90 ebp=001fef90 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
003200a4 80790c00 cmp byte ptr [ecx+0Ch],0 ds:0023:015316ec=00

0:000> !IP2MD 003200a4
MethodDesc:      000d3048
Method Name:    ClassMain.Main()
Class:           000d1180
MethodTable:     000d3060
mdToken:         06000002
Module:          000d2c3c
IsJitted:        yes
m_CodeOrIL:      00320098

0:000> .asm no_code_bytes
Assembly options: no_code_bytes

0:000> !U 003200a4
Normal JIT generated code
ClassMain.Main()
Begin 00320098, size 13
00320098 cmp byte ptr [ecx+0Ch],0
0032009c jne 003200aa
0032009e mov eax,dword ptr [ecx+4]
003200a1 xor dword ptr [ecx+8],eax
>>> 003200a4 cmp byte ptr [ecx+0Ch],0
003200a8 je 0032009e
003200aa ret
```

We see that *DoWork* code was inlined into *Main* function code.

## Unmanaged Code

Sometimes compilers optimize code by replacing function calls with their bodies. This procedure is called function inlining and functions themselves are called inline. On one platform, we can see the real function call on the stack trace but on another platform or product version we only see the same problem instruction. Fortunately, the rest of stack trace should be the same. Therefore when comparing **Stack Traces** (page 926), we should not pay attention only to the top function call.

This pattern is frequently seen when threads crash while copying or moving memory. Consider this stack trace:

```
0: kd> kL
ChildEBP RetAddr
f22efaf4 f279ec3d driver!QueueValue+0x26b
f22efb30 8081dcdf driver!BufferAppendData+0x35f
f22efc7c 808f47b7 nt!IoCallDriver+0x45
f22efc90 808f24ee nt!IoSyncrhousServiceTail+0x10b
f22efd38 80888c7c nt!NtWriteFile+0x65a
f22efd38 7c82ed54 nt!KiFastCallEntry+0xfc
```

When looking at *rep movs* instruction we might suspect that *QueueValue* was copying memory:

```
0: kd> r
eax=00000640 ebx=89b23000 ecx=00000190 edx=89b3c828 esi=02124220 edi=e2108f58
eip=f279c797 esp=f22efadc ebp=f22efaf4 iopl=0 nv up ei pl nz na pe nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010206
driver!QueueValue+0x26b:
f279c797 f3a5 rep movs dword ptr es:[edi],dword ptr [esi] es:0023:e2108f58=dfefbecf
ds:0023:02124220=????????
```

On x64 platform, the same driver had the similar stack trace but with *memcpy* at its top:

```
fffffadf`8955f4a8 fffffadf`8d1bef46 driver!memcpy+0x1c0
fffffadf`8955f4b0 fffffadf`8d1c15c9 driver!QueueValue+0x2fe
fffffadf`8955f550 fffff800`01273ed9 driver!BufferAppendData+0x481
...
```

We also see how *QueueValue+0x2fe* and *QueueValue+0x26b* are close. In fact, the source code for the driver calls *RtlCopyMemory* function only once, and it is defined as *memcpy* in *wdm.h*. The latter function is also exported from *nt* module:

```
0: kd> x nt!
...
80881780 nt!memcpy = <no type information>
...
```

but usually can be found in any driver that links with C runtime library, for example, on x64 Windows:

```
1: kd> x nt!memcpy
fffff800`01c464e0 nt!memcpy = <no type information>
```

```
1: kd> x srv!memcpy  
fffff980`0eafdf20 srv!memcpy = <no type information>  
  
1: kd> x win32k!memcpy  
fffff960`000c1b40 win32k!memcpy = <no type information>
```

Therefore, we see that when compiling for x86 platform Visual C++ compiler decided to inline *memcpy* code but AMD compiler on the x64 platform did not inline it. The overall stack trace without offsets is very similar, and we can suppose that the problem was identical.

## Instrumentation Information

Application and Driver Verifiers (including *gflags.exe* tool from Debugging Tools for Windows) set flags that modify the behavior of the system that is reflected in additional information being collected such as memory allocation history and in WinDbg output changes such as stack traces. These tools belong to a broad class of instrumentation tools. To check in a minidump, kernel, and complete memory dumps whether Driver Verifier was enabled we use **!Verifier** WinDbg command:

```
1: kd> !Verifier

Verify Level 0 ... enabled options are:

Summary of All Verifier Statistics

RaiseIrqls          0x0
AcquireSpinLocks    0x0
Synch Executions    0x0
Trims               0x0

Pool Allocations Attempted   0x0
Pool Allocations Succeeded   0x0
Pool Allocations Succeeded SpecialPool 0x0
Pool Allocations With NO TAG 0x0
Pool Allocations Failed      0x0
Resource Allocations Failed Deliberately 0x0

Current paged pool allocations 0x0 for 00000000 bytes
Peak paged pool allocations   0x0 for 00000000 bytes
Current nonpaged pool allocations 0x0 for 00000000 bytes
Peak nonpaged pool allocations 0x0 for 00000000 bytes

0: kd> !Verifier

Verify Level 3 ... enabled options are:
  Special pool
  Special irql

Summary of All Verifier Statistics

RaiseIrqls          0xdea5
AcquireSpinLocks    0x87b5c
Synch Executions    0x17b5
Trims               0xab36

Pool Allocations Attempted   0x8990e
Pool Allocations Succeeded   0x8990e
Pool Allocations Succeeded SpecialPool 0x29c0
Pool Allocations With NO TAG 0x1
Pool Allocations Failed      0x0
Resource Allocations Failed Deliberately 0x0
```

Current paged pool allocations	0x0 for 00000000 bytes
Peak paged pool allocations	0x0 for 00000000 bytes
Current nonpaged pool allocations	0x0 for 00000000 bytes
Peak nonpaged pool allocations	0x0 for 00000000 bytes

To check in a process user dump that Application Verifier (and gflags) was enabled we use **!avr** and **!gflags** WinDbg extension commands:

```
0:001> !avr
Application verifier is not enabled for this process.
Page heap has been enabled separately.

0:001> !gflag
Current NtGlobalFlag contents: 0x02000000
    hpa - Place heap allocations at ends of pages
```

Here is an example of an instrumented stack trace:

```
68546e88 verifier!AVrfpDphFindBusyMemoryNoCheck+0xb8
68546f95 verifier!AVrfpDphFindBusyMemory+0x15
68547240 verifier!AVrfpDphFindBusyMemoryAndRemoveFromBusyList+0x20
68549080 verifier!AVrfDebugPageHeapFree+0x90
77190aac ntdll!RtlDebugFreeHeap+0x2f
7714a8ff ntdll!RtlpFreeHeap+0x5d
770f2a32 ntdll!RtlFreeHeap+0x142
75fb14d1 kernel32!HeapFree+0x14
748d4c39 msocr80!free+0xcd
[...]
00a02bb2 ServiceA!ServiceMain+0x302
767175a8 sechost!ScSvccctrlThreadA+0x21
75fb3677 kernel32!BaseThreadInitThunk+0xe
770f9d42 ntdll!__RtlUserThreadStart+0x70
770f9d15 ntdll!_RtlUserThreadStart+0x1b
```

Here is another example that shows instrumentation difference. We run **Double Free** (page 267) fault modeling application and see its stack trace from a crash dump:

```
0:000> !gflag
Current NtGlobalFlag contents: 0x00000000
```

```
0:000> kL 100
Child-SP          RetAddr          Call Site
00000000`002dec38 00000000`77735ce2 ntdll!NtWaitForSingleObject+0xa
00000000`002dec40 00000000`77735e85 ntdll!RtlReportExceptionEx+0x1d2
00000000`002ded30 00000000`77735eea ntdll!RtlReportException+0xb5
00000000`002dedb0 00000000`77736d25 ntdll!RtlpTerminateFailureFilter+0x1a
00000000`002dede0 00000000`77685148 ntdll!RtlReportCriticalFailure+0x96
00000000`002dee10 00000000`776a554d ntdll!_C_specific_handler+0x8c
00000000`002dee80 00000000`77685d1c ntdll!RtlpExecuteHandlerForException+0xd
00000000`002deeb0 00000000`776862ee ntdll!RtlDispatchException+0x3cb
00000000`002df590 00000000`77736cd2 ntdll!RtlRaiseException+0x221
00000000`002dfbd0 00000000`77737396 ntdll!RtlReportCriticalFailure+0x62
00000000`002dfca0 00000000`777386c2 ntdll!RtlpReportHeapFailure+0x26
00000000`002dfcd0 00000000`7773a0c4 ntdll!RtlpHeapHandleError+0x12
00000000`002dfd00 00000000`776dd1cd ntdll!RtlpLogHeapFailure+0xa4
00000000`002dfd30 00000000`77472c7a ntdll! ?? ::FNODOBFM::`string'+0x123b4
00000000`002dfdb0 00000000`6243c7bc kernel32!HeapFree+0xa
00000000`002dfde0 00000001`3f8f1033 msvr90!free+0x1c
00000000`002dfe10 00000001`3f8f11f2 InstrumentedApp!wmain+0x33
00000000`002dfe50 00000000`7746f56d InstrumentedApp!__tmainCRTStartup+0x11a
00000000`002dfe80 00000000`776a3281 kernel32!BaseThreadInitThunk+0xd
00000000`002dfeb0 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

Then we enable Application Verifier and full page heap in *gflags.exe* GUI. Actually two crash dumps are saved at the same time (we had set up *LocalDumps* registry key<sup>105</sup> on x64 W2K8 R2) with slightly different stack traces:

```
0:000> !gflag
Current NtGlobalFlag contents: 0x02000100
  vrf - Enable application verifier
  hpa - Place heap allocations at ends of pages

0:000> kL 100
Child-SP          RetAddr          Call Site
00000000`0022e438 00000000`77735ce2 ntdll!NtWaitForSingleObject+0xa
00000000`0022e440 00000000`77735e85 ntdll!RtlReportExceptionEx+0x1d2
00000000`0022e530 000007fe`f3ed26fb ntdll!RtlReportException+0xb5
00000000`0022e5b0 00000000`77688a8f verifier!AVrfpVectoredExceptionHandler+0x26b
00000000`0022e640 00000000`776859b2 ntdll!RtlpCallVectoredHandlers+0xa8
00000000`0022e6b0 00000000`776bfe48 ntdll!RtlDispatchException+0x22
00000000`0022ed90 000007fe`f3eca668 ntdll!KiUserExceptionDispatcher+0x2e
00000000`0022f350 000007fe`f3ec931d verifier!VerifierStopMessage+0x1f0
00000000`0022f400 000007fe`f3ec9736 verifier!AVrfpDphReportCorruptedBlock+0x155
00000000`0022f4c0 000007fe`f3ec99cd verifier!AVrfpDphCheckNormalHeapBlock+0xce
00000000`0022f530 000007fe`f3ec873a verifier!AVrfpDphNormalHeapFree+0x29
00000000`0022f560 00000000`7773c415 verifier!AVrfDebugPageHeapFree+0xb6
00000000`0022f5c0 00000000`776dd0fe ntdll!RtlDebugFreeHeap+0x35
00000000`0022f620 00000000`776c2075 ntdll! ?? ::FNODOBFM::`string'+0x122e2
00000000`0022f960 000007fe`f3edf4e1 ntdll!RtlFreeHeap+0x1a2
```

<sup>105</sup> Local Crash Dumps in Vista, Memory Dump Analysis Anthology, Volume 1, page 606

```

00000000`0022f9e0 00000000`77472c7a verifier!AVrfpRtlFreeHeap+0xa5
00000000`0022fa80 000007fe`f3ee09ae kernel32!HeapFree+0xa
00000000`0022fab0 00000000`642bc7bc verifier!AVrfpHeapFree+0xc6
00000000`0022fb40 00000001`3fac1033 msrvr90!free+0x1c
00000000`0022fb70 00000001`3fac11f2 InstrumentedApp!wmain+0x33
00000000`0022fbbo 00000000`7746f56d InstrumentedApp!__tmainCRTStartup+0x11a
00000000`0022fbe0 00000000`776a3281 kernel32!BaseThreadInitThunk+0xd
00000000`0022fc10 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

0:000> kL 100
Child-SP      RetAddr          Call Site
00000000`0022e198 000007fe`f3ee0f82 ntdll!NtWaitForMultipleObjects+0xa
00000000`0022e1a0 000007fe`fd8513a6 verifier!AVrfpNtWaitForMultipleObjects+0x4e
00000000`0022e1e0 000007fe`f3ee0e2d KERNELBASE!WaitForMultipleObjectsEx+0xe8
00000000`0022e2e0 000007fe`f3ee0edd verifier!AVrfpWaitForMultipleObjectsExCommon+0xad
00000000`0022e320 00000000`77473143 verifier!AVrfpKernelbaseWaitForMultipleObjectsEx+0x2d
00000000`0022e370 00000000`774e9025 kernel32!WaitForMultipleObjectsExImplementation+0xb3
00000000`0022e400 00000000`774e91a7 kernel32!WerFaultInternal+0x215
00000000`0022e4a0 00000000`774e91ff kernel32!WerReportFault+0x77
00000000`0022e4d0 00000000`774e941c kernel32!BaseReportFault+0x1f
00000000`0022e500 00000000`7770573c kernel32!UnhandledExceptionFilter+0x1fc
00000000`0022e5e0 00000000`77685148 ntdll! ?? ::FNODOBFM::`string'+0x2365
00000000`0022e610 00000000`776a554d ntdll!_C_specific_handler+0x8c
00000000`0022e680 00000000`77685d1c ntdll!RtlpExecuteHandlerForException+0xd
00000000`0022e6b0 00000000`776bfe48 ntdll!RtlDispatchException+0x3cb
00000000`0022ed90 000007fe`f3eca668 ntdll!KiUserExceptionDispatcher+0x2e
00000000`0022f350 000007fe`f3ec931d verifier!VerifierStopMessage+0x1f0
00000000`0022f400 000007fe`f3ec9736 verifier!AVrfpDphReportCorruptedBlock+0x155
00000000`0022f4c0 000007fe`f3ec99cd verifier!AVrfpDphCheckNormalHeapBlock+0xce
00000000`0022f530 000007fe`f3ec873a verifier!AVrfpDphNormalHeapFree+0x29
00000000`0022f560 00000000`7773c415 verifier!AVrfDebugPageHeapFree+0xb6
00000000`0022f5c0 00000000`776dd0fe ntdll!RtlDebugFreeHeap+0x35
00000000`0022f620 00000000`776c2075 ntdll! ?? ::FNODOBFM::`string'+0x122e2
00000000`0022f960 000007fe`f3edf4e1 ntdll!RtlFreeHeap+0x1a2
00000000`0022f9e0 00000000`77472c7a verifier!AVrfpRtlFreeHeap+0xa5
00000000`0022fa80 000007fe`f3ee09ae kernel32!HeapFree+0xa
00000000`0022fab0 00000000`642bc7bc verifier!AVrfpHeapFree+0xc6
00000000`0022fb40 00000001`3fac1033 msrvr90!free+0x1c
00000000`0022fb70 00000001`3fac11f2 InstrumentedApp!wmain+0x33
00000000`0022fbbo 00000000`7746f56d InstrumentedApp!__tmainCRTStartup+0x11a
00000000`0022fbe0 00000000`776a3281 kernel32!BaseThreadInitThunk+0xd
00000000`0022fc10 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

```

We also see above that enabling instrumentation triggers debug functions of runtime heap (*RtlDebugFreeHeap*).

## Instrumentation Side Effect

Sometimes added instrumentation via gflags, Application and Driver Verifier options affect system, service or application performance and resources. For example, after enabling full page heap, one process on an x64 machine was growing up to 24GB, and its user memory dump shows that every heap allocation was recorded in a stack trace database:

```
0:055> !gflag
Current NtGlobalFlag contents: 0x02000000
hpa - Place heap allocations at ends of pages

0:055> ~*kc

[...]

48 Id: 117fc.c164 Suspend: 1 Teb: 000007ff`ffff52000 Unfrozen
Call Site
ntdll!ZwWaitForSingleObject
ntdll!RtlpWaitOnCriticalSection
ntdll!RtlEnterCriticalSection
Verifier!AVrfpDphEnterCriticalSection
Verifier!AVrfpDphPreProcessing
Verifier!AVrfDebugPageHeapAllocate
ntdll!RtlDebugAllocateHeap
ntdll! ?? ::FNODOBFM::`string'
ntdll!RtlAllocateHeap
msvcrt!malloc
ModuleA!foo1
[...]

49 Id: 117fc.de80 Suspend: 1 Teb: 000007ff`ffff54000 Unfrozen
Call Site
ntdll!RtlCompareMemory
ntdll!RtlpLogCapturedStackTrace
ntdll!RtlLogStackTrace
Verifier!AVrfpDphPlaceOnFreeList
Verifier!AVrfDebugPageHeapFree
ntdll!RtlDebugFreeHeap
ntdll! ?? ::FNODOBFM::`string'
ntdll!RtlFreeHeap
kernel32!HeapFree
msvcrt!free
ModuleB!foo2
[...]
```

```

50 Id: 117fc.3700 Suspend: 1 Teb: 000007ff`fff4e000 Unfrozen
Call Site
ntdll!ZwWaitForSingleObject
ntdll!RtlpWaitOnCriticalSection
ntdll!RtlEnterCriticalSection
Verifier!AVrfpDphEnterCriticalSection
Verifier!AVrfpDphPreProcessing
Verifier!AVrfpDebugPageHeapFree
ntdll!RtlDebugFreeHeap
ntdll! ?? ::FNODOBFM::`string'
ntdll!RtlFreeHeap
kernel32!HeapFree
msvcrt!free
ModuleC!foo3
[...]

0:055> !runaway
User Mode Time
Thread Time
38:d090      0 days 0:02:28.793
44:ca48      0 days 0:01:04.459
48:c164      0 days 0:00:56.909
43:4458      0 days 0:00:54.475
50:3700      0 days 0:00:43.992
45:6f98      0 days 0:00:38.953
49:de80      0 days 0:00:24.211
1:391c      0 days 0:00:00.639
0:7e90      0 days 0:00:00.109
55:a300      0 days 0:00:00.046
34:10c9c     0 days 0:00:00.015
21:d054      0 days 0:00:00.015
56:b0a0      0 days 0:00:00.000
54:8b78      0 days 0:00:00.000
53:155b8     0 days 0:00:00.000
52:b444      0 days 0:00:00.000

```

**Top Modules** (page 1012) ModuleA(B, C) from **Spiking** (page 888) and heap intensive threads are from **Same Vendor** (page 842).

We were able to get a 200×27349 slice from that dump using ImageMagick<sup>106</sup>, and it shows almost all virtual memory space filled with traces of this pictorial form (magnified by x8):

---

<sup>106</sup> <http://www.imagemagick.org/>



## Comments

Another side-effect from object tracking:

```
kd> !poolused 3
Sorting by NonPaged Pool Consumed

Pool Used:
NonPaged Paged
Tag    Allocs   Frees   Diff Used      Allocs   Frees   Diff Used
0bRt  4384299 4383686 615  54256308 0       0     0     0   object reference stack tracing , Binary: nt!ob
```

## Insufficient Memory

### Committed Memory

**Insufficient Memory** pattern can be seen in many complete and kernel memory dumps. This condition can cause a system to crash, become slow, hang or refuse to provide the expected functionality, for example, refuse new terminal server connections. There are many types of memory resources, and we can classify them initially into the following categories:

- Committed memory
- Virtual memory
  - Kernel space
    - Paged pool
    - Non-paged pool
    - Session pool
    - PTE limits
    - Desktop heap
    - GDI limits
  - User space
    - Virtual regions
    - Process heap

What we outline here is committed memory exhaustion. Committed memory is an allocated memory backed up by some physical memory or by a reserved space in the page file(s). Reserving the space needs to be done in case OS wants to swap out that memory data to disk when it is not used, and there is no physical memory available for other processes. If that data is needed again, OS brings it back to physical memory. If there is no space in the page file(s), then physical memory is filled up. If committed memory is exhausted, most likely, the system will hang or result in a bugcheck soon so checking memory statistics shall always be done when we get a kernel or a complete memory dump. Even access violation bugchecks could result from insufficient memory when some memory allocation operation failed, but a kernel mode component didn't check the return value for NULL. Here is an example:

```
BugCheck 8E, {c0000005, 809203af, aa647c0c, 0}

0: kd> !analyze -v
...
...
...
TRAP_FRAME: aa647c0c -- (.trap ffffffaa647c0c)
...
...
...

0: kd> .trap ffffffaa647c0c
ErrCode = 00000000
```

```

eax=00000000 ebx=bc1f3cfc ecx=89589250 edx=000018c1 esi=bc1f3ce0 edi=aa647d14
eip=809203af esp=aa647c80 ebp=aa647c80 iopl=0 nv up ei pl zr na pe nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010246
nt!SeTokenType+0x8:
809203af 8b8080000000 mov eax,dword ptr [eax+80h] ds:0023:00000080=?????????

0: kd> k
ChildEBP RetAddr
aa647c80 bf8173c5 nt!SeTokenType+0x8
aa647cdc bf81713b win32k!GreGetSpoolMessage+0xb0
aa647d4c 80834d3f win32k!NtGdiGetSpoolMessage+0x96
aa647d4c 7c82ed54 nt!KiFastCallEntry+0xfc

```

If we enter **!vm** command to display memory statistics we would see that all committed memory is filled up:

```

0: kd> !vm
*** Virtual Memory Usage ***
Physical Memory: 999294 ( 3997176 Kb)
Page File: \??\C:\pagefile.sys
  Current: 4193280 Kb  Free Space: 533744 Kb
  Minimum: 4193280 Kb  Maximum: 4193280 Kb
Available Pages: 18698 ( 74792 Kb)
ResAvail Pages: 865019 ( 3460076 Kb)
Locked IO Pages: 290 ( 1160 Kb)
Free System PTEs: 155265 ( 621060 Kb)
Free NP PTEs: 32766 ( 131064 Kb)
Free Special NP: 0 ( 0 Kb)
Modified Pages: 113 ( 452 Kb)
Modified PF Pages: 61 ( 244 Kb)
NonPagedPool Usage: 12380 ( 49520 Kb)
NonPagedPool Max: 64799 ( 259196 Kb)
PagedPool 0 Usage: 40291 ( 161164 Kb)
PagedPool 1 Usage: 2463 ( 9852 Kb)
PagedPool 2 Usage: 2455 ( 9820 Kb)
PagedPool 3 Usage: 2453 ( 9812 Kb)
PagedPool 4 Usage: 2488 ( 9952 Kb)
PagedPool Usage: 50150 ( 200600 Kb)
PagedPool Maximum: 67584 ( 270336 Kb)

***** 18 pool allocations have failed *****

Shared Commit: 87304 ( 349216 Kb)
Special Pool: 0 ( 0 Kb)
Shared Process: 56241 ( 224964 Kb)
PagedPool Commit: 50198 ( 200792 Kb)
Driver Commit: 1892 ( 7568 Kb)
Committed pages: 2006945 ( 8027780 Kb)
Commit limit: 2008205 ( 8032820 Kb)

***** 1216024 commit requests have failed *****

```

There might have been a memory leak or too many terminal sessions with fat applications to fit in physical memory and the page file.

## Control Blocks

Certain system and subsystem architectures and designs may put a hard limit on the amount of data structures created to manage resources. If there is a dependency on such resources from other subsystems, there could be starvation and blockage conditions resulting in sluggish system behavior, the absence of functional response and even in some cases perceived system, service or application freeze. For example, **!filecache** WinDbg command diagnoses low VACB (Virtual [or View] Address Control Block) conditions.

```
7: kd> !filecache
***** Dump file cache*****
Reading and sorting VACBs ...
Removed 0 nonactive VACBs, processing 1907 active VACBs ...
File Cache Information
Current size 408276 kb
Peak size    468992 kb
1907 Control Areas
[...]
```

## Handle Leak

Sometimes **Handle Leaks** (page 416) also result in insufficient memory especially if handles point to structures allocated by OS. Here is the typical example of the handle leak resulted in freezing several servers. The complete memory dump shows exhausted non-paged pool:

```
0: kd> !vm

*** Virtual Memory Usage ***
Physical Memory: 1048352 ( 4193408 Kb)
Page File: \??\C:\pagefile.sys
  Current: 4190208 Kb Free Space: 3749732 Kb
  Minimum: 4190208 Kb Maximum: 4190208 Kb
Available Pages: 697734 ( 2790936 Kb)
ResAvail Pages: 958085 ( 3832340 Kb)
Locked IO Pages: 95 ( 380 Kb)
Free System PTEs: 199971 ( 799884 Kb)
Free NP PTEs: 105 ( 420 Kb)
Free Special NP: 0 ( 0 Kb)
Modified Pages: 195 ( 780 Kb)
Modified PF Pages: 195 ( 780 Kb)
NonPagedPool Usage: 65244 ( 260976 Kb)
NonPagedPool Max: 65503 ( 262012 Kb)
***** Excessive NonPaged Pool Usage *****
PagedPool 0 Usage: 6576 ( 26304 Kb)
PagedPool 1 Usage: 629 ( 2516 Kb)
PagedPool 2 Usage: 624 ( 2496 Kb)
PagedPool 3 Usage: 608 ( 2432 Kb)
PagedPool 4 Usage: 625 ( 2500 Kb)
PagedPool Usage: 9062 ( 36248 Kb)
PagedPool Maximum: 66560 ( 266240 Kb)

***** 184 pool allocations have failed *****

Shared Commit: 7711 ( 30844 Kb)
Special Pool: 0 ( 0 Kb)
Shared Process: 10625 ( 42500 Kb)
PagedPool Commit: 9102 ( 36408 Kb)
Driver Commit: 1759 ( 7036 Kb)
Committed pages: 425816 ( 1703264 Kb)
Commit limit: 2052560 ( 8210240 Kb)
```

Looking at non-paged pool consumption reveals an excessive number of thread objects:

```
0: kd> !poolused 3
Sorting by NonPaged Pool Consumed

Pool Used:
    NonPaged
Tag    Allocs   Frees    Diff     Used
Thre  772672   463590  309082 192867168 Thread objects , Binary: nt!ps
MmCm      42        9       33 12153104 Calls made to MmAllocateContiguousMemory , Binary: nt!mm
...
...
...
```

The next logical step would be to list processes and find their handle usage. Indeed, there is such a process:

```
0: kd> !process 0 0
...
...
...
PROCESS 88b75020 SessionId: 7 Cid: 172e4 Peb: 7ffdf000 ParentCid: 17238
  DirBase: c7fb6bc0 ObjectTable: e17f50a0 HandleCount: 143428.
  Image: iexplore.exe
...
...
...
```

Making the process current and listing its handles shows contiguously allocated handles to thread objects:

```
0: kd> .process 88b75020
Implicit process is now 88b75020

0: kd> .reload /user

0: kd> !handle
...
...
...
0d94: Object: 88a6b020 GrantedAccess: 001f03ff Entry: e35e1b28
Object: 88a6b020 Type: (8b780c68) Thread
  ObjectHeader: 88a6b008
    HandleCount: 1 PointerCount: 1

0d98: Object: 88a97320 GrantedAccess: 001f03ff Entry: e35e1b30
Object: 88a97320 Type: (8b780c68) Thread
  ObjectHeader: 88a97308
    HandleCount: 1 PointerCount: 1

0d9c: Object: 88b2b020 GrantedAccess: 001f03ff Entry: e35e1b38
Object: 88b2b020 Type: (8b780c68) Thread
  ObjectHeader: 88b2b008
    HandleCount: 1 PointerCount: 1
```

```
0da0: Object: 88b2a730 GrantedAccess: 001f03ff Entry: e35e1b40
Object: 88b2a730 Type: (8b780c68) Thread
    ObjectHeader: 88b2a718
        HandleCount: 1 PointerCount: 1

0da4: Object: 88b929a0 GrantedAccess: 001f03ff Entry: e35e1b48
Object: 88b929a0 Type: (8b780c68) Thread
    ObjectHeader: 88b92988
        HandleCount: 1 PointerCount: 1

0da8: Object: 88a57db0 GrantedAccess: 001f03ff Entry: e35e1b50
Object: 88a57db0 Type: (8b780c68) Thread
    ObjectHeader: 88a57d98
        HandleCount: 1 PointerCount: 1

0dac: Object: 88b92db0 GrantedAccess: 001f03ff Entry: e35e1b58
Object: 88b92db0 Type: (8b780c68) Thread
    ObjectHeader: 88b92d98
        HandleCount: 1 PointerCount: 1

0db0: Object: 88b4a730 GrantedAccess: 001f03ff Entry: e35e1b60
Object: 88b4a730 Type: (8b780c68) Thread
    ObjectHeader: 88b4a718
        HandleCount: 1 PointerCount: 1

0db4: Object: 88a7e730 GrantedAccess: 001f03ff Entry: e35e1b68
Object: 88a7e730 Type: (8b780c68) Thread
    ObjectHeader: 88a7e718
        HandleCount: 1 PointerCount: 1

0db8: Object: 88a349a0 GrantedAccess: 001f03ff Entry: e35e1b70
Object: 88a349a0 Type: (8b780c68) Thread
    ObjectHeader: 88a34988
        HandleCount: 1 PointerCount: 1

0dbc: Object: 88a554c0 GrantedAccess: 001f03ff Entry: e35e1b78
Object: 88a554c0 Type: (8b780c68) Thread
    ObjectHeader: 88a554a8
        HandleCount: 1 PointerCount: 1
...
```

Examination of these threads shows their stack traces and start addresses:

```
0: kd> !thread 88b4a730
THREAD 88b4a730 Cid 0004.1885c Teb: 00000000 Win32Thread: 00000000 TERMINATED
Not impersonating
DeviceMap          e1000930
Owning Process    8b7807a8      Image:       System
Wait Start TickCount 975361      Ticks: 980987 (0:04:15:27.921)
Context Switch Count 1
UserTime           00:00:00.0000
KernelTime         00:00:00.0000
Start Address mydriver!StatusWaitThread (0xf5c5d128)
Stack Init 0 Current f3c4cc98 Base f3c4d000 Limit f3c4a000 Call 0
Priority 8 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr Args to Child
f3c4ccac 8083129e ffdf5f0 8697ba00 a674c913 hal!KfLowerIrql+0x62
f3c4ccc8 00000000 808ae498 8697ba00 00000000 nt!KiExitDispatcher+0x130

0: kd> !thread 88a554c0
THREAD 88a554c0 Cid 0004.1888c Teb: 00000000 Win32Thread: 00000000 TERMINATED
Not impersonating
DeviceMap          e1000930
Owning Process    8b7807a8      Image:       System
Wait Start TickCount 975380      Ticks: 980968 (0:04:15:27.625)
Context Switch Count 1
UserTime           00:00:00.0000
KernelTime         00:00:00.0000
Start Address mydriver!StatusWaitThread (0xf5c5d128)
Stack Init 0 Current f3c4cc98 Base f3c4d000 Limit f3c4a000 Call 0
Priority 8 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr Args to Child
f3c4ccac 8083129e ffdf5f0 8697ba00 a674c913 hal!KfLowerIrql+0x62
f3c4ccc8 00000000 808ae498 8697ba00 00000000 nt!KiExitDispatcher+0x130
```

We can see that these threads have been terminated, and their start address belongs to *mydriver.sys*. Therefore, we can say that mydriver code has to be examined to find the source of our handle leak.

## Comments

Enabling Application Verifier using `gflags.exe` before running the leaking application and then using `!htrace` WinDbg command can show stack traces for handle usage. For example, quick test for `notepad.exe`:

```
0:001> !htrace 0x00000100
-----
Handle = 0x00000100 - CLOSE
Thread ID = 0x00002d7c, Process ID = 0x0000237c

00x7c836b0c: ntdll!LdrGetDllHandleEx+0x000002b8
0x7c836cf9: ntdll!LdrGetDllHandle+0x00000018
0x77e645c2: kernel32!GetModuleHandleForUnicodeString+0x00000049
0x77e66516: kernel32!BasepGetModuleHandleExW+0x00000017f
0x77e663f4: kernel32!GetModuleHandleW+0x00000029
0x773a2eb3: USER32!_InitializeImmEntryTable+0x00000047
0x773c5369: USER32!_UserClientDllInitialize+0x00000163
0x5a61897f: verifier!AVrfpStandardDllEntryPointRoutine+0x0000014f
0x7c82257a: ntdll!LdrpCallInitRoutine+0x00000014
0x7c8358fb: ntdll!LdrpRunInitializeRoutines+0x00000367
-----
Handle = 0x00000100 - OPEN
Thread ID = 0x00002d7c, Process ID = 0x0000237c

00x7c836b0c: ntdll!LdrGetDllHandleEx+0x000002b8
0x7c836cf9: ntdll!LdrGetDllHandle+0x00000018
0x77e645c2: kernel32!GetModuleHandleForUnicodeString+0x00000049
0x77e66516: kernel32!BasepGetModuleHandleExW+0x00000017f
0x77e663f4: kernel32!GetModuleHandleW+0x00000029
0x773a2eb3: USER32!_InitializeImmEntryTable+0x00000047
0x773c5369: USER32!_UserClientDllInitialize+0x00000163
0x5a61897f: verifier!AVrfpStandardDllEntryPointRoutine+0x0000014f
-----
Parsed 0x2EC stack traces.
Dumped 0x2 stack traces.
```

Another example from the complete memory dump where Application Verifier was enabled for `MyApp.exe`:

```
03: kd> .process /p /r 0x8896e670
Implicit process is now 8896e670
Loading User Symbols

03: kd> dt _PEB 7ffdd000
ntdll!_PEB
+0x000 InheritedAddressSpace : 0 "
+0x001 ReadImageFileExecOptions : 0x1 "
+0x002 BeingDebugged : 0 "
+0x003 BitField : 0 "
+0x003 ImageUsesLargePages : 0y0
+0x003 SpareBits : 0y0000000 (0)
+0x004 Mutant : 0xffffffff
+0x008 ImageBaseAddress : 0x00400000
```

```
+0x00c Ldr : 0x7c889e00 _PEB_LDR_DATA
+0x010 ProcessParameters : 0x00020000 _RTL_USER_PROCESS_PARAMETERS
+0x014 SubSystemData : (null)
+0x018 ProcessHeap : 0x00140000
+0x01c FastPebLock : 0x7c889d40 _RTL_CRITICAL_SECTION
+0x020 AtlThunkSListPtr : (null)
+0x024 SparePtr2 : (null)
+0x028 EnvironmentUpdateCount : 1
+0x02c KernelCallbackTable : 0x77382970
+0x030 SystemReserved : [1] 0
+0x034 SpareUlong : 0
+0x038 FreeList : (null)
+0x03c TlsExpansionCounter : 0
+0x040 TlsBitmap : 0x7c888910
+0x044 TlsBitmapBits : [2] 0xffff
+0x04c ReadOnlySharedMemoryBase : 0x7f6f0000
+0x050 ReadOnlySharedMemoryHeap : 0x7f6f0000
+0x054 ReadOnlyStaticServerData : 0x7f6f0688 -> (null)
+0x058 AnsiCodePageData : 0x7ffb0000
+0x05c OemCodePageData : 0x7ffc1000
+0x060 UnicodeCaseTableData : 0x7ffd2000
+0x064 NumberOfProcessors : 4
+0x068 NtGlobalFlag : 0x100
+0x070 CriticalSectionTimeout : _LARGE_INTEGER 0xfffffe86d`079b8000
+0x078 HeapSegmentReserve : 0x100000
+0x07c HeapSegmentCommit : 0x2000
+0x080 HeapDeCommitTotalFreeThreshold : 0x1000
+0x084 HeapDeCommitFreeBlockThreshold : 0x1000
+0x088 NumberOfHeaps : 0xb
+0x08c MaximumNumberOfHeaps : 0x10
+0x090 ProcessHeaps : 0x7c888340 -> 0x00140000
+0x094 GdiSharedHandleTable : 0x015f0000
+0x098 ProcessStarterHelper : (null)
+0x09c GdiDCAttributeList : 0x14
+0x0a0 LoaderLock : 0x7c889d94 _RTL_CRITICAL_SECTION
+0x0a4 OSMajorVersion : 5
+0x0a8 OSMinorVersion : 2
+0x0ac OSBuildNumber : 0xece
+0x0ae OSCSDVersion : 0x100
+0x0b0 OSPlatformId : 2
+0x0b4 ImageSubsystem : 2
+0x0b8 ImageSubsystemMajorVersion : 4
+0x0bc ImageSubsystemMinorVersion : 0
+0x0c0 ImageProcessAffinityMask : 0
+0x0c4 GdiHandleBuffer : [34] 0
+0x14c PostProcessInitRoutine : (null)
+0x150 TlsExpansionBitmap : 0x7c888908
+0x154 TlsExpansionBitmapBits : [32] 1
+0x1d4 SessionId : 1
+0x1d8 AppCompatFlags : _ULARGE_INTEGER 0x0
+0x1e0 AppCompatFlagsUser : _ULARGE_INTEGER 0x0
+0x1e8 pShimData : (null)
+0x1ec AppCompatInfo : (null)
+0x1f0 CSDVersion : _UNICODE_STRING "Service Pack 1"
+0x1f8 ActivationContextData : (null)
+0x1fc ProcessAssemblyStorageMap : (null)
```

```
+0x200 SystemDefaultActivationContextData : 0x00130000 _ACTIVATION_CONTEXT_DATA
+0x204 SystemAssemblyStorageMap : (null)
+0x208 MinimumStackCommit : 0
+0x20c FlsCallback : 0x001454a8 -> (null)
+0x210 FlsListHead : _LIST_ENTRY [ 0x141f60 - 0x169018 ]
+0x218 FlsBitmap : 0x7c888388
+0x21c FlsBitmapBits : [4] 7
+0x22c FlsHighIndex : 2

03: kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS 8896e670 SessionId: 1 Cid: 24b4 Peb: 7ffdd000 ParentCid: 1270
DirBase: 48e9f000 ObjectTable: e11109d0 HandleCount: 8717.
Image: MyApp.exe

03: kd> !handle
processor number 3, process 8896e670
PROCESS 8896e670 SessionId: 1 Cid: 24b4 Peb: 7ffdd000 ParentCid: 1270
DirBase: 48e9f000 ObjectTable: e11109d0 HandleCount: 8717.
Image: MyApp.exe

...
Handle table at e50ed000 with 8717 Entries in use
0004: Object: 88f39978 GrantedAccess: 00000410 Entry: e102a008
Object: 88f39978 Type: (8a392ca0) Process
ObjectHeader: 88f39960 (old version)
HandleCount: 8459 PointerCount: 8507

00008: Object: 88f39978 GrantedAccess: 00000410 Entry: e102a010
Object: 88f39978 Type: (8a392ca0) Process
ObjectHeader: 88f39960 (old version)
HandleCount: 8459 PointerCount: 8507

0000c: Object: 88f39978 GrantedAccess: 00000410 Entry: e102a018
Object: 88f39978 Type: (8a392ca0) Process
ObjectHeader: 88f39960 (old version)
HandleCount: 8459 PointerCount: 8507

...
03: kd> !htrace 0 8896e670
Process 0x8896e670
ObjectTable 0xe11109d0

-----
Handle 0x8878 - OPEN
Thread ID = 0x00000ca4, Process ID = 0x000024b4

00x809b94d8: nt!ExpUpdateDebugInfo+0x16D
0x8097232e: nt!ExCreateHandle+0x4A
0x8092b6ba: nt!ObpCreateHandle+0x3DE
0x80927674: nt!ObOpenObjectByPointer+0xA8
```

```

0x8093c2a5: nt!NtOpenProcess+0x22C
0x80834d3f: nt!KiFastCallEntry+0xFC
0x66861404: MyDLL!GetProcHandle+0x20
0x66862966: MyDLL!MainWndProc+0x1CC
0x7739c3b7: USER32!InternalCallWinProc+0x28
0x7739c484: USER32!UserCallWinProcCheckWow+0x151
0x7739ca68: USER32!DispatchClientMessage+0xD9
0x7739ce7a: USER32!__fnDWORD+0x24
0x7c82ec9e: ntdll!KiUserCallbackDispatcher+0x2E
0x66863244: MyDLL!Load+0x377
0x5a614cb0: verifier!AVrfpStandardThreadFunction+0x60

```

Example of handle leak in System process:

```

kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS 8d1615d8 SessionId: none Cid: 0004 Peb: 00000000 ParentCid: 0000
DirBase: bffba020 ObjectTable: e1002e08 HandleCount: 10879.
Image: System

3: kd> !handle 0 12

Kernel Handle table at e1765000 with 10879 Entries in use

1c20: Object: e4fcfa908 GrantedAccess: 00000004 Entry: e4b18840
Object: e4fcfa908 Type: (8d18a110) Directory
ObjectHeader: e4fcfa8f0 (old version)

1c24: Object: e5bf14c8 GrantedAccess: 00000004 Entry: e4b18848
Object: e5bf14c8 Type: (8d18a110) Directory
ObjectHeader: e5bf14b0 (old version)

1c28: Object: e496d490 GrantedAccess: 00000004 Entry: e4b18850
Object: e496d490 Type: (8d18a110) Directory
ObjectHeader: e496d478 (old version)

1c2c: Object: ea4a1370 GrantedAccess: 00000004 Entry: e4b18858
Object: ea4a1370 Type: (8d18a110) Directory
ObjectHeader: ea4a1358 (old version)

1c30: Object: e68558d0 GrantedAccess: 00000004 Entry: e4b18860
Object: e68558d0 Type: (8d18a110) Directory
ObjectHeader: e68558b8 (old version)

1c34: Object: e49f7d88 GrantedAccess: 00000004 Entry: e4b18868
Object: e49f7d88 Type: (8d18a110) Directory
ObjectHeader: e49f7d70 (old version)

1c38: Object: e3d8c6e8 GrantedAccess: 00000004 Entry: e4b18870
Object: e3d8c6e8 Type: (8d18a110) Directory
ObjectHeader: e3d8c6d0 (old version)

```

We can also use “Miscellaneous Checks” option in Driver Verifier to enable handle traces in the System process<sup>107</sup>.

<sup>107</sup> <http://msdn.microsoft.com/en-us/library/ff549253.aspx>

## Kernel Pool

Although **Handle Leaks** (page 416) may result in insufficient pool memory, many drivers allocate their own private memory and specify a 4-letter ASCII tag, for example, here is the non-paged pool from my x64 Vista workstation:

```
1kd> !poolused 3
Sorting by NonPaged Pool Consumed

Pool Used:
NonPaged
Tag    Allocs   Frees     Diff     Used
EtwB      304       134      170  6550080 Etw Buffer , Binary: nt!etw
File 32630649 32618671 11978 3752928 File objects
Pool      16        11       5  3363472 Pool tables, etc.
Ntfr    204791  187152  17639 2258704 ERESOURCE , Binary: ntfs.sys
FMsl    199039  187685 11354 2179968 STREAM_LIST_CTRL structure , Binary: fltmgr.sys
MmCa    250092  240351  9741 2134368 Mm control areas for mapped files , Binary: nt!mm
ViMm    135503  134021  1482 1783824 Video memory manager , Binary: dxgkrnl.sys
Cont      53        12       41  1567664 Contiguous physical memory allocations for device drivers
Thre    72558   71527  1031 1234064 Thread objects , Binary: nt!ps
VoSm     872       851       21 1220544 Bitmap allocations , Binary: volsnap.sys
NtFs   8122505  8110933 11572 1190960 StrucSup.c , Binary: ntfs.sys
AmIh      1         0       1 1048576 ACPI AMLI Pooltags
SaSc    20281   14820  5461 1048512 UNKNOWN pooltag 'SaSc', please update pooltag.txt
RaRS     1000       0     1000  960000 UNKNOWN pooltag 'RaRS', please update pooltag.txt
...
...
...
```

If the pool tag is unknown, we can use recommendations from Microsoft article KB298102 that explains how to locate the corresponding driver<sup>108</sup>. We can also use memory search in WinDbg to locate kernel space addresses and see what modules they correspond to (**Value References** pattern, page 1057).

WinDbg shows the number of failed pool allocations and also shows a message when pool usage is nearly its maximum. Below I put some examples with possible troubleshooting hints.

### Session pool

```
3: kd> !vm

*** Virtual Memory Usage ***
Physical Memory: 1572637 ( 6290548 Kb)
Page File: \??\C:\pagefile.sys
Current: 3145728 Kb Free Space: 3001132 Kb
Minimum: 3145728 Kb Maximum: 3145728 Kb
Available Pages: 1317401 ( 5269604 Kb)
```

<sup>108</sup> <http://support.microsoft.com/kb/298102>

```

ResAvail Pages:      1478498 (  5913992 Kb)
Locked IO Pages:     114 (      456 Kb)
Free System PTEs:    194059 (  776236 Kb)
Free NP PTEs:       32766 ( 131064 Kb)
Free Special NP:     0 (      0 Kb)
Modified Pages:      443 (   1772 Kb)
Modified PF Pages:   442 (   1768 Kb)
NonPagedPool Usage: 13183 (  52732 Kb)
NonPagedPool Max: 65215 ( 260860 Kb)
PagedPool 0 Usage:  11328 (  45312 Kb)
PagedPool 1 Usage:  1473 (   5892 Kb)
PagedPool 2 Usage:  1486 (   5944 Kb)
PagedPool 3 Usage:  1458 (   5832 Kb)
PagedPool 4 Usage:  1505 (   6020 Kb)
PagedPool Usage: 17250 ( 69000 Kb)
PagedPool Maximum:  65536 ( 262144 Kb)

```

\*\*\*\*\* 3441 pool allocations have failed \*\*\*\*\*

```

Shared Commit:        8137 (   32548 Kb)
Special Pool:        0 (      0 Kb)
Shared Process:      8954 (  35816 Kb)
PagedPool Commit:    17312 (  69248 Kb)
Driver Commit:       2095 (   8380 Kb)
Committed pages:    212476 ( 849904 Kb)
Commit limit:        2312654 ( 9250616 Kb)

```

Paged and non-paged pool usage is far from maximum, therefore, we check session pool:

3: kd> !vm 4

Terminal Server Memory Usage By Session:

```

Session Paged Pool Maximum is 32768K
Session View Space Maximum is 20480K

```

```

Session ID 0 @ f79a1000:
Paged Pool Usage:      9824K
Commit Usage:          10148K

```

```

Session ID 2 @ f7989000:
Paged Pool Usage:      1212K
Commit Usage:          2180K
[...]

```

```

Session ID 9 @ f79b5000:
Paged Pool Usage:      32552K

```

\*\*\* 7837 Pool Allocation Failures \*\*\*

```

Commit Usage:          33652K

```

Here Microsoft article KB840342 may help<sup>109</sup>.

### Paged pool

We might have a direct warning here:

```
1: kd> !vm

*** Virtual Memory Usage ***
Physical Memory: 511881 ( 2047524 Kb)
Page File: \??\S:\pagefile.sys
    Current: 2098176Kb Free Space: 1837740Kb
    Minimum: 2098176Kb Maximum: 2098176Kb
Page File: \??\R:\pagefile.sys
    Current: 1048576Kb Free Space: 792360Kb
    Minimum: 1048576Kb Maximum: 1048576Kb
Available Pages: 201353 ( 805412 Kb)
ResAvail Pages: 426839 ( 1707356 Kb)
Modified Pages: 45405 ( 181620 Kb)
NonPagedPool Usage: 10042 ( 40168 Kb)
NonPagedPool Max: 68537 ( 274148 Kb)
PagedPool 0 Usage: 26820 ( 107280 Kb)
PagedPool 1 Usage: 1491 ( 5964 Kb)
PagedPool 2 Usage: 1521 ( 6084 Kb)
PagedPool 3 Usage: 1502 ( 6008 Kb)
PagedPool 4 Usage: 1516 ( 6064 Kb)
***** Excessive Paged Pool Usage *****
PagedPool Usage: 32850 ( 131400 Kb)
PagedPool Maximum: 40960 ( 163840 Kb)
Shared Commit: 14479 ( 57916 Kb)
Special Pool: 0 ( 0 Kb)
Free System PTEs: 135832 ( 543328 Kb)
Shared Process: 15186 ( 60744 Kb)
PagedPool Commit: 32850 ( 131400 Kb)
Driver Commit: 1322 ( 5288 Kb)
Committed pages: 426786 ( 1707144 Kb)
Commit limit: 1259456 ( 5037824 Kb)
```

If there is no warning we can check the size manually, and if paged pool usage is close to its maximum, but for the non-paged pool it is not, then, most likely, failed allocations were from the paged pool:

---

<sup>109</sup> <http://support.microsoft.com/kb/840342>

```
0: kd> !vm

*** Virtual Memory Usage ***
Physical Memory: 4193696 ( 16774784 Kb)
Page File: \??\C:\pagefile.sys
  Current: 4193280 Kb  Free Space: 3313120 Kb
  Minimum: 4193280 Kb  Maximum: 4193280 Kb
Available Pages: 3210617 ( 12842468 Kb)
ResAvail Pages: 4031978 ( 16127912 Kb)
Locked IO Pages: 120 ( 480 Kb)
Free System PTEs: 99633 ( 398532 Kb)
Free NP PTEs: 26875 ( 107500 Kb)
Free Special NP: 0 ( 0 Kb)
Modified Pages: 611 ( 2444 Kb)
Modified PF Pages: 590 ( 2360 Kb)
NonPagedPool 0 Used: 8271 ( 33084 Kb)
NonPagedPool 1 Used: 13828 ( 55312 Kb)
NonPagedPool Usage: 37846 ( 151384 Kb)
NonPagedPool Max: 65215 ( 260860 Kb)
PagedPool 0 Usage: 82308 ( 329232 Kb)
PagedPool 1 Usage: 12700 ( 50800 Kb)
PagedPool 2 Usage: 25702 ( 102808 Kb)
PagedPool Usage: 120710 ( 482840 Kb)
PagedPool Maximum: 134144 ( 536576 Kb)

***** 818 pool allocations have failed *****

Shared Commit: 80168 ( 320672 Kb)
Special Pool: 0 ( 0 Kb)
Shared Process: 55654 ( 222616 Kb)
PagedPool Commit: 120772 ( 483088 Kb)
Driver Commit: 1890 ( 7560 Kb)
Committed pages: 1344388 ( 5377552 Kb)
Commit limit: 5177766 ( 20711064 Kb)
```

**!poolused 4** WinDbg command will sort paged pool consumption by pool tag:

```
0: kd> !poolused 4
Sorting by Paged Pool Consumed

Pool Used:
      NonPaged          Paged
Tag    Allocs     Used   Allocs     Used
MmSt      0        0    85622 140642616      Mm section object prototype ptes , Binary: nt!mm
Ntff      5      1040    63715 51991440      FCB_DATA , Binary: ntfs.sys
```

Here Microsoft article KB312362 may help<sup>110</sup>.

---

<sup>110</sup> <http://support.microsoft.com/kb/312362>

Non-paged pool

```
0: kd> !vm

*** Virtual Memory Usage ***
Physical Memory: 851775 ( 3407100 Kb)
Page File: \??\C:\pagefile.sys
  Current: 4190208 Kb  Free Space: 4175708 Kb
  Minimum: 4190208 Kb  Maximum: 4190208 Kb
Available Pages: 147274 ( 589096 Kb)
ResAvail Pages: 769287 ( 3077148 Kb)
Locked IO Pages: 118 ( 472 Kb)
Free System PTEs: 184910 ( 739640 Kb)
Free NP PTEs: 110 ( 440 Kb)
Free Special NP: 0 ( 0 Kb)
Modified Pages: 168 ( 672 Kb)
Modified PF Pages: 168 ( 672 Kb)
NonPagedPool Usage: 64445 ( 257780 Kb)
NonPagedPool Max: 64640 ( 258560 Kb)
***** Excessive NonPaged Pool Usage *****
PagedPool 0 Usage: 21912 ( 87648 Kb)
PagedPool 1 Usage: 691 ( 2764 Kb)
PagedPool 2 Usage: 706 ( 2824 Kb)
PagedPool 3 Usage: 704 ( 2816 Kb)
PagedPool 4 Usage: 708 ( 2832 Kb)
PagedPool Usage: 24721 ( 98884 Kb)
PagedPool Maximum: 134144 ( 536576 Kb)
```

```
***** 429 pool allocations have failed *****
```

```
Shared Commit: 5274 ( 21096 Kb)
Special Pool: 0 ( 0 Kb)
Shared Process: 3958 ( 15832 Kb)
PagedPool Commit: 24785 ( 99140 Kb)
Driver Commit: 19289 ( 77156 Kb)
Committed pages: 646282 ( 2585128 Kb)
Commit limit: 1860990 ( 7443960 Kb)
```

**!poolused 3** WinDbg command will sort non-paged pool consumption by pool tag:

```
0: kd> !poolused 3
Sorting by NonPaged Pool Consumed

Pool Used:
  NonPaged
Tag    Allocs   Frees   Diff
Ddk    9074558  3859522  5215036 Default for driver allocated memory (user's of ntddk.h)
MmCm    43787    42677    1110 Calls made to MmAllocateContiguousMemory , Binary: nt!mm
LSwi      1       0       1 initial work context
TCPt   3281838  3281808     30 TCP/IP network protocol , Binary: TCP
```

<sup>111</sup> Regarding Ddk tag, please see “The Search for Tags” case study.

The following Microsoft article KB293857<sup>112</sup> explains how we can use **xpool** command from old *kdex2x86.dll* extension which even works for Windows 2003 dumps:

```
0: kd> !w2kfre\kdex2x86.xpool -map
unable to get NT!MmSizeOfNonPagedMustSucceed location
unable to get NT!MmSubsectionTopPage location
unable to get NT!MmKseg2Frame location
unable to get NT!MmNonPagedMustSucceed location
```

Status Map of Pool Area Pages

```
=====
': one page in use ('P': paged out)
'<': start page of contiguous pages in use ('{': paged out)
'>': last page of contiguous pages in use ('}': paged out)
'=': intermediate page of contiguous pages in use ('-': paged out)
'.': one page not used
```

## Non-Paged Pool Area Summary

Maximum Number of Pages = 64640 pages  
Number of Pages In Use = 36721 pages (56.8%)

<sup>111</sup> The Search for Tags, Memory Dump Analysis Anthology, Volume 1, page 206

<sup>112</sup> <http://support.microsoft.com/kb/293857>

89540000: ..=..... . =..... . =..... . =..... . =..... =..... =..... =.....  
89580000: .....=.=..... =.=..... ,==== ==.=.=..... =..... =..... =..... =.=.=..  
895c0000: =.....==..... . =..... . =..... . =..... =..... =..... =.....  
89600000: .....=.=.=.=..... =..... =..... =.=.=..... =..... =..... =..... =.....  
89640000: ..=.....==..... =..... =..... =.=.=..... =.=.=..... =..... =.=.....  
...  
...

Here is another example:

```
0: kd> !vm

*** Virtual Memory Usage ***
Physical Memory: 786299 ( 3145196 Kb)
Page File: \??\C:\pagefile.sys
    Current: 4193280Kb Free Space: 3407908Kb
    Minimum: 4193280Kb Maximum: 4193280Kb
Available Pages: 200189 ( 800756 Kb)
ResAvail Pages: 657130 ( 2628520 Kb)
Modified Pages: 762 ( 3048 Kb)
NonPagedPool Usage: 22948 ( 91792 Kb)
NonPagedPool Max: 70145 ( 280580 Kb)
PagedPool 0 Usage: 19666 ( 78664 Kb)
PagedPool 1 Usage: 3358 ( 13432 Kb)
PagedPool 2 Usage: 3306 ( 13224 Kb)
PagedPool 3 Usage: 3312 ( 13248 Kb)
PagedPool 4 Usage: 3309 ( 13236 Kb)
***** Excessive Paged Pool Usage *****
PagedPool Usage: 32951 ( 131804 Kb)
PagedPool Maximum: 40960 ( 163840 Kb)
Shared Commit: 9664 ( 38656 Kb)
Special Pool: 0 ( 0 Kb)
Free System PTEs: 103335 ( 413340 Kb)
Shared Process: 45024 ( 180096 Kb)
PagedPool Commit: 32951 ( 131804 Kb)
Driver Commit: 1398 ( 5592 Kb)
Committed pages: 864175 ( 3456700 Kb)
Commit limit: 1793827 ( 7175308 Kb)
```

```
0: kd> !poolused 4
      Sorting by Paged Pool Consumer
```

Pool Used:		NonPaged		Paged	
Tag	Allocs	Used	Allocs	Used	
CM	85	5440	11045	47915424	
MyAV	0	0	186	14391520	
MmSt	0	0	11795	13235744	
Dbtb	709	90752	2712	11108352	
Ntff	5	1120	9886	8541504	
..					

*MyAV* tag seems to be the prefix for *MyAVDrv* module, and this is hardly a coincidence. Looking at the list of drivers we see that *MyAVDrv.sys* was loaded and unloaded several times. Could it be that it did not free its non-paged pool allocations?

```
0: kd> lmv m MyAVDrv.sys
start      end          module name

Unloaded modules:
a5069000 a5084000  MyAVDrv.sys
    Timestamp: unavailable (00000000)
    Checksum:  00000000
a5069000 a5084000  MyAVDrv.sys
    Timestamp: unavailable (00000000)
    Checksum:  00000000
a5069000 a5084000  MyAVDrv.sys
    Timestamp: unavailable (00000000)
    Checksum:  00000000
b93e1000 b93fc000  MyAVDrv.sys
    Timestamp: unavailable (00000000)
    Checksum:  00000000
b9ae5000 b9b00000  MyAVDrv.sys
    Timestamp: unavailable (00000000)
    Checksum:  00000000
be775000 be790000  MyAVDrv.sys
    Timestamp: unavailable (00000000)
    Checksum:  00000000
```

Also, we see that CM tag has the most allocations and **!locks** command shows hundreds of threads waiting for the registry, one example of **High Contention** pattern (page 477):

```
0: kd> !locks

Resource @ nt!CmpRegistryLock (0x80478b00)      Shared 10 owning threads
Contention Count = 9149810
NumberOfSharedWaiters = 718
NumberOfExclusiveWaiters = 21
```

Therefore, we see at least two problems in this memory dump: excessive paged pool usage and high thread contention around registry resource slowing down if not halting the system.

## Comments

Another example from ntdebugging blog<sup>113</sup>:

To see session pool usage we need to switch to a session process first:

```
1: kd> !sprocess 3
[...]
PROCESS 88b0dd88 SessionId: 3 Cid: 1b90 Peb: 7ffde000 ParentCid: 1b84
DirBase: 4072d000 ObjectTable: d8651450 HandleCount: 414.
Image: ApplicationA.exe
[...]

1: kd> .process /r /p 88b0dd88
Implicit process is now 88b0dd88
Loading User Symbols

1: kd> !poolused 8
Sorting by Session Tag
[...]
```

Excessive nonpaged pool memory usage messages may be false positive for some WinDbg versions<sup>114</sup>.

**!poolused 5** produces the same level of detail for paged pool as **!poolused 3** for nonpaged pool.

---

<sup>113</sup> <http://blogs.msdn.com/ntdebugging/archive/2008/03/26/nonpagedpool-depletion.aspx>

<sup>114</sup> <http://support.microsoft.com/kb/2509968>

## Module Fragmentation

Here we discuss the user space case when we don't have enough virtual memory available for reservation due to memory fragmentation. For example, a java virtual machine is pre-allocating memory for its garbage-collected heap. However, after installing some 3rd-party software, the amount of pre-allocated memory is less than expected. In such cases, it is possible to do comparative memory dump analysis to see the difference in virtual address spaces. Original memory dump has this module distribution in memory:

```
0:000> lm
start   end     module name
00400000 0040b000  javaw      (deferred)
009e0000 009e7000  hpi        (deferred)
00a30000 00a3e000  verify     (deferred)
00a40000 00a59000  java       (deferred)
00a60000 00a6d000  zip        (deferred)
03ff0000 03fff000  net        (deferred)
040a0000 040a8000  nio        (deferred)
040b0000 0410a000  hnetcfg    (deferred)
041d0000 042e2000  awt        (deferred)
04540000 04591000  fontmanager (deferred)
04620000 04670000  msctf      (deferred)
047c0000 047de000  jpeg       (deferred)
05820000 05842000  dcpr       (deferred)
05920000 05932000  pkcs11wrapper (deferred)
08000000 08139000  jvm        (deferred)
10000000 100e0000  moduleA    (deferred)
68000000 68035000  rsaenh     (deferred)
6e220000 6e226000  RMProcessLink (deferred)
71ae0000 71ae8000  wshtcpip   (deferred)
71b20000 71b61000  mswsock    (deferred)
71bf0000 71bf8000  ws2help    (deferred)
71c00000 71c17000  ws2_32    (deferred)
71c20000 71c32000  tsappcmp   (deferred)
71c40000 71c97000  netapi32   (deferred)
73070000 73097000  winspool   (deferred)
76290000 762ad000  imm32     (deferred)
76920000 769e2000  userenv    (deferred)
76aa0000 76acd000  winmm     (deferred)
76b70000 76b7b000  psapi      (deferred)
76d00000 76efa000  dnsapi    (deferred)
76f10000 76f3e000  wldap32   (deferred)
76f50000 76f63000  secur32   (deferred)
76f70000 76f77000  winrnr    (deferred)
76f80000 76f85000  rasadhlp   (deferred)
77380000 77411000  user32    (deferred)
77670000 777a9000  ole32     (deferred)
77ba0000 77bfa000  msvcrt    (deferred)
77c00000 77c48000  gdi32     (deferred)
77c50000 77cef000  rpcrt4    (deferred)
77e40000 77f42000  kernel32   (deferred)
77f50000 77feb000  advapi32   (deferred)
78130000 781cb000  msvcr80   (deferred)
7c800000 7c8c0000  ntdll     (pdb symbols)
```

We see the big gap between 100e0000 and 68000000 addresses. This means that it is theoretically possible to reserve and/or commit up to 57F20000 bytes (about 1.4Gb). **!address** WinDbg command shows that at least 1.1Gb region (shown in bold below) was reserved indeed:

```
0:000> !address
00000000 : 00000000 - 00010000
    Type      00000000
    Protect   00000001 PAGE_NOACCESS
    State     00010000 MEM_FREE
    Usage     RegionUsageFree
00010000 : 00010000 - 00001000
    Type      00020000 MEM_PRIVATE
    Protect   00000004 PAGE_READWRITE
    State     00001000 MEM_COMMIT
    Usage     RegionUsageEnvironmentBlock
00011000 : 00011000 - 0000f000
    Type      00000000
    Protect   00000001 PAGE_NOACCESS
    State     00010000 MEM_FREE
    Usage     RegionUsageFree
00020000 : 00020000 - 00001000
    Type      00020000 MEM_PRIVATE
    Protect   00000004 PAGE_READWRITE
    State     00001000 MEM_COMMIT
    Usage     RegionUsageProcessParametrs
00021000 : 00021000 - 0000f000
    Type      00000000
    Protect   00000001 PAGE_NOACCESS
    State     00010000 MEM_FREE
    Usage     RegionUsageFree
00030000 : 00030000 - 00003000
    Type      00020000 MEM_PRIVATE
    Protect   00000140 <unk>
    State     00001000 MEM_COMMIT
    Usage     RegionUsageStack
    Pid.Tid  97c.1b3c
...
...
...
100e0000 : 100e0000 - 000a0000
    Type      00020000 MEM_PRIVATE
    Protect   00000040 PAGE_EXECUTE_READWRITE
    State     00001000 MEM_COMMIT
    Usage     RegionUsageIsVAD
10180000 - 05cc0000
    Type      00020000 MEM_PRIVATE
    State     00002000 MEM_RESERVE
    Usage     RegionUsageIsVAD
15e40000 - 004f3000
    Type      00020000 MEM_PRIVATE
    Protect   00000040 PAGE_EXECUTE_READWRITE
    State     00001000 MEM_COMMIT
    Usage     RegionUsageIsVAD
16333000 - 45bad000
    Type      00020000 MEM_PRIVATE
    State     00002000 MEM_RESERVE
```

```

Usage      RegionUsageIsVAD
5bee0000 - 00ac0000
    Type      00020000 MEM_PRIVATE
    Protect   00000040 PAGE_EXECUTE_READWRITE
    State     00001000 MEM_COMMIT
    Usage     RegionUsageIsVAD
5c9a0000 - 03540000
    Type      00020000 MEM_PRIVATE
    State     00002000 MEM_RESERVE
    Usage     RegionUsageIsVAD
5fee0000 : 5fee0000 - 00120000
    Type      00000000
    Protect   00000001 PAGE_NOACCESS
    State     00010000 MEM_FREE
    Usage     RegionUsageFree
...
...
...

```

Looking at the problem memory dump, we see that the gap is smaller (less than 1.1Gb):

```

0:000> lm
start      end        module name
00400000 0040b000  javaw      (deferred)
08000000 08139000  jvm        (deferred)
10000000 10007000  hpi        (deferred)
51120000 511bb000  msrvcr80 # (private pdb symbols)
520f0000 520fe000  verify     (deferred)
52100000 52119000  java       (deferred)
52120000 5212d000  zip        (deferred)
52130000 5213f000  net        (deferred)
52140000 52148000  nio        (deferred)
52150000 52262000  awt        (deferred)
52270000 522c1000  fontmanager (deferred)
522d0000 52320000  MSCTF      (deferred)
52330000 5234e000  jpeg       (deferred)
52350000 52372000  dcpr       (deferred)
52510000 52522000  pkcs11wrapper (deferred)
5f270000 5f2ca000  hnetcfg    (deferred)
60000000 60029000  3rdPartyHook (deferred)
61e80000 61e86000  detoured    (export symbols)
68000000 68035000  rsaenh     (deferred)
71ae0000 71ae8000  wshtcpip   (deferred)
71b20000 71b61000  mswsock    (deferred)
71bf0000 71bf8000  ws2help    (deferred)
71c00000 71c17000  ws2_32    (deferred)
71c20000 71c32000  tsappcmp   (deferred)
71c40000 71c97000  netapi32   (deferred)
73070000 73097000  winspool   (deferred)
76290000 762ad000  imm32     (deferred)
76920000 769e2000  userenv    (deferred)
76aa0000 76acd000  winmm     (deferred)
76b70000 76b7b000  psapi      (deferred)
76ed0000 76efa000  dnsapi    (deferred)
76f10000 76f3e000  wldap32    (deferred)
76f50000 76f63000  secur32   (deferred)

```

76f70000	76f77000	winrnr	(deferred)
76f80000	76f85000	rasadhlp	(deferred)
77380000	77411000	user32	(pdb symbols)
77670000	777a9000	ole32	(deferred)
77ba0000	77bfa000	msvcrt	(deferred)
77c00000	77c48000	gdi32	(deferred)
77c50000	77cef000	rpcrt4	(deferred)
77e40000	77f42000	kernel32	(pdb symbols)
77f50000	77feb000	advapi32	(pdb symbols)
7c340000	7c396000	msvcr71	(deferred)
7c800000	7c8c0000	ntdll	(pdb symbols)

**!address** command shows that less memory was reserved in the latter case (about 896Mb):

```
0:000> !address
...
...
...
10010000 : 10010000 - 000a0000
    Type      00020000 MEM_PRIVATE
    Protect   00000040 PAGE_EXECUTE_READWRITE
    State     00001000 MEM_COMMIT
    Usage     RegionUsageIsVAD
100b0000 - 04a70000
    Type      00020000 MEM_PRIVATE
    Protect   00000040 PAGE_EXECUTE_READWRITE
    State     00002000 MEM_RESERVE
    Usage     RegionUsageIsVAD
14b20000 - 004a6000
    Type      00020000 MEM_PRIVATE
    Protect   00000040 PAGE_EXECUTE_READWRITE
    State     00001000 MEM_COMMIT
    Usage     RegionUsageIsVAD
14fc6000 - 3804a000
    Type      00020000 MEM_PRIVATE
    State     00002000 MEM_RESERVE
    Usage     RegionUsageIsVAD
4d010000 - 00ac0000
    Type      00020000 MEM_PRIVATE
    Protect   00000040 PAGE_EXECUTE_READWRITE
    State     00001000 MEM_COMMIT
    Usage     RegionUsageIsVAD
4dad0000 - 03540000
    Type      00020000 MEM_PRIVATE
    State     00002000 MEM_RESERVE
    Usage     RegionUsageIsVAD
51010000 : 51010000 - 00110000
    Type      00000000
    Protect   00000001 PAGE_NOACCESS
    State     00010000 MEM_FREE
    Usage     RegionUsageFree
...
...
...
```

Looking at module list again we notice that most java runtime modules were shifted to 50000000 address range. We also notice that new the *3rdPartyHook* and detoured modules appear in our problem memory dump.

Module information is missing for detoured module:

```
0:000> lmv m detoured
start      end          module name
61e80000 61e86000  detoured  (deferred)
  Image path: C:\WINDOWS\system32\detoured.dll
  Image name: detoured.dll
  Timestamp:    Thu Feb 07 04:14:16 2008 (47AA8598)
  CheckSum:     0000EF91
  ImageSize:    00006000
  File version: 0.0.0.0
  Product version: 0.0.0.0
  File flags:    0 (Mask 0)
  File OS:       0 Unknown Base
  File type:     0.0 Unknown
  File date:    00000000.00000000
  Translations: 0000.04b0 0000.04e0 0409.04b0 0409.04e0
```

Applying **Unknown Component** pattern (page 1035) we see that is Microsoft Research Detours Package:

```
0:000> db 61e80000 61e86000
61e80000  MZ.....
61e80010  .....@.....
61e80020  .....
61e80030  .....
61e80040  .....!..L.!Th
61e80050  is program canno
61e80060  t be run in DOS
61e80070  mode....$.....
61e80080  5...q...q...q...
61e80090  ....r...q...p...
61e800a0  V%..p...V%..p...
61e800b0  V%..p...V%..p...
61e800c0  Richq.....
61e800d0  .....
61e800e0  PE..L.....G....
61e800f0  .....!.....
61e80100  .....
...
...
...
...
61e84390  ..P.r.o.d.u.c.t.
61e843a0  N.a.m.e...M.i.
61e843b0  c.r.o.s.o.f.t. .
61e843c0  R.e.s.e.a.r.c.h.
61e843d0  .D.e.t.o.u.r.s.
61e843e0  .P.a.c.k.a.g.e.
61e843f0  ...j.#...P.r.o.
61e84400  d.u.c.t.V.e.r.s.
```

```

61e84410 i.o.n...P.r.o.f.
61e84420 e.s.s.i.o.n.a.l.
61e84430 .V.e.r.s.i.o.n.
61e84440 .2...1. .B.u.i.
61e84450 l.d._.2.1.0....
61e84460 D....V.a.r.F.i.

...
...
...
...
```

0:000> du 61e843a0+C  
61e843ac "Microsoft Research Detours Packa"  
61e843ec "ge"

We can also see that *3rdPartyHook* module imports this library and lots of *kernel32* API related to memory allocation, file mapping and loading DLLs (see **No Component Symbols** pattern, page 734):

```

0:000> !dh 60000000
...
...
...
OPTIONAL HEADER VALUES
  10B magic #
  8.00 linker version
 18000 size of code
 F000 size of initialized data
    0 size of uninitialized data
 13336 address of entry point
 1000 base of code
    ----- new -----
60000000 image base
  1000 section alignment
  1000 file alignment
    2 subsystem (Windows GUI)
  4.00 operating system version
  0.00 image version
  4.00 subsystem version
 29000 size of image
  1000 size of headers
  3376F checksum
00100000 size of stack reserve
00001000 size of stack commit
00100000 size of heap reserve
00001000 size of heap commit
    0 [      0] address [size] of Export Directory
  218CC [     8C] address [size] of Import Directory
  25000 [    5F4] address [size] of Resource Directory
    0 [      0] address [size] of Exception Directory
  28000 [   19E0] address [size] of Security Directory
  26000 [   2670] address [size] of Base Relocation Directory
 19320 [     1C] address [size] of Debug Directory
    0 [      0] address [size] of Description Directory
    0 [      0] address [size] of Special Directory
    0 [      0] address [size] of Thread Storage Directory
 1F3C0 [     40] address [size] of Load Configuration Directory
    0 [      0] address [size] of Bound Import Directory
```

```
19000 [ 2B0] address [size] of Import Address Table Directory
 0 [     0] address [size] of Delay Import Directory
 0 [     0] address [size] of COR20 Header Directory
 0 [     0] address [size] of Reserved Directory
...
...
...
0:000> dds 60000000+19000 60000000+19000+2B0
60019064 7c82b0dc ntdll!RtlReAllocateHeap
60019068 77e4ec39 kernel32!HeapDestroy
6001906c 77e41fba kernel32!GetSystemTimeAsFileTime
60019070 77e619d1 kernel32!GetTickCount
60019074 77e69577 kernel32!QueryPerformanceCounter
60019078 7c82a9be ntdll!RtlSizeHeap
6001907c 77e82060 kernel32!SetUnhandledExceptionFilter
60019080 77e7690d kernel32!UnhandledExceptionFilter
60019084 77e42004 kernel32!TerminateProcess
60019088 7c82a136 ntdll!RtlRestoreLastWin32Error
6001908c 77e77a5f kernel32!SuspendThread
60019090 77e76a26 kernel32!SetThreadContext
60019094 77e77ae3 kernel32!GetThreadContext
60019098 77e73347 kernel32!FlushInstructionCache
6001909c 77e5f38b kernel32!ResumeThread
600190a0 77e616a8 kernel32!InterlockedCompareExchange
600190a4 77e645a9 kernel32!VirtualAlloc
600190a8 77e41fe3 kernel32!VirtualProtect
600190ac 77e66ed1 kernel32!VirtualQuery
600190b0 77e44960 kernel32!GetLogicalDriveStringsA
600190b4 77eab401 kernel32!GetVolumeNameForVolumeMountPointA
600190b8 77e6794d kernel32!GetACP
600190bc 77e6f3cf kernel32!GetLocaleInfoA
600190c0 77e622b7 kernel32!GetThreadLocale
600190c4 77e69d74 kernel32!GetVersionExA
600190c8 77e4beab kernel32!RaiseException
600190cc 77e60037 kernel32!GetSystemDirectoryA
600190d0 77e52bf4 kernel32!GetWindowsDirectoryA
600190d4 77e5c7a8 kernel32!lstrcmpA
600190d8 77e46c99 kernel32!OutputDebugStringA
600190dc 77e5bd7d kernel32!CreateEventA
600190e0 77e62311 kernel32!SetEvent
600190e4 77e51281 kernel32!ExpandEnvironmentStringsA
600190e8 77e9f365 kernel32!MoveFileA
600190ec 77e5da00 kernel32!IsDebuggerPresent
600190f0 77e9e4b1 kernel32!QueryDosDeviceA
600190f4 7c829e08 ntdll!RtlGetLastWin32Error
600190f8 77e63d7a kernel32!GetProcAddress
600190fc 77e41dc6 kernel32!LoadLibraryA
60019100 7c829e17 ntdll!RtlFreeHeap
60019104 77e62419 kernel32!LocalFree
60019108 7c829fd6 ntdll!RtlAllocateHeap
6001910c 77e63ec7 kernel32!GetProcessHeap
60019110 77ea2186 kernel32!VerifyVersionInfoA
60019114 7c81379f ntdll!VerSetConditionMask
60019118 77e63143 kernel32!WideCharToMultiByte
6001911c 77e70550 kernel32!SizeofResource
```

```

60019120 77e6b11b kernel32!SetHandleCount
60019124 77e69bf9 kernel32!LoadResource
60019128 77e511e1 kernel32!FindResourceA
6001912c 77e7388c kernel32!FindResourceExA
60019130 77e5be30 kernel32!lstrlenA
60019134 77e424de kernel32!Sleep
60019138 77ea2cb1 kernel32!WaitNamedPipeA
6001913c 77e63e6f kernel32!CloseHandle
60019140 77e5e123 kernel32!OpenEventA
60019144 77e622c9 kernel32!lstrlenW
60019148 77e62fc7 kernel32!GetCurrentThreadId
6001914c 77e5fdd4 kernel32!OpenProcess
60019150 77e63c78 kernel32!GetCurrentProcessId
60019154 7c81a3ab ntdll!RtlLeaveCriticalSection
60019158 7c81a360 ntdll!RtlEnterCriticalSection
6001915c 77e4cabf kernel32!GetComputerNameA
60019160 77e6f032 kernel32!ProcessIdToSessionId
60019164 77e645ff kernel32!GetModuleFileNameA
60019168 77e6474a kernel32!GetModuleHandleA
6001916c 77e62f9d kernel32!GetCurrentProcess
60019170 77e49968 kernel32!GetCurrentDirectoryA
60019174 77e61c7b kernel32!WaitForSingleObject
60019178 77e63868 kernel32!GetCurrentThread
6001917c 7c82c988 ntdll!RtlDeleteCriticalSection
60019180 77e67861 kernel32!InitializeCriticalSection
60019184 77e6b1a1 kernel32!FreeLibrary
60019188 77e63f41 kernel32!UnmapViewOfFile
6001918c 77e643f1 kernel32!MapViewOfFile
60019190 77e6b65f kernel32!OpenFileMappingA
60019194 77e61694 kernel32!InterlockedExchange
60019198 77e4d2fb kernel32!DeleteFileA
...
...
...
60019294 00000000
60019298 61e81000 detoured!Detoured
6001929c 00000000
...
...
...

```

This warrants the suspicion that *3rdPartyHook* somehow optimized the virtual address space for its own purposes, and this resulted in more fragmented virtual address space.

## Comments

**!address -summary** WinDbg command also gives various output about region type usage and the largest available blocks per region type.

## Physical Memory

Sometimes there is not enough physical memory, and a system experiences the so-called **disk or page file thrashing** trying to resolve page faults. This can be seen in some memory dumps coming from frozen environments showing signs of double traps in running threads, the first trap is a normal memory access fault (shown in bold) and the second is forced NMI bugcheck<sup>115</sup> to save a memory dump (shown in bold italics):

```
1: kd> .bugcheck
Bugcheck code 00000080
Arguments 004f4454 00000000 00000000 00000000

1: kd> !thread
THREAD 88939b20 Cid 360.378 Teb: 7ffdb000 Win32Thread: a20a7ac8 RUNNING
IRP List:
 86be9e68: (0006,0100) Flags: 00000070 Mdl: 00000000
 88939e68: (0006,0100) Flags: 00000070 Mdl: 00000000
 88939128: (0006,0100) Flags: 00000070 Mdl: 00000000
Not impersonating
Owning Process 889456e0
Wait Start TickCount      2357431      Elapsed Ticks: 9
Context Switch Count      18267           LargeStack
UserTime                  0:00:08.0218
KernelTime                0:12:28.0109
Start Address KERNEL32!BaseThreadStartThunk (0x7c57b740)
Win32 Start Address msafd!SockAsyncThread (0x74fd3113)
Stack Init be9e000 Current be9db60 Base be9e000 Limit be9b000 Call 0
Priority 11 BasePriority 11 PriorityDecrement 0 DecrementCount 0

ChildEBP RetAddr
8904aff0 80469211 hal!HalHandleNMI+0x193
8904aff0 80438621 nt!KiTrap02+0x41
bef9dc10 8043799a nt!MiTrimWorkingSet+0xa7
bef9dc38 804378ec nt!MiDoReplacement+0x2e
bef9dc50 804453cf nt!MiLocateAndReserveWsle+0x1e
bef9dc68 804444e0 nt!MiAddValidPageToWorkingSet+0x89
bef9dc8c 804443a2 nt!MiCompleteProtoPteFault+0xf6
bef9dc8b 804436e8 nt!MiResolveProtoPteFault+0x160
bef9dcfc 8044cccd0 nt!MiDispatchFault+0xfc
bef9dd4c 8046b063 nt!MmAccessFault+0xd1c
bef9dd4c 74fd31e0 nt!KiTrap0E+0xc7
016effb4 7c57b3bc msafd!SockAsyncThread+0xcd
016effec 00000000 KERNEL32!BaseThreadStart+0x52
```

<sup>115</sup> Bugchecks Depicted, NMI\_HARDWARE\_FAILURE, Memory Dump Analysis Anthology, Volume 1, page 135

If we check virtual memory stats we see the low number of available pages:

```
1: kd> !vm

*** Virtual Memory Usage ***
Physical Memory: 524165 ( 2096660 Kb)
Page File: \??\C:\pagefile.sys
  Current: 4190208Kb Free Space: 3298704Kb
  Minimum: 4190208Kb Maximum: 4190208Kb
Page File: \??\E:\pagefile.sys
  Current: 4190208Kb Free Space: 3339860Kb
  Minimum: 4190208Kb Maximum: 4190208Kb
Available Pages: 1098 ( 4392 Kb)
ResAvail Pages: 410646 ( 1642584 Kb)
Modified Pages: 282384 ( 1129536 Kb)
NonPagedPool Usage: 10046 ( 40184 Kb)
NonPagedPool Max: 68609 ( 274436 Kb)
PagedPool 0 Usage: 15391 ( 61564 Kb)
PagedPool 1 Usage: 1906 ( 7624 Kb)
PagedPool 2 Usage: 1925 ( 7700 Kb)
PagedPool 3 Usage: 1937 ( 7748 Kb)
PagedPool 4 Usage: 1892 ( 7568 Kb)
PagedPool Usage: 23051 ( 92204 Kb)
PagedPool Maximum: 87040 ( 348160 Kb)
Shared Commit: 16867 ( 67468 Kb)
Special Pool: 0 ( 0 Kb)
Free System PTEs: 65288 ( 261152 Kb)
Shared Process: 38655 ( 154620 Kb)
PagedPool Commit: 23051 ( 92204 Kb)
Driver Commit: 1060 ( 4240 Kb)
Committed pages: 1049592 ( 4198368 Kb)
Commit limit: 2580155 (10320620 Kb)
[...]
```

In W2K dumps we can also see locking on a working set resource (the name is guessed from **Ws** shortcut here):

```
1: kd> !locks
**** DUMP OF ALL RESOURCE OBJECTS ****

Resource @ nt!MmSystemWsLock (0x804869c0)    Exclusively owned
  Contention Count = 33083
  NumberOfExclusiveWaiters = 237
[...]
```

and the huge number of threads in the Ready state for every thread priority.

Looking at the current process owning the running thread shows the large number of page faults and increased kernel CPU time compared to time spent in user mode:

```
1: kd> !process 889456e0
PROCESS 889456e0 SessionId: 0 Cid: 0360 Peb: 7ffdf000 ParentCid: 01a8
  DirBase: 102af000 ObjectTable: 88945c08 TableSize: 622.
  Image: Application.EXE
  VadRoot 88944468 Clone 0 Private 838. Modified 30691412. Locked 188.
  DeviceMap 89049288
    Token e28db550
    ElapsedTime 10:13:30.0684
    UserTime 0:00:12.0578
    KernelTime 0:12:38.0625
    QuotaPoolUsage[PagedPool] 31568
    QuotaPoolUsage[NonPagedPool] 68266
    Working Set Sizes (now,min,max) (49, 50, 345) (196KB, 200KB, 1380KB)
    PeakWorkingSetSize 1956
    VirtualSize 131 Mb
    PeakVirtualSize 131 Mb
    PageFaultCount 46180598
    MemoryPriority BACKGROUND
    BasePriority 10
    CommitCharge 1247
```

## PTE

In order to maintain virtual to physical address translation, OS needs page tables. These tables occupy memory too. If memory is not enough for new tables the system will fail to create processes, allocate I/O buffers and memory from pools. We might see the following diagnostic message from WinDbg:

```
4: kd> !vm

*** Virtual Memory Usage ***
Physical Memory: 851422 ( 3405688 Kb)
Page File: \??\C:\pagefile.sys
  Current: 2095104 Kb  Free Space: 2081452 Kb
  Minimum: 2095104 Kb  Maximum: 4190208 Kb
Available Pages: 683464 ( 2733856 Kb)
ResAvail Pages: 800927 ( 3203708 Kb)
Locked IO Pages: 145 (      580 Kb)
Free System PTEs: 23980 ( 95920 Kb)

***** 356363 system PTE allocations have failed *****

Free NP PTEs: 6238 ( 24952 Kb)
Free Special NP: 0 ( 0 Kb)
Modified Pages: 482 ( 1928 Kb)
Modified PF Pages: 482 ( 1928 Kb)
NonPagedPool Usage: 18509 ( 74036 Kb)
NonPagedPool Max: 31970 ( 127880 Kb)
PagedPool 0 Usage: 8091 ( 32364 Kb)
PagedPool 1 Usage: 2495 ( 9980 Kb)
PagedPool 2 Usage: 2580 ( 10320 Kb)
PagedPool 3 Usage: 2552 ( 10208 Kb)
PagedPool 4 Usage: 2584 ( 10336 Kb)
PagedPool Usage: 18302 ( 73208 Kb)
PagedPool Maximum: 39936 ( 159744 Kb)

***** 48530 pool allocations have failed *****

Shared Commit: 5422 ( 21688 Kb)
Special Pool: 0 ( 0 Kb)
Shared Process: 5762 ( 23048 Kb)
PagedPool Commit: 18365 ( 73460 Kb)
Driver Commit: 2347 ( 9388 Kb)
Committed pages: 129014 ( 516056 Kb)
Commit limit: 1342979 ( 5371916 Kb)
```

We can also see another diagnostic message about pool allocation failures which may be the consequence of PTE allocation failures.

The cause of system PTE allocation failures might be the incorrect value of SystemPages registry key that needs to be adjusted as explained in the following TechNet article: The number of free page table entries is low, which can cause system instability<sup>116</sup>.

Another cause would be /3GB boot option on x86 systems especially used for hosting terminal sessions. This case is explained in the following blog post which also shows how to detect /3GB kernel and complete memory dumps: Consequences of running 3GB and PAE together<sup>117</sup>.

In our case, the system was booted with /3GB:

```
4: kd> vertarget
Windows Server 2003 Kernel Version 3790 (Service Pack 2) MP (8 procs) Free x86 compatible
Product: Server, suite: Enterprise TerminalServer
Built by: 3790.srv03_sp2_gdr.070304-2240
Kernel base = 0xe0800000 PsLoadedModuleList = 0xe08af9c8
Debug session time: Fri Feb 1 09:10:17.703 2008 (GMT+0)
System Uptime: 6 days 17:14:45.528
```

Normal Windows 2003 systems have different kernel base address which can be checked from Reference Stack Traces for Windows Server 2003 (Appendix A):

```
kd> vertarget
Windows Server 2003 Kernel Version 3790 (Service Pack 2) UP Free x86 compatible
Product: Server, suite: Enterprise TerminalServer SingleUserTS
Built by: 3790.srv03_sp2_rtm.070216-1710
Kernel base = 0x80800000 PsLoadedModuleList = 0x8089ffa8
Debug session time: Wed Jan 30 17:54:13.390 2008 (GMT+0)
System Uptime: 0 days 0:30:12.000
```

## Comments

---

**!sysptes** has been fixed, and we can use public symbols<sup>118</sup>.

According to Windows Internals 6<sup>th</sup> Edition Part 2, page 236, we can create DWORD TrackPtes value in the HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management key and set it to 1, and then see allocation history by using **!sysptes 4** WinDbg command.

---

<sup>116</sup> <http://technet.microsoft.com/en-us/library/aa995783.aspx>

<sup>117</sup> [http://blogs.technet.com/brad\\_rutkowski/archive/2007/04/03/consequences-of-running-3gb-and-pae-together.aspx](http://blogs.technet.com/brad_rutkowski/archive/2007/04/03/consequences-of-running-3gb-and-pae-together.aspx)

<sup>118</sup> [http://blogs.technet.com/brad\\_rutkowski/archive/2008/02/21/i-pte-the-fool-sysptes-4-works-in-vista-sp1-ws08.aspx](http://blogs.technet.com/brad_rutkowski/archive/2008/02/21/i-pte-the-fool-sysptes-4-works-in-vista-sp1-ws08.aspx)

## Region

While working on **Insufficient Memory** pattern for stack trace database (page 563) we noticed the expansion of certain memory regions. Of course, after some time expanding region consumes remaining free or reserved space available before some other region. Generalizing from this, we may say there can be **Insufficient Memory** pattern variant for any expanding region. Region expansion may also be implemented via its move into some other position in memory virtual address space. This movement also has its limits. For example, we created this modeling application and found out it stops reallocating memory long before it reaches 2,000,000,000 byte size:

```
int _tmain(int argc, _TCHAR* argv[])
{
    int i = 100000000;
    void *p = malloc(i);
    for (i = 200000000; i < 2000000000; i+=100000000)
    {
        p = realloc(p, i);
        getc(stdin);
    }
    return 0;
}
```

We took memory dumps after each loop iteration and after 6 or 8 iterations the memory size was constant, and there were no further reallocations:

```
0:000> !heap -s
[...]
Virtual block: 000000006370000 - 000000006370000 (size 0000000000000000)
[...]

0:000> !address
[...]


| <u>: Start</u> | <u>End</u> | <u>Size</u>                                                                                               |
|----------------|------------|-----------------------------------------------------------------------------------------------------------|
| + 0`00550000   | 0`06370000 | 0`05e20000 MEM_FREE PAGE_NOACCESS Free                                                                    |
| + 0`06370000   | 0`1222d000 | 0`0beb0d00 MEM_PRIVATE MEM_COMMIT PAGE_READWRITE Heap [ID: 0; Handle: 000000000310000; Type: Large block] |
| + 0`1222d000   | 0`77710000 | 0`654e3000 MEM_FREE PAGE_NOACCESS Free                                                                    |
| + 0`77710000   | 0`77711000 | 0`00001000 MEM_IMAGE MEM_COMMIT PAGE_READONLY Image [kernel32; "C:\windows\system32\kernel32.dll"]        |


[...]

0:000> !heap -s
[...]
Virtual block: 0000000012230000 - 0000000012230000 (size 0000000000000000)
[...]

0:000> !address
[...]


| <u>: Start</u> | <u>End</u> | <u>Size</u>                                                                                               |
|----------------|------------|-----------------------------------------------------------------------------------------------------------|
| + 0`005d0000   | 0`12230000 | 0`11c60000 MEM_FREE PAGE_NOACCESS Free                                                                    |
| + 0`12230000   | 0`2404b000 | 0`11e1b000 MEM_PRIVATE MEM_COMMIT PAGE_READWRITE Heap [ID: 0; Handle: 000000000310000; Type: Large block] |
| + 0`2404b000   | 0`77710000 | 0`536c5000 MEM_FREE PAGE_NOACCESS Free                                                                    |
| + 0`77710000   | 0`77711000 | 0`00001000 MEM_IMAGE MEM_COMMIT PAGE_READONLY Image [kernel32; "C:\windows\system32\kernel32.dll"]        |


[...]

0:000> !heap -s
[...]
Virtual block: 0000000024050000 - 0000000024050000 (size 0000000000000000)
[...]
```

```
0:000> !address
[...]
+ 0`00590000 0`24050000 0`23ac0000 MEM_FREE PAGE_NOACCESS Free
+ 0`24050000 0`3bd9000 0`17d79000 MEM_PRIVATE MEM_COMMIT PAGE_READWRITE Heap [ID: 0; Handle: 0000000000310000; Type: Large block]
+ 0`3bd9000 0`77710000 0`3b947000 MEM_FREE PAGE_NOACCESS Free
+ 0`77710000 0`77711000 0`00001000 MEM_IMAGE MEM_COMMIT PAGE_READONLY Image [kernel32; "C:\windows\system32\kernel32.dll"]
[...]
```

We skip a few iterations and finally come to a region that does not move and not increase:

```
0:000> !heap -s
[...]
Virtual block: 0000000041d30000 - 0000000041d30000 (size 0000000000000000)
[...]

0:000> !address
[...]
+ 0`006c0000 0`41d30000 0`41670000 MEM_FREE PAGE_NOACCESS Free
+ 0`41d30000 0`6b8c3000 0`29b93000 MEM_PRIVATE MEM_COMMIT PAGE_READWRITE Heap [ID: 0; Handle: 0000000000310000; Type: Large block]
+ 0`6b8c3000 0`77710000 0`0be4d000 MEM_FREE PAGE_NOACCESS Free
+ 0`77710000 0`77711000 0`00001000 MEM_IMAGE MEM_COMMIT PAGE_READONLY Image [kernel32; "C:\windows\system32\kernel32.dll"]
[...]
```

## Reserved Virtual Memory

Allocated dynamic memory such as process heap can remain reserved after deallocation, and its virtual memory region might become unavailable for usage. One example of this we encountered while debugging a .NET service. During peak usage, it reported various out-of-memory events, but its managed heap was healthy and didn't consume much. However, its process heap statistics showed a large reserved heap segment missing in a similar memory dump from a development environment. Remaining allocated entries in that heap segment contained a specific **Module Hint** (page 696) that allowed us to suggest removing a 3rd-party product from a production environment.

In order to provide a proof of that possible scenario of reserved heap regions we created a special modeling application:

```
int _tmain(int argc, _TCHAR* argv[])
{
    static char *pAlloc[1000000];
    for (int i = 0; i < 1000000; i++)
    {
        pAlloc[i] = (char *)malloc (1000);
    }
    getc(stdin);
    for (int i = 0; i < 1000000; i++)
    {
        free(pAlloc[i]);
    }
    getc(stdin);
    return 0;
}
```

Here is the debugging log:

```
0:001> .symfix c:\mss
0:001> .reload
Reloading current modules
.....
```

After allocation:

```
0:001> !heap -s
LFH Key : 0x156356e0
Termination on corruption : ENABLED
Heap      Flags   Reserv  Commit   Virt     Free     List   UCR   Virt Lock Fast
(k)       (k)     (k)     (k)      length   blocks cont. heap
-----
00520000 00000002 1024     112     1024     8       1       1     0     0     LFH
007e0000 00001002 1019328 1012444 1019328 131     68      67     0     0     LFH
-----
```

```
0:001> g
(1588.14b0): Break instruction exception - code 80000003 (first chance)
eax=7efda000 ebx=00000000 ecx=00000000 edx=770ff85a esi=00000000 edi=00000000
eip=7707000c esp=00f0f7e4 ebp=00f0f810 iopl=0 nv up ei pl zr na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000246
ntdll!DbgBreakPoint:
7707000c cc int 3
```

After deallocation:

```
0:001> !heap -s
LFH Key : 0x156356e0
Termination on corruption : ENABLED
Heap Flags Reserv Commit Virt Free List UCR Virt Lock Fast
(k) (k) (k) (k) length blocks cont. heap
-----
00520000 00000002 1024 112 1024 8 1 1 0 0 LFH
007e0000 00001002 1019328 73040 1019328 71365 419 165 0 0 LFH
External fragmentation 97 % (419 free blocks)
Virtual address fragmentation 92 % (165 uncommitted ranges)
-----

0:001> !address -summary
--- Usage Summary -----
RgnCount Total Size %ofBusy %ofTotal
Free 26 3fbe7000 (1019.902 Mb) 49.80%
<unclassified> 752 3f8ec000 (1016.922 Mb) 98.92% 49.66%
Image 41 76b000 ( 7.418 Mb) 0.72% 0.36%
Stack 6 200000 ( 2.000 Mb) 0.19% 0.10%
MemoryMappedFile 8 1af000 ( 1.684 Mb) 0.16% 0.08%
TEB 2 2000 ( 8.000 kb) 0.00% 0.00%
PEB 1 1000 ( 4.000 kb) 0.00% 0.00%

--- Type Summary (for busy) -----
RgnCount Total Size %ofBusy %ofTotal
MEM_PRIVATE 734 3f8a2000 (1016.633 Mb) 98.89% 49.64%
MEM_IMAGE 68 9b8000 ( 9.719 Mb) 0.95% 0.47%
MEM_MAPPED 8 1af000 ( 1.684 Mb) 0.16% 0.08%

--- State Summary -----
RgnCount Total Size %ofBusy %ofTotal
MEM_FREE 26 3fbe7000 (1019.902 Mb) 49.80%
MEM_RESERVE 374 3f6e8000 (1014.906 Mb) 98.72% 49.56%
MEM_COMMIT 436 d21000 ( 13.129 Mb) 1.28% 0.64%

--- Protect Summary (for commit) -
RgnCount Total Size %ofBusy %ofTotal
PAGE_READWRITE 383 725000 ( 7.145 Mb) 0.69% 0.35%
PAGE_EXECUTE_READ 10 414000 ( 4.078 Mb) 0.40% 0.20%
PAGE_READONLY 29 1cd000 ( 1.801 Mb) 0.18% 0.09%
PAGE_WRITECOPY 10 12000 ( 72.000 kb) 0.01% 0.00%
PAGE_READWRITE|PAGE_GUARD 4 9000 ( 36.000 kb) 0.00% 0.00%
```

--- Largest Region by Usage -----	Base Address -----	Region Size -----
<u>Free</u>	3f0c0000	33050000 ( 816.313 Mb)
<unclassified>	158a1000	fcf000 ( 15.809 Mb)
Image	1083000	3d1000 ( 3.816 Mb)
Stack	200000	fd000 (1012.000 kb)
MemoryMappedFile	7efe5000	fb000 (1004.000 kb)
TEB	7efda000	1000 ( 4.000 kb)
PEB	7efde000	1000 ( 4.000 kb)

We see that free memory available for allocation was only 816 Mb.

## Session Pool

Although we briefly mentioned session pool in **Insufficient Memory** (kernel pool) pattern (page 535) we factored it into a separate (sub)pattern to provide WinDbg commands for analysis of possible leaks. The following output shows the sequence of commands that gives an idea although the example itself was taken from a healthy dump so no highlights in bold font (we had seen leaks in session pool happening mostly in 32-bit cases):

```
1: kd> !vm 4

Terminal Server Memory Usage By Session:

Session ID 0 @ fffff8800324d000:
Paged Pool Usage:      4128K
Commit Usage:          7488K

Session ID 1 @ fffff88002f65000:
Paged Pool Usage:      32852K
Commit Usage:          36488K

1: kd> !session
Sessions on machine: 2
Valid Sessions: 0 1
Error in reading current session

1: kd> !session -s 1
Sessions on machine: 2
Implicit process is now ffffffa80`07d79730
Using session 1

1: kd> !poolused 8
Sorting by Session Tag

Pool Used:
NonPaged          Paged
Tag    Allocs     Used    Allocs     Used
TOTAL        4      4208      9500  33475120
[...]
```

## Stack Trace Database

Once we have seen a sequence of process memory dumps with the largest one almost 4GB. They were all saved from the process with growing memory consumption from 200MB initially. Initially, we suspected process heap **Memory Leak** (page 650). However, heap statistics (`!heap -s`) was normal. There were not even large block allocations<sup>119</sup>. The dumps were also supplied with UMDH logs, but their difference only showed **Memory Fluctuation** (page 634) and not increase. **Stack Trace Collection** (page 933) revealed one **Spiking Thread** (page 885) was logging a heap allocation into user mode stack trace database. We could also see that it was **Distributed Spike** (page 243). Inspection of address space showed a large number of sequential regions of the same size with **Stack Trace Database** (page 919) entries inside. So we concluded that it was stack trace logging **Instrumentation Side Effect** (page 520) and advised to limit stack backtrace size in `gflags.exe`.

To make sure we understood that problem correctly, we decided to model it. We did not come to the same results probably due to different logging implementation, but the memory dumps clearly show the possibility of **Insufficient Memory** pattern variant. Here's the source code:

```
void foo20 (int size)
{
    free(malloc(size));
}

#define FOO(x,y) void foo##x (int size) { foo##y(size); }

FOO(19,20)
FOO(18,19)
FOO(17,18)
FOO(16,17)
FOO(15,16)
FOO(14,15)
FOO(13,14)
FOO(12,13)
FOO(11,12)
FOO(10,11)
FOO(9,10)
FOO(8,9)
FOO(7,8)
FOO(6,7)
FOO(5,6)
FOO(4,5)
FOO(3,4)
FOO(2,3)
FOO(1,2)

typedef void (*PFN) (int);

#define ARRSZ 20
```

---

<sup>119</sup> Models of Software Behaviour, Memory Leak (Process Heap) Pattern, Memory Dump Analysis Anthology, Volume 5, page 315

```

PFN pfnArr[ARRSZ] = {foo1, foo2, foo3, foo4, foo5, foo6, foo7,
                     foo8, foo9, foo10, foo11, foo12, foo13, foo14,
                     foo15, foo16, foo17, foo18, foo19, foo20};

int _tmain(int argc, _TCHAR* argv[])
{
    int i;
    for (i = 1; i < 1000000000; ++i)
    {
        pfnArr[i%ARRSZ](i);
    }
    Sleep(-1);
    return 0;
}

```

It allocates and then frees heap entries of different size from 1 byte to 1,000,000,000 bytes all with different 20 possible stack traces. We choose different stack traces to increase the number of different *{size, stack backtrace}* pairs as several allocations of similar size having the same stack trace may be recorded only once in the database. We emulate different stack traces by calling different entries in *pfnArr*. Each call then leads to *foo20*, but the resulting stack trace depth is different. We also enabled “*Create user mode stack trace database*” checkbox in *gflags.exe* for our application called *AllocFree.exe*.

Then we see the expansion of **Stack Trace Database** regions (addresses are different because memory dumps were taken from different application runs):

```

0:000> !address
[...]


| <u>Start</u> | <u>End</u> | <u>Size</u> |                                                                    |
|--------------|------------|-------------|--------------------------------------------------------------------|
| + 0`00240000 | 0`00312000 | 0`000d2000  | MEM_PRIVATE MEM_COMMIT PAGE_READWRITE Other [Stack Trace Database] |
| 0`00312000   | 0`01a37000 | 0`01725000  | MEM_PRIVATE MEM_RESERVE Other [Stack Trace Database]               |
| 0`01a37000   | 0`01a40000 | 0`00009000  | MEM_PRIVATE MEM_COMMIT PAGE_READWRITE Other [Stack Trace Database] |


0:000> !address
[...]


| <u>Start</u> | <u>End</u> | <u>Size</u>       |                                                                    |
|--------------|------------|-------------------|--------------------------------------------------------------------|
| + 0`001b0000 | 0`0188c000 | <u>0`016dc000</u> | MEM_PRIVATE MEM_COMMIT PAGE_READWRITE Other [Stack Trace Database] |
| 0`0188c000   | 0`0188d000 | 0`00001000        | MEM_PRIVATE MEM_RESERVE Other [Stack Trace Database]               |
| 0`0188d000   | 0`019b0000 | 0`00123000        | MEM_PRIVATE MEM_COMMIT PAGE_READWRITE Other [Stack Trace Database] |


```

Heap stays the same:

```
0:000> !heap -s
NtGlobalFlag enables following debugging aids for new heaps:
stack back traces
LFH Key : 0x000000f841c4f9c0
Termination on corruption : ENABLED
      Heap   Flags  Reserv Commit Virt  Free  List   UCR  Virt  Lock  Fast
      (k)     (k)    (k)   (k)   (k)  length
-----+
0000000001a40000 08000002  4096  1444  4096  1164    4     3    0    0  LFH
External fragmentation 80 % (4 free blocks)
0000000000010000 08008000    64     4    64     1     1    1    0    0
0000000000020000 08008000    64    64    64    61     1    1    0    0
-----+
```

```
0:000> !heap -s
NtGlobalFlag enables following debugging aids for new heaps:
stack back traces
LFH Key : 0x000000473a639107
Termination on corruption : ENABLED
      Heap   Flags  Reserv Commit Virt  Free  List   UCR  Virt  Lock  Fast
      (k)     (k)    (k)   (k)   (k)  length
-----+
00000000019c0000 08000002  4096  1444  4096  1164    4     3    0    0
LFH
External fragmentation 80 % (4 free blocks)
0000000000010000 08008000    64     4    64     1     1    1    0    0
0000000000020000 08008000    64    64    64    61     1    1    0    0
-----+
```

However, we see the thread consuming much CPU (**Spiking Thread**, page 888) and that it was caught while logging stack backtrace:

```
0:000> kc
Call Site
ntdll!RtlpStdLogCapturedStackTrace
ntdll!RtlStdLogStackTrace
ntdll!RtlLogStackBackTraceEx
ntdll!RtlpAllocateHeap
ntdll!RtlAllocateHeap
AllocFree!_heap_alloc
AllocFree!malloc
AllocFree!foo20
AllocFree!foo19
AllocFree!foo18
AllocFree!foo17
AllocFree!foo16
AllocFree!foo15
AllocFree!foo14
AllocFree!foo13
AllocFree!foo12
AllocFree!foo11
AllocFree!foo10
AllocFree!foo9
```

```

AllocFree!foo8
AllocFree!foo7
AllocFree!foo6
AllocFree!foo5
AllocFree!foo4
AllocFree!foo3
AllocFree!foo2
AllocFree!foo1
AllocFree!wmain
AllocFree!__tmainCRTStartup
kernel32!BaseThreadInitThunk
ntdll!RtlUserThreadStart

0:000> !runaway f
User Mode Time
 Thread      Time
 0:53b8      0 days 3:22:02.354
Kernel Mode Time
 Thread      Time
 0:53b8      0 days 0:20:39.022
Elapsed Time
 Thread      Time
 0:53b8      0 days 10:11:23.596

```

If we dump some portion of the region we see recorded stack backtraces:

```

0:000> dps 0`0188c000-200 L200/8
00000000`0188be00 00000000`77891142 ntdll!RtlpAllocateHeap+0x33bd
00000000`0188be08 00000000`778834d8 ntdll!RtlAllocateHeap+0x16c
00000000`0188be10 00000001`3fcc13cb AllocFree!malloc+0x5b
00000000`0188be18 00000001`3fcc1015 AllocFree!foo20+0x15
00000000`0188be20 00000001`3fcc1041 AllocFree!foo19+0x11
00000000`0188be28 00000001`3fcc1061 AllocFree!foo18+0x11
00000000`0188be30 00000001`3fcc12e3 AllocFree!wmain+0x53
00000000`0188be38 00000001`3fcc156c AllocFree!__tmainCRTStartup+0x144
00000000`0188be40 00000000`777259ed kernel32!BaseThreadInitThunk+0xd
00000000`0188be48 00000000`7785c541 ntdll!RtlUserThreadStart+0x1d
00000000`0188be50 00000000`0188b1d0
00000000`0188be58 0009457d`00024fff
00000000`0188be60 00000000`77891142 ntdll!RtlpAllocateHeap+0x33bd
00000000`0188be68 00000000`778834d8 ntdll!RtlAllocateHeap+0x16c
00000000`0188be70 00000001`3fcc13cb AllocFree!malloc+0x5b
00000000`0188be78 00000001`3fcc1015 AllocFree!foo20+0x15
00000000`0188be80 00000001`3fcc1041 AllocFree!foo19+0x11
00000000`0188be88 00000001`3fcc12e3 AllocFree!wmain+0x53
00000000`0188be90 00000001`3fcc156c AllocFree!__tmainCRTStartup+0x144
00000000`0188be98 00000000`777259ed kernel32!BaseThreadInitThunk+0xd
00000000`0188bea0 00000000`7785c541 ntdll!RtlUserThreadStart+0x1d
00000000`0188bea8 00000000`00000000
00000000`0188beb0 00000000`0188b230
00000000`0188beb8 0008457e`00023fff
00000000`0188bec0 00000000`77891142 ntdll!RtlpAllocateHeap+0x33bd
00000000`0188bec8 00000000`778834d8 ntdll!RtlAllocateHeap+0x16c
00000000`0188bed0 00000001`3fcc13cb AllocFree!malloc+0x5b
00000000`0188bed8 00000001`3fcc1015 AllocFree!foo20+0x15
00000000`0188bee0 00000001`3fcc12e3 AllocFree!wmain+0x53

```

```
00000000`0188bee8 00000001`3fcc156c AllocFree!__tmainCRTStartup+0x144
00000000`0188bef0 00000000`777259ed kernel32!BaseThreadInitThunk+0xd
00000000`0188bef8 00000000`7785c541 ntdll!RtlUserThreadStart+0x1d
00000000`0188bf00 00000000`0188b280
00000000`0188bf08 001b457f`0002dff
00000000`0188bf10 00000000`77891142 ntdll!RtlpAllocateHeap+0x33bd
00000000`0188bf18 00000000`778834d8 ntdll!RtlAllocateHeap+0x16c
00000000`0188bf20 00000001`3fcc13cb AllocFree!malloc+0x5b
00000000`0188bf28 00000001`3fcc1015 AllocFree!foo20+0x15
00000000`0188bf30 00000001`3fcc1041 AllocFree!foo19+0x11
00000000`0188bf38 00000001`3fcc1061 AllocFree!foo18+0x11
00000000`0188bf40 00000001`3fcc1081 AllocFree!foo17+0x11
00000000`0188bf48 00000001`3fcc10a1 AllocFree!foo16+0x11
00000000`0188bf50 00000001`3fcc10c1 AllocFree!foo15+0x11
00000000`0188bf58 00000001`3fcc10e1 AllocFree!foo14+0x11
00000000`0188bf60 00000001`3fcc1101 AllocFree!foo13+0x11
00000000`0188bf68 00000001`3fcc1121 AllocFree!foo12+0x11
00000000`0188bf70 00000001`3fcc1141 AllocFree!foo11+0x11
00000000`0188bf78 00000001`3fcc1161 AllocFree!foo10+0x11
00000000`0188bf80 00000001`3fcc1181 AllocFree!foo9+0x11
00000000`0188bf88 00000001`3fcc11a1 AllocFree!foo8+0x11
00000000`0188bf90 00000001`3fcc11c1 AllocFree!foo7+0x11
00000000`0188bf98 00000001`3fcc11e1 AllocFree!foo6+0x11
00000000`0188bfa0 00000001`3fcc1201 AllocFree!foo5+0x11
00000000`0188bfa8 00000001`3fcc1221 AllocFree!foo4+0x11
00000000`0188bfb0 00000001`3fcc1241 AllocFree!foo3+0x11
00000000`0188bfb8 00000001`3fcc1261 AllocFree!foo2+0x11
00000000`0188bfc0 00000001`3fcc1281 AllocFree!foo1+0x11
00000000`0188bfc8 00000001`3fcc12e3 AllocFree!wmain+0x53
00000000`0188bfd0 00000001`3fcc156c AllocFree!__tmainCRTStartup+0x144
00000000`0188bfd8 00000000`777259ed kernel32!BaseThreadInitThunk+0xd
00000000`0188bfe0 00000000`7785c541 ntdll!RtlUserThreadStart+0x1d
00000000`0188bfe8 00000000`00000000
00000000`0188bff0 00000000`00000000
00000000`0188bff8 00000000`00000000
```

## Internal Stack Trace

Occasionally, we look at **Stack Trace Collection** (page 943) and notice **Internal Stack Trace**. This is a stack trace that is shouldn't be seen in a normal crash dump because statistically, it is rare (we planned to name this pattern **Rare Stack Trace** initially). This stack trace is also not **Special Stack Trace** (page 882) because it is not associated with the special system events or problems. It is also not a stack trace that belongs to various **Wait Chains** (page 1082) or **Spiking Threads** (page 885). This is also a real stack trace and not a reconstructed or hypothetical stack trace such as **Rough Stack Trace** (page 839) or **Past Stack Trace** (page 800). This is simply a thread stack trace that shows some internal operation, for example, where it suggests that message hooking was involved:

```
THREAD ffffffa8123702b00 Cid 11cc.0448 Teb: 000007fffffd000 Win32Thread: ffffff900c1e6ec20 WAIT:
(WrUserRequest) UserMode Non-Alertable
fffffa81230cf4e0 SynchronizationEvent
Not impersonating
DeviceMap fffff8a0058745e0
Owning Process ffffffa81237a8b30 Image: ProcessA.exe
Attached Process N/A Image: N/A
Wait Start TickCount 1258266 Ticks: 18 (0:00:00:00.280)
Context Switch Count 13752 IdealProcessor: 1 NoStackSwap LargeStack
UserTime 00:00:00.468
KernelTime 00:00:00.187
Win32 Start Address ProcessA!ThreadProc (0x000007feff17c608)
Stack Init fffff8800878c700 Current fffff8800878ba10
Base fffff8800878d000 Limit fffff88008781000 Call fffff8800878c750
Priority 12 BasePriority 8 UnusualBoost 0 ForegroundBoost 2 IoPriority 2 PagePriority 5
Child-SP RetAddr Call Site
fffff880`0878ba50 fffff800`01a6c8f2 nt!KiSwapContext+0x7a
fffff880`0878bb90 fffff800`01a7dc9f nt!KiCommitThreadWait+0x1d2
fffff880`0878bc20 fffff960`0010dbd7 nt!KeWaitForSingleObject+0x19f
fffff880`0878bcc0 fffff960`0010dc71 win32k!xxxRealSleepThread+0x257
fffff880`0878bd60 fffff960`000c4bf7 win32k!xxxSleepThread+0x59
fffff880`0878bd90 fffff960`000d07a5 win32k!xxxInterSendMsgEx+0x112a
fffff880`0878bea0 fffff960`00151bf8 win32k!xxxCallHook2+0x62d
fffff880`0878c010 fffff960`000d2454 win32k!xxxCallMouseHook+0x40
fffff880`0878c050 fffff960`0010bf23 win32k!xxxScanSysQueue+0x1828
fffff880`0878c390 fffff960`00118fae win32k!xxxRealInternalGetMessage+0x453
fffff880`0878c470 fffff800`01a76113 win32k!NtUserRealInternalGetMessage+0x7e
fffff880`0878c500 00000000`771b913a nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ fffff880`0878c570)
00000000`053ff258 000007fe`fac910f4 USER32!NtUserRealInternalGetMessage+0xa
00000000`053ff260 000007fe`fac911fa DUUser!CoreSC::xwProcessNL+0x173
00000000`053ff2d0 00000000`771b9181 DUUser!MphProcessMessage+0xbd
00000000`053ff330 00000000`774111f5 USER32!_ClientGetMessageMPH+0x3d
00000000`053ff3c0 00000000`771b908a ntdll!KiUserCallbackDispatcherContinue (TrapFrame @
00000000`053ff288)
00000000`053ff438 00000000`771b9055 USER32!NtUserPeekMessage+0xa
00000000`053ff440 000007fe`ebae03fa USER32!PeekMessageW+0x105
00000000`053ff490 000007fe`ebae4925 ProcessA+0x5a
[...]
00000000`053ff820 00000000`773ec541 kernel32!BaseThreadInitThunk+0xd
00000000`053ff850 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

We see that this thread was neither waiting for significant time nor consuming CPU. It was reported that *ProcessA.exe* was very slow responding. So perhaps this was slowly punctuated thread execution with periodic small waits. In fact, **Execution Residue** (page 371) analysis revealed Non-Coincidental Symbolic Information (page 137) of the 3rd-party **Message Hook** (page 663) and its **Module Product Process** (page 698) was identified. Its removal resolved the problem.

## Invalid Exception Information

Here we show how to recognize this pattern and get a stack trace right when a debugger is not able to locate a crash point. For example, WinDbg default analysis command is not able to locate the exception context for a crash and provides a heuristic stack trace:

```
0:000> !analyze -v

[...]

EXCEPTION_RECORD: 001150fc -- (.exr 0x1150fc)
ExceptionAddress: 7c7e2afb (kernel32!RaiseException+0x00000053)
  ExceptionCode: 0eedfade
  ExceptionFlags: 00000001
NumberParameters: 7
  Parameter[0]: 0098fa49
  Parameter[1]: 0374c200
  Parameter[2]: 00000000
  Parameter[3]: 005919b4
  Parameter[4]: 01d80010
  Parameter[5]: 00115704
  Parameter[6]: 001154a4

[...]

CONTEXT: 0012fffb4 - (.cxr 0x12fffb4)
eax=00000000 ebx=00000000 ecx=0000019c edx=00000214 esi=00000000 edi=00000000
eip=000003b0 esp=000002d8 ebp=2d59495b iopl=0 nv up ei ng nz na pe nc
cs=0032 ss=0010 ds=0002 es=0000 fs=0000 gs=0000 efl=000003e4
0032:000003b0 ??          ???

[...]

STACK_TEXT:
7c910328 ntdll!`string'+0x4
7c7db7d0 kernel32!ConsoleApp+0xe
7c7db7a4 kernel32!ConDlInitialize+0x20f
7c7db7b9 kernel32!ConDlInitialize+0x224
7c915239 ntdll!bsearch+0x42
7c91542b ntdll!RtlpLocateActivationContextSection+0x15a
7c915474 ntdll!RtlpCompareActivationContextDataTOCEEntryById+0x0
7c916104 ntdll!RtlpFindUnicodeStringInSection+0x23d
7c91534a ntdll!RtlpFindNextActivationContextSection+0x61
7c915742 ntdll!RtlFindNextActivationContextSection+0x46
7c9155ed ntdll!RtlFindActivationContextSectionString+0xde
7c915ce9 ntdll!RtlDecodeSystemPointer+0x9e7
7c915d47 ntdll!RtlDecodeSystemPointer+0xb0b
7c9158ff ntdll!RtlDecodeSystemPointer+0x45b
7c915bf8 ntdll!RtlDosApplyFileIsolationRedirection_Ustr+0x346
7c915c5d ntdll!RtlDosApplyFileIsolationRedirection_Ustr+0x3de
7c97e214 ntdll!DllExtension+0xc
00800000 ApplicationA+0x400000
```

```
7c910000 ntdll!RtlFreeHeap+0x1a4
7c914a53 ntdll!LdrLockLoaderLock+0x146
7c912d04 ntdll!LdrLockLoaderLock+0x1d2
7c912d71 ntdll!LdrUnlockLoaderLock+0x88
7c916768 ntdll!LdrGetDllHandleEx+0xc9
7c912d80 ntdll!`string'+0x84
7c91690e ntdll!LdrGetDllHandleEx+0x2f1
7c912d78 ntdll!LdrUnlockLoaderLock+0xb1
7c97ecc0 ntdll!LdrpHotpatchCount+0x8
7c9167e8 ntdll!`string'+0xc4
7c9168d6 ntdll!LdrGetDllHandleEx+0x2de
7c9166b8 ntdll!LdrGetDllHandle+0x18
7c7de534 kernel32!GetModuleHandleForUnicodeString+0x1d
7c7de544 kernel32!GetModuleHandleForUnicodeString+0xa0
7c7de64b kernel32!BasepGetModuleHandleExW+0x18e
7c7de6cb kernel32!BasepGetModuleHandleExW+0x250
79000000 mscoree!_imp__EnterCriticalSection <PERF> +0x0
7c809ad8 kernel32!_except_handler3+0x0
7c7de548 kernel32!`string'+0x28
79002280 mscoree!`string'+0x0
02080000 xpsp2res+0xc0000
7c7db6d4 kernel32!_BaseDllInitialize+0x7a
7c7db6e9 kernel32!_BaseDllInitialize+0x488
7c917ef3 ntdll!LdrpSnapThunk+0xbd
7c9048b8 ntdll!$VProc_ImageExportDirectory+0x14b8
7c9000d0 ntdll!RtlDosPathSeparatorsString <PERF> +0x0
7c905d48 ntdll!$VProc_ImageExportDirectory+0x2948
7c910228 ntdll!RtlpRunTable+0x448
7c910222 ntdll!RtlpAllocateFromHeapLookaside+0x42
7c911086 ntdll!RtlAllocateHeap+0x43d
7c903400 ntdll!$VProc_ImageExportDirectory+0x0
7c7d9036 kernel32!$VProc_ImageExportDirectory+0x6a0a
791c6f2d mscorewks!D11Main+0x117
7c917e10 ntdll!`string'+0xc
7c918047 ntdll!LdrpSnapThunk+0x317
7c7d00f0 kernel32!_imp__wcsnicmp <PERF> +0x0
7c7d903c kernel32!$VProc_ImageExportDirectory+0x6a10
7c917dba ntdll!LdrpGetProcedureAddress+0x186
7c900000 ntdll!RtlDosPathSeparatorsString <PERF> +0x0
7c917e5f ntdll!LdrpGetProcedureAddress+0x29b
7c7d262c kernel32!$VProc_ImageExportDirectory+0x0
7c7d0000 kernel32!_imp__wcsnicmp <PERF> +0x0
79513870 mscoresn!D11Main+0x119
7c913425 ntdll!RtlDecodePointer+0x0
00726574 ApplicationA+0x326574
7c917e09 ntdll!LdrpGetProcedureAddress+0xa6
7c917ec0 ntdll!LdrGetProcedureAddress+0x18
7c9101e0 ntdll!CheckHeapFillPattern+0x54
7c9101db ntdll!RtlAllocateHeap+0xeac
40ae17ea msxml16!calloc+0xa9
40ae181f msxml16!calloc+0xde
40a30000 msxml16!_imp__OpenThreadToken <PERF> +0x0
7c910323 ntdll!RtlpImageNtHeader+0x56
7c910385 ntdll!RtlImageDirectoryEntryToData+0x57
00400100 ApplicationA+0x100
7c928595 ntdll!LdrpCallTlsInitializers+0x1d
```

```

00400000 ApplicationA+0x0
7c9285c7 ntdll!LdrpCallTlsInitializers+0xd8
7c90118a ntdll!LdrpCallInitRoutine+0x14
00a23010 ApplicationA+0x623010
7c9285d0 ntdll!`string'+0x18
7c935e24 ntdll!LdrpInitializeThread+0x147
7c91b1b7 ntdll!LdrpInitializeThread+0x13b
778e159a SETUPAPI!_D11MainCRTStartup+0x0
7c91b100 ntdll!`string'+0x88
7c91b0a4 ntdll!_LdrpInitialize+0x25b
7c90de9a ntdll!NtTestAlert+0xc
7c91b030 ntdll!`string'+0xc8
7c91b02a ntdll!_LdrpInitialize+0x246
7c90d06a ntdll!NtContinue+0xc
7c90e45f ntdll!KiUserApcDispatcher+0xf
00780010 ApplicationA+0x380010
7c951e13 ntdll!DbgUiRemoteBreakin+0x0
7c97e178 ntdll!LdrpLoaderLock+0x0
00d10000 ApplicationA+0x910000
7c951e40 ntdll!DbgUiRemoteBreakin+0x2d
7c90e920 ntdll!_except_handler3+0x0
7c951e60 ntdll!`string'+0x7c

```

Compare our invalid context data with the normal one having good efl and segment register values:

```
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010206
```

We look at our stack trace after resetting the context and using **kv** command. We see that *KiUserExceptionDispatcher* has the valid exception context, but exception pointers for *UnhandledExceptionFilter* are not valid:

```

0:000> .ecxr

0:000> kv
ChildEBP RetAddr  Args to Child
001132d0 7c90df4a 7c7d9590 00000002 001132fc ntdll!KiFastSystemCallRet
001132d4 7c7d9590 00000002 001132fc 00000001 ntdll!ZwWaitForMultipleObjects+0xc
00113370 7c7da115 00000002 001134a0 00000000 kernel32!WaitForMultipleObjectsEx+0x12c
0011338c 6993763c 00000002 001134a0 00000000 kernel32!WaitForMultipleObjects+0x18
00113d20 699382b1 00115018 00000001 00198312 faultrep!StartDWException+0x5df
00114d94 7c834526 00115018 00000001 00000000 faultrep!ReportFault+0x533
00115008 0040550c 00115018 7c9032a8 001150fc kernel32!UnhandledExceptionFilter+0x55b
WARNING: Stack unwind information not available. Following frames may be wrong.
00115034 7c90327a 001150fc 0012ffb4 0011512c ApplicationA+0x550c
001150e4 7c90e48a 00000000 0011512c 001150fc ntdll!ExecuteHandler+0x24
001150e4 7c7e2afb 00000000 0011512c 001150fc ntdll!KiUserExceptionDispatcher+0xe (CONTEXT @ 0011512c)
0011544c 0057ac37 0eedfade 00000001 00000007 kernel32!RaiseException+0x53
00115470 0098fa49 0eedfade 00000001 00000007 ApplicationA+0x17ac37
[...]
0012268c 7e398816 017d0f87 000607e8 0000001a USER32!InternalCallWinProc+0x28
001226f4 7e3a8ea0 00000000 017d0f87 000607e8 USER32!UserCallWinProcCheckWow+0x150

```

```
0:000> dd 00115018 L4
00115018 001150fc 0012fffb4 0011512c 001150d0
```

So we use the valid context pointer now:

```
0:000> .cxr 0011512c
eax=001153fc ebx=0eedfade ecx=00000000 edx=001537a8 esi=001154a4 edi=00000007
eip=7c7e2afb esp=001153f8 ebp=0011544c iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200202
kernel32!RaiseException+0x53:
7c7e2afb 5e          pop     esi

0:000> kv
*** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr  Args to Child
0011544c 0057ac37 0eedfade 00000001 00000007 kernel32!RaiseException+0x53
WARNING: Stack unwind information not available. Following frames may be wrong.
00115470 0098fa49 0eedfade 00000001 00000007 ApplicationA+0x17ac37
[...]
0012268c 7e398816 017d0f87 000607e8 0000001a USER32!InternalCallWinProc+0x28
001226f4 7e3a8ea0 00000000 017d0f87 000607e8 USER32!UserCallWinProcCheckWow+0x150
00122748 7e3aacd1 00fd2ad0 0000001a 00000000 USER32!DispatchClientMessage+0xa3
00122778 7c90e473 00122788 00000030 00000030 USER32!_fnINSTRING+0x37
001227b4 7e3993e9 7e3993a8 00122840 00000000 ntdll!KiUserCallbackDispatcher+0x13
001227e0 7e3aa43b 00122840 00000000 00000000 USER32!NtUserPeekMessage+0xc
0012280c 004794d9 00122840 00000000 00000000 USER32!PeekMessageA+0xeb
001228bc 00461667 0012ff7c 00461680 001228e0 ApplicationA+0x794d9
[...]
```

## Invalid Handle

### General

Invalid handle exception (0xC0000008) is frequently seen in crash dumps. It results from an invalid handle value passed to *CloseHandle* function and other Win32 API or when a handle or a return status is checked manually for validity and the same exception is raised via *RaiseException* or internally via *RtlRaiseStatus* (**Software Exception**, page 875).

For example, critical sections are implemented using events and invalid event handle can result in this exception:

```
STACK_TEXT:
025bff00 7c94243c c0000008 7c9010ed 00231af0 ntdll!RtlRaiseStatus+0x26
025bff80 7c90104b 0015b4ac 77e76a6f 0015b4ac ntdll!RtlpWaitForCriticalSection+0x204
025bfff88 77e76a6f 0015b4ac 010d2040 00000000 ntdll!RtlEnterCriticalSection+0x46
025bffa8 77e76c0a 0015b420 025bffec 7c80b683 rpcrt4!BaseCachedThreadRoutine+0xad
025bbfb4 7c80b683 001feae8 010d2040 00000000 rpcrt4!ThreadStartRoutine+0x1a
025bffc4 00000000 77e76bf0 001feae8 00000000 kernel32!BaseThreadStart+0x37
```

By default, unless raised manually, this exception doesn't result in a default postmortem debugger launched to save a crash dump. In order to do this we need to run the application under a debugger and save a crash dump upon this exception or use exception monitoring tools that save first-chance exceptions like Debug Diagnostics, ADPlus or Exception Monitor (see **Early Crash Dump** pattern, page 312):

```
0:002> g
(7b0.d1c): Invalid handle - code c0000008 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00000000 ecx=00000000 edx=00000000 esi=7d999906 edi=00403378
eip=7d61c92d esp=0012ff68 ebp=0012ff70 iopl=0 nv up ei pl nz na po nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000202
ntdll!NtClose+0x12:
7d61c92d c20400          ret     4

0:000> g
(7b0.d1c): Invalid handle - code c0000008 (!!! second chance !!!)
eax=00000001 ebx=00000000 ecx=00000000 edx=00000000 esi=7d999906 edi=00403378
eip=7d61c92d esp=0012ff68 ebp=0012ff70 iopl=0 nv up ei pl nz na po nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000202
ntdll!NtClose+0x12:
7d61c92d c20400          ret     4
```

In order to catch it using postmortem debuggers, we can use Application Verifier and configure its basic checks to include invalid handles. Then we will have crash dumps if a postmortem debugger or WER is properly configured. The typical stack might look like below and point straight to the problem component:

```

EXCEPTION_RECORD: ffffffff -- (.exr 0xffffffffffffffff)
ExceptionAddress: 6b006369
  ExceptionCode: 80000003 (Break instruction exception)
  ExceptionFlags: 00000000
NumberParameters: 1
  Parameter[0]: 00000000

DEFAULT_BUCKET_ID: STATUS_BREAKPOINT

0:000> kL
ChildEBP RetAddr
0301ff44 0489a480 ntdll!NtClose+0x12
WARNING: Stack unwind information not available. Following frames may be wrong.
0301ff54 7d4d8e4f vfbasics+0xa480
0301ff60 04894df9 kernel32!CloseHandle+0x59
0301ff70 00401022 vfbasics+0x4df9
0301ffc0 7d4e7d2a BadHandle+0x1022
0301fff0 00000000 kernel32!BaseProcessStart+0x28

```

or it might look like this:

```

0:000> kL
Child-SP      RetAddr          Call Site
00000000`0012ed58 00000000`01f9395a ntdll!DbgBreakPoint
00000000`0012ed60 00000000`023e29a7 vrfcore!VerifierStopMessageEx+0x846
00000000`0012f090 00000000`023d9384 vfbasics+0x129a7
00000000`0012f0f0 00000000`77f251ec vfbasics+0x9384
00000000`0012f180 00000000`77ee5f36 ntdll!RtlpCallVectoredHandlers+0x26f
00000000`0012f210 00000000`77ee6812 ntdll!RtlDispatchException+0x46
00000000`0012f8c0 00000000`77ef325a ntdll!RtlRaiseException+0xae
00000000`0012fe00 00000000`77d6e314 ntdll!KiRaiseUserExceptionDispatcher+0x3a
00000000`0012fed0 00000001`40001028 kernel32!CloseHandle+0x5f
00000000`0012ff00 00000001`40001294 BadHandle+0x1028
00000000`0012ff30 00000000`77d5964c BadHandle+0x1294
00000000`0012ff80 00000000`00000000 kernel32!BaseProcessStart+0x29

```

*vf~~basics~~* and *vrfcore* modules are Application Verifier DLLs that possibly translate an invalid handle exception to a breakpoint exception and, therefore, trigger the launch of a postmortem debugger from an unhandled exception filter. Application Verifier version (x64 or x86) must match the application platform (64-bit or 32-bit)<sup>120</sup>.

If invalid handle exception is raised manually we get the status code and possibly problem component immediately from **!analyze** command:

```

FAULTING_IP:
kernel32!RaiseException+53
7d4e2366 5e          pop     esi

```

---

<sup>120</sup> Memory Dump Analysis Anthology, Volume 2, page 413

```

EXCEPTION_RECORD: ffffffff -- (.exr 0xffffffffffff)
ExceptionAddress: 7d4e2366 (kernel32!RaiseException+0x00000053)
  ExceptionCode: c0000008 (Invalid handle)
  ExceptionFlags: 00000000
NumberParameters: 0
Thread tried to close a handle that was invalid or illegal to close

DEFAULT_BUCKET_ID: STATUS_INVALID_HANDLE

PROCESS_NAME: BadHandle.exe

ERROR_CODE: (NTSTATUS) 0xc0000008 - An invalid HANDLE was specified.

STACK_TEXT:
0012ff64 00401043 c0000008 00000000 00000000 kernel32!RaiseException+0x53
WARNING: Stack unwind information not available. Following frames may be wrong.
0012ffc0 7d4e7d2a 00000000 00000000 7efde000 BadHandle+0x1043
0012fff0 00000000 004012f9 00000000 00000000 kernel32!BaseProcessStart+0x28

FAULTING_THREAD: 00000b64

PRIMARY_PROBLEM_CLASS: STATUS_INVALID_HANDLE

BUGCHECK_STR: APPLICATION_FAULT_STATUS_INVALID_HANDLE

```

Because we have WinDbg warning about stack unwind we can double check the disassembly of *RaiseException* return address:

```

0:000> ub 00401043
BadHandle+0x1029:
00401029 push    offset BadHandle+0x212c (0040212c)
0040102e push    0
00401030 call    esi
00401032 push    0
00401034 push    0
00401036 push    0
00401038 push    0C0000008h
0040103d call    dword ptr [BadHandle+0x2004 (00402004)]

0:000> dps 00402004 11
00402004 7d4e2318 kernel32!RaiseException

```

In such cases, the real problem could be memory corruption overwriting stored valid handle values.

## Comments

We can also check if the handle value is valid or not:

```
STACK_TEXT:  
0fbef7f8 7581c455 0000aa4 0000aa4 0fbef818 ntdll!ZwClose+0x12  
0fbef808 76051438 0000aa4 000000d 00000000 KERNELBASE!CloseHandle+0x2d  
0fbef818 5fa2632a 0000aa4 5fa24b9c 00000000 kernel32!CloseHandleImplementation+0x3f  
[...]  
  
0:017> !handle 0000aa4  
Handle 0000aa4  
Type  
  
0:017> !handle 0000aa0  
Handle 0000aa0  
Type Thread  
  
0:017> !handle 0000aa8  
Handle 0000aa8  
Type Event
```

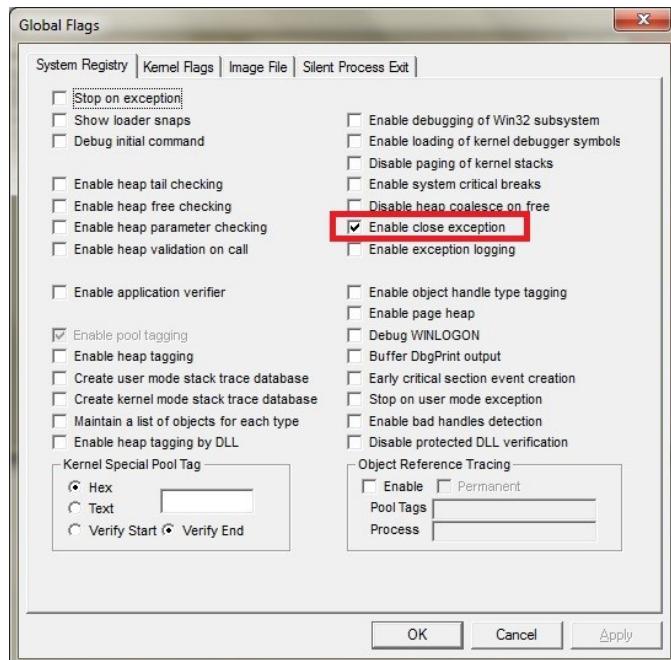
Enabling application verifier in *gflags.exe* to catch handle close stack traces may also reveal who closed the handle originally before we got the second invalid handle close. Use **!htrace** command there.

## Managed Space

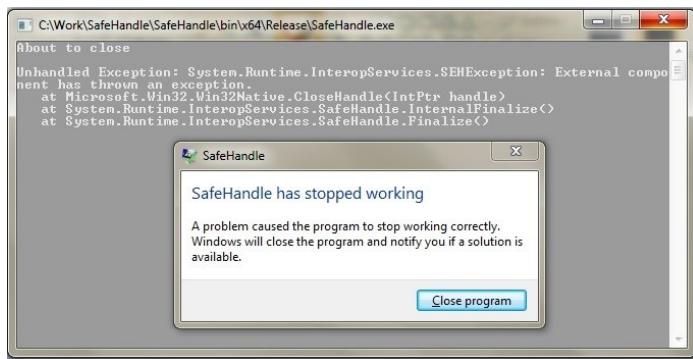
We recently encountered **Invalid Handle** pattern in the context of .NET program execution. We decided to model it and wrote a small C# program:

```
namespace SafeHandle
{
    class Program
    {
        static void Main(string[] args)
        {
            SafeFileHandle hFile =
                new SafeFileHandle(new IntPtr(0xDEAD), true);
            Console.WriteLine("About to close");
            Console.ReadKey();
        }
    }
}
```

Of course, when we execute it nothing happens. Invalid handles are ignored by default. However, to change the behavior we enabled "*Enable close exception*" in *gflags.exe*:



Moreover, if we run it we get this **Managed Stack Trace** (page 624):



We could have detected invalid handle if we enabled *Application Verifier*, but then we would not have **Managed Code Exception** (page 617).

So we load a crash dump (saved because we enabled LocalDumps<sup>121</sup>) and load SOS extension:

```
0:002> lmv m clr
start end module name
000007fe`ed880000 000007fe`ee1eb000 clr (pdb symbols)
Loaded symbol image file: clr.dll
Image path: C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clr.dll
[...]

0:002> .load C:\Windows\Microsoft.NET\Framework64\v4.0.30319\sos

0:002> !pe
Exception object: 0000000002ab5fe8
Exception type: System.Runtime.InteropServices.SEHException
Message: External component has thrown an exception.
InnerException:
StackTrace (generated):
SP IP Function
000000001B40EDD0 0000000000000000 mscorelib_ni!Microsoft.Win32.Win32Native.CloseHandle(IntPtr)+0x1
000000001B40F2F0 0000000000000000
mscorelib_ni!System.Runtime.InteropServices.SafeHandle.InternalFinalize()+0x1
000000001B40F2F0 000007FEEC62F7A6 mscorelib_ni!System.Runtime.InteropServices.SafeHandle.Finalize()+0x26

StackTraceString:
HRESULT: 80004005
```

Our unmanaged **CLR Thread** (page 124) **Exception Stack Trace** (page 363) is quite simple:

---

<sup>121</sup> <http://msdn.microsoft.com/en-us/library/bb787181.aspx>

```

0:002> k
Child-SP RetAddr Call Site
00000000`1b40d6e8 000007fe`fd651430 ntdll!NtWaitForMultipleObjects+0xa
00000000`1b40d6f0 00000000`77621723 KERNELBASE!WaitForMultipleObjectsEx+0xe8
00000000`1b40d7f0 00000000`7769b5e5 kernel32!WaitForMultipleObjectsExImplementation+0xb3
00000000`1b40d880 00000000`7769b767 kernel32!WerpReportFaultInternal+0x215
00000000`1b40d920 00000000`7769b7bf kernel32!WerpReportFault+0x77
00000000`1b40d950 00000000`7769b9dc kernel32!BaseReportFault+0x1f
00000000`1b40d980 00000000`778b3398 kernel32!UnhandledExceptionFilter+0x1fc
00000000`1b40da60 00000000`778385c8 ntdll! ?? ::FNODOBFM::`string'+0x2365
00000000`1b40da90 00000000`77849d2d ntdll!_C_specific_handler+0x8c
00000000`1b40db00 00000000`778391cf ntdll!RtlpExecuteHandlerForException+0xd
00000000`1b40db30 00000000`778397c8 ntdll!RtlDispatchException+0x45a
00000000`1b40e210 00000000`778712c7 ntdll!RtlRaiseException+0x22f
00000000`1b40ebc0 000007fe`fd651873 ntdll!KiRaiseUserExceptionDispatcher+0x3a
00000000`1b40ec90 00000000`77621991 KERNELBASE!CloseHandle+0x13
00000000`1b40ecc0 000007fe`ec720418 kernel32!CloseHandleImplementation+0x3d
00000000`1b40ed00 000007fe`ed8e9e03 mscorelib_ni+0x580418
00000000`1b40eea0 000007fe`ed8e9e7e clr!CallDescrWorkerInternal+0x83
00000000`1b40eee0 000007fe`ed8ec860 clr!CallDescrWorkerWithHandler+0x4a
00000000`1b40ef20 000007fe`ed8f1a1d clr!DispatchCallSimple+0x85
00000000`1b40efb0 000007fe`ed8f19ac clr!SafeHandle::RunReleaseMethod+0x69
00000000`1b40f050 000007fe`ed8f180a clr!SafeHandle::Release+0x122
00000000`1b40f120 000007fe`eda4863e clr!SafeHandle::Dispose+0x36
00000000`1b40f190 000007fe`ec62f7a6 clr!SafeHandle::Finalize+0xa2
00000000`1b40f2f0 000007fe`ed8e9d56 mscorelib_ni+0x48f7a6
00000000`1b40f330 000007fe`eda83c4e clr!FastCallFinalizeWorker+0x6
00000000`1b40f360 000007fe`eda83bc3 clr!MethodDesc::RequiresFullSlotNumber+0x72
00000000`1b40f3a0 000007fe`eda83b0f clr!MethodTable::CallFinalizer+0xa3
00000000`1b40f3e0 000007fe`ed9fee46 clr!SVR::CallFinalizer+0x5f
00000000`1b40f420 000007fe`ed9aac5b clr!SVR::CallFinalizer+0x102
00000000`1b40f4e0 000007fe`ed8f458c clr!WKS::GCHeap::IsPromoted+0xee
00000000`1b40f520 000007fe`ed8f451a clr!Frame::Pop+0x50
00000000`1b40f560 000007fe`ed8f4491 clr!COMCustomAttribute::PopSecurityContextFrame+0x192
00000000`1b40f660 000007fe`ed9d1bfe clr!COMCustomAttribute::PopSecurityContextFrame+0xbd
00000000`1b40f6f0 000007fe`ed9d1e59 clr!ManagedThreadBase_NoADTransition+0x3f
00000000`1b40f750 000007fe`ed9533de clr!WKS::GCHeap::FinalizerThreadStart+0x193
00000000`1b40f790 00000000`776159ed clr!Thread::intermediateThreadProc+0x7d
00000000`1b40f850 00000000`7784c541 kernel32!BaseThreadInitThunk+0xd
00000000`1b40f880 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

```

We see that exception processing happened during object finalization. We can infer the value of the handle (maybe **Small Value**, page 873) via disassembly if this is possible:

```

0:002> kn
# Child-SP RetAddr Call Site
00 00000000`1b40d6e8 000007fe`fd651430 ntdll!NtWaitForMultipleObjects+0xa
01 00000000`1b40d6f0 00000000`77621723 KERNELBASE!WaitForMultipleObjectsEx+0xe8
02 00000000`1b40d7f0 00000000`7769b5e5 kernel32!WaitForMultipleObjectsExImplementation+0xb3
03 00000000`1b40d880 00000000`7769b767 kernel32!WerFaultInternal+0x215
04 00000000`1b40d920 00000000`7769b7bf kernel32!WerFault+0x77
05 00000000`1b40d950 00000000`7769b9dc kernel32!BaseWerFault+0x1f
06 00000000`1b40d980 00000000`778b3398 kernel32!UnhandledExceptionFilter+0xfc
07 00000000`1b40da60 00000000`778385c8 ntdll! ?? ::FNODOBFM::`string'+0x2365
08 00000000`1b40da90 00000000`77849d2d ntdll!_C_specific_handler+0x8c
09 00000000`1b40db00 00000000`778391cf ntdll!RtlpExecuteHandlerForException+0xd
0a 00000000`1b40db30 00000000`778397c8 ntdll!RtlDispatchException+0x45a
0b 00000000`1b40e210 00000000`778712c7 ntdll!RtlRaiseException+0x22f
0c 00000000`1b40ebc0 000007fe`fd651873 ntdll!KiRaiseUserExceptionDispatcher+0x3a
0d 00000000`1b40ec90 00000000`77621991 KERNELBASE!CloseHandle+0x13
0e 00000000`1b40ecc0 000007fe`ec720418 kernel32!CloseHandleImplementation+0x3d
0f 00000000`1b40edd0 000007fe`ed8e9e03 mscorelib_ni+0x580418
10 00000000`1b40eea0 000007fe`ed8e9e7e clr!CallDescrWorkerInternal+0x83
11 00000000`1b40eee0 000007fe`ed8ec860 clr!CallDescrWorkerWithHandler+0x4a
12 00000000`1b40ef20 000007fe`ed8f1a1d clr!DispatchCallSimple+0x85
13 00000000`1b40efb0 000007fe`ed8f19ac clr!SafeHandle::RunReleaseMethod+0x69
14 00000000`1b40f050 000007fe`ed8f180a clr!SafeHandle::Release+0x122
15 00000000`1b40f120 000007fe`eda4863e clr!SafeHandle::Dispose+0x36
16 00000000`1b40f190 000007fe`ec62f7a6 clr!SafeHandle::Finalize+0xa2
17 00000000`1b40f2f0 000007fe`ed8e9d56 mscorelib_ni+0x48f7a6
18 00000000`1b40f330 000007fe`eda83c4e clr!FastCallFinalizeWorker+0x6
19 00000000`1b40f360 000007fe`eda83bc3 clr!MethodDesc::RequiresFullSlotNumber+0x72
1a 00000000`1b40f3a0 000007fe`eda83b0f clr!MethodTable::CallFinalizer+0xa3
1b 00000000`1b40f3e0 000007fe`ed9fee46 clr!SVR::CallFinalizer+0x5f
1c 00000000`1b40f420 000007fe`ed9aac5b clr!SVR::CallFinalizer+0x102
1d 00000000`1b40f4e0 000007fe`ed8f458c clr!WKS::GCHeap::IsPromoted+0xee
1e 00000000`1b40f520 000007fe`ed8f451a clr!Frame::Pop+0x50
1f 00000000`1b40f560 000007fe`ed8f4491 clr!COMCustomAttribute::PopSecurityContextFrame+0x192
20 00000000`1b40f660 000007fe`ed9d1bfe clr!COMCustomAttribute::PopSecurityContextFrame+0xbd
21 00000000`1b40f6f0 000007fe`ed9d1e59 clr!ManagedThreadBase_NoADTransition+0x3f
22 00000000`1b40f750 000007fe`ed9533de clr!WKS::GCHeap::FinalizerThreadStart+0x193
23 00000000`1b40f790 00000000`776159ed clr!Thread::intermediateThreadProc+0x7d
24 00000000`1b40f850 00000000`7784c541 kernel32!BaseThreadInitThunk+0xd
25 00000000`1b40f880 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

```

```

0:002> .frame /c d
0d 00000000`1b40ec90 00000000`77621991 KERNELBASE!CloseHandle+0x13
rax=00000000c0000001 rbx=000000000000dead rcx=00000000009a0000
rdx=0000000000000001 rsi=000000001b40efd0 rdi=000000001b40eff8
rip=000007fefdf651873 rsp=000000001b40ec90 rbp=000000001b40edf0
r8=000000001b40ce08 r9=000000001b40cf70 r10=0000000000000000
r11=000000000000246 r12=0000000000000001 r13=0000000040000000
r14=000000001b40ef40 r15=0000000000000000
iopl=0 nv up ei pl zr na po nc
cs=0033 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000246
KERNELBASE!CloseHandle+0x13:
000007fe`fd651873 85c0 test eax,eax

```

```
0:002> ub 00000000`77621991
kernel32!CloseHandleImplementation+0x1e:
00000000`7762196e 83f9f4      cmp ecx,0FFFFFFF4h
00000000`77621971 0f83952e0100 jae kernel32!TlsGetValue+0x3ef0 (00000000`7763480c)
00000000`77621977 488bc3      mov rax,rbx
00000000`7762197a 2503000010 and eax,10000003h
00000000`7762197f 4883f803    cmp rax,3
00000000`77621983 0f847f8dff fe je kernel32!CloseHandleImplementation+0x56 (00000000`7760a708)
00000000`77621989 488bcb      mov rcx,rbx
00000000`7762198c e81f000000  call kernel32!CloseHandle (00000000`776219b0)
```

Here we also check the value from the managed stack trace or **Execution Residue** (page 369):

```
0:002> !CLRStack -a
OS Thread Id: 0x1390 (2)
Child SP IP Call Site
000000001b40edf8 000000007787186a [InlinedCallFrame: 000000001b40edf8]
Microsoft.Win32.Win32Native.CloseHandle(IntPtr)
000000001b40edf8 000007feec720418 [InlinedCallFrame: 000000001b40edf8]
Microsoft.Win32.Win32Native.CloseHandle(IntPtr)
000000001b40eddf0 000007feec720418 DomainNeutralILStubClass.IL_STUB_PInvoke(IntPtr)
PARAMETERS:
<no data>

000000001b40eff8 000007feed8e9e03 [GCFrame: 000000001b40eff8]
000000001b40f148 000007feed8e9e03 [GCFrame: 000000001b40f148]
000000001b40f1f8 000007feed8e9e03 [HelperMethodFrame_1OBJ: 000000001b40f1f8]
System.Runtime.InteropServices.SafeHandle.InternalFinalize()
000000001b40f2f0 000007feec62f7a6 System.Runtime.InteropServices.SafeHandle.Finalize()
PARAMETERS:
this (0x000000001b40f330) = 0x0000000002ab2d78

000000001b40f6a8 000007feed8e9d56 [DebuggerU2MCatchHandlerFrame: 000000001b40f6a8]

0:002> !dso
OS Thread Id: 0x1390 (2)
RSP/REG Object Name
000000001B40EEA0 000000002ab2d78 Microsoft.Win32.SafeHandles.SafeFileHandle
000000001B40EF00 000000002ab2d78 Microsoft.Win32.SafeHandles.SafeFileHandle
000000001B40F038 000000002ab2d78 Microsoft.Win32.SafeHandles.SafeFileHandle
000000001B40F050 000000002ab2d78 Microsoft.Win32.SafeHandles.SafeFileHandle
000000001B40F090 000000002ab2d78 Microsoft.Win32.SafeHandles.SafeFileHandle
000000001B40F120 000000002ab2d78 Microsoft.Win32.SafeHandles.SafeFileHandle
000000001B40F190 000000002ab2d78 Microsoft.Win32.SafeHandles.SafeFileHandle
000000001B40F1B8 000000002ab2d78 Microsoft.Win32.SafeHandles.SafeFileHandle
000000001B40F240 000000002ab2d78 Microsoft.Win32.SafeHandles.SafeFileHandle
000000001B40F2F8 000000002ab2d78 Microsoft.Win32.SafeHandles.SafeFileHandle
000000001B40F330 000000002ab2d78 Microsoft.Win32.SafeHandles.SafeFileHandle
000000001B40F360 000000002ab5e10 System.Threading.Thread
000000001B40F390 000000002ab2d78 Microsoft.Win32.SafeHandles.SafeFileHandle
000000001B40F3E0 000000002ab2d78 Microsoft.Win32.SafeHandles.SafeFileHandle
000000001B40F3F0 000000002ab2d78 Microsoft.Win32.SafeHandles.SafeFileHandle
000000001B40F430 000000002ab58a8 Microsoft.Win32.SafeHandles.SafeViewOfFileHandle
000000001B40F4E0 000000002ab2d78 Microsoft.Win32.SafeHandles.SafeFileHandle
```

```
0:002> !do 0000000002ab2d78
Name: Microsoft.Win32.SafeHandles.SafeFileHandle
MethodTable: 000007feec88a260
EEClass: 000007feec34d340
Size: 32(0x20) bytes
File: C:\windows\Microsoft.Net\assembly\GAC_64\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorlib.dll
Fields:
MT Field Offset Type VT Attr Value Name
000007feec88a338 400060d 8 System.IntPtr 1 instance dead handle
000007feec8892b8 400060e 10 System.Int32 1 instance 3 _state
000007feec887de0 400060f 14 System.Boolean 1 instance 1 _ownsHandle
000007feec887de0 4000610 15 System.Boolean 1 instance 1 _fullyInitialized
```

Please note that we do not have global application flags:

```
0:002> !gflag
Current NtGlobalFlag contents: 0x00000000
```

Here is the exception stack trace from a different crash dump when we enable *Application Verifier*:

```
0:002> !gflag
Current NtGlobalFlag contents: 0x02000100
vrf - Enable application verifier
hpa - Place heap allocations at ends of pages

0:002> k
Child-SP RetAddr Call Site
0000000`24bac4a8 0000000`77cd3072 ntdll!NtWaitForSingleObject+0xa
0000000`24bac4b0 0000000`77cd32b5 ntdll!RtlReportExceptionEx+0x1d2
0000000`24bac5a0 000007fe`fa2c26fb ntdll!RtlReportException+0xb5
0000000`24bac620 0000000`77c2a5db verifier!AVrfpVectoredExceptionHandler+0x26b
0000000`24bac6b0 0000000`77c28e62 ntdll!RtlpCallVectoredHandlers+0xa8
0000000`24bac720 0000000`77c61248 ntdll!RtlDispatchException+0x22
0000000`24bace00 000007fe`fa2bae03 ntdll!KiUserExceptionDispatch+0x2e
0000000`24bad500 000007fe`fa2c268a verifier!VerifierStopMessageEx+0x6fb
0000000`24bad850 0000000`77c2a5db verifier!AVrfpVectoredExceptionHandler+0x1fa
0000000`24bad8e0 0000000`77c28e62 ntdll!RtlpCallVectoredHandlers+0xa8
0000000`24bad950 0000000`77c297c8 ntdll!RtlDispatchException+0x22
0000000`24bae030 0000000`77c612c7 ntdll!RtlRaiseException+0x22f
0000000`24bae9e0 000007fe`fa2d2386 ntdll!KiRaiseUserExceptionDispatcher+0x3a
0000000`24baeb0 000007fe`fdbd1873 verifier!AVrfpNtClose+0xbe
0000000`24baeae0 000007fe`fa2d4031 KERNELBASE!CloseHandle+0x13
0000000`24baeb10 000007fe`fa2d40cb verifier!AVrfpCloseHandleCommon+0x95
0000000`24baeb40 0000000`77a11991 verifier!AVrfpKernelbaseCloseHandle+0x23
0000000`24baeb80 000007fe`fa2d4031 kernel32!CloseHandleImplementation+0x3d
0000000`24baec90 000007fe`fa2d409c verifier!AVrfpCloseHandleCommon+0x95
*** WARNING: Unable to verify checksum for mscorelib.ni.dll
0000000`24baecc0 000007fe`e6a40418 verifier!AVrfpKernel32CloseHandle+0x2c
0000000`24baed0 000007fe`ec0e9e03 mscorelib_ni+0x580418
0000000`24baedd0 000007fe`ec0e9e7e clr!CallDescrWorkerInternal+0x83
0000000`24baee10 000007fe`ec0ec860 clr!CallDescrWorkerWithHandler+0x4a
0000000`24baee50 000007fe`ec0f1a1d clr!DispatchCallSimple+0x85
0000000`24baeee0 000007fe`ec0f19ac clr!SafeHandle::RunReleaseMethod+0x69
0000000`24baef80 000007fe`ec0f180a clr!SafeHandle::Release+0x122
```

```

00000000`24baef050 000007fe`ec24863e clr!SafeHandle::Dispose+0x36
00000000`24baef0c0 000007fe`e694f7a6 clr!SafeHandle::Finalize+0xa2
00000000`24baef220 000007fe`ec0e9d56 mscorelib_ni+0x48f7a6
00000000`24baef260 000007fe`ec283c4e clr!FastCallFinalizeWorker+0x6
00000000`24baef290 000007fe`ec283bc3 clr!MethodDesc::RequiresFullSlotNumber+0x72
00000000`24baef2d0 000007fe`ec283b0f clr!MethodTable::CallFinalizer+0xa3
00000000`24baef310 000007fe`ec1fee46 clr!SVR::CallFinalizer+0x5f
00000000`24baef350 000007fe`ec1aac5b clr!SVR::CallFinalizer+0x102
00000000`24baef410 000007fe`ec0f458c clr!WKS::GCHeap::IsPromoted+0xee
00000000`24baef450 000007fe`ec0f451a clr!Frame::Pop+0x50
00000000`24baef490 000007fe`ec0f4491 clr!COMCustomAttribute::PopSecurityContextFrame+0x192
00000000`24baef590 000007fe`ec1d1bfe clr!COMCustomAttribute::PopSecurityContextFrame+0xbd
00000000`24baef620 000007fe`ec1d1e59 clr!ManagedThreadBase_NoADTransition+0x3f
00000000`24baef680 000007fe`ec1533de clr!WKS::GCHeap::FinalizerThreadStart+0x193
00000000`24baef6c0 000007fe`fa2d4b87 clr!Thread::intermediateThreadProc+0x7d
00000000`24baef780 00000000`77a059ed verifier!AVrfpStandardThreadFunction+0x2b
00000000`24baef7c0 00000000`77c3c541 kernel32!BaseThreadInitThunk+0xd
00000000`24baef7f0 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

```

```

0:002> !pe
There is no current managed exception on this thread

```

```

0:002> !CLRStack
OS Thread Id: 0x51e4 (2)
Child SP IP Call Site
0000000024baed28 0000000077c612fa [InlinedCallFrame: 0000000024baed28]
Microsoft.Win32.Win32Native.CloseHandle(IntPtr)
0000000024baed28 000007fee6a40418 [InlinedCallFrame: 0000000024baed28]
Microsoft.Win32.Win32Native.CloseHandle(IntPtr)
0000000024baed00 000007fee6a40418 DomainNeutralILStubClass.IL_STUB_PInvoke(IntPtr)
0000000024baef28 000007feec0e9e03 [GCFrame: 0000000024baef28]
0000000024baf078 000007feec0e9e03 [GCFrame: 0000000024baf078]
0000000024baf128 000007feec0e9e03 [HelperMethodFrame_1OBJ: 0000000024baf128]
System.Runtime.InteropServices.SafeHandle.InternalFinalize()
0000000024baf220 000007fee694f7a6 System.Runtime.InteropServices.SafeHandle.Finalize()
0000000024baf5d8 000007feec0e9d56 [DebuggerU2MCatchHandlerFrame: 0000000024baf5d8]

```

```

0:002> !dso
OS Thread Id: 0x51e4 (2)
RSP/REG Object Name
0000000024BAE000 00000000c282d78 Microsoft.Win32.SafeHandles.SafeFileHandle
0000000024BAE000 00000000c282d78 Microsoft.Win32.SafeHandles.SafeFileHandle
0000000024BAE068 00000000c282d78 Microsoft.Win32.SafeHandles.SafeFileHandle
0000000024BAE080 00000000c282d78 Microsoft.Win32.SafeHandles.SafeFileHandle
0000000024BAEFC0 00000000c282d78 Microsoft.Win32.SafeHandles.SafeFileHandle
0000000024BAF050 00000000c282d78 Microsoft.Win32.SafeHandles.SafeFileHandle
0000000024BAF0C0 00000000c282d78 Microsoft.Win32.SafeHandles.SafeFileHandle
0000000024BAF0E8 00000000c282d78 Microsoft.Win32.SafeHandles.SafeFileHandle
0000000024BAF170 00000000c282d78 Microsoft.Win32.SafeHandles.SafeFileHandle
0000000024BAF228 00000000c282d78 Microsoft.Win32.SafeHandles.SafeFileHandle
0000000024BAF260 00000000c282d78 Microsoft.Win32.SafeHandles.SafeFileHandle
0000000024BAF290 00000000c285e10 System.Threading.Thread
0000000024BAF2C0 00000000c282d78 Microsoft.Win32.SafeHandles.SafeFileHandle
0000000024BAF310 00000000c282d78 Microsoft.Win32.SafeHandles.SafeFileHandle
0000000024BAF320 00000000c282d78 Microsoft.Win32.SafeHandles.SafeFileHandle

```

```

0000000024BAF360 00000000c2858a8 Microsoft.Win32.SafeHandles.SafeViewOfFileHandle
0000000024BAF410 00000000c282d78 Microsoft.Win32.SafeHandles.SafeFileHandle

0:002> !CLRStack -a
OS Thread Id: 0x51e4 (2)
Child SP          IP Call Site
0000000024baed28 0000000077c612fa [InlinedCallFrame: 0000000024baed28]
Microsoft.Win32.Win32Native.CloseHandle(IntPtr)
0000000024baed28 000007fee6a40418 [InlinedCallFrame: 0000000024baed28]
Microsoft.Win32.Win32Native.CloseHandle(IntPtr)
0000000024baed00 000007fee6a40418 DomainNeutralILStubClass.IL_STUB_PInvoke(IntPtr)
PARAMETERS:
<no data>

0000000024baef28 000007feec0e9e03 [GCFrame: 0000000024baef28]
0000000024baf078 000007feec0e9e03 [GCFrame: 0000000024baf078]
0000000024baf128 000007feec0e9e03 [HelperMethodFrame_1OBJ: 0000000024baf128]
System.Runtime.InteropServices.SafeHandle.InternalFinalize()
0000000024baf220 000007fee694f7a6 System.Runtime.InteropServices.SafeHandle.Finalize()
PARAMETERS:
this (0x0000000024baf260) = 0x00000000c282d78

0000000024baf5d8 000007feec0e9d56 [DebuggerU2MCatchHandlerFrame: 0000000024baf5d8]

0:002> !do 0x0000000000c282d78
Name: Microsoft.Win32.SafeHandles.SafeFileHandle
MethodTable: 000007fee6baa260
EEClass: 000007fee666d340
Size: 32(0x20) bytes
File: C:\Windows\Microsoft.Net\assembly\GAC_64\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorlib.dll
Fields:
MT      Field  Offset Type       VT Attr   Value Name
000007fee6baa338 400060d 8     System.IntPtr 1 instance dead handle
000007fee6ba92b8 400060e 10    System.Int32 1 instance 3 _state
000007fee6ba7de0 400060f 14    System.Boolean 1 instance 1 _ownsHandle
000007fee6ba7de0 4000610 15    System.Boolean 1 instance 1 _fullyInitialized

```

## Comments

Enabling application verifier in *gflags.exe* to catch handle close stack traces may also reveal who closed the handle originally before we got the second invalid handle close. Use **!htrace** command there.

## Invalid Parameter

### Process Heap

This is a general pattern of passing unexpected values to functions. Here we look at invalid heap block parameter pattern specialization. It is different from [Dynamic Memory Corruption](#) (page 307) or [Double Free](#) (page 267) pattern because no corruption happens in heap structures before detection and the parameter value has never been correct before its use. For example, we have this stack trace:

```
0:003> kL 100
ChildEBP RetAddr
01b2e6f0 77f27d0c ntdll!ZwWaitForSingleObject+0x15
01b2e774 77f27e3a ntdll!RtlReportExceptionEx+0x14b
01b2e7cc 77f4dc2e ntdll!RtlReportException+0x86
01b2e7e0 77f4dcab ntdll!RtlpTerminateFailureFilter+0x14
01b2e7ec 77ef05c4 ntdll!RtlReportCriticalFailure+0x67
01b2e800 77ef0469 ntdll!_EH4_CallFilterFunc+0x12
01b2e828 77ed8799 ntdll!_except_handler4+0x8e
01b2e84c 77ed876b ntdll!ExecuteHandler2+0x26
01b2e8fc 77e9010f ntdll!ExecuteHandler+0x24
01b2e8fc 77f4dc9b ntdll!KiUserExceptionDispatcher+0xf
01b2ecc4 77f4eba1 ntdll!RtlReportCriticalFailure+0x57
01b2ecd4 77f4ec81 ntdll!RtlpReportHeapFailure+0x21
01b2ed08 77efdd0 ntDLL!RtlpLogHeapFailure+0xa1
01b2ed38 76bc14d1 ntdll!RtlFreeHeap+0x64
01b2ed4c 75694c39 kernel32!HeapFree+0x14
01b2ed98 726f167d msVCR80!free+0xcd
01b2eda4 7270613d DLLA!FreeData+0xd
[...]
01b2fe38 77eb9d42 kernel32!BaseThreadInitThunk+0xe
01b2fe78 77eb9d15 ntdll!__RtlUserThreadStart+0x70
01b2fe90 00000000 ntdll!_RtlUserThreadStart+0x1b
```

We see that the failure was detected and logged immediately without any [Instrumentation Information](#) (page 516):

```
0:003> !gflag
Current NtGlobalFlag contents: 0x00000000
```

If we enable full page heap we get this default analysis output and the following stack trace:

```
0:003> !gflag
Current NtGlobalFlag contents: 0x02000000
hpa - Place heap allocations at ends of pages
```

```
0:003> !analyze -v
```

[...]

APPLICATION\_VERIFIER\_HEAPS\_CORRUPTED\_HEAP\_BLOCK\_EXCEPTION\_RAISED\_FOR\_PROBING (c)  
Exception raised while verifying the heap block.

This situation happens if we really cannot determine any particular type of corruption for the block. For instance you will get this if during a heap free operation you pass an address that points to a non-accessible memory area.

This can also happen for double free situations if we do not find the block among full page heap blocks and we probe it as a light page heap block.

Arguments:

Arg1: 05eb1000, Heap handle used in the call.

**Arg2: 00720071, Heap block involved in the operation.**

Arg3: 00000000, Size of the heap block.

Arg4: c0000005, Reserved.

[...]

```
0:003> kL 100
ChildEBP RetAddr
0818dca4 75fa0962 ntdll!ZwWaitForMultipleObjects+0x15
0818dd40 76bc162d KERNELBASE!WaitForMultipleObjectsEx+0x100
0818dd88 76bc1921 kernel32!WaitForMultipleObjectsExImplementation+0xe0
0818dda4 76be9b0d kernel32!WaitForMultipleObjects+0x18
0818de10 76be9baa kernel32!WerpReportFaultInternal+0x186
0818de24 76be98d8 kernel32!WerpReportFault+0x70
0818de34 76be9855 kernel32!BasepReportFault+0x20
0818dec0 77ef06e7 kernel32!UnhandledExceptionFilter+0x1af
0818dec8 77ef05c4 ntdll!_RtlUserThreadStart+0x62
0818dedc 77ef0469 ntdll!_EH4_CallFilterFunc+0x12
0818df04 77ed8799 ntdll!_except_handler4+0x8e
0818df28 77ed876b ntdll!ExecuteHandler2+0x26
0818dfd8 77e9010f ntdll!ExecuteHandler+0x24
0818fdf8 71a6ba58 ntdll!KiUserExceptionDispatcher+0xf
0818e344 71a69ee0 verifier!VerifierStopMessage+0x1f8
0818e3a8 71a66f11 verifier!AVrfpDphReportCorruptedBlock+0x2b0
0818e3bc 71a819ec verifier!AVrfpDphFindBusyMemoryNoCheck+0x141
0818e3d0 71a8174e verifier!_EH4_CallFilterFunc+0x12
0818e3f8 77ed8799 verifier!_except_handler4+0x8e
0818e41c 77ed876b ntdll!ExecuteHandler2+0x26
0818e4cc 77e9010f ntdll!ExecuteHandler+0x24
0818e4cc 71a66e88 ntdll!KiUserExceptionDispatcher+0xf
0818e868 71a66f95 verifier!AVrfpDphFindBusyMemoryNoCheck+0xb8
0818e88c 71a67240 verifier!AVrfpDphFindBusyMemory+0x15
0818e8a8 71a69080 verifier!AVrfpDphFindBusyMemoryAndRemoveFromBusyList+0x20
0818e8c4 77f50aac verifier!AVrfDebugPageHeapFree+0x90
0818e90c 77f0a8ff ntdll!RtlDebugFreeHeap+0x2f
0818ea00 77eb2a32 ntdll!RtlpFreeHeap+0x5d
0818ea20 76bc14d1 ntdll!RtlFreeHeap+0x142
0818ea34 75694c39 kernel32!HeapFree+0x14
0818ea80 726f167d msrvcr80!free+0xcd
0818ea8c 7270613d DllA!FreeData+0xd
[...]
0818fb20 77eb9d42 kernel32!BaseThreadInitThunk+0xe
```

```
0818fb60 77eb9d15 ntdll!__RtlUserThreadStart+0x70
0818fb78 00000000 ntdll!__RtlUserThreadStart+0x1b
```

In both examples above we see that 00720071 value was passed to *free* function (we also verify from the code using **ub** command that there was no parameter optimization, **Optimized Code** pattern, page 767):

```
0:003> kv
ChildEBP RetAddr Args to Child
[...]
01b2ed98 726f167d 00720071 01b2edb0 7270613d msvcr80!free+0xcd
[...]
```

We recognize that value as Unicode as an example of **Wild Pointer** (page 1151), but parameters need not be pointers in a general case. We can also consider **Invalid Handle** (page 574) pattern as another specialization of **Invalid Parameter** pattern.

## Invalid Pointer

### General

**Invalid Pointer** pattern is just a number saved in a register or in a memory location, and when we try to interpret it as a memory address itself and follow it (dereference) to fetch memory contents (value) it points to, OS with the help of hardware tells us that the address doesn't exist or inaccessible due to security restrictions.

In Windows, we have our process memory partitioned into two big regions: kernel space and process space. Space partition is a different concept than execution mode (kernel or user, ring 0 or ring 3) which is a processor state. Code executing in kernel mode (a driver or OS, for example) can access memory that belongs to user space.

Based on this, we can make the distinction between invalid pointers containing kernel space addresses (starting from 0x80000000 on x86, no /3Gb switch) and invalid pointers containing user space addresses (below 0x7FFFFFFF).

On Windows x64 user space addresses are below 0x000007ffffffff, and kernel space addresses start from 0xFFFFF80000000000.

When we dereference invalid kernel space address we get a bugcheck immediately:

```
UNEXPECTED_KERNEL_MODE_TRAP (7F)
```

```
PAGE_FAULT_IN_NONPAGED_AREA (50)
```

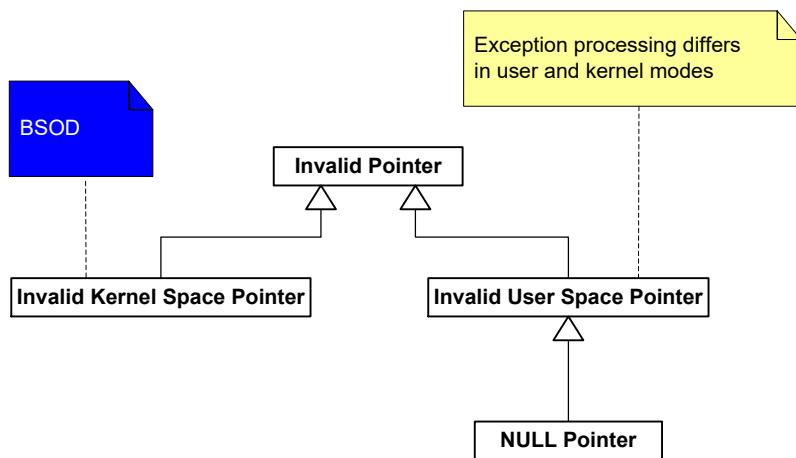
There is no way we can catch it in our code (by using SEH).

However, when we dereference user space address the course of action depends on whether our processor is in kernel mode (ring 0) or in user mode (ring 3). In any mode, we can catch the exception (by using appropriate SEH handler) or leave this to the operating system or a debugger. If there was no component willing to process the exception when it happened in user mode we get our process crash and in kernel mode, we get the following bugchecks:

```
SYSTEM_THREAD_EXCEPTION_NOT_HANDLED (7E)
```

```
KERNEL_MODE_EXCEPTION_NOT_HANDLED (8E)
```

We summarized all of this on the following UML class diagram:



NULL pointer is a special class of user space pointers. Usually, its value is in the range of 0x00000000 - 0x0000FFFF. We can see them used in instructions like

```
mov    esi, dword ptr [ecx+0x10]
```

where ecx value is 0x00000000, and we try to access the value located at 0x00000010 memory address.

When we get a crash dump, and we see an invalid pointer pattern the next step is to interpret the pointer value which should help in understanding possible steps that led to the crash.

## J

## JIT Code

## .NET

Sometimes the assembly code looks almost wild (not like generated by a favorite compiler, **Wild Code**, page 1148). Here is an example that also shows .NET runtime native unhandled exception processing:

```
0:000> kL 100
ChildEBP RetAddr
0014dbb4 77189254 ntdll!KiFastSystemCallRet
0014dbb8 75fec244 ntdll!ZwWaitForSingleObject+0xc
0014dc28 75fec1b2 kernel32!WaitForSingleObjectEx+0xbe
0014dc3c 72605389 kernel32!WaitForSingleObject+0x12
0014dc6c 726058e7 mscorewks!CIrWaitForSingleObject+0x24
0014e128 72608084 mscorewks!RunWatson+0x1df
0014e86c 7260874a mscorewks!DoFaultReportWorker+0xb59
0014e8a8 72657452 mscorewks!DoFaultReport+0xc3
0014e8cc 7265c0c7 mscorewks!WatsonLastChance+0x3f
0014e924 7265c173 mscorewks!CLRAddVectoredHandlers+0x209
0014e92c 7603f4be mscorewks!InternalUnhandledExceptionFilter+0x22
0014e9e8 771a85b7 kernel32!UnhandledExceptionFilter+0x127
0014e9f0 77139a14 ntdll!__RtlUserThreadStart+0x6f
0014ea04 771340f4 ntdll!_EH4_CallFilterFunc+0x12
0014ea2c 77189b99 ntdll!_except_handler4+0x8e
0014ea50 77189b6b ntdll!ExecuteHandler2+0x26
0014eb00 771899f7 ntdll!ExecuteHandler+0x24
0014eb00 03ca0141 ntdll!KiUserExceptionDispatcher+0xf
WARNING: Frame IP not in any known module. Following frames may be wrong.
0014ee28 634c2f42 0x3ca0141
0014ee34 67715e44 System_ni+0x132f42
0014ee70 72431b4c System_ServiceProcess_ni+0x25e44
0014ee80 724421f9 mscorewks!CallDescrWorker+0x33
0014ef00 72456571 mscorewks!CallDescrWorkerWithHandler+0xa3
0014f03c 724565a4 mscorewks!MethodDesc::CallDescr+0x19c
0014f058 724565c2 mscorewks!MethodDesc::CallTargetWorker+0x1f
0014f070 724afac5 mscorewks!MethodDescCallSite::CallWithValueTypes+0x1a
0014f1d4 724af9e5 mscorewks!ClassLoader::RunMain+0x223
0014f43c 724aff35 mscorewks!Assembly::ExecuteMainMethod+0xa6
0014f90c 724b011f mscorewks!SystemDomain::ExecuteMainMethod+0x456
0014f95c 724b004f mscorewks!ExecuteEXE+0x59
0014f9a4 72f57c24 mscorewks!_CorExeMain+0x15c
0014f9b4 75fe4911 mscoree!_CoreExeMain+0x2c
0014f9c0 7716e4b6 kernel32!BaseThreadInitThunk+0xe
0014fa00 7716e489 ntdll!__RtlUserThreadStart+0x23
0014fa18 00000000 ntdll!_RtlUserThreadStart+0x1b
```

We set exception context:

```
0:000> kv 100
ChildEBP RetAddr  Args to Child
[...]
0014e9e8 771a85b7 0014ea18 77139a14 00000000 kernel32!UnhandledExceptionFilter+0x127 (FPO: [SEH])
[...]

0:000> .expr 0014ea18

----- Exception record at 0014eb18:
ExceptionAddress: 03ca0141
    ExceptionCode: c0000005 (Access violation)
    ExceptionFlags: 00000000
NumberParameters: 2
    Parameter[0]: 00000000
    Parameter[1]: 00000000
Attempt to read from address 00000000

----- Context record at 0014eb34:
eax=00000001 ebx=08394ff8 ecx=00000000 edx=00000001 esi=056a2a94 edi=00000000
eip=03ca0141 esp=0014ee00 ebp=0014ee28 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010246
03ca0141 3909 cmp dword ptr [ecx],ecx ds:0023:00000000=????????
```

Then we disassemble the code at the crash point, and it looks strange because it includes calls through DS data segment:

```
0:000> .asm no_code_bytes
Assembly options: no_code_bytes

0:000> u 03ca0141
03ca0141 cmp    dword ptr [ecx],ecx
03ca0143 call   dword ptr ds:[36067C0h]
03ca0149 mov    ecx,dword ptr [esi+5Ch]
03ca014c cmp    dword ptr [ecx],ecx
03ca014e call   dword ptr ds:[3606D10h]
03ca0154 mov    dword ptr [ebp-1Ch],0
03ca015b mov    dword ptr [ebp-18h],0FCCh
03ca0162 push   3CA0180h
```

However further disassembly finally reaches RET instruction:

```
0:000> u
03ca0167 jmp    03ca0169
03ca0169 movzx  edx,byte ptr [ebp-24h]
03ca016d mov    ecx,dword ptr [ebp-28h]
03ca0170 call   System_ServiceProcess_ni+0x25140 (67715140)
03ca0175 pop    eax
03ca0176 jmp    eax
03ca0178 lea    esp,[ebp-0Ch]
03ca017b pop    ebx
```

```
0:000> u
03ca017c pop    esi
03ca017d pop    edi
03ca017e pop    ebp
03ca017f ret
[...]
```

Backward disassembling shows the matching function prolog code:

```
0:000> ub 03ca0141
03ca0127 movzx  eax,byte ptr [ebp-24h]
03ca012b test   eax,eax
03ca012d je     03ca0154
03ca012f cmp    dword ptr [esi+60h],0
[...]

0:000> ub 03ca0127
03ca0114 push   esi
03ca0115 push   ebx
03ca0116 sub    esp,1Ch
03ca0119 xor    eax,eax
03ca011b mov    dword ptr [ebp-18h],eax
03ca011e mov    dword ptr [ebp-28h],ecx
03ca0121 mov    dword ptr [ebp-24h],edx
03ca0124 mov    esi,dword ptr [ebp-28h]

0:000> ub 03ca0114
03ca0102 retf
03ca0103 add    eax,dword ptr [eax+36h]
03ca0106 retf
03ca0107 add    ebx,dword ptr [esi+esi-35h]
03ca010b add    esi,esp
03ca010d cmp    eax,8B550360h
03ca0112 in    al,dx
03ca0113 push   edi
```

From the stack trace, we suspect this code as JIT-compiled .NET code of the the main assembly method. And indeed, we can find the similar call signatures in the following MSDN article “Drill Into .NET Framework Internals to See How the CLR Creates Runtime Objects”<sup>122</sup>:

```
03ca0141 cmp    dword ptr [ecx],ecx
03ca0143 call   dword ptr ds:[36067C0h]
```

## Comments

---

**!IP2MD** extension command from SOS will give us method name, class and module addresses for 0x3ca0141.

---

<sup>122</sup> <http://web.archive.org/web/20150515023057/https://msdn.microsoft.com/en-us/magazine/cc163791.aspx>

## Java

JIT compiling is not restricted to .NET in Windows and we decided to add Java variant of **JIT Code (.NET)** analysis pattern (page 591). Here's one thread example from *java.exe* process memory dump:

```
0:071> k
# ChildEBP RetAddr
00 536cf424 770c15ce ntdll!NtWaitForSingleObject+0x15
01 536cf490 76f31194 KERNELBASE!WaitForSingleObjectEx+0x98
02 536cf4a8 76f31148 kernel32!WaitForSingleObjectExImplementation+0x75
03 536cf4bc 59207cb3 kernel32!WaitForSingleObject+0x12
WARNING: Stack unwind information not available. Following frames may be wrong.
04 536cf4e4 5918dbb1 jvm!JVM_FindSignal+0x5833
05 536cf558 03b6db25 jvm!JVM_Clone+0x30161
06 536cf588 03c4b0f4 0x3b6db25
07 536cf690 0348339a 0x3c4b0f4
08 536cf7d8 034803d7 0x348339a
09 536cf7e4 591a0732 0x34803d7
0a 536cf870 75bb9cde jvm!JVM_Clone+0x42ce2
0b 536cf87c 5926529e msvcrt!_VEC_memzero+0x82
0c 536cf8c4 591a1035 jvm!JVM_FindSignal+0x62e1e
0d 536cf908 591a1097 jvm!JVM_Clone+0x435e5
0e 536cf978 5914c49f jvm!JVM_Clone+0x43647
0f 536cf9d4 591c22dc jvm!jio_printf+0xaf
10 536cfa20 591c2d37 jvm!JVM_Clone+0x6488c
11 536cfa58 592071e9 jvm!JVM_Clone+0x652e7
12 536cf98 5d34c556 jvm!JVM_FindSignal+0x4d69
13 536cfcd0 5d34c600 msvcr100!_endthreadex+0x3f
14 536cfcdc 76f3338a msvcr100!_endthreadex+0xce
15 536cfce8 77829902 kernel32!BaseThreadInitThunk+0xe
16 536cfcd8 778298d5 ntdll!__RtlUserThreadStart+0x70
17 536cfcd40 00000000 ntdll!_RtlUserThreadStart+0x1b
```

We see that the return addresses are indeed return addresses saved on the stack with the preceding call instruction:

```
0:071> ub 03b6db25
03b6db03 50          push    eax
03b6db04 57          push    edi
03b6db05 e876586455  call    jvm!JVM_Clone+0x55930 (591b3380)
03b6db0a 83c408       add     esp,8
03b6db0d 8d9730010000 lea     edx,[edi+130h]
03b6db13 891424       mov     dword ptr [esp],edx
03b6db16 c7876c01000004000000 mov     dword ptr [edi+16Ch],4
03b6db20 e8dbff6155  call    jvm!JVM_Clone+0x300b0 (5918db00)
```

```
0:071> ub 03c4b0f4
03c4b0cd 891c24      mov    dword ptr [esp],ebx
03c4b0d0 894c2404    mov    dword ptr [esp+4],ecx
03c4b0d4 899c2480000000 mov    dword ptr [esp+80h],ebx
03c4b0db 898c2484000000 mov    dword ptr [esp+84h],ecx
03c4b0e2 b928b0b91a   mov    ecx,1AB9B028h
03c4b0e7 89bc248c000000 mov    dword ptr [esp+8Ch],edi
03c4b0ee 90          nop
03c4b0ef e8ac29f2ff   call   03b6daa0

0:071> ub 034803d7
034803c6 89049c      mov    dword ptr [esp+ebx*4],eax
034803c9 43          inc    ebx
034803ca 49          dec    ecx
034803cb 75f5        jne    034803c2
034803cd 8b5d14      mov    ebx,dword ptr [ebp+14h]
034803d0 8b4518      mov    eax,dword ptr [ebp+18h]
034803d3 8bf4        mov    esi,esp
034803d5 ffd0        call   eax

0:071> ub 591a0732
jvm!JVM_Clone+0x42ccc:
591a071c 57          push   edi
591a071d 89461c      mov    dword ptr [esi+1Ch],eax
591a0720 e8ab110000   call   jvm!JVM_Clone+0x43e80 (591a18d0)
591a0725 6a08        push   8
591a0727 6a06        push   6
591a0729 57          push   edi
591a072a 894514      mov    dword ptr [ebp+14h],eax
591a072d e86e9af2ff   call   jvm+0x6a1a0 (590ca1a0)
```

## Last Error Collection

Sometimes a dump file looks normal inside, and we don't see any suspicious past activity. However, as it often happens, the dump was saved manually as a response to some failure. This pattern might help in finding further troubleshooting suggestions. If we have a process memory dump we can get all errors and NTSTATUS values at once using !gle command with -all parameter:

```
0:000> !gle -all
Last error for thread 0:
LastErrorValue: (Win32) 0x3e5 (997) - Overlapped I/O operation is in progress.
LastStatusValue: (NTSTATUS) 0x103 - The operation that was requested is pending completion.

Last error for thread 1:
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0

Last error for thread 2:
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0

Last error for thread 3:
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0

...
Last error for thread 28:
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0

Last error for thread 29:
LastErrorValue: (Win32) 0x6ba (1722) - The RPC server is unavailable.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0

Last error for thread 2a:
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0

Last error for thread 2b:
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0

...
```

For complete memory dumps we can employ the following command or similar to it:

```
!for_each_thread ".thread /r /p @#Thread; .if (@$teb != 0) {!teb; !gle;}"
0: kd> !for_each_thread ".thread /r /p @#Thread; .if (@$teb != 0) { !teb; !gle; }"
...
Implicit thread is now 8941eb40
Implicit process is now 8a4ac498
Loading User Symbols
TEB at 7ff3e000
ExceptionList: 0280ffa8
StackBase: 02810000
StackLimit: 0280b000
SubSystemTib: 00000000
FiberData: 00001e00
ArbitraryUserPointer: 00000000
Self: 7ff3e000
EnvironmentPointer: 00000000
ClientId: 00001034 . 000012b0
RpcHandle: 00000000
Tls Storage: 00000000
PEB Address: 7ffdde000
LastErrorValue: 0
LastStatusValue: c00000a3
Count Owned Locks: 0
HardErrorMode: 0
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0xc00000a3 - {Drive Not Ready} The drive is not ready for use; its door may
be open. Please check drive %hs and make sure that a disk is inserted and that the drive door is closed.
...

```

## Comments

In case of **Virtualized Processes** (page 1068) **!gle** may show Win32 last error and status values incorrectly:

```
0:000:x86> !gle
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0
Wow64 TEB status:
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0

```

```
0:000:x86> !teb
Wow64 TEB32 at 00000000ffffde000
ExceptionList: 00000000002fb108
StackBase:    0000000000390000
StackLimit:   0000000000255000
SubSystemTib: 0000000000000000
FiberData:    0000000000001e00
ArbitraryUserPointer: 0000000000000000
Self:        00000000ffffde000
EnvironmentPointer: 0000000000000000
ClientId:    00000000000016d8 . 00000000000011e0
RpcHandle:   0000000000000000
Tls Storage: 0000000000e12978
PEB Address: 00000000ffffdf000
LastErrorValue: 38
LastStatusValue: c0000011
Count Owned Locks: 0
HardErrorMode: 0

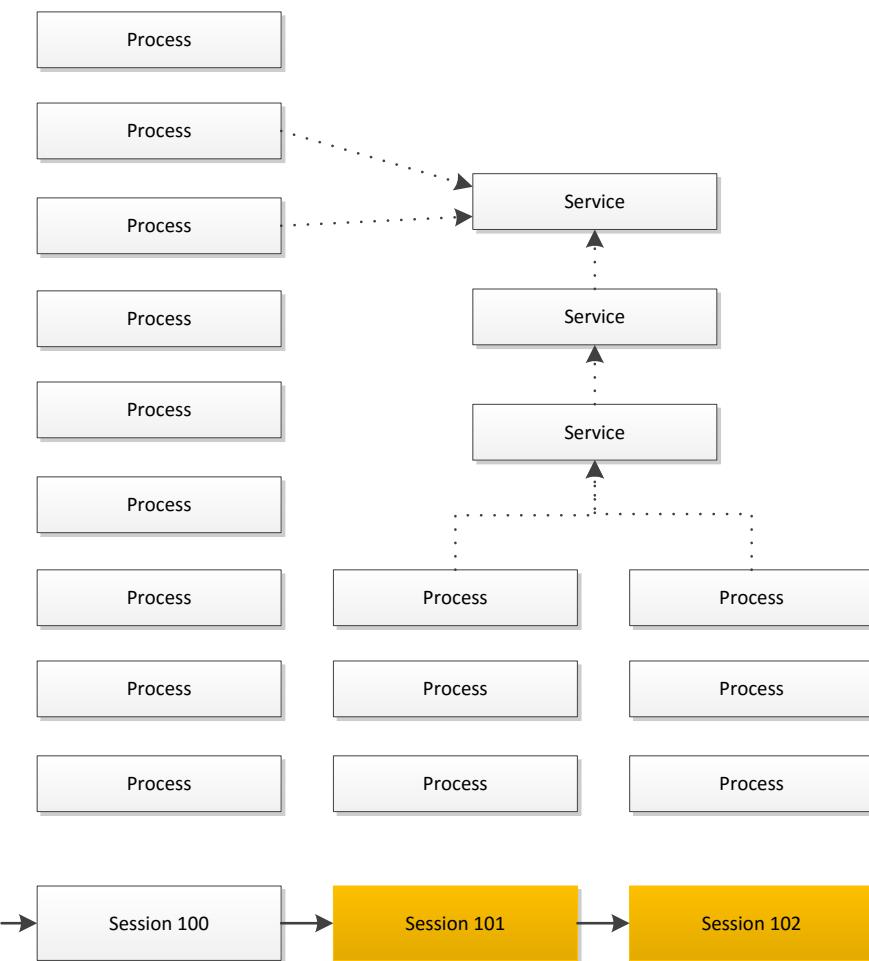
Wow64 TEB at 00000000ffffdc000
ExceptionList: 00000000ffffde000
StackBase:    00000000008fd30
StackLimit:   000000000083000
SubSystemTib: 0000000000000000
FiberData:    0000000000001e00
ArbitraryUserPointer: 0000000000000000
Self:        00000000ffffdc000
EnvironmentPointer: 0000000000000000
ClientId:    00000000000016d8 . 00000000000011e0
RpcHandle:   0000000000000000
Tls Storage: 0000000000000000
PEB Address: 00000000ffffd6000
LastErrorValue: 0
LastStatusValue: 0
Count Owned Locks: 0
HardErrorMode: 0
```

## Last Object

Although in the case of system hangs we, usually, recommend dumping **Stack Trace Collection** (page 943), in some cases, it is very time-consuming, especially when it involves thousands of processes such as in modern terminal services environments. In such a case, if the problem description indicates the last action such as a not progressing user logon or a recently launched process we first check the tail of the corresponding linked list where **Last Object** is usually added to the tail of the list:



Sometimes we can simply check the end of some enumerated collection, such as sessions (dotted lines represent ALPC **Wait Chains**, page 1097):



This analysis pattern can be added to the first tier of RSDP<sup>123</sup>. If nothing found around a couple of **Last Objects**, we then resort to the analysis of entire linked lists.

## Comments

---

Regarding many terminal sessions on Windows we can dump processes sorted by session via **!sprocess -4** to spot the last **Incomplete Session** (page 496).

---

<sup>123</sup> Rapid Software Diagnostics Process (RSDP), Memory Dump Analysis Anthology, Volume 7, page 444

## Late Crash Dump

This is a binary opposition counterpart to **Early Crash Dump** pattern (page 312). It is usually saved when patterns representing a problem such as an exception thread stack trace are already gone. Most often we see one thread with process termination functions (**Special Stack Trace** pattern, page 882):

```
0:000> ~*k
ChildEBP RetAddr
0037fcf0 770bd55c ntdll!ZwTerminateProcess+0x12
0037fd0c 750f79f4 ntdll!RtlExitUserProcess+0x85
0037fdf8 750f339a kernel32!ExitProcessStub+0x12
0037fe04 770a9ef2 kernel32!BaseThreadInitThunk+0xe
0037fe44 770a9ec5 ntdll!__RtlUserThreadStart+0x70
0037fe5c 00000000 ntdll!_RtlUserThreadStart+0x1b

0:000> ~*k
ChildEBP RetAddr
0032faf0 77a9d55c ntdll!ZwTerminateProcess+0x12
0032fb0c 775579f4 ntdll!RtlExitUserProcess+0x85
0032fb20 74ac1720 kernel32!ExitProcessStub+0x12
0032fb28 74ac1a03 msrvcr80!__crtExitProcess+0x14
0032fb64 74ac1a4b msrvcr80!_cinit+0x101
0032fb74 01339bb3 msrvcr80!exit+0xd
0032fbf8 7755339a App!_tmainCRTStartup+0x155
0032fc04 77a89ef2 kernel32!BaseThreadInitThunk+0xe
0032fc44 77a89ec5 ntdll!__RtlUserThreadStart+0x70
0032fc5c 00000000 ntdll!_RtlUserThreadStart+0x1b
```

However, sometimes it is possible to see some **Execution Residue** (page 371) left on a raw stack such as **Hidden Exceptions** (page 457), **Module Hints** (page 696), error codes and **Handled Exceptions** (page 434) that might shed light on possible problem causes.

Another variant of this pattern is when a memory dump is saved after a problem message box is dismissed, or potentially disastrous exceptions such as access violations are handled until the fault in exception handling mechanism or severe corruption resulted in unresponsive process or system (hang).

## Lateral Damage

### Linux

This is a Linux variant of **Lateral Damage** pattern previously described for Windows (page 603) platform. It also covers memory dumps where some usual commands may not work, and we have to find a workaround to simulate their output, for example, by using other commands:

```
(gdb) info threads  
Cannot find new threads: generic error

(gdb) thread apply all bt  
Cannot find new threads: generic error

(gdb) thread 2  
[Switching to thread 2 (LWP 12567)]  
#0 0x0000000042ff51 in nanosleep ()

(gdb) thread 3  
[Switching to thread 3 (LWP 12566)]  
#0 0x0000000041482e in _int_malloc ()
```

## Windows

After looking at one dump where all thread environment blocks were zeroed, import table was corrupt and recalling some similar cases encountered previously we came up with the next pattern: **Lateral Damage**.

When this problem happens, we don't have much choice, and our first temptation is to apply **Alien Component** anti-pattern<sup>124</sup> unless our module list is corrupt and we have a manifestation of another common pattern **Corrupt Dump** (page 137).

An anti-pattern is not always a bad solution if complemented by subsequent verification and backed by experience. If we get a damaged process and thread structures, we can point to a suspicious component (supported by some evidence from the raw stack analysis and educated guess) and then request additional crash dumps in hope to get less damaged process space or see that component again. At the very end, if removing suspicious component stabilizes the customer environment it proves that we were right.

## Comments

Sometimes we get some vital information missing for default or usual analysis techniques, and we need to seek other ways to look at memory dump data - this is the essence of **Lateral Damage** pattern.

In the case of process memory dumps where **.ecxr** doesn't work we may try **.exr -1** to display the last exception record which may show an exception address, code, and parameters (such as a write or read address).

```
0: ??> ~0s
...
WARNING: Many commands will not work
...

0:000> .ecxr
Unable to get exception context, HRESULT 0x80004005

0:000> .exr -1
ExceptionAddress: ... (ModuleA!func+offset)
```

<sup>124</sup> Alien Component, Memory Dump Analysis Anthology, Volume 1, page 493

## Least Common Frame

Sometimes simple comparison of **Crash Signatures** (page 153) is not enough to find similar support incidents. We then traverse stack trace frames to find a least common frame matching similar stack traces in a database. For example, consider this signature:

```
0:026> r
eax=011349ec ebx=01136738 ecx=79f943e1 edx=00000000 esi=011349ec edi=0888f3b8
eip=00dfbef8 esp=0888f348 ebp=0888f3c8 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
00dfbef8 3902 cmp dword ptr [edx],eax ds:0023:00000000=???????
```

```
0:026> k 100
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
0888f3c8 792a842f 0xdfbeef8
0888f3e4 792a839b mscorelib_ni+0x1e842f
0888f3fc 79e71b4c mscorelib_ni+0x1e839b
0888f40c 79e821b9 mscorewks!CallDescrWorker+0x33
0888f48c 79e8281f mscorewks!CallDescrWorkerWithHandler+0xa3
0888f4ac 79e82860 mscorewks!DispatchCallBody+0x1e
0888f510 79e828d1 mscorewks!DispatchCallDebuggerWrapper+0x3d
0888f544 79ec50f5 mscorewks!DispatchCallNoEH+0x51
0888f5a0 79e9848f mscorewks!AddTimerCallback_Worker+0x66
0888f5b4 79e9842b mscorewks!Thread::DoADCallBack+0x32a
0888f648 79e98351 mscorewks!Thread::ShouldChangeAbortToUnload+0xe3
0888f684 79e984dd mscorewks!Thread::ShouldChangeAbortToUnload+0x30a
0888f6ac 79ec4a84 mscorewks!Thread::ShouldChangeAbortToUnload+0x33e
0888f6c4 79ec5075 mscorewks!ManagedThreadBase::ThreadPool+0x13
0888f70c 79ec50a4 mscorewks!AddTimerCallbackEx+0x83
0888f720 79ec514a mscorewks!AddTimerCallback+0x10
0888f75c 79ec4e0c mscorewks!ThreadpoolMgr::AsyncTimerCallbackCompletion+0x64
0888f7a8 79ec471e mscorewks!UnManagedPerAppDomainTPCount::DispatchWorkItem+0x9a
0888f7bc 79ec4892 mscorewks!ThreadpoolMgr::ExecuteWorkRequest+0xaf
0888f814 79f75715 mscorewks!ThreadpoolMgr::WorkerThreadStart+0x20b
0888fffb4 7c80b729 mscorewks!Thread::intermediateThreadProc+0x49
0888ffec 00000000 kernel32!BaseThreadStart+0x37
```

Most likely we won't find any similar stack trace when searching for *0xdfbef8*. The search for *mscorelib\_ni+0x1e842f* brings several results, but they are not crashes but hangs with the frame in the middle of a call stack. The same is for *mscorelib\_ni+0x1e839b*. So we finally try searching a problem database for *CallDescrWorker+0x33* but limit results to stack traces having the same application module name. And indeed we find the similar software incident with the same stack trace after our least common frame:

```
0:004> r
eax=00000024 ebx=03e6f738 ecx=738129d8 edx=00495ef0 esi=01a87c4c edi=019c5f1c
eip=00a92037 esp=03e6f6cc ebp=03e6f6e8 iopl=0 nv up ei pl nz na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010206
00a92037 ?? ???
```

```
0:004> k 100
*** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
03e6f6c8 737d6bb5 0xa92037
03e6f6e8 737a509f mscorel!mscorlib_ni+0x216bb5
03e6f6f8 737a834c mscorel!mscorlib_ni+0x1e509f
03e6f70c 74171b6c mscorel!mscorlib_ni+0x1e834c
03e6f71c 74182209 mscorewks!CallDescrWorker+0x33
03e6f79c 7418286f mscorewks!CallDescrWorkerWithHandler+0xa3
03e6f7bc 741828b0 mscorewks!DispatchCallBody+0xe
03e6f820 74182921 mscorewks!DispatchCallDebuggerWrapper+0x3d
03e6f854 742ced79 mscorewks!DispatchCallNoEH+0x51
03e6f8b0 7419846f mscorewks!AddTimerCallback_Worker+0x66
03e6f8c4 7419840b mscorewks!Thread::DoADCallBack+0x32a
03e6f958 74198331 mscorewks!Thread::ShouldChangeAbortToUnload+0xe3
03e6f994 741984bd mscorewks!Thread::ShouldChangeAbortToUnload+0x30a
03e6f9bc 742ce708 mscorewks!Thread::ShouldChangeAbortToUnload+0x33e
03e6f9d4 742cecf9 mscorewks!ManagedThreadBase::ThreadPool+0x13
03e6fa1c 742ced28 mscorewks!AddTimerCallbackEx+0x83
03e6fa30 742cedce mscorewks!AddTimerCallback+0x10
03e6fa6c 742cea90 mscorewks!ThreadpoolMgr::AsyncTimerCallbackCompletion+0x64
03e6fab8 742ce3a2 mscorewks!UnManagedPerAppDomainTPCount::DispatchWorkItem+0x9a
03e6facc 742ce516 mscorewks!ThreadpoolMgr::ExecuteWorkRequest+0xaf
03e6fb64 74441ec9 mscorewks!ThreadpoolMgr::WorkerThreadStart+0x20b
03e6fc84 76813677 mscorewks!Thread::intermediateThreadProc+0x49
03e6fc90 77219d72 kernel32!BaseThreadInitThunk+0xe
03e6fcfd 77219d45 ntdll!_RtlUserThreadStart+0x70
03e6fce8 00000000 ntdll!_RtlUserThreadStart+0x1b
```

## Livelock

Usually, we see **Livelock** pattern when two threads are looping while acquiring and releasing a resource but not progressing. We have these signs in selected WinDbg output below:

- **High Contention** patterns (page 1168) or context switch counts
- Increased CPU time in user and / or kernel mode (**Spiking Thread**, page 888)
- At least one **Livelocked** thread is scheduled for execution
- One of the **Livelocked** threads has an unusual priority boost
- The same thread **Stack Trace** (page 926) for both **Livelocked** threads having similar stats like spent time and context switch counts
- Zero waiting *Ticks*

```
1: kd> !locks

Resource @ 0xfffffa8008464528 Exclusively owned
Contention Count = 43743004
NumberOfExclusiveWaiters = 1
Threads: fffffa8008315b60-01<*>
Threads Waiting On Exclusive Access:
fffffa8005769660

41080 total locks, 1 locks currently held

1: kd> !running

Prcbs Current Next
1 fffff88001e68180 fffff88001e72fc0 fffffa8008315b60 .....
```

We have these stack traces from **Stack Trace Collection** (page 943):

```
THREAD fffffa8008315b60 Cid 0724.2a28 Teb: 000007ffff9c000 Win32Thread: 0000000000000000 ?????
IRP List:
fffffa80082e5990: (0006,0118) Flags: 00060000 Mdl: 00000000
Not impersonating
DeviceMap fffff8a0009f434e0
Owning Process fffffa8005726360 Image: ProcessA.exe
Attached Process N/A Image: N/A
Wait Start TickCount 522197 Ticks: 0
Context Switch Count 21665144
UserTime 00:00:40.373
KernelTime 00:02:42.791
Win32 Start Address 0x000007fef6939430
Stack Init fffff88007321db0 Current fffff88007321520
Base fffff88007322000 Limit fffff8800731c000 Call 0
Priority 8 BasePriority 6 UnusualBoost 1 ForegroundBoost 0 IoPriority 2 PagePriority 5
Child-SP RetAddr Call Site
fffffa80`07321560 fffff800`0168a992 nt!KiSwapContext+0x7a
fffffa80`073216a0 fffff800`0168ccff nt!KiCommitThreadWait+0x1d2
```

```

fffff880`07321730 fffff800`0164c242 nt!KeWaitForSingleObject+0x19f
fffff880`073217d0 fffff800`0168b5ac nt!ExpWaitForResource+0xae
fffff880`07321840 fffff880`04608d91 nt!ExAcquireResourceExclusiveLite+0x14f
fffff880`073218b0 fffff880`04609e4e DriverA!foo+0x42
[...]
fffff880`07321a10 fffff800`0199ef66 nt!IopXxxControlFile+0x607
fffff880`07321b40 fffff800`01682993 nt!NtDeviceIoControlFile+0x56
fffff880`07321bb0 00000000`76ffff2a nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ fffff880`07321c20)
00000000`03a1f488 000007fe`fd06b399 ntdll!NtDeviceIoControlFile+0xa
00000000`03a1f490 00000000`76ea610f KERNELBASE!DeviceIoControl+0x75
00000000`03a1f500 000007fe`f74f3d7c kernel32!DeviceIoControlImplementation+0x7f
[...]

THREAD fffffa8005769660 Cid 0724.10b0 Teb: 000007fffffa6000 Win32Thread: 0000000000000000 WAIT:
(WrResource) KernelMode Non-Alertable
fffffa8006661f20 SynchronizationEvent
IRP List:
fffffa8006b1ac10: (0006,0118) Flags: 00060000 Mdl: 00000000
Not impersonating
DeviceMap fffff8a009f434e0
Owning Process fffffa8005726360 Image: ProcessA.exe
Attached Process N/A Image: N/A
Wait Start TickCount 522197 Ticks: 0
Context Switch Count 21601988
UserTime 00:00:30.147
KernelTime 00:02:30.972
Win32 Start Address 0x0000007fef6939430
Stack Init fffff880071bbdb0 Current fffff880071bb520
Base fffff880071bc000 Limit fffff880071b6000 Call 0
Priority 7 BasePriority 6 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
Child-SP RetAddr Call Site
fffff880`071bb560 fffff800`0168a992 nt!KiSwapContext+0x7a
fffff880`071bb6a0 fffff800`0168ccff nt!KiCommitThreadWait+0x1d2
fffff880`071bb730 fffff800`0164c242 nt!KeWaitForSingleObject+0x19f
fffff880`071bb7d0 fffff800`0168b5ac nt!ExpWaitForResource+0xae
fffff880`071bb840 fffff880`04608d91 nt!ExAcquireResourceExclusiveLite+0x14f
fffff880`071bb8b0 fffff880`04609e4e DriverA!foo+0x42
[...]
fffff880`071bba10 fffff800`0199ef66 nt!IopXxxControlFile+0x607
fffff880`071bbb40 fffff800`01682993 nt!NtDeviceIoControlFile+0x56
fffff880`071bbbb0 00000000`76ffff2a nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ fffff880`071bbc20)
00000000`033bf708 000007fe`fd06b399 ntdll!NtDeviceIoControlFile+0xa
00000000`033bf710 00000000`76ea610f KERNELBASE!DeviceIoControl+0x75
00000000`033bf780 000007fe`f74f3d7c kernel32!DeviceIoControlImplementation+0x7f
[...]

```

In both traces, we have DriverA as **Blocking Module** (page 96).

## Local Buffer Overflow

### Linux

This is a Linux variant of **Local Buffer Overflow** pattern previously described for Mac OS X (page 609) and Windows (page 611) platforms. Most of the time simple mistakes in using memory and string manipulation functions are easily detected by the runtime. The more sophisticated example which overwrites stack trace without being detected involves overwriting indirectly via a pointer to a local buffer passed to the called function. In such cases, we might see incorrect and truncated stack traces:

```
(gdb) bt
#0  0x0000000000000000 in ?? ()
#1  0x0000000000000000 in ?? ()

(gdb) x/100a $rsp
[...]
0x7fc3dd9dece8: 0x0 0x0
0x7fc3dd9decf8: 0x0 0x0
0x7fc3dd9ded08: 0x0 0x0
0x7fc3dd9ded18: 0x0 0x0
0x7fc3dd9ded28: 0x7fc3dd9ded48 0x4005cc <procA+40>
0x7fc3dd9ded38: 0x422077654e20794d 0x7542207265676769
0x7fc3dd9ded48: 0x72656666 0x0
0x7fc3dd9ded58: 0x0 0x0
0x7fc3dd9ded68: 0x0 0x0
0x7fc3dd9ded78: 0x0 0x0
[...]
```

## Mac OS X

This is a Mac OS X / GDB counterpart to **Local Buffer Overflow** pattern. Most of the time, simple mistakes in using memory and string manipulation functions are easily detected by runtime:

```
(gdb) bt
#0 0x00007fff885e982a in __kill ()
#1 0x00007fff83288b6c in __abort ()
#2 0x00007fff8325a89f in __chk_fail ()
#3 0x00007fff8325a83e in __memcpy_chk ()
#4 0x000000010914edf3 in bar ()
#5 0x000000010914ee5e in foo ()
#6 0x000000010914ee9b in main (argc=1, argv=0x7fff68d4daf0)
```

This detection happens in a default optimized release version as well:

```
(gdb) bt
#0 0x00007fff885e982a in __kill ()
#1 0x00007fff83288b6c in __abort ()
#2 0x00007fff8325a89f in __chk_fail ()
#3 0x00007fff8325a83e in __memcpy_chk ()
#4 0x000000010f59cea8 in bar [inlined] ()
#5 0x000000010f59cea8 in foo [inlined] ()
#6 0x000000010f59cea8 in main (argc=,
argv=)
```

The more sophisticated example which overwrites stack trace without being detected involves overwriting indirectly via a pointer to a local buffer passed to a called function. In such cases, we may see incorrect and truncated stack traces:

```
(gdb) bt
#0 0x00007fff885e982a in __kill ()
#1 0x00007fff83288b6c in __abort ()
#2 0x00007fff83285070 in __stack_chk_fail ()
#3 0x000000010524de77 in foo ()
#4 0xca4000007fff64e5 in ?? ()
```

```
(gdb) bt
#0 0x00007fff885e982a in __kill ()
#1 0x00007fff83288b6c in __abort ()
#2 0x00007fff83285070 in __stack_chk_fail ()
#3 0x0000000105ad8df7 in foo ()
```

Inspection of the raw stack shows ASCII-like memory values around *foo* symbolic reference instead of expected *main* and *start* functions:

```
(gdb) info r rsp
rsp 0x7fff656d79d8 0x7fff656d79d8

(gdb) x/100a 0x7fff656d79d8
0x7fff656d79d8: 0x7fff83288b6c <__abort+193> 0x0
0x7fff656d79e8: 0x0 0xffffffffdf
0x7fff656d79f8: 0x7fff656d7a40 0x7fff656d7a80
0x7fff656d7a08: 0x7fff83285070 <__guard_setup> 0x6675426c61636f4c
0x7fff656d7a18: 0x7265764f726566 0x0
0x7fff656d7a28: 0x0 0x0
0x7fff656d7a38: 0x0 0x73205d343336325b
0x7fff656d7a48: 0x65766f206b636174 0x776f6c6672
0x7fff656d7a58: 0x0 0x0
0x7fff656d7a68: 0x0 0x343336326d7ab0
0x7fff656d7a78: 0x0 0x7fff656d7ab0
0x7fff656d7a88: 0x105ad8df7 0xb1887b8452358ac4
0x7fff656d7a98: 0x794d000000000000 0x6769422077654e20
0x7fff656d7aa8: 0x6666754220726567 0x7265
0x7fff656d7ab8: 0x0 0x0
0x7fff656d7ac8: 0x0 0x0
0x7fff656d7ad8: 0x0 0x0
0x7fff656d7ae8: 0x0 0x0
[...]
```

The modeling application source code:

```
void bar(char *buffer)
{
    char data[100] = "My New Bigger Buffer";
    memcpy (buffer, data, sizeof(data));
}

void foo()
{
    char data[10] = "My Buffer";
    bar(data);
}

int main(int argc, const char * argv[])
{
    foo();
    return 0;
}
```

## Windows

**Local Buffer Overflow pattern** is observed on x86 platforms when a local variable and a function return address and/or saved frame pointer EBP are overwritten with some data. As a result, the instruction pointer EIP becomes **Wild Pointer** (see page 1151), and we have a process crash in user mode or a bugcheck in kernel mode. Sometimes this pattern is diagnosed by looking at mismatched EBP and ESP values, and, in the case of ASCII or UNICODE buffer overflow, EIP register may contain 4-char or 2-wchar\_t value, and ESP or EBP or both registers might point at some string fragment like in the example below:

```
0:000> r
eax=000fa101 ebx=0000c026 ecx=01010001 edx=bd43a010 esi=000003e0 edi=00000000
eip=0048004a esp=0012f158 ebp=00510044 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=0038 gs=0000 efl=00000202
0048004a 0000 add     byte ptr [eax],al  ds:0023:000fa101=??

0:000> kL
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
0012f154 00420047 0x48004a
0012f158 00440077 0x420047
0012f15c 00420043 0x440077
0012f160 00510076 0x420043
0012f164 00420049 0x510076
0012f168 00540041 0x420049
0012f16c 00540041 0x540041
...
...
...
```

## Comments

We may also see “Security check failure or stack buffer overrun” c0000409 **Translated Exception** (page 1013) pointing to c0000005, for example.

## Lost Opportunity

It is important to save crash dumps at the right time, for example, when an error message box is shown<sup>125</sup>.

However, sometimes, crash dumps are saved after a visual indicator of the problem disappeared and the opportunity to see the stack trace was lost. Here is one example of a service memory dump saved manually after it became unresponsive. In the dump file there was only one thread left excluding the thread created by a debugger (shown in smaller font for visual clarity):

```
0:001> ~*kv

0 Id: a3c.700 Suspend: 1 Teb: 7ff59000 Unfrozen
ChildEBP RetAddr Args to Child
1178fd60 7c822124 77e6bad8 00000574 00000000 ntdll!KiFastSystemCallRet
1178fd64 77e6bad8 00000574 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
1178fdd4 77e6ba42 00000574 ffffffff 00000000 kernel32!WaitForSingleObjectEx+0xac
1178fde8 67e223dd 00000574 ffffffff 1178fe10 kernel32!WaitForSingleObject+0x12
1178fdfc 7c82257a 67e20000 00000000 00000001 componentA!DllInitialize+0xed
1178fe1c 7c8118b0 67e222f0 67e20000 00000000 ntdll!LdrpCallInitRoutine+0x14
1178feb8 77e53002 00000000 00000000 00000000 ntdll!LdrShutdownProcess+0x130
1178ffa4 77e53065 c0000005 77e8f3b0 ffffffff kernel32!_ExitProcess+0x43
1178ffb8 77e84277 c0000005 00000000 00000000 kernel32!ExitProcess+0x14
1178ffec 00000000 77c5de6d 0a078138 00000000 kernel32!BaseThreadStart+0x5f

# 1 Id: a3c.18bc Suspend: 1 Teb: 7ffdde000 Unfrozen
ChildEBP RetAddr Args to Child
0a6cfffc8 7c845ea0 00000005 00000004 00000001 ntdll!DbgBreakPoint
0a6cff44 00000000 00000000 00905a4d 00000003 ntdll!DbgUiRemoteBreakin+0x36
```

We also see exception code 0xc0000005 as *ExitProcess* parameter. The raw stack reveals the call to *NtRaiseHardError* function that definitely resulted in some error message box:

```
0:001> ~0s
...
0:000> !teb
TEB at 7ff59000
  ExceptionList:      1178fdc4
  StackBase:          11790000
  StackLimit:         11789000
  SubSystemTib:       00000000
  FiberData:          00001e00
  ArbitraryUserPointer: 00000000
  Self:               7ff59000
  EnvironmentPointer: 00000000
  ClientId:           00000a3c . 00000700
  RpcHandle:          00000000
  Tls Storage:        00000000
```

<sup>125</sup> Proactive Crash Dumps, Memory Dump Analysis Anthology, Volume 1, page 39

```
Peb Address:      7ffd000
LastErrorValue:   0
LastStatusValue:  c0000008
Count Owned Locks: 0
HardErrorMode:    0

0:000> dds 11789000 11790000
11789000 00000000
11789004 00000000
11789008 00000000
1178900c 00000000
11789010 00000000
...
1178f058 0a4016f4
1178f05c 1178efe0
1178f060 695040c4 <Unloaded_faultrep.dll>+0x40c4
1178f064 7ffd000
1178f068 00000000
1178f06c 0a4016f4
1178f070 0a4016f4
1178f074 00000000
1178f078 1178f06c
1178f07c 0a4016b8
1178f080 0a4016b8
1178f084 1178efa0
1178f088 7c821b74 ntdll!NtRaiseHardError+0xc
1178f08c 77e99af9 kernel32!UnhandledExceptionFilter+0x54b
1178f090 d0000144
1178f094 00000004
1178f098 00000000
1178f09c 1178f164
1178f0a0 00000001
1178f0a4 77e996a7 kernel32!UnhandledExceptionFilter+0x873
1178f0a8 00000000
1178f0ac 00000000
1178f0b0 00000000
1178f0b4 02f049f0
1178f0b8 1178f13c
1178f0bc 00000000
...
```

It was that time when the dump should have been saved. See also another example and its explanation<sup>126</sup>.

---

<sup>126</sup> Process Crash - Getting the Dump Manually, Memory Dump Analysis Anthology, Volume 1, page 624

## M

## Main Thread

When we look at a thread, and it is not in **Passive Thread** pattern list (page 793), and it looks more like **Blocked Thread** (see page 82) we may ask whether it is **Main Thread**. Every process has at least one thread of execution called main or primary thread. Most GUI applications have window message processing loop inside their main process thread. When a memory dump is saved, it is most likely that this thread is blocked waiting for window or user-defined messages to arrive and can be considered as **Passive Thread**. If we see it blocked on something else waiting for some time, we may consider the application is hanging. Here is an example of the normal *iexplore.exe* thread stack taken from a kernel dump:

```

PROCESS 88de4140 SessionId: 3 Cid: 15a8 Peb: 7ffdf000 ParentCid: 0e28
DirBase: 0a43df40 ObjectTable: 88efe008 TableSize: 852.
Image: IEXPLORE.EXE
VadRoot 88dbbcfa8 Clone 0 Private 6604. Modified 951. Locked 0.
DeviceMap 88de6408
Token e3f5ccf0
Elapsed Time 0:10:52.0281
UserTime 0:00:06.0250
KernelTime 0:00:10.0421
QuotaPoolUsage[PagedPool] 126784
QuotaPoolUsage[NonPagedPool] 197704
Working Set Sizes (now,min,max) (8347, 50, 345) (33388KB, 200KB, 1380KB)
PeakWorkingSetSize 10000
VirtualSize 280 Mb
PeakVirtualSize 291 Mb
PageFaultCount 15627
MemoryPriority FOREGROUND
BasePriority 8
CommitCharge 7440

THREAD 88ee2b00 Cid 15a8.1654 Teb: 7fffde000 Win32Thread: a2242018 WAIT: (WrUserRequest) UserMode Non-
Alertable
88f82ee0 SynchronizationEvent
Not impersonating
Owning Process 88de4140
Wait Start TickCount 104916 Elapsed Ticks: 0
Context Switch Count 100208 LargeStack
UserTime 0:00:04.0484
KernelTime 0:00:09.0859
Start Address KERNEL32!BaseProcessStartThunk (0x7c57b70c)
Stack Init be597000 Current be596cc8 Base be597000 Limit be58f000 Call 0
Priority 12 BasePriority 8 PriorityDecrement 0 DecrementCount 0
ChildEBP RetAddr
be596ce0 8042d8d7 nt!KiSwapThread+0x1b1
be596d08 a00019c2 nt!KeWaitForSingleObject+0x1a3
be596d44 a00138c5 win32k!xxxSleepThread+0x18a
be596d54 a00138d1 win32k!xxxWaitMessage+0xe
be596d5c 8046b2a9 win32k!NtUserWaitMessage+0xb
be596d5c 77e3c7cd nt!KiSystemService+0xc9

```

In the same kernel dump, there is another *iexplore.exe* process with the following main thread stack which had been blocked for 31 seconds:

```

PROCESS 8811ca00 SessionId: 21 Cid: 4d18 Peb: 7ffdf000 ParentCid: 34c8
  DirBase: 0a086ee0 ObjectTable: 87d07528 TableSize: 677.
  Image: IEXPLORE.EXE
  VadRoot 87a92ae8 Clone 0 Private 4600. Modified 227. Locked 0.
  DeviceMap 88b174e8
    Token e49508d0
    ElapsedTime 0:08:03.0062
    UserTime 0:00:01.0531
    KernelTime 0:00:10.0375
    QuotaPoolUsage[PagedPool] 120792
    QuotaPoolUsage[NonPagedPool] 198376
    Working Set Sizes (now,min,max) (7726, 50, 345) (30904KB, 200KB, 1380KB)
    PeakWorkingSetSize 7735
    VirtualSize 272 Mb
    PeakVirtualSize 275 Mb
    PageFaultCount 11688
    MemoryPriority BACKGROUND
    BasePriority 8
    CommitCharge 6498

THREAD 87ce6da0 Cid 4d18.4c68 Teb: 7fffde000 Win32Thread: a22157b8 WAIT: (Executive) KernelMode Non-Alertable
  b5bd6370 NotificationEvent
IRP List:
  885d4808: (0006,00dc) Flags: 00000014 Mdl: 00000000
Not impersonating
Owning Process 8811ca00
  Wait Start TickCount 102908 Elapsed Ticks: 2008
  Context Switch Count 130138 LargeStack
  UserTime 0:00:01.0125
  KernelTime 0:00:08.0843
Start Address KERNEL32!BaseProcessStartThunk (0x7c57b70c)
Stack Init b5bd7000 Current b5bd62f4 Base b5bd7000 Limit b5bcf000 Call 0
Priority 8 BasePriority 8 PriorityDecrement 0 DecrementCount 0
ChildEBP RetAddr
b5bd630c 8042d8d7 nt!KiSwapThread+0x1b1
b5bd6334 bf09342d nt!KeWaitForSingleObject+0x1a3
b5bd6380 bf08896f mrxsmb!SmbCeAssociateExchangeWithMid+0x24b
b5bd63b0 bf0aa0ef mrxsmb!SmbCeTranceive+0xff
b5bd6490 bf0a92df mrxsmb!SmbTransactExchangeStart+0x559
b5bd64a8 bf0a9987 mrxsmb!SmbCeInitiateExchange+0x2ac
b5bd64c4 bf0a96e2 mrxsmb!SmbCeSubmitTransactionRequest+0x124
b5bd6524 bf0ac7c3 mrxsmb!_SmbCeTransact+0x86
b5bd6608 bf104ea0 mrxsmb!MRxSmbQueryFileInformation+0x553
b5bd66b4 bf103aff rdbss!__RxInitializeTopLevelIrpContext+0x52
b5bd6784 bf10da73 rdbss!WPP_SF_ZL+0x4b
b5bd67b4 bf0a8b29 rdbss!RxCleanupPipeQueues+0x117
b5bd67d4 8041ef05 mrxsmb!MRxSmbFsdDispatch+0x118
b5bd67e8 eb833839 nt!IopfCallDriver+0x35
b5bd6890 804a8109 nt!IopQueryXxxInformation+0x164
b5bd68b0 804c7d63 nt!IoQueryFileInformation+0x19
b5bd6a4c 80456562 nt!IopParseDevice+0xe8f

```

```
b5bd6ac4 804de0c0 nt!ObpLookupObjectName+0x504
b5bd6bd4 804a929b nt!ObOpenObjectByName+0xc8
b5bd6d54 8046b2a9 nt!NtQueryFullAttributesFile+0xe7
b5bd6d54 77f88887 nt!KiSystemService+0xc9
```

```
0: kd> !whattime 0n2008
2008 Ticks in Standard Time: 31.375s
```

Main Thread need not be a GUI thread. Most input console applications have *ReadConsole* calls in normal main process thread stack:

```
0:000> kL
ChildEBP RetAddr
0012fc6c 77d20190 ntdll!KiFastSystemCallRet
0012fc70 77d27fdf ntdll!NtRequestWaitReplyPort+0xc
0012fc90 765d705c ntdll!CsrClientCallServer+0xc2
0012fd8c 76634674 kernel32!ReadConsoleInternal+0x1cd
0012fe14 765eaf6a kernel32!ReadConsoleA+0x40
0012fe7c 6ec35196 kernel32!ReadFile+0x84
0012fec0 6ec35616 MSVCR80!_read_nolock+0x201
0012ff04 6ec45928 MSVCR80!_read+0xc0
0012ff1c 6ec49e47 MSVCR80!_filbuf+0x78
0012ff54 0040100d MSVCR80!getc+0x13
0012ff5c 0040117c ConsoleTest!wmain+0xd
0012ffa0 765d3833 ConsoleTest!__tmainCRTStartup+0x10f
0012ffac 77cfa9bd kernel32!BaseThreadInitThunk+0xe
0012ffec 00000000 ntdll!_RtlUserThreadStart+0x23
```

```
0:000> kL
ChildEBP RetAddr
001cf594 77d20190 ntdll!KiFastSystemCallRet
001cf598 77d27fdf ntdll!NtRequestWaitReplyPort+0xc
001cf5b8 765d705c ntdll!CsrClientCallServer+0xc2
001cf6b4 765d6efe kernel32!ReadConsoleInternal+0x1cd
001cf740 49ecd538 kernel32!ReadConsoleW+0x47
001cf7a8 49ecd645 cmd!ReadBuffFromConsole+0xb5
001cf7d4 49ec2247 cmd!FillBuf+0x175
001cf7d8 49ec2165 cmd!GetByte+0x11
001cf7f4 49ec20d8 cmd!Lex+0x75
001cf80c 49ec207f cmd!GeToken+0x27
001cf81c 49ec200a cmd!ParseStatement+0x36
001cf830 49ec6038 cmd!Parser+0x46
001cf878 49ecc703 cmd!main+0x1de
001cf8bc 765d3833 cmd!_initterm_e+0x163
001cf8c8 77cfa9bd kernel32!BaseThreadInitThunk+0xe
001cf908 00000000 ntdll!_RtlUserThreadStart+0x23
```

## Managed Code Exception

.NET programs also crash either from defects in .NET runtime (Common Language Runtime, CLR) or from non-handled runtime exceptions in managed code executed by .NET virtual machine. The latter exceptions are re-thrown from .NET runtime to be handled by the operating system and intercepted by native debuggers. Therefore our next crash dump analysis pattern is called **Managed Code Exception**.

When we get a crash dump from .NET application, it is the dump from a native process. **!analyze -v** output can usually tell us that the exception is actually CLR exception and give us other hints to look at managed code stack (CLR stack):

```

FAULTING_IP:
kernel32!RaiseException+53
77e4bee7 5e          pop     esi

EXCEPTION_RECORD: ffffffff -- (.exr 0xffffffffffff)
ExceptionAddress: 77e4bee7 (kernel32!RaiseException+0x00000053)
  ExceptionCode: e0434f4d (CLR exception)
  ExceptionFlags: 00000001
NumberParameters: 1
Parameter[0]: 80131604

DEFAULT_BUCKET_ID: CLR_EXCEPTION

PROCESS_NAME: mmc.exe

ERROR_CODE: (NTSTATUS) 0xe0434f4d - <Unable to get error code text>

MANAGED_STACK: !dumpstack -EE
No export dumpstack found

STACK_TEXT:
05faf3d8 79f97065 e0434f4d 00000001 00000001 kernel32!RaiseException+0x53
WARNING: Stack unwind information not available. Following frames may be wrong.
05faf438 7a0945a4 023f31e0 00000000 00000000 mscorewks!DllCanUnloadNowInternal+0x37a9
05faf4fc 00f2f00a 02066be4 02085ee8 023d0df0 mscorewks!CorLaunchApplication+0x12005
05faf500 02066be4 02085ee8 023d0df0 023d0e2c 0xf2f00a
05faf504 02085ee8 023d0df0 023d0e2c 05e00dfa 0x2066be4
05faf508 023d0df0 023d0e2c 05e00dfa 023d0e10 0x2085ee8
05faf50c 023d0e2c 05e00dfa 023d0e10 05351d30 0x23d0df0
05faf510 05e00dfa 023d0e10 05351d30 023d0e10 0x23d0e2c

FOLLOWUP_IP:
mscorewks!DllCanUnloadNowInternal+37a9
79f97065 c745fcfefffff mov     dword ptr [ebp-4],0FFFFFFFEh

SYMBOL_NAME: mscorewks!DllCanUnloadNowInternal+37a9

MODULE_NAME: mscorewks

```

```
IMAGE_NAME: mscorwks.dll
```

```
PRIMARY_PROBLEM_CLASS: CLR_EXCEPTION
```

```
BUGCHECK_STR: APPLICATION_FAULT_CLR_EXCEPTION
```

Sometimes we can see *mscorwks.dll* on the raw stack or see it loaded and can find it on other thread stacks than the current one.

When we get such hints, we might want to get managed code stack as well. First, we need to load the appropriate WinDbg SOS extension (Son of Strike) corresponding to .NET runtime version. This can be done by the following command:

```
0:015> .loadby sos mscorwks
```

We can check which SOS extension version was loaded by using **.chain** command:

```
0:015> .chain
Extension DLL search Path:
...
...
...
Extension DLL chain:
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\sos: image 2.0.50727.42, API 1.0.0, built Fri Sep 23
08:27:26 2005
    [path: C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\sos.dll]
    dbghelp: image 6.6.0007.5, API 6.0.6, built Sat Jul 08 21:11:32 2006
        [path: C:\Program Files\Debugging Tools for Windows\dbghelp.dll]
    ext: image 6.6.0007.5, API 1.0.0, built Sat Jul 08 21:10:52 2006
        [path: C:\Program Files\Debugging Tools for Windows\winext\ext.dll]
    exts: image 6.6.0007.5, API 1.0.0, built Sat Jul 08 21:10:48 2006
        [path: C:\Program Files\Debugging Tools for Windows\WINXP\exts.dll]
    uext: image 6.6.0007.5, API 1.0.0, built Sat Jul 08 21:11:02 2006
        [path: C:\Program Files\Debugging Tools for Windows\winext\uext.dll]
    ntsdexts: image 6.0.5457.0, API 1.0.0, built Sat Jul 08 21:29:38 2006
        [path: C:\Program Files\Debugging Tools for Windows\WINXP\ntsdexts.dll]
```

Then we can use **!dumpstack** to dump the current stack or **!EEStack** command to dump all thread stacks. The native **Stack Trace** (page 926) would be mixed with **Managed Stack Trace** (page 624):

```
0:015> !dumpstack
OS Thread Id: 0x16e8 (15)
Current frame: kernel32!RaiseException+0x53
ChildEBP RetAddr Caller,Callee
05faf390 77e4bee7 kernel32!RaiseException+0x53, calling ntdll!RtlRaiseException
05faf3a8 79e814da mscorwks!Binder::RawGetClass+0x23, calling mscorwks!Module::LookupTypeDef
05faf3bc 79e87ff4 mscorwks!Binder::IsClass+0x21, calling mscorwks!Binder::RawGetClass
05faf3c8 79f958b8 mscorwks!Binder::IsException+0x13, calling mscorwks!Binder::IsClass
05faf3d8 79f97065 mscorwks!RaiseTheExceptionInternalOnly+0x226, calling kernel32!RaiseException
05faf438 7a0945a4 mscorwks!JIT_Throw+0xd0, calling mscorwks!RaiseTheExceptionInternalOnly
05faf4ac 7a0944ea mscorwks!JIT_Throw+0x1e, calling mscorwks!LazyMachStateCaptureState
05faf4c8 793d424e (MethodDesc 0x7924ad68 +0x2e System.Threading.WaitHandle.WaitOne(Int64, Boolean)),
```

```

calling mscorewks!WaitHandleNative::CorWaitOneNative
05faf4fc 00f2f00a (MethodDesc 0x4f97500 +0x9a
Ironring.Management.MMC.SnapinBase+MmcWindow.Invoke(System.Delegate, System.Object[])), calling
mscorwks!JIT_Throw
05faf510 05e00dfa (MethodDesc 0x4f98fd8 +0xca MyNamespace.MyClass.MyMethod(Boolean)), calling 05fc7124
05faf55c 00f62fb0 (MethodDesc 0x4f95f90 +0x16f4 MyNamespace.MyClass.MyMethod.Initialise(System.Object))
05faf740 793d912f (MethodDesc 0x7925fc70 +0x2f
System.Threading._ThreadPoolWaitCallback.WaitCallback_Context(System.Object))
05faf748 793683dd (MethodDesc 0x7913f3d0 +0x81
System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object))
05faf75c 793d9218 (MethodDesc 0x7925fc80 +0x6c
System.Threading._ThreadPoolWaitCallback.PerformWaitCallback(System.Object)), calling (MethodDesc
0x7913f3d0 +0 System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object))
05faf774 79e88f63 mscorewks!CallDescrWorker+0x33
05faf784 79e88ee4 mscorewks!CallDescrWorkerWithHandler+0xa3, calling mscorewks!CallDescrWorker
05faf804 79f20212 mscorewks!DispatchCallBody+0x1e, calling mscorewks!CallDescrWorkerWithHandler
05faf824 79f201bc mscorewks!DispatchCallDebuggerWrapper+0x3d, calling mscorewks!DispatchCallBody
05faf888 79f2024b mscorewks!DispatchCallNoEH+0x51, calling mscorewks!DispatchCallDebuggerWrapper
05faf8bc 7a07bdf0 mscorewks!Holder,2>::~Holder,2>+0xbb, calling mscorewks!DispatchCallNoEH
05faf90c 77e61d1e kernel32!WaitForSingleObjectEx+0xac, calling ntdll!ZwWaitForSingleObject
05faf91c 79ecb4a4 mscorewks!Thread::UserResumeThread+0xfb
05faf92c 79ecb442 mscorewks!Thread::DoADCallBack+0x355, calling mscorewks!Thread::UserResumeThread+0xae
05faf950 79e74afe mscorewks!Thread::EnterRuntimeNoThrow+0x9b, calling mscorewks!_EH_epilog3
05faf988 79e77fe8 mscorewks!PEImage::LoadImage+0x1e1, calling mscorewks!_SEH_epilog4
05faf9c0 79ecb364 mscorewks!Thread::DoADCallBack+0x541, calling mscorewks!Thread::DoADCallBack+0x2a5
05faf9fc 7a0e1b7e mscorewks!Thread::DoADCallBack+0x575, calling mscorewks!Thread::DoADCallBack+0x4d4
05fafa24 7a0e1bab mscorewks!ManagedThreadBase::ThreadPool+0x13, calling
mscorewks!Thread::DoADCallBack+0x550
05fafa38 7a07cae8 mscorewks!QueueUserWorkItemCallback+0x9d, calling mscorewks!ManagedThreadBase::ThreadPool
05fafa54 7a07ca48 mscorewks!QueueUserWorkItemCallback, calling
mscorewks!UnwindAndContinueRethrowHelperAfterCatch
05fafa90 7a110f08 mscorewks!ThreadpoolMgr::ExecuteWorkRequest+0x40
05fafa8 7a112328 mscorewks!ThreadpoolMgr::WorkerThreadStart+0x1f2, calling
mscorewks!ThreadpoolMgr::ExecuteWorkRequest
05fafad0 79e7839d mscorewks!EEHeapFreeInProcessHeap+0x21, calling mscorewks!EEHeapFree
05fafae0 79e782dc mscorewks!operator delete[]+0x30, calling mscorewks!EEHeapFreeInProcessHeap
05fafb14 79ecb00b mscorewks!Thread::intermediateThreadProc+0x49
05fafb48 77e65512 kernel32!FlsSetValue+0xc7, calling kernel32!_SEH_epilog
05fafb6c 75da14d0 sxs!_calloc_crt+0x19, calling sxs!calloc
05fafb80 77e65512 kernel32!FlsSetValue+0xc7, calling kernel32!_SEH_epilog
05fafb88 75da1401 sxs!_CRT_INIT+0x17e, calling sxs!_initptd
05fafb8c 75da1408 sxs!_CRT_INIT+0x185, calling kernel32!GetCurrentThreadId
05fafb9c 30403805 MMCFormsShim!D1lMain+0x15, calling MMCFormsShim!PrxDllMain
05fafbb0 30418b69 MMCFormsShim!__D1lMainCRTStartup+0x7a, calling MMCFormsShim!D1lMain
05fafbdc 75de0e4c sxs!_SxsD1lMain+0x87, calling sxs!DllStartup_CrtInit
05fafbf0 30418bf9 MMCFormsShim!__D1lMainCRTStartup+0x10a, calling MMCFormsShim!__SEH_epilog4
05fafbf4 30418c22 MMCFormsShim!__D1lMainCRTStartup+0x1d, calling MMCFormsShim!__D1lMainCRTStartup
05fafbfc 7c81a352 ntdll!LdrpCallInitRoutine+0x14
05fafc24 7c82ee8b ntdll!LdrpInitializeThread+0x1a5, calling ntdll!RtlLeaveCriticalSection
05fafc2c 7c82edec ntdll!LdrpInitializeThread+0x18f, calling ntdll!_SEH_epilog
05fafc7c 7c82ed71 ntdll!LdrpInitializeThread+0xd8, calling ntdll!RtlActivateActivationContextUnsafeFast
05fafc80 7c82ed35 ntdll!LdrpInitializeThread+0x12c, calling
ntdll!RtlDeactivateActivationContextUnsafeFast
05fafcb4 7c82edec ntdll!LdrpInitializeThread+0x18f, calling ntdll!_SEH_epilog

```

```
05fafcb8 7c827c3b ntdll!NtTestAlert+0xc
05fafcbc 7c82ecb1 ntdll!_LdrpInitialize+0x1de, calling ntdll!_SEH_epilog
05fafd10 7c82ecb1 ntdll!_LdrpInitialize+0x1de, calling ntdll!_SEH_epilog
05fafd14 7c826d9b ntdll!NtContinue+0xc
05fafd18 7c8284da ntdll!KiUserApcDispatcher+0x3a, calling ntdll!NtContinue
05faffa4 79ecaff9 mscorewks!Thread::intermediateThreadProc+0x37, calling mscorewks!_alloca_probe_16
05faffb8 77e64829 kernel32!BaseThreadStart+0x34
```

.NET language symbolic names are usually reconstructed from .NET assembly metadata.

We can examine a CLR exception and get managed stack trace by using **!PrintException** and **!CLRStack** commands, for example:

```
0:014> !PrintException
Exception object: 02320314
Exception type: System.Reflection.TargetInvocationException
Message: Exception has been thrown by the target of an invocation.
InnerException: System.Runtime.InteropServices.COMException, use !PrintException 023201a8 to see more
StackTrace (generated):
    SP      IP          Function
    075AF4FC 016BFD9A Ironring.Management.MMCM.SnapinBase+MmcWindow.Invoke(System.Delegate,
System.Object[])
    ...
    ...
    ...
    075AF740 793D87AF System.Threading._ThreadPoolWaitCallback.WaitCallback_Context(System.Object)
    075AF748 793608FD System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
    075AF760 793D8898 System.Threading._ThreadPoolWaitCallback.PerformWaitCallback(System.Object)

StackTraceString: <none>
HRESULT: 80131604

0:014> !PrintException 023201a8
Exception object: 023201a8
Exception type: System.Runtime.InteropServices.COMException
Message: Error HRESULT E_FAIL has been returned from a call to a COM component.
InnerException: <none>
StackTrace (generated):
    SP      IP          Function
    00000000 00000001 Ironring.Management.MMCM.IMMCFormsShim.HostUserControl3(System.Object,
System.Object, System.String, System.String, Int32, Int32)
    0007F724 073875B9 Ironring.Management.MMCM.FormNode.SetShimControl(System.Object)
    0007F738 053D9DDE Ironring.Management.MMCM.FormNode.set_ControlType(System.Type)
    ...
    ...
    ...

StackTraceString: <none>
HRESULT: 80004005
```

```
0:014> !CLRStack
OS Thread Id: 0x11ec (14)
ESP      EIP
075af4fc 016bfd9a Ironring.Management.MMC.SnapinBase+MmcWindow.Invoke(System.Delegate, System.Object[])
...
...
...
075af740 793d87af System.Threading._ThreadPoolWaitCallback.WaitCallback_Context(System.Object)
075af748 793608fd System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
075af760 793d8898 System.Threading._ThreadPoolWaitCallback.PerformWaitCallback(System.Object)
075af8f0 79e7be1b [GCFrame: 075af8f0]
```

**!help** command gives the list of other available SOS extension commands:

```
0:014> !help

Object Inspection

DumpObj (do)
DumpArray (da)
DumpStackObjects (dso)
DumpHeap
DumpVC
GCRoot
ObjSize
FinalizeQueue
PrintException (pe)
TraverseHeap

Examining code and stacks

Threads
CLRStack
IP2MD
U
DumpStack
EStack
GCIInfo
EHInfo
COMState
BPMD

Examining CLR data structures

DumpDomain
EEHeap
Name2EE
SyncBlk
DumpMT
DumpClass
DumpMD
Token2EE
EEVersion
```

DumpModule  
 ThreadPool  
 DumpAssembly  
 DumpMethodSig  
 DumpRuntimeTypes  
 DumpSig  
 RCWCleanupList  
 DumpIL

#### Diagnostic Utilities

VerifyHeap  
 DumpLog  
 FindAppDomain  
 SaveModule  
 GCHandles  
 GCHandleLeaks  
 VMMap  
 VMStat  
 ProcInfo  
 StopOnException (soe)  
 MinidumpMode

#### Other

#### FAQ

In the case where .NET CLR runtime is version 1.x we might get messages pointing to some .NET DLL and this could be the indication that some threads have managed code:

```
*** WARNING: Unable to verify checksum for mscorelib.dll
*** ERROR: Module load completed but symbols could not be loaded for mscorelib.dll
```

In some cases we cannot load the appropriate SOS extension automatically:

```
0:000> .loadby sos mscorewks
Unable to find module "mscorewks"
```

Then we can try SOS version 1.0

```
0:000> !clr10\sos.EEStack
Loaded Son of Strike data table version 5 from
"C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\mscorsvr.dll"
```

The following message means that the server version of CLR is used:

```
0:000> .loadby sos mscorewks
Unable to find module "mscorewks"
0:000> .loadby sos mscorsvr
```

```
0:000> !help  
SOS : Help
```

For some crash dumps we get the following message saying that *sos.dll* cannot be found:

```
0:000> .loadby sos mscorewks  
The call to LoadLibrary(C:\WIN_N0_SP\Microsoft.NET\Framework\v2.0.50727\sos) failed, Win32 error 0n126  
"The specified module could not be found."  
Please check your debugger configuration and/or network access
```

Here we need to check where *Microsoft.NET\Framework\v2.0.50727\sos.dll* is installed on our crash dump analysis host and use **.load** command:

```
0:000> .load C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\sos.dll
```

The version of WinDbg since 6.8.4.0 and **!analyze -v** command show both native and managed stack traces from .NET 64-bit application memory dump so there is no need to load SOS manually there.

## Managed Stack Trace

Typical examples of this pattern are stack traces from **!CLRStack** and **!pe** extension commands or subtraces from **!DumpStack** and **!EEStack** extension commands:

```
0:000> !pe
Exception object: 000000005a976b8
Exception type: System.FormatException
Message: Index (zero based) must be greater than or equal to zero and less than the size of the argument
list.
InnerException: <none>
StackTrace (generated):
SP IP Function
00000000D0BE40 000007FEEC2153B0
mscorlib_ni!System.Text.StringBuilder.AppendFormat(System.IFormatProvider, System.String,
System.Object[]) +0x999280
00000000D0BEE0 000007FEEB87C0FA mscorlib_ni!System.String.Format(System.IFormatProvider, System.String,
System.Object[]) +0x5a
00000000D0BF30 000007FF00AB336B ModuleA!ClassB.get() +0xeb

0:010> !DumpStack
OS Thread Id: 0x8dc (15)
Child-SP RetAddr Call Site
00000001f69e808 00000000774b4bc4 user32!ZwUserWaitMessage+0xa
00000001f69e810 00000000774b4edd user32!DialogBox2+0x274
00000001f69e8a0 0000000077502920 user32!InternalDialogBox+0x135
00000001f69e900 0000000077501c15 user32!SoftModalMessageBox+0x9b4
00000001f69ea30 000000007750146b user32!MessageBoxWorker+0x31d
00000001f69ebf0 0000000077501362 user32!MessageBoxTimeoutW+0xb3
00000001f69ecc0 000007fef1590ce7 user32!MessageBoxW+0x4e
00000001f69ed00 000007feeb0f5c59 mscorwks!DoNDirectCall__PatchGetThreadCall+0x7b
[...]
00000001f69e030 000007ff00a9ba1c ModuleA!ClassA.foo() +0x47
[...]
00000001f69fe30 000000007781c521 kernel32!BaseThreadInitThunk+0xd
00000001f69fe60 0000000000000000 ntdll!RtlUserThreadStart+0x1d
```

## Manual Dump

### Kernel

Some memory dumps are generated on purpose to troubleshoot process and system hangs. They are usually called **Manual Dumps**, manual crash dumps or manual memory dumps. Kernel, complete and kernel minidumps can be generated using the famous keyboard method described in the Microsoft article<sup>127</sup>.

The crash dump will show E2 bugcheck:

```
MANUALLY_INITIATED_CRASH (e2)
The user manually initiated this crash dump.
Arguments:
Arg1: 00000000
Arg2: 00000000
Arg3: 00000000
Arg4: 00000000
```

Various tools including Citrix SystemDump reuse E2 bug check code and its arguments. There are many other 3rd-party tools used to bugcheck Windows OS such as BANG! from OSR or NotMyFault from Sysinternals. The old one is crash.exe that loads *crashdrv.sys* and uses the following bugcheck:

```
Unknown bugcheck code (69696969)
Unknown bugcheck description
Arguments:
Arg1: 00000000
Arg2: 00000000
Arg3: 00000000
Arg4: 00000000
```

In a memory dump, we would see its characteristic **Stack Trace** (page 926) pointing to *crashdrv* module:

```
STACK_TEXT:
b5b3eb0 f61588d nt!KeBugCheck+0xf
WARNING: Stack unwind information not available. Following frames may be wrong.
b5b3ebec f61584e3 crashdrv+0x88d
b5b3ec00 8041eec9 crashdrv+0x4e3
b5b3ec14 804b328a nt!IopfCallDriver+0x35
b5b3ec28 804b40de nt!IopSynchronousServiceTail+0x60
b5b3ed00 804abd0a nt!IopXxxControlFile+0x5d6
b5b3ed34 80468379 nt!NtDeviceIoControlFile+0x28
b5b3ed34 77f82ca0 nt!KiSystemService+0xc9
0006fed4 7c5794f4 nt!NtDeviceIoControlFile+0xb
0006ff38 01001a74 KERNEL32!DeviceIoControl+0xf8
0006ff70 01001981 crash+0x1a74
```

<sup>127</sup> <http://support.microsoft.com/kb/244139>

```
0006ff80 01001f93 crash+0x1981
0006ffc0 7c5989a5 crash+0x1f93
0006fff0 00000000 KERNEL32!BaseProcessStart+0x3d
```

Sometimes various hardware buttons are used to trigger NMI and generate a crash dump when the keyboard is not available. The bugcheck will be:

```
NMI_HARDWARE_FAILURE (80)
This is typically due to a hardware malfunction. The hardware supplier should be called.
Arguments:
Arg1: 004f4454
Arg2: 00000000
Arg3: 00000000
Arg4: 00000000
```

Critical process termination such as session 0 csrss.exe is also used to force a memory dump:

```
CRITICAL_OBJECT_TERMINATION (f4)
A process or thread crucial to system operation has unexpectedly exited or been terminated.
Several processes and threads are necessary for the operation of the system; when they are terminated
(for any reason), the system can no longer function.
Arguments:
Arg1: 00000003, Process
Arg2: 8a090d88, Terminating object
Arg3: 8a090eec, Process image file name
Arg4: 80967b74, Explanatory message (ascii)
```

## Comments

If you see *myfault* module on the stack then the dump was generated by NotMyFault Sysinternals tool:

```
1: kd> k
Child-SP RetAddr Call Site
fffffa60`06bf7558 fffff800`0165712e nt!KeBugCheckEx
fffffa60`06bf7560 fffff800`0165600b nt!KiBugCheckDispatch+0x6e
fffffa60`06bf76a0 fffffa60`053da17a nt!KiPageFault+0x20b
fffffa60`06bf7830 fffffa60`053da397 myfault+0x117a
fffffa60`06bf7990 fffff800`018dd25a myfault+0x1397
fffffa60`06bf79f0 fffff800`018f5f76 nt!IopXxxControlFile+0x5da
fffffa60`06bf7b40 fffff800`01656e33 nt!NtDeviceIoControlFile+0x56
fffffa60`06bf7bb0 00000000`77525aea nt!KiSystemServiceCopyEnd+0x13
```

Some virtualized environments (**Virtualized System**, page 1075) may have their own methods to trigger a crash dump. For example, on XenServer<sup>128</sup> we can see these bugchecks:

---

<sup>128</sup> <http://support.citrix.com/article/CTX123177>

; x86:

BugCheck D1, {f001, 2, 0, f001}

DRIVER\_IRQL\_NOT\_LESS\_OR\_EQUAL (d1)

An attempt was made to access a pageable (or completely invalid) address at an interrupt request level (IRQL) that is too high. This is usually caused by drivers using improper addresses.

If kernel debugger is available get stack backtrace.

Arguments:

Arg1: 0000f001, memory referenced

Arg2: 00000002, IRQL

Arg3: 00000000, value 0 = read operation, 1 = write operation

Arg4: 0000f001, address which referenced memory

STACK\_TEXT:

805573c0 0000f001 nt!KiTrap0E+0x238

80557430 ffdfffc70 0xf001

80557450 804dcbef 0xffffdffc70

80557454 00000000 nt!KiIdleLoop+0x10

; x64:

BugCheck 1E, {fffffffffc0000005, f001, 8, f001}

KMODE\_EXCEPTION\_NOT\_HANDLED (1e)

This is a very common bugcheck. Usually the exception address pinpoints the driver/function that caused the problem. Always note this address as well as the link date of the driver/image that contains this address.

Arguments:

Arg1: ffffffff00000005, The exception code that was not handled

Arg2: 000000000000f001, The address that the exception occurred at

Arg3: 0000000000000008, Parameter 0 of the exception

Arg4: 000000000000f001, Parameter 1 of the exception

STACK\_TEXT:

fffff800`062b8358 fffff800`01896747 : nt!KeBugCheckEx

fffff800`062b8360 fffff800`018b1ce9 : nt! ?? ::FNODOBFM::`string'+0x250e7

fffff800`062b8960 fffff800`018b0ae5 : nt!KiExceptionDispatch+0xa9

fffff800`062b8b40 00000000`0000f001 : nt!KiPageFault+0x1e5

fffff800`062b8cd8 ffffffa60`02a7e685 : 0xf001

fffff800`062b8ce0 fffff800`018b6583 : intelppm!C1Idle+0x9

fffff800`062b8d10 fffff800`018b62a1 : nt!PoIdle+0x183

fffff800`062b8d80 fffff800`01a88860 : nt!KiIdleLoop+0x21

fffff800`062b8db0 00000000`fffff800 : nt!zzz\_AsmCodeRange\_End+0x4

fffff800`062b20b0 00000000`00000000 : 0xfffff800

We can also see code f001 on the following thread stack as well:

```
DRIVER_VERIFIER_DETECTED_VIOLATION (c4)
A device driver attempting to corrupt the system has been caught. This is
because the driver was specified in the registry as being suspect (by the
administrator) and the kernel has enabled substantial checking of this driver.
If the driver attempts to corrupt the system, bugchecks 0xC4, 0xC1 and 0xA will
be among the most commonly seen crashes.
Arguments:
Arg1: 0000000000000091, A driver switched stacks using a method that is not supported by
the operating system. The only supported way to extend a kernel
mode stack is by using KeExpandKernelStackAndCallout.
Arg2: 0000000000000000
Arg3: ffffff8000185bc40
Arg4: 0000000000000000

0: kd> kL 100
Child-SP RetAddr Call Site
fffff880`0891f8c8 ffffff800`01729c8a nt!KeBugCheckEx
fffff880`0891f8d0 ffffff800`01700573 nt! ?? ::FNODOBFM::`string'+0x4904
fffff880`0891f910 ffffff800`0170d8df nt!RtlDispatchException+0x33
fffff880`0891fff0 ffffff800`016d2c42 nt!KiDispatchException+0x16f
fffff880`08920680 ffffff800`016d17ba nt!KiExceptionDispatch+0xc2
fffff880`08920860 ffffff800`016ca533 nt!KiPageFault+0x23a
fffff880`089209f8 ffffff800`016a88c0 nt!memcpy+0x223
fffff880`08920a00 ffffff800`016a8638 nt!KiOpFetchBytes+0x30
fffff880`08920a30 ffffff800`0170dd6f nt!KiOpDecode+0x68
fffff880`08920a80 ffffff800`0170d896 nt!KiPreprocessFault+0x53
fffff880`08920b10 ffffff800`016d2c42 nt!KiDispatchException+0x126
fffff880`089211a0 ffffff800`016d17ba nt!KiExceptionDispatch+0xc2
fffff880`08921380 00000000`0000f001 nt!KiPageFault+0x23a
fffff880`08921518 ffffff800`016d9b4b 0xf001
fffff880`08921520 ffffff800`016d94da nt!EnlightenedSwapContext_PatchXSave+0xa8
fffff880`08921560 ffffff800`016da752 nt!KiSwapContext+0x7a
fffff880`089216a0 ffffff800`016dc8af nt!KiCommitThreadWait+0x1d2
fffff880`08921730 ffffff800`0169c1de nt!KeWaitForSingleObject+0x19f
fffff880`089217d0 ffffff800`016db8bc nt!ExpWaitForResource+0xae
fffff880`08921840 ffffff880`042fc91 nt!ExAcquireResourceExclusiveLite+0x14f
[...]
fffff880`08921a10 ffffff800`019eff16 nt!IopXxxControlFile+0x607
fffff880`08921b40 ffffff800`016d2853 nt!NtDeviceIoControlFile+0x56
fffff880`08921bb0 00000000`7755ff2a nt!KiSystemServiceCopyEnd+0x13
00000000`0343f708 00000000`00000000 0x7755ff2a
```

Another example of hardware manual dump is BugCheck 8E, {c0000005, ...

```
BUCKET_ID: IP_MISALIGNED

0: kd> .trap 0xffffffff99189ce4
ErrCode = 00000002
eax=00000115 ebx=8092596a ecx=00000000 edx=00cbff30 esi=00cbff38 edi=99189d64
eip=8092596c esp=99189d58 ebp=99189d64 iopl=0 nv up ei ng nz na pe cy
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010287
nt!NtUnmapViewOfSection+0x2:
8092596c 8c28 mov word ptr [eax],gs ds:0023:00000115=?????????

0: kd> u nt!NtUnmapViewOfSection
nt!NtUnmapViewOfSection:
8092596a e9cd8c2877    jmp driverA+0x4263c (f7bae63c)
8092596f 51             push ecx
80925970 64a124010000  mov eax,dword ptr fs:[00000124h]
80925976 8a80d7000000  mov al,byte ptr [eax+0D7h]
8092597c 3c01           cmp al,1
8092597e 56             push esi
8092597f 8b750c         mov esi,dword ptr [ebp+0Ch]
80925982 8845fc         mov byte ptr [ebp-4],al
```

The server was reported hung, and all CPUs were busy.

Memory Dump API capability is available in Windows 8.1<sup>129</sup>.

---

<sup>129</sup> <https://crashdump.wordpress.com/2014/08/04/livedump-1-0-is-available/>

## Process

Now we discuss **Manual Dump** pattern as seen in process memory dumps. It is not possible to reliably identify manual dumps here because a debugger or another process dumper might have been attached to a process noninvasively and not leaving traces of intervention so we can only rely on the following information:

### *Comment field*

```
Loading Dump File [C:\kktools\userdump8.1\x64\notepad.dmp]
User Mini Dump File with Full Memory: Only application data is available
```

```
Comment: 'Userdump generated complete user-mode minidump with Standalone function on COMPUTER-NAME'
```

### *Absence of exceptions*

```
Loading Dump File [C:\UserDumps\notepad.dmp]
User Mini Dump File with Full Memory: Only application data is available

Symbol search path is: srv*c:\mss*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows Vista Version 6000 MP (2 procs) Free x64
Product: WinNt, suite: SingleUserTS
Debug session time: Mon Dec 17 16:31:31.000 2007 (GMT+0)
System Uptime: 0 days 0:45:11.148
Process Uptime: 0 days 0:00:36.000
.....
user32!ZwUserGetMessage+0xa:
00000000`76c8e6aa c3          ret
0:000> ~*kL

. 0  Id: 1b8.ed4 Suspend: 1 Teb: 000007ff`ffffdc000 Unfrozen
Child-SP      RetAddr      Call Site
00000000`0029f618 00000000`76c8e6ea user32!ZwUserGetMessage+0xa
00000000`0029f620 00000000`ff2b6eca user32!GetMessageW+0x34
00000000`0029f650 00000000`ff2bcf8b notepad!WinMain+0x176
00000000`0029f6d0 00000000`76d7cdcd notepad!IsTextUTF8+0x24f
00000000`0029f790 00000000`76ecc6e1 kernel32!BaseThreadInitThunk+0xd
00000000`0029f7c0 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

*Wake debugger exception*

```
Loading Dump File [C:\UserDumps\notepad2.dmp]
User Mini Dump File with Full Memory: Only application data is available

Symbol search path is: srv*c:\mss*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows Vista Version 6000 MP (2 procs) Free x64
Product: WinNt, suite: SingleUserTS
Debug session time: Mon Dec 17 16:35:37.000 2007 (GMT+0)
System Uptime: 0 days 0:49:13.806
Process Uptime: 0 days 0:02:54.000
.....
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
(314.1b4): Wake debugger - code 80000007 (first/second chance not available)"
user32!ZwUserGetMessage+0xa:
00000000`76c8e6aa c3          ret
```

*Break instruction exception*

```
Loading Dump File [C:\UserDumps\notepad3.dmp]
User Mini Dump File with Full Memory: Only application data is available

Symbol search path is: srv*c:\mss*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows Vista Version 6000 MP (2 procs) Free x64
Product: WinNt, suite: SingleUserTS
Debug session time: Mon Dec 17 16:45:15.000 2007 (GMT+0)
System Uptime: 0 days 0:58:52.699
Process Uptime: 0 days 0:14:20.000
.....
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
ntdll!DbgBreakPoint:
00000000`76ecfdf0 cc          int     3

0:001> ~*kL

    0  Id: 1b8.ed4 Suspend: 1 Teb: 000007ff`ffffdc000 Unfrozen
Child-SP      RetAddr      Call Site
00000000`0029f618 00000000`76c8e6ea user32!ZwUserGetMessage+0xa
00000000`0029f620 00000000`ff2b6eca user32!GetMessageW+0x34
00000000`0029f650 00000000`ff2bcf8b notepad!WinMain+0x176
00000000`0029f6d0 00000000`76d7cdcd notepad!IsTextUTF8+0x24f
00000000`0029f790 00000000`76ecc6e1 kernel32!BaseThreadInitThunk+0xd
00000000`0029f7c0 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

```
# 1 Id: 1b8.ec4 Suspend: 1 Teb: 000007ff`ffffda000 Unfrozen
Child-SP           RetAddr          Call Site
00000000`030df798 00000000`76f633e8 ntdll!DbgBreakPoint
00000000`030df7a0 00000000`76d7cdcd ntdll!DbgUiRemoteBreakin+0x38
00000000`030df7d0 00000000`76ecc6e1 kernel32!BaseThreadInitThunk+0xd
00000000`030df800 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

The latter might also be some assertion statement in the code leading to a process crash like in the following instance of **Dynamic Memory Corruption** pattern (page 304):

```
FAULTING_IP:
ntdll!DbgBreakPoint+0
77f813b1 cc int 3

EXCEPTION_RECORD: ffffffff -- (.exr fffffffffffff)
ExceptionAddress: 77f813b1 (ntdll!DbgBreakPoint)
ExceptionCode: 80000003 (Break instruction exception)
ExceptionFlags: 00000000
NumberParameters: 3
Parameter[0]: 00000000
Parameter[1]: 09aef2ac
Parameter[2]: 09aeeee8

STACK_TEXT:
09aef0bc 77fb76aa ntdll!DbgBreakPoint
09aef0c4 77fa65c2 ntdll!RtlpBreakPointHeap+0x26
09aef2bc 77fb5367 ntdll!RtlAllocateHeapSlowly+0x212
09aef340 77fa64f6 ntdll!RtlDebugAllocateHeap+0xcb
09aef540 77fcc9e3 ntdll!RtlAllocateHeapSlowly+0x5a
09aef720 786f3f11 ntdll!RtlAllocateHeap+0x954
09aef730 786fd10e rpcrt4!operator new+0x12
09aef748 786fc042 rpcrt4!OSF_CCONNECTION::OSF_CCONNECTION+0x174
09aef79c 786fbe0d rpcrt4!OSF_CASSOCIATION::AllocateCCall+0xfa
09aef808 786fbdb3 rpcrt4!OSF_BINDING_HANDLE::AllocateCCall+0x1cd
09aef83c 786f1f2f rpcrt4!OSF_BINDING_HANDLE::GetBuffer+0x28
09aef854 786f1ee4 rpcrt4!I_RpcGetBufferWithObject+0x6e
09aef860 786f1ea4 rpcrt4!I_RpcGetBuffer+0xb
09aef86c 78754762 rpcrt4!NdrGetBuffer+0x2b
09aefab8 796d78b5 rpcrt4!NdrClientCall2+0x3f9
09aefac8 796d7821 advapi32!LsarOpenPolicy2+0x14
09aefb1c 796d8b04 advapi32!LsaOpenPolicy+0xaf
09aefb84 796d8aa9 advapi32!LookupAccountSidInternal+0x63
09aefbac 0aaf5d8b advapi32!LookupAccountSidW+0x1f
WARNING: Stack unwind information not available. Following frames may be wrong.
09aeff40 0aad1665 ComponentDLL+0x35d8b
09aeff5c 3f69264c ComponentDLL+0x11665
09aeff7c 780085bc ComponentDLL+0x264c
09aeffb4 77e5438b msrvct!_endthreadex+0xc1
09aeffec 00000000 kernel32!BaseThreadStart+0x52
```

## Comments

---

When using ProcDump, we can see the comment in WinDbg output when we load a memory dump:

```
*** procdump.exe -ma 8792
*** Manual dump'
```

Recently we added **Clone Dump** analysis pattern (page 118).

One of the questions asked:

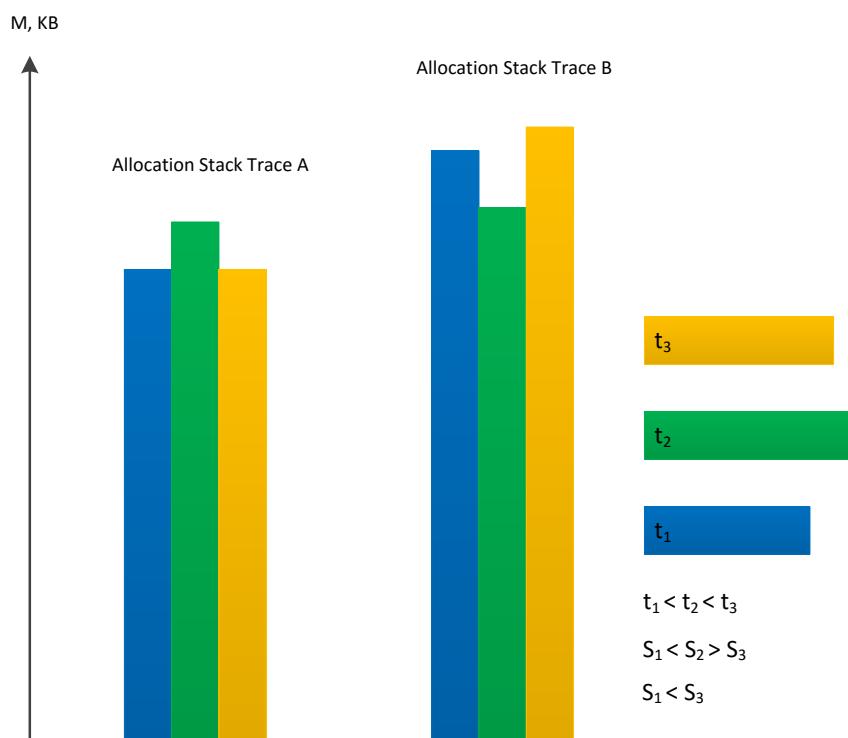
**Q.** In one of our applications we frequently face the exception “Break Instruction Exception”. We have collected the memory dumps but unable to make out why this was happening. There aren’t any debuggers or debugging tools installed on the machine since it’s a test machine. So we wonder how could this exception occur and cause this application to crash. Is it possible that this was due to a memory corruption, and the logged exception type was wrong?

**A.** This may be the case of **Multiple Exceptions** (page 714) and **Hidden Exceptions** (page 457).

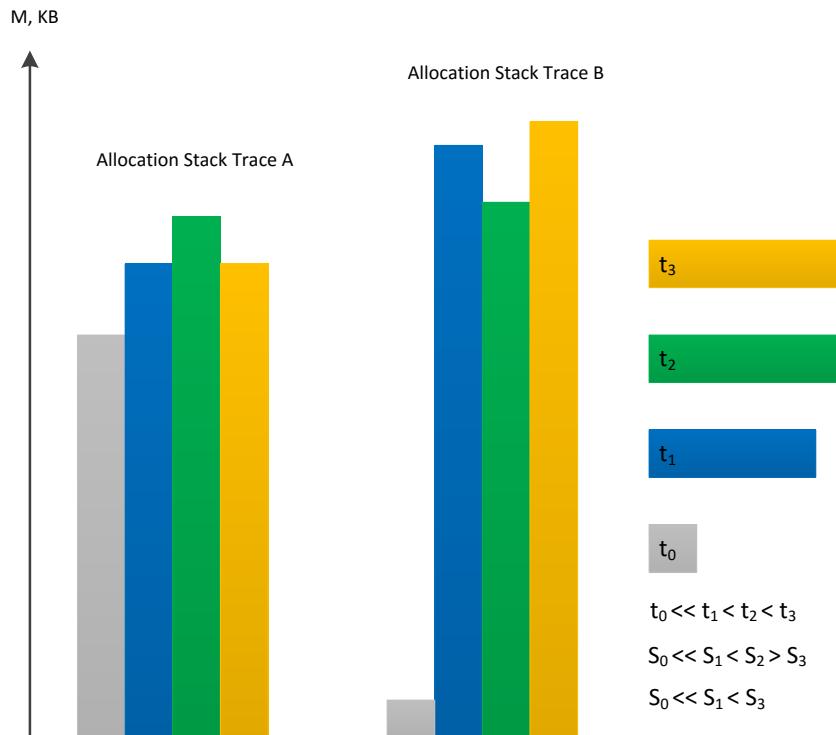
## Memory Fluctuation

### Process Heap

In process heap **Memory Leak** (page 650) pattern, we recommended acquiring sequential memory dumps spaced by 100MB. Unfortunately, customers may send memory dumps spaced more closely, say by 10 - 20 MB or less after memory consumption growth already started sometime in the past, for example, when they feel further process growth may impact their system performance. The analysis of process heap from memory dumps with enabled user mode stack database and corresponding UMDH log differences may show only **Memory Fluctuation**, where memory increases for specific stack trace allocations may follow by decreases or by small increases ( $S_i$  is for memory dump size [horizontal bars],  $t_i$  is for memory acquisition time):



In such cases, it is difficult to choose among various local memory fluctuations to continue further investigation. However, a baseline process memory dump, for example, just after process start, helps to choose which stack trace allocations investigate first: those having bigger absolute memory allocation increase (Allocation Stack Trace B):



## Comments

We may need to assess object distribution for any anomalies (see **Object Distribution Anomaly**, page 756).

## Memory Leak

### .NET Heap

Sometimes the process size constantly grows, but there is no difference in the process heap size. In such cases, we need to check whether the process uses Microsoft .NET runtime (CLR). If one of the loaded modules is *mscorwks.dll* or *mscorsvr.dll*, then it is most likely. Then we should check CLR heap statistics.

In .NET world dynamically allocated objects are garbage collected (GC) and therefore simple allocate-and-forget memory leaks are not possible. To simulate that we created the following C# program:

```
using System;

namespace CLRHeapLeak
{
    class Leak
    {
        private byte[] m_data;

        public Leak()
        {
            m_data = new byte[1024];
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Leak leak = new Leak();

            while (true)
            {
                leak = new Leak();
                System.Threading.Thread.Sleep(100);
            }
        }
    }
}
```

If we run it, the process size will never grow. GC thread will collect and free unreferenced Leak classes. This can be seen from inspecting memory dumps saved by *userdump.exe* after the start, 2, 6, and 12 minutes later. The GC heap never grows higher than 1Mb and the number of *CLRHeapLeak.Leak* and *System.Byte[]* objects always fluctuates between 100 and 500. For example, for the 12th minute we have the following statistics:

```
0:000> .loadby sos mscorewks

0:000> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x0147160c
generation 1 starts at 0x0147100c
generation 2 starts at 0x01471000
ephemeral segment allocation context: (0x014dc53c, 0x014dd618)
    segment      begin allocated      size
004aedb8 790d7ae4 790f7064 0x0001f580(128384)
01470000 01471000 014dd618 0x0006c618(443928)
Large object heap starts at 0x02471000
    segment      begin allocated      size
02470000 02471000 02473250 0x00002250(8784)
Total Size 0x8dde8(581096)
-----
GC Heap Size 0x8dde8(581096)

0:000> !dumpheap -stat
total 2901 objects
Statistics:
Count TotalSize Class Name
 1          12 System.Security.Permissions.SecurityPermission
 1          24 System.OperatingSystem
 1          24 System.Version
 1          24 System.Reflection.Assembly
 1          28 System.SharedStatics
 1          36 System.Int64[]
 1          40 System.AppDomainSetup
 3          60 System.RuntimeType
 5          60 System.Object
 2          72 System.Security.PermissionSet
 1          72 System.ExecutionEngineException
 1          72 System.StackOverflowException
 1          72 System.OutOfMemoryException
 1         100 System.AppDomain
 7          100     Free
 2         144 System.Threading.ThreadAbortException
 4          328 System.Char[]
418        5016 CLRHeapLeak.Leak
 5          8816 System.Object[]
2026       128632 System.String
418        433048 System.Byte[]
Total 2901 objects
```

However, we can make *Leak* objects always referenced by introducing the following changes into the program:

```
using System;

namespace CLRHeapLeak
{
    class Leak
    {
        private byte[] m_data;
        private Leak m_prevLeak;

        public Leak()
        {
            m_data = new byte[1024];
        }

        public Leak(Leak prevLeak)
        {
            m_prevLeak = prevLeak;
            m_data = new byte[1024];
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Leak leak = new Leak();

            while (true)
            {
                leak = new Leak(leak);
                System.Threading.Thread.Sleep(100);
            }
        }
    }
}
```

Then, if we run the program, we would see in Task Manager that it grows over time. Taking consecutive memory dumps after the start, 10 and 16 minutes, shows that Win32 heap segments have always the same size:

```
0:000> !heap 0 0
Index   Address   Name      Debugging options enabled
1: 00530000
Segment at 00530000 to 00630000 (0003d000 bytes committed)
2: 00010000
Segment at 00010000 to 00020000 (00003000 bytes committed)
3: 00520000
Segment at 00520000 to 00530000 (00003000 bytes committed)
4: 00b10000
Segment at 00b10000 to 00b50000 (00001000 bytes committed)
5: 001a0000
Segment at 001a0000 to 001b0000 (00003000 bytes committed)
```

```

6: 00170000
  Segment at 00170000 to 00180000 (00008000 bytes committed)
7: 013b0000
  Segment at 013b0000 to 013c0000 (00003000 bytes committed)

```

but GC heap and the number of *Leak* and *System.Byte[]* objects in it were growing significantly:

```

Process Uptime: 0 days 0:00:04.000

0:000> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x013c1018
generation 1 starts at 0x013c100c
generation 2 starts at 0x013c1000
ephemeral segment allocation context: (0x013cd804, 0x013cdff4)
  segment      begin allocated      size
0055ee08 790d7ae4 790f7064 0x0001f580(128384)
013c0000 013c1000 013cdff4 0x0000cff4(53236)
Large object heap starts at 0x023c1000
  segment      begin allocated      size
023c0000 023c1000 023c3250 0x000002250(8784)
Total Size 0x2e7c4(190404)
-----
GC Heap Size 0x2e7c4(190404)


```

```

0:000> !dumpheap -stat
total 2176 objects
Statistics:
Count      TotalSize Class Name
...
...
...
46          736 CLRHeapLeak.Leak
5           8816 System.Object[]
46          47656 System.Byte[]
2035        129604 System.String
Total 2176 objects

```

```
Process Uptime: 0 days 0:09:56.000
```

```

0:000> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x018cddbc
generation 1 starts at 0x01541ec4
generation 2 starts at 0x013c1000
ephemeral segment allocation context: (0x0192d668, 0x0192ddc8)
  segment      begin allocated      size
0055ee08 790d7ae4 790f7064 0x0001f580(128384)
013c0000 013c1000 0192ddc8 0x00056cdc8(5688776)
Large object heap starts at 0x023c1000
  segment      begin allocated      size
023c0000 023c1000 023c3240 0x000002240(8768)
Total Size 0x58e588(5825928)

```

```
-----
GC Heap Size 0x58e588(5825928)

0:000> !dumpheap -stat
total 12887 objects
Statistics:
Count    TotalSize Class Name
...
...
...
5        8816 System.Object[]
5403    86448 CLRHeapLeak.Leak
2026    128632 System.String
5403    5597508 System.Byte[]
Total 12887 objects

Process Uptime: 0 days 0:16:33.000

0:000> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x01c59cb4
generation 1 starts at 0x0194fd20
generation 2 starts at 0x013c1000
ephemeral segment allocation context: (0x01cd3050, 0x01cd3cc0)
segment begin allocated size
0055ee08 790d7ae4 790f7064 0x0001f580(128384)
013c0000 013c1000 01cd3cc0 0x00912cc0(9514176)
Large object heap starts at 0x023c1000
segment begin allocated size
023c0000 023c1000 023c3240 0x00002240(8768)
Total Size 0x934480(9651328)
-----
GC Heap Size 0x934480(9651328)

0:000> !dumpheap -stat
total 20164 objects
Statistics:
Count    TotalSize Class Name
5        8816 System.Object[]
2026    128632 System.String
9038    144608 CLRHeapLeak.Leak
9038    9363368 System.Byte[]
Total 20164 objects
```

This is not the traditional memory leak because we have the reference chain. However, uncontrolled memory growth can be considered as a memory leak too, caused by poor application design, bad input validation or error handling, etc.

There are situations when customers think there is a memory leak, but it is not. One of them is the unusually big size of a process when running it on a multi-processor server. If *dllhost.exe* hosting a typical .NET assembly DLL occupies less than 100Mb on a local workstation starts consuming more than 300Mb on a 4 processor server than it can be the case that the server version of CLR uses per processor GC heaps:

```
0:000> .loadby sos mscorsvr

0:000> !EEHeap -gc
generation 0 starts at 0x05c80154
generation 1 starts at 0x05c7720c
generation 2 starts at 0x102d0030
generation 0 starts at 0x179a0444
generation 1 starts at 0x1799b7a4
generation 2 starts at 0x142d0030
generation 0 starts at 0x0999ac88
generation 1 starts at 0x09990cc4
generation 2 starts at 0x182d0030
generation 0 starts at 0x242eccb0
generation 1 starts at 0x242d0030
generation 2 starts at 0x1c2d0030
...
...
...
GC Heap Size 0x109702ec(278332140)
```

or if this is CLR 1.x the old extension will tell us the same too:

```
0:000> !.\clr10\sos.eeheap -gc
Loaded Son of Strike data table version 5 from
"C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\mscorsvr.dll"
Number of GC Heaps: 4
-----
Heap 0 (0x000f9af0)
generation 0 starts at 0x05c80154
generation 1 starts at 0x05c7720c
generation 2 starts at 0x102d0030
...
...
...
Heap Size 0x515ed60(85,323,104)
-----
Heap 1 (0x000fa070)
generation 0 starts at 0x179a0444
generation 1 starts at 0x1799b7a4
generation 2 starts at 0x142d0030
...
...
...
Heap Size 0x37c7bf0(58,489,840)
-----
Heap 2 (0x000fab80)
generation 0 starts at 0x0999ac88
generation 1 starts at 0x09990cc4
generation 2 starts at 0x182d0030
...
...
...
Heap Size 0x485de34(75,882,036)
-----
Heap 3 (0x000fb448)
```

```
generation 0 starts at 0x242eccb0
generation 1 starts at 0x242d0030
generation 2 starts at 0x1c2d0030
...
...
...
Heap Size 0x41ea570(69,117,296)
-----
Reserved segments:
-----
GC Heap Size 0x1136ecf4(288,812,276)
```

The more processors we have, the more heaps are contributing to the overall VM size. Although the process occupies almost 400Mb if it doesn't grow constantly over time beyond that value then it is normal.

---

## Comments

We may also get specific objects by **!DumpHeap -mt** and then their possible roots on specific stacks via **!gcroot**.

Sometimes, there is not a leak but just some distribution anomaly (see **Object Distribution Anomaly**, page 756).

## I/O Completion Packets

This is a specialization of **Insufficient Memory** (kernel pool) pattern (page 535). The currently unique diagnostics this pattern provides in comparison to other kernel pool tags is that the pool allocation entries show the leaking process:

```
0: kd> !poolused 3
Sorting by NonPaged Pool Consumed

Pool Used:
NonPaged          Paged
Tag    Allocs   Frees   Diff   Used   Allocs   Frees   Diff   Used
Icp    1294074   42875  1251199  96642976       0       0       0       0 I/O completion packets queue
on a completion ports
[...]

0: kd> !poolfind Icp

Scanning large pool allocation table for Tag: Icp (fffffa8013e00000 : fffffa8014100000)

*fffffa800e188260 size: 50 previous size: 40 (Allocated) Icp Process: fffffa800899dc40
*fffffa800e1882e0 size: 50 previous size: 30 (Allocated) Icp Process: fffffa800899dc40
*fffffa800e188330 size: 50 previous size: 50 (Allocated) Icp Process: fffffa800899dc40
*fffffa800e188380 size: 50 previous size: 50 (Allocated) Icp Process: fffffa800899dc40
*fffffa800e1883d0 size: 50 previous size: 50 (Allocated) Icp Process: fffffa800899dc40
*fffffa800e188420 size: 50 previous size: 50 (Allocated) Icp Process: fffffa800899dc40
*fffffa800e188470 size: 50 previous size: 50 (Allocated) Icp Process: fffffa800899dc40
*fffffa800e1884c0 size: 50 previous size: 50 (Allocated) Icp Process: fffffa800899dc40

0: kd> !process fffffa800899dc40 1
PROCESS fffffa800899dc40
SessionId: 0 Cid: 43a4 Peb: 7efdf000 ParentCid: 0412
DirBase: 09d6b000 ObjectTable: fffff8a0046c8c10 HandleCount: 1068.
Image: ServiceA.exe
[...]
```

## Page Tables

Sometimes we have memory leaks related to the growing number of page tables. One reason for that could be the growing number of **Zombie Processes** (page 1158) noticeable with tens of thousands of them.

```
1: kd> !process 0 0
[...]
PROCESS ffffffa80266bd6f0
  SessionId: 0 Cid: 0a6c Peb: 7fffffff000 ParentCid: 03ac
  DirBase: 9d35a000 ObjectTable: ffffffa00170ac80 HandleCount: 152.
  Image: svchost.exe
[...]
PROCESS ffffffa8027de9b30
  SessionId: 0 Cid: 21d0 Peb: 7fffffff000 ParentCid: 02e0
  DirBase: 37881000 ObjectTable: 00000000 HandleCount: 0.
  Image: conhost.exe
[...]
PROCESS ffffffa8028eb0600
  SessionId: 0 Cid: ab88 Peb: 7fffffff000 ParentCid: 02e0
  DirBase: 27a2f000 ObjectTable: 00000000 HandleCount: 0.
  Image: conhost.exe
[...]
```

Even zombies have at least one remaining page (page directory) from the former page tables of their virtual to physical memory mapping (**!dd** is the same as **dd** WinDbg command but for physical memory):

```
1: kd> !dd 9d35a000
#9d35a000 9dd62867 03c00000 00000000 00000000
#9d35a010 00000000 00000000 00000000 00000000
#9d35a020 00000000 00000000 00000000 00000000
#9d35a030 00000000 00000000 00000000 00000000
#9d35a040 00000000 00000000 00000000 00000000
#9d35a050 00000000 00000000 00000000 00000000
#9d35a060 00000000 00000000 00000000 00000000
#9d35a070 00000000 00000000 9d45e867 49500000
```

```
1: kd> !dd 37881000
#37881000 00000000 00000000 00000000 00000000
#37881010 00000000 00000000 00000000 00000000
#37881020 00000000 00000000 00000000 00000000
#37881030 00000000 00000000 00000000 00000000
#37881040 00000000 00000000 00000000 00000000
#37881050 00000000 00000000 00000000 00000000
#37881060 00000000 00000000 00000000 00000000
#37881070 00000000 00000000 00000000 00000000
```

```
1: kd> !dd 27a2f000
#27a2f000 00000000 00000000 00000000 00000000
#27a2f010 00000000 00000000 00000000 00000000
#27a2f020 00000000 00000000 00000000 00000000
#27a2f030 00000000 00000000 00000000 00000000
#27a2f040 00000000 00000000 00000000 00000000
#27a2f050 00000000 00000000 00000000 00000000
```

```
#27a2f060 00000000 00000000 00000000 00000000
#27a2f070 00000000 00000000 00000000 00000000
```

We also see that 2 *conhost.exe* processes have identical physical to virtual mapping because their user space mappings are no longer valid (zeroed) and the *svchost.exe* process has user space mapping:

```
1: kd> !ptov 27a2f000
Amd64PtoV: pagedir 27a2f000
27a2f000 ffffff6fb`7dbed000
71530000 ffffff6fb`7dbee000
19d000 ffffff6fb`7dbef000
199000 ffffff6fb`7dbf0000
b6a04000 ffffff6fb`7dbf1000
b1f57000 ffffff6fb`7dbf2000
29c4000 ffffff6fb`7dbf3000
1c53000 ffffff6fb`7dbf5000
[...]
2e4d8000 ffffffa80`28f2d000
2c3d7000 ffffffa80`28f2e000
30ed6000 ffffffa80`28f2f000
2efd5000 ffffffa80`28f30000
2ded4000 ffffffa80`28f31000
2a5d3000 ffffffa80`28f32000
bb400000 ffffffa80`29600000 (large page)
bb200000 ffffffa80`29800000 (large page)
100000 ffffffff`ffd00000
105000 ffffffff`ffd01000
101000 ffffffff`ffd02000
102000 ffffffff`ffd03000
103000 ffffffff`ffd04000
104000 ffffffff`ffd05000
fec00000 ffffffff`ffd06000
1000 ffffffff`ffd07000
106000 ffffffff`ffd08000
123000 ffffffff`ffd09000
0 ffffffff`ffd0a000
124000 ffffffff`ffd0b000
2000 ffffffff`ffd0c000
e00c7000 ffffffff`ffd0d000
e0080000 ffffffff`ffd0e000
107000 ffffffff`ffd25000
108000 ffffffff`ffd26000
109000 ffffffff`ffd27000
10a000 ffffffff`ffd28000
10b000 ffffffff`ffd29000
10c000 ffffffff`ffd2a000
10d000 ffffffff`ffd2b000
10e000 ffffffff`ffd2c000
10f000 ffffffff`ffd2d000
110000 ffffffff`ffd2e000
111000 ffffffff`ffd2f000
112000 ffffffff`ffd30000
113000 ffffffff`ffd31000
114000 ffffffff`ffd32000
115000 ffffffff`ffd33000
```

```

116000 ffffffff`ffd34000
117000 ffffffff`ffd35000
118000 ffffffff`ffd36000
119000 ffffffff`ffd37000
11a000 ffffffff`ffd38000
11b000 ffffffff`ffd39000
11c000 ffffffff`ffd3a000
11d000 ffffffff`ffd3b000
11e000 ffffffff`ffd3c000
11f000 ffffffff`ffd3d000
120000 ffffffff`ffd3e000
121000 ffffffff`ffd3f000
122000 ffffffff`ffd40000
fee00000 ffffffff`fffe0000

```

1: kd> !ptov [37881000](#)

Amd64PtoV: pagedir 37881000

```

37881000 fffff6fb`7dbed000
8d482000 fffff6fb`7dbee000
19d000 fffff6fb`7dbef000
199000 fffff6fb`7dbf0000
b6a04000 fffff6fb`7dbf1000
b1f57000 fffff6fb`7dbf2000
29c4000 fffff6fb`7dbf3000
1c53000 fffff6fb`7dbf5000
[...]

```

[2e4d8000 ffffffa80`28f2d000](#)

```

2c3d7000 ffffffa80`28f2e000
30ed6000 ffffffa80`28f2f000
2efd5000 ffffffa80`28f30000
2ded4000 ffffffa80`28f31000
2a5d3000 ffffffa80`28f32000
bb400000 ffffffa80`29600000 (large page)
bb200000 ffffffa80`29800000 (large page)
100000 ffffffff`ffd00000
105000 ffffffff`ffd01000
101000 ffffffff`ffd02000
102000 ffffffff`ffd03000
103000 ffffffff`ffd04000
104000 ffffffff`ffd05000
fec00000 ffffffff`ffd06000
1000 ffffffff`ffd07000
106000 ffffffff`ffd08000
123000 ffffffff`ffd09000
0 ffffffff`ffd0a000
124000 ffffffff`ffd0b000
2000 ffffffff`ffd0c000
e00c7000 ffffffff`ffd0d000
e0080000 ffffffff`ffd0e000
107000 ffffffff`ffd25000
108000 ffffffff`ffd26000
109000 ffffffff`ffd27000
10a000 ffffffff`ffd28000
10b000 ffffffff`ffd29000
10c000 ffffffff`ffd2a000
10d000 ffffffff`ffd2b000

```

```
10e000 ffffffff`ffd2c000
10f000 ffffffff`ffd2d000
110000 ffffffff`ffd2e000
111000 ffffffff`ffd2f000
112000 ffffffff`ffd30000
113000 ffffffff`ffd31000
114000 ffffffff`ffd32000
115000 ffffffff`ffd33000
116000 ffffffff`ffd34000
117000 ffffffff`ffd35000
118000 ffffffff`ffd36000
119000 ffffffff`ffd37000
11a000 ffffffff`ffd38000
11b000 ffffffff`ffd39000
11c000 ffffffff`ffd3a000
11d000 ffffffff`ffd3b000
11e000 ffffffff`ffd3c000
11f000 ffffffff`ffd3d000
120000 ffffffff`ffd3e000
121000 ffffffff`ffd3f000
122000 ffffffff`ffd40000
fee00000 ffffffff`fffe0000
```

```
1: kd> !ptov 9d35a000
Amd64PtoV: pagedir 9d35a000
9E587000 10000
6871E000 20000
AF5AA000 30000
AF5AB000 31000
AFAAC000 32000
AFBAD000 33000
AF2F5000 40000
9D66B000 50000
22199000 60000
9D962000 E5000
9D261000 E6000
9DC60000 E7000
9D256000 EA000
9D84F000 EB000
9E4EC000 EC000
9E081000 ED000
9D876000 EE000
9E271000 EF000
B8BF000 F0000
B8EFE000 F1000
B86FF000 F2000
B5302000 F3000
B5202000 F4000
B5502000 F5000
B7F03000 F6000
B8404000 F7000
B8415000 100000
B8B16000 101000
B1B17000 102000
[...]
2CD4000 77512000
```

```
5D7000 77515000
5D8000 77516000
4D9000 77517000
B358F000 77590000
AEF04000 77591000
68624000 77592000
64B26000 77593000
AF4C6000 77595000
B2042000 7EFE0000
B2143000 7EFE1000
B1A56000 7EFE2000
B1A57000 7EFE3000
B1B58000 7EFE4000
1BA000 7FFE0000
9DA69000 BFEB0000
AEEAE000 FFEA0000
AF191000 FFEA1000
9D76A000 FFEA2000
AE793000 FFEA3000
9DC8E000 FFEA5000
B7EB7000 FFEA6000
9DFFC000 FFEA7000
[...]
2e4d8000 ffffffa80`28f2d000
2c3d7000 ffffffa80`28f2e000
30ed6000 ffffffa80`28f2f000
2efd5000 ffffffa80`28f30000
2ded4000 ffffffa80`28f31000
2a5d3000 ffffffa80`28f32000
bb400000 ffffffa80`29600000 (large page)
bb200000 ffffffa80`29800000 (large page)
100000 ffffffff`ffd00000
105000 ffffffff`ffd01000
101000 ffffffff`ffd02000
102000 ffffffff`ffd03000
103000 ffffffff`ffd04000
104000 ffffffff`ffd05000
fec00000 ffffffff`ffd06000
1000 ffffffff`ffd07000
106000 ffffffff`ffd08000
123000 ffffffff`ffd09000
0 ffffffff`ffd0a000
124000 ffffffff`ffd0b000
2000 ffffffff`ffd0c000
e00c7000 ffffffff`ffd0d000
e0080000 ffffffff`ffd0e000
107000 ffffffff`ffd25000
108000 ffffffff`ffd26000
109000 ffffffff`ffd27000
10a000 ffffffff`ffd28000
10b000 ffffffff`ffd29000
10c000 ffffffff`ffd2a000
10d000 ffffffff`ffd2b000
10e000 ffffffff`ffd2c000
10f000 ffffffff`ffd2d000
110000 ffffffff`ffd2e000
```

```

111000 ffffffff`ffd2f000
112000 ffffffff`ffd30000
113000 ffffffff`ffd31000
114000 ffffffff`ffd32000
115000 ffffffff`ffd33000
116000 ffffffff`ffd34000
117000 ffffffff`ffd35000
118000 ffffffff`ffd36000
119000 ffffffff`ffd37000
11a000 ffffffff`ffd38000
11b000 ffffffff`ffd39000
11c000 ffffffff`ffd3a000
11d000 ffffffff`ffd3b000
11e000 ffffffff`ffd3c000
11f000 ffffffff`ffd3d000
120000 ffffffff`ffd3e000
121000 ffffffff`ffd3f000
122000 ffffffff`ffd40000
fee00000 ffffffff`ffffe0000

```

In order to check user space virtual addresses we have to switch to the corresponding process context:

```

1: kd> !pte fffffa80`28f2d000
VA ffffffa8028f2d000
PXE at FFFF6FB7DBEDFA8 PPE at FFFF6FB7DBF5000 PDE at FFFF6FB7EA00A38 PTE at FFFF6FD40147968
contains 000000001C53863 contains 000000001C54863 contains 0000000049320863 contains 000000002E4D8963
pfn 1c53 -DA-KWEV pfn 1c54 -DA-KWEV pfn 49320 -DA-KWEV pfn 2e4d8 -G-DA-KWEV

1: kd> .process /r /p ffffffa80266bd6f0
Implicit process is now ffffffa80`266bd6f0
Loading User Symbols

1: kd> !pte 10000
VA 0000000000010000
PXE at FFFF6FB7DBED000 PPE at FFFF6FB7DA00000 PDE at FFFF6FB40000000 PTE at FFFF68000000080
contains 03C000009DD62867 contains 031000009D865867 contains 7C2000009DD66867 contains 9CB000009E587867
pfn 9dd62 -DA-UWEV pfn 9d865 -DA-UWEV pfn 9dd66 -DA-UWEV pfn 9e587 -DA-UW-V

```

This pattern came to our attention after seeing memory dumps generated after the growing number of memory allocated for page tables exceeded a gigabyte.

## Process Heap

**Memory Leak** is another pattern that may be finally manifested as **Insufficient Memory** pattern (page 523) in a crash dump. Here we cover process heap memory leaks. They are usually identified when the process virtual memory size grows over time. It starts with 80Mb and instead of fluctuating normally below 100Mb it suddenly starts growing to 150Mb after some time and then to 300Mb the next day and then grows to 600Mb, and so on.

Usually, a process heap is under suspicion here. To confirm this, we need to sample 2-3 consecutive user memory dumps at process sizes 100Mb, 200Mb, and 300Mb, for example. This can be done by using Microsoft userdump.exe command line tool. Then we can see whether there is any heap growth by using **!heap -s WinDbg** command:

### 1st dump

Heap	Flags	Reserv (k)	Commit (k)	Virt (k)
00140000 00000002		2048	1048	1112
00240000 00008000		64	12	12
00310000 00001002		7232	4308	4600
00420000 00001002		1024	520	520
00340000 00001002		256	40	40
00720000 00001002		64	32	32
00760000 00001002		64	48	48
01020000 00001002		256	24	24
02060000 00001002		64	16	16
02070000 00001003		256	120	120
020b0000 00001003		256	4	4
020f0000 00001003		256	4	4
02130000 00001003		256	4	4
02170000 00001003		256	4	4
021f0000 00001002		1088	76	76
021e0000 00001002		64	16	16
02330000 00001002		1088	428	428
02340000 00011002		256	12	12
02380000 00001002		64	12	12
024c0000 00001003		64	8	8
028d0000 00001002		7232	3756	6188
02ce0000 00001003		64	8	8
07710000 00001002		64	20	20
07b20000 00001002		64	16	16
07f30000 00001002		64	16	16
09050000 00001002		256	12	12
09c80000 00001002		130304	102340	102684
007d0000 00001003		256	192	192
00810000 00001003		256	4	4
0bdd0000 00001003		256	4	4
0be10000 00001003		256	4	4
0be50000 00001003		256	4	4
0be90000 00001003		256	56	56
0bed0000 00001003		256	4	4

0bf10000 00001003	256	4	4
0bf50000 00001003	256	4	4
0bf90000 00001003	256	4	4
00860000 00001002	64	20	20
00870000 00001002	64	20	20
0d760000 00001002	256	12	12
0dc60000 00001002	1088	220	220
0c3a0000 00001002	64	12	12
0c3d0000 00001002	1088	160	364
08420000 00001002	64	64	64

**2nd dump**

0:000> !heap -s				
Heap	Flags	Reserv (k)	Commit (k)	Virt (k)
-----				
00140000 00000002		8192	4600	4600
00240000 00008000		64	12	12
00310000 00001002		7232	4516	4600
00420000 00001002		1024	520	520
00340000 00001002		256	44	44
00720000 00001002		64	32	32
00760000 00001002		64	48	48
01020000 00001002		256	24	24
02060000 00001002		64	16	16
02070000 00001003		256	124	124
020b0000 00001003		256	4	4
020f0000 00001003		256	4	4
02130000 00001003		256	4	4
02170000 00001003		256	4	4
021f0000 00001002		1088	76	76
021e0000 00001002		64	16	16
02330000 00001002		1088	428	428
02340000 000011002		256	12	12
02380000 00001002		64	12	12
024c0000 00001003		64	8	8
028d0000 00001002		7232	3796	6768
02ce0000 00001003		64	8	8
07710000 00001002		64	20	20
07b20000 00001002		64	16	16
07f30000 00001002		64	16	16
09050000 00001002		256	12	12
09c80000 00001002	261376	221152	221928	
007d0000 00001003		256	192	192
00810000 00001003		256	4	4
0bdd0000 00001003		256	4	4
0be10000 00001003		256	4	4
0be50000 00001003		256	4	4
0be90000 00001003		256	60	60
0bed0000 00001003		256	4	4
0bf10000 00001003		256	4	4
0bf50000 00001003		256	4	4
0bf90000 00001003		256	4	4
00860000 00001002		64	20	20
00870000 00001002		64	20	20

0d760000 00001002	256	12	12
0dc60000 00001002	1088	228	228
0c3a0000 00001002	64	12	12
0c3d0000 00001002	1088	168	224
08450000 00001002	64	64	64

We see that the only significant heap growth is at 09c80000 address, from 130Mb to 260Mb. However, this doesn't say which code uses it. In order to find the code, we need to enable the so-called user mode stack trace database. Please refer to the Citrix article on how to configure it<sup>130</sup>. The example in the article is for Citrix IMA service, but we can replace *ImaSrv.exe* with any other executable name.

Suppose that after enabling user mode stack trace database and restarting the program or service we see the growth, and we get memory dumps with the following suspicious heap highlighted in bold:

```
0:000> !gflag
Current NtGlobalFlag contents: 0x00001000
ust - Create user mode stack trace database

0:000> !heap -s
NtGlobalFlag enables following debugging aids for new heaps:
  stack back traces
LFH Key: 0x2687ed29
  Heap   Flags   Reserv Commit Virt
      (k)     (k)    (k)
-----
00140000 58000062  4096   488   676
00240000 58008060   64    12    12
00360000 58001062 3136  1152  1216
003b0000 58001062   64    32    32
01690000 58001062  256    32    32
016d0000 58001062 1024   520   520
003e0000 58001062   64    48    48
02310000 58001062  256    24    24
02b30000 58001062   64    16    16
02b40000 58001063  256    64    64
02b80000 58001063  256     4     4
02bc0000 58001063  256     4     4
02c00000 58001063  256     4     4
02c40000 58001063  256     4     4
02c80000 58001063  256     4     4
02cc0000 58001063  256     4     4
02d30000 58001063   64     4     4
03140000 58001062 7232  4160  4896
03550000 58001063   64     4     4
07f70000 58001062   64    12    12
08380000 58001062   64    12    12
08790000 58001062   64    12    12
091d0000 58011062  256    12    12
```

<sup>130</sup> <http://support.citrix.com/article/CTX106970>

09210000	58001062	64	16	16
09220000	58001062	64	12	12
092a0000	58001062	64	12	12
09740000	58001062	256	12	12
0b1a0000	58001062	64	12	12
<b>0b670000</b>	<b>58001062</b>	<b>64768</b>	<b>39508</b>	<b>39700</b>
0b7b0000	58001062	64	12	12
0c650000	58001062	1088	192	192

Every heap is subdivided into several segments and to see which segments have grown the most we can use **!heap -m <heap address>** command:

```
0:000> !heap -m 0b670000
Index   Address   Name      Debugging options enabled
29: 0b670000
Segment at 0b670000 to 0b6b0000 (00040000 bytes committed)
Segment at 0c760000 to 0c860000 (00100000 bytes committed)
Segment at 0c980000 to 0cb80000 (001fe000 bytes committed)
Segment at 0cb80000 to 0cf80000 (003cc000 bytes committed)
Segment at 0dc30000 to 0e430000 (00800000 bytes committed)
Segment at 12330000 to 13330000 (01000000 bytes committed)
Segment at 13330000 to 15330000 (0078b000 bytes committed)
...
...
...
```

If we use **!heap -a <heap address>** command then in addition to the list of heap segments individual heap allocation entries will be dumped as well. This could be very big output, and we should open the log file in advance by using **.logopen <file name>** command.

The output can be like this (taken from another dump):

```
0:000> !heap -a 000a0000
...
...
...
Segment00 at 000a0000:
  Flags:          00000000
  Base:           000a0000
  First Entry:   000a0580
  Last Entry:    000b0000
  Total Pages:   00000010
  Total UnCommit: 00000002
  Largest UnCommit:00000000
  UnCommitted Ranges: (1)

  Heap entries for Segment00 in Heap 000a0000
  000a0000: 00000 . 00580 [101] - busy (57f)
  000a0580: 00580 . 00240 [101] - busy (23f)
  000a07c0: 00240 . 00248 [101] - busy (22c)
  000a0a08: 00248 . 00218 [101] - busy (200)
  000a0c20: 00218 . 00ce0 [100]
  000a1900: 00ce0 . 00f88 [101] - busy (f6a)
```

```

000a2888: 00f88 . 04418 [101] - busy (4400)
000a6ca0: 04418 . 05958 [101] - busy (5940)
000ac5f8: 05958 . 00928 [101] - busy (90c)
000acf20: 00928 . 010c0 [100]
000adfe0: 010c0 . 00020 [111] - busy (1d)
000ae000:     00002000 - uncommitted bytes.

```

Then we can inspect individual entries to see stack traces that allocated them by using **!heap -p -a <heap entry address>** command:

```

0:000> !heap -p -a 000a6ca0
address 000a6ca0 found in
_HEAP @ a0000
HEAP_ENTRY Size Prev Flags      UserPtr UserSize - state
000a6ca0 0b2b 0000  [00]    000a6cb8    05940 - (busy)
Trace: 2156ac
7704dab4 ntdll!RtlAllocateHeap+0x0000021d
75c59b12 USP10!UspAllocCache+0x0000002b
75c62381 USP10!AllocSizeCache+0x00000048
75c61c74 USP10!FindOrCreateSizeCacheWithoutRealizationID+0x00000124
75c61bc0 USP10!FindOrCreateSizeCacheUsingRealizationID+0x00000070
75c59a97 USP10!UpdateCache+0x0000002b
75c59a61 USP10!ScriptCheckCache+0x0000005c
75c59d04 USP10!ScriptStringAnalyse+0x0000012a
7711140f LPK!LpkStringAnalyse+0x00000114
7711159e LPK!LpkCharsetDraw+0x00000302
77111488 LPK!LpkDrawTextEx+0x00000044
76a4beb3 USER32!DT_DrawStr+0x0000013a
76a4be45 USER32!DT_DrawJustifiedLine+0x0000005f
76a49d68 USER32!AddEllipsisAndDrawLine+0x000000186
76a4bc31 USER32!DrawTextExWorker+0x000001b1
76a4bedc USER32!DrawTextExW+0x0000001e
746051d8 uxtheme!CTextDraw::GetTextExtent+0x000000be
7460515a uxtheme!GetThemeTextExtent+0x00000065
74611ed4 uxtheme!CThemeMenuBar::MeasureItem+0x00000124
746119c1 uxtheme!CThemeMenu::OnMeasureItem+0x0000003f
74611978 uxtheme!CThemeWnd::_PreDefWindowProc+0x00000117
74601ea5 uxtheme!_ThemeDefWindowProc+0x00000090
74601f61 uxtheme!ThemeDefWindowProcW+0x00000018
76a4a09e USER32!DefWindowProcW+0x00000068
931406 notepad!NPWndProc+0x00000084
76a51a10 USER32!InternalCallWinProc+0x00000023
76a51ae8 USER32!UserCallWinProcCheckWow+0x0000014b
76a51c03 USER32!DispatchClientMessage+0x000000da
76a3bc24 USER32!__fnINOUTLPUAHMEASUREMENUITEM+0x00000027
77040e6e ntdll!KiUserCallbackDispatcher+0x0000002e
76a51d87 USER32!RealDefWindowProcW+0x00000047
74601f2f uxtheme!_ThemeDefWindowProc+0x000001b8

```

If we want to dump all heap entries with their corresponding stack traces we can use **!heap -k -h <heap address>** command.

Sometimes all these commands do not work. In such a case, we can use old Windows 2000 extension<sup>131</sup>.

Some prefer to use *umdh.exe* and get text file logs, but the advantage of embedding heap allocation stack traces in a crash dump is that we are not concerned with sending and configuring symbol files at a customer site.

When analyzing heap various pageheap options **!heap -p** are useful such as (taken from WinDbg help):

**-t[c|s] [Traces]**

*"Causes the debugger to display the collected traces of the heavy heap users. Traces specifies the number of traces to display; the default is four. If there are more traces than the specified number, the earliest traces are displayed. If -t or -tc is used, the traces are sorted by count usage. If -ts is used, the traces are sorted by size."*

We can also use Microsoft Debug Diagnostics tool<sup>132</sup>.

---

<sup>131</sup> Heap Stack Traces, Memory Dump Analysis Anthology, Volume 1, page 182

<sup>132</sup> <http://blogs.msdn.com/debugdiag/>

## Comments

We can get distribution stats for different block sizes and then filter all stack traces based on the specific size we are interested in:

```
0:001> !heap -stat -h 06a00000
heap @ 06a00000
group-by: TOTSIZE max-display: 20
size #blocks total ( %) (percent of total busy bytes)
240 13eb9 - 31cce80 (40.67)
48 2a813 - bf4558 (9.76)
88 14213 - ab1a18 (8.73)
28 2a6a1 - 6a0928 (5.41)
4 187062 - 61c188 (4.99)
30 1e140 - 5a3c00 (4.61)
53218d 1 - 53218d (4.24)
50 fa85 - 4e4990 (4.00)
39d2d4 1 - 39d2d4 (2.95)
c 3592b - 282e04 (2.05)
3c 9591 - 230dfc (1.79)
b8 2b2b - 1f06e8 (1.58)
9c 31a9 - 1e42fc (1.54)
70 3592 - 176fe0 (1.20)
3ec 5dc - 16fad0 (1.17)
33c 5dc - 12f390 (0.97)
14 b4d5 - e20a4 (0.72)
20 6101 - c2020 (0.62)
6c 135e - 82ba8 (0.42)
60 1082 - 630c0 (0.32)

0:001> !heap -flt s 240
[...]
1c6c0db8 0055 0055 [00] 1c6c0dd0 00240 - (busy)
? ModuleA!DllUnregisterServer+272c7c
1c6c1060 0055 0055 [00] 1c6c1078 00240 - (busy)
? ModuleA!DllUnregisterServer+272c7c
1c6c1308 0055 0055 [00] 1c6c1320 00240 - (busy)
? ModuleA!DllUnregisterServer+272c7c
1c6c15b0 0055 0055 [00] 1c6c15c8 00240 - (busy)
? ModuleA!DllUnregisterServer+272c7c
1c6c1858 0055 0055 [00] 1c6c1870 00240 - (busy)
[...]
```

## Regions

The set of memory dumps that prompted to introduce **Insufficient Memory** pattern for stack trace database (page 563) also prompted to include a variant of **Memory Leak** pattern related to regions of virtual memory address space. We created this simple modeling application:

```
int _tmain(int argc, _TCHAR* argv[])
{
    int i,j;
    for (i = 1; i < 1000; ++i)
    {
        for (j = 1; j < 1000; ++j)
        {
            VirtualAlloc(NULL, 0x10000, MEM_RESERVE,
                         PAGE_EXECUTE_READWRITE);
        }
        getc(stdin);
    }
    return 0;
}
```

We allocated only reserved memory regions. Committing them would probably at some stage manifest **Insufficient Memory** patterns for committed memory (page 523) and physical memory (page 552). So we took a few consecutive memory dumps and saw the ever increasing number of regions allocated at greater and greater virtual addresses:

```
0:000> !address
[...]
* 0`04070000 0`04080000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`04080000 0`04090000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`04090000 0`040a0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`040a0000 0`040b0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`040b0000 0`040c0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`040c0000 0`040d0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`040d0000 0`040e0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`040e0000 0`040f0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`040f0000 0`04100000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`04100000 0`04110000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`04110000 0`04120000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`04120000 0`04130000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`04130000 0`04140000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`04140000 0`04260000 0`00120000 MEM_FREE PAGE_NOACCESS Free
[...]

0:000> !address
[...]
* 0`2eec0000 0`2eed0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`2eed0000 0`2eee0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`2eee0000 0`2eff0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`2eff0000 0`2eff0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`2ef00000 0`2ef10000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`2ef10000 0`2ef20000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`2ef20000 0`2ef30000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`2ef30000 0`2ef40000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`2ef40000 0`2ef50000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`2ef50000 0`2ef60000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`2ef60000 0`2ef70000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`2ef70000 0`2ef80000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`2ef80000 0`2ef90000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`2ef90000 0`2fa0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`2fa0000 0`2fb0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
```

```

* 0`2efb0000 0`2efc0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`2fc0000 0`2efd0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`2efd0000 0`2efe0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`2fe0000 0`2eff0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`2eff0000 0`2f000000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`2f000000 0`2f010000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`2f010000 0`2f170000 0`00160000 MEM_FREE PAGE_NOACCESS Free
[...]

0:000> !address
[...]
* 0`697f0000 0`69800000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`69800000 0`69810000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`69810000 0`69820000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`69820000 0`69830000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`69830000 0`69840000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`69840000 0`69850000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`69850000 0`69860000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`69860000 0`69870000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`69870000 0`69880000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`69880000 0`69890000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`69890000 0`698a0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`698a0000 0`699e0000 0`00140000 MEM_FREE PAGE_NOACCESS Free
[...]

0:000> !address
[...]
* 0`c08c0000 0`c08d0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`c08d0000 0`c08e0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`c08e0000 0`c08f0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`c08f0000 0`c0900000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`c0900000 0`c0910000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`c0910000 0`c0920000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`c0920000 0`c0930000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 0`c0930000 0`c0960000 0`00030000 MEM_FREE PAGE_NOACCESS Free
[...]

0:000> !address
[...]
* 1`3d6a0000 1`3d6b0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 1`3d6b0000 1`3d6c0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 1`3d6c0000 1`3d6d0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 1`3d6d0000 1`3d6e0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 1`3d6e0000 1`3d6f0000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 1`3d6f0000 1`3d700000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 1`3d700000 1`3d710000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 1`3d710000 1`3d720000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 1`3d720000 1`3d730000 0`00010000 MEM_PRIVATE MEM_RESERVE <unclassified>
* 1`3d730000 1`3d7a0000 0`00070000 MEM_FREE PAGE_NOACCESS Free
[...]

0:000> !address -summary

--- Usage Summary ----- RgnCount ----- Total Size ----- %ofBusy %ofTotal
Free 15 7fe`c275e000 ( 7.995 Tb) 99.94%
<unclassified> 80928 1`3d193000 ( 4.955 Gb) 99.86% 0.06%
Image 28 0`0034b000 ( 3.293 Mb) 0.06% 0.00%
Stack 6 0`00200000 ( 2.000 Mb) 0.04% 0.00%
MemoryMappedFile 8 0`001af000 ( 1.684 Mb) 0.03% 0.00%
TEB 2 0`00004000 ( 16.000 kb) 0.00% 0.00%
PEB 1 0`00001000 ( 4.000 kb) 0.00% 0.00%

--- Type Summary (for busy) ----- RgnCount ----- Total Size ----- %ofBusy %ofTotal
MEM_PRIVATE 80936 1`3d397000 ( 4.957 Gb) 99.90% 0.06%
MEM_IMAGE 29 0`0034c000 ( 3.297 Mb) 0.06% 0.00%
MEM_MAPPED 8 0`001af000 ( 1.684 Mb) 0.03% 0.00%

```

--- State Summary -----		RgnCount	Total Size -----	%ofBusy	%ofTotal
MEM_FREE		15	7fe`c275e000 ( 7.995 Tb)	99.94%	
MEM_RESERVE		80926	1`3d438000 ( 4.957 Gb)	99.91%	0.06%
MEM_COMMIT		47	0`0045a000 ( 4.352 Mb)	0.09%	0.00%
--- Protect Summary (for commit) - RgnCount -----			Total Size -----	%ofBusy	%ofTotal
PAGE_EXECUTE_READ		4	0`001ef000 ( 1.934 Mb)	0.04%	0.00%
PAGE_READONLY		19	0`001de000 ( 1.867 Mb)	0.04%	0.00%
PAGE_READWRITE		17	0`00080000 ( 512.000 kb)	0.01%	0.00%
PAGE_WRITECOPY		5	0`00008000 ( 32.000 kb)	0.00%	0.00%
PAGE_READWRITE PAGE_GUARD		2	0`00005000 ( 20.000 kb)	0.00%	0.00%
--- Largest Region by Usage -----		Base Address -----	Region Size -----		
Free		1`3fac7000	7fd`bdc79000 ( 7.991 Tb)		
<unclassified>		0`7f0e0000	0`00f00000 ( 15.000 Mb)		
Image		0`77831000	0`00102000 ( 1.008 Mb)		
Stack		0`00170000	0`000fb000 (1004.000 kb)		
MemoryMappedFile		0`7efe5000	0`000fb000 (1004.000 kb)		
TEB		7ff`ffffdc000	0`00002000 ( 8.000 kb)		
PEB		7ff`ffffd3000	0`00001000 ( 4.000 kb)		

Examination of such regions for **Execution Residue** (page 371) such as **Module Hint** (page 696) may point to further troubleshooting directions especially if live debugging is not possible.

## Message Box

Other suspicious threads in crash dumps are GUI threads executing message box code. Usually, message boxes are displayed to show some error, and we can see it by dumping the second and the third *MessageBox* parameters:

```
int MessageBox(
    HWND hWnd,
    LPCTSTR lpText,
    LPCTSTR lpCaption,
    UINT uType);
```

Sometimes message boxes block processes as shown in the example illustrating **Coupled Processes** pattern (page 149). Other threads might point to possibly “hang” sessions and processes in memory dumps coming from terminal services environments. This is another example of the collective **Blocked Thread** pattern (page 82).

Let's look at one example where message box pointed to the right troubleshooting direction. A user process was reported hanging from time to time. However, it was not specified which one. Searching for *MessageBox* in the log of all threads in the system produced by **!process 0 3f** WinDbg command revealed the following thread:

```
THREAD 88b14da8 Cid 0a04.0c14 Peb: 7fffd000 Win32Thread: e5e50ab0 WAIT: (WrUserRequest) UserMode Non-
Alertable
87b74358 SynchronizationEvent
IRP List:
87a0ba00: (0006,0244) Flags: 00000000 Mdl: 00000000
Not impersonating
DeviceMap          e14bec28
Owning Process     888ffb60      Image:           OUTLOOK.EXE
Wait Start TickCount 1275435      Ticks: 210 (0:00:00:03.281)
Context Switch Count 1050203      LargeStack
UserTime            00:00:16.812
KernelTime          00:00:18.000
Win32 Start Address OUTLOOK (0x30001084)
Start Address kernel32!BaseProcessStartThunk (0x7c810665)
Stack Init a0c98000 Current a0c97cb0 Base a0c98000 Limit a0c90000 Call 0
Priority 11 BasePriority 8 PriorityDecrement 2 DecrementCount 16
ChildEBP RetAddr
a0c97cc8 804e1bd2 nt!KiSwapContext+0x2f
a0c97cd4 804e1c1e nt!KiSwapThread+0x8a
a0c97cf8 bf802f70 nt!KeWaitForSingleObject+0x1c2
a0c97d38 bf803776 win32k!xxxSleepThread+0x192
a0c97d4c bf803793 win32k!xxxRealWaitMessageEx+0x12
a0c97d5c 804dd99f win32k!NtUserWaitMessage+0x14
a0c97d5c 7c90eb94 nt!KiFastCallEntry+0xfc
0013f3a8 7e419418 ntdll!KiFastSystemCallRet
0013f3e0 7e42593f USER32!NtUserWaitMessage+0xc
0013f408 7e43a91e USER32!InternalDialogBox+0xd0
0013f6c8 7e43a284 USER32!SoftModalMessageBox+0x938
0013f818 7e4661d3 USER32!MessageBoxWorker+0x2ba
0013f870 7e466278 USER32!MessageBoxTimeoutW+0x7a
```

```

0013f8a4 7e450617 USER32!MessageBoxTimeoutA+0x9c
0013f8c4 7e4505cf USER32!MessageBoxExA+0x1b
0013f8e0 088098a9 USER32!MessageBoxA+0x45
...
...
...
0: kd> .process /r /p 888ffb60
Implicit process is now 888ffb60
Loading User Symbols

0: kd> !thread 88b14da8
...
...
...
ChildEBP RetAddr  Args to Child
...
...
...
0013f8e0 088098a9 00000000 0013f944 088708f0 USER32!MessageBoxA+0x45
...
...
...
0: kd> dA 0013f944
0013f944 "Cannot contact database, Retry"

```

This immediately raised suspicion and looking at other threads in the same application revealed that many of them were trying to open a network connection, for example:

```

THREAD 87a70da8 Cid 0a04.0cc0 Teb: 7ff83000 Win32Thread: 00000000 WAIT: (UserRequest) UserMode Non-
Alertable
    87d690b0 NotificationEvent
    87a70e98 NotificationTimer
IRP List:
    87af7bc8: (0006,0244) Flags: 00000000 Mdl: 00000000
Not impersonating
DeviceMap          e14bec28
Owning Process     888ffb60      Image:        OUTLOOK.EXE
Wait Start TickCount 1267130      Ticks: 8515 (0:00:02:13.046)
Context Switch Count 18
UserTime           00:00:00.000
KernelTime         00:00:00.000
Win32 Start Address msmai32!FOpenThreadImpersonationToken (0x35f76963)
Start Address kernel32!BaseThreadStartThunk (0x7c810659)
Stack Init 9fbc5000 Current 9fbc4ca0 Base 9fbc5000 Limit 9fbc2000 Call 0
Priority 8 BasePriority 8 PriorityDecrement 0 DecrementCount 16
Kernel stack not resident.
ChildEBP RetAddr
9fbc4cb8 804e1bd2 nt!KiSwapContext+0x2f
9fbc4cc4 804e1c1e nt!KiSwapThread+0x8a
9fbc4cec 8056d2f9 nt!KeWaitForSingleObject+0x1c2
9fbc4d50 804dd99f nt!NtWaitForSingleObject+0x9a
9fbc4d50 7c90eb94 nt!KiFastCallEntry+0xfc
1bd8f52c 7c90e9c0 ntdll!KiFastSystemCallRet
1bd8f530 7c8025cb ntdll!ZwWaitForSingleObject+0xc

```

```

1bd8f594 7c802532 kernel32!WaitForSingleObjectEx+0xa8
1bd8f5a8 77eec4c6 kernel32!WaitForSingleObject+0x12
1bd8f6a4 77eec8b7 RPCRT4!WS_Open+0x31d
1bd8f7c8 77eec96d RPCRT4!TCPOrHTTP_Open+0x19e
1bd8f800 77e83e8d RPCRT4!TCP_Open+0x55
1bd8f84c 77e843f7 RPCRT4!OSF_CCONNECTION::TransOpen+0x5e
1bd8f8b4 77e84581 RPCRT4!OSF_CCONNECTION::OpenConnectionAndBind+0xbc
1bd8f8f8 77e844d0 RPCRT4!OSF_CCALL::BindToServer+0x104
1bd8f95c 77e7f99c RPCRT4!OSF_BINDING_HANDLE::AllocateCCall+0x2b0
1bd8f98c 77e791c1 RPCRT4!OSF_BINDING_HANDLE::NegotiateTransferSyntax+0x28
1bd8f9a4 77e791f8 RPCRT4!I_RpcGetBufferWithObject+0xb
1bd8f9b4 77e79825 RPCRT4!I_RpcGetBuffer+0xf
1bd8f9c4 77ef460b RPCRT4!NdrGetBuffer+0x28
1bd8fda4 35bae645 RPCRT4!NdrClientCall12+0x195
...
...
...

```

Looking at IRP showed the possible problem with the network at TDI level:

```

0: kd> !irp 87af7bc8
Irp is active with 4 stacks 2 is current (= 0x87af7c5c)  No Mdl: No System Buffer: Thread 87a70da8:  Irp
stack trace.
    cmd   flg cl Device      File      Completion-Context
[ 0, 0] 0 0 00000000 00000000 00000000-00000000
    Args: 00000000 00000000 00000000 00000000
>[ f, 3] 0 e1 897b0310 87bb24c8 a1ad7080-87b6f1f0 Success Error Cancel pending
    \Driver\Tcpip      MYTDI!WaitForNetwork
    Args: 00000000 87a33988 87a33af8 a1a57600
[ f, 3] 0 e1 88c13020 87bb24c8 a1a6bea5-87a33988 Success Error Cancel pending
    \Driver\SYMTDI afd!AfdRestartSuperConnect
    Args: 00000000 87a33988 87a33af8 a1a57600
[ e,31] 5 0 88bedf18 87d3af90 00000000-00000000
    \Driver\AFD
    Args: 00000000 00000016 000120c7 1bd8f52c

```

## Comments

Another similar example is a service thread that opens a dialog, but the service is not allowed to interact with the desktop. This effectively blocks the thread and potentially other threads if the blocked thread owns synchronization objects.

## Message Hooks

In addition to hooking functions via code patching, there is another function pre- and post-processing done via windows message hooking<sup>133</sup> mechanism that we call **Message Hooks** pattern to differentiate it from **Hooked Functions** pattern (page 488). In some cases, message hooking becomes a source of aberrant software behavior including spikes, hangs, and crashes. We can identify such residue looking at the problem thread raw stack:

```
0:000> !teb
TEB at 7ffdde000
ExceptionList:      0012fc0dc
StackBase:          00130000
StackLimit:         0011b000
SubSystemTib:       00000000
FiberData:          00001e00
ArbitraryUserPointer: 00000000
Self:               7ffdde000
EnvironmentPointer: 00000000
ClientId:           0000050c . 000004b8
RpcHandle:          00000000
Tls Storage:        00000000
PEB Address:        7ffdf000
LastErrorValue:     0
LastStatusValue:    c0000034
Count Owned Locks:  0
HardErrorMode:      0

0:000> dps 0011b000 00130000
[...]
0012fc78 7e4318d1 user32!DispatchHookA
0012fc7c 0012fc08
0012fc80 7472467f
0012fc84 7e43e1ad user32!NtUserCallNextHookEx+0xc
0012fc88 7e43e18a user32!CallNextHookEx+0x6f
0012fc8c 00000003
0012fc90 00000011
0012fc94 001d0001
0012fc98 00000001
0012fc9c 00000003
0012fc00 00000000
0012fc04 001d0001
0012fc08 0012fc0ec
0012fcac 74730844 DllA!ThreadKeyboardProc+0x77
0012fc00 001e04f7
0012fc04 00000003
0012fc08 00000011
0012fc0c 001d0001
0012fcc0 00000003
```

---

<sup>133</sup> [http://msdn.microsoft.com/en-us/library/ms632589\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms632589(VS.85).aspx)

```
0012fcc4 00020003
0012fcc8 001d0001
0012fccc 00000000
0012fcdo 001e04f7
0012fcda 0012fcc0
0012fcdb 00000000
0012fcdc 0012fd4c
0012fce0 7475f1a6
0012fce4 74730850
0012fce8 ffffffff
0012fce0 0012fd20
0012fcf0 7e431923 user32!DispatchHookA+0x101
0012fcf4 00000003
0012fcf8 00000011
0012fcfc 001d0001
0012fd00 00000000
0012fd04 0012fe94
0012fd08 00000102
0012fd0c 7ffd0000
0012fd10 00000000
0012fd14 00000001
0012fd18 00000003
0012fd1c 7e42b326 user32!CallHookWithSEH+0x44
0012fd20 0012fd5c
0012fd24 7e42b317 user32!CallHookWithSEH+0x21
0012fd28 00020003
0012fd2c 00000011
0012fd30 001d0001
0012fd34 747307c3
0012fd38 00000000
0012fd3c 0012fe94
0012fd40 00000102
[...]
```

```
0.000> ub 74730844
D11A!ThreadKeyboardProc+0x5e:
7473082b jne    D11A!ThreadKeyboardProc+0x77 (74730844)
7473082d cmp    dword ptr [ebp-1Ch],esi
74730830 je     D11A!ThreadKeyboardProc+0x77 (74730844)
74730832 push   dword ptr [ebp+10h]
74730835 push   dword ptr [ebp+0Ch]
74730838 push   dword ptr [ebp+8]
7473083b push   dword ptr [ebp-1Ch]
7473083e call   dword ptr [D11A!_imp__CallNextHookEx (74721248)]
```

Sometimes we can even reconstruct stack trace fragments<sup>134</sup> that show message hooking call stack. When threads are spiking or blocked in a message hook procedure we can see a hooking module too:

```
0:000> kL
ChildEBP RetAddr
0012fc80 7e43e1ad ntdll!KiFastSystemCallRet
0012fca8 74730844 user32!NtUserCallNextHookEx+0xc
0012fce0 7e431923 D11A!ThreadKeyboardProc+0x77
0012fd20 7e42b317 user32!DispatchHookA+0x101
0012fd5c 7e430238 user32!CallHookWithSEH+0x21
0012fd80 7c90e473 user32!__fnHkINDWORD+0x24
0012fda4 7e4193e9 ntdll!KiUserCallbackDispatcher+0x13
0012fdd0 7e419402 user32!NtUserPeekMessage+0xc
0012fdfc 747528ee user32!PeekMessageW+0xbc
[...]
0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

## Comments

---

Another example:

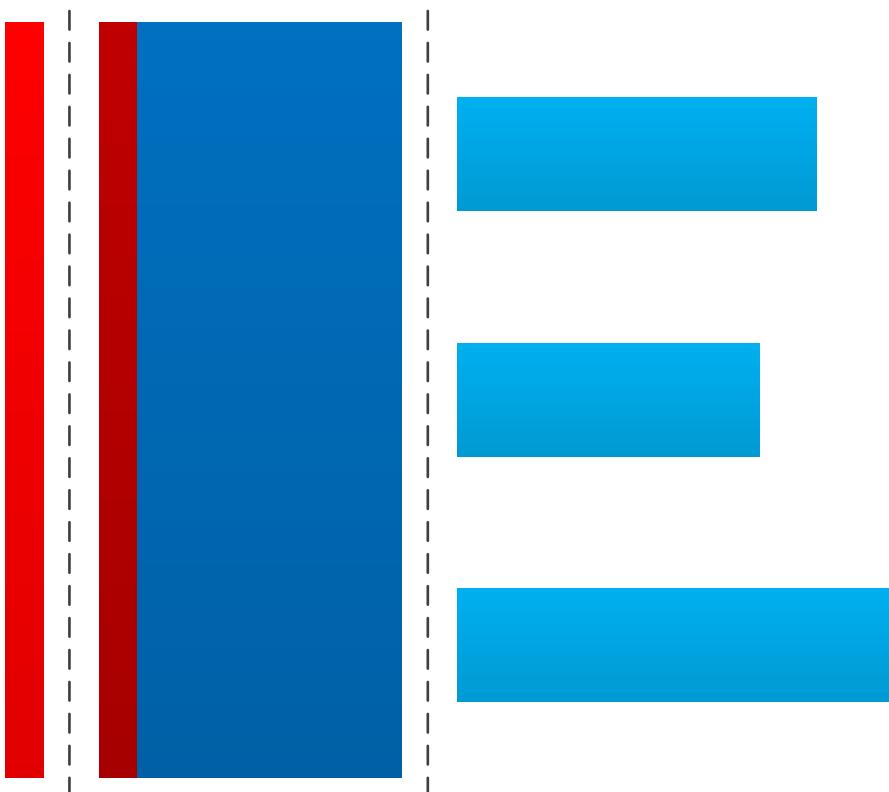
```
002afcd0 765614ab ntdll!ZwWaitForSingleObject+0x15
002afd3c 768b1194 KERNELBASE!WaitForSingleObjectEx+0x98
002afd54 768b1148 kernel32!WaitForSingleObjectExImplementation+0x75
002afd68 002e95de kernel32!WaitForSingleObject+0x12
WARNING: Stack unwind information not available. Following frames may be wrong.
002afd9c 767c8336 HookA+0x95de
002afdcc 767b80a9 USER32!DispatchHookA+0x10e
002afe0c 767c2bfd USER32!CallHookWithSEH+0x21
002afe3c 775b010a USER32!__fnHkINLPMOUSEHOOKSTRUCTEX+0x29
002afed4 00471c64 ntdll!KiUserCallbackDispatcher+0x2e
002afefc 00471d1f ProcessA+0x62d60
[...]
002aff94 775d9f72 kernel32!BaseThreadInitThunk+0xe
002affd4 775d9f45 ntdll!__RtlUserThreadStart+0x70
002affec 00000000 ntdll!_RtlUserThreadStart+0x1b
```

---

<sup>134</sup> Manual Stack Trace Reconstruction, Memory Dump Analysis Anthology, Volume 1, page 157

## Mirror Dump Set

If OS is not inside a virtual machine, it is difficult to get consistent live snapshots of physical memory (see **Inconsistent Dump** analysis pattern, page 498). Mirror dump options in LiveKd<sup>135</sup> can save a consistent kernel memory dump. Then we can either use Fiber Bundle<sup>136</sup> technique of saving individual process memory dumps or create inconsistent complete memory dump using LiveKd or both. We call this pattern **Mirror Dump Set**.



We can identify mirror dump with the following current stack trace:

```
0: kd> version
...
64-bit Kernel bitmap dump: ...
...
```

<sup>135</sup> <https://technet.microsoft.com/en-us/sysinternals/livekd>

<sup>136</sup> Memory Dump Analysis Anthology, Volume 4, page 357

```
0: kd> k
# Child-SP RetAddr Call Site
00 fffffd000`26121700 ffffff803`cf5f5ee3 nt!IopLiveDumpEndMirroringCallback+0x7f
01 fffffd000`26121750 ffffff803`cf60561b nt!MmDuplicateMemory+0x807
02 fffffd000`26121830 ffffff803`cf851c60 nt!IopLiveDumpCaptureMemoryPages+0x53
03 fffffd000`26121890 ffffff803`cf447443 nt!IoCaptureLiveDump+0xf8
04 fffffd000`261218e0 ffffff803`cf8ceb0d nt!DbgkCaptureLiveKernelDump+0x2e7
05 fffffd000`26121970 ffffff803`cf3debb3 nt!NtSystemDebugControl+0x3f5
06 fffffd000`26121a90 000007ffa`2925205a nt!KiSystemServiceCopyEnd+0x13
07 000000a3`5bcddb48 00000000`00000000 0x000007ffa`2925205a
```

In one analysis case, we got such a set where we analyzed ALPC **Wait Chains** (page 1097) with user space stack traces in a complete memory having the endpoint blocked in a filter driver (page 924). But the search for stack traces having filter manager in their frames failed due to inconsistency:

```
0: kd> version
...
64-bit Full kernel dump: ...
...

0: kd> !stacks 2 FltMgr
...
TYPE mismatch for thread object at fffffe001b804d638
4.----- NO ETHREAD DATA
TYPE mismatch for thread object at fffffe001b804d638
4.----- NO ETHREAD DATA
TYPE mismatch for thread object at fffffe001b804d638
4.----- NO ETHREAD DATA
TYPE mismatch for thread object at fffffe001b804d638
4.----- NO ETHREAD DATA
TYPE mismatch for thread object at fffffe001b804d638
4.----- NO ETHREAD DATA
...
...
```

So we found such kernel space stack traces from the consistent mirror dump.

## Missing Component

### General

Sometimes the code raises an exception when certain DLL is missing. We need to guess that component name if we don't have symbols and source code. This can be done by inspecting raw stack data in the close proximity of the exception ESP/RSP values.

Consider the crash dump of *App.exe* with the following incomplete unmanaged stack trace:

```
EXCEPTION_RECORD: ffffffff -- (.exr 0xffffffffffffffff)
ExceptionAddress: 76f442eb (kernel32!RaiseException+0x00000058)
  ExceptionCode: c06d007f
  ExceptionFlags: 00000000
NumberParameters: 1
  Parameter[0]: 0024f21c

0:000> kL
ChildEBP RetAddr
0024f1f8 6eb1081e kernel32!RaiseException+0x58
WARNING: Stack unwind information not available. Following frames may be wrong.
0024f260 6eac62fb AppNativeLib!AppLibraryExports::InteropNotifyUnAdvise+0x6aa9
0024f2ac 6ea9e269 AppNativeLib!AppLibraryExports::Phase2Initialization+0x24c9
0024f32c 79e74d79 AppNativeLib!AppLibraryExports::QueryDatabase+0x99da
0024f3d4 664bd6af mscorewks!MethodTable::IsValueType+0x35
0024f3e8 319cece AppShell_ni+0x2d6af
0024f3f4 31a15d19 UIX_ni+0x1ec9e
0024f3f8 00000000 UIX_ni+0x65d19
```

We can try to interpret the crash as **Managed Code Exception** (page 617) but let's first to check the exception code. Google search shows that the error code c06d007f means "DelayLoad Export Missing" and this definitely has to do with some missing DLL. It is not possible to tell which one was missing from the stack trace output. Additional digging is required.

Let's look at the raw stack. First, we can try to see whether there are any calls to *LoadLibrary* on thread raw stack data:

```
0:000> !teb
TEB at 7ffd000
  ExceptionList: 0024f8c4
  StackBase: 00250000
  StackLimit: 00249000
  SubSystemTib: 00000000
  FiberData: 00001e00
  ArbitraryUserPointer: 00000000
  Self: 7ffd000
  EnvironmentPointer: 00000000
  ClientId: 000012f4 . 00001080
  RpcHandle: 00000000
  Tls Storage: 004e8a18
  PEB Address: 7ffde000
```

```

LastErrorValue:      126
LastStatusValue:    c0000135
Count Owned Locks: 0
HardErrorMode:      0

0:000> dds 00249000 00250000
00249000 00000000
00249004 00000000
...
0024f1a0 00000000
0024f1a4 00000000
0024f1a8 c06d007f
0024f1ac 00000000
0024f1b0 00000000
0024f1b4 76f442eb kernel32!RaiseException+0x58
0024f1b8 00000001
0024f1bc 0024f21c
0024f1c0 00000000
0024f1c4 00000000
0024f1c8 00000000
0024f1cc 00000000
0024f1d0 76f00000 kernel32!_imp__aullrem (kernel32+0x0)
0024f1d4 f7bd2a5d
0024f1d8 0024f1e8
0024f1dc 76fb8e8f kernel32!LookupHandler+0x10
0024f1e0 6ecb9b40 AppNativeLib!ShutdownSingletonMgr+0x15b024
0024f1e4 0024f21c
0024f1e8 0024f200
0024f1ec 6ec74e2a AppNativeLib!ShutdownSingletonMgr+0x11630e
0024f1f0 6ecb9b40 AppNativeLib!ShutdownSingletonMgr+0x15b024
0024f1f4 6ecb9ff0 AppNativeLib!ShutdownSingletonMgr+0x15b4d4
0024f1f8 0024f260
0024f1fc 6eb1081e AppNativeLib!AppLibraryExports::InteropNotifyUnAdvise+0x6aa9
0024f200 c06d007f
0024f204 00000000
0024f208 00000001
...

```

There are no such calls in our crash dump. Then we can try to interpret raw stack data as a byte stream to see “.dll” strings:

```

0:000> db 00249000 00250000
00249000 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ..... .
00249010 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 ..... .
00249020 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 ..... .
00249030 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 ..... .
...

```

There are no such strings except “user32.dll”.

Now we can try to interpret every double word as a pointer to a Unicode string:

```
0:000> dpu 00249000 00250000
...
```

There are no strings with “.dll” inside. Finally, if we try to interpret every double word as a pointer to an ASCII string we get a few references to “AppService.dll”:

```
0:000> dpa 00249000 00250000
...
0024f1d0 76f00000 "MZ."
0024f1d4 f7bd2a5d
0024f1d8 0024f1e8 ""
0024f1dc 76fb8e8f "..t-.E."
0024f1e0 6ecb9b40 "AppService.dll"
0024f1e4 0024f21c "$"
0024f1e8 0024f200 "."
0024f1ec 6ec74e2a "...^.._]"
0024f1f0 6ecb9b40 "AppService.dll"
0024f1f4 6ecb9ff0 "CreateServiceInstance"
0024f1f8 0024f260 "..$"
0024f1fc 6eb1081e ".].....e."
0024f200 c06d007f
0024f204 00000000
0024f208 00000001
0024f20c 0024f268 "..$"
0024f210 00000000
0024f214 0024f2c8 "...n ..n<.$"
0024f218 6ecbe220 ""
0024f21c 00000024
0024f220 6ecb9960 "."
0024f224 6ecbe05c ".c.n.2.n"
0024f228 6ecb9b40 "AppService.dll"
0024f22c 00000001
0024f230 6ecb9ff0 "CreateServiceInstance"
0024f234 ffffffff
0024f238 00000000
```

If we search for 0024f1e0 pointer in **dps** WinDbg command output we would see that it is in the close proximity to *RaiseException* call and it seems that all our pointers to “AppService.dll” string fall into *AppNativeLib* address range:

```
0024f1b4 76f442eb kernel32!RaiseException+0x58
0024f1b8 00000001
0024f1bc 0024f21c
0024f1c0 00000000
0024f1c4 00000000
0024f1c8 00000000
0024f1cc 00000000
0024f1d0 76f00000 kernel32!_imp___aullrem (kernel32+0x0)
0024f1d4 f7bd2a5d
0024f1d8 0024f1e8
0024f1dc 76fb8e8f kernel32!LookupHandler+0x10
0024f1e0 6ecb9b40 AppNativeLib!ShutdownSingletonMgr+0x15b024
0024f1e4 0024f21c
```

```
0024f1e8 0024f200
0024f1ec 6ec74e2a AppNativeLib!ShutdownSingletonMgr+0x11630e
0024f1f0 6ecb9b40 AppNativeLib!ShutdownSingletonMgr+0x15b024
0024f1f4 6ecb9ff0 AppNativeLib!ShutdownSingletonMgr+0x15b4d4
0024f1f8 0024f260
0024f1fc 6eb1081e AppNativeLib!AppLibraryExports::InteropNotifyUnAdvise+0x6aa9
0024f200 c06d007f
0024f204 00000000
0024f208 00000001
0024f20c 0024f268
0024f210 00000000
0024f214 0024f2c8
0024f218 6ecbe220 AppNativeLib!ShutdownSingletonMgr+0x15f704
0024f21c 00000024
0024f220 6ecb9960 AppNativeLib!ShutdownSingletonMgr+0x15ae44
0024f224 6ecbe05c AppNativeLib!ShutdownSingletonMgr+0x15f540
0024f228 6ecb9b40 AppNativeLib!ShutdownSingletonMgr+0x15b024
0024f22c 00000001
0024f230 6ecb9ff0 AppNativeLib!ShutdownSingletonMgr+0x15b4d4
0024f234 ffffffff
0024f238 00000000
```

When examining the system, it was found that *AppService.dll* was missing there indeed.

## Static Linkage

### User Mode

In the general description of **Missing Component** pattern (page 668) the example and emphasis were on dynamically loaded modules. Here we cover statically linked modules. Failure for a loader to find one of them results in **Software Exception** (page 875). The most frequent of them are (numbers were taken from Google search):

```
C0000142 918
C0000143 919
C0000145 1,530
C0000135 24,900

0:001> !error c0000142
Error code: (NTSTATUS) 0xc0000142 (3221225794) - {DLL Initialization Failed} Initialization of the
dynamic link library %hs failed. The process is terminating abnormally.

0:001> !error c0000143
Error code: (NTSTATUS) 0xc0000143 (3221225795) - {Missing System File} The required system file %hs is
bad or missing.

0:001> !error c0000145
Error code: (NTSTATUS) 0xc0000145 (3221225797) - {Application Error} The application failed to
initialize properly (0x%lx). Click on OK to terminate the application.

0:000> !error c0000135
Error code: (NTSTATUS) 0xc0000135 (3221225781) - {Unable To Locate Component} This application has
failed to start because %hs was not found. Re-installing the application may fix this problem.
```

We only consider user mode exceptions. If we have a default debugger configured it usually saves a crash dump. To model this problem one of the applications was modified by changing all occurrences of *KERNEL32.DLL* to *KERNEL32.DL* using Visual Studio Binary Editor. CDB was configured as a default postmortem debugger<sup>137</sup>. When the application was launched CDB attached to it and saved a crash dump. If we open it in WinDbg we get characteristic **Special Stack Trace** (page 882) involving loader functions:

```
Loading Dump File [C:\UserDumps\CDAPatternMissingComponent.dmp]
User Mini Dump File with Full Memory: Only application data is available

Symbol search path is: srv*c:\mss*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows Vista Version 6000 MP (2 procs) Free x86 compatible
Product: WinNt, suite: SingleUserTS
Debug session time: Thu Jun 12 12:03:28.000 2008 (GMT+1)
System Uptime: 1 days 8:46:23.167
Process Uptime: 0 days 0:00:48.000
```

<sup>137</sup> Custom Postmortem Debuggers in Vista, Memory Dump Analysis Anthology, Volume 1, page 618

```
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
(da4.f60): Wake debugger - code 80000007 (first/second chance not available)
eax=00000000 ebx=77c4a174 ecx=75ce3cf9 edx=00000000 esi=7efde028 edi=7efdd000
eip=77bcf1d1 esp=0017fcfa4 ebp=0017fd00 iopl=0 nv up ei pl zr na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000246
ntdll!_LdrpInitialize+0x6d:
77bcf1d1 8b45b8 mov eax,dword ptr [ebp-48h] ss:002b:0017fcbb=7efde000

0:000> kL
ChildEBP RetAddr
0017fd00 77b937ea ntdll!_LdrpInitialize+0x6d
0017fd10 00000000 ntdll!LdrInitializeThunk+0x10
```

Verbose analysis command doesn't give us an indication of what had happened, so we need to dig further:

```
0:000> !analyze -v
...
FAULTING_IP:
+0
00000000 ??          ???

EXCEPTION_RECORD: ffffffff -- (.exr 0xffffffffffffffff)
ExceptionAddress: 00000000
    ExceptionCode: 80000007 (Wake debugger)
    ExceptionFlags: 00000000
NumberParameters: 0

BUGCHECK_STR: 80000007

PROCESS_NAME: StackOverflow.exe

ERROR_CODE: (NTSTATUS) 0x80000007 - {Kernel Debugger Awakened} the system debugger was awakened by an
interrupt.

NTGLOBALFLAG: 400

APPLICATION_VERIFIER_FLAGS: 0

DERIVED_WAIT_CHAIN:

Dl Eid Cid      WaitType
--- -----
 0   da4.f60 Unknown

WAIT_CHAIN_COMMAND: ~0s;k;

BLOCKING_THREAD: 00000f60

DEFAULT_BUCKET_ID: APPLICATION_HANG_BusyHang

PRIMARY_PROBLEM_CLASS: APPLICATION_HANG_BusyHang

LAST_CONTROL_TRANSFER: from 77b937ea to 77bcf1d1

FAULTING_THREAD: 00000000
```

```

STACK_TEXT:
0017fd00 77b937ea 0017fd24 77b60000 00000000 ntdll!_LdrpInitialize+0x6d
0017fd10 00000000 0017fd24 77b60000 00000000 ntdll!LdrInitializeThunk+0x10

FOLLOWUP_IP:
ntdll!_LdrpInitialize+6d
77bcf1d1 8b45b8      mov     eax,dword ptr [ebp-48h]

SYMBOL_STACK_INDEX:  0

SYMBOL_NAME:  ntdll!_LdrpInitialize+6d

FOLLOWUP_NAME:  MachineOwner

MODULE_NAME: ntdll

IMAGE_NAME:  ntdll.dll

DEBUG_FLR_IMAGE_TIMESTAMP:  4549bdf8

STACK_COMMAND: ~0s ; kb

BUCKET_ID:  80000007_ntdll!_LdrpInitialize+6d

FAILURE_BUCKET_ID:  ntdll.dll!_LdrpInitialize_80000007_APPLICATION_HANG_BusyHang

Followup: MachineOwner

```

Last event and error code are not helpful too:

```

0:000> .lastevent
Last event: da4.f60: Wake debugger - code 80000007 (first/second chance not available)
debugger time: Thu Jun 12 15:04:38.917 2008 (GMT+1)

0:000> !gle
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0

```

Now we search thread raw stack data for any signs of exceptions:

```

0:000> !teb
TEB at 7efdd000
ExceptionList:      0017fcf0
StackBase:          00180000
StackLimit:         0017e000
SubSystemTib:       00000000
FiberData:          00001e00
ArbitraryUserPointer: 00000000
Self:               7efdd000
EnvironmentPointer: 00000000
ClientId:           00000da4 . 00000f60
RpcHandle:          00000000
Tls Storage:        00000000
PEB Address:        7efde000
LastErrorValue:     0

```

```
LastStatusValue:      0
Count Owned Locks:   0
HardErrorMode:       0

0:000> dds 0017e000 00180000
...
0017f8d8  7efdd000
0017f8dc  0017f964
0017f8e0  77c11c78 ntdll!_except_handler4
0017f8e4  00000000
0017f8e8  0017f988
0017f8ec  0017f900
0017f8f0  77ba1ddd ntdll!RtlCallVectoredContinueHandlers+0x15
0017f8f4  0017f988
0017f8f8  0017f9d8
0017f8fc  77c40370 ntdll!RtlpCallbackEntryList
0017f900  0017f970
0017f904  77ba1db5 ntdll!RtlDispatchException+0x11f
0017f908  0017f988
0017f90c  0017f9d8
0017f910  7efde028
0017f914  00000001
0017f918  77630000 kernel32!_imp__aullrem <PERF> (kernel32+0x0)
0017f91c  00000001
0017f920  776ced81 kernel32!_DllMainCRTStartupForGS2+0x10
0017f924  0017f938
0017f928  7765d4d9 kernel32!BaseDllInitialize+0x18
0017f92c  76042340 user32!$$VProc_ImageExportDirectory
0017f930  00000001
0017f934  00000000
0017f938  0017f9e0
0017f93c  77b8f890 ntdll!LdrpSnapThunk+0xc9
0017f940  0040977a StackOverflow+0x977a
0017f944  0000030b
0017f948  76030000 user32!_imp__RegSetValueExW <PERF> (user32+0x0)
0017f94c  76042f94 user32!$$VProc_ImageExportDirectory+0xc54
0017f950  77bb8881 ntdll!LdrpSnapThunk+0x40d
0017f954  0017bb30
0017f958  00409770 StackOverflow+0x9770
0017f95c  00881a50
0017f960  004098b2 StackOverflow+0x98b2
0017f964  77bac282 ntdll!ZwRaiseException+0x12
0017f968  00180000
0017f96c  0017fc48
0017f970  0017fd00
0017f974  77bac282 ntdll!ZwRaiseException+0x12
0017f978  77b7ee72 ntdll!KiUserExceptionDispatcher+0x2a
0017f97c  0017f988 ; exception record
0017f980  0017f9d8 ; exception context
0017f984  00000000
0017f988  c0000135
0017f98c  00000001
0017f990  00000000
0017f994  77bcf1d1 ntdll!_LdrpInitialize+0x6d
0017f998  00000000
0017f99c  77c11c78 ntdll!_except_handler4
```

```

0017f9a0 77b8dab8 ntdll!RtlpRunTable+0x218
0017f9a4 fffffffe
0017f9a8 77ba2515 ntdll!vDbgPrintExWithPrefixInternal+0x214
0017f9ac 77ba253b ntdll!DbgPrintEx+0x1e
0017f9b0 77b7f356 ntdll! ?? ::FNODOBFM::`string'
0017f9b4 00000055
0017f9b8 00000003
0017f9bc 77b809c2 ntdll! ?? ::FNODOBFM::`string'
0017f9c0 0017fc9c
0017f9c4 00000001
0017f9c8 0017fd00
0017f9cc 77bcf28e ntdll!_LdrpInitialize+0x12a
0017f9d0 00000055
0017f9d4 75ce3cf9
0017f9d8 0001003f
0017f9dc 00000000
0017f9e0 00000000
0017f9e4 00000000
0017f9e8 00000000
0017f9ec 00000000
...

```

We see exception dispatching calls highlighted above (**Hidden Exception**, page 457). One of their parameters is an exception record, and we try to get one:

```

0:000> .exr 0017f988
ExceptionAddress: 77bcf1d1 (ntdll!_LdrpInitialize+0x0000006d)
  ExceptionCode: c0000135
  ExceptionFlags: 00000001
NumberParameters: 0

```

Error c0000135 means that the loader was unable to locate a component. Now we try to examine the same raw stack data for any string patterns. For example, the following UNICODE pattern is clearly visible:

```

0017f2fc 00000000
0017f300 00880ec4
0017f304 77b910d7 ntdll!RtlpDosPathNameToRelativeNtPathName_Ustr+0x344
0017f308 00000000
0017f30c 43000043
0017f310 0042002a
0017f314 0017f33c
0017f318 00000000
0017f31c 00000002
0017f320 00000008
0017f324 00000000
0017f328 0000008c
0017f32c 000a0008
0017f330 77b91670 ntdll!`string'
0017f334 00b92bd6
0017f338 0017f5d4
0017f33c 003a0043
0017f340 0050005c
0017f344 006f0072
0017f348 00720067
0017f34c 006d0061

```

0017f350	00460020
0017f354	006c0069
0017f358	00730065
0017f35c	00280020
0017f360	00380078
0017f364	00290036
0017f368	0043005c
0017f36c	006d006f
0017f370	006f006d
0017f374	0020006e
0017f378	00690046
0017f37c	0065006c
0017f380	005c0073
0017f384	006f0052
0017f388	00690078
0017f38c	0020006f
0017f390	00680053
0017f394	00720061
0017f398	00640065
0017f39c	0044005c
0017f3a0	004c004c
0017f3a4	00680053
0017f3a8	00720061
0017f3ac	00640065
0017f3b0	004b005c
0017f3b4	00520045
0017f3b8	0045004e
0017f3bc	0033004c
0017f3c0	002e0032
0017f3c4	006c0064
0017f3c8	00000000
0017f3cc	00000000

It is a path to a DLL that was probably missing:

```
0:000> du 0017f33c
0017f33c  "C:\Program Files (x86)\Common Fi"
0017f37c  "les\Roxio Shared\DLLShared\KERNE"
0017f3bc  "L32.dll"
```

We think the loader was trying to find *KERNEL32.dll* following the DLL search order, and this was the last path element:

```
0:000> !peb
PEB at 7efde000
InheritedAddressSpace: No
ReadImageFileExecOptions: No
BeingDebugged: Yes
ImageBaseAddress: 00400000
Ldr 77c40080
Ldr.Initialized: Yes
Ldr.InInitializationOrderModuleList: 00881ad0 . 008831b8
Ldr.InLoadOrderModuleList: 00881a50 . 00883dc8
Ldr.InMemoryOrderModuleList: 00881a58 . 00883dd0
    Base TimeStamp           Module
```

```
...
    Environment: 00881de8
...
Path=C:\Windows\system32; C:\Windows; C:\Windows\System32\Wbem; C:\Program Files\ATI
Technologies\ATI.ACE; c:\Program Files (x86)\Microsoft SQL Server\90\Tools\binn\; C:\Program Files
(x86)\Common Files\Roxio Shared\DLLShared\
...
```

In similar situations **!dlls** command might help. It shows the load order (-l switch) and points to the last processed DLL:

```
0:001> !dlls -l

0x004740e8: C:\Program Files\Application\Application.exe
  Base 0x012a0000 EntryPoint 0x012b0903 Size      0x00057000
  Flags 0x00004010 LoadCount 0x0000ffff TlsIndex 0x00000000
        LDRP_ENTRY_PROCESSED

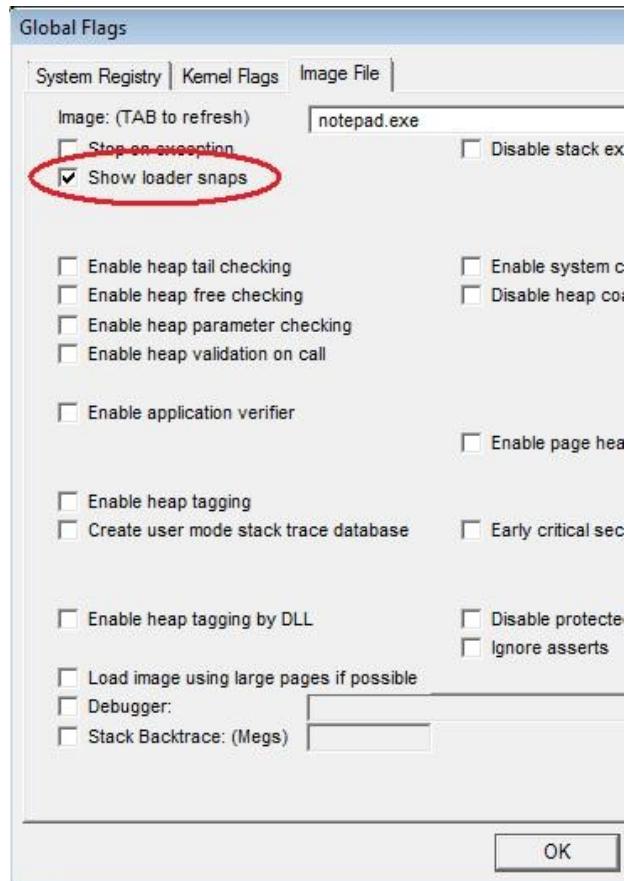
0x00474158: C:\Windows\SysWOW64\ntdll.dll
  Base 0x77d00000 EntryPoint 0x00000000 Size      0x00160000
  Flags 0x00004014 LoadCount 0x0000ffff TlsIndex 0x00000000
        LDRP_IMAGE_DLL
        LDRP_ENTRY_PROCESSED

0x00474440: C:\Windows\syswow64\kernel32.dll
  Base 0x77590000 EntryPoint 0x775a1f3e Size      0x00110000
  Flags 0x00084014 LoadCount 0x0000ffff TlsIndex 0x00000000
        LDRP_IMAGE_DLL
        LDRP_ENTRY_PROCESSED
        LDRP_PROCESS_ATTACH_CALLED

...
0x00498ff8: C:\Windows\WinSxS\x86_microsoft.windows.common-controls_...\\comctl32.dll
  Base 0x74d90000 EntryPoint 0x74dc43e5 Size      0x0019e000
  Flags 0x100c4014 LoadCount 0x00000003 TlsIndex 0x00000000
        LDRP_IMAGE_DLL
        LDRP_ENTRY_PROCESSED
        LDRP_DONT_CALL_FOR_THREADS
        LDRP_PROCESS_ATTACH_CALLED

0x004991b8: C:\Windows\WinSxS\x86_microsoft.vc80.mfcloc_...\\MFC80ENU.DLL
  Base 0x71b10000 EntryPoint 0x00000000 Size      0x0000e000
  Flags 0x10004014 LoadCount 0x00000001 TlsIndex 0x00000000
        LDRP_IMAGE_DLL
        LDRP_ENTRY_PROCESSED
```

If it is difficult to identify what had really happened in crash dumps, we can enable loader snaps using `gflags` and run the application under a debugger. For example, for `notepad.exe` we have:



```
Microsoft (R) Windows Debugger Version 6.8.0004.0 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.
```

```
CommandLine: C:\Windows\notepad.exe
Symbol search path is: srv*c:\msd\http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00000000`ffac0000 00000000`ffaef000  notepad.exe
ModLoad: 00000000`779b0000 00000000`77b2a000  ntdll.dll
LDR: NEW PROCESS
  Image Path: C:\Windows\notepad.exe (notepad.exe)
  Current Directory: C:\Program Files\Debugging Tools for Windows 64-bit\
  Search Path: C:\Windows; C:\Windows\system32; C:\Windows\system; C:\Windows\.; C:\Program
Files\Debugging Tools for Windows 64-bit\winext\arcade; C:\Windows\system32;C:\Windows;
C:\Windows\System32\Wbem; C:\Program Files\ATI Technologies\ATI.ACE; c:\Program Files (x86)\Microsoft SQL
Server\90\Tools\binn\; C:\Program Files (x86)\Common Files\Roxio Shared\DLLShared\
LDR: LdrLoadDll, loading kernel32.dll from
ModLoad: 00000000`777a0000 00000000`778d1000  C:\Windows\system32\kernel32.dll
LDR: kernel32.dll bound to ntdll.dll
LDR: kernel32.dll has stale binding to ntdll.dll
LDR: Stale Bind ntdll.dll from kernel32.dll
```

```
LDR: LdrGetProcedureAddress by NAME - BaseThreadInitThunk
[3d8,1278] LDR: Real INIT LIST for process C:\Windows\notepad.exe pid 984 0x3d8
[3d8,1278]   C:\Windows\system32\kernel32.dll init routine 00000000777DC960
[3d8,1278] LDR: kernel32.dll loaded - Calling init routine at 00000000777DC960
LDR: notepad.exe bound to ADVAPI32.dll
ModLoad: 000007fe`fe520000 000007fe`fe61f000  C:\Windows\system32\ADVAPI32.dll
LDR: ADVAPI32.dll bound to ntdll.dll
LDR: ADVAPI32.dll has stale binding to ntdll.dll
LDR: Stale Bind ntdll.dll from ADVAPI32.dll
LDR: ADVAPI32.dll bound to KERNEL32.dll
LDR: ADVAPI32.dll has stale binding to KERNEL32.dll
LDR: ADVAPI32.dll bound to ntdll.dll via forwarder(s) from kernel32.dll
LDR: ADVAPI32.dll has stale binding to ntdll.dll
LDR: Stale Bind KERNEL32.dll from ADVAPI32.dll
LDR: LdrGetProcedureAddress by NAME - RtlAllocateHeap
LDR: LdrGetProcedureAddress by NAME - RtlReAllocateHeap
LDR: LdrGetProcedureAddress by NAME - RtlEncodePointer
LDR: LdrGetProcedureAddress by NAME - RtlDecodePointer
LDR: LdrGetProcedureAddress by NAME - RtlSizeHeap
LDR: LdrGetProcedureAddress by NAME - RtlDeleteCriticalSection
LDR: LdrGetProcedureAddress by NAME - RtlEnterCriticalSection
LDR: LdrGetProcedureAddress by NAME - RtlLeaveCriticalSection
LDR: ADVAPI32.dll bound to RPCRT4.dll
...
...
```

This technique only works for native platform loader snaps. For example, it doesn't show loader snaps for 32-bit modules loaded under WOW64:

```
Microsoft (R) Windows Debugger Version 6.8.0004.0 X86
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: C:\Apps\StackOverflow.exe
Symbol search path is: srv*c:\mss*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00400000 00418000  StackOverflow.exe
ModLoad: 77b60000 77cb0000  ntdll.dll
LDR: NEW PROCESS
  Image Path: C:\Apps\StackOverflow.exe (StackOverflow.exe)
...
LDR: Loading (STATIC, NON_REDIRECTED) C:\Windows\system32\wow64cpu.dll
LDR: wow64cpu.dll bound to ntdll.dll
LDR: wow64cpu.dll has stale binding to ntdll.dll
LDR: Stale Bind ntdll.dll from wow64cpu.dll
LDR: wow64cpu.dll bound to wow64.dll
LDR: wow64cpu.dll has stale binding to wow64.dll
LDR: Stale Bind wow64.dll from wow64cpu.dll
LDR: wow64.dll has stale binding to wow64cpu.dll
LDR: Stale Bind wow64cpu.dll from wow64.dll
LDR: Refcount wow64cpu.dll (1)
LDR: Refcount wow64.dll (2)
LDR: Refcount wow64win.dll (1)
LDR: Refcount wow64.dll (3)
LDR: LdrGetProcedureAddress by NAME - Wow64LdrpInitialize
...
ModLoad: 77630000 77740000  C:\Windows\syswow64\kernel32.dll
ModLoad: 76030000 76100000  C:\Windows\syswow64\USER32.dll
```

```

ModLoad: 775a0000 77630000 C:\Windows\syswow64\GDI32.dll
ModLoad: 76d00000 76dbf000 C:\Windows\syswow64\ADVAPI32.dll
ModLoad: 76df0000 76ee0000 C:\Windows\syswow64\RPCRT4.dll
ModLoad: 75d60000 75dc0000 C:\Windows\syswow64\Secur32.dll
(1ec.1290): Unknown exception - code c0000135 (first chance)
(1ec.1290): Unknown exception - code c0000135 (!!! second chance !!!)
eax=00000000 ebx=77c4a174 ecx=75ce3cf9 edx=00000000 esi=7efde028 edi=7efdd000
eip=77bcf1d1 esp=0017fca4 ebp=0017fd00 iopl=0 nv up ei pl zr na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000246
ntdll!_LdrpInitialize+0x6d:
77bcf1d1 8b45b8 mov eax,dword ptr [ebp-48h] ss:002b:0017fcb8=7efde000

```

The dump file that we used can be downloaded from FTP to play with<sup>138</sup>.

## Comments

Sometimes, we may not see exceptions, but error and status codes in the output of **!teb** command, for example:

```

Error code: (Win32) 0x7e (126) - The specified module could not be found.
Error code: (NTSTATUS) 0xc0000135 (3221225781) - The program can't start because %hs is missing from your
computer. Try reinstalling the program to fix this problem.

```

---

<sup>138</sup> <ftp://dumpanalysis.org/pub/CDAPatternMissingComponent.zip>

## Missing Process

Sometimes the functionality of a system depends on a specific application or service process. For example, in a database server environment it might be a database process, in a printing environment, it is a print spooler process, or in a terminal services environment, it is a terminal services process (`termsvc`, hosted by `svchost.exe`). In system failure scenarios, we should check these processes for their presence (and also the presence of any **Coupled Processes**, page 149). However, if the vital process is present in the output we should check if it is exited but references to it exist, or whether there are any **Missing Threads** (page 683) or **Missing Components** (page 666) inside it, any **Suspended Threads** (page 964) and **Special Processes** (page 875) like a postmortem debugger. We shouldn't also forget about service dependencies and their relevant process startup order. For example, we know that our service is hosted by `svchost.exe` and we see one such process exited, but its object is still referenced somewhere:

```
0: kd> !vm

*** Virtual Memory Usage ***
[...]
  0ed8 svchost.exe      0 (        0 Kb)
[...]
```

However, another command shows that it could be a different service hosted by the same image, `svchost.exe` if we know that ServiceA depends on our service:

```
0: kd> !process 0 0

[...]

PROCESS 8aabed88 SessionId: 0 Cid: 0854 Peb: 7ffd6000 ParentCid: 0220
  DirBase: bff4d4a0 ObjectTable: e1c867a8 HandleCount: 778.
  Image: ServiceA.exe

[...]

PROCESS 8aaa6510 SessionId: 0 Cid: 0ed8 Peb: 7ffd4000 ParentCid: 0220
  DirBase: bff4d580 ObjectTable: 00000000 HandleCount: 0.
  Image: svchost.exe

[...]
```

Another alternative is that our service was restarted but then exited. If our process is not visible, it could be possible that it was either stopped or simply crashed before.

## Comments

Another example is a GUI message **Wait Chain** (page 1132) directed to a window that once belonged to no longer running process.

## Missing Thread

Sometimes it is possible that a process crash dump doesn't have all usual threads inside. For example, we expect at least 4 threads including the main process thread, but in the dump, we see only 3. If we know that some access violations were reported in the event log before (not necessarily for the same PID), we may suspect that one of the threads had been terminated due to some reason. We call this pattern **Missing Thread**.

In order to simulate this problem, we created a small multithreaded program in Visual C++:

```
#include "stdafx.h"
#include <process.h>

void thread_request(void *param)
{
    while (true);
}

int _tmain(int argc, _TCHAR* argv[])
{
    _beginthread(thread_request, 0, NULL);

    try
    {
        if (argc == 2)
        {
            *(int *)NULL = 0;
        }
    }
    catch (...)
    {
        _endthread();
    }

    while (true);

    return 0;
}
```

If there is a command line argument, then the main thread simulates access violation and finishes in the exception handler. In order to use SEH exceptions with C++ try/catch blocks we have to enable /EHa option in C++ Code Generation properties:

Enable String Pooling	No
Enable Minimal Rebuild	No
Enable C++ Exceptions	<b>Yes With SEH Exceptions (/EHs)</b>
Smaller Type Check	No
Basic Runtime Checks	Default
Runtime Library	<b>Multi-threaded (/MT)</b>
Struct Member Alignment	Default
Buffer Security Check	Yes
Enable Function-Level Linking	No
Enable Enhanced Instruction Set	Not Set
Floating Point Model	Precise (/fp:precise)
Enable Floating Point Exceptions	No

If we run the program without command line parameter and take a manual dump from it, we see 2 threads:

```
0:000> ~*kL

. 0  Id: 1208.fdc Suspend: 1 Teb: 7efdd000 Unfrozen
ChildEBP RetAddr
0012ff70 00401403 MissingThread!wmain+0x58
0012ffc0 7d4e7d2a MissingThread!__tmainCRTStartup+0x15e
0012fff0 00000000 kernel32!BaseProcessStart+0x28

1  Id: 1208.102c Suspend: 1 Teb: 7efda000 Unfrozen
ChildEBP RetAddr
005dff7c 004010ef MissingThread!thread_request
005dffb4 00401188 MissingThread!_callthreadstart+0x1b
005dffb8 7d4dfe21 MissingThread!_threadstart+0x73
005dffec 00000000 kernel32!BaseThreadStart+0x34

0:000> ~
. 0  Id: 1208.fdc Suspend: 1 Teb: 7efdd000 Unfrozen
1  Id: 1208.102c Suspend: 1 Teb: 7efda000 Unfrozen

0:000> dd 7efdd000 14
7efdd000 0012ff64 00130000 0012e000 00000000
```

We also dumped TEB of the main thread. However, if we run the program with any command line parameter and look at its manual dump we would see only one thread with the main thread missing:

```
0:000> ~*kL

. 0  Id: 1004.12e8 Suspend: 1 Teb: 7efda000 Unfrozen
ChildEBP RetAddr
005dff7c 004010ef MissingThread!thread_request
005dffb4 00401188 MissingThread!_callthreadstart+0x1b
005dffb8 7d4dfe21 MissingThread!_threadstart+0x73
005dffec 00000000 kernel32!BaseThreadStart+0x34

0:000> ~
. 0  Id: 1004.12e8 Suspend: 1 Teb: 7efda000 Unfrozen
```

If we try to dump TEB address and stack data from the missing main thread we would see that the memory was already decommitted:

```
0:000> dd 7efdd000 14
7efdd000  ??????? ??????? ??????? ????????

0:000> dds 0012e000  00130000
0012e000  ????????
0012e004  ????????
0012e008  ????????
0012e00c  ????????
0012e010  ????????
0012e014  ????????
0012e018  ????????
0012e01c  ????????
0012e020  ????????
0012e024  ????????
```

The same effect can be achieved in the similar program that exits the thread in the custom unhandled exception filter:

```
#include "stdafx.h"
#include <process.h>
#include <windows.h>

LONG WINAPI CustomUnhandledExceptionFilter(struct _EXCEPTION_POINTERS* ExceptionInfo)
{
    ExitThread(-1);
}

void thread_request(void *param)
{
    while (true);
}

int _tmain(int argc, _TCHAR* argv[])
{
    _beginthread(thread_request, 0, NULL);
    SetUnhandledExceptionFilter(CustomUnhandledExceptionFilter);

    *(int *)NULL = 0;

    while (true);

    return 0;
}
```

The solution to catch an exception that results in a thread termination would be to run the program under WinDbg or any other debugger:

```
CommandLine: C:\MissingThread\MissingThread.exe 1
Symbol search path is: SRV*c:\websymbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00400000 0040f000 MissingThread.exe
ModLoad: 7d4c0000 7d5f0000 NOT_AN_IMAGE
ModLoad: 7d600000 7d6f0000 C:\W2K3\SysWow64\ntdll32.dll
ModLoad: 7d4c0000 7d5f0000 C:\W2K3\syswow64\kernel32.dll
(df0.12f0): Break instruction exception - code 80000003 (first chance)
eax=7d600000 ebx=7efde000 ecx=00000005 edx=00000020 esi=7d6a01f4 edi=00221f38
eip=7d61002d esp=0012fb4c ebp=0012fcac iopl=0 nv up ei pl nz na po nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000020
ntdll32!DbgBreakPoint:
7d61002d cc int 3

0:000> g
ModLoad: 71c20000 71c32000 C:\W2K3\SysWow64\tsappcmp.dll
ModLoad: 77ba0000 77bfa000 C:\W2K3\syswow64\msvcrt.dll
ModLoad: 00410000 004ab000 C:\W2K3\syswow64\ADVAPI32.dll
ModLoad: 7da20000 7db00000 C:\W2K3\syswow64\RPCRT4.dll
ModLoad: 7d8d0000 7d920000 C:\W2K3\syswow64\Secur32.dll
(df0.12f0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=000007a0 ebx=7d4d8df9 ecx=78b842d9 edx=00000000 esi=00000002 edi=00000ece
eip=00401057 esp=0012ff50 ebp=0012ff70 iopl=0 nv up ei pl zr na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010246
MissingThread!wmain+0x47:
00401057 c7050000000000000000 mov dword ptr ds:[0],0 ds:002b:00000000=?????????

0:000> kL
ChildEBP RetAddr
0012ff70 00401403 MissingThread!wmain+0x47
0012ffc0 7d4e7d2a MissingThread!__tmainCRTStartup+0x15e
0012fff0 00000000 kernel32!BaseProcessStart+0x28
```

If live debugging is not possible and we are interested in crash dumps saved upon a first chance exception before it is processed by an exception handler we can also use MS userdump tool after we install it and enable All Exceptions in the Process Monitoring Rules dialog box. Another tool can be used is ADPlus in crash mode from Debugging Tools for Windows.

## Comments

The thread might also be missing when it exists and leaves some synchronization objects in the non-signalled state:

```
0:088> !locks

CritSec DLL!MyFunctionEx+178 at 67caf998
WaiterWoken No
LockCount 71
RecursionCount 17
OwningThread 1910
EntryCount 0
ContentionCount 47
*** Locked
```

Thread 1910 is not present in the dump. Also !analyze -v -hang reports that the critical section is orphaned:

```
PRIMARY_PROBLEM_CLASS: APPLICATION_HANG_Orphaned_CriticalSection
```

## Mixed Exception

Sometimes we have **Managed Code Exception** (page 617) that was enveloping a handled unmanaged code exception, (**Mixed**) Nested Exception (page 723):

```
0:000> !analyze -v

[...]

EXCEPTION_RECORD: ffffffff -- (.exr 0xfffffffffffff)
ExceptionAddress: 00000000
  ExceptionCode: 80000003 (Break instruction exception)
  ExceptionFlags: 00000000
NumberParameters: 0

ERROR_CODE: (NTSTATUS) 0x80000003 - {EXCEPTION}  Breakpoint  A breakpoint has been reached.

FAULTING_THREAD: 00000cf0

[...]

EXCEPTION_OBJECT: !pe 1f9af1ac
Exception object: 1f9af1ac
Exception type: System.AccessViolationException
Message: Attempted to read or write protected memory. This is often an indication that other memory is
corrupt.
InnerException: <none>
StackTrace (generated):
    SP      IP          Function
  0012EF3C 28DD9AF9 DLLA!Component.getFirstField()+0x11
  [...]
  0012EFC8 7B194170 System_Windows_Forms_ni!System.Windows.Forms.Control.OnClick(System.EventArgs)+0x70
  0012EFE0 7B6F74B4
System_Windows_Forms_ni!System.Windows.Forms.Control.WmMouseUp(System.Windows.Forms.Message ByRef,
System.Windows.Forms.MouseButtons, Int32)+0x170
  0012F06C 7BA29B66
System_Windows_Forms_ni!System.Windows.Forms.Control.WndProc(System.Windows.Forms.Message ByRef)+0x861516
  0012F0C4 7B1D1D6A
System_Windows_Forms_ni!System.Windows.Forms.ScrollableControl.WndProc(System.Windows.Forms.Message
ByRef)+0x2a
  0012F0D0 7B1C8640
System_Windows_Forms_ni!System.Windows.Forms.Control+ControlNativeWindow.OnMessage(System.Windows.Forms.M
essage ByRef)+0x10
  0012F0D8 7B1C85C1
System_Windows_Forms_ni!System.Windows.Forms.Control+ControlNativeWindow.WndProc(System.Windows.Forms.M
essage ByRef)+0x31
  0012F0EC 7B1C849A System_Windows_Forms_ni!System.Windows.Forms.NativeWindow.Callback(IntPtr, Int32,
IntPtr, IntPtr)+0x5a

[...]
```

We see that it was an access violation exception and check the thread with TID cfc:

```
0:000> kL
ChildEBP RetAddr
0012db54 77d70dde ntdll!KiFastSystemCallRet
0012db58 7b1d8e48 user32!NtUserWaitMessage+0xc
0012dbec 7b1d8937 System_Windows_Forms_ni+0x208e48
0012dc44 7b1d8781 System_Windows_Forms_ni+0x208937
0012dc74 7b6edd1f System_Windows_Forms_ni+0x208781
0012dc8c 7b72246b System_Windows_Forms_ni+0x71dd1f
0012dd18 7b722683 System_Windows_Forms_ni+0x75246b
0012dd58 7b6f77f6 System_Windows_Forms_ni+0x752683
0012dd64 7b6fa27c System_Windows_Forms_ni+0x7277f6
0012f148 77d6f8d2 System_Windows_Forms_ni+0x72a27c
0012f174 77d6f794 user32!InternalCallWinProc+0x23
0012f1ec 77d70008 user32!UserCallWinProcCheckWow+0x14b
0012f250 77d70060 user32!DispatchMessageWorker+0x322
0012f260 0a1412fa user32!DispatchMessageW+0xf
WARNING: Frame IP not in any known module. Following frames may be wrong.
0012f27c 578439f7 0xa1412fa
0012f2ec 578430c9 WindowsBase_ni+0x939f7
0012f2f8 5784306c WindowsBase_ni+0x930c9
0012f304 55bed46e WindowsBase_ni+0x9306c
0012f310 55bec76f PresentationFramework_ni+0x1cd46e
0012f334 55bd3aa6 PresentationFramework_ni+0x1cc76f
```

If there was an exception it must be **Hidden Exception** (page 457) so we inspect the thread raw stack:

```
0:000> !teb
TEB at 7ffd000
  ExceptionList:      0012e470
  StackBase:          00130000
  StackLimit:         0011e000
  SubSystemTib:       00000000
  FiberData:          00001e00
  ArbitraryUserPointer: 00000000
  Self:               7ffd000
  EnvironmentPointer: 00000000
  ClientId:           00000b6c . 00000cf0
  RpcHandle:          00000000
  Tls Storage:        7ffd02c
  PEB Address:        7ffd4000
  LastErrorValue:     0
  LastStatusValue:    c0000139
  Count Owned Locks:  0
  HardErrorMode:      0

0:000> dps 0011e000 00130000
0011e000 00000000
0011e004 00000000
0011e008 00000000
[...]
0012e72c 00130000
0012e730 0011e000
0012e734 00ee350d
```

```

0012e738 0012ea3c
0012e73c 77f299f7 ntdll!KiUserExceptionDispatcher+0xf
0012e740 0012e750
0012e744 0012e76c
0012e748 0012e750
0012e74c 0012e76c
0012e750 c0000005
0012e754 00000000
0012e758 00000000
0012e75c 77f17d89 ntdll!RtlLeaveCriticalSection+0x9
0012e760 00000002
0012e764 00000001
0012e768 00000028
0012e76c 0001003f
0012e770 00000000
0012e774 00000000
0012e778 00000000
0012e77c 00000000
[...]

0:000> .cxr 0012e76c
eax=00000020 ebx=09ca1fa0 ecx=781c1b78 edx=00000001 esi=00000020 edi=09ca1ff8
eip=77f17d89 esp=0012ea38 ebp=0012ea3c iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010202
ntdll!RtlLeaveCriticalSection+0x9:
77f17d89 834608ff add dword ptr [esi+8],0FFFFFFFh ds:0023:00000028=?????????

0:000> kL
*** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr
0012ea3c 7813e5b5 ntdll!RtlLeaveCriticalSection+0x9
0012ea44 2071c9ba msocr80!_unlock_file+0x35
WARNING: Stack unwind information not available. Following frames may be wrong.
0012ea68 2071c31e D11B!getType+0x286a
0012ee34 206bbfb8 D11B!getType+0x3eb
0012ee68 206c8abb D11C+0xbfb8
0012ee98 79e71ca7 D11C!getFirstField+0x3b
0012f148 77d6f8d2 msocrwks!NDirectGenericStubReturnFromCall
0012f1ec 77d70008 user32!InternalCallWinProc+0x23
0012f240 77db51b9 user32!DispatchMessageWorker+0x322
0012f4a4 79e95feb user32!_W32ExceptionHandler+0x18
0012f4fc 79e968b0 msocrwks!MetaSig::HasRetBuffArg+0x5
0012f50c 79e9643e msocrwks!MetaSig::MetaSig+0x3a
0012f610 79e96534 msocrwks!MethodDesc::CallDescr+0xaf
0012f62c 79e96552 msocrwks!MethodDesc::CallTargetWorker+0x1f
0012f644 79effa45 msocrwks!MethodDescCallSite::CallWithValueTypes+0x1a
0012f7a8 79eff965 msocrwks!ClassLoader::RunMain+0x223

```

Therefore, we identified *DIB* and *DIIC* components as suspicious. If we check the exception chain, we see that .NET runtime registered **Custom Exception Handlers** (page 171):

```
0:000> !exchain
0012e470: mscorwks!COMPlusNestedExceptionHandler+0 (79edd6d7)
0012f13c: mscorwks!FastNExportExceptHandler+0 (7a00a2e7)
0012f1dc: user32!_except_handler4+0 (77db51ba)
0012f240: user32!_except_handler4+0 (77db51ba)
0012f46c: mscorwks!COMPlusFrameHandler+0 (79edc3bc)
0012f4c0: mscorwks!_except_handler4+0 (79f908a2)
0012f798: mscorwks!_except_handler4+0 (79f908a2)
0012fa04: mscorwks!GetManagedNameForTypeInfo+a680 (7a328d90)
0012fed4: mscorwks!GetManagedNameForTypeInfo+82c8 (7a325a3a)
0012ff20: mscorwks!_except_handler4+0 (79f908a2)
0012ff6c: mscorwks!GetManagedNameForTypeInfo+a6e (7a319ee4)
0012ffc4: ntdll!_except_handler4+0 (77ed9834)
Invalid exception stack at ffffffff
```

We check that *GetManagedNameForTypeInfo+a6e* (7a319ee4) is an exception handler indeed:

```
0:000> .asm no_code_bytes
Assembly options: no_code_bytes

0:000> uf 7a319ee4
msvcr80!__CxxFrameHandler:
78158aeb push ebp
78158aec mov ebp,esp
78158aee sub esp,8
78158af1 push ebx
78158af2 push esi
78158af3 push edi
78158af4 cld
78158af5 mov dword ptr [ebp-4],eax
78158af8 xor eax, eax
78158afa push eax
78158afb push eax
78158afc push eax
78158afd push dword ptr [ebp-4]
78158b00 push dword ptr [ebp+14h]
78158b03 push dword ptr [ebp+10h]
78158b06 push dword ptr [ebp+0Ch]
78158b09 push dword ptr [ebp+8]
78158b0c call msvcr80!__InternalCxxFrameHandler (7815897e)
78158b11 add esp,20h
78158b14 mov dword ptr [ebp-8],eax
78158b17 pop edi
78158b18 pop esi
78158b19 pop ebx
78158b1a mov eax,dword ptr [ebp-8]
78158b1d mov esp,ebp
78158b1f pop ebp
78158b20 ret

mscorwks!__CxxFrameHandler3:
79f5f258 jmp dword ptr [mscorwks!_imp___CxxFrameHandler3 (79e711c4)]
```

```
mscorwks!GetManagedNameForTypeInfo+0xa6e:
7a319ee4 mov edx,dword ptr [esp+8]
7a319ee8 lea eax,[edx+0Ch]
7a319eeb mov ecx,dword ptr [edx-30h]
7a319eee xor ecx,ecx
7a319ef0 call mscorwks!__security_check_cookie (79e72037)
7a319ef5 mov eax,offset mscorwks!_CT??_R0H+0xc14 (7a319f00)
7a319efa jmp mscorwks!__CxxFrameHandler3 (79f5f258)
```

## Comments

---

The other way around, when CLR exception is enveloped by unmanaged exception is possible like in ntdebugging blog example<sup>139</sup>.

One of the questions asked: From

```
0012e73c 77f299f7 ntdll!KiUserExceptionDispatcher+0xf
0012e740 0012e750
0012e744 0012e76c
```

Does this correspond to the args for the call to *ntdll!RtlDispatchException*? If so, is then 0012e76c the 2nd arg to *ntdll!RtlDispatchException* that is the context record?

A. Yes, it corresponds because it calls it:

```
0:000> ub 77f299f7
ntdll!KiUserCallbackDispatcher+0x42:
77f299e2 ret 0Ch
77f299e5 lea ecx,[ecx]
ntdll!KiUserExceptionDispatcher:
77f299e8 cld
77f299e9 mov ecx,dword ptr [esp+4]
77f299ed mov ebx,dword ptr [esp]
77f299f0 push ecx
77f299f1 push ebx
77f299f2 call ntdll!RtlDispatchException (77f0d132)
```

---

<sup>139</sup> <http://blogs.msdn.com/b/ntdebugging/archive/2014/05/28/debugging-a-windows-8-1-store-app-crash-dump-part-2.aspx>

## Module Collection

### General

In addition to **Stack Trace Collection** (page 943) we are often interested in **Module Collection** (we initially called this pattern **Vendor Collection**), especially if we would like to check whether a particular vendor DLL is present in some process address space in a complete memory dump (kernel module list or module list from a process memory dump is trivial). Or we need to check for some vendor information from a problem description (**!lmv** command). If we have a complete memory dump from x64 system then listing modules for each process is not enough. For example, we might have this:

```
0: kd> !mu
start           end             module name
00000000`00ab0000 00000000`00ae8000  AppA      (deferred)
00000000`74fe0000 00000000`7502e000  wow64win  (deferred)
00000000`75030000 00000000`75075000  wow64     (deferred)
00000000`750c0000 00000000`750c9000  wow64cpu   (deferred)
00000000`77b70000 00000000`77cf7000  ntdll     (pdb symbols)
```

*AppA* is a 32-bit process and has an additional 32-bit module list that is more useful. We can set x86 context for a thread from that process and get the list of 32-bit modules:

```
0: kd> .load wow64exts

0: kd> .thread /w ffffffa800e372060
Implicit thread is now ffffffa80 0e372060
x86 context set

0: kd:x86> .reload
Loading Kernel Symbols
Loading User Symbols
Loading unloaded module list
Loading Wow64 Symbols

0: kd:x86> !mu
start           end             module name
00000000`00ab0000 00000000`00ae8000  AppA      (deferred)
00000000`73490000 00000000`73515000  COMCTL32  (deferred)
00000000`73520000 00000000`735c3000  MSVCR90  (deferred)
00000000`735d0000 00000000`7365e000  MSVCP90  (deferred)
00000000`74920000 00000000`7493e000  USERENV   (deferred)
00000000`74940000 00000000`74ade000  comctl32_74940000 (deferred)
00000000`74af0000 00000000`74b02000  MSASN1    (deferred)
00000000`74b10000 00000000`74c03000  CRYPT32   (deferred)
00000000`74dc0000 00000000`74e5b000  MSVCR80  (deferred)
00000000`74f60000 00000000`74fd6000  NETAPI32  (deferred)
00000000`74fe0000 00000000`7502e000  wow64win  (deferred)
00000000`75030000 00000000`75075000  wow64     (deferred)
00000000`750b0000 00000000`750ba000  WTSAPI32 (deferred)
00000000`750c0000 00000000`750c9000  wow64cpu   (deferred)
```

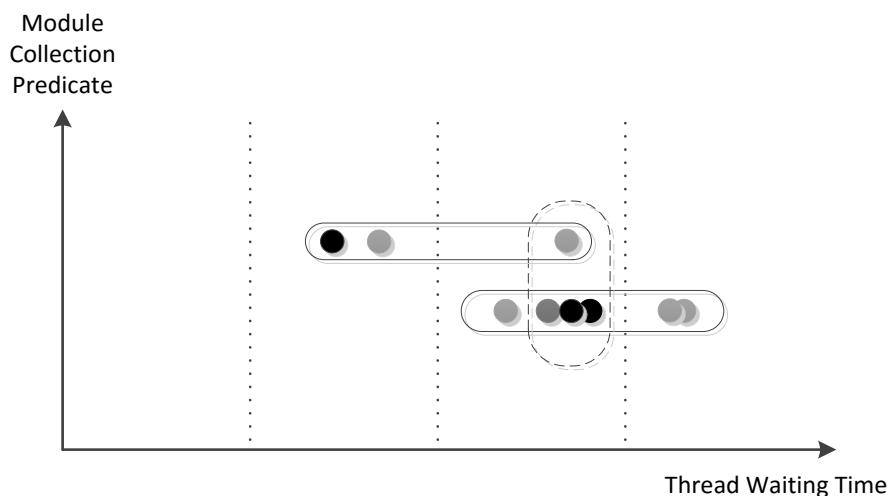
00000000`75cf0000 00000000`75d50000	Secur32	(deferred)
00000000`75d50000 00000000`76861000	SHELL32	(deferred)
00000000`76a10000 00000000`76aa0000	GDI32	(deferred)
00000000`76b30000 00000000`76b90000	IMM32	(deferred)
00000000`76be0000 00000000`76cf0000	kernel32	(deferred)
00000000`76e30000 00000000`76f75000	ole32	(deferred)
00000000`76f80000 00000000`7702a000	msvcrt	(deferred)
00000000`77030000 00000000`77037000	PSAPI	(deferred)
00000000`77040000 00000000`77110000	USER32	(deferred)
00000000`77110000 00000000`77169000	SHLWAPI	(deferred)
00000000`77170000 00000000`771ed000	USP10	(deferred)
00000000`77380000 00000000`7740d000	OLEAUT32	(deferred)
00000000`77640000 00000000`77649000	LPK	(deferred)
00000000`776e0000 00000000`777d0000	RPCRT4	(deferred)
00000000`777d0000 00000000`77898000	MSCTF	(deferred)
00000000`778a0000 00000000`77966000	ADVAPI32	(deferred)
00000000`77b70000 00000000`77cf7000	ntdll	(pdb symbols)
00000000`77d30000 00000000`77e90000	ntdll_77d30000 #	(pdb symbols)

So it looks like we need to dump modules for each thread. However, the output would be enormous unless we skip threads having the same PID. After some tinkering we wrote this WinDbg script with moderate output volume:

```
.load wow64exts
!for_each_thread ".thread @#Thread; .if (@$t0 != @@c++(@$thread->Cid.UniqueProcess)) { .reload
/user;lmvu;.thread /w @#Thread;.reload /user;lmvu;r $t0 = @@c++(@$thread->Cid.UniqueProcess); .effmach
AMD64; }"
```

## Predicate

While working on **Thread Cluster** pattern (page 1003) we realized that we need a predicate version of **Module Collection** pattern, similar to the predicate version of **Stack Trace Collection** (page 943) pattern. A predicate can be anything: company vendors, semantic proximity, functionality such as printing, remote file management, and so on. Such module sub-collections can be used instead of modules in more complex patterns: an example of software diagnostics pattern substitution and composition. For example, we might be able to identify a possible coupling between 2 semantically different module groups explained by IPC **Wait Chains** (page 1082) such as on this diagram:



## Module Hint

This pattern is frequently observed in dynamic memory corruption<sup>140</sup> incidents. It is similar to **Execution Residue** (page 371), or **String Parameter** (page 962) patterns where we have ASCII or UNICODE fragments providing troubleshooting and debugging hints. **Module Hint** is, therefore, a more specialized pattern where we can link module names to raw data. For example, a kernel memory dump saved after the detected pool corruption (**Dynamic Memory Corruption** pattern, page 292) shows *P12345.DLL* module name in a pool entry that can provide a link to the corresponding functionally to be reconfigured or removed:

```
BAD_POOL_HEADER (19)
The pool is already corrupt at the time of the current request.
This may or may not be due to the caller.
The internal pool links must be walked to figure out a possible cause of the problem, and then special
pool applied to the suspect tags or the driver verifier to a suspect driver.
Arguments:
Arg1: 00000020, a pool block header size is corrupt.
Arg2: 8b79d078, The pool entry we were looking for within the page.
Arg3: 8b79d158, The next pool entry.
Arg4: 8a1c0004, (reserved)

STACK_TEXT:
b3e0aa4c 808947bb 00000019 00000020 8b79d078 nt!KeBugCheckEx+0x1b
b3e0aab4 b368c00f 8b79d080 00000000 00000000 nt!ExFreePoolWithTag+0x477
b3e0aac4 b366c68e 8b79d080 00000000 00000000 DriverA!MemFree+0xf
[...]
b3e0ac44 8081e0c3 808f77c9 b3e0ac64 808f77c9 nt!IovCallDriver+0x112
b3e0ac50 808f77c9 8a8eef60 8b6862a8 8a8eef60 nt!IofCallDriver+0x13
b3e0ac64 808f856b 8ce456b0 8a8eef60 8b6862a8 nt!IopSynchronousServiceTail+0x10b
b3e0ad00 808f109a 000009dc 00000000 00000000 nt!IopXxxControlFile+0x5e5
b3e0ad34 8088b45c 000009dc 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
b3e0ad34 7c82847c 000009dc 00000000 00000000 nt!KiFastCallEntry+0xfc
WARNING: Frame IP not in any known module. Following frames may be wrong.
00f5fb18 00000000 00000000 00000000 00000000 0x7c82847c

2: kd> !pool 8b79d080
Pool page 8b79d080 region is Unknown
8b79d000 size: 30 previous size: 0 (Allocated) FSfm
8b79d030 size: 28 previous size: 30 (Allocated) VadS
8b79d058 size: 20 previous size: 28 (Allocated) ReEv
*8b79d078 size: e0 previous size: 20 (Allocated) *DRIV
Owning component : Unknown (update pooltag.txt)
8b79d158 is not a valid large pool allocation, checking large session pool...
8b79d158 is freed (or corrupt) pool
Bad previous allocation size @8b79d158, last size was 1c

*** 
*** An error (or corruption) in the pool was detected;
```

<sup>140</sup> <http://www.dumpanalysis.org/blog/index.php/2009/02/17/dynamic-memory-corruption-patterns/>

```
*** Pool Region unknown (0xFFFFFFFF8B79D158)
***  
*** Use !poolval 8b79d000 for more details.  
***  
  
2: kd> dc 8b79d078  
8b79d078 [...] ..DRIV .....AP  
8b79d088 [...] P12345.DLL.....  
8b79d098 [...] .....<%n.....  
8b79d0a8 [...] ....$....:F...X.  
[...]
```

## Comments

---

Another example is the presence of debugging or error handling modules in the module list.

It can also be a valid (non-coincidental, see **Coincidental Symbolic Information** analysis pattern, page 137) symbolic name via **dps** or **dpS** command.

We can also use search commands **s-sa** and **s-su**.

## Module Product Process

If we found module related patterns (page 1167) in a complete memory dump and suspect a particular module, it may be worth looking at module product related process if it exists especially if this module (component, DLL) has product information or some related hint (`!lmv` or `!lmi` commands). In complex environments, such modules may be loaded not only by hooking mechanisms but also as plugins. If we are not sure whether there is such a process, the best way is to get **Module Collection** (page 693) and find a process module that has the same vendor as the module in question. Then such process should also be analyzed for anomalies.

## Module Stack Trace

### Linux

This is a Linux variant of **Module Stack Trace** pattern previously described for Windows platform (page 700). Linux core dumps are **Abridged Dumps** (page 49) by default with shared libraries code and paged out pages missing. To enable saving full process dumps use this command (see core man page<sup>141</sup> for more details):

```
[training@localhost CentOS]$ echo 0x7f > /proc/$$/coredump_filter
```

Compare the file sizes for *sleep* process core dump generated before (*core.3252*) and after (*core.3268*) changing the *coredump\_filter* value:

```
[training@localhost CentOS]$ ls -l
-rwxrwxrwx. 1 root root 323584 Oct 3 07:39 core.3252
-rwxrwxrwx. 1 root root 103337984 Oct 3 07:40 core.3268
```

Although GDB is not able to get a symbolic stack trace for both dumps above due to the absence of symbols, CODA tool<sup>142</sup> is able to show stack trace variant with modules (with **Reduced Symbolic Information**, page 829):

```
(gdb) bt
#0 0x00000032bd4accc0 in ?? ()
#1 0x0000000000403ce8 in ?? ()
#2 0x000000000000004d2 in ?? ()
#3 0x0000000000000000 in ?? ()

[training@localhost CentOS]$ ./coda/coda -i core.3268
Welcome to coda interactive command line.
THE SOFTWARE IS PROVIDED 'AS IS', WITHOUT WARRANTY OF ANY KIND.
Supported on x86_64-linux.
Coredump generated by command line => sleep 1234
coda > bt
[0] 0x00000032bd4accc0 <0x00000032bd4accb0 - 0x00000032bd4acd0e> __nanosleep+0x10 [RO TEXT]:/lib64/libc.so.6
[1] 0x0000000000403ce8 <0x0000000000403c40 - 0x0000000000403cf3> close_stdout+0x2378 [RO TEXT]:sleep
[2] 0x000000000040336d <0x00000000004032c0 - 0x00000000004033e9> close_stdout+0x19fd [RO TEXT]:sleep
[3] 0x00000000004016bc <0x00000000004014c0 - 0x0000000000401775> usage+0x3fc [RO TEXT]:sleep
[4] 0x00000032bd41ed1d <0x00000032bd41ec20 - 0x00000032bd41ede7> __libc_start_main+0xfd [RO TEXT]:/lib64/libc.so.6
[5] 0x00000000004011f9 <0x00000000004012c0 - 0x00000000004014bb> ?? [RO TEXT]:sleep
[6] 0x0007ffff89d82a68 <----RANGE UNKNOWN----> ?? [RW DATA]:
```

<sup>141</sup> <http://man7.org/linux/man-pages/man5/core.5.html>

<sup>142</sup> <https://github.com/npamnani/coda>

## Windows

For completeness, we introduce **Module Stack Trace** analysis pattern, which is a stack trace with modules (or components) and offsets into their address range. Offsets distinguish it from **Collapsed Stack Trace**<sup>143</sup>. We see only modules names in case of **No Component Symbols** (page 734) or **Unrecognizable Symbolic Information** (page 1045). Sometimes, we also have exported functions present with resulting offsets reduced in value but still large. This is a case of **Reduced Symbolic Information** (page 829). Such emerging symbols may form **Incorrect Stack Trace** (page 499) frames. On some platforms **Module Stack Traces** become **Truncated Stack Traces** (page 1015) because a debugger is not able to reconstruct the stack trace. The following stack traces shows **Module Stack Trace** fragment for PHOTOSAPP\_WINDOWS (and also frames with exported functions for *ntdll*, *combase*, *twinapi\_appcore*):

```
0:034> kL
# Child-SP RetAddr Call Site
00 0000005b`03acc5c0 00007ff9`b6288935 KERNELBASE!RaiseFailFastException+0x74
01 0000005b`03accb90 00007ff9`b399654f combase!RoParameterizedTypeExtraGetTypeSignature+0x8db5
02 0000005b`03acccd0 00007ff9`b39965d0 twinapi_appcore!BiNotifyNewSession+0x2628f
03 0000005b`03accd10 00007ff9`b6186e1a twinapi_appcore!BiNotifyNewSession+0x26310
04 0000005b`03accd40 00007ff9`9eaca4a0 combase!RoReportUnhandledError+0xea
05 0000005b`03accdc0 00007ff9`af37be60
MSVCP140_APP!Concurrency::details::_ExceptionHolder::ReportUnhandledError`::`1'::catch$3+0x39
06 0000005b`03acce00 00007ff9`af3729b2 VCRUNTIME140_APP!CallSettingFrame+0x20
07 0000005b`03acce30 00007ff9`b8625c53 VCRUNTIME140_APP!_CxxCallCatchBlock+0x122
08 0000005b`03accef0 00007ff9`9ea99129 ntdll!RtlCaptureContext+0x3c3
09 0000005b`03acf490 00007ff9`861137e6
MSVCP140_APP!Concurrency::details::_ExceptionHolder::ReportUnhandledError+0x29
0a 0000005b`03acf4e0 00007ff9`86112142 PhotosApp_Windows+0x737e6
0b 0000005b`03acf520 00007ff9`86111e6c PhotosApp_Windows+0x72142
0c 0000005b`03acf560 00007ff9`86113e38 PhotosApp_Windows+0x71e6c
0d 0000005b`03acf590 00007ff9`8611307d PhotosApp_Windows+0x73e38
0e 0000005b`03acf5f0 00007ff9`86113619 PhotosApp_Windows+0x7307d
0f 0000005b`03acf630 00007ff9`b85caef9 PhotosApp_Windows+0x73619
10 0000005b`03acf680 00007ff9`b85c97ea ntdll!EtwEventRegister+0x1e3a
11 0000005b`03acf790 00007ff9`b7df2d92 ntdll!EtwEventRegister+0x72a
12 0000005b`03acfb90 00007ff9`b8599f64 kernel32!BaseThreadInitThunk+0x22
13 0000005b`03acfb0 00000000`00000000 ntdll!RtlUserThreadStart+0x34
```

<sup>143</sup> Memory Dump Analysis Anthology, Volume 3, page 381

## Module Variable

In addition to functions, we also have module variables like `nt!MmPagedPoolCommit` in Windows 7:

```
0: kd> x nt!MmPagedPool*
fffff800`031148d0 nt!MmPagedPoolInfo = <no type information>
fffff800`03092d20 nt!MmPagedPoolCommit = <no type information>
fffff800`031141a0 nt!MmPagedPoolEnd = <no type information>
fffff800`031175c0 nt!MmPagedPoolWs = <no type information>
```

If we are not sure whether we have a function or **Module Variable** we can try to disassemble its address:

```
0: kd> u nt!MmPagedPoolCommit
nt!MmPagedPoolCommit:
fffff800`03092d20 e3b2 jrcxz nt!MmTotalNonPagedPoolQuota+0x4 (fffff800`03092cd4)
fffff800`03092d22 0000 add byte ptr [rax],al
fffff800`03092d24 0000 add byte ptr [rax],al
fffff800`03092d26 0000 add byte ptr [rax],al
fffff800`03092d28 0000 add byte ptr [rax],al
fffff800`03092d2a 0000 add byte ptr [rax],al
fffff800`03092d2c 0000 add byte ptr [rax],al
fffff800`03092d2e 0000 add byte ptr [rax],al
```

Here the value is probably in pages, so we multiply by 4 to get a value in Kb and compare to the output of **!vm** command:

```
0: kd> dp nt!MmPagedPoolCommit
fffff800`03092d20 00000000`0000b2e3 00000000`00000000
fffff800`03092d30 00000000`00000000 00000000`00000000
fffff800`03092d40 00000000`00000001 00000000`00000000
fffff800`03092d50 00000000`00000000 00000000`00060187
fffff800`03092d60 fffff800`03092d60 fffff800`03092d60
fffff800`03092d70 00000000`00000000 00000000`0001e972
fffff800`03092d80 fffff900`c0000000 00000000`00000002
fffff800`03092d90 fffff880`071dc0a8 fffff880`057340a8

0: kd> ? b2e3 * 4
Evaluate expression: 183180 = 00000000`0002cb8c
```

```
0: kd> !vm

*** Virtual Memory Usage ***
Physical Memory:      1035228 ( 4140912 Kb)
Page File:             \??\C:\pagefile.sys
Current: 4448112 Kb    Free Space: 4448108 Kb
Minimum: 4448112 Kb    Maximum: 12422736 Kb
Unimplemented error for MiSystemVaTypeCount
Available Pages:      594029 ( 2376116 Kb)
ResAvail Pages:       889795 ( 3559180 Kb)
Locked IO Pages:      0 ( 0 Kb)
Free System PTEs:     33556870 ( 134227480 Kb)
Modified Pages:       20079 ( 80316 Kb)
Modified PF Pages:    19441 ( 77764 Kb)
NonPagedPool Usage:   50865104 ( 203460416 Kb)
NonPagedPoolNx Usage: 28163 ( 112652 Kb)
NonPagedPool Max:    763396 ( 3053584 Kb)
***** Excessive NonPaged Pool Usage *****
PagedPool 0 Usage:   39420 ( 157680 Kb)
PagedPool 1 Usage:   5194 ( 20776 Kb)
PagedPool 2 Usage:   367 ( 1468 Kb)
PagedPool 3 Usage:   338 ( 1352 Kb)
PagedPool 4 Usage:   440 ( 1760 Kb)
PagedPool Usage:     45759 ( 183036 Kb)
PagedPool Maximum:   33554432 ( 134217728 Kb)
Session Commit:      8112 ( 32448 Kb)
Shared Commit:       31802 ( 127208 Kb)
Special Pool:        0 ( 0 Kb)
Shared Process:      10765 ( 43060 Kb)
PagedPool Commit:   45795 ( 183180 Kb)
Driver Commit:       13773 ( 55092 Kb)
Committed pages:    540998 ( 2163992 Kb)
Commit limit:        2146794 ( 8587176 Kb)
[...]
```

Knowledge of available module variables is useful because some of them are not included in WinDbg extension command output. For their list, please consult Windows Internals book. Useful variables can also be found in other modules as well, for example, `srv!srvcomputername`<sup>144</sup>:

```
0: kd> dS srv!srvcomputername
fffff8a0`0344b090 "MYNOTEBOOK"
```

---

<sup>144</sup> Where did the Crash Dump Come from?, Memory Dump Analysis Anthology, Volume 1, page 616

## Module Variety

Sometimes when we look at the list of loaded modules in a process address space, we see an instance of the pattern that we call **Module Variety**. It means, literally, that there are so many different loaded modules that we start thinking that their coexistence created the problem. We can also call this pattern **Component Variety** or **DLL Variety**, but we prefer the former because WinDbg refers to loaded executables, DLLs, drivers, ActiveX controls as modules.

Modules can be roughly classified into 4 broad categories:

- Application modules - components that were developed specifically for this application, one of them is the main application module.
- 3rd-party modules - we can easily identify them if the company name is the same in the output of **!mv** WinDbg command.
- Common system modules - Windows DLLs supplied by OS and implementing native OS calls, Windows API and also C/C++ runtime functions, for example, *ntdll.dll*, *kernel32.dll*, *user32.dll*, *gdi32.dll*, *advapi32.dll*, *msvcrt.dll*.
- Specific system modules - optional Windows DLLs supplied by Microsoft that are specific to the application functionality and implementation, like MFC DLLs, .NET runtime, or *tapi32.dll*.

Although **!mv** is verbose for a quick check of component timestamps, we can use **!mt** WinDbg command. Here is an example of the great module variety from Windows Server 2003:

```
Loading Dump File [application.dmp]
...
...
...
Windows Server 2003 Version 3790 (Service Pack 1)
...
...
...
0:001> !mt
start      end        module name
00400000 030ba000  app_main    Mon Dec  4 21:22:42 2006
04120000 04193000  Dformd     Mon Jan 31 02:27:58 2000
041a0000 04382000  sqllib2    Mon May 29 22:50:11 2006
04490000 044d3000  udNet      Mon May 29 23:22:43 2006
04e30000 04f10000  abchook    Wed Aug  1 20:47:17 2006
05e10000 05e15000  token_manager Fri Mar 12 11:54:17 1999
06030000 06044000  ODBCINT    Thu Mar 24 22:59:58 2005
06150000 0618d000  sgl5NET    Mon May 29 23:25:22 2006
06190000 0622f000  OPENGL32   Mon Nov  6 21:30:52 2006
06230000 06240000  pwrpc32   Thu Oct 22 16:22:40 1998
06240000 07411000  app_d11_1  Tue Aug  8 12:14:39 2006
07420000 07633000  app_d11_2  Mon Dec  4 22:11:59 2006
07640000 07652000  zlib       Fri Aug 30 08:12:24 2002
07660000 07f23000  app_d11_3  Wed Oct 19 11:43:34 2005
0dec0000 0dedc000  app_d11_4  Mon Dec  4 22:11:36 2006
```

10000000 110be000	des	Tue Jul 18 20:42:02 2006
129c0000 12f1b000	xpsp2res	Fri Mar 25 00:26:47 2005
1b000000 1b170000	msjet40	Tue Jul 06 19:16:05 2004
1b2c0000 1b2cd000	msjter40	Thu May 09 19:09:53 2002
1b2d0000 1b2ea000	msjint40	Thu May 09 19:09:53 2002
1b570000 1b5c5000	msjetoledb40	Thu Nov 13 23:40:06 2003
1b5d0000 1b665000	mswstr10	Thu May 09 19:09:56 2002
1e000000 1e0f0000	python23	Fri Jan 30 13:03:24 2004
4b070000 4b0c1000	MSCTF	Fri Mar 25 02:10:36 2005
4b610000 4b64d000	ODBC32	Fri Mar 25 02:09:33 2005
4b9e0000 4ba59000	OLEDB32	Fri Mar 25 02:09:56 2005
4c310000 4c31d000	OLEDB32R	Fri Mar 25 02:09:57 2005
4c3b0000 4c3de000	MSCTFIME	Fri Mar 25 02:10:37 2005
5f400000 5f4f2000	mfc42	Wed Oct 27 22:35:22 1999
62130000 6213d000	mfc42loc	Wed Mar 26 03:35:58 2003
62460000 6246e000	msadrh15	Fri Mar 25 02:10:29 2005
65340000 653d2000	OLEAUT32	Wed Sep 01 00:15:11 1999
68000000 6802f000	rsaenh	Fri Mar 25 00:30:55 2005
68a50000 68a70000	glu32	Fri Mar 25 02:09:03 2005
71990000 71998000	wshtcpip	Wed Mar 26 03:34:24 2003
71a80000 71a91000	mpr	Wed Mar 26 03:34:24 2003
71aa0000 71aa8000	ws2help	Fri Mar 25 02:10:19 2005
71ab0000 71ac7000	ws2_32	Fri Mar 25 02:10:18 2005
71ad0000 71ae2000	tsappcmp	Fri Mar 25 02:09:56 2005
71af0000 71b48000	netapi32	Fri Aug 11 11:00:07 2006
72ec0000 72ee7000	winspool	Fri Mar 25 02:09:48 2005
73290000 73295000	riched32	Wed Mar 26 03:34:14 2003
73ee0000 73ee5000	icmp	Wed Mar 26 03:34:09 2003
74920000 7493a000	msdart	Fri Mar 25 02:10:48 2005
74b10000 74b80000	riched20	Fri Mar 25 02:09:36 2005
75220000 75281000	usp10	Fri Mar 25 02:09:51 2005
75810000 758d0000	userenv	Fri Mar 25 02:09:50 2005
75d00000 75d27000	apphelp	Fri Mar 25 02:09:21 2005
76120000 7613d000	imm32	Fri Mar 25 02:09:37 2005
76140000 76188000	comdlg32	Fri Mar 25 02:10:11 2005
76810000 76949000	comsvcs	Fri Aug 26 23:19:45 2005
76a60000 76a6b000	psapi	Fri Mar 25 02:09:57 2005
76c00000 76c1a000	iphlpapi	Fri May 19 04:21:07 2006
76de0000 76e0f000	dnsapi	Wed Jul 12 20:02:12 2006
76e20000 76e4e000	wldap32	Fri Mar 25 02:09:59 2005
76e60000 76e73000	secur32	Fri Mar 25 02:10:01 2005
76e80000 76e87000	winrnrr	Fri Mar 25 02:09:45 2005
76e90000 76e98000	rasadhlp	Wed Jul 12 20:02:15 2006
76f20000 77087000	comres	Wed Mar 26 03:33:48 2003
77330000 773c7000	comctl32	Mon Aug 28 09:26:02 2006
77470000 775a4000	ole32	Thu Jul 21 04:25:12 2005
77640000 776c3000	clbcatq	Thu Jul 21 04:25:13 2005
77b30000 77b38000	version	Fri Mar 25 02:09:50 2005
77b40000 77b9a000	msvcrt	Fri Mar 25 02:11:59 2005
77ba0000 77be8000	gdi32	Tue Mar 07 03:55:05 2006
77bf0000 77c8f000	rpcrt4	Fri Mar 25 02:09:42 2005
77ca0000 77da3000	comctl32_77ca0000	Mon Aug 28 09:25:59 2006
77db0000 77dc1000	winsta	Fri Mar 25 02:09:51 2005
77de0000 77e71000	user32	Fri Mar 25 02:09:49 2005
77e80000 77ed2000	shlwapi	Wed Sep 20 01:33:12 2006
77ee0000 77ef1000	regapi	Fri Mar 25 02:09:51 2005

77f20000 77fc0000	advapi32	Fri Mar 25 02:09:06 2005
780a0000 780b2000	MSVCIRT	Wed Jun 17 19:45:46 1998
780c0000 78121000	MSVCP60	Wed Jun 17 19:52:10 1998
79040000 79085000	fusion	Fri Feb 18 20:57:41 2005
79170000 79198000	mscoree	Fri Feb 18 20:57:48 2005
791b0000 79417000	mscorwks	Fri Feb 18 20:59:56 2005
79510000 79523000	mscorsn	Fri Feb 18 20:30:38 2005
79780000 7998c000	mscorlib	Fri Feb 18 20:48:36 2005
79990000 79cce000	mscorlib_79990000	Thu Nov 02 04:53:27 2006
7c340000 7c396000	msvcr71	Fri Feb 21 12:42:20 2003
7c800000 7c93e000	kernel32	Tue Jul 25 13:37:16 2006
7c940000 7ca19000	ntdll	Fri Mar 25 02:09:53 2005
7ca20000 7d20a000	shell32	Thu Jul 13 13:58:56 2006

We can also use **!mtD** command to take the advantage of WinDbg hypertext commands. In that case, we can quickly click on a module name to view its detailed information.

We see that some components are very old, 1998-1999, and some are from 2006. We also see 3rd-party libraries: OpenGL, Visual Fortran RTL, Python language runtime. Common system modules include two versions of C/C++ runtime library, 6.0 and 7.0. Specific system modules include MFC and .NET, MSJET, ODBC and OLE DB support. There is a sign of “DLL Hell<sup>145</sup>” here too. OLE Automation DLL in system32 folder seems to be very old and doesn’t correspond to Windows 2003 SP1 which should have file version 5.2.3790.1830:

```
0:001> !mv m OLEAUT32
start      end        module name
65340000 653d2000  OLEAUT32  (deferred)
Image path: C:\WINDOWS\system32\OLEAUT32.DLL
Image name: OLEAUT32.DLL
Timestamp:   Wed Sep 01 00:15:11 1999 (37CC61FF)
CheckSum:    0009475A
ImageSize:   00092000
File version: 2.40.4277.1
Product version: 2.40.4277.1
File flags:   2 (Mask 3F) Pre-release
File OS:     40004 NT Win32
File type:   2.0 Dll
File date:   00000000.00000000
Translations: 0409.04e4
CompanyName: Microsoft Corporation
ProductName: Microsoft OLE 2.40 for Windows NT(TM) and Windows 95(TM) Operating Systems
InternalName: OLEAUT32.DLL
ProductVersion: 2.40.4277
FileVersion: 2.40.4277
FileDescription: Microsoft OLE 2.40 for Windows NT(TM) and Windows 95(TM) Operating Systems
LegalCopyright: Copyright © Microsoft Corp. 1993-1998.
LegalTrademarks: Microsoft® is a registered trademark of Microsoft Corporation. Windows NT(TM) and Windows 95(TM) are trademarks of Microsoft Corporation.
Comments:    Microsoft OLE 2.40 for Windows NT(TM) and Windows 95(TM) Operating Systems
```

<sup>145</sup> <http://web.archive.org/web/20091227014606/http://msdn.microsoft.com/en-us/library/ms811694.aspx>

## Multiple Exceptions

### Mac OS X

The Windows pattern in user mode (page 714) now has Mac OS X equivalent. In the example below there are 3 threads, and two of them experienced Data **NULL Pointer** (page 749) access violation exception:

```
(gdb) thread apply all bt full

Thread 3 (core thread 2):
#0 0x00000001062ffe4e in thread_two (arg=0x0)
at main.c:24
    p = (int *) 0x0
#1 0x00007fff8abf58bf in _pthread_start ()
No symbol table info available.
#2 0x00007fff8abf8b75 in thread_start ()
No symbol table info available.

Thread 2 (core thread 1):
#0 0x00000001062ffe1e in thread_one (arg=0x0)
at main.c:16
    p = (int *) 0x0
#1 0x00007fff8abf58bf in _pthread_start ()
No symbol table info available.
#2 0x00007fff8abf8b75 in thread_start ()
No symbol table info available.

Thread 1 (core thread 0):
#0 0x00007fff854e0e42 in __semwait_signal ()
No symbol table info available.
#1 0x00007fff8ababdea in nanosleep ()
No symbol table info available.
#2 0x00007fff8ababc2c in sleep ()
No symbol table info available.
#3 0x00000001062ffec3 in main (argc=1, argv=0x7fff65efeab8)
at main.c:36
    threadID_one = (pthread_t) 0x1063b4000
    threadID_two = (pthread_t) 0x106581000

(gdb) thread 2
[Switching to thread 2 (core thread 1)]
0x00000001062ffe1e in thread_one (arg=0x0)
at main.c:16
16      *p = 1;

(gdb) p/x p
$1 = 0x0
```

```
(gdb) thread 3
[Switching to thread 3 (core thread 2)]
0x0000001062ffe4e in thread_two (arg=0x0)
at main.c:24
24      *p = 2;
```

```
(gdb) p/x p
$2 = 0x0
```

## Windows

### Kernel Mode

This pattern is about multiple exceptions or faults in kernel mode. Here we distinguish them from **Nested Exceptions** (page 723). The latter ones in the kernel result in double faults like seen in **Stack Overflow** pattern (page 900). For example, at first glance it looks like the dump was saved **Manually** (page 625):

```
0: kd> !analyze -v
[...]
MANUALLY_INITIATED_CRASH (e2)
The user manually initiated this crash dump.

Arguments:
Arg1: 00000000
Arg2: 00000000
Arg3: 00000000
Arg4: 00000000
[...]
```

However, further down in analysis report there is the presence of a page fault:

```
TRAP_FRAME: a38df520 -- (.trap 0xfffffffffa38df520)
ErrCode = 00000002
eax=b6d9220f ebx=b6ab4ffb ecx=00000304 edx=eaf2fdea esi=b6d9214c edi=b6ab8189
eip=bfa10e6e esp=a38df594 ebp=a38df5ac iopl=0 nv up ei ng nz ac po cy
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010293
driver+0x2ae6e:
bfa10e6e 895304    mov     dword ptr [ebx+4],edx ds:0023:b6ab4fff=???????
Resetting default scope

STACK_TEXT:
a38df410 b48aa532 000000e2 00000000 00000000 nt!KeBugCheckEx+0x1b
a38df440 b48a9d2c 000eba28 9282c8c6 00000000 i8042prt!I8xProcessCrashDump+0x256
a38df488 80839595 89d0c008 8a0eb970 0001000a i8042prt!I8042KeyboardInterruptService+0x225
a38df488 80836bfa 89d0c008 8a0eb970 0001000a nt!KiInterruptDispatch+0x49
a38df520 bfa10e6e badb0d00 eaf2fdea 8867cbe8 nt!KiTrap0E+0xbc
WARNING: Stack unwind information not available. Following frames may be wrong.
a38df5ac bfa22461 b6ab423b 000003dc 00000007 driver+0x2ae6e
[...]
```

Looking at the b6ab4fff address shows that it crosses a page boundary, see **Data Alignment** pattern (page 174).

We also see that this thread was running and consumed too much kernel time, see **Spiking Thread** pattern (page 885):

```
0: kd> !thread
THREAD 88e686d8 Cid 1e48.1f7c Peb: 7ffffd000 Win32Thread: b669de70 RUNNING on processor 0
Not impersonating
DeviceMap          dc971120
Owning Process    889e0d88      Image:      ProcessA.EXE
Wait Start TickCount 9231345      Ticks: 0
Context Switch Count 2196221      LargeStack
UserTime           00:00:35.562
KernelTime         04:51:23.656
[...]
```

We see another running thread on the second processor:

```
0: kd> !running
Prcb   Current  Next
0  ffdfff120  88e686d8      .....
1  f7727120  88bd33f8      .....

0: kd> !thread 88bd33f8
THREAD 88bd33f8 Cid 2fdc.27f0 Peb: 7ffffd000 Win32Thread: b6640ab8 RUNNING on processor 1
Not impersonating
DeviceMap          d7a13b40
Owning Process    89e45200      Image:      ProcessA.EXE
Wait Start TickCount 9231345      Ticks: 0
Context Switch Count 2324364      LargeStack
UserTime           00:00:21.171
KernelTime         05:02:09.500
Win32 Start Address ProcessA (0x30001e28)
Start Address kernel32!BaseProcessStartThunk (0x77e617f8)
Stack Init ac5e7bd0 Current ac5e7078 Base ac5e8000 Limit ac5e1000 Call ac5e7bd8
Priority 6 BasePriority 6 PriorityDecrement 0
ChildEBP RetAddr Args to Child
ac5e7150 bfa10e6e badb0d00dbeaffdb 89a793d8 nt!KiTrap0E+0xbc (FPO: [0,0] TrapFrame @ ac5e7150)
WARNING: Stack unwind information not available. Following frames may be wrong.
ac5e71dc bfa22461 b701f15f ffffff24 00000007 driver+0x2ae6e
[...]
```

We see it is spiking CPU too, and we detect a possible loop in the page fault handler:

```
0: kd> .thread 88bd33f8
Implicit thread is now 88bd33f8

0: kd> ~1s

1: kd> r
eax=fffff81c ebx=ac5e71dc ecx=88bd33f8 edx=dbeaffdb esi=b6f81168 edi=b701ffff
eip=80836bfa esp=ac5e7150 ebp=ac5e7150 iopl=0 nv up ei pl nz na pe nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00000206
nt!KiTrap0E+0xbc:
80836bfa 0f84e5010000 je nt!KiTrap0E+0x2a7 (80836de5) [br=0]
```

When looking at the raw stack, we see that the loop happened after processing this exception:

```
1: kd> .trap ac5e7150
ErrCode = 00000002
eax=b6f8122b ebx=b701ffff ecx=fffffe4c edx=dbeafffdb esi=b6f81168 edi=b70201a0
eip=bfa10e6e esp=ac5e71c4 ebp=ac5e71dc iopl=0 nv up ei ng nz ac po cy
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 ef1=00010293
driver+0x2ae6e:
bfa10e6e 895304 mov dword ptr [ebx+4],edx ds:0023:b701ffff=????????
```

The address crosses the page boundary too:

```
1: kd> !pte b701ffff
          VA b701ffff
PDE at    C0300B70      PTE at C02DC07C
contains 642CF863      contains 2F336863
pfn 642cf ---DA--KWEV  pfn 2f336 ---DA--KWEV

1: kd> !pte b701ffff+3
          VA b7020001
PDE at    C0300B70      PTE at C02DC080
contains 642CF863      contains 00000080
pfn 642cf ---DA--KWEV          not valid
                           DemandZero
                           Protect: 4 - ReadWrite
```

This is because trap processing code is found below the current ESP value and also 3rd-party virtual block drivers (we guess) were trying to satisfy the page fault (the latter not shown in the raw stack fragment here):

```
1: kd> dds esp-1000 esp
[...]
ac5e6f78 00000002
ac5e6f7c 899c05b0
ac5e6f80 88bd33f8
ac5e6f84 00000010
ac5e6f88 ac5e702c
ac5e6f8c 808457ff nt!KeContextFromKframes+0x9b
ac5e6f90 00000023
ac5e6f94 f7727120
ac5e6f98 00000000
ac5e6f9c 808458fd nt!KeContextFromKframes+0x2bc
ac5e6fa0 ac5e70dc
ac5e6fa4 1f840a42
ac5e6fa8 00000000
ac5e6fac f7727000
ac5e6fb0 00000000
ac5e6fb4 f7727a7c
ac5e6fb8 ac5e6fd4
ac5e6fbc 808398d4 nt!KiDispatchInterrupt+0xd8
ac5e6fc0 00000000
ac5e6fc4 80a801ae hal!HalpDispatchSoftwareInterrupt+0x5e
ac5e6fc8 ac5e700c
ac5e6fcc ac5e7001
```

ac5e6fd0	00000002
ac5e6fd4	ac5e6ff0
ac5e6fd8	80a80397 hal!HalpCheckForSoftwareInterrupt+0x3f
ac5e6fdc	00000002
ac5e6fe0	ac5e700c
ac5e6fe4	ac5e700c
ac5e6fe8	ac5e70b0
ac5e6fec	00000001
ac5e6ff0	f772f120
ac5e6ff4	88bd33f8
ac5e6ff8	00000002
ac5e6ffc	ac5e700c
ac5e7000	8a0a88a0
ac5e7004	88bd33f8
ac5e7008	f7727002
ac5e700c	80a8057e hal!HalEndSystemInterrupt+0x6e
ac5e7010	88bd33f8
ac5e7014	f7727002
ac5e7018	00000002
ac5e701c	ac5e702c
ac5e7020	80a80456 hal!KfLowerIrql+0x62
ac5e7024	f7727000
ac5e7028	0000bb40
ac5e702c	ac5e70ac
ac5e7030	808093eb nt!KiSaveProcessorState+0x20
ac5e7034	ac5e70dc
ac5e7038	00000000
ac5e703c	808093f0 nt!KiSaveProcessorState+0x25
ac5e7040	f772713c
ac5e7044	8087dcbd nt!KiFreezeTargetExecution+0x6a
ac5e7048	ac5e70dc
ac5e704c	00000000
ac5e7050	f7727120
ac5e7054	00000000
ac5e7058	80a7e501 hal!KeAcquireQueuedSpinLockRaiseToSynch+0x21
ac5e705c	88bd3401
ac5e7060	ac5e7070
ac5e7064	80a80456 hal!KfLowerIrql+0x62
ac5e7068	80a7e530 hal!KeReleaseInStackQueuedSpinLock
ac5e706c	88bd3401
ac5e7070	ac5e70b0
ac5e7074	80a7e56d hal!KeReleaseQueuedSpinLock+0x2d
ac5e7078	80823822 nt!KiDeliverApc+0x1cc
ac5e707c	00000000
ac5e7080	ac806e00
ac5e7084	00000200
ac5e7088	00000000
ac5e708c	88bd343c
ac5e7090	00000001
ac5e7094	ac5e7934
ac5e7098	89e45200
ac5e709c	809282c8 nt!CmpPostApc
ac5e70a0	00000000
ac5e70a4	0000010c
ac5e70a8	1d01f008
ac5e70ac	ac5e70dc

```
ac5e70b0 80837c86 nt!KiIpiServiceRoutine+0x8b
ac5e70b4 ac5e70dc
ac5e70b8 00000000
ac5e70bc 80836bfa nt!KiTrap0E+0xbc
ac5e70c0 b6f81168
ac5e70c4 ac5e7150
ac5e70c8 80a7d8fc hal!HalpIpiHandler+0xcc
ac5e70cc ac5e70dc
ac5e70d0 00000000
ac5e70d4 80a80300 hal!HalpLowerIrqlHardwareInterrupts+0x10c
ac5e70d8 000000e1
ac5e70dc ac5e7150
ac5e70e0 80836bfa nt!KiTrap0E+0xbc
ac5e70e4 badb0d00
ac5e70e8dbeaffdb
ac5e70ec ac5e70fc
ac5e70f0 80a80456 hal!KfLowerIrql+0x62
ac5e70f4 2f336801
ac5e70f8 ac806e00
ac5e70fc ac5e7138
ac5e7100 8081a2bf nt!MmAccessFault+0x558
ac5e7104 b701ffffe
ac5e7108 00000000
ac5e710c 00000000
ac5e7110 00000023
ac5e7114 00000023
ac5e7118dbeaffdb
ac5e711c 88bd33f8
ac5e7120 ffffff81c
ac5e7124 00000000
ac5e7128 ac5e72b0
ac5e712c 00000030
ac5e7130 b701ffffe
ac5e7134 b6f81168
ac5e7138 ac5e71dc
ac5e713c ac5e7150
ac5e7140 00000000
ac5e7144 80836bfa nt!KiTrap0E+0xbc
ac5e7148 00000008
ac5e714c 00000206
ac5e7150 ac5e71dc
```

What we may guess here is the fact that 2 page faults happened simultaneously or nearly at the same time and one of them possibly during the attempt to satisfy the second and this resulted in two processors looping. The whole system was frozen, and the usual keyboard method via Scroll Lock was used to generate **Manual Dump** (page 625).

## Managed Space

In addition to **Multiple Exceptions** in user mode / space (page 714) and kernel mode / space (page 708) patterns we can have multiple exceptions in “managed space”. After SOS extension is loaded we can use the following commands to list such exceptions (some output was skipped for formatting clarity):

```
0:000> !Threads
[...]
ID OSID Exception
0 1 12c System.IO.FileNotFoundException (000000003bd6230)
8 2 e24 (Finalizer)
10 3 c1c System.Reflection.TargetInvocationException (00000000492a388)
11 4 cb0 (Threadpool Completion Port)
12 5 c10
13 6 1e8 (Threadpool Completion Port)
15 7 c14 (Threadpool Worker)
16 8 edc (Threadpool Worker)
[...]
23 e 1084 System.NullReferenceException (00000000492a300)

0:000> ~*e !pe
Exception object: 000000003bd6230
Exception type: System.IO.FileNotFoundException
Message: Could not load file or assembly [...]
InnerException: System.IO.FileNotFoundException, use !PrintException 000000003bd6938 to see more
StackTrace (generated):
SP IP Function
[...]
Exception object: 00000000492a388
Exception type: System.Reflection.TargetInvocationException
Message: Exception has been thrown by the target of an invocation.
InnerException: System.NullReferenceException, use !PrintException 00000000492a300 to see more
StackTrace (generated):
SP IP Function
[...]
```

## Stowed

We do not write a special pattern for individual stowed exceptions (0xC000027B, an application-internal exception) because we consider them examples of **Mixed Exceptions**, page 688 (for example, as **Managed Code Exceptions**, page 617, enveloped by unmanaged code **Software Exception**, page 875, see stowed exception examples<sup>146</sup>). However, we introduce a variant of **Multiple Exceptions** analysis patterns for them because the default analysis WinDbg command (**!analyze -v**) only shows the first exception or error message stored in the array of stowed exception information structures. The following example shows 3 exceptions. When we load the dump we get this indication of the stowed exceptions:

```
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
(2784.2068): Unknown exception - code c000027b (first/second chance not available)
eax=059be538 ebx=00000000 ecx=00000000 edx=00000000 esi=059be898 edi=059be538
eip=750510c0 esp=059be81c ebp=059be940 iopl=0 nv up ei pl nz ac po nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000212
combase!RoFailFastWithErrorContextInternal2+0x109 [inlined in
combase!RoFailFastWithErrorContextInternal+0x240]:
750510c0 6a03 push 3

0:008> !error c000027b
Error code: (NTSTATUS) 0xc000027b (3221226107) - An application-internal exception has occurred.
```

We check **Stored Exception** (page 959):

```
0:008> .exr -1
ExceptionAddress: 66fc2408 (Windows_UI_Xaml!DirectUI::ErrorHelper::ProcessUnhandledError+0x00432e07)
ExceptionCode: c000027b
ExceptionFlags: 00000001
NumberParameters: 2
Parameter[0]: 08d9bfa8
Parameter[1]: 00000003
```

We see it has 3 stowed exceptions pointed to by array elements:

```
0:008> dp 08d9bfa8 L3
08d9bfa8 08d8e084 08d928a8 08d928d0
```

---

<sup>146</sup> <http://blogs.msdn.com/b/ntdebugging/archive/2014/05/28/debugging-a-windows-8-1-store-app-crash-dump-part-2.aspx>

We check what kind of stowed information structures they point to:

```
0:008> dt -a3 08d9bfa8 _STOWED_EXCEPTION_INFORMATION_HEADER*
[0] @ 08d9bfa8
-----
0x08d8e084
+0x000 Size      : 0x28
+0x004 Signature : 0x53453032

[1] @ 08d9bfa8
-----
0x08d928a8
+0x000 Size      : 0x28
+0x004 Signature : 0x53453032

[2] @ 08d9bfb0
-----
0x08d928d0
+0x000 Size      : 0x28
+0x004 Signature : 0x53453032

0:008> .formats 0x53453032
Evaluate expression:
Hex:    53453032
Decimal: 397043250
Octal:  12321230062
Binary: 01010011 01000101 00110000 00110010
Chars:  SE02
Time:   Wed Apr 09 12:34:10 2014
Float:  low 8.46917e+011 high 0
Double: 6.90231e-315
```

Since these are version 2 stowed information structures we use this command:

```
0:008> dt -a3 08d9bfa8 _STOWED_EXCEPTION_INFORMATION_V2*
[0] @ 08d9bfa8
-----
0x08d8e084
+0x000 Header      : _STOWED_EXCEPTION_INFORMATION_HEADER
+0x008 ResultCode  : 802b000a
+0x00c ExceptionForm : 0y01
+0x00c ThreadId   : 0y00000000000000000000100000011010 (0x81a)
+0x010 ExceptionAddress : 0x74f562f2 Void
+0x014 StackTraceWordSize : 4
+0x018 StackTraceWords : 0x3c
+0x01c StackTrace   : 0x07731c08 Void
+0x010 ErrorText   : 0x74f562f2 “趨歸???”
+0x020 NestedExceptionType : 0
+0x024 NestedException : (null)
```

```
[1] @ 08d9bfac
-----
0x08d928a8
+0x000 Header : _STOWED_EXCEPTION_INFORMATION_HEADER
+0x008 ResultCode : 80004001
+0x00c ExceptionForm : 0y01
+0x00c ThreadId : 0y00000000000000000000000000000000 (0)
+0x010 ExceptionAddress : (null)
+0x014 StackTraceWordSize : 4
+0x018 StackTraceWords : 0x3c
+0x01c StackTrace : 0x077d5d4c Void
+0x010 ErrorText : (null)
+0x020 NestedExceptionType : 0
+0x024 NestedException : (null)

[2] @ 08d9bfb0
-----
0x08d928d0
+0x000 Header : _STOWED_EXCEPTION_INFORMATION_HEADER
+0x008 ResultCode : 80004005
+0x00c ExceptionForm : 0y01
+0x00c ThreadId : 0y00000000000000000000000000000000 (0)
+0x010 ExceptionAddress : (null)
+0x014 StackTraceWordSize : 4
+0x018 StackTraceWords : 0x19
+0x01c StackTrace : 0x0772df74 Void
+0x010 ErrorText : (null)
+0x020 NestedExceptionType : 0
+0x024 NestedException : (null)
```

Now we check **Database Stack Traces** (page 919) for individual entries and their error codes:

```
0:008> !error 802b000a
Error code: (HRESULT) 0x802b000a (2150301706) - <Unable to get error code text>

0:008> .lines -d
Line number information will not be loaded

0:008> dps 0x07731c08 L3c
07731c08 74fba2be combbase!RoOriginateErrorW+0x3e
07731c0c 66b900d7 Windows_UI_Xaml!DirectUI::ErrorHelper::OriginateError+0x8d
07731c10 66b8fe5f Windows_UI_Xaml!CJupiterErrorServiceListener::NotifyErrorAdded+0xaf
07731c14 66b8fd60 Windows_UI_Xaml!CErrorService::AddError+0x130
07731c18 66b8f0f8 Windows_UI_Xaml!CErrorService::ReportParserError+0x88
07731c1c 66b8f000 Windows_UI_Xaml!ParserErrorService::ReportError+0xd0
07731c20 670f9654 Windows_UI_Xaml!ParserErrorReporter::SetError+0x61
07731c24 670fbe70 Windows_UI_Xaml!XamlBinaryMetadataReader2::LogError+0x6c
07731c28 670fdb37 Windows_UI_Xaml!<lambda_1d4791754290213f1bf5bc456a504cc5>::operator() +0x41
07731c2c 67032601 Windows_UI_Xaml!XamlBinaryMetadataReader2::LoadProperty+0x1f7e82
07731c30 66cab9d9 Windows_UI_Xaml!XamlBinaryMetadataReader2::GetProperty+0x5c
07731c34 66d528e0 Windows_UI_Xaml!XamlBinaryFormatSubReader2::ReadXamlProperty+0x1d0
07731c38 66d520cc Windows_UI_Xaml!XamlBinaryFormatSubReader2::ReadSetValueConstantNode+0x3c
07731c3c 66d53b55 Windows_UI_Xaml!XamlBinaryFormatSubReader2::TryRead+0x135
07731c40 66d539ec Windows_UI_Xaml!XamlBinaryFormatSubReader2::TryReadHRESULT+0x3c
07731c44 66ca329e Windows_UI_Xaml!CParser::LoadXamlCore+0x5ae
07731c48 66da83d2 Windows_UI_Xaml!CCoreServices::ParseXamlWithExistingFrameworkRoot+0x100
07731c4c 66da8131 Windows_UI_Xaml!Application::LoadComponent+0x261
07731c50 66da7e56 Windows_UI_Xaml!Application_LoadComponent+0xa
07731c54 66da7fcf Windows_UI_Xaml!DirectUI::FrameworkApplication::LoadComponent+0xc2
07731c58 66dab49d Windows_UI_Xaml!DirectUI::FrameworkApplicationFactory:: LoadComponentWithResourceLocationImpl+0x6a
```

```

07731c5c 66dab418 Windows_UI_Xaml!DirectUI::FrameworkApplicationFactory:: LoadComponentWithResourceLocation+0x28
07731c60 6c186b95*** WARNING: Unable to verify checksum for Microsoft.Msn.Weather.dll
*** ERROR: Symbol file could not be found. Defaulted to export symbols for Microsoft.Msn.Weather.dll -
Microsoft_Msn_Weather_6bb10000!RHBinder_ShimExeMain+0x1b2515
07731c64 6c186af1 Microsoft_Msn_Weather_6bb10000!RHBinder_ShimExeMain+0x1b2471
07731c68 6c199d37 Microsoft_Msn_Weather_6bb10000!RHBinder_ShimExeMain+0x1c56b7
07731c6c 6c366cd8 Microsoft_Msn_Weather_6bb10000!DllGetActivationFactory+0xa0f038
07731c70 6c366c8d Microsoft_Msn_Weather_6bb10000!DllGetActivationFactory+0xaefed
07731c74 6c366b4d Microsoft_Msn_Weather_6bb10000!DllGetActivationFactory+0xaeaed
07731c78 6c365b50 Microsoft_Msn_Weather_6bb10000!DllGetActivationFactory+0xaddeb0
07731c7c 6c3cc1d7 Microsoft_Msn_Weather_6bb10000!DllGetActivationFactory+0x114537
07731c80 6c3ccb9a Microsoft_Msn_Weather_6bb10000!DllGetActivationFactory+0x113efa
07731c84 6c3cb9c0 Microsoft_Msn_Weather_6bb10000!DllGetActivationFactory+0x113d20
07731c88 6c35a658 Microsoft_Msn_Weather_6bb10000!DllGetActivationFactory+0xa29b8
07731c8c 6c0fd56e Microsoft_Msn_Weather_6bb10000!RHBinder_ShimExeMain+0x128eee
07731c90 6c21e3d5 Microsoft_Msn_Weather_6bb10000!RHBinder_ShimExeMain+0x249d55
07731c94 66e50d37 Windows_UI_Xaml!DirectUI::FrameworkApplicationGenerated::OnLaunchedProtected+0x47
07731c98 66e509d0 Windows_UI_Xaml!DirectUI::FrameworkView::OnActivated+0x240
07731c9c 66cf4b28 Windows_UI_Xaml!Microsoft::WRL::Callback< Windows::Foundation::ITypedEventHandler< Windows::UI::Core::CoreWindow
*, IInspectable *>, DirectUI::Page, Windows::UI::Core::ICoreWindow *, IInspectable *>::`2`::ComObject::Invoke+0x28
07731ca0 7035361e twinapi_appcore!Microsoft::WRL::InvokeTraits<-2>::InvokeDelegates< <lambda_45dc3980e5a5ff53a9eb289d8a61b7e3>,
Windows::Foundation::ITypedEventHandler< Windows::ApplicationModel::Core::CoreApplicationView
*, Windows::ApplicationModel::Activation::IActivatedEventArgs *> >+0x4f
07731ca4 70353493 twinapi_appcore!Microsoft::WRL::EventSource< Windows::Foundation::ITypedEventHandler<
Windows::ApplicationModel::Core::CoreApplicationView *, Windows::ApplicationModel::Activation::IActivatedEventArgs
*>, Microsoft::WRL::InvokeModeOptions<-2> >::DoInvoke< <lambda_45dc3980e5a5ff53a9eb289d8a61b7e3> >+0x42
07731ca8 70342a06 twinapi_appcore!Windows::ApplicationModel::Core::CoreApplicationView::Activate+0x296
07731cbc 77425ebc rpcrt4!Invoke+0x34
07731cb0 773f37e3 rpcrt4!NdrStubCall2+0x2e3
07731cbc 74efc1ce combbase!CStdStubBuffer_Invoke+0xde
07731cb8 7742364c rpcrt4!CStdStubBuffer_Invoke+0x2c
07731cbc 74fb659b combbase!ObjectMethodExceptionHandlingAction< <lambda_adf5d6ba83bff890864fd80ca2bbf1eb> >+0x7b
07731cc0 74f83091 combbase!DefaultStubInvoke+0x211
07731cc4 74f8d59e combbase!ServerCall::ContextInvoke+0x38e
07731cc8 74f8ecc5 combbase!AppInvoke+0xb75
07731ccc 74f81c8c combbase!ComInvokeWithLockAndIPID+0x62c
07731cd0 74f66d72 combbase!CComApartment::ASTAHandleMessage+0x2c2
07731cd4 74f65bfa combbase!ASTAWaitContext::Wait+0x47a
07731cd8 74fbff1b4 combbase!ASTAWaitInNewContext+0x81
07731cdc 74fbff0ee combbase!ASTATHreadWaitForHandles+0x4e
07731ce0 74fbaf1a combbase!CoWaitForMultipleHandles+0xaa
07731ce4 7035772d twinapi_appcore!CTSimpleArray<COSTaskCompletion::TaskContext *, 4294967294,
CTPolicyCoTaskMem<COSTaskCompletion::TaskContext *>, CSimpleArrayStandardCompareHelper<COSTaskCompletion::TaskContext
*>, CSimpleArrayStandardMergeHelper<COSTaskCompletion::TaskContext *> >::RemoveAt+0x9f
07731ce8 747d3bea SHCore!CTSimpleArray<Microsoft::WRL::ComPtr< IWindowMonitorChangeListener >, 4294967294,
CTPolicyCoTaskMem<Microsoft::WRL::ComPtr< IWindowMonitorChangeListener > >, CSimpleArrayStandardCompareHelper<Microsoft::WRL::ComPtr<
IWindowMonitorChangeListener > >, CSimpleArrayStandardMergeHelper<Microsoft::WRL::ComPtr< IWindowMonitorChangeListener > >
>::Add<Microsoft::WRL::ComPtr< IWindowMonitorChangeListener > const &>+0x120
07731cec 75423744 kernel32!BaseThreadInitThunk+0x24
07731cf0 776d9e54 ntdll!_RtlUserThreadStart+0x2f
07731cf4 776d9e1f ntdll!_RtlUserThreadStart+0x1b

```

0:008> !error 80004001

Error code: (HRESULT) 0x80004001 (2147500033) - Not implemented

```

0:008> dps 0x077d5d4c L3c
077d5d4c 66f42bbe Windows_UI_Xaml!DirectUI::MetadataAPI::ImportClassInfo+0x2ed89e
077d5d50 66e451ce Windows_UI_Xaml!DirectUI::MetadataAPI::ImportClassInfoFromMetadataProvider+0x86
077d5d54 66c87bf9 Windows_UI_Xaml!DirectUI::MetadataAPI::GetClassInfoByTypeName+0x219
077d5d58 66c55ea0 Windows_UI_Xaml!DirectUI::MetadataAPI::GetClassInfoByXamlType+0x50
077d5d5c 66c5365d Windows_UI_Xaml!DirectUI::MetadataAPI::ImportPropertyInfo+0x8a
077d5d60 66c542a0 Windows_UI_Xaml!DirectUI::MetadataAPI::TryImportPropertyInfoFromMetadataProvider+0x9f
077d5d64 66c53f94 Windows_UI_Xaml!DirectUI::MetadataAPI::TryGetPropertyByName+0xde
077d5d68 66c564e7 Windows_UI_Xaml!XamlManagedTypeInfoProvider::ResolvePropertyName+0x47
077d5d6c 66c930c2 Windows_UI_Xaml!XamlType::GetProperty+0x1a2
077d5d70 66e3a830 Windows_UI_Xaml!XamlBinaryDataReader2::LoadProperty+0xb1
077d5d74 66cab9d9 Windows_UI_Xaml!XamlBinaryDataReader2::GetProperty+0x5c
077d5d78 66d528e0 Windows_UI_Xaml!XamlBinaryFormatSubReader2::ReadXamlProperty+0x1d0
077d5d7c 66d520cc Windows_UI_Xaml!XamlBinaryFormatSubReader2::ReadSetValueConstantNode+0x3c
077d5d80 66d53b55 Windows_UI_Xaml!XamlBinaryFormatSubReader2::TryRead+0x135
077d5d84 66d539ec Windows_UI_Xaml!XamlBinaryFormatSubReader2::TryReadHRESULT+0x3c
077d5d88 66ca329e Windows_UI_Xaml!CParse::LoadXamlCore+0x5ae
077d5d8c 66da83d2 Windows_UI_Xaml!CCoreServices::ParseXamlWithExistingFrameworkRoot+0x100

```

```

077d5d90 66da8131 Windows_UI_Xaml!CApplication::LoadComponent+0x261
077d5d94 66da7e56 Windows_UI_Xaml!Application_LoadComponent+0xa
077d5d98 66da7cfc Windows_UI_Xaml!DirectUI::FrameworkApplication::LoadComponent+0xc2
077d5d9c 66dab49d Windows_UI_Xaml!DirectUI::FrameworkApplicationFactory:: LoadComponentWithResourceLocationImpl+0x6a
077d5d9d 66dab418 Windows_UI_Xaml!DirectUI::FrameworkApplicationFactory:: LoadComponentWithResourceLocation+0x28
077d5d4a 6c186b95 Microsoft_Msn_Weather_6bb10000!RHBinder_ShimExeMain+0x1b2515
077d5d48 6c186af1 Microsoft_Msn_Weather_6bb10000!RHBinder_ShimExeMain+0x1b2471
077d5dac 6c199d37 Microsoft_Msn_Weather_6bb10000!RHBinder_ShimExeMain+0x1c56b7
077d5db0 6c366cd8 Microsoft_Msn_Weather_6bb10000!DllGetActivationFactory+0xaf038
077d5db4 6c366c8d Microsoft_Msn_Weather_6bb10000!DllGetActivationFactory+0xaeafed
077d5db8 6c366b4d Microsoft_Msn_Weather_6bb10000!DllGetActivationFactory+0xaeaed
077d5dbc 6c365b50 Microsoft_Msn_Weather_6bb10000!DllGetActivationFactory+0xadef0
077d5dc0 6c3cc1d7 Microsoft_Msn_Weather_6bb10000!DllGetActivationFactory+0x114537
077d5dc4 6c3cb9a9 Microsoft_Msn_Weather_6bb10000!DllGetActivationFactory+0x113efa
077d5dc8 6c3cb9c0 Microsoft_Msn_Weather_6bb10000!DllGetActivationFactory+0x113d20
077d5dcc 6c35a658 Microsoft_Msn_Weather_6bb10000!DllGetActivationFactory+0xa29b8
077d5dd0 6c0fd56e Microsoft_Msn_Weather_6bb10000!RHBinder_ShimExeMain+0x128eee
077d5dd4 6c21e3d5 Microsoft_Msn_Weather_6bb10000!RHBinder_ShimExeMain+0x249d55
077d5dd8 66e50d37 Windows_UI_Xaml!DirectUI::FrameworkApplicationGenerated::OnLaunchedProtected+0x47
077d5ddc 66e509d0 Windows_UI_Xaml!DirectUI::FrameworkView::OnActivated+0x240
077d5de0 66cf4b28 Windows_UI_Xaml!Microsoft::WRL::Callback< Windows::Foundation::ITypedEventHandler< Windows::UI::Core::CoreWindow
*, IInspectable *>, DirectUI::Page, Windows::UI::Core::ICoreWindow *, IInspectable *>`::`2`::ComObject::Invoke+0x28
077d5de4 7035361e twinapi_appcore!Microsoft::WRL::InvokeTraits<-2>::InvokeDelegates< <lambda_45dc3980e5a5ff53a9eb289d8a61b7e3>,
Windows::Foundation::ITypedEventHandler< Windows::ApplicationModel::Core::CoreApplicationView
*, Windows::ApplicationModel::Activation::IActivatedEventArgs *> >+0x4f
077d5de8 70353493 twinapi_appcore!Microsoft::WRL::EventSource< Windows::Foundation::ITypedEventHandler<
Windows::ApplicationModel::Core::CoreApplicationView*, Windows::ApplicationModel::Activation::IActivatedEventArgs
*>, Microsoft::WRL::InvokeModeOptions<-2> ::DoInvoke< <lambda_45dc3980e5a5ff53a9eb289d8a61b7e3> >+0x42
077d5dec 70342a06 twinapi_appcore!Windows::ApplicationModel::Core::CoreApplicationView::Activate+0x296
077d5df0 77425ebc rpcrt4!Invoke+0x34
077d5df4 773f37e3 rpcrt4!NdrStubCall2+0x2e3
077d5df8 74efc1ce combase!CStdStubBuffer_Invoke+0xde
077d5dfc 7742364c rpcrt4!CStdStubBuffer_Invoke+0x2c
077d5e00 74fb659b combase!ObjectMethodXceptionHandlingAction< <lambda_adf5d6ba83bfff890864fd80ca2bbf1eb> >+0x7b
077d5e04 74f83091 combase!DefaultStubInvoke+0x211
077d5e08 74f8d59e combase!ServerCall::ContextInvoke+0x38e
077d5e0c 74f8ecc5 combase!AppInvoke+0xb75
077d5e10 74f81c8c combase!ComInvokeWithLockAndIPID+0x62c
077d5e14 74f66d72 combase!CComApartment::ASTAHandleMessage+0x2c2
077d5e18 74f65bfa combase!ASTAwaitContext::Wait+0x47a
077d5e1c 74fbff1b4 combase!ASTAwaitInNewContext+0x81
077d5e20 74fbff0ee combase!ASTAThreadWaitForHandles+0x4e
077d5e24 74fbaf1a combase!CoWaitForMultipleHandles+0xaa
077d5e28 7035772d twinapi_appcore!CTSimpleArray<COSTaskCompletion::TaskContext *, 4294967294,
CTPolicyCoTaskMem<COSTaskCompletion::TaskContext *>, CSimpleArrayStandardCompareHelper<COSTaskCompletion::TaskContext
*>, CSimpleArrayStandardMergeHelper<COSTaskCompletion::TaskContext *> >; RemoveAt+0x9f
077d5e2c 747d3bea SHCore!CTSimpleArray<Microsoft::WRL::ComPtr< IWindowMonitorChangeListener >, 4294967294,
CTPolicyCoTaskMem<Microsoft::WRL::ComPtr< IWindowMonitorChangeListener > >, CSimpleArrayStandardCompareHelper<Microsoft::WRL::ComPtr<
IWindowMonitorChangeListener >, CSimpleArrayStandardMergeHelper<Microsoft::WRL::ComPtr< IWindowMonitorChangeListener > >
>; _Add<Microsoft::WRL::ComPtr< IWindowMonitorChangeListener > const &>+0x120
077d5e30 75423744 kernel32!BaseThreadInitThunk+0x24
077d5e34 776d9e54 ntdll!_RtlUserThreadStart+0x2f
077d5e38 776d9e1f ntdll!_RtlUserThreadStart+0x1b

```

0:008> ub **66f42bbe**

```
Windows_UI_Xaml!DirectUI::MetadataAPI::ImportClassInfo+0x2ed87f:
```

```

66f42b9f 8bce          mov    ecx,esi
66f42ba1 e8b96ecfff   call   Windows_UI_Xaml!OnFailure<1354> (66c39a5f)
66f42ba6 e92d2bd1ff   jmp   Windows_UI_Xaml!DirectUI::MetadataAPI:: ImportClassInfo+0x3b8 (66c556d8)
66f42bab 8bce          mov    ecx,esi
66f42bad e8ad6ecfff   call   Windows_UI_Xaml!OnFailure<1354> (66c39a5f)
66f42bb2 e9212bd1ff   jmp   Windows_UI_Xaml!DirectUI::MetadataAPI:: ImportClassInfo+0x3b8 (66c556d8)
66f42bb7 8bce          mov    ecx,esi
66f42bb9 e8a16ecfff   call   Windows_UI_Xaml!OnFailure<1354> (66c39a5f)

```

0:008> !error **80004005**

Error code: (HRESULT) 0x80004005 (214750037) - Unspecified error

```

0:008> dps 0x0772df74 L19
0772df74 66bc3963 Windows_UI_Xaml!CWindowsServices::CreatePalAppChromeProxy+0x23
0772df78 66bc3912 Windows_UI_Xaml!AppChromeProxy::EnsureAppChromeProxy+0x44
0772df7c 66bc38b6 Windows_UI_Xaml!AppChromeProxy::SetThemeOverride+0xf
0772df80 66e51074 Windows_UI_Xaml!DirectUI::DXamlCore::ConfigureCoreWindow+0x95
0772df84 66e50e6f Windows_UI_Xaml!DirectUI::FrameworkView::SetWindow+0x1f
0772df88 70342def twinapi_appcore!Windows::ApplicationModel::Core::CoreApplicationView:: SetWindowAndGetDispatcher + 0x20f
0772df8c 77425ebc rpcrt4!Invoke+0x34
0772df90 773f37e3 rpcrt4!NdrStubCall12+0x2e3
0772df94 74fc1ce combbase!CStdStubBuffer_Invoke+0xde
0772df98 7742364c rpcrt4!CStdStubBuffer_Invoke+0x2c
0772df9c 74fb659b combbase!ObjectMethodExceptionHandlingAction< <lambda_adf5d6ba83bfff890864fd80ca2bbf1eb> >+0x7b
0772dfa0 74f83091 combbase!DefaultStubInvoke+0x211
0772dfa4 74f8d59e combbase!ServerCall::ContextInvoke+0x38e
0772dfa8 74f8ecc5 combbase!AppInvoke+0xb75
0772dfac 74f81c8c combbase!ComInvokeWithLockAndIPID+0x62c
0772dfb0 74f66d72 combbase!CComApartment::ASTAHandleMessage+0x2c2
0772dfb4 74f65bfa combbase!ASTAWaitContext::Wait+0x47a
0772dfb8 74fb1b4 combbase!ASTAWaitInNewContext+0x81
0772dfbc 74fb0ee0 combbase!ASTAThreadWaitForHandles+0x4e
0772dfc0 74fbaf1a combbase!CoWaitForMultipleHandles+0xaa
0772dfc4 7035772d twinapi_appcore!CTSSimpleArray<COSTaskCompletion::TaskContext *,4294967294,
CTPolicyCoTaskMem<COSTaskCompletion::TaskContext *>,CSimpleArrayStandardCompareHelper<COSTaskCompletion::TaskContext
*>,CSimpleArrayStandardMergeHelper<COSTaskCompletion::TaskContext *> >::RemoveAt+0x9f
0772dfc8 747d3bea SHCore!CTSSimpleArray<Microsoft::WRL::ComPtr< IWindowMonitorChangeListener>,4294967294,
CTPolicyCoTaskMem<Microsoft::WRL::ComPtr< IWindowMonitorChangeListener> >,CSimpleArrayStandardCompareHelper<Microsoft::WRL::ComPtr<
IWindowMonitorChangeListener >,CSimpleArrayStandardMergeHelper<Microsoft::WRL::ComPtr< IWindowMonitorChangeListener > >
>::_Add<Microsoft::WRL::ComPtr< IWindowMonitorChangeListener > const &>+0x120
0772dfcc 75423744 kernel32!BaseThreadInitThunk+0x24
0772dfd0 776d9e54 ntdll!_RtlUserThreadStart+0x2f
0772dfd4 776d9e1f ntdll!_RtlUserThreadStart+0x1b

0:008> ub 66bc3963
Windows_UI_Xaml!CWindowsServices::CreatePalAppChromeProxy+0x5:
66bc3945 56          push    esi
66bc3946 b87508      mov     esi,dword ptr [ebp+8]
66bc3949 832600      and    dword ptr [esi],0
66bc394c e83ebd2d00  call    Windows_UI_Xaml!IsConfigureStatusBarDefaultsPresent (66e9f68f)
66bc3951 84c0          test   al,al
66bc3953 0f856c1a4100 jne    Windows_UI_Xaml!CWindowsServices:: CreatePalAppChromeProxy+0x411a85 (66fd53c5)
66bc3959 be05400080    mov    esi,80004005h
66bc395e e80c000000  call   Windows_UI_Xaml!OnFailure<91> (66bc396f)

```

## User Mode

**Multiple Exceptions** captures the known fact that there could be as many exceptions ("crashes") as many threads in a process. The following UML diagram depicts the relationship between Process, Thread, and Exception entities:



Every process in Windows has at least one execution thread so there could be at least one exception per thread (like invalid memory reference) if things go wrong. There could be a second exception in that thread if exception handling code experiences another exception or the first exception was handled.

The general solution is to look at all threads and their stacks and do not rely on what tools say.

Here is a concrete example from one of the dumps. Internet Explorer crashed, and we opened it in WinDbg and ran **!analyze -v** command. This is what we got in WinDbg output:

```

ExceptionAddress: 7c822583 (ntdll!DbgBreakPoint)
ExceptionCode: 80000003 (Break instruction exception)
ExceptionFlags: 00000000
NumberParameters: 3
Parameter[0]: 00000000
Parameter[1]: 8fb834b8
Parameter[2]: 00000003
  
```

Break instruction, we might think, shows that the dump was taken manually from the running application, and there was no crash - the customer sent the wrong dump or misunderstood troubleshooting instructions. However, we looked at all threads, and noticed the following two stacks (threads 15 and 16):

```

0:016>~*kL
...
15  Id: 1734.8f4 Suspend: 1 Teb: 7ffab000 Unfrozen
ntdll!KiFastSystemCallRet
ntdll!NtRaiseHardError+0xc
kernel32!UnhandledExceptionFilter+0x54b
kernel32!BaseThreadStart+0x4a
kernel32!_except_handler3+0x61
ntdll!ExecuteHandler2+0x26
ntdll!ExecuteHandler+0x24
ntdll!KiUserExceptionDispatcher+0xe
componentA!xxx
componentB!xxx
mshtml!xxx
kernel32!BaseThreadStart+0x34

# 16  Id: 1734.11a4 Suspend: 1 Teb: 7ffaa000 Unfrozen
ntdll!DbgBreakPoint
ntdll!DbgUiRemoteBreakin+0x36
  
```

We see here that the real crash happened in *componentA.dll* and *componentB.dll* or *mshtml.dll* might have influenced that. Why did this happen? The customer might have dumped Internet Explorer manually while it was displaying an exception message box. *NtRaiseHardError* displays a message box containing an error message.

Perhaps something else happened. Many cases, where we see multiple thread exceptions in one process dump, happened because crashed threads displayed message boxes like Visual C++ debug message box and preventing that process from termination. In our dump under discussion, WinDbg automatic analysis command recognized only the last breakpoint exception (shown as # 16). In conclusion, we shouldn't rely on "automatic analysis" often anyway.

**N**

## Namespace

As usual, a new pattern arises with the need to communicate analysis findings. Most often, when analyzing malware, we don't have symbol files (**No Component Symbols**, page 734) for **Unknown Module** (page 1041). By looking at IAT (Import Address Table, if any present), we can guess the module purpose. Sometimes a module itself is not malicious but is used in a larger malicious context such as screen grabbing:

```
[...]
10002000 76376101 gdi32!CreateCompatibleDC
10002004 763793d6 gdi32!StretchBlt
10002008 76377461 gdi32!CreateDIBSection
1000200c 763762a0 gdi32!SelectObject
10002010 00000000
10002024 77429ced user32!ReleaseDC
10002028 77423ba7 user32!NtUserGetWindowDC
1000202c 77430e21 user32!GetWindowRect
10002030 00000000
10002034 744a75e9 GdiPlus!GdiplusStartup
10002038 744976dd GdiPlus!GdipSaveImageToStream
1000203c 744cd38 GdiPlus!GdipGetImageEncodersSize
10002040 744971cf GdiPlus!GdipDisposeImage
10002044 744a8591 GdiPlus!GdipCreateBitmapFromHBITMAP
10002048 744cdbae GdiPlus!GdipGetImageEncoders
[...]
```

There are also cases where these API names are not in IAT but found as **String Hint** (page 960) in raw data such *LoadLibrary* / *GetProcAddress* and even a group of modules themselves as a collective API:

```
[...]
00058e20 "kernel32.dll"
00058e3c "user32.dll"
00058e54 "ws2_32.dll"
00058e6c "ntdll.dll"
00058e80 "wininet.dll"
00058e98 "nspr4.dll"
00058eac "ss13.dll"
[...]
```

## Nested Exceptions

### Managed Code

Nested exception analysis is much simpler in managed code than in unmanaged (page 726). Exception object references the inner exception if there is any (*Exception.InnerException*<sup>147</sup>).

WinDbg does a good job of traversing all nested exceptions when executing **!analyze -v** command. In the following example of a Windows forms application crash, *ObjectDisposedException* (shown in bold) was re-thrown as *Exception* object with “Critical error” message (shown in bold italics) which was re-thrown several times as *Exception* object with “Critical program error” message (shown in bold underlined) that finally resulted in the process termination request from the top level exception handler:

```
MANAGED_STACK:
(TransitionMU)
001374B0 0B313757 System!System.Diagnostics.Process.Kill() +0x37
001374E4 0B3129C7 Component!Foo.HandleUnhandledException(System.Exception) +0x137
001374F4 07C0A7D3 Component!Foo+FooBarProcessMenuCommand(System.String) +0x33
(TransitionUM)
(TransitionMU)
...
EXCEPTION OBJECT: !pe 3398614
Exception object: 03398614
Exception type: System.Exception
Message: Critical program error
InnerException: System.Exception, use !PrintException 03398560 to see more StackTrace (generated):
    SP      IP      Function
    001371A8 07C0CDD8 Foo.BarUserInteraction.Process(System.String)
    00137258 07C0CCA6 Foo.BarUserInteraction.ProcessUserInteraction(Sub, BarStepType)
    00137268 07C0A9BA Foo.BarMenu.Process(CMD)
    00137544 07C0A8D8 Foo.BarMenu.ProcessCMD(CMD)
    0013756C 07C0A7BE Foo+FooBar.ProcessBarMenuCommand(System.String)

StackTraceString: <none>
HRESULT: 80131500

EXCEPTION OBJECT: !pe 3398560
Exception object: 03398560
Exception type: System.Exception
Message: Critical program error
InnerException: System.Exception, use !PrintException 033984ac to see more StackTrace (generated):
    SP      IP      Function
    00137154 07C0D4CA Foo.BarThreads+ProcessOpenQuery.Execute()
    00137218 07C0D3B3 Foo.BarMenu.ProcessQuery()
    00137220 07C0CCF3 Foo.BarUserInteraction.Process(System.String)
```

<sup>147</sup> <http://msdn.microsoft.com/en-us/library/system.exception.innerexception.aspx>

```

StackTraceString: <none>
HResult: 80131500

EXCEPTION OBJECT: !pe 33984ac
Exception object: 033984ac
Exception type: System.Exception
Message: Critical program error
InnerException: System.Exception, use !PrintException 033983ec to see more StackTrace (generated):
    SP      IP      Function
    0013704C 0A6149DD Foo.Bar.OpenQueryThreaded(Foo.BarParameter)
    00137154 0A6140D0 Foo.BarThreads+ProcessParameter.Execute()
    ...

StackTraceString: <none>
HResult: 80131500

EXCEPTION OBJECT: !pe 33983ec
Exception object: 033983ec
Exception type: System.Exception
Message: Critical program error
InnerException: System.Exception, use !PrintException 033982fc to see more StackTrace (generated):
    SP      IP      Function
    00137008 0ACA59F1 Foo.BarApplication.Refresh(Boolean, Boolean)
    001370C4 0A6144E0 Foo.Bar.OpenQueryThreaded(Foo.BarParameter)

StackTraceString: <none>
HResult: 80131500

EXCEPTION OBJECT: !pe 33982fc
Exception object: 033982fc
Exception type: System.Exception
Message: Critical program error
InnerException: System.Exception, use !PrintException 03398260 to see more StackTrace (generated):
    SP      IP      Function
    00136F3C 0AE24983 Foo.BarDisplay.ShowVariableScreen(Foo.variables.BarVariables)
    00136FDC 0AE204F6 Foo.variables.BarVariables.ShowVariableScreen()
    00137070 0ACAFE1D Foo.BarApplication.ShowVariableScreen(Boolean)
    00137080 0ACA5977 Foo.BarApplication.Refresh(Boolean, Boolean)

StackTraceString: <none>
HResult: 80131500

EXCEPTION_OBJECT: !pe 3398260
Exception object: 03398260
Exception type: System.Exception
Message: Critical error
InnerException: System.ObjectDisposedException, use !PrintException 03397db8 to see more StackTrace (generated):
    SP      IP      Function
    00136FB4 0AE24905 Foo.BarDisplay.ShowVariableScreen(Foo.variables.BarVariables)

StackTraceString: <none>
HResult: 80131500

EXCEPTION_OBJECT: !pe 3397db8
Exception object: 03397db8
Exception type: System.ObjectDisposedException

```

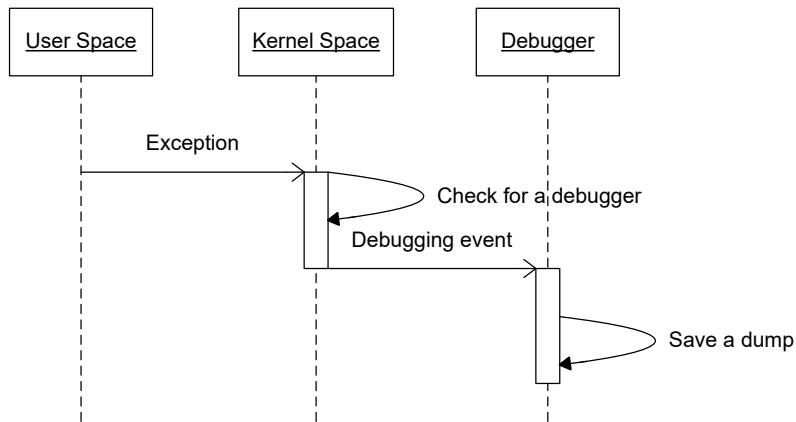
```
Message: Cannot access a disposed object.  
InnerException: <none>  
StackTrace (generated):  
    SP      IP      Function  
00136258 06D36158 System.Windows.Forms.Control.CreateHandle()  
001362B8 06D38F96 System.Windows.Forms.Control.get_Handle()  
001362C4 0B0C8C68 System.Windows.Forms.Control.PointToScreen(System.Drawing.Point)  
001362F0 0B0CECB4 System.Windows.Forms.Button.OnMouseUp(System.Windows.Forms.MouseEventArgs)  
00136314 0B0C8BB7 System.Windows.Forms.Control.WmMouseUp(System.Windows.Forms.Message ByRef,  
System.Windows.Forms.MouseButtons, Int32)  
00136384 06D385A0 System.Windows.Forms.Control.WndProc(System.Windows.Forms.Message ByRef)  
001363E8 0A69C73E System.Windows.Forms.ButtonBase.WndProc(System.Windows.Forms.Message ByRef)  
00136424 0A69C54D System.Windows.Forms.Button.WndProc(System.Windows.Forms.Message ByRef)  
0013642C 06D37FAD  
System.Windows.Forms.Control+ControlNativeWindow.OnMessage(System.Windows.Forms.Message ByRef)  
00136430 06D37F87  
System.Windows.Forms.Control+ControlNativeWindow.WndProc(System.Windows.Forms.Message ByRef)  
00136440 06D37D9F System.Windows.Forms.NativeWindow.Callback(IntPtr, Int32, IntPtr, IntPtr)  
  
StackTraceString: <none>  
HRESULT: 80131622  
  
EXCEPTION_MESSAGE: Cannot access a disposed object.  
  
STACK_TEXT:  
00137448 7c827c1b ntdll!KiFastSystemCallRet  
0013744c 77e4201b ntdll!NtTerminateProcess+0xc  
0013745c 05d78202 kernel32!TerminateProcess+0x20  
...
```

## Comments

For **Distributed Exception** (page 243) objects we may also need to check `_remoteStackTraceString` field.

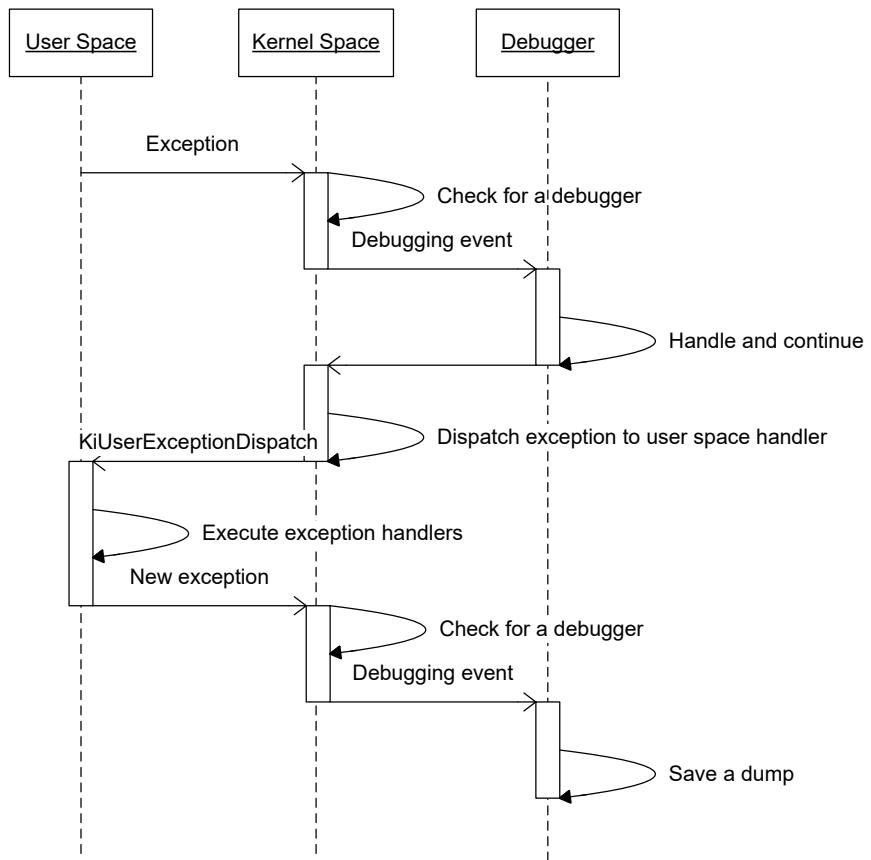
## Unmanaged Code

In the case of a first-chance exception it is not possible to see it in a process crash dump because the entire exception processing was done in the kernel space<sup>148</sup>:



However, the picture changes when we have nested exceptions. In this case, we should expect traces of inner exception processing like exception dispatcher code or exception handlers to be present on a raw stack dump:

<sup>148</sup> How to Distinguish Between 1st and 2nd Chances, Memory Dump Analysis Anthology, Volume 1, page 109



Consider the following C++ code with two exception handlers:

```

__try
{
__try
{
*(int *)NULL = 0; // Exception 1
// Dump1 1st chance
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
    std::cout << "Inner" << std::endl;
*(int *)NULL = 0; // Exception 2
// Dump2 1st chance
}
}
  
```

```
_except (EXCEPTION_EXECUTE_HANDLER)
{
    std::cout << "Outer" << std::endl;
    *(int *)NULL = 0;      // Exception 3
                           // Dump3 1st chance
                           // Dump4 2nd chance
}
```

If we run the actual program, and we have set a default postmortem debugger<sup>149</sup>, we get a second-chance exception dump (Dump4). The program first outputs “Inner” and then “Outer” on a console and then crashes. When we look at the dump we see second-chance exception processing code where the exception record for *NtRaiseException* is the same and points to Exception 3 context (shown in bold underlined):

```
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
(11dc.f94): Access violation - code c0000005 (first/second chance not available)
*** ERROR: Module load completed but symbols could not be loaded for NestedException.exe
NestedException+0x1a72:
00000001`40001a72 c70425000000000000000000 mov dword ptr [0],0 ds:00000000`00000000=????????
0:000> !teb
TEB at 000007fffffde000
ExceptionList: 0000000000000000
StackBase: 0000000000130000
StackLimit: 000000000012d000
SubSystemTib: 0000000000000000
FiberData: 000000000001e00
ArbitraryUserPointer: 0000000000000000
Self: 000007fffffde000
EnvironmentPointer: 0000000000000000
ClientId: 00000000000011dc . 000000000000f94
RpcHandle: 0000000000000000
Tls Storage: 000007fffffde058
PEB Address: 000007fffffd5000
LastErrorValue: 0
LastStatusValue: c000000d
Count Owned Locks: 0
HardErrorMode: 0

0:000> dqs 000000000012d000 0000000000130000
...
00000000`0012f918 00000000`00000006
00000000`0012f920 00000000`00000000
00000000`0012f928 00000000`775a208d ntdll!KiUserExceptionDispatch+0x53
00000000`0012f930 00000000`00000000
00000000`0012f938 00000000`0012f930 ; exception context
00000000`0012f940 01c8d5f0`00000000
00000000`0012f948 00000000`00000000
...
```

<sup>149</sup> Custom Postmortem Debuggers in Vista, Memory Dump Analysis Anthology, Volume 1, page 618

```
0:000> ub ntdll!KiUserExceptionDispatch+0x53
ntdll!KiUserExceptionDispatch+0x35:
00000000`775a206f xor     edx,edx
00000000`775a2071 call    ntdll!RtlRestoreContext (00000000`775a2255)
00000000`775a2076 jmp    ntdll!KiUserExceptionDispatch+0x53 (00000000`775a208d)
[...]
00000000`775a2082 mov     rdx,rsp
00000000`775a2085 xor     r8b,r8b
00000000`775a2088 call   ntdll!NtRaiseException (00000000`775a1550)

0:000> .cxr 00000000`0012f930
rax=00000001400223d0 rbx=0000000000000000 rcx=0000000140022128
rdx=0000000000000001 rsi=0000000000000006 rdi=0000000140022120
rip=0000000140001a72 rsp=000000000012fed0 rbp=0000000000000000
r8=000007fffffdde000 r9=0000000000000001 r10=0000000000000000
r11=0000000000000246 r12=0000000000000000 r13=0000000000000002
r14=0000000000000000 r15=0000000000000000
iopl=0          nv up ei pl zr ac po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b          efl=00010256
NestedException+0x1a72:
00000001`40001a72 c7042500000000000000000000000000 mov dword ptr [0],0 ds:00000000`00000000=????????
```

However, if we have a first-chance exception Dump3 from some exception monitoring program (**Early Crash Dump** pattern, page 312) we see that *NtRaiseException* parameter points to "Inner" Exception 2 context (a different and earlier address, shown in italics underlined):

```
This dump file has an exception of interest stored in it.  
The stored exception information can be accessed via .ecxr.  
(11dc.f94): Access violation - code c0000005 (first/second chance not available)  
*** ERROR: Module load completed but symbols could not be loaded for NestedException.exe  
NestedException+0x1a72:  
00000001`40001a72 c70425000000000000000000 mov dword ptr [0],0 ds:00000000`00000000=????????  
  
0:000> dqs 000000000012d000 0000000000130000  
...  
00000000`0012f928 00000000`775a2068 ntdll!KiUserExceptionDispatch+0x2e  
00000000`0012f930 00000000`00000000  
00000000`0012f938 00000000`0012f930 ; exception context  
  
0:000> .cxr 00000000`0012f930  
[...]  
NestedException+0x19fa:  
00000001`400019fa c70425000000000000000000 mov dword ptr [0],0 ds:00000000`00000000=????????
```

Similar can be said about Dump2 where *NtRaiseException* parameter points to Exception 1 context. But Dump1 doesn't have any traces of exception processing as expected. All 4 dump files can be downloaded from [FTP](#) to play with<sup>150</sup>.

<sup>150</sup> <ftp://dumpanalysis.org/pub/CDAPatternNestedExceptions.zip>

## Nested Offender

Sometimes we suspect one component, but there is another, **Nested Offender**, which raised an exception. That exception propagated through exception filters and handlers and prompted the suspected component to respond with a diagnostic message. Here is an example of a thread showing the runtime error message box:



The corresponding **Stack Trace** (page 926) points to *ComponentB* module:

```
0:087> kL
ChildEBP RetAddr
5546ddf0 76f50dde nt!KiFastSystemCallRet
5546ddf4 76f3b0b2 user32!NtUserWaitMessage+0xc
5546de28 76f3bcda user32!DialogBox2+0x202
5546de50 76f8ccdc user32!InternalDialogBox+0xd0
5546def0 76f8d25e user32!SoftModalMessageBox+0x69f
5546e040 76f8d394 user32!MessageBoxWorker+0x2c7
5546e098 76f8d43e user32!MessageBoxTimeoutW+0x7f
5546e0cc 76f8d5ec user32!MessageBoxTimeoutA+0xa1
5546e0ec 6f245ac7 user32!MessageBoxExA+0x1b
5546e10c 76f8d65e ieframe!Detour_MessageBoxExA+0x2c
5546e128 0c841c28 user32!MessageBoxA+0x45
WARNING: Stack unwind information not available. Following frames may be wrong.
5546e16c 0c83d1b9 ComponentB!DllUnregisterServer+0x9437
5546e190 0c83cf72 ComponentB!DllUnregisterServer+0x49c8
00000000 00000000 ComponentB!DllUnregisterServer+0x4781

0:087> .asm no_code_bytes
Assembly options: no_code_bytes
```

```
0:087> ub 0c841c28
ComponentB!DllUnregisterServer+0x941d:
0c841c0e push    dword ptr [ebp+10h]
0c841c11 push    dword ptr [ebp+0Ch]
0c841c14 push    dword ptr [ebp+8]
0c841c17 push    dword ptr [ebp-4]
0c841c1a push    dword ptr [ComponentB!DllUnregisterServer+0x135ab (0c84bd9c)]
0c841c20 call    ComponentB!DllUnregisterServer+0x4aab (0c83d29c)
0c841c25 pop     ecx
0c841c26 call    eax
```

However, when looking at the raw stack data, we see exception processing residue (**Hidden Exception pattern**, page 457) that points to *ComponentA* that tried to allocate more memory than was available (*bad\_alloc C++ Exception*, page 108):

```
0:087> !teb
TEB at 7ff84000
ExceptionList:      5546e4e8
StackBase:          55470000
StackLimit:         55457000
SubSystemTib:       00000000
FiberData:          00001e00
ArbitraryUserPointer: 00000000
Self:               7ff84000
EnvironmentPointer: 00000000
ClientId:           0000136c . 00001714
RpcHandle:          00000000
Tls Storage:        5826eef0
PEB Address:        7ffd7000
LastErrorValue:     0
LastStatusValue:    0
Count Owned Locks: 0
HardErrorMode:      0

0:087> dds 55457000 55470000
55457000 00000000
[...]
5546e694 55470000
5546e698 55457000
5546e69c 00274cb0
5546e6a0 5546e9f4
5546e6a4 772899f7 ntdll!KiUserExceptionDispatcher+0xf
5546e6a8 0046e6b8
5546e6ac 5546e6d8
5546e6b0 5546e6b8
[...]

0:087> .cxr 5546e6d8
eax=5546e9a4 ebx=00001000 ecx=00000003 edx=00000000 esi=689a9c54 edi=6b330000
eip=771942eb esp=5546e9a4 ebp=5546e9f4 iopl=0 nv up ei pl nz ac po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00240212
kernel32!RaiseException+0x58:
771942eb c9             leave
```

```

0:087> .exr 5546e6b8
ExceptionAddress: 771942eb (kernel32!RaiseException+0x00000058)
  ExceptionCode: e06d7363 (C++ EH exception)
  ExceptionFlags: 00000001
NumberParameters: 3
  Parameter[0]: 19930520
  Parameter[1]: 5546ea3c
  Parameter[2]: 688b7954
pExceptionObject: 5546ea3c
_s_ThrowInfo : 688b7954
Type : class std::bad_alloc
Type : class std::exception

0:087> kL
*** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr
5546e9f4 6882dead kernel32!RaiseException+0x58
WARNING: Stack unwind information not available. Following frames may be wrong.
5546ea2c 6882a59d ComponentA!DllUnregisterServer+0x1adbe0
5546ea48 6868157b ComponentA!DllUnregisterServer+0x1aa2d0
5546ea74 6869d2c6 ComponentA!DllUnregisterServer+0x12ae
5546ea88 6868a415 ComponentA!DllUnregisterServer+0x1cff9
5546eaa0 685a165e ComponentA!DllUnregisterServer+0xa148
5546eac8 685a9828 ComponentA+0x5165e
5546ebc8 68605fdd ComponentA+0x59828
5546ebf0 6a807aba ComponentA+0xb5fdd
5546ec08 6a807a77 mshtml!CViewDispClient::Invalidate+0x59
5546ec20 00000000 mshtml!CDispRoot::InvalidateRoot+0x1d

0:087> ub 6882dead
ComponentA!DllUnregisterServer+0x1adbc4:
6882de91 je      ComponentA!DllUnregisterServer+0x1abcd (6882de9a)
6882de93 mov     dword ptr [ebp-0Ch],1994000h
6882de9a lea     eax,[ebp-0Ch]
6882de9d push    eax
6882de9e push    dword ptr [ebp-10h]
6882dea1 push    dword ptr [ebp-1Ch]
6882dea4 push    dword ptr [ebp-20h]
6882dea7 call    dword ptr [ComponentA!DllUnregisterServer+0x1ba0a3 (6883a370)]

```

It happens that *ComponentB* has an exception filter that shows the runtime error:

```

0:087> !exchain
5546e4e8: ComponentB!DllUnregisterServer+18df (0c83a0d0)
5546e578: kernel32!_except_handler4+0 (7715e289)
5546e5e4: ntdll!ExecuteHandler2+3a (77289bad)
5546faa8: user32!_except_handler4+0 (76f951ba)
5546fb0c: user32!_except_handler4+0 (76f951ba)
5546fb0d: ntdll!_except_handler4+0 (77239834)
Invalid exception stack at ffffffff

```

Nested Offender is different from **Nested Exception** pattern (page 726). The latter is about an exception handler that experiences or throws another exception.

## Network Packet Buildup

When looking at the network packet pools using NDIS WinDbg extension, we might see increased number of allocated blocks possibly correlated with network problems, for example:

```
0: kd> !ndiskd.pktpools * normal
Pool      Allocator BlocksAllocated BlockSize PktsPerBlock PacketLength
8a467e20 b9090f96 0x1          0x1000    0x14        0xc8  tcpip!ARPRegister+119
8a491460 ba4eea56 0x1          0x1000    0x14        0xc8  wanarp!WanpInitializeNdis+a8
8a466508 b905d368 0x1          0x1000    0xd         0x138  tcpip!InitForwardingPools+53
8a373578 b905becb 0x3          0x1000    0x11        0xe8  tcpip!AllocIPPacketList+59
8a466580 b9095ac5 0x1          0x1000    0xe         0x118  tcpip!IPInit+e0
8a460958 bac40a97 0xb          0x1000    0x14        0xc8  vmxnet+a97

0: kd> !ndiskd.pktpools * no sent packets
Pool      Allocator BlocksAllocated BlockSize PktsPerBlock PacketLength
8a467e20 b9090f96 0x1          0x1000    0x14        0xc8  tcpip!ARPRegister+119
8a491460 ba4eea56 0x1          0x1000    0x14        0xc8  wanarp!WanpInitializeNdis+a8
8a466508 b905d368 0x1          0x1000    0xd         0x138  tcpip!InitForwardingPools+53
8a373578 b905becb 0xa3        0x1000    0x11        0xe8  tcpip!AllocIPPacketList+59
8a466580 b9095ac5 0x1          0x1000    0xe         0x118  tcpip!IPInit+e0
8a460958 bac40a97 0x9b        0x1000    0x14        0xc8  vmxnet+a97
```

## No Component Symbols

**No Component Symbols** often happens in crash dumps. In this case, we can guess what a component does by looking at its name, overall thread stack where it is called and also its import table. Here is an example. We have *component.sys* driver visible on some thread stack in a kernel dump but we don't know what that component can potentially do. Because we don't have symbols we cannot see its imported functions:

```
kd> x component!*
```

We use **!dh** command to dump its image headers:

```
kd> lmv m component
start           end             module name
fffffadf`e0eb5000 ffffffadf`e0ebc000  component  (no symbols)
Loaded symbol image file: component.sys
Image path: \??\C:\Component\x64\component.sys
Image name: component.sys
Timestamp:      Sat Jul 01 19:06:16 2006 (44A6B998)
CheckSum:        000074EF
ImageSize:       00007000
Translations:    0000.04b0 0000.04e0 0409.04b0 0409.04e0

kd> !dh fffffadf`e0eb5000
File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
 8664 machine (X64)
 6 number of sections
44A6B998 time date stamp Sat Jul 01 19:06:16 2006
 0 file pointer to symbol table
 0 number of symbols
F0 size of optional header
22 characteristics
  Executable
  App can handle >2gb addresses
OPTIONAL HEADER VALUES
 20B magic #
 8.00 linker version
 C00 size of code
 A00 size of initialized data
 0 size of uninitialized data
5100 address of entry point
1000 base of code
----- new -----
0000000000010000 image base
 1000 section alignment
 200 file alignment
 1 subsystem (Native)
5.02 operating system version
5.02 image version
5.02 subsystem version
7000 size of image
```

```

400 size of headers
74EF checksum
0000000000040000 size of stack reserve
0000000000010000 size of stack commit
0000000000100000 size of heap reserve
0000000000100000 size of heap commit
    0 [      0] address [size] of Export Directory
    51B0 [     28] address [size] of Import Directory
    6000 [    3B8] address [size] of Resource Directory
    4000 [     6C] address [size] of Exception Directory
    0 [      0] address [size] of Security Directory
    0 [      0] address [size] of Base Relocation Directory
    2090 [     1C] address [size] of Debug Directory
    0 [      0] address [size] of Description Directory
    0 [      0] address [size] of Special Directory
    0 [      0] address [size] of Thread Storage Directory
    0 [      0] address [size] of Load Configuration Directory
    0 [      0] address [size] of Bound Import Directory
2000 [    88] address [size] of Import Address Table Directory
    0 [      0] address [size] of Delay Import Directory
    0 [      0] address [size] of COR20 Header Directory
    0 [      0] address [size] of Reserved Directory
...
...
...

```

Then we display the contents of Import Address Table Directory using **dps** command:

```

kd> dps ffffffadf`e0eb5000+2000 ffffffadf`e0eb5000+2000+88
fffffadf`e0eb7000  fffff800`01044370 nt!IoCompleteRequest
fffffadf`e0eb7008  fffff800`01019700 nt!IoDeleteDevice
fffffadf`e0eb7010  fffff800`012551a0 nt!IoDeleteSymbolicLink
fffffadf`e0eb7018  fffff800`01056a90 nt!MiResolveTransitionFault+0x7c2
fffffadf`e0eb7020  fffff800`0103a380 nt!ObDereferenceObject
fffffadf`e0eb7028  fffff800`0103ace0 nt!KeWaitForSingleObject
fffffadf`e0eb7030  fffff800`0103c570 nt!KeSetTimer
fffffadf`e0eb7038  fffff800`0102d070 nt!IoBuildPartialMdl+0x3
fffffadf`e0eb7040  fffff800`012d4480 nt!PsTerminateSystemThread
fffffadf`e0eb7048  fffff800`01041690 nt!KeBugCheckEx
fffffadf`e0eb7050  fffff800`010381b0 nt!KeInitializeTimer
fffffadf`e0eb7058  fffff800`0103ceb0 nt!ZwClose
fffffadf`e0eb7060  fffff800`012b39f0 nt!ObReferenceObjectByHandle
fffffadf`e0eb7068  fffff800`012b7380 nt!PsCreateSystemThread
fffffadf`e0eb7070  fffff800`01251f90 nt!FsRtlpIsDfsEnabled+0x114
fffffadf`e0eb7078  fffff800`01275160 nt!IoCreateDevice
fffffadf`e0eb7080  00000000`00000000
fffffadf`e0eb7088  00000000`00000000

```

We see that this driver under certain circumstances can bugcheck the system using KeBugCheckEx, it creates system thread(s) (*PsCreateSystemThread*) and uses timer(s) (*KeInitializeTimer*, *KeSetTimer*).

If we see *name+offset* in the import table (I think this is an effect of **OMAP Code Optimization**, page 756) we can get the function by using **In** command (list nearest symbols):

```
kd> ln ffffff800`01056a90
(ffffff800`01056760)  nt!MiResolveTransitionFault+0x7c2  |  (fffff800`01056a92)  nt!RtlInitUnicodeString

kd> ln ffffff800`01251f90
(ffffff800`01251e90)  nt!FsRt1pIsDfsEnabled+0x114  |  (fffff800`01251f92)  nt!IoCreateSymbolicLink
```

This technique is useful if we have a bugcheck that happens when a driver calls certain functions or must call certain functions in pairs, like bugcheck 0x20:

```
kd> !analyze -show 0x20
KERNEL_APP_PENDING_DURING_EXIT (20)
The key data item is the thread's APC disable count. If this is non-zero, then this is the source of the problem. The APC disable count is decremented each time a driver calls KeEnterCriticalSection, KeInitializeMutex, or FsRtlEnterFileSystem. The APC disable count is incremented each time a driver calls KeLeaveCriticalSection, KeReleaseMutex, or FsRtlExitFileSystem. Since these calls should always be in pairs, this value should be zero when a thread exits. A negative value indicates that a driver has disabled APC calls without re-enabling them. A positive value indicates that the reverse is true. If you ever see this error, be very suspicious of all drivers installed on the machine – especially unusual or non-standard drivers. Third party file system redirectors are especially suspicious since they do not generally receive the heavy duty testing that NTFS, FAT, RDR, etc receive. This current IRQL should also be 0. If it is not, that a driver's cancelation routine can cause this bugcheck by returning at an elevated IRQL. Always attempt to note what you were doing/closing at the time of the crash, and note all of the installed drivers at the time of the crash. This symptom is usually a severe bug in a third party driver.
```

Then we can see at least whether the suspicious driver could have potentially used those functions and if it imports one of them we can see whether it imports the corresponding counterpart function.

**No Component Symbols** pattern can be easily identified in stack traces by huge function offsets or no exported functions at all:

```
STACK_TEXT:
WARNING: Stack unwind information not available. Following frames may be wrong.
00b2f42c 091607aa mydll!foo+0x8338
00b2f4cc 7c83ab9e mydll2+0x8fe3
```

## No Current Thread

**No Current Thread** pattern is rare, but we observed a few occurrences in a large set of process memory dumps:

```
0:???> k
WARNING: The debugger does not have a current process or thread
WARNING: Many commands will not work
^ Illegal thread error in 'k'

0:???> ~
WARNING: The debugger does not have a current process or thread
WARNING: Many commands will not work
0 Id: 95f4.6780 Suspend: 1 Teb: 7efdd000 Unfrozen
```

Setting a current thread helps:

```
0:???> ~0s
WARNING: The debugger does not have a current process or thread
WARNING: Many commands will not work
eax=037d0010 ebx=0002bda0 ecx=03b1a010 edx=00000007 esi=037d0010 edi=03b069fc
eip=0397939f esp=0018fd98 ebp=0018fdd8 iopl=0 nv up ei pl nz na po nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00200202
D11A+0x939f:
0397939f 8b10 mov edx,dword ptr [eax] ds:002b:037d0010=03b1a010

0:000> k
ChildEBP RetAddr
WARNING: Stack unwind information not available. Following frames may be wrong.
0018fdd8 03975257 D11A+0x939f
0018fdf8 03975577 D11A+0x5257
0018fe58 772bb9a0 D11A+0x5577
0018fe78 772d9b96 ntdll!LdrpCallInitRoutine+0x14
0018ff1c 772d9a38 ntdll!LdrShutdownProcess+0x1aa
0018ff30 752279f4 ntdll!RtlExitUserProcess+0x74
0018ff44 0040625d kernel32!ExitProcessStub+0x12
0018ff5c 012528e5 Application+0x625d
0018ff88 7522339a Application!foo+0xdc88f1
0018ff94 772bbf42 kernel32!BaseThreadInitThunk+0xe
0018ffd4 772bbf15 ntdll!__RtlUserThreadStart+0x70
0018ffec 00000000 ntdll!_RtlUserThreadStart+0x1b
```

However, EIP of the new current thread doesn't point to any access violation, and the dereferenced address is valid:

```
0:000> !address 037d0010
Usage: <unclassified>
Allocation Base: 037d0000
Base Address: 037d0000
End Address: 038dd000
Region Size: 0010d000
Type: 00020000 MEM_PRIVATE
```

State:	00001000 MEM_COMMIT
Protect:	00000004 PAGE_READWRITE

Also, if we inspect the raw stack data, we don't find any **Hidden Exceptions** (page 457) there. So we conclude that the missing thread was exceptional. Indeed, there is a saved exception context in the process memory dump:

```
0:000> .exr -1
ExceptionAddress: 08a9ae18 (<Unloaded_DllB.dll>+0x001cae18)
ExceptionCode: c0000005 (Access violation)
ExceptionFlags: 00000001
NumberParameters: 1
Parameter[0]: 00000008
```

## No Data Types

Sometimes we don't have symbols (**No Component Symbols** pattern, page 734) or have only a restricted set. For example, in a base OS we have data types:

```
0:016> dt ntdll!*
ntdll!LIST_ENTRY64
ntdll!LIST_ENTRY32
ntdll!_KUSER_SHARED_DATA
ntdll!_KSYSTEM_TIME
ntdll!_KSYSTEM_TIME
ntdll!_NT_PRODUCT_TYPE
[...]
```

In the “private” version we don’t have them although the symbol file exists:

```
0:015> dt ntdll!*
0:015> !lmi ntdll
Loaded Module Info:      [ntdll]
Module:                  ntdll
Base Address:            0000000076de0000
Image Name:              ntdll.dll
Machine Type:            34404 (X64)
Time Stamp:              4dc9861 Fri May 13 21:45:21 2011
Size:                    17f000
CheckSum:                188814
Characteristics:         2022 perf
Debug Data Dirs:          Type Size VA Pointer
CODEVIEW 22, f72a8, f66a8 RSDS - GUID: {05A648A7-625D-42E7-B736-7816F0CA1E0C}
Age: 2, Pdb:               ntdll.pdb
CLSID 8, f72a0, f66a0 [Data not mapped]
Image Type: MEMORY - Image read successfully from loaded memory.
Symbol Type: PDB - Symbols loaded successfully from symbol server.
c:\mss\ntdll.pdb\05A648A7625D42E7B7367816F0CA1E0C2\ntdll.pdb
Load Report: public symbols , not source indexed
c:\mss\ntdll.pdb\05A648A7625D42E7B7367816F0CA1E0C2\ntdll.pdb
```

In such cases manually loading a proximate module may help<sup>151</sup> although we haven’t yet tested it on x64 systems. We also thought of naming the pattern as **Private Modification**, but that would not cover many other cases where types were missing from the very beginning.

---

<sup>151</sup> Coping with Missing Symbolic Information, Memory Dump Analysis Anthology, Volume 1, page 199

## No Process Dumps

The absence of crash dumps when we expect them can be considered as a pattern on its own. This can happen due to a variety of reasons and troubleshooting should be based on the distinction between crashes and hangs<sup>152</sup>. We have three combinations here:

1. A process is visible in Task Manager and is functioning normally
2. A process is visible in Task Manager and has stopped functioning normally
3. A process is not visible in Task Manager

If a process is visible in the task list and is functioning normally, then the following reasons should be considered:

- Exceptions haven't happened yet due to different code execution paths or the time has not come yet, and we need to wait
- Exceptions haven't happened yet due to a different memory layout. This can be the instance of **Changed Environment** pattern (page 114).

If a process is visible in Task Manager and has stopped functioning normally, then it may be hanging and waiting for some input. In such cases, it is better to get process dumps proactively<sup>153</sup>. If a process is not visible in Task Manager, then the following reasons should be considered:

- Debugger value for *AeDebug* key is invalid, missing or points to a wrong path, or a command line has wrong arguments. For examples see the discussion about custom postmortem debuggers<sup>154</sup> or NTSD<sup>155</sup>.
- Something is wrong with the exception handling mechanism or WER settings. Use Process Monitor to see what processes are launched, and modules are loaded when an exception happens. Check WER settings in Control panel.
- Try *LocalDumps* registry key<sup>156</sup>
- Use live debugging techniques like attaching to a process or running a process under a debugger to monitor exceptions and saving first chance exception crash dumps (**Early Crash Dump** pattern, page 312).

This is a very important pattern for technical support environments that rely on post-mortem analysis.

<sup>152</sup> Crashes and Hangs Differentiated, Memory Dump Analysis Anthology, Volume 1, page 36

<sup>153</sup> Proactive Crash Dumps, Memory Dump Analysis Anthology, Volume 1, page 39

<sup>154</sup> Custom Postmortem Debuggers in Vista, Memory Dump Analysis Anthology, Volume 1, page 618

<sup>155</sup> NTSD on x64 Windows, Memory Dump Analysis Anthology, Volume 1, page 633

<sup>156</sup> Local Crash Dumps in Vista, Memory Dump Analysis Anthology, Volume 1, page 606

## No System Dumps

This is a corresponding pattern similar to **No Process Dumps** (page 740) where the system bluescreens either on demand or because of a bugcheck condition but no kernel or complete dumps are saved. In such cases, we would advise checking free space on a drive where memory dumps are supposed to be saved. This is because crash dumps are saved to a page file first and then copied to a separate file at boot time, by default to memory.dmp file<sup>157</sup>. In case we have enough free space but not enough page file space we might get an instance of **Truncated Dump** (page 1015) or **Corrupt Dump** (page 137) patterns.

## Comments

In some cases of severe corruption or system malfunction, the system doesn't have any chance to execute system code to save a dump and display a blue screen. To check this case, it is recommended to disable "Automatically Restart" option in Control Panel. I once experimented with a driver by calling a user space code from kernel mode and from it tried to call a GDI32 function. The system rebooted instantly without BSOD screen.

---

<sup>157</sup> Savedump.exe and Pagefile, Memory Dump Analysis Anthology, Volume 1, page 628

## Not My Thread

We found out that in Windows 10 (at least on our working system) Notepad is no longer a single threaded application even without opening any common dialogs (like in **Eevental Dumps** analysis pattern example, page 328). It has at least 3 additional threads (and other modeling applications we use for our training also have additional threads):

```
0:000> ~*k

0 Id: 3a64.3b38 Suspend: 1 Teb: 00007ff6`a914d000 Unfrozen
# Child-SP RetAddr Call Site
00 000000e5`6298f938 00007ffa`e57cf8e5 USER32!NtUserGetMessage+0xa
01 000000e5`6298f940 00007ffa`a9603470 USER32!GetMessageW+0x25
02 000000e5`6298f970 00007ffa`a96141f5 notepad!WinMain+0x178
03 000000e5`6298f9f0 00007ffa`e3b42d92 notepad!WinMainCRTStartup+0x1c5
04 000000e5`6298fab0 00007ffa`e5bc9f64 KERNEL32!BaseThreadInitThunk+0x22
05 000000e5`6298fae0 00000000`00000000 ntdll!RtlUserThreadStart+0x34

1 Id: 3a64.38b0 Suspend: 1 Teb: 00007ff6`a914b000 Unfrozen
# Child-SP RetAddr Call Site
00 000000e5`62bffa58 00007ffa`e5bf93a5 ntdll!NtWaitForWorkViaWorkerFactory+0xa
01 000000e5`62bffa60 00007ffa`e3b42d92 ntdll!TppWorkerThread+0x295
02 000000e5`62bffe60 00007ffa`e5bc9f64 KERNEL32!BaseThreadInitThunk+0x22
03 000000e5`62bffe90 00000000`00000000 ntdll!RtlUserThreadStart+0x34

2 Id: 3a64.3940 Suspend: 1 Teb: 00007ff6`a9149000 Unfrozen
# Child-SP RetAddr Call Site
00 000000e5`62c7f718 00007ffa`e5bf93a5 ntdll!NtWaitForWorkViaWorkerFactory+0xa
01 000000e5`62c7f720 00007ffa`e3b42d92 ntdll!TppWorkerThread+0x295
02 000000e5`62c7fb20 00007ffa`e5bc9f64 KERNEL32!BaseThreadInitThunk+0x22
03 000000e5`62c7fb50 00000000`00000000 ntdll!RtlUserThreadStart+0x34

3 Id: 3a64.1030 Suspend: 1 Teb: 00007ff6`a9147000 Unfrozen
# Child-SP RetAddr Call Site
00 000000e5`62d1f878 00007ffa`e5bf93a5 ntdll!NtWaitForWorkViaWorkerFactory+0xa
01 000000e5`62d1f880 00007ffa`e3b42d92 ntdll!TppWorkerThread+0x295
02 000000e5`62d1fc80 00007ffa`e5bc9f64 KERNEL32!BaseThreadInitThunk+0x22
03 000000e5`62d1fc00 00000000`00000000 ntdll!RtlUserThreadStart+0x34
```

This gave us an idea for the analysis pattern we call **Not My Thread** since additional threads can be started by any other process DLLs, for example, by **Hooksware** (page 1175). However, we need to distinguish between unexpectedly added threads, threads with **Special Stack Traces** (page 882) and **Special Threads** (page 883), for example, from .NET support.

## Not My Version

### Hardware

This is a hardware counterpart of **Not My Version** pattern for software (page 744). Some problems manifest themselves on different hardware not used at the time of the product testing. In such cases we can look at the kernel and complete memory dumps, extract hardware information using **!sysinfo** command and compare differences. This is similar to **Virtualized System** pattern (page 1075) and might provide troubleshooting hints. One example we have seen in the past, involved a graphics intensive application that relied heavily on hardware acceleration features. It was tested with certain processors and chipsets but after a few years failed to work on one computer despite the same OS image and drivers. **!sysinfo** command revealed significant hardware differences: the failing client computer was a newer faster multiprocessor machine.

## Software

**Not My Version** is another basic pattern of DLL Hell variety. It is when we look at component timestamps and paths and realize that one of the modules from the production environment is older than we had during development and testing. The **!mft** WinDbg command will produce the necessary output. If there are many modules, we may want to create a CAD graph (Component Age Diagram<sup>158</sup>) to spot anomalies visually. Component version check is one of the basic troubleshooting and system administration activities. Here is one example (module start and end load addresses are removed for visual clarity):

```
0:000> kL
Child-SP          RetAddr          Call Site
00000000`0012fed8 00000001`40001093 MyDLL!fnMyDLL
00000000`0012fee0 00000001`40001344 2DLLs+0x1093
00000000`0012ff10 00000000`773acdcd 2DLLs+0x1344
00000000`0012ff60 00000000`774fc6e1 kernel32!BaseThreadInitThunk+0xd
00000000`0012ff90 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

0:000> !mft
module name
MyDLL      C:\OLD\MyDLL.dll Wed Jun 18 14:49:13 2004
user32     C:\Windows\System32\user32.dll Thu Feb 15 05:22:33 2007
kernel32   C:\Windows\System32\kernel32.dll Thu Nov 02 11:14:48 2006
ntdll1     C:\Windows\System32\ntdll.dll Thu Nov 02 11:16:02 2006
2DLLs      C:\2DLLs\2DLLs.exe Thu Jun 19 10:46:44 2008 (485A2B04)
uxtheme    C:\Windows\System32\uxtheme.dll Thu Nov 02 11:15:07 2006
rpcrt4    C:\Windows\System32\rpcrt4.dll Tue Jul 17 05:21:15 2007
lpk        C:\Windows\System32\lpk.dll Thu Nov 02 11:12:33 2006
oleaut32   C:\Windows\System32\oleaut32.dll Thu Dec 06 05:09:35 2007
usp10       C:\Windows\System32\usp10.dll Thu Nov 02 11:15:03 2006
ole32       C:\Windows\System32\ole32.dll Thu Nov 02 11:14:31 2006
advapi32   C:\Windows\System32\advapi32.dll Thu Nov 02 11:11:35 2006
gdi32       C:\Windows\System32\gdi32.dll Thu Feb 21 04:40:51 2008
msvcrt     C:\Windows\System32\msvcrt.dll Thu Nov 02 11:13:37 2006
imm32       C:\Windows\System32\imm32.dll Thu Nov 02 11:13:15 2006
msctf       C:\Windows\System32\msctf.dll Thu Nov 02 11:13:42 2006
```

This pattern should be checked when we have instances of **Module Variety** (page 703) and, especially, **Duplicated Module** (page 287). Note that this pattern can also easily become an anti-pattern when applied to **Unknown Component** (page 1035)<sup>159</sup>.

<sup>158</sup> How Old Is Your Application or System?, Memory Dump Analysis Anthology, Volume 2, page 126

<sup>159</sup> Crash Dump Analysis AntiPatterns, Alien Component, Memory Dump Analysis Anthology, Volume 1, page 493

## NULL Pointer

### Linux

#### Code

This is a Linux variant of **NULL Pointer** (code) pattern previously described for Mac OS X (page 747) and Windows (page 750) platforms:

```
(gdb) bt
#0  0x0000000000000000 in ?? ()
#1  0x0000000000400531 in procB ()
#2  0x00000000004005f8 in bar_four ()
#3  0x0000000000400608 in foo_four ()
#4  0x0000000000400620 in thread_four ()
#5  0x0000000000401630 in start_thread (arg=<optimized out>
at pthread_create.c:304
#6  0x00000000004324e9 in clone ()
#7  0x0000000000000000 in ?? ()

(gdb) disassemble procB
Dump of assembler code for function procB:
0x0000000000400516 <+0>: push    %rbp
0x0000000000400517 <+1>: mov     %rsp,%rbp
0x000000000040051a <+4>: sub    $0x10,%rsp
0x000000000040051e <+8>: movq   $0x0,-0x8(%rbp)
0x0000000000400526 <+16>: mov    -0x8(%rbp),%rdx
0x000000000040052a <+20>: mov    $0x0,%eax
0x000000000040052f <+25>: callq  *%rdx
0x0000000000400531 <+27>: leaveq
0x0000000000400532 <+28>: retq
End of assembler dump.

(gdb) info r rdx
rdx          0x0 0
```

## Data

This is a Linux variant of **NULL Pointer** (data) pattern previously described for Mac OS X (page 745) and Windows (page 752) platforms:

```
(gdb) bt
#0  0x0000000000400500 in procA ()
#1  0x000000000040057a in bar_two ()
#2  0x000000000040058a in foo_two ()
#3  0x00000000004005a2 in thread_two ()
#4  0x0000000000401630 in start_thread (arg=<optimized out>) at pthread_create.c:304
#5  0x00000000004324e9 in clone ()
#6  0x0000000000000000 in ?? ()

(gdb) x/i 0x400500
=> 0x400500 <procA+16>: movl    $0x1,(%rax)

(gdb) info r $rax
rax            0x0 0

(gdb) x $rax
0x0: Cannot access memory at address 0x0
```

## Mac OS X

## Code

This is a Mac OS X / GDB counterpart to **NULL Pointer (Code)** pattern:

```
(gdb) bt
#0 0x0000000000000000 in ?? ()
#1 0x000000010e8cce73 in bar (ps=0x7fff6e4cbac0)
#2 0x000000010e8cce95 in foo (ps=0x7fff6e4cbac0)
#3 0x000000010e8cced5 in main (argc=1, argv=0x7fff6e4cbb08)

(gdb) disass 0x000000010e8cce73-3 0x000000010e8cce73
Dump of assembler code from 0x10e8cce70 to 0x10e8cce73:
0x000000010e8cce70 : callq *0x8(%rdi)
End of assembler dump.

(gdb) info r rdi
rdi 0x7fff6e4cbac0 140735043910336

(gdb) x/2 0x7fff6e4cbac0
0x7fff6e4cbac0: 0x0000000a 0x00000000

(gdb) p/x *($rdi+8)
$7 = 0x0

(gdb) bt
#0 0x0000000000000000 in ?? ()
#1 0x000000010e8cce73 in bar (ps=0x7fff6e4cbac0)
#2 0x000000010e8cce95 in foo (ps=0x7fff6e4cbac0)
#3 0x000000010e8cced5 in main (argc=1, argv=0x7fff6e4cbb08)

(gdb) ptype MYSTRUCT
type = struct _MyStruct_tag {
    int data;
    PFUNC pfunc;
}

(gdb) print {MYSTRUCT}0x7fff6e4cbac0
$2 = {data = 10, pfunc = 0}
```

Here's the source code of the modeling application:

```
typedef void (*PFUNC)(void);

typedef struct _MyStruct_tag
{
    int data;
    PFUNC pfunc;
} MYSTRUCT;

void bar(MYSTRUCT *ps)
{
    ps->pfunc();
}

void foo(MYSTRUCT *ps)
{
    bar(ps);
}

int main(int argc, const char * argv[])
{
    MYSTRUCT pstruct = {10, NULL};

    foo(&pstruct);

    return 0;
}
```

## Data

This is a Mac OS X / GDB counterpart to **NULL Pointer (Data)** pattern:

```
(gdb) bt
#0 0x000000010d3b0e90 in bar () at main.c:15
#1 0x000000010d3b0ea9 in foo () at main.c:20
#2 0x000000010d3b0ec4 in main (argc=1,
argv=0x7fff6cfafbf8) at main.c:25

(gdb) disassemble
Dump of assembler code for function bar:
0x000000010d3b0e80 <bar+0>:    push %rbp
0x000000010d3b0e81 <bar+1>:    mov %rsp,%rbp
0x000000010d3b0e84 <bar+4>:    movq $0x0,-0x8(%rbp)
0x000000010d3b0e8c <bar+12>:   mov -0x8(%rbp),%rax
0x000000010d3b0e90 <bar+16>:   movl $0x1,(%rax)
0x000000010d3b0e96 <bar+22>:   pop %bp
0x000000010d3b0e97 <bar+23>:   retq
End of assembler dump.

(gdb) p/x $rax
$1 = 0x0
```

## Windows

## Code

**NULL Pointer** is a specialization of **Invalid Pointer** pattern (page 589), and it is the most easily recognized pattern with a straightforward fix most of the time according to our debugging experience. Checking the pointer value to be non-NULL might not work if the pointer value is random (**Wild Pointer** pattern, page 1151) but at least it eliminates this class of problems. NULL pointers can be **NULL Data Pointers** (page 752) or **NULL Code Pointers**. The latter happens when we have a pointer to some function, and we try to call it. Consider this example:

```
0:002> r
eax=00000000 ebx=00000000 ecx=93630000 edx=00000000 esi=00000000 edi=00000000
eip=00000000 esp=0222ffbc ebp=0222ffec iopl=0 nv up ei pl zr na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010246
00000000 ??          ???

0:002> kv
ChildEBP RetAddr  Args to Child
WARNING: Frame IP not in any known module. Following frames may be wrong.
0222ffb8 7d4dfe21 00000000 00000000 00000000 0x0
0222ffec 00000000 00000000 00000000 00000000 kernel32!BaseThreadStart+0x34
```

Clearly, we have a NULL code pointer here and if we disassemble backward the return address `7d4dfe21` or `BaseThreadStart+0x34` address we would suspect that `BaseThreadStart` function tried to call a thread start procedure:

```
0:002> ub 7d4dfe21
kernel32!BaseThreadStart+0x10:
7d4dfdfd mov     eax,dword ptr fs:[00000018h]
7d4dfe03 cmp     dword ptr [eax+10h],1E00h
7d4dfe0a jne     kernel32!BaseThreadStart+0x2e (7d4dfe1b)
7d4dfe0c cmp     byte ptr [kernel32!BaseRunningInServerProcess (7d560008)],0
7d4dfe13 jne     kernel32!BaseThreadStart+0x2e (7d4dfe1b)
7d4dfe15 call    dword ptr [kernel32!_imp__CsrNewThread (7d4d0310)]
7d4dfe1b push   dword ptr [ebp+0Ch]
7d4dfe1e call    dword ptr [ebp+8]

0:002> dp ebp+8 11
0222fff4 00000000
```

To confirm this suspicion, we can write code that calls *CreateThread* function similar to this one:

```
typedef DWORD (WINAPI *THREADPROC)(PVOID);

DWORD WINAPI ThreadProc(PVOID pvParam)
{
    // Does some work
    return 0;
}

void foo()
{
    /**
     * ...
     */
    THREADPROC thProc = ThreadProc;
    /**
     * ...
     */
    // thProc becomes NULL because of a bug
    /**
     * ...
     */
    HANDLE Thread = CreateThread(NULL, 0, thProc, 0, 0, NULL);
    CloseHandle(Thread);
}
```

## Data

This is a special version of the more general **Invalid Pointer** pattern (page 589) like **NULL Code Pointer** (page 750). The effective address is below 0xFFFF, and it is usually a register with 0 value and a small offset, for example:

```
0: kd> r
Last set context:
eax=8923b008 ebx=00000000 ecx=00000000 edx=8923b008 esi=891312d0 edi=89f0b300
eip=8081c7c4 esp=f1b5d7a4 ebp=f1b5d7a4 iopl=0 nv up ei ng nz ac pe nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010296
nt!IoIsOperationSynchronous+0xe:
8081c7c4 f6412c02 test byte ptr [ecx+2Ch],2 ds:0023:0000002c=??
```

Here, after disassembling the function backward, we see the succession of dereferences starting from [EBP+8] and this means that a pointer to a structure (an IRP here) was passed to the function, and it had a data pointer in it, pointing to another structure and the latter contained a NULL pointer:

```
0: kd> ub 8081c7c4
nt!IoIsOperationSynchronous:
8081c7b6 8bff      mov     edi,edi
8081c7b8 55        push    ebp
8081c7b9 8bec      mov     ebp,esp
8081c7bb 8b4508    mov     eax,dword ptr [ebp+8]
8081c7be 8b4860    mov     ecx,dword ptr [eax+60h]
8081c7c1 8b4918    mov     ecx,dword ptr [ecx+18h]
```

## Comments

x64 example that also shows .cxr command in x64 context:

```
0:006> k
# Child-SP RetAddr Call Site
00 000000a9`be019378 00007ffe`1b11918f ntdll!NtWaitForMultipleObjects+0xa
01 000000a9`be019380 00007ffe`1b11908e KERNELBASE!WaitForMultipleObjectsEx+0xef
02 000000a9`be019680 00007ffe`1b92155c KERNELBASE!WaitForMultipleObjects+0xe
03 000000a9`be0196c0 00007ffe`1b921088 kernel32!BaseReportFault+0x54c
04 000000a9`be019c30 00007ffe`1b1403cd kernel32!BaseReportFault+0x78
05 000000a9`be019c60 00007ffe`1dd8dbf6 KERNELBASE!UnhandledExceptionFilter+0x1fd
06 000000a9`be019d60 00007ffe`1dd65680 ntdll!LdrpLogFatalUserCallbackException+0x56
07 000000a9`be019e90 00007ffe`1dd6666d ntdll!KiUserCallbackDispatcherHandler+0x20
08 000000a9`be019ed0 00007ffe`1dce3c00 ntdll!RtlpExecuteHandlerForException+0xd
09 000000a9`be019f00 00007ffe`1dd6577a ntdll!RtlDispatchException+0x370
0a 000000a9`be01a600 00007ffd`f98e8f69 ntdll!KiUserExceptionDispatch+0x3a
0b 000000a9`be01ad00 00007ffd`f98e5c94 edgehtml!CWebPlatformTridentHost::LoadNewWindowContent+0x35
0c 000000a9`be01ad90 00007ffe`1b84b0b3 edgehtml!CWebPlatform::LoadNewWindowContent+0x64
0d 000000a9`be01ae10 00007ffe`1b80521e rpcrt4!Invoke+0x73
0e 000000a9`be01aea0 00007ffe`1b83aaba rpcrt4!NdrStubCall2+0x34e
0f 000000a9`be01b4f0 00007ffe`1d414b1b rpcrt4!NdrStubCall3+0xea
10 000000a9`be01b560 00007ffe`1d4c25b2 combase!CStdStubBuffer_Invoke+0x6b
```

```

11 000000a9`be01b5a0 00007ffe`1d490845 combase!CoGetContextToken+0x262
12 000000a9`be01b610 00007ffe`1d47f95e combase!CoCreateFreeThreadedMarshaler+0x5735
13 000000a9`be01b830 00007ffe`1d48219e combase!CoGetObjectContext+0x9bce
14 000000a9`be01bb00 00007ffe`1d4842bf combase!CoGetObjectContext+0xc40e
15 000000a9`be01bc00 00007ffe`1d47da9c combase!CoGetObjectContext+0xe52f
16 000000a9`be01bf40 00007ffe`1ba300dc combase!CoGetObjectContext+0x7d0c
17 000000a9`be01c090 00007ffe`1ba2fc07 user32!UserCallWinProcCheckWow+0x1fc
18 000000a9`be01c180 00007ffe`1d4865e9 user32!DispatchMessageWorker+0x1a7
19 000000a9`be01c200 00007ffe`1d486b8f combase!CoGetObjectContext+0x10859
1a 000000a9`be01c270 00007ffe`1d491c2d combase!CoGetObjectContext+0x10dff
1b 000000a9`be01c2d0 00007ffe`1d48d7a9 combase!CoCreateFreeThreadedMarshaler+0x6b1d
1c 000000a9`be01c420 00007ffe`1d48e215 combase!CoCreateFreeThreadedMarshaler+0x2699
1d 000000a9`be01c600 00007ffe`1d41475b combase!CoCreateFreeThreadedMarshaler+0x3105
1e 000000a9`be01c7c0 00007ffe`1b8aa340 combase!NdrOleD1lGetClassObject+0xf2b
1f 000000a9`be01c830 00007ffe`1d414544 rpcrt4!NdrpClientCall13+0x460
20 000000a9`be01cc20 00007ffe`1d51f192 combase!NdrOleD1lGetClassObject+0xd14
21 000000a9`be01cfb0 00007ffe`1d4a8b8d combase!ObjectStublessClient32+0xfc32
22 000000a9`be01d000 00007ffe`1d4a8a65 combase!CoWaitForMultipleHandles+0x3cd
23 000000a9`be01d070 00007ffe`1d49849d combase!CoWaitForMultipleHandles+0x2a5
24 000000a9`be01d110 00007ffe`1d49cc3a combase!CoCreateFreeThreadedMarshaler+0xd38d
25 000000a9`be01d2e0 00007ffe`1d48b6d0 combase!CoCreateFreeThreadedMarshaler+0x11b2a
26 000000a9`be01d3e0 00007ffe`1d4113f7 combase!CoCreateFreeThreadedMarshaler+0x5c0
27 000000a9`be01d430 00007ffe`1d94bd66 combase!CStdStubBuffer2_QueryInterface+0x117
28 000000a9`be01d460 00007ffd`f05ebef0 oleaut32!VariantClear+0x176
29 000000a9`be01d490 00007ffd`f05ed839 eModel!CIEFrameAuto::SetOwner+0x2b0
2a 000000a9`be01d4f0 00007ffd`f05f7892 eModel!CBrowserTabBase::v_OnDestroy+0x69
2b 000000a9`be01d520 00007ffd`f05f5247 eModel!CBrowserTab::v_OnDestroy+0x12
2c 000000a9`be01d550 00007ffd`f05f1b1b eModel!CBrowserTab::v_WndProc+0x447
2d 000000a9`be01d710 00007ffe`1ba300dc eModel!CBrowserTab::s_WndProc+0x5b
2e 000000a9`be01d760 00007ffe`1ba2fe52 user32!UserCallWinProcCheckWow+0x1fc
2f 000000a9`be01d850 00007ffe`1ba3d3fe user32!DispatchClientMessage+0xa2
30 000000a9`be01d8b0 00007ffe`1dd65714 user32!_fnDWORD+0x3e
31 000000a9`be01d910 00007ffe`1ba5061a ntdll!KiUserCallbackDispatcherContinue
32 000000a9`be01d998 00007ffd`f05f6c8c user32!NtUserDestroyWindow+0xa
33 000000a9`be01d9a0 00007ffd`f05f6ebb eModel!CBrowserTab::_DoFinalCleanup+0x35c
34 000000a9`be01da50 00007ffd`f05f6620 eModel!CBrowserTab::_OnConfirmedClose+0x2f
35 000000a9`be01da80 00007ffd`f05cf026 eModel!CBrowserTab::OnClose+0x170
36 000000a9`be01dae0 00007ffd`f063065b eModel!CTabWindow::_TabWindowThreadProc+0x7a6
37 000000a9`be01fd40 00007ffe`1326856f eModel!LCIETab_ThreadProc+0x2bb
38 000000a9`be01fe70 00007ffe`1b912d92 iertutil!IEGetTabWindowExports+0x2f
39 000000a9`be01fea0 00007ffe`1dcdf9f64 kernel32!BaseThreadInitThunk+0x22
3a 000000a9`be01fed0 00000000`00000000 ntdll!RtlUserThreadStart+0x34

```

0:006> .cxr 000000a9`be01a600

```

rax=0000000001000002 rbx=0000000000000009 rcx=0000000000000000
rdx=000000a9be2d1300 rsi=000000a9be01ade0 rdi=0000000000000000
rip=00007ffd98e8f69 rsp=000000a9be01ad00 rbp=000000a9be2d1190
r8=000000a9be2d1190 r9=000000a9be01ade0 r10=00007ffd98e5c30
r11=0000000000001000 r12=00007ffd1b47382 r13=000000a9c01fc358
r14=000000a9be2d1300 r15=000000a9be01b5a0
iopl=0 nv up ei pl zr na po nc
cs=0033 ss=002b ds=002b es=002b fs=0053 gs=002b ef1=00010246
edgehtml!CWebPlatformTridentHost::LoadNewWindowContent+0x35:
00007ffd`f98e8f69 488b07 mov rax,qword ptr [rdi] ds:00000000`00000000=?????????????????
```

```
0:006> kc
*** Stack trace for last set context - .thread/.cxr resets it
# Call Site
00 edgehtml!CWebPlatformTridentHost::LoadNewWindowContent
01 edgehtml!CWebPlatform::LoadNewWindowContent
02 rpcrt4!Invoke
03 rpcrt4!NdrStubCall2
04 rpcrt4!NdrStubCall3
05 combase!CStdStubBuffer_Invoke
06 combase!CoGetContextToken
07 combase!CoCreateFreeThreadedMarshaler
08 combase!CoGetObjectContext
09 combase!CoGetObjectContext
0a combase!CoGetObjectContext
0b combase!CoGetObjectContext
0c user32!UserCallWinProcCheckWow
0d user32!DispatchMessageWorker
0e combase!CoGetObjectContext
0f combase!CoGetObjectContext
10 combase!CoCreateFreeThreadedMarshaler
11 combase!CoCreateFreeThreadedMarshaler
12 combase!CoCreateFreeThreadedMarshaler
13 combase!NdrOleD11GetClassObject
14 rpcrt4!NdrpClientCall3
15 combase!NdrOleD11GetClassObject
16 combase!ObjectStublessClient32
17 combase!CoWaitForMultipleHandles
18 combase!CoWaitForMultipleHandles
19 combase!CoCreateFreeThreadedMarshaler
1a combase!CoCreateFreeThreadedMarshaler
1b combase!CoCreateFreeThreadedMarshaler
1c combase!CStdStubBuffer2_QueryInterface
1d oleaut32!VariantClear
1e eModel!CIEFrameAuto::SetOwner
1f eModel!CBrowserTabBase::v_OnDestroy
20 eModel!CBrowserTab::v_OnDestroy
21 eModel!CBrowserTab::v_WndProc
22 eModel!CBrowserTab::s_WndProc
23 user32!UserCallWinProcCheckWow
24 user32!DispatchClientMessage
25 user32!_fnDWORD
26 ntdll!KiUserCallbackDispatcherContinue
27 user32!NtUserDestroyWindow
28 eModel!CBrowserTab::_DoFinalCleanup
29 eModel!CBrowserTab::_OnConfirmedClose
2a eModel!CBrowserTab::OnClose
2b eModel!CTabWindow::_TabWindowThreadProc
2c eModel!LCIETab_ThreadProc
2d iertutil!IEGetTabWindowExports
2e kernel32!BaseThreadInitThunk
2f ntdll!RtlUserThreadStart
```

Another x64 example:

```
0:004> r
rax=0000000000000000 rbx=000000dd5253ce30 rcx=000000d515923a00
rdx=0000000000000000 rsi=0000000000000002 rdi=000000dd5253c940
rip=00007ffd1fbc98cc rsp=000000dd5253d060 rbp=000000dd5253d220
r8=000000d515923a00 r9=00000000fffffff r10=0000000000000000
r11=000000dd5253d3c8 r12=00007ffd20002460 r13=000000dd636a3101
r14=000000dd51738000 r15=0000000000000c00
iopl=0 nv up ei pl nz na pe nc
cs=0033 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010200
edgehtml!ComWindowProxy::PrivateAddRef+0x3c:
00007ffd`1fbc98cc 488b7868    mov     rdi,qword ptr [rax+68h] ds:00000000`00000068=?????????????????
```

```
0:004> k
# Child-SP          RetAddr           Call Site
00 000000dd`5253d060 00007ffd`1fbc9850 edgehtml!ComWindowProxy::PrivateAddRef+0x3c
01 000000dd`5253d090 00007ffd`1fbc97eb edgehtml!CEventPathBuilder::SetProxy+0x20
02 000000dd`5253d0c0 00007ffd`1fbcb79b edgehtml!CEventPathBuilder::AppendWindowTarget+0x47
03 000000dd`5253d0f0 00007ffd`1fbc5189 edgehtml!CWindow::BuildEventPath+0x1b
04 000000dd`5253d120 00007ffd`1faec7f1 edgehtml!CEventMgr::Dispatch+0x5c9
05 000000dd`5253d3d0 00007ffd`1faeb7cc edgehtml!CMessagePort::HandlePostMessage+0x10d
06 000000dd`5253d450 00007ffd`1fc4b667 edgehtml!CMessageDispatcher::ProcessNotification+0x5c
07 000000dd`5253d480 00007ffd`1fd521d1 edgehtml!GlobalWndOnPaintPriorityMethodCall+0x457
08 000000dd`5253d570 00007ffd`43a900dc edgehtml!GlobalWndProc+0x101
09 000000dd`5253d5f0 00007ffd`43a8fe52 user32!UserCallWinProcCheckWow+0x1fc
0a 000000dd`5253d6e0 00007ffd`43a9d3fe user32!DispatchClientMessage+0xa2
0b 000000dd`5253d740 00007ffd`462f5714 user32!_fnDWORD+0x3e
0c 000000dd`5253d7a0 00007ffd`43aaffba ntdll!KiUserCallbackDispatcherContinue
0d 000000dd`5253d828 00007ffd`43a8fc47 user32!NtUserDispatchMessage+0xa
0e 000000dd`5253d830 00007ffd`19c6eee8 user32!DispatchMessageWorker+0x247
0f 000000dd`5253d8b0 00007ffd`19cd0bdb eModel!CTabWindow::_TabWindowThreadProc+0x5b8
10 000000dd`5253fb10 00007ffd`3630864f eModel!LCIETab_ThreadProc+0x2bb
11 000000dd`5253fc40 00007ffd`45f42d92 iertutil!_IsoThreadProc_WrapperToReleaseScope+0x1f
12 000000dd`5253fc70 00007ffd`46269f64 kernel32!BaseThreadInitThunk+0x22
13 000000dd`5253fcfa0 00000000`00000000 ntdll!RtlUserThreadStart+0x34
```

## O

### Object Distribution Anomaly

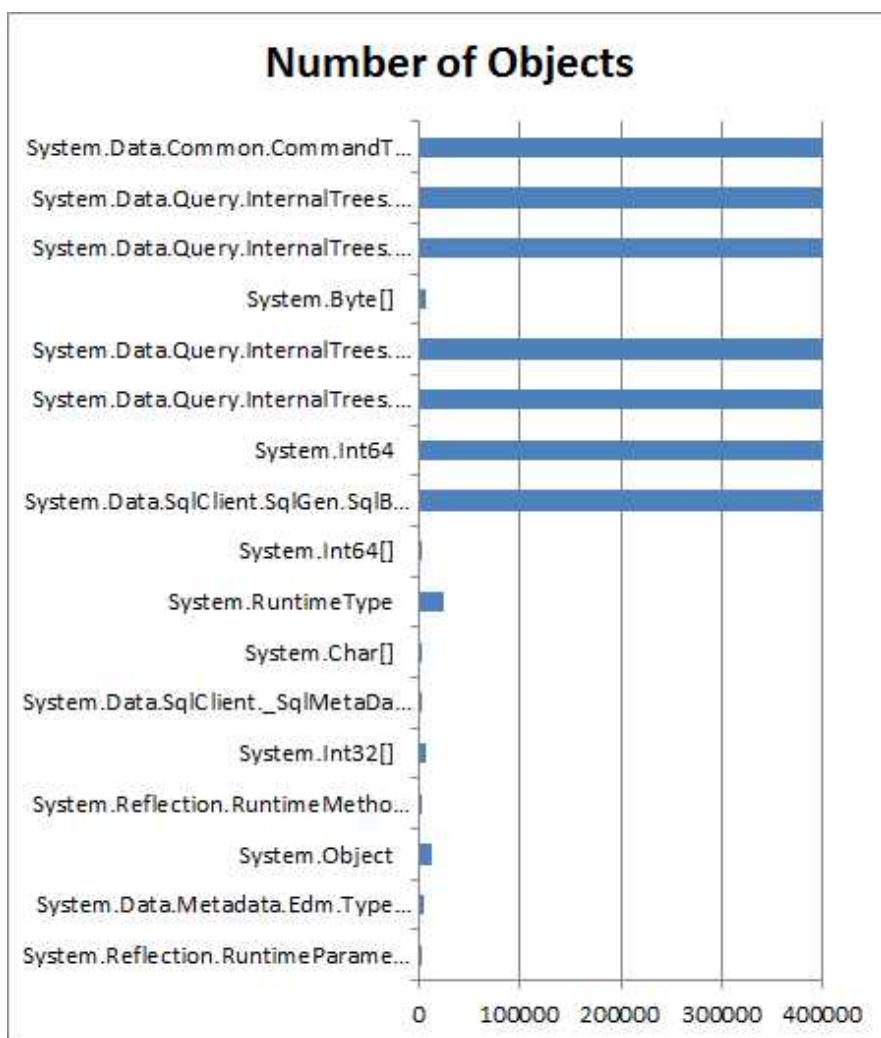
#### .NET Heap

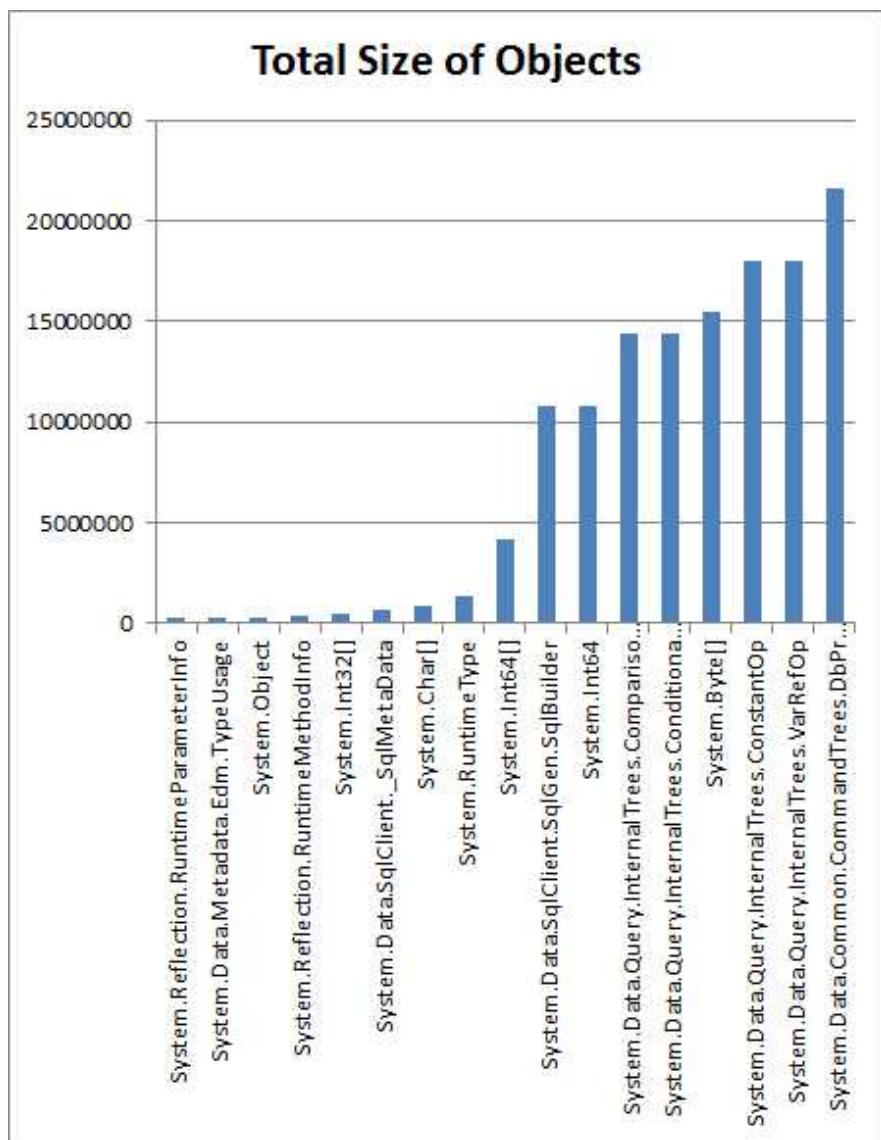
Sometimes we notice the anomalies in object distribution in heaps and pools ([IRP](#), page 759). Memory consumption may be high in case of big objects. Such anomalies may point to possible memory (page 526), handle (page 416), and object leaks. But it may also be a temporary condition ([Memory Fluctuation](#), page 634) due to a large amount of queued or postponed work that can be solved by proper software configuration. Diagnosed anomalies may also direction troubleshooting efforts if they cluster around the certain component(s) or specific functionality. The distribution can be assessed by both the total memory consumption and the total number of objects of a particular class.

Here's an example of **Object Distribution Anomaly** analysis pattern from .NET heap. The output of **!DumpHeap -stat** WinDbg SOS extension command shows the abnormal distribution of objects related to SQL data queries:

```
Count TotalSize
[...]
2342    281040  System.Reflection.RuntimeParameterInfo
3868    309440  System.Data.Metadata.Edm.TypeUsage
13218   317232  System.Object
3484    390208  System.Reflection.RuntimeMethodInfo
6092    508044  System.Int32[]
2756    617344  System.Data.SqlClient._SqlMetaData
2770    822870  System.Char[]
24560   1375360 System.RuntimeType
18      4195296 System.Int64[]
449691  10792584 System.Data.SqlClient.SqlGen.SqlBuilder
449961  10799064 System.Int64
449691  14390112 System.Data.Query.InternalTrees.ComparisonOp
449695  14390240 System.Data.Query.InternalTrees.ConditionalOp
6360    15509435 System.Byte[]
449690  17987600 System.Data.Query.InternalTrees.ConstantOp
449938  17997520 System.Data.Query.InternalTrees.VarRefOp
450898  21643104 System.Data.Common.CommandTrees.DbPropertyExpression
[...]
```

The anomalous character of the distribution is also illustrated in the following diagrams:

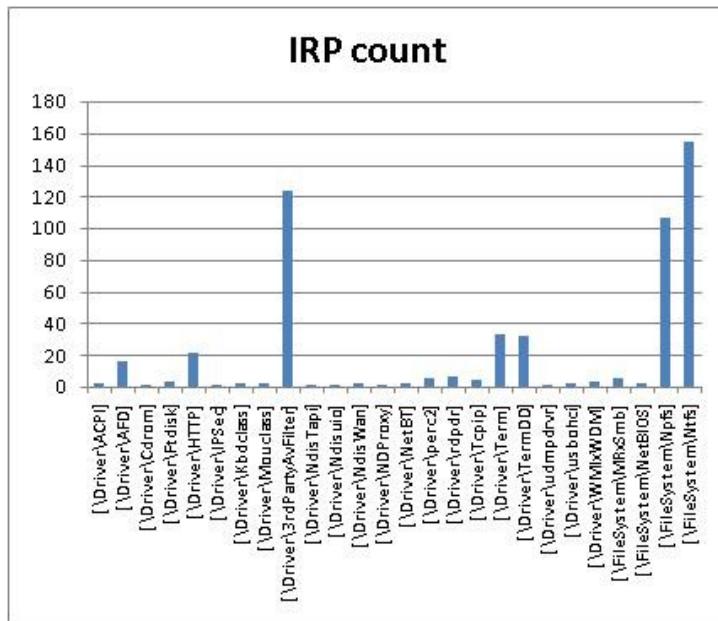




## IRP

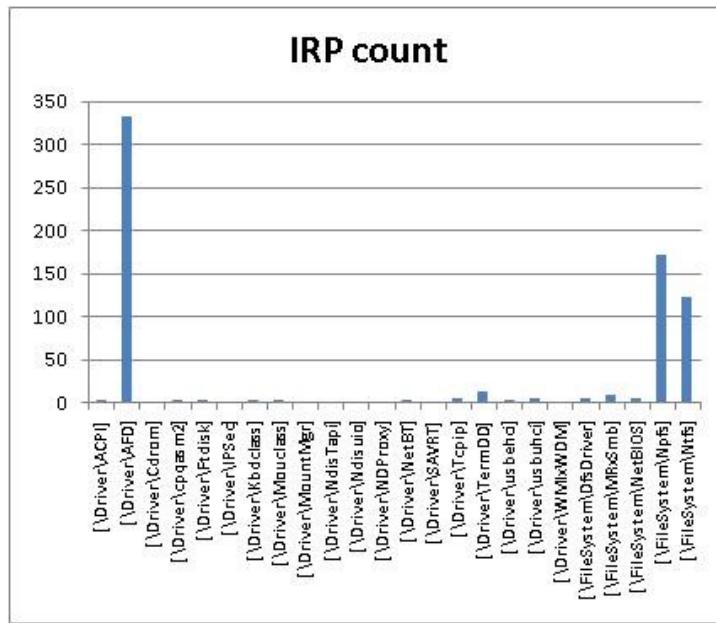
In the kernel or complete memory dumps coming from hanging or slow workstations and servers **!irpfind** WinDbg command may show **IRP Object Distribution Anomaly** pattern when certain drivers have an excessive count of active IRPs not observed under normal circumstances. We created two IRP distribution graphs from two problem kernel dumps by preprocessing command output using Visual Studio keyboard macros to eliminate completed IRPs and then using Excel. In one case it was a big number of I/O request packets from 3rd-party antivirus filter driver:

\Driver\3rdPartyAvFilter

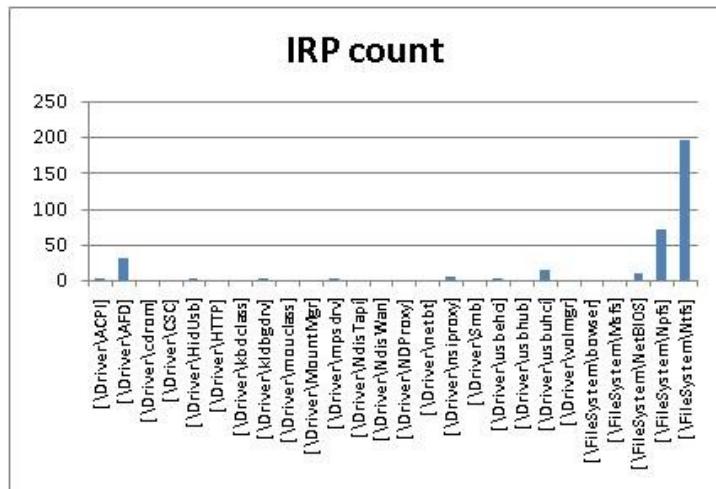


In the second case it was the huge number of active IRPs targeted to kernel socket ancillary function driver:

\Driver\AFD



Two other peaks on both graphs are related to NTPS and NTFS, pipes and file system and usually normal. Here is IRP distribution graph from our Vista workstation captured while we were first writing about this pattern:



### Comment

This pattern was previously called **IRP Distribution Anomaly**.

## OMAP Code Optimization

**OMAP Code Optimization** is used to make the code that needs to be present in memory smaller. So instead of flat address space for compiled function, we have pieces of it scattered here and there. This leads to ambiguity when we try to disassemble OMAP code at its address because WinDbg doesn't know whether it should treat address range as a function offset (starting from the beginning of the function source code) or just a memory layout offset (starting from the address of that function). We illustrate the pattern using *IoCreateDevice* function code.

Let's first evaluate a random address starting from the first address of the function (memory layout offset):

```
kd> ? nt!IoCreateDevice
Evaluate expression: -8796073668256 = ffffff800`01275160

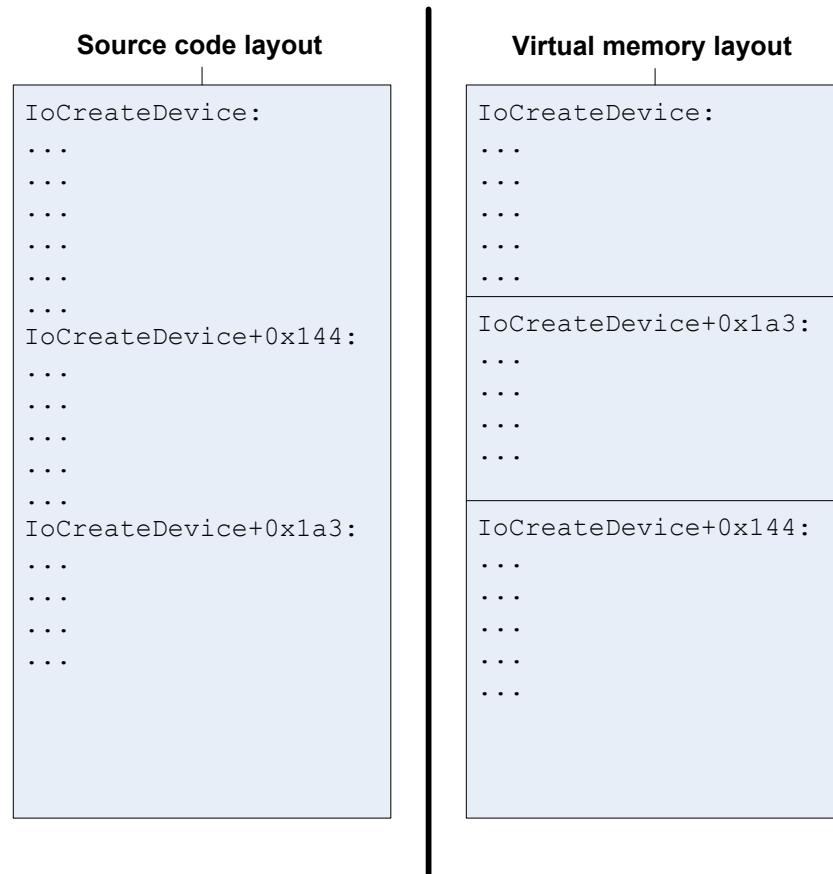
kd> ? nt!IoCreateDevice+0x144
Evaluate expression: -8796073667932 = ffffff800`012752a4

kd> ? ffffff800`012752a4-fffff800`01275160
Evaluate expression: 324 = 00000000`00000144
```

If we try to disassemble code at the same address, the expression will also be evaluated as the memory layout offset:

```
kd> u nt!IoCreateDevice+0x144
nt!IoCreateDevice+0x1a3:
fffff800`012752a4 or      eax,10h
fffff800`012752a7 mov     dword ptr [rsp+0B0h],eax
fffff800`012752ae test    ebp,ebp
fffff800`012752b0 mov     ebx,ebp
fffff800`012752b2 jne     nt!IoCreateDevice+0x1b3
fffff800`012752b8 add     ebx,dword ptr [rsp+54h]
fffff800`012752bc mov     rdx,qword ptr [nt!IoDeviceObjectType]
fffff800`012752c3 lea     rcx,[rsp+88h]
```

We see the difference: we give +0x144 offset, but the code is shown from +0x1a3! This is because OMAP optimization moved the code from the function offset +0x1a3 to memory locations starting from +0x144. The following picture illustrates this:



If we see this when disassembling a *function name+offset* address from a thread stack trace we can use a raw address instead:

```

kd> k
Child-SP          RetAddr          Call Site
fffffadf`e3a18d30 fffff800`012b331e component!function+0x72
fffffadf`e3a18d70 fffff800`01044196 nt!PspSystemThreadStartup+0x3e
fffffadf`e3a18dd0 00000000`00000000 nt!KxStartSystemThread+0x16

kd> u fffff800`012b331e
nt!PspSystemThreadStartup+0x3e:
fffff800`012b331e nop
fffff800`012b331f test    byte ptr [rbx+3FCh],40h
fffff800`012b3326 jne     nt!PspSystemThreadStartup+0x4c
fffff800`012b332c mov     rax,qword ptr gs:[188h]
fffff800`012b3335 cmp     rbx,rax
fffff800`012b3338 jne     nt!PspSystemThreadStartup+0x10c
fffff800`012b333e or      dword ptr [rbx+3FCh],1
fffff800`012b3345 xor     ecx,ecx

```

We also see OMAP in action when we try to disassemble the function body using **uf** command:

```
kd> uf nt!IoCreateDevice
nt!IoCreateDevice+0x34d:
fffff800`0123907d or      dword ptr [rdi+30h],8
fffff800`01239081 jmp    nt!IoCreateDevice+0x351
...
...
...
nt!IoCreateDevice+0x14c:
fffff800`0126f320 mov    r14w,200h
fffff800`0126f325 jmp    nt!IoCreateDevice+0x158
nt!IoCreateDevice+0x3cc:
fffff800`01270bd0 lea    rax,[rdi+50h]
fffff800`01270bd4 mov    qword ptr [rax+8],rax
fffff800`01270bd8 mov    qword ptr [rax],rax
fffff800`01270bdb jmp    nt!IoCreateDevice+0x3d7
nt!IoCreateDevice+0xa4:
fffff800`01273eb9 mov    r8d,1
fffff800`01273ebf lea    rdx,[nt!`string']
fffff800`01273ec6 lea    rcx,[rsp+0D8h]
fffff800`01273ece xadd  dword ptr [nt!IopUniqueDeviceObjectNumber],r8d
fffff800`01273ed6 inc    r8d
fffff800`01273ed9 call   nt!swprintf
fffff800`01273ede test   r13b,r13b
fffff800`01273ee1 jne    nt!IoCreateDevice+0xce
...
...
...
```

Another example of **OMAP Optimization** when we try to disassemble backward:

```
ChildEBP RetAddr Args to Child
0006f87c 01034efb application!MultiUserLogonAttempt+0x5ba
0006fee4 01037120 application!LogonAttempt+0x406

1: kd> ub application!LogonAttempt+0x406
^ Unable to find valid previous instruction for 'ub application!LogonAttempt+0x406'

1: kd> u application!LogonAttempt+0x406
application!LogonAttempt+0x20b:
010348d5 add dword ptr [edx+10h],ebp
```

We have to specify the return address for *MultiUserLogonAttempt* then:

```
1: kd> ub 01034efb
application!LogonAttempt+0x3e4:
01034ed9 mov  dword ptr [ebp-628h],ecx
01034edf mov  ecx,dword ptr [ebp-61Ch]
01034ee5 mov  dword ptr [eax+24h],ecx
01034ee8 push dword ptr [ebp-61Ch]
01034eee lea   eax,[ebp-638h]
01034ef4 push eax
01034ef5 push ebx
01034ef6 call application!MultiUserLogonAttempt (0102c822)

1: kd> u 01034efb
application!LogonAttempt+0x406:
01034efb mov  ecx,dword ptr [ebp-628h]
01034f01 mov  dword ptr [ebp-608h],eax
01034f07 mov  eax,dword ptr [ebx+8]
01034f0a mov  dword ptr [eax+24h],ecx
01034f0d cmp  dword ptr [application!g_SessionId (010742dc)],0
01034f14 je   application!LogonAttempt+0x47e (01034f73)
01034f16 lea   eax,[ebx+1078h]
01034f1c push eax
```

## Comments

See also Microsoft article about debugging performance-optimized code<sup>160</sup>.

---

<sup>160</sup> <https://msdn.microsoft.com/en-us/library/windows/hardware/ff541382>

## One-Thread Process

Processes with one thread like Notepad are rare. Such a process is always suspicious especially if it is a service or belongs to a complex product. Usually, this happens when all other threads terminated, and the remaining thread is blocked in some wait chain. For example, this process has a thread which is blocked (**Blocked Queue** pattern, page 77) in an ALPC request to itself (the same process):

```
0: kd> !process ffffffa8013ed9b30 3f
PROCESS ffffffa8013ed9b30
SessionId: 0 Cid: 44b4 Peb: 7fffffd8000 ParentCid: 0114
DirBase: 2da448000 ObjectTable: ffffff8a01948c670 HandleCount: 660.
Image: ServiceA.exe
VadRoot ffffffa801356dd10 Vads 398 Clone 0 Private 5795. Modified 204253. Locked 0.
DeviceMap ffffff8a0000008340
Token ffffff8a01b546060
ElapsedTime 01:32:37.622
UserTime 00:00:01.421
KernelTime 00:00:01.578
QuotaPoolUsage[PagedPool] 0
QuotaPoolUsage[NonPagedPool] 0
Working Set Sizes (now,min,max) (1525, 50, 345) (6100KB, 200KB, 1380KB)
PeakWorkingSetSize 7607
VirtualSize 178 Mb
PeakVirtualSize 182 Mb
PageFaultCount 752709
MemoryPriority BACKGROUND
BasePriority 8
CommitCharge 8043

THREAD ffffffa8012caab50 Cid 44b4.4f70 Teb: 000007fffff5a000 Win32Thread: 0000000000000000 WAIT:
(WrLpcReply) KernelMode Non-Alertable
fffffa8012caaf18 Semaphore Limit 0x1
Waiting for reply to ALPC Message ffffff8a0194d4780 : queued at port ffffffa8012911c80 : owned by
process fffffa8013ed9b30
IRP List:
fffffa8013923300: (0006,0118) Flags: 00060000 Mdl: 00000000
Not impersonating
DeviceMap ffffff8a000008340
Owning Process ffffffa8013ed9b30 Image: ServiceA.exe
Attached Process N/A Image: N/A
Wait Start TickCount 139828 Ticks: 347372 (0:01:30:27.687)
Context Switch Count 7380
UserTime 00:00:00.031
KernelTime 00:00:04.890
Win32 Start Address ServiceA (0x00000001401156e0)
Stack Init fffff88014c9ddb0 Current fffff88014c9c6b0
Base fffff88014c9e000 Limit fffff88014c98000 Call 0
Priority 8 BasePriority 8 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
Child-SP RetAddr Call Site
fffff880`14c9c6f0 ffffff800`01873652 nt!KiSwapContext+0x7a
fffff880`14c9c830 ffffff800`01884a9f nt!KiCommitThreadWait+0x1d2
fffff880`14c9c8c0 ffffff800`0189f04f nt!KeWaitForSingleObject+0x19f
fffff880`14c9c960 ffffff800`01b919f6 nt!AlpcpSignalAndWait+0x8f
```

```
fffff880`14c9ca10 fffff800`01b910f0 nt!AlpcpReceiveSynchronousReply+0x46
fffff880`14c9ca70 fffff800`01b9519d nt!AlpcpProcessSynchronousRequest+0x33d
fffff880`14c9cbb0 fffff800`01b95276 nt!LpcpRequestWaitReplyPort+0x9c
fffff880`14c9cc10 fffff800`0187ced3 nt!NtRequestWaitReplyPort+0x76
fffff880`14c9cc60 fffff800`01879490 nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @
fffff880`14c9cc60)
fffff880`14c9cdf8 fffff880`05c31050 nt!KiServiceLinkage
fffff880`14c9ce70 fffff880`045ce005 ModuleA+0x12468
[...]
fffff880`14c9da10 fffff800`01b9d3b6 nt!IopXxxControlFile+0x607
fffff880`14c9db40 fffff800`0187ced3 nt!NtDeviceIoControlFile+0x56
fffff880`14c9dbb0 00000000`76d8138a nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @
fffff880`14c9dc20)
00000000`082af028 000007fe`fd366cf6 ntdll!NtDeviceIoControlFile+0xa
00000000`082af030 00000000`76c2683f KERNELBASE!TlsGetValue+0x1a36
00000000`082af0a0 00000001`4019d38c kernel32!DeviceIoControlImplementation+0x7f
[...]
```

## Optimized Code

In a case of **Optimized Code**, we should not trust our crash dump analysis tools like WinDbg. We advise always suspecting that the compiler generated code is optimized if we see any suspicious or strange behavior of the tool. Let's consider this stack trace fragment:

```
Args to Child
77e44c24 000001ac 00000000 ntdll!KiFastSystemCallRet
000001ac 00000000 00000000 ntdll!NtFsControlFile+0xc
00000034 0000bb8 0013e3f4 kernel32!WaitNamedPipeW+0x2c3
0016fc60 00000000 67c14804 MyModule!PipeCreate+0x48
```

Third party function *PipeCreate* from *MyModule* opens a named pipe, and its first parameter (0016fc60) points to a pipe name L"\.\pipe\MyPipe". Inside the source code, it calls Win32 API function *WaitNamedPipeW* to wait for the pipe to be available for connection) and passes the same pipe name. But we see that the first parameter to *WaitNamedPipeW* is 00000034 which cannot be the pointer to a valid Unicode string. And the program should have been crashed if 00000034 were a pointer value.

Everything becomes clear if we look at *WaitNamedPipeW* disassembly:

```
0:000> uf kernel32!WaitNamedPipeW
mov    edi,edi
push   ebp
mov    ebp,esp
sub    esp,50h
push   dword ptr [ebp+8] ; Use pipe name
lea    eax,[ebp-18h]
push   eax
call   dword ptr [kernel32!_imp__RtlCreateUnicodeString (77e411c8)]
...
...
...
call   dword ptr [kernel32!_imp__NtOpenFile (77e41014)]
cmp    dword ptr [ebp-4],edi
mov    esi,eax
jne   kernel32!WaitNamedPipeW+0x1d5 (77e93316)
cmp    esi,edi
j1    kernel32!WaitNamedPipeW+0x1ef (77e93331)
movzx  eax,word ptr [ebp-10h]
mov    ecx,dword ptr fs:[18h]
add    eax,0Eh
push   eax
push   dword ptr [kernel32!BaseDllTag (77ecd14c)]
mov    dword ptr [ebp+8],eax ; reuse parameter slot
```

As we know [ebp+8] is the first function parameter for non-FPO calls and we see it reused because after converting LPWSTR to UNICODE\_STRING and calling *NtOpenFile* to get a handle we no longer need our parameter slot and the compiler can reuse it to store other information.

There is another compiler optimization we should be aware of, and it is called **OMAP** (page 756). It moves the code inside the code section and puts the most frequently accessed code fragments together. In that case, if we type in WinDbg, for example,

```
0:000> uf nt!someFunction
```

we get a different code than if we type (assuming f4794100 is the address of the function we obtained from stack trace or disassembly)

```
0:000> uf f4794100
```

In conclusion, the advice is to be alert and conscious during crash dump analysis and inspect any inconsistencies closer.

---

## Comments

Another example of the parameter reuse in kernel mode can be found in this article<sup>161</sup>.

---

<sup>161</sup> [http://www.codemachine.com/article\\_paramreuse.html](http://www.codemachine.com/article_paramreuse.html)

## Optimized VM Layout

This pattern is a specialization of the general **Changed Environment** pattern (page 114) where the whole modules are moved in virtual memory by changing their load order and addresses. This can result in dormant bugs being exposed, and one of the workarounds usually is to disable such external optimization programs and services or adding applications that behave improperly to corresponding exclusion lists. Some optimized virtual memory cases can be easily detected by looking at module list where system DLLs are remapped to lower addresses instead of 0x7X000000 range:

```
0:000> lm
start   end     module name
00400000 00416000 Application
00470000 0050b000 advapi32
00520000 00572000 shlwapi
02340000 023cb000 oleaut32
04b80000 0523e000 System_Data_ni
1a400000 1a524000 urlmon
4dd60000 4df07000 GdiPlus
5f120000 5f12e000 ntlanman
5f860000 5f891000 netui1
5f8a0000 5f8b6000 netui0
637a0000 63d28000 System_Xml_ni
64890000 6498c000 System_Configuration_ni
64e70000 6515c000 System_Data
65ce0000 65ecc000 System_Web_Services_ni
71bd0000 71be1000 mpr
71bf0000 71bf8000 ws2help
71c00000 71c17000 ws2_32
71c20000 71c32000 tsappcmp
71c40000 71c97000 netapi32
73070000 73097000 winspool
75e90000 75e97000 drprov
75ea0000 75eaa000 davclnt
76190000 761a2000 msasn1
761b0000 76243000 crypt32
76a80000 76a92000 atl
76b80000 76bae000 credui
76dc0000 76de8000 adsldpc
76df0000 76e24000 activeds
76f00000 76f08000 wtsapi32
76f10000 76f3e000 wldap32
771f0000 77201000 winsta
77670000 777a9000 ole32
77ba0000 77bfa000 msvcrt
78130000 781cb000 msvcr80
79000000 79046000 mscoree
79060000 790b6000 mscojit
790c0000 79bf6000 mscorlib_ni
79e70000 7a3ff000 mscorwks
7a440000 7ac2a000 System_ni
7ade0000 7af7c000 System_Drawing_ni
7afdf0000 7bc6c000 System_Windows_Forms_ni
7c340000 7c396000 msrvcr71
```

7c8d0000	7d0ce000	shell32
7d4c0000	7d5f0000	kernel32
7d600000	7d6f0000	ntdll
7d800000	7d890000	gdi32
7d8d0000	7d920000	secur32
7d930000	7da00000	user32
7da20000	7db00000	rpcrt4
7dbd0000	7dc3000	comctl32
7df50000	7dfc0000	uxtheme
7e020000	7e02f000	samlib

The similar address space reshuffling happens with ASLR-enabled applications<sup>162</sup> with the difference that system modules are never remapped below 0x70000000 address.

---

<sup>162</sup> ASLR: Address Space Layout Randomization, Memory Dump Analysis Anthology, Volume 1, page 674

## Origin Module

**Origin Module** is a module that may have originated the problem behavior. For example, when we look at **Stack Trace** (page 926) we may skip **Top Modules** (page 1012) due to our knowledge of the product, for example, if they are not known as **Problem Modules** (page 812) or known as **Well-Tested Modules** (page 1147). In the case of **Truncated Stack Traces** (page 1015), we may designate bottom modules as possible problem origins. For example, for **Reference Leak** pattern example (page 830) we may consider checking reference counting for selected modules such as *ModuleA* and *ModuleB*:

```
ad377ae8 +1 Dflt nt! ?? ::FNODOBFM::`string'+18f1d
nt!ObpCallPreOperationCallbacks+4e
nt!ObpPreInterceptHandleCreate+af
nt! ?? ::NNGAKEGL::`string'+2c31f
nt!ObOpenObjectByPointerWithTag+109
nt!PsOpenProcess+1a2
nt!NtOpenProcess+23
nt!KiSystemServiceCopyEnd+13
nt!KiServiceLinkage+0
ModuleA+dca63
ModuleA+b5bc
ModuleA+c9c2e
ModuleA+bae56
ModuleA+b938d
ModuleA+c0ec6
ModuleA+afce7

ad377aeb -1 Dflt nt! ?? ::FNODOBFM::`string'+4886e
nt!ObpCallPreOperationCallbacks+277
nt!ObpPreInterceptHandleCreate+af
nt! ?? ::NNGAKEGL::`string'+2c31f
nt!ObOpenObjectByPointerWithTag+109
nt!PsOpenProcess+1a2
nt!NtOpenProcess+23
nt!KiSystemServiceCopyEnd+13
nt!KiServiceLinkage+0
ModuleA+dca63
ModuleA+b5bc
ModuleA+c9c2e
ModuleA+bae56
ModuleA+b938d
ModuleA+c0ec6
ModuleA+afce7

ad377af7 +1 Dflt nt! ?? ::NNGAKEGL::`string'+1fb41
nt!ObReferenceObjectByHandle+25
ModuleA+dcade
ModuleA+b5bc
ModuleA+c9c2e
ModuleA+bae56
ModuleA+b938d
ModuleA+c0ec6
ModuleA+afce7
```

```
ModuleA+87ca
ModuleA+834a
ModuleA+a522c
ModuleA+a51b6
ModuleA+a4787
ModuleB+19c0c
ModuleB+19b28

ad377afa -1 Dflt nt! ?? ::FNODOBFM::`string'+4886e
ModuleA+dcbbe
ModuleA+b5bc
ModuleA+c9c2e
ModuleA+bae56
ModuleA+b938d
ModuleA+c0ec6
ModuleA+afce7
ModuleA+87ca
ModuleA+834a
ModuleA+a522c
ModuleA+a51b6
ModuleA+a4787
ModuleB+19c0c
ModuleB+19b28
ModuleB+b652
```

## Out-of-Module Pointer

This pattern is about pointers to addresses outside the container module range. A typical example here would be some kernel table or structure, for example, a driver IRP dispatch table having pointers to outside that driver module address range. Other examples may include 32-bit SSDT pointing outside *nt* module range and IDT entries pointing outside *hal* and expected drivers:

```
[...]
818809dc 8193c4e7 nt!NtQueryOpenSubKeys
818809e0 8193c76b nt!NtQueryOpenSubKeysEx
818809e4 81a909b0 nt!NtQueryPerformanceCounter
818809e8 819920e7 nt!NtQueryQuotaInformationFile
818809ec 819e34f2 nt!NtQuerySection
818809f0 819f470b nt!NtQuerySecurityObject
818809f4 81a882fe nt!NtQuerySemaphore
818809f8 819eff54 nt!NtQuerySymbolicLinkObject
818809fc 81a8a223 nt!NtQuerySystemEnvironmentValue
81880a00 81a8a831 nt!NtQuerySystemEnvironmentValueEx
81880a04 96ca1a73
81880a08 81a7ac06 nt!NtQuerySystemTime
81880a0c 81a8f913 nt!NtQueryTimer
81880a10 81a7aeeb nt!NtQueryTimerResolution
81880a14 8193985a nt!NtQueryValueKey
81880a18 819e9273 nt!NtQueryVirtualMemory
81880a1c 8199274e nt!NtQueryVolumeInformationFile
81880a20 81a1a655 nt!NtQueueApcThread
[...]

0: kd> lm m nt
start end module name
81800000 81ba1000 nt
```

Such pointers may also be **Raw Pointers** (page 828), but it also could be the case that all pointers are raw in the absence of symbols with only a few outside of the expected range.

## Overaged System

Software aging can be the cause of some problems. Sometimes a look at the following WinDbg output can give irresistible temptation to suggest periodic reboots:

```
Debug session time: Wed April 28 15:36:52.330 2008 (GMT+0)
System Uptime: 124 days 6:27:16.658
```

---

## Comments

One of the readers provided us with an example of virtual address fragmentation on their **Overaged System**:

```
0:020> !heap -s
Heap Flags Reserv Commit Virt Free List UCR Virt Lock Fast
(k) (k) (k) (k) length blocks cont. heap
-----
006c0000 00001002 2565820 7232 658924 1302 160 200 0 44f3b4 LFH
External fragmentation 18 % (160 free blocks)
Virtual address fragmentation 98 % (200 uncommitted ranges)
```

# P

## Packed Code

**Packed Code** is a frequent ingredient of armored malware. Here we demonstrate a few WinDbg commands to detect UPX packed modules with little or no expected strings:

```
0:000> !dh 00000000`00fd40b0

File Type: DLL
FILE HEADER VALUES
14C machine (i386)
3 number of sections
time date stamp Fri Jan 18 21:27:25 2013

0 file pointer to symbol table
0 number of symbols
E0 size of optional header
2102 characteristics
Executable
32 bit word machine
DLL

OPTIONAL HEADER VALUES
10B magic #
11.00 linker version
6000 size of code
1000 size of initialized data
F000 size of uninitialized data
15600 address of entry point
10000 base of code
----- new -----
0000000010000000 image base
1000 section alignment
200 file alignment
2 subsystem (Windows GUI)
6.00 operating system version
0.00 image version
6.00 subsystem version
17000 size of image
1000 size of headers
0 checksum
0000000000100000 size of stack reserve
000000000001000 size of stack commit
0000000000100000 size of heap reserve
0000000000001000 size of heap commit
140 DLL characteristics
Dynamic base
NX compatible
16274 [      AC] address [size] of Export Directory
161DC [      98] address [size] of Import Directory
16000 [     1DC] address [size] of Resource Directory
```

```
0 [      0] address [size] of Exception Directory
0 [      0] address [size] of Security Directory
16320 [     10] address [size] of Base Relocation Directory
0 [      0] address [size] of Debug Directory
0 [      0] address [size] of Description Directory
0 [      0] address [size] of Special Directory
0 [      0] address [size] of Thread Storage Directory
157CC [     48] address [size] of Load Configuration Directory
0 [      0] address [size] of Bound Import Directory
0 [      0] address [size] of Import Address Table Directory
0 [      0] address [size] of Delay Import Directory
0 [      0] address [size] of COR20 Header Directory
0 [      0] address [size] of Reserved Directory
```

**SECTION HEADER #1**

**UPX0 name**  
F000 virtual size  
1000 virtual address  
0 size of raw data  
400 file pointer to raw data  
0 file pointer to relocation table  
0 file pointer to line numbers  
0 number of relocations  
0 number of line numbers  
E0000080 flags  
Uninitialized Data  
(no align specified)  
Execute Read Write

**SECTION HEADER #2**

**UPX1 name**  
6000 virtual size  
10000 virtual address  
5A00 size of raw data  
400 file pointer to raw data  
0 file pointer to relocation table  
0 file pointer to line numbers  
0 number of relocations  
0 number of line numbers  
E0000040 flags  
Initialized Data  
(no align specified)  
Execute Read Write

**SECTION HEADER #3**

**.rsrc name**  
1000 virtual size  
16000 virtual address  
400 size of raw data  
5E00 file pointer to raw data  
0 file pointer to relocation table  
0 file pointer to line numbers  
0 number of relocations  
0 number of line numbers  
C0000040 flags  
Initialized Data

```
(no align specified)
Read Write

0:000> s-sa 00000000`00fd40b0 L6600
00000000`00fd40fd  "!This program cannot be run in D"
00000000`00fd411d  "OS mode."
00000000`00fd4188  "Rich"
00000000`00fd4290  "UPX0"
00000000`00fd42b8  "UPX1"
00000000`00fd42e0  ".rsrc"
00000000`00fd448b  "3.08"
00000000`00fd4490  "UPX!_"
00000000`00fd449b  "YhHM4"
00000000`00fd44d1  "vqx"
[...]
```

Such in-memory modules (not yet initialized by a loader) can be saved to disk using **.writemem** command and unpacked. Once loaded and relocated to some address they still have UPX sections, but they now have more strings:

```
0:000> s-sa 00000000`691c0000 L300
00000000`691c004d  "!This program cannot be run in D"
00000000`691c006d  "OS mode."
00000000`691c00d8  "Rich"
00000000`691c01e0  "UPX0"
00000000`691c0207  "`UPX1"
00000000`691c022f  "`rsrc"
[...]
00000000`691d620b  "uGC"
00000000`691d621c  "KERNEL32.DLL"
00000000`691d622a  "LoadLibraryA"
00000000`691d6238  "GetProcAddress"
00000000`691d6248  "VirtualProtect"
00000000`691d6258  "VirtualAlloc"
00000000`691d6266  "VirtualFree"
[...]

0:000> s-su 00000000`691c0000 L(00000000`691d7000-00000000`691c0000)
[...]
00000000`691c8178  "http://www.patterndiagnostics.com"
00000000`691c8260  "mscoree.dll"
[...]
```

## Paged Out Data

One analysis problem that happens frequently is the absence of stack traces due to kernel stack pages being paged-out and, therefore, not present in a complete memory dump that only contains physical memory. This shouldn't be a problem for kernel or process memory dumps because they contain virtual memory. The problem usually manifests itself either on **Busy Systems** (page 99) utilizing almost all physical memory (**Insufficient Memory**, page 552) or on **Overaged Systems** (page 774) where certain processes hadn't been used for a long time. It could also be the case when a problem happened some time ago and only diagnosed much later. For example, this LPC **Wait Chain** (page 1097) for **Coupled Processes** (page 149) happened to be 2 days ago before the dump was saved:

```
0: kd> !thread ffffffadfcf9e8bf0 1f
THREAD ffffffadfcf9e8bf0 Cid 61f0.2c70 Teb: 000007fffffd000 Win32Thread: fffff97ff381a480 WAIT:
(Unknown) UserMode Non-Alertable
    ffffffadfcf9e8f58 Semaphore Limit 0x1
Waiting for reply to LPC MessageId 01e2cb39:
Current LPC port ffffffa800e5a9d10
Impersonation token: ffffffa80039cd050 (Level Impersonation)
Owning Process ffffffadffc7c7c20
Image: applicationA.exe
Wait Start TickCount      12018444
Ticks: 11312740 (2:01:06:01.562)
Context Switch Count      456          LargeStack
UserTime                  00:00:00.046
KernelTime                00:00:00.078
Start Address applicationA (0x0000000100061411)
Stack Init fffffadc125d4e00 Current fffffadc125d48e0
Base fffffadc125d5000 Limit fffffadc125cc000 Call 0
Priority 15 BasePriority 15 PriorityDecrement 0
Kernel stack not resident.

0: kd> !lpc message 01e2cb39
Searching message 1e2cb39 in threads ...
    Server thread fffffadff93c5bf0 is working on message 1e2cb39
```

```
0: kd> !thread fffffadff93c5bf0 1f
THREAD fffffadff93c5bf0 Cid 0218.5130 Teb: 000007ffffcbc000 Win32Thread: 0000000000000000 WAIT:
(Unknown) UserMode Non-Alertable
    fffffadff6c71c70 SynchronizationEvent
Impersonation token: fffffa803bde5060 (Level Impersonation)
Owning Process fffffadcde439280
Image: applicationB.exe
Wait Start TickCount 12018444
Ticks: 11312740 (2:01:06:01.562)
Context Switch Count 12
UserTime 00:00:00.000
KernelTime 00:00:00.000
Win32 Start Address 0x0000000001e2cb39
LPC Server thread working on message Id 1e2cb39
Start Address kernel32 (0x0000000077d6b6a0)
Stack Init fffffadc28b19e00 Current fffffadc28b19950
Base fffffadc28b1a000 Limit fffffadc28b14000 Call 0
Priority 14 BasePriority 13 PriorityDecrement 0
Kernel stack not resident.
```

One of the tricks we can recommend in such cases is to save user dumps of processes that could possibly be paged out before forcing a complete memory dump.

## Parameter Flow

Sometimes we identify a function parameter that caused abnormal behavior, for example, **Invalid Pointer** (page 589) access violation and want to see from which module or function it was probably originated:

```
0:009> kvL
# ChildEBP RetAddr Args to Child
00 0381f048 74e715e9 00000002 0381f098 00000001 ntdll!NtWaitForMultipleObjects+0x15
01 0381f0e4 76ce19fc 0381f098 0381f10c 00000000 KERNELBASE!WaitForMultipleObjectsEx+0x100
02 0381f12c 76ce41d8 00000002 7efde000 00000000 kernel32!WaitForMultipleObjectsExImplementation+0xe0
03 0381f148 76d08074 00000002 0381f17c 00000000 kernel32!WaitForMultipleObjects+0x18
04 0381f1b4 76d07f33 0381f294 00000001 00000001 kernel32!WerReportFaultInternal+0x186
05 0381f1c8 76d07828 0381f294 00000001 0381f264 kernel32!WerReportFault+0x70
06 0381f1d8 76d077a7 0381f294 00000001 9a131a15 kernel32!BaseReportFault+0x20
07 0381f264 772574ff 00000000 772573dc 00000000 kernel32!UnhandledExceptionFilter+0x1af
08 0381f26c 772573dc 00000000 0381ffd4 7720c550 ntdll!_RtlUserThreadStart+0x62
09 0381f280 77257281 00000000 00000000 00000000 ntdll!_EH4_CallFilterFunc+0x12
0a 0381f2a8 7723b499 fffffffe 0381ffc4 0381f3e4 ntdll!_except_handler4+0x8e
0b 0381f2cc 7723b46b 0381f394 0381ffc4 0381f3e4 ntdll!ExecuteHandler2+0x26
0c 0381f2f0 7723b40e 0381f394 0381ffc4 0381f3e4 ntdll!ExecuteHandler+0x24
0d 0381f37c 771f0133 0181f394 0381f3e4 0381f394 ntdll!RtlDispatchException+0x127
0e 0381f37c 00010101 0181f394 0381f3e4 0381f394 ntdll!KiUserExceptionDispatcher+0xf (CONTEXT @ 0381f3e4)
WARNING: Frame IP not in any known module. Following frames may be wrong.
0f 0381f844 00df3d1e 064bf018 0b925970 06501dd8 0x10101
10 0381f860 00ca0a15 064bf018 0b925970 0c86a949 ModuleA!Bar+0x3e
11 0381f888 00ca07d4 064bf018 0b925970 09584580 ModuleB!Foo2+0x75
12 0381f8e4 00ca0ae3 0b925970 00000000 09584008 ModuleB!Foo1+0xb4
[...]
```

Here we backtrack the invalid function call pointer to a function argument parameter:

```
0:009> .cxr 0381f3e4
eax=06501dd8 ebx=0381f8c4 ecx=0b925970 edx=0b925527 esi=064bf018 edi=0381f898
eip=00010101 esp=0381f848 ebp=0c86a949 iopl=0 nv up ei pl nz na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010206
00010101 1f pop ds

0:009> k 1
*** Stack trace for last set context - .thread/.cxr resets it
# ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
00 0381f844 00df3d1e 0x10101

0:009> ub 00df3d1e
ModuleA!Bar+0x2e:
00df3d0a mov ecx,dword ptr [esp+10h]
00df3d0e mov edi,dword ptr [esp+18h]
00df3d12 push edi
00df3d13 push eax
00df3d14 mov edx,dword ptr [ecx]
00df3d16 push ecx
00df3d17 push esi
```

```
00df3d18 mov edx,dword ptr [edx+4]
00df3d1b call dword ptr [edx+10h]
```

The value of EDX comes from EDX+4 which comes from ECX pointer value which was probably one of function parameters:

```
0:009> dp ecx L1
0b925970 0b925978

0:009> dp poi(ecx)+4 L1
0b92597c 0b925527

0:009> dc 0b925527
0b925527 20202020 80202020 01010101 01010101 .....
0b925537 00010101 01010101 01010000 01000101 .....
[...]
```

Since the data came from *ModuleA* function parameter it was passed from the previous call:

```
0:009> kL 4
# ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
00 0381f844 00df3d1e 0x10101
01 0381f860 00ca0a15 ModuleA!Bar+0x3e
02 0381f888 00ca07d4 ModuleB!Foo2+0x75
03 0381f8e4 00ca0ae3 ModuleB!Foo1+0xb4

0:009> ub 00ca0a15
ModuleB!Foo2+0x61:
00ca0a01 add esp,10h
00ca0a04 and edi,0FFFFFFF8h
00ca0a07 lea edx,[esp+20h]
00ca0a0b push edx
00ca0a0c push ebp
00ca0a0d push edi
00ca0a0e push esi
00ca0a0f call dword ptr [ModuleA!_imp__Bar (00cc32c0)]
```

The passed parameter looks like the second argument of *Bar* and *ModuleB!Foo2*, and the first argument of *ModuleB!Foo1*:

```
0:009> dps 0381f860
0381f860 064bf018
0381f864 00ca0a15 ModuleB!Foo2+0x75 ; return address when calling ModuleA!Bar
0381f868 064bf018
0381f86c 0b925970
0381f870 0c86a949
0381f874 0381f898
0381f878 0b925970
0381f87c 09584580
0381f880 064bf018
0381f884 0381f8c4
```

```

0381f888 0b925970
0381f88c 00ca07d4 ModuleB!Foo1+0xb4 ; return address when calling ModuleB!Foo2
0381f890 064bf018
0381f894 0b925970
0381f898 09584580
0381f89c 064bf018
0381f8a0 00cccd7d0
0381f8a4 0381f954
0381f8a8 00000000
0381f8ac 00000001
0381f8b0 00000000
0381f8b4 0b926d18
0381f8b8 00000000
0381f8bc 00000005
0381f8c0 00000000
0381f8c4 0b925970
0381f8c8 00000000
0381f8cc 00000000
0381f8d0 00000000
0381f8d4 00000000
0381f8d8 00000000
0381f8dc 00000000
0381f8e0 00000000
0381f8e4 00000000
0381f8e8 00ca0ae3 ModuleB!iFoo0+0x13 ; return address when calling ModuleB!Foo1
0381f8ec 0b925970
0381f8f0 00000000

```

We see **Parameter Flow** analysis pattern:

```

0:009> kvL 4
# ChildEBP RetAddr Args to Child
WARNING: Frame IP not in any known module. Following frames may be wrong.
00 0381f844 00df3d1e 064bf018 0b925970 06501dd8 0x10101
01 0381f860 00ca0a15 064bf018 0b925970 0c86a949 ModuleA!Bar+0x3e
02 0381f888 00ca07d4 064bf018 0b925970 09584580 ModuleB!Foo2+0x75
03 0381f8e4 00ca0ae3 0b925970 00000000 09584008 ModuleB!Foo1+0xb4

```

The example is from x86 Windows where parameters are passed through the stack, and, therefore, are seen in the verbose stack trace output (**kv**). For x64 Windows systems such parameters may be **Hidden Parameters** (page 465) and what we see in verbose output can be **False Function Parameters** (page 390). However, we can still track in some cases certain values found in **Execution Residue** (page 371) and associate them with particular stack trace frames (including **Past Stack Traces**, page 800).

**Parameter Flow** memory analysis pattern is similar to trace analysis **Data Flow**<sup>163</sup> pattern.

---

<sup>163</sup> Memory Dump Analysis Anthology, Volume 7, page 296

## Paratext

### Linux

This is Linux variant of **Paratext** pattern for Mac OS X (page 785). Because of debugger tool limitations, additional software logs and the output of other tools may help in memory dump analysis. Typical examples of such pattern usage can be the list of modules with version and path info, application crash specific information from instrumentation tools such as Valgrind, memory region names with attribution and boundaries, and CPU usage information. For example, *top* and *pmap* commands output:

```
top - 22:32:00 up 28 min, 1 user, load average: 0.95, 0.38, 0.17
Tasks: 64 total, 1 running, 63 sleeping, 0 stopped, 0 zombie
%Cpu(s):100.0 us, 0.0 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 505464 total, 174140 used, 331324 free, 11872 buffers
KiB Swap: 223228 total, 0 used, 223228 free, 122696 cached

PID-USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
8712 training 20 0 42040 4 0 S 99.9 0.0 2:06.38 App3
1 root 20 0 10648 836 704 S 0.0 0.2 0:01.34 init
2 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kthreadd
3 root 20 0 0 0 0 S 0.0 0.0 0:00.08 ksoftirqd/0
5 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kworker/u:0
6 root rt 0 0 0 0 S 0.0 0.0 0:00.00 migration/0
7 root rt 0 0 0 0 S 0.0 0.0 0:00.01 watchdog/0
8 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 cpuset
9 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 khelper
10 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kdevtmpfs
11 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 netns
12 root 20 0 0 0 0 S 0.0 0.0 0:00.01 sync_supers
13 root 20 0 0 0 0 S 0.0 0.0 0:00.00 bdi-default
14 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 kintegrityd
15 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 kblockd
16 root 20 0 0 0 0 S 0.0 0.0 0:00.00 khungtaskd
17 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kswapd0
18 root 25 5 0 0 0 S 0.0 0.0 0:00.00 ksmd
```

```
14039: ./App1.shared
0000000000400000 4K r-x-- /home/training/ALCDA/App1/App1.shared
0000000000600000 4K rw--- /home/training/ALCDA/App1/App1.shared
0000000000611000 132K rw--- [ anon ]
00007fe8999a6000 4K ----- [ anon ]
00007fe8999a7000 8192K rw--- [ anon ]
00007fe89a1a7000 4K ----- [ anon ]
00007fe89a1a8000 8192K rw--- [ anon ]
00007fe89a9a8000 4K ----- [ anon ]
00007fe89a9a9000 8192K rw--- [ anon ]
00007fe89b1a9000 4K ----- [ anon ]
00007fe89b1aa000 8192K rw--- [ anon ]
00007fe89b9aa000 4K ----- [ anon ]
00007fe89b9ab000 8192K rw--- [ anon ]
00007fe89c1ab000 1540K r-x-- /lib/x86_64-linux-gnu/libc-2.13.so
00007fe89c32c000 2048K ----- /lib/x86_64-linux-gnu/libc-2.13.so
00007fe89c52c000 16K r---- /lib/x86_64-linux-gnu/libc-2.13.so
00007fe89c530000 4K rw--- /lib/x86_64-linux-gnu/libc-2.13.so
00007fe89c531000 20K rw--- [ anon ]
00007fe89c536000 92K r-x-- /lib/x86_64-linux-gnu/libpthread-2.13.so
00007fe89c54d000 2044K ----- /lib/x86_64-linux-gnu/libpthread-2.13.so
00007fe89c74c000 4K r---- /lib/x86_64-linux-gnu/libpthread-2.13.so
00007fe89c74d000 4K rw--- /lib/x86_64-linux-gnu/libpthread-2.13.so
00007fe89c74e000 16K rw--- [ anon ]
00007fe89c752000 128K r-x-- /lib/x86_64-linux-gnu/ld-2.13.so
```

```
00007fe89c966000 12K      rw--- [ anon ]
00007fe89c96f000 8K      rw--- [ anon ]
00007fe89c971000 4K      r---- /lib/x86_64-linux-gnu/ld-2.13.so
00007fe89c972000 4K      rw--- /lib/x86_64-linux-gnu/ld-2.13.so
00007fe89c973000 4K      rw--- [ anon ]
00007ffd458c1000 132K    rw--- [ stack ]
00007ffd459e9000 4K      r-x-- [ anon ]
ffffffff600000 4K       r-x-- [ anon ]
total 47208K
```

## Mac OS X

This is the first pattern that emerged after applying pattern-driven software diagnostics methodology developed first on Windows platforms to Mac OS X. We had problems using GDB which is so portable that hardly has operating system support like WinDbg debugger has. Fortunately, we found a workaround by complementing core dumps with logs and reports from OS such as crash reports and vmmap data. The name of this pattern was borrowed from the concept of extended software trace<sup>164</sup> and software narratology<sup>165</sup> where it borrowed the same concept from literary interpretation (paratext<sup>166</sup>). Typical examples of such pattern usage can be the list of modules with version and path info, application crash specific information, memory region names with attribution and boundaries:

*// from .crash reports*

```
0x108f99000 - 0x109044ff7 com.apple.FontBook (198.4 - 198) <7244D36E-4563-3E42-BA46-1F279D30A6CE> /Applications/Font
Book.app/Contents/MacOS/Font Book

Exception Type: EXC_BAD_INSTRUCTION (SIGILL)
Exception Codes: 0x0000000000000001, 0x0000000000000000

Application Specific Information:
objc[195]: garbage collection is OFF
*** error for object 0x7fd7fb818e08: incorrect checksum for freed object - object was probably modified after being
freed.
```

*// from vmmap logs*

```
[...]
==== Writable regions for process 966
[...]
Stack 0000000101f71000-0000000101ff3000 [ 520K] rw-/rwx SM=PRV thread 1
MALLOC_LARGE 0000000103998000-00000001039b8000 [ 128K] rw-/rwx SM=PRV DefaultMallocZone_0x101e6e000
MALLOC_SMALL (freed) 00000001039b9000-00000001039bb000 [ 8K] rw-/rwx SM=PRV
mapped file 0000000103a05000-0000000103f32000 [ 5300K] rw-/rwx SM=COW ...box.framework/Versions/A/Resources/Extras2.rsrc
mapped file 0000000104409000-00000001046d2000 [ 2852K] rw-/rwx SM=COW /System/Library/Fonts/Helvetica.ttf
MALLOC_LARGE 0000000104f6e000-0000000104f8e000 [ 128K] rw-/rwx SM=PRV DefaultMallocZone_0x101e6e000
MALLOC_LARGE (freed) 0000000108413000-0000000108540000 [ 1204K] rw-/rwx SM=COW
MALLOC_LARGE (freed) 0000000108540000-0000000108541000 [ 4K] rw-/rwx SM=PRV
MALLOC_TINY 00007fefef0c00000-00007fefef0d0000 [ 1024K] rw-/rwx SM=COW DefaultMallocZone_0x101e6e000
MALLOC_TINY 00007fefef0d00000-00007fefef0e0000 [ 1024K] rw-/rwx SM=PRV DispatchContinuations_0x101f38000
MALLOC_TINY 00007fefef0e00000-00007fefef0f0000 [ 1024K] rw-/rwx SM=COW DefaultMallocZone_0x101e6e000
MALLOC_SMALL 00007fefef1000000-00007fefef107b000 [ 492K] rw-/rwx SM=ZER DefaultMallocZone_0x101e6e000
MALLOC_SMALL 00007fefef107b000-00007fefef1083000 [ 32K] rw-/rwx SM=PRV DefaultMallocZone_0x101e6e000
MALLOC_SMALL 00007fefef1083000-00007fefef1149000 [ 792K] rw-/rwx SM=ZER DefaultMallocZone_0x101e6e000
MALLOC_SMALL (freed) 00007fefef1149000-00007fefef1166000 [ 116K] rw-/rwx SM=PRV DefaultMallocZone_0x101e6e000
```

<sup>164</sup> The Extended Software Trace, Memory Dump Analysis Anthology, Volume 5, page 277

<sup>165</sup> <http://www.patterndiagnostics.com/Introduction-Software-Narratology-materials>

<sup>166</sup> <http://en.wikipedia.org/wiki/Paratext>

```
MALLOC_SMALL (freed) 00007fefe1166000-00007fefe1800000 [ 6760K] rw-/rwx SM=ZER DefaultMallocZone_0x101e6e000
MALLOC_SMALL 00007fefe1800000-00007fefe18ff000 [ 1020K] rw-/rwx SM=ZER DefaultMallocZone_0x101e6e000
MALLOC_SMALL (freed) 00007fefe18ff000-00007fefe1901000 [ 8K] rw-/rwx SM=PRV DefaultMallocZone_0x101e6e000
MALLOC_SMALL 00007fefe1901000-00007fefe2000000 [ 7164K] rw-/rwx SM=ZER DefaultMallocZone_0x101e6e000
MALLOC_TINY (freed) 00007fefe2000000-00007fefe2100000 [ 1024K] rw-/rwx SM=PRV DispatchContinuations_0x101f38000
MALLOC_TINY 00007fefe2100000-00007fefe2200000 [ 1024K] rw-/rwx SM=PRV DefaultMallocZone_0x101e6e000
Stack 00007ffff61186000-00007ffff61985000 [ 8188K] rw-/rwx SM=ZER thread 0
Stack 00007ffff61985000-00007ffff61986000 [ 4K] rw-/rwx SM=COW
[...]
```

## Comments

An example from managed space<sup>167</sup>.

---

<sup>167</sup> <http://www.codeproject.com/Articles/495208/VB-NET-Global-Try-Catch-in-the-Application-Framework>

## Pass Through Function

When constantly looking at **Stack Trace Collections** (page 943) from complete or kernel memory dumps we notice that certain processes are always present and remember them. They are no longer suspicious. The same about thread stacks. Some are always present, and some are not suspicious because of their function or status, like **Passive Threads** (page 793) or **Passive System Threads** (page 789). Going more fine-grained we can talk about components and their specific functions. For example, certain kernel space components have special filter functions; they get an IRP and pass it down the device stack. It doesn't take much code to check an IRP and forward it. This is usually reflected in small function offsets, for example:

```
ChildEBP RetAddr
aced780 80833ec5 nt!KiSwapContext+0x26
aced7ac 80829bc0 nt!KiSwapThread+0x2e5
aced7f4 badfffece nt!KeWaitForSingleObject+0x346
WARNING: Stack unwind information not available. Following frames may be wrong.
aced824 bae00208 AVFilterB+0x1ece
aced868 bae0e45a AVFilterB+0x2208
aced8a0 8081e095 AVFilterB+0x1045a
aced8b4 b946673b nt!IofCallDriver+0x45
aced8c4 b94626ee driverB!FS_Dispatch+0xfb
aced8d4 8081e095 driverB!dispatch+0x6e
aced8e8 b96e04e1 nt!IofCallDriver+0x45
aced90c b96e0755 driverA!PassThrough+0xd1
aced92c 8081e095 driverA!Create+0x155
aced940 b882df08 nt!IofCallDriver+0x45
aceda5c 8081e095 AVFilterA!DispatchPassThrough+0x48
aceda70 808fb13b nt!IofCallDriver+0x45
acedb58 80939c6a nt!IopParseDevice+0xa35
acedbd8 80935d9e nt!ObpLookupObjectName+0x5b0
acedc2c 808ece57 nt!ObOpenObjectByName+0xea
acedca8 808ee0f1 nt!IopCreateFile+0x447
acedd04 808f1e31 nt!IoCreateFile+0xa3
acedd44 8088ad3c nt!NtOpenFile+0x27
[...]
```

Here, if the thread is blocked, *AVFilterB* module is more suspicious than *AVFilterA* module because it is on top of the stack, waiting and *AVFilterA* just passed an IRP to the driver and the latter module seems also relayed the IRP to *driverB* and the latter relayed it to *AVFilterB*.

Another x64 example shows how these filter functions can be identified. They have “Dispatch” or “PassThrough” in their function names:

```
Child-SP      RetAddr      Call Site
fffffa60`12610880  fffff800`01875f8a nt!KiSwapContext+0x7f
fffffa60`126109c0  fffff800`0187776a nt!KiSwapThread+0x2fa
fffffa60`12610a30  fffff800`01ab16d6 nt!KeWaitForSingleObject+0x2da
[...]
fffffa60`12610fe0  fffffa60`06c5191a rdbss!RxFsdCommonDispatch+0x786
fffffa60`126110d0  fffffa60`07e4f21f rdbss!RxFsdDispatch+0x21a
fffffa60`12611140  fffffa60`011e05f5 mrxsmbs!MRxSmbFsdDispatch+0xbf
fffffa60`12611180  fffffa60`011e0130 mup!MupiCallUncProvider+0x159
fffffa60`126111f0  fffffa60`011e17af mup!MupStateMachine+0x120
fffffa60`12611240  fffffa60`00d200b4 mup!MupCreate+0x2c3
fffffa60`126112c0  fffffa60`06d332d6 fltmgr!FltpCreate+0xa4
fffffa60`12611370  fffffa60`06d786c7 driverB!FS_Dispatch+0x156
fffffa60`126113a0  fffffa60`06d7894d driverA!PassThrough+0x177
fffffa60`12611400  fffffa60`090b3f30 driverA!Create+0x14d
fffffa60`12611430  fffff800`01aef360 AVDriverA!LowerDevicePassThrough+0x5c
fffffa60`12611700  fffff800`01aeafa59 nt!IopParseDevice+0x5e3
fffffa60`126118a0  fffff800`01af3944 nt!ObpLookupObjectName+0x5eb
fffffa60`126119b0  fffff800`01affee0 nt!ObOpenObjectByName+0x2f4
fffffa60`12611a80  fffff800`01b00a0c nt!IopCreateFile+0x290
fffffa60`12611b20  fffff800`0186fdf3 nt!NtCreateFile+0x78
[...]
```

---

## Comments

Sometimes we need to check return addresses to see whether a function introduced an alternative execution path such as with hooking:

```
[...]
retA ntdll!NtCreateFile
retB DLL!HookCreateFile
retC kernel32!OtherFunction
[...]
```

Here we can check what function was called from *kernel32!OtherFunction*:

```
ub retB
[...]
call [kernel32!_imp_NtCreateFile]
```

If it is the same function semantically we can assume the hook is pass-through but if it is not then we have a divergence of execution flow, and we need to pay attention to that.

## Passive System Thread

### Kernel Space

In Windows, **Passive System Threads** don't run in any user process context. These threads belong to the so-called System process, don't have any user space stack and their full stack traces can be seen from the output of **!process** command (if not completely **Paged Out**, page 778) or from [System] portion of **!stacks 2** command:

```
1: kd> !process 0 3f System
```

Some system threads from that list belong to core OS functionality and are not passive (function offsets can vary for different OS versions and service packs):

```
nt!KiSwapContext+0x84
nt!KiSwapThread+0x125
nt!KeWaitForSingleObject+0x5f5
nt!MmZeroPageThread+0x180
nt!Phase1Initialization+0xe
nt!PspSystemThreadStartup+0x5b
nt!KiStartSystemThread+0x16
```

```
nt!KiSwapContext+0x84
nt!KiSwapThread+0x125
nt!KeWaitForSingleObject+0x5f5
nt!MiModifiedPageWriter+0x59
nt!PspSystemThreadStartup+0x5b
nt!KiStartSystemThread+0x16
```

```
nt!KiSwapContext+0x84
nt!KiSwapThread+0x125
nt!KeWaitForMultipleObjects+0x703
nt!MiMappedPageWriter+0xad
nt!PspSystemThreadStartup+0x5b
nt!KiStartSystemThread+0x16
```

```
nt!KiSwapContext+0x84
nt!KiSwapThread+0x125
nt!KeWaitForMultipleObjects+0x703
nt!KeBalanceSetManager+0x101
nt!PspSystemThreadStartup+0x5b
nt!KiStartSystemThread+0x16
```

```
nt!KiSwapContext+0x84
nt!KiSwapThread+0x125
nt!KeWaitForSingleObject+0x5f5
nt!KeSwapProcessOrStack+0x44
nt!PspSystemThreadStartup+0x5b
nt!KiStartSystemThread+0x16
```

```
nt!KiSwapContext+0x84
nt!KiSwapThread+0x125
```

```

nt!KeWaitForSingleObject+0x5f5
nt!EtwpLogger+0xdd
nt!PspSystemThreadStartup+0x5b
nt!KiStartSystemThread+0x16

nt!KiSwapContext+0x84
nt!KiSwapThread+0x125
nt!KeWaitForSingleObject+0x5f5
nt!KiExecuteDpc+0x198
nt!PspSystemThreadStartup+0x5b
nt!KiStartSystemThread+0x16

nt!KiSwapContext+0x84
nt!KiSwapThread+0x125
nt!KeWaitForMultipleObjects+0x703
nt!CcQueueLazyWriteScanThread+0x73
nt!PspSystemThreadStartup+0x5b
nt!KiStartSystemThread+0x16

nt!KiSwapContext+0x84
nt!KiSwapThread+0x125
nt!KeWaitForMultipleObjects+0x703
nt!ExpWorkerThreadBalanceManager+0x85
nt!PspSystemThreadStartup+0x5b
nt!KiStartSystemThread+0x16

```

Other threads belong to various worker queues (they can also be seen from **!exqueue ff** command output), and they wait for data items to arrive (passive threads):

```

nt!KiSwapContext+0x84
nt!KiSwapThread+0x125
nt!KeRemoveQueueEx+0x848
nt!ExpWorkerThread+0x104
nt!PspSystemThreadStartup+0x5b
nt!KiStartSystemThread+0x16

```

or

```

nt!KiSwapContext+0x26
nt!KiSwapThread+0x2e5
nt!KeRemoveQueue+0x417
nt!ExpWorkerThread+0xc8
nt!PspSystemThreadStartup+0x2e
nt!KiThreadStartup+0x16

```

Non-*Exp* system threads (no *nt!ExpWorkerThread* in their stack traces) having “Worker”, “Logging” or “Logger” substrings in their function names are passive threads, and they wait for data too, for example:

```
nt!KiSwapContext+0x84
nt!KiSwapThread+0x125
nt!KeWaitForMultipleObjects+0x703
nt!PfTLoggingWorker+0x81
nt!PspSystemThreadStartup+0x5b
nt!KiStartSystemThread+0x16

nt!KiSwapContext+0x84
nt!KiSwapThread+0x125
nt!KeWaitForSingleObject+0x5f5
nt!EtwpLogger+0xdd
nt!PspSystemThreadStartup+0x5b
nt!KiStartSystemThread+0x16

nt!KiSwapContext+0x84
nt!KiSwapThread+0x125
nt!KeRemoveQueueEx+0x848
nt!KeRemoveQueue+0x21
rdpdr!RxpWorkerThreadDispatcher+0x6f
nt!PspSystemThreadStartup+0x5b
nt!KiStartSystemThread+0x16

nt!KiSwapContext+0x84
nt!KiSwapThread+0x125
nt!KeWaitForSingleObject+0x5f5
HTTP!UlpThreadPoolWorker+0x26c
nt!PspSystemThreadStartup+0x5b
nt!KiStartSystemThread+0x16

nt!KiSwapContext+0x84
nt!KiSwapThread+0x125
nt!KeRemoveQueueEx+0x848
nt!KeRemoveQueue+0x21
srv2!SrvProcWorkerThread+0x74
nt!PspSystemThreadStartup+0x5b
nt!KiStartSystemThread+0x16

nt!KiSwapContext+0x84
nt!KiSwapThread+0x125
nt!KeRemoveQueueEx+0x848
nt!KeRemoveQueue+0x21
srv!WorkerThread+0x90
nt!PspSystemThreadStartup+0x5b
nt!KiStartSystemThread+0x16
```

Any deviations in a memory dump can raise suspicion like in the stack below for *driver.sys*:

```
nt!KiSwapContext+0x26
nt!KiSwapThread+0x284
nt!KeWaitForSingleObject+0x346
nt!ExpWaitForResource+0xd5
nt!ExAcquireResourceExclusiveLite+0x8d
nt!ExEnterCriticalSectionAndAcquireResourceExclusive+0x19
driver!ProcessItem+0x2f
driver!DelayedWorker+0x27
nt!ExpWorkerThread+0x104
nt!PspSystemThreadStartup+0x5b
nt!KiStartSystemThread+0x16
```

## Passive Thread

### User Space

When trying to understand why the particular application or service hangs, we look at **Stack Trace Collection** pattern (page 943) and hope to find some suspicious threads that are waiting for a response. These are active blocked threads. Other threads may appear waiting, but they are merely waiting for some notification or data that may or may not come during their lifetime and, therefore, are normal. In other words, they are passive and hence the name of the pattern **Passive Thread**. Typical examples from user space include

- The main service thread and dispatch threads (when idle).
- A thread is waiting for file or registry notifications.
- A generic RPC/LPC/COM thread is waiting for messages.
- Worker threads are waiting for data to appear in a queue.
- Window message loops (when idle).
- Socket and network protocol threads (when idle).
- A thread with function names on its stack trace suggesting that it is a notification or listener thread.

Of course, sometimes these passive threads can be the reason for an application or service hang, but from our experience, most of the time they are not unless there are other threads which they block. Let's now look at example stack traces.

Generic threads spawned to service various requests and waiting for data to arrive can be filtered using **!uniqstack** WinDbg command. Conceptually these threads are part of the so-called thread pool software design pattern.

#### LPC/RPC/COM threads waiting for requests:

```
70 Id: 8f8.1100 Suspend: 1 Teb: 7ff80000 Unfrozen
ChildEBP RetAddr
0d82fe18 7c82783b ntdll!KiFastSystemCallRet
0d82fe1c 77c885ac ntdll!NtReplyWaitReceivePortEx+0xc
0d82ff84 77c88792 rpct4!LRPC_ADDRESS::ReceiveLotsaCalls+0x198
0d82ff8c 77c8872d rpct4!RecvLotsaCallsWrapper+0xd
0d82ffac 77c7b110 rpct4!BaseCachedThreadRoutine+0x9d
0d82ffb8 77e64829 rpct4!ThreadStartRoutine+0x1b
0d82ffec 00000000 kernel32!BaseThreadStart+0x34
```

```
71 Id: 8f8.1e44 Suspend: 1 Teb: 7ffd8000 Unfrozen
ChildEBP RetAddr
0c01fe18 7c82783b ntdll!KiFastSystemCallRet
0c01fe1c 77c885ac ntdll!NtReplyWaitReceivePortEx+0xc
0c01ff84 77c88792 rpct4!LRPC_ADDRESS::ReceiveLotsaCalls+0x198
0c01ff8c 77c8872d rpct4!RecvLotsaCallsWrapper+0xd
0c01ffac 77c7b110 rpct4!BaseCachedThreadRoutine+0x9d
0c01ffb8 77e64829 rpct4!ThreadStartRoutine+0x1b
0c01ffec 00000000 kernel32!BaseThreadStart+0x34
```

```

72 Id: 8f8.1804 Suspend: 1 Teb: 7ff90000 Unfrozen
ChildEBP RetAddr
0e22fe18 7c82783b ntdll!KiFastSystemCallRet
0e22fe1c 77c885ac ntdll!NtReplyWaitReceivePortEx+0xc
0e22ff84 77c88792 rp crt4!LRPC_ADDRESS::ReceiveLotsaCalls+0x198
0e22ff8c 77c8872d rp crt4!RecvLotsaCallsWrapper+0xd
0e22ffac 77c7b110 rp crt4!BaseCachedThreadRoutine+0x9d
0e22ffb8 77e64829 rp crt4!ThreadStartRoutine+0x1b
0e22ffec 00000000 kernel32!BaseThreadStart+0x34

73 Id: 8f8.1860 Suspend: 1 Teb: 7ff79000 Unfrozen
ChildEBP RetAddr
0da2fe18 7c82783b ntdll!KiFastSystemCallRet
0da2fe1c 77c885ac ntdll!NtReplyWaitReceivePortEx+0xc
0da2ff84 77c88792 rp crt4!LRPC_ADDRESS::ReceiveLotsaCalls+0x198
0da2ff8c 77c8872d rp crt4!RecvLotsaCallsWrapper+0xd
0da2ffac 77c7b110 rp crt4!BaseCachedThreadRoutine+0x9d
0da2ffb8 77e64829 rp crt4!ThreadStartRoutine+0x1b
0da2ffec 00000000 kernel32!BaseThreadStart+0x34

74 Id: 8f8.f24 Suspend: 1 Teb: 7ff7e000 Unfrozen
ChildEBP RetAddr
0d20feac 7c8277db ntdll!KiFastSystemCallRet
0d20feb0 77e5bea2 ntdll!ZwRemoveIoCompletion+0xc
0d20fedc 77c7b900 kernel32!GetQueuedCompletionStatus+0x29
0d20ff18 77c7b703 rp crt4!COMMON_ProcessCalls+0xa1
0d20ff84 77c7b9b5 rp crt4!LOADABLE_TRANSPORT::ProcessIOEvents+0x117
0d20ff8c 77c8872d rp crt4!ProcessIOEventsWrapper+0xd
0d20ffac 77c7b110 rp crt4!BaseCachedThreadRoutine+0x9d
0d20ffb8 77e64829 rp crt4!ThreadStartRoutine+0x1b
0d20ffec 00000000 kernel32!BaseThreadStart+0x34

75 Id: 8f8.11f8 Suspend: 1 Teb: 7ffa1000 Unfrozen
ChildEBP RetAddr
08e0feac 7c8277db ntdll!KiFastSystemCallRet
08e0feb0 77e5bea2 ntdll!ZwRemoveIoCompletion+0xc
08e0fedc 77c7b900 kernel32!GetQueuedCompletionStatus+0x29
08e0ff18 77c7b703 rp crt4!COMMON_ProcessCalls+0xa1
08e0ff84 77c7b9b5 rp crt4!LOADABLE_TRANSPORT::ProcessIOEvents+0x117
08e0ff8c 77c8872d rp crt4!ProcessIOEventsWrapper+0xd
08e0ffac 77c7b110 rp crt4!BaseCachedThreadRoutine+0x9d
08e0ffb8 77e64829 rp crt4!ThreadStartRoutine+0x1b
08e0ffec 00000000 kernel32!BaseThreadStart+0x34

2 Id: ecc.c94 Suspend: 1 Teb: 7efac000 Unfrozen
ChildEBP RetAddr
0382f760 76e31330 ntdll!NtDelayExecution+0x15
0382f7c8 76e30dac kernel32!SleepEx+0x62
0382f7d8 75ec40f4 kernel32!Sleep+0xf
0382f7e4 75eafc0d ole32!CROIDTable::WorkerThreadLoop+0x14
0382f800 75eafc73 ole32!CRpcThread::WorkerLoop+0x26
0382f80c 76ea19f1 ole32!CRpcThreadCache::RpcWorkerThreadEntry+0x20
0382f818 7797d109 kernel32!BaseThreadInitThunk+0xe
0382f858 00000000 ntdll!_RtlUserThreadStart+0x23

```

Worker threads waiting for data items to process:

```

43 Id: 8f8.17c0 Suspend: 1 Teb: 7ff8c000 Unfrozen
ChildEBP RetAddr
0c64ff20 7c8277db ntdll!KiFastSystemCallRet
0c64ff24 77e5bea2 ntdll!ZwRemoveIoCompletion+0xc
0c64ff50 67823549 kernel32!GetQueuedCompletionStatus+0x29
0c64ff84 77bcb530 component!WorkItemThread+0xa9
0c64ffb8 77e64829 msrvct!_endthreadex+0xa3
0c64ffec 00000000 kernel32!BaseThreadStart+0x34

44 Id: 8f8.7b4 Suspend: 1 Teb: 7ff8b000 Unfrozen
ChildEBP RetAddr
0c77ff20 7c8277db ntdll!KiFastSystemCallRet
0c77ff24 77e5bea2 ntdll!ZwRemoveIoCompletion+0xc
0c77ff50 67823549 kernel32!GetQueuedCompletionStatus+0x29
0c77ff84 77bcb530 component!WorkItemThread+0xa9
0c77ffb8 77e64829 msrvct!_endthreadex+0xa3
0c77ffec 00000000 kernel32!BaseThreadStart+0x34

45 Id: 8f8.1708 Suspend: 1 Teb: 7ff8a000 Unfrozen
ChildEBP RetAddr
0c87ff20 7c8277db ntdll!KiFastSystemCallRet
0c87ff24 77e5bea2 ntdll!ZwRemoveIoCompletion+0xc
0c87ff50 67823549 kernel32!GetQueuedCompletionStatus+0x29
0c87ff84 77bcb530 component!WorkItemThread+0xa9
0c87ffb8 77e64829 msrvct!_endthreadex+0xa3
0c87ffec 00000000 kernel32!BaseThreadStart+0x34

5 Id: 11fc.16f4 Suspend: 1 Teb: 7ffd9000 Unfrozen
ChildEBP RetAddr
0109bf10 7c822124 ntdll!KiFastSystemCallRet
0109bf14 77e6baa8 ntdll!NtWaitForSingleObject+0xc
0109bf84 77e6ba12 kernel32!WaitForSingleObjectEx+0xac
0109bf98 66886519 kernel32!WaitForSingleObject+0x12
0109ff84 77bcb530 component!WorkerThread+0xe8
0109ffb8 77e66063 msrvct!_endthreadex+0xa3
0109ffec 00000000 kernel32!BaseThreadStart+0x34

```

A thread waiting for registry change notification:

```

1 Id: 13c4.350 Suspend: 1 Teb: 000007ff`fffde000 Unfrozen
Child-SP           RetAddr          Call Site
00000000`0012fdd8 000007fe`fd62c361 ntdll!ZwNotifyChangeKey+0xa
00000000`0012fde0 00000001`40001181 ADVAPI32!RegNotifyChangeKeyValue+0x115
00000000`0012ff30 00000000`76d9cdcd sample12!WaitForRegChange+0xe
00000000`0012ff60 00000000`76eec6e1 kernel32!BaseThreadInitThunk+0xd
00000000`0012ff90 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

```

Idle main service thread and service dispatch threads:

```

. 0 Id: 65c.660 Suspend: 1 Teb: 000007ff`ffffdc000 Unfrozen
Child-SP      RetAddr      Call Site
00000000`0011f2c8 00000000`76d926da ntdll!NtReadFile+0xa
00000000`0011f2d0 000007fe`fd6665aa kernel32!ReadFile+0x8a
00000000`0011f360 000007fe`fd6662e3 ADVAPI32!ScGetPipeInput+0x4a
00000000`0011f440 000007fe`fd6650f3 ADVAPI32!ScDispatcherLoop+0x9a
00000000`0011f540 00000000`ff0423a3 ADVAPI32!StartServiceCtrlDispatcherW+0x176
00000000`0011f7e0 00000000`ff042e66 spoolsv!main+0x23
00000000`0011f850 00000000`76eec6e1 kernel32!BaseThreadInitThunk+0xd
00000000`0011f880 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

1 Id: 65c.664 Suspend: 1 Teb: 000007ff`ffffda000 Unfrozen
Child-SP      RetAddr      Call Site
00000000`0009f9c8 00000000`76d9d820 ntdll!NtWaitForSingleObject+0xa
00000000`0009f9d0 00000000`ff04307f kernel32!WaitForSingleObjectEx+0x9c
00000000`0009fa90 000007fe`fd664bf5 spoolsv!SPOOLER_main+0x80
00000000`0009fac0 00000000`76d9cdcd ADVAPI32!ScSvcctrlThreadW+0x25
00000000`0009faf0 00000000`76eec6e1 kernel32!BaseThreadInitThunk+0xd
00000000`0009fb20 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

```

Idle window message loops:

```

10 Id: 65c.514 Suspend: 1 Teb: 000007ff`ffffa2000 Unfrozen
Child-SP      RetAddr      Call Site
00000000`02c5fc18 00000000`76cae6ea USER32!ZwUserGetMessage+0xa
00000000`02c5fc20 000007fe`f88523f0 USER32!GetMessageW+0x34
00000000`02c5fc50 00000000`76d9cdcd usbmon!CPNPNotifications::WindowMessageThread+0x1a0
00000000`02c5fd20 00000000`76eec6e1 kernel32!BaseThreadInitThunk+0xd
00000000`02c5fd50 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

11 Id: 65c.9bc Suspend: 1 Teb: 000007ff`ffffa0000 Unfrozen
Child-SP      RetAddr      Call Site
00000000`037cf798 00000000`76cae6ea USER32!ZwUserGetMessage+0xa
00000000`037cf7a0 000007fe`f7ea0d3a USER32!GetMessageW+0x34
00000000`037cf7d0 00000000`76d9cdcd WSDMon!Ncd::TPower::WindowMessageThread+0xe6
00000000`037cf870 00000000`76eec6e1 kernel32!BaseThreadInitThunk+0xd
00000000`037cf8a0 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

13 Id: ecc.b34 Suspend: 1 Teb: 7ef85000 Unfrozen
ChildEBP RetAddr
0621fc18 75b86458 USER32!NtUserGetMessage+0x15
0621fc3c 74aa1404 USER32!GetMessageA+0xa2
0621fc74 76ea19f1 WINMM!mciwindow+0x102
0621fc80 7797d109 kernel32!BaseThreadInitThunk+0xe
0621fcc0 00000000 ntdll!_RtlUserThreadStart+0x23

```

Idle socket and network protocol threads:

```

5 Id: ecc.920 Suspend: 1 Teb: 7efa3000 Unfrozen
ChildEBP RetAddr
0412f534 751b3b28 ntdll!ZwWaitForSingleObject+0x15
0412f574 751b2690 mssock!SockWaitForSingleObject+0x19f
0412f660 771d3781 mssock!WSPSelect+0x38c
0412f6dc 760f60fd ws2_32!select+0x456
0412fa34 760f2a78 WININET!ICASyncThread::SelectThread+0x242
0412fa3c 76ea19f1 WININET!ICASyncThread::SelectThreadWrapper+0xd
0412fa48 7797d109 kernel32!BaseThreadInitThunk+0xe
0412fa88 00000000 ntdll!_RtlUserThreadStart+0x23

6 Id: ecc.b1c Suspend: 1 Teb: 7ef9d000 Unfrozen
ChildEBP RetAddr
047afa6c 751b1b25 ntdll!NtRemoveIoCompletion+0x15
047afaa4 76ea19f1 mssock!SockAsyncThread+0x69
047afab0 7797d109 kernel32!BaseThreadInitThunk+0xe
047afaf0 00000000 ntdll!_RtlUserThreadStart+0x23

7 Id: 820.f90 Suspend: 1 Teb: 7ffd9000 Unfrozen
ChildEBP RetAddr
018dff84 7c93e9ab ntdll!KiFastSystemCallRet
018dff88 60620e6c ntdll!ZwWaitForMultipleObjects+0xc
018dffb4 7c80b683 NETAPI32!NetbiosWaiter+0x73
018dffec 00000000 kernel32!BaseThreadStart+0x37

```

Function names showing passive nature of threads:

```

8 Id: 65c.b40 Suspend: 1 Teb: 000007ff`ffffa6000 Unfrozen
Child-SP          RetAddr          Call Site
00000000`0259fdc8 00000000`76d9d820 ntdll!NtWaitForSingleObject+0xa
00000000`0259fdd0 000007fe`f8258084 kernel32!WaitForSingleObjectEx+0x9c
00000000`0259fe90 000007fe`fee994e7 wsntp32!thrNotify+0x9c
00000000`0259fef0 000007fe`fee9967d msvrt!endthreadex+0x47
00000000`0259ff20 00000000`76d9cdcd msvrt!endthreadex+0x100
00000000`0259ff50 00000000`76eec6e1 kernel32!BaseThreadInitThunk+0xd
00000000`0259ff80 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

12 Id: 65c.908 Suspend: 1 Teb: 000007ff`fff9e000 Unfrozen
Child-SP          RetAddr          Call Site
00000000`0368fd48 00000000`76d9d820 ntdll!NtWaitForSingleObject+0xa
00000000`0368fd50 000007fe`fa49af0 kernel32!WaitForSingleObjectEx+0x9c
00000000`0368fe10 00000000`76d9cdcd FunDisc!CNotificationQueue::ThreadProc+0x2ec
00000000`0368fe70 00000000`76eec6e1 kernel32!BaseThreadInitThunk+0xd
00000000`0368fea0 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

```

```

13 Id: 65c.904 Suspend: 1 Teb: 000007ff`fff9c000 Unfrozen
Child-SP      RetAddr          Call Site
00000000`034af9f8 00000000`76d9ed73 ntdll!NtWaitForMultipleObjects+0xa
00000000`034afa00 00000000`76cae96d kernel32!WaitForMultipleObjectsEx+0x10b
00000000`034afb10 00000000`76cae85e USER32!RealMsgWaitForMultipleObjectsEx+0x129
00000000`034afbb0 00000000`76ca3680 USER32!MsgWaitForMultipleObjectsEx+0x46
00000000`034afbf0 000007fe`fa49b60a USER32!MsgWaitForMultipleObjects+0x20
00000000`034afc30 00000000`76d9cdcd FunDisc!ListenerThread+0x1a6
00000000`034afd20 00000000`76eec6e1 kernel32!BaseThreadInitThunk+0xd
00000000`034afd50 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

64 Id: 8f8.1050 Suspend: 1 Teb: 7ff74000 Unfrozen
ChildEBP RetAddr
0ef5fa48 7c82787b ntdll!KiFastSystemCall1Ret
0ef5fa4c 77c80a6e ntdll!NtRequestWaitReplyPort+0xc
0ef5fa98 77c7fcf0 rpckt4!LRPC_CCALL::SendReceive+0x230
0ef5faa4 77c80673 rpckt4!I_RpcSendReceive+0x24
0ef5fab8 77ce315a rpckt4!NdrSendReceive+0x2b
0ef5fea0 771f4fb0 rpckt4!NdrClientCall2+0x22e
0ef5feb8 771f4f60 winsta!RpcWinStationWaitSystemEvent+0x1c
0ef5ff00 76f01422 winsta!WinStationWaitSystemEvent+0x51
0ef5ff24 0c922ace wtsapi32!WTSWaitSystemEvent+0x97
0ef5ff48 67823331 component!MonitorEvents+0xaf
0ef5ffb8 77e64829 msvcrt!_endthreadex+0xa3
0ef5ffec 00000000 kernel32!BaseThreadStart+0x34

11 Id: 140c.e8c Suspend: 1 Teb: 7ffaf000 Unfrozen
ChildEBP RetAddr
01e3fec0 7c822114 ntdll!KiFastSystemCall1Ret
01e3fec4 77e6711b ntdll!NtWaitForMultipleObjects+0xc
01e3ff6c 77e61075 kernel32!WaitForMultipleObjectsEx+0x11a
01e3ff88 76928415 kernel32!WaitForMultipleObjects+0x18
01e3ffb8 77e66063 userenv!!NotificationThread+0x5f
01e3ffec 00000000 kernel32!BaseThreadStart+0x34

```

When in doubt it is always a good idea to examine threads in non-hanging processes to see their normal idle stack traces. See Appendix A: Reference Stack Traces.

## Comments

---

There were some questions asked:

**Q.** When I attach WinDbg I see all my suspended threads are passive threads. How should I analyze in this particular scenario?

**A.** In this case, you probably need access to its source code or understand its architecture. If it is a GUI application running in a terminal services environment, it can be the case that mouse/keyboard events don't reach it.

**Q.** I have been debugging a service that reads data from the USB. However, all the ReadFile calls never finish (return `ERROR_IO_PENDING`) even after calling `WaitForSingleObject` (I am using overlapped I/O). Interestingly enough, if I run the binary as an app, everything works smoothly. This behavior is only present in Windows Vista; XP is fine. I have been trying to debug the vista HIDUSB driver (it's a HID device), and all the IRPs are in a pending state. Any suggestions on why this is happening?

**A.** It may be due to the separation of services (session 0) and console applications (session 1) in Vista.

## Past Stack Trace

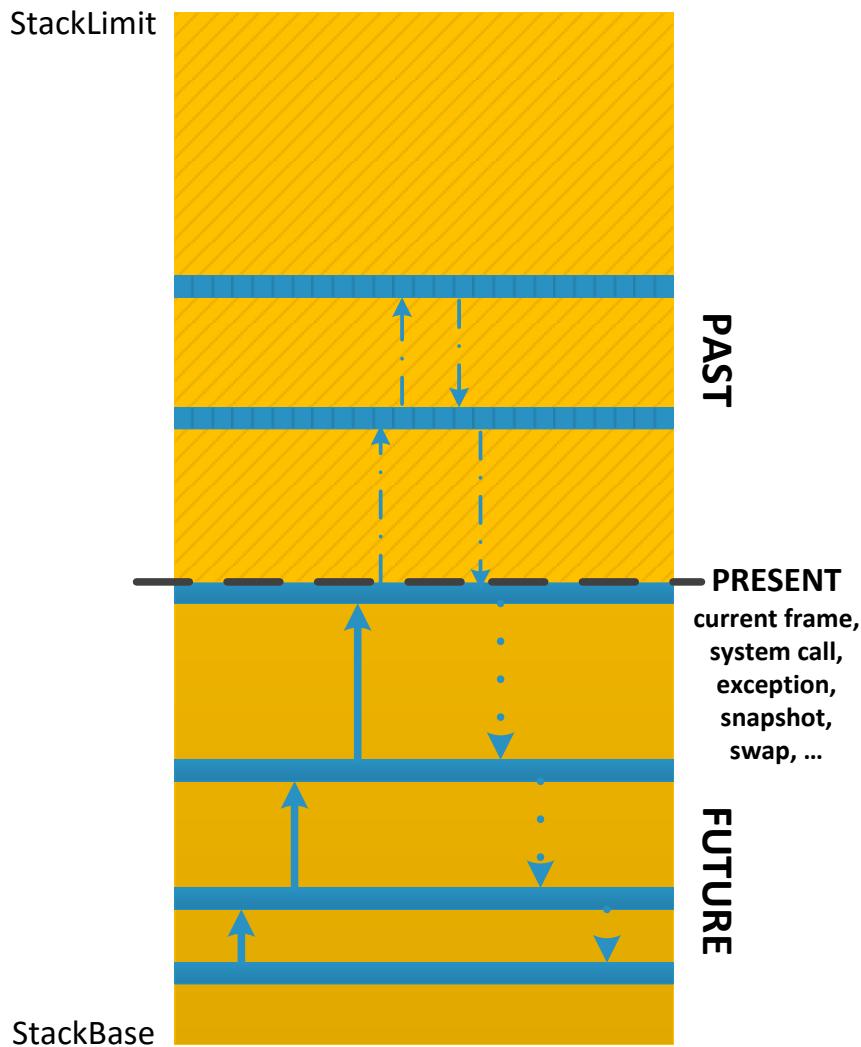
When we look at a stack trace in a memory dump, we see only the current thread execution snapshot of function calls. Consider this stack trace, for example, from [Spiking Thread](#) (page 888):

```
0:000> k
Child-SP RetAddr  Call Site
00000000`0012d010 00000000`76eb59ed App!WinMain+0x1eda
00000000`0012f7c0 00000000`770ec541 kernel32!BaseThreadInitThunk+0xd
00000000`0012f7f0 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

By looking at [Rough Stack Trace](#) (page 839), we may be able to reconstruct **Past Stack Trace** of what had happened just before the memory snapshot was taken:

```
0:000> k
Child-SP RetAddr  Call Site
00000000`0012cf8 00000000`76fd9e9e user32!ZwUserGetMessage+0xa
00000000`0012cfe0 00000000`ffd91a8c user32!GetMessageW+0x34
00000000`0012d010 00000000`76eb59ed App!WinMain+0x1dca
00000000`0012f7c0 00000000`770ec541 kernel32!BaseThreadInitThunk+0xd
00000000`0012f7f0 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

The stack region “time” zones are illustrated in the following picture:



The “Future” zone takes its name from the not yet executed returns. Of course, each stack subtrace generates its own partition. A similar version of this pattern was first introduced in Debugging TV Frames episode 0x24. You can watch the video<sup>168</sup> and find the source code, WinDbg logs, and presentation on Debugging TV website<sup>169</sup>.

<sup>168</sup> <https://www.youtube.com/watch?v=qRaabDzW3ww>

<sup>169</sup> <http://www.debugging.tv/>

## Patched Code

Hooksware (page 1175) patterns originally came from memory dump analysis pattern catalog and were too general for malware analysis pattern catalog. So we decided to factor out 3 separate patterns. The first one includes cases such as in-place patching:

```
0:004> u ntdll!ZwQueryDirectoryFile
ntdll!ZwQueryDirectoryFile:
77814db4 b8da000000      mov     eax,0DAh
77814db9 bae8af0500      mov     edx,5AFE8h
77814dbe ff12            call    dword ptr [edx]
77814dc0 c22c00          ret    2Ch
77814dc3 90              nop
ntdll!NtQueryDirectoryObject:
77814dc4 b8db000000      mov     eax,0DBh
77814dc9 ba0003fe7f      mov     edx,offset SharedUserData!SystemCallStub (7ffe0300)
77814dce ff12            call    dword ptr [edx]
```

and detour patching:

```
0:004> u wininet!InternetReadFile
wininet!InternetReadFile:
7758654b e98044ac88      jmp    0004a9d0
77586550 83ec24          sub    esp,24h
77586553 53              push   ebx
77586554 56              push   esi
77586555 57              push   edi
77586556 33ff            xor    edi,edi
77586558 393db8116277    cmp    dword ptr [wininet!GlobalDataInitialized (776211b8)],edi
7758655e 897df4          mov    dword ptr [ebp-0Ch],edi
```

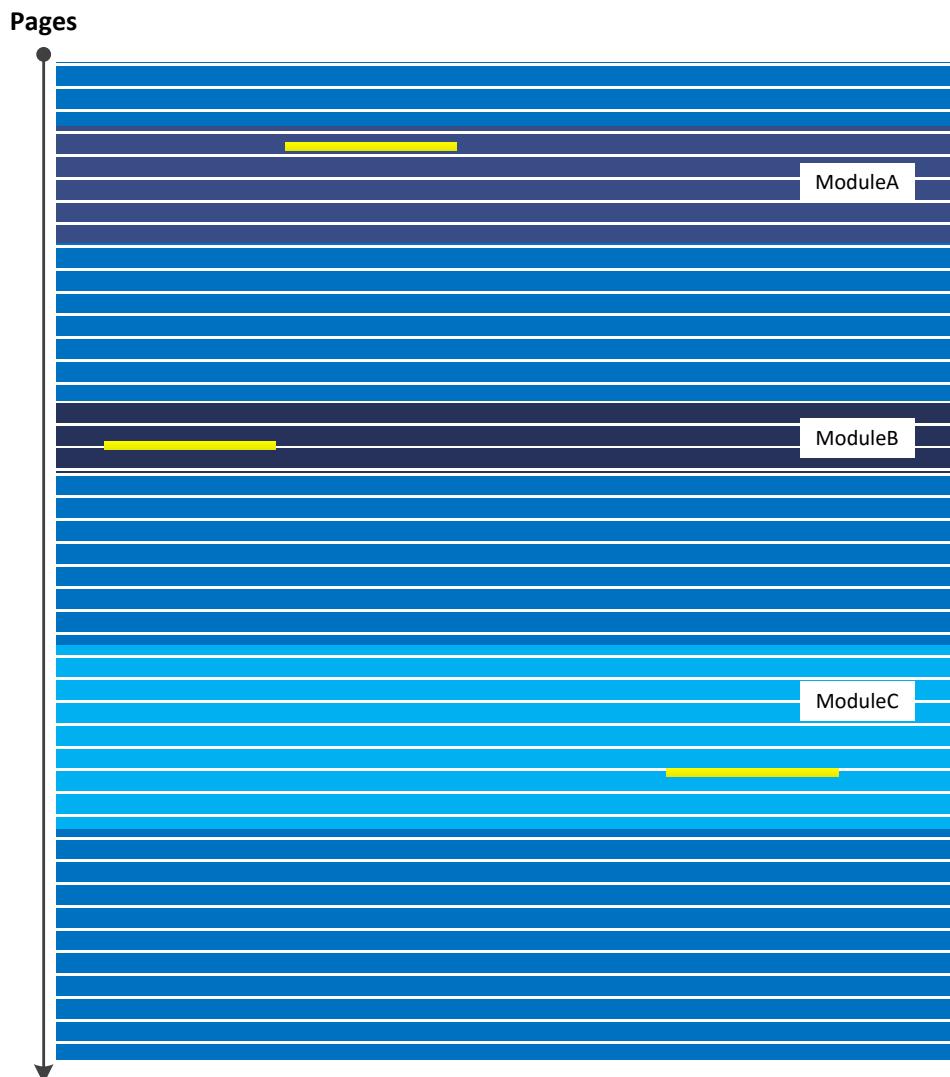
In the case of WinDbg, such pattern is usually detected on the crash spot such as from **RIP Stack Trace** (page 834) or from **!chkimg** command output.

## Pervasive System

Sometimes when looking at a module list (**!mv** WinDbg command) we see the whole presence of this pattern. It is not just a module that does function (**Hooked Functions**, page 484) or message (**Message Hooks**, page 663) hooking, but the whole system of modules from a single vendor that is context-aware (for example, reads its configuration from registry) and consists of several components that communicate with other processes. The penetrated system is supposed to add some additional value or to coexist peacefully in a larger environment. The system thus becomes strongly coupled (**Coupled Processes**, page 149) or weakly (page 151) with other processes it was never intended to work with as opposed to intended **Module Variety** (page 703). At one extreme, modules from the pervasive system can be **Ubiquitous Components** (page 1020) and, at the other end, **Hidden Modules** (page 463). In such cases troubleshooting consists of the total removal of the pervasive modules and, if the problem disappears, their exclusion one by one to find the problem component.

## Place Trace

Most **Execution Residue** (page 365) traces in memory dumps are not explicitly temporal (see *Special and General Trace and Log Analysis*<sup>170</sup>) but may be ordered by some space coordinate, such as memory addresses or page frame numbers. Furthermore, virtual space can be further subdivided into **places** such as modules, and physical space may be restructured into places such as processes. Simple space trace of some data value can be constructed using **Value References** (page 1057) analysis pattern. These and higher structural space trace constructs can be named as a general **Place Trace** analysis pattern illustrated in this diagram:



<sup>170</sup> Memory Dump Analysis Anthology, Volume 8b, page 119

Memory attributes, such as page protection, or derived attributes from memory contents can also be considered as **Place Trace** data. Sometimes, time ordering can be reconstructed by looking at time information for place containers, for example, elapsed process time or ordering in the process list, or thread order and times for stack region thread owners.

---

## Comments

Data can also be place traced across different thread stack regions.

## Platform-Specific Debugger

Analysis of .NET managed code requires processor architectural platform specific SOS extension. For example, x64 WinDbg is not able to analyze the managed stack for **Managed Code Exception** (page 617) in the 32-bit process:

```
0:010> !analyze -v

[...]

FAULTING_IP:
kernel32!RaiseException+53
77e4bee7 5e          pop     esi

EXCEPTION_RECORD: ffffffff -- (.exr 0xffffffffffff)
ExceptionAddress: 77e4bee7 (kernel32!RaiseException+0x00000053)
  ExceptionCode: e0434f4d (CLR exception)
  ExceptionFlags: 00000001
NumberParameters: 1
  Parameter[0]: 80131509

[...]

MANAGED_STACK: !dumpstack -EE
No export dumpstack found

MANAGED_BITNESS_MISMATCH:
Managed code needs matching platform of sos.dll for proper analysis. Use 'x86' debugger.

[...]

0:010> kL 100
ChildEBP RetAddr
0573f0a4 79f071ac kernel32!RaiseException+0x53
0573f104 79f0a780 mscorewks!RaiseTheExceptionInternalOnly+0x2a8
0573f1a8 058ed3b3 mscorewks!JIT_Rethrow+0xbf
WARNING: Frame IP not in any known module. Following frames may be wrong.
0573f33c 793b0d1f <Unloaded_D11A.dll>+0x58ed3b2
0573f344 79373ecd mscorlib_ni+0x2f0d1f
0573f358 793b0c68 mscorlib_ni+0x2b3ecd
0573f370 79e7c74b mscorlib_ni+0x2f0c68
0573f380 79e7c6cc mscorewks!CallDescrWorker+0x33
0573f400 79e7c8e1 mscorewks!CallDescrWorkerWithHandler+0xa3
0573f53c 79e7c783 mscorewks!MethodDesc::CallDescr+0x19c
0573f558 79e7c90d mscorewks!MethodDesc::CallTargetWorker+0x1f
0573f56c 79fc58cd mscorewks!MethodDescCallSite::Call_RetArgSlot+0x18
0573f754 79ef3207 mscorewks!ThreadNative::KickOffThread_Worker+0x190
0573f768 79ef31a3 mscorewks!Thread::DoADCallBack+0x32a
0573f7fc 79ef30c3 mscorewks!Thread::ShouldChangeAbortToUnload+0xe3
0573f838 79ef4826 mscorewks!Thread::ShouldChangeAbortToUnload+0x30a
0573f860 79fc57b1 mscorewks!Thread::ShouldChangeAbortToUnload+0x33e
0573f878 79fc56ac mscorewks!ManagedThreadBase::KickOff+0x13
0573f914 79f95a2e mscorewks!ThreadNative::KickOffThread+0x269
```

```
0573ffb8 77e64829 mscorewks!Thread::intermediateThreadProc+0x49
0573ffec 00000000 kernel32!BaseThreadStart+0x34
```

So we run x86 WinDbg and get the better picture of **Nested Exceptions** (page 723):

```
0:010> !analyze -v

[...]

MANAGED_STACK: !dumpstack -EE
OS Thread Id: 0xc68 (15)
Current frame:
ChildEBP RetAddr Caller,Callee

EXCEPTION_OBJECT: !pe 16584f0
Exception object: 016584f0
Exception type: System.InvalidOperationException
Message: There is an error in XML document (12, 12182).
InnerException: System.IO.IOException, use !PrintException 0164f6dc to see more
[...]

StackTraceString: <none>
HRESULT: 80131509
There are nested exceptions on this thread. Run with -nested for details

EXCEPTION_OBJECT: !pe 164f6dc
Exception object: 0164f6dc
Exception type: System.IO.IOException
Message: Unable to read data from the transport connection: The connection was closed.
InnerException: <none>
[...]

StackTraceString: <none>
HRESULT: 80131620
There are nested exceptions on this thread. Run with -nested for details

MANAGED_OBJECT: !dumpobj 1655a38
Name: System.String
MethodTable: 790fd8c4
EEClass: 790fd824
Size: 270(0x10e) bytes
(C:\WINDOWS\assembly\GAC_32\mscorlib\2.0.0.0__[...]\mscorlib.dll)
String: Unable to read data from the transport connection: The connection was closed.

EXCEPTION_MESSAGE: Unable to read data from the transport connection: The connection was closed.

MANAGED_OBJECT_NAME: System.IO.IOException
```

There are other pattern instances of this kind when we need a **Platform-Specific Debugger**, for example, to do live debugging of an x86 process on an x64 machine (we need an x64 debugger) or we need to load an old 32-bit DLL extension (we need an x86 debugger) for a postmortem analysis.

## Pleiades

This is a cluster of modules in **!m** WinDbg command output that serve a similar function, like print drivers in the print spooler or Citrix printing services. Usually, we know that anyone of them could be at fault. Another example is a group of process modules in a complete memory dump serving the same function in separate terminal services sessions.

## Pre-Obfuscation Residue

This pattern is closely linked to **Packed** (page 775) and obfuscated code. Depending on a level of obfuscation and packing, some initial code and data structures, and patterns including fragments of strings may leak in post-obfuscation data giving a clue to intended software behavior:

```
0:000> s-sa 00000000`00fd4000 L6000
[...]
00000000`00fd943d  "o__"
00000000`00fd9449  "91!We"
00000000`00fd945d  "H5!"
00000000`00fd94d2  "zQ@"
00000000`00fd94dd  "ommandS"
00000000`00fd94f4  "IsDeb"
00000000`00fd94fd  "uggerP"
00000000`00fd9507  "Enc"
00000000`00fd950c  "v)3Po4t"
00000000`00fd9515  "DeXU"
00000000`00fd9520  "xFe"
00000000`00fd952a  "5Eb"
00000000`00fd9533  "SI=18kev"
00000000`00fd953e  "Z_1m"
00000000`00fd9547  "@IF"
[...]
```

## Problem Exception Handler

This pattern usually happens with custom exception handlers that are not written according to the prescribed rules (for example, a handler for a non-continuable exception<sup>171</sup>) or have other defects common to normal code. Please refer to the case study that models the former<sup>172</sup>.

In the example below we have a different stack trace epilog for a similar issue that shows a relationship with **Custom Exception Handler** (page 171):

```
0:000> kv 1000
ChildEBP RetAddr  Args to Child
0003300c 77af9904 77b8929c 792ea99b 00000000 ntdll!RtlAcquireSRWLockShared+0x1a
00033058 77af9867 00406ef8 00033098 000330a0 ntdll!RtlLookupFunctionTable+0x2a
000330a8 77af97f9 00406ef8 00000000 00000000 ntdll!RtlIsValidHandler+0x26
00033128 77b25dd7 01033140 00033154 00033140 ntdll!RtlDispatchException+0x10b
00033128 77b40726 01033140 00033154 00033140 ntdll!KiUserExceptionDispatcher+0xf (CONTEXT @ 00033154)
00033490 77b25dd7 010334a8 000334bc 000334a8 ntdll!RtlDispatchException+0x18a
00033490 77b40726 010334a8 000334bc 000334a8 ntdll!KiUserExceptionDispatcher+0xf (CONTEXT @ 000334bc)
000337f8 77b25dd7 01033810 00033824 00033810 ntdll!RtlDispatchException+0x18a
[...]
0012f228 77b40726 0112f240 0012f254 0012f240 ntdll!KiUserExceptionDispatcher+0xf (CONTEXT @ 0012f254)
0012f590 77b25dd7 0112f5a8 0012f5d8 0012f5a8 ntdll!RtlDispatchException+0x18a
0012f590 768bfbae 0112f5a8 0012f5d8 0012f5a8 ntdll!KiUserExceptionDispatcher+0xf (CONTEXT @ 0012f5d8)
0012f8f4 0059ecad 0eedfade 00000001 00000007 kernel32!RaiseException+0x58
WARNING: Stack unwind information not available. Following frames may be wrong.
0012f918 00473599 0eedfade 00000001 00000007 Application+0x19ecad
[...]
0012ff88 768cd0e9 7ffd000 0012ffd4 77b019bb Application+0x70f8
0012ff94 77b019bb 7ffd000 793f6617 00000000 kernel32!BaseThreadInitThunk+0xe
0012ffd4 77b0198e 011263c0 7ffd000 ffffffff ntdll!__RtlUserThreadStart+0x23
0012ffec 00000000 011263c0 7ffd000 00000000 ntdll!__RtlUserThreadStart+0x1b

0:000> !exchain
00033048: ntdll!_except_handler4+0 (77ac99fa)
0012ff78: Application+6ef8 (00406ef8)
0012ffc4: ntdll!_except_handler4+0 (77ac99fa)
0012ffe4: ntdll!FinalExceptionHandler+0 (77b66f9b)
Invalid exception stack at ffffffff
```

<sup>171</sup> <http://msdn.microsoft.com/en-us/library/aa259964.aspx>

<sup>172</sup> <http://www.debuggingexperts.com/modeling-exception-handling>

## Comments

The exception handler may belong to **Unloaded Module** (page 1041) causing **Stack Overflow** (page 912):

```
[...]
00000000`018cd4e0 00000000`772b96c5 ntdll!RtlpCallVectoredHandlers+0xac
00000000`018cd550 00000000`772c722a ntdll!RtlDispatchException+0x25
00000000`018cdbf0 000007fe`f992acc0 ntdll!KiUserExceptionDispatcher+0x2e
00000000`018ce188 00000000`772a38bb <Unloaded_ModuleA.dll>+0x3acc0
00000000`018ce190 00000000`772b96c5 ntdll!RtlpCallVectoredHandlers+0xac
00000000`018ce200 00000000`772c722a ntdll!RtlDispatchException+0x25
00000000`018ce8a0 00000000`772b3738 ntdll!KiUserExceptionDispatcher+0x2e
00000000`018cee40 00000000`772b3c64 ntdll!LdrpSearchResourceSection_U+0x16e
00000000`018cef70 00000000`7716e8b0 ntdll!LdrFindResource_U+0x44
00000000`018cef80 00000000`6eadac42 kernel32!FindResourceExW+0x84
[...]
00000000`018cffef0 00000000`772a6991 kernel32!BaseThreadInitThunk+0xd
00000000`018cff20 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

## Problem Module

Sometimes this pattern can help in troubleshooting. Problem modules (including process names) are components that due to their value adding behavior might break normal software behavior and, therefore, require some troubleshooting workarounds from minor configuration changes to their complete removal. Typical examples include memory optimization services<sup>173</sup> for terminal services environments or **Hooksware** (page 1175). We can see main process modules in the output of **!vm** or **!process 0 0** WinDbg commands. **!m** command will list module names such as DLLs from a process memory dump, **!mk** command can give us the list of kernel space modules (for example, drivers) from the kernel and complete memory dumps, and the following command lists all user space modules for each process in a complete memory dump:

```
!for_each_process ".process /r /p @@Process; !mu"
```

Of course, we can also try various **!m** command variants if we are interested in timestamps and module information.

## Comments

We can also search for the problem module name in **Stack Trace Collection** (page 944), for example, **!stacks 2 ModuleName** in the complete memory dumps or debugger logs.

---

<sup>173</sup> <https://support.citrix.com/article/CTX136287>

## Problem Vocabulary

One way to quickly check for something suspicious in a memory dump is to convert it to a debugger log<sup>174</sup> for example, **Stack Trace Collection**, (page 943) and search for textual strings such as “Waiting for”, “Terminate”, “Stop”, “Mutant”, “Exception”, “Crit”, “MessageBox”, and “SuspendCount”. The vocabulary, of course, is OS dependent, can have false positives, and can change over time. This pattern is similar to **Vocabulary Index** software trace analysis pattern<sup>175</sup>.

For example, recently in a complete memory dump involving lots of ALPC Wait Chains (page 1097) with potential inter-process deadlock, we found the following thread having long **Waiting Thread Time** (page 1137, exceeding ALPC threads waiting times) pointing to a process object to examine further:

```
THREAD ffffffa801338b950 Cid 02a0.7498 Teb: 000007fffffd8000 Win32Thread: 0000000000000000 WAIT:
(UserRequest) UserMode Non-Alertable
fffffa8012a39b30 ProcessObject
Not impersonating
DeviceMap fffff8a000008a70
Owning Process fffffa800a31d040 Image: smss.exe
Attached Process N/A Image: N/A
Wait Start TickCount 9752080 Ticks: 5334204 (0:23:09:06.937)
Context Switch Count 38
UserTime 00:00:00.000
KernelTime 00:00:00.000
Win32 Start Address ntdll!TppWorkerThread (0x000000007722fbc0)
Stack Init fffff88020259db0 Current fffff88020259900
Base fffff8802025a000 Limit fffff88020254000 Call 0
Priority 11 BasePriority 11 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
Kernel stack not resident.
Child-SP RetAddr Call Site
ffffff880`20259940 fffff800`01693f92 nt!KiSwapContext+0x7a
ffffff880`20259a80 fffff800`016967af nt!KiCommitThreadWait+0x1d2
ffffff880`20259b10 fffff800`01984b2e nt!KeWaitForSingleObject+0x19f
ffffff880`20259bb0 fffff800`0168df93 nt!NtWaitForSingleObject+0xde
ffffff880`20259c20 00000000`7726135a nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ fffff880`20259c20)
00000000`0048f648 00000000`48026517 ntdll!NtWaitForSingleObject+0xa
00000000`0048f650 00000000`480269c4 smss!SmpTerminateCSR+0xa3
00000000`0048f6a0 00000000`48023670 smss!SmpStopCsr+0x44
00000000`0048f6d0 00000000`77288137 smss!SmpApiCallback+0x338
00000000`0048f900 00000000`7722feff ntdll! ?? ::FNODOBFM::`string'+0x1f718
00000000`0048f990 00000000`77274a00 ntdll!TppWorkerThread+0x3f8
00000000`0048fc90 00000000`00000000 ntdll!RtlUserThreadStart+0x25
```

In that process, we could see **Blocking Module** (page 96) and recommended to contact its vendor.

<sup>174</sup> Architecture of CARE , Memory Dump Analysis Anthology, Volume 5, page 41

<sup>175</sup> Vocabulary Index, Memory Dump Analysis Anthology, Volume 4, page 349

## Process Factory

In my old days of PDP-11 system programming, we learned about the system call to spawn processes and wrote a program in assembly language that was spawning itself. This recursive spawning resulted in geometrical progression of running tasks and brought the RSX-11M system to a halt very quickly. Recently we observed the similar but non-recursive **Process Factory** pattern in one of memory dumps: explorer was relentlessly creating *application.exe* processes and by the time some effect was noticed there were more than 5,000 of them:

```
1: kd> !vm
[...]
5d20 application.exe 212 ( 848 Kb)
5d08 application.exe 212 ( 848 Kb)
5d04 application.exe 212 ( 848 Kb)
5cf8 application.exe 212 ( 848 Kb)
5cf0 application.exe 212 ( 848 Kb)
5ce8 application.exe 212 ( 848 Kb)
5cdc application.exe 212 ( 848 Kb)
5ccc application.exe 212 ( 848 Kb)
5cc8 application.exe 212 ( 848 Kb)
5cc0 application.exe 212 ( 848 Kb)
5ca8 application.exe 212 ( 848 Kb)
5c9c application.exe 212 ( 848 Kb)
5c98 application.exe 212 ( 848 Kb)
5c90 application.exe 212 ( 848 Kb)
5c88 application.exe 212 ( 848 Kb)
5c7c application.exe 212 ( 848 Kb)
5c70 application.exe 212 ( 848 Kb)
5c68 application.exe 212 ( 848 Kb)
5c64 application.exe 212 ( 848 Kb)
5c60 application.exe 212 ( 848 Kb)
5c50 application.exe 212 ( 848 Kb)
5c4c application.exe 212 ( 848 Kb)
5c44 application.exe 212 ( 848 Kb)
5c3c application.exe 212 ( 848 Kb)
5c34 application.exe 212 ( 848 Kb)
5c2c application.exe 212 ( 848 Kb)
5c24 application.exe 212 ( 848 Kb)
5c1c application.exe 212 ( 848 Kb)
5bf8 application.exe 212 ( 848 Kb)
5be0 application.exe 212 ( 848 Kb)
5bd4 application.exe 212 ( 848 Kb)
5bd0 application.exe 212 ( 848 Kb)
5ba4 application.exe 212 ( 848 Kb)
5b58 application.exe 212 ( 848 Kb)
5b50 application.exe 212 ( 848 Kb)
5b44 application.exe 212 ( 848 Kb)
5b38 application.exe 212 ( 848 Kb)
5b30 application.exe 212 ( 848 Kb)
5b04 application.exe 212 ( 848 Kb)
5af4 application.exe 212 ( 848 Kb)
5ad8 application.exe 212 ( 848 Kb)
5ad4 application.exe 212 ( 848 Kb)
5ac8 application.exe 212 ( 848 Kb)
5ac4 application.exe 212 ( 848 Kb)
```

```
5ab4 application.exe 212 ( 848 Kb)
5aa4 application.exe 212 ( 848 Kb)
5a9c application.exe 212 ( 848 Kb)
5a94 application.exe 212 ( 848 Kb)
5a8c application.exe 212 ( 848 Kb)
5a88 application.exe 212 ( 848 Kb)
5a74 application.exe 212 ( 848 Kb)
[...]

1: kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS 8b57f020 SessionId: none Cid: 0004 Peb: 00000000 ParentCid: 0000
  DirBase: cffb3020 ObjectTable: e1003da0 HandleCount: 3932.
  Image: System

PROCESS 8a9f8d88 SessionId: none Cid: 01b8 Peb: 7ffdf000 ParentCid: 0004
  DirBase: cffb3040 ObjectTable: e13e3f68 HandleCount: 111.
  Image: smss.exe

PROCESS 89f0d508 SessionId: 0 Cid: 01f0 Peb: 7ffd8000 ParentCid: 01b8
  DirBase: cffb3060 ObjectTable: e16bc370 HandleCount: 1292.
  Image: csrss.exe

PROCESS 89eea7c8 SessionId: 0 Cid: 0208 Peb: 7ffdde000 ParentCid: 01b8
  DirBase: cffb3080 ObjectTable: e14b4160 HandleCount: 564.
  Image: winlogon.exe

[...]

PROCESS 8607c020 SessionId: 1 Cid: 44c8 Peb: 7ffdc000 ParentCid: 4cf8
  DirBase: cffb7080 ObjectTable: e3c9fd38 HandleCount: 25407.
  Image: explorer.exe

[...]

PROCESS 85e1d020 SessionId: 1 Cid: 538c Peb: 7ffda000 ParentCid: 44c8
  DirBase: cffb8980 ObjectTable: e8065b20 HandleCount: 39.
  Image: application.exe

PROCESS 85c74610 SessionId: 1 Cid: 5394 Peb: 7ffd9000 ParentCid: 44c8
  DirBase: cffb89a0 ObjectTable: e6951878 HandleCount: 39.
  Image: application.exe

PROCESS 85c81020 SessionId: 1 Cid: 53a4 Peb: 7ffd7000 ParentCid: 44c8
  DirBase: cffb89c0 ObjectTable: e6d2f600 HandleCount: 39.
  Image: application.exe

PROCESS 85c6fb18 SessionId: 1 Cid: 53a8 Peb: 7ffd7000 ParentCid: 44c8
  DirBase: cffb89e0 ObjectTable: e54df078 HandleCount: 39.
  Image: application.exe
```

```
PROCESS 85c60020 SessionId: 1 Cid: 53bc Peb: 7ffd000 ParentCid: 44c8
  DirBase: cffb8a40 ObjectTable: e1214e90 HandleCount: 39.
  Image: application.exe

PROCESS 85c5d380 SessionId: 1 Cid: 53c8 Peb: 7ffde000 ParentCid: 44c8
  DirBase: cffb8a60 ObjectTable: e7baf638 HandleCount: 39.
  Image: application.exe

PROCESS 85c648b8 SessionId: 1 Cid: 53dc Peb: 7ffde000 ParentCid: 44c8
  DirBase: cffb8a80 ObjectTable: e759d060 HandleCount: 39.
  Image: application.exe

PROCESS 85c62528 SessionId: 1 Cid: 53e0 Peb: 7ffde000 ParentCid: 44c8
  DirBase: cffb8aa0 ObjectTable: e3b8fa00 HandleCount: 39.
  Image: application.exe

PROCESS 85c59d88 SessionId: 1 Cid: 53e8 Peb: 7ffdc000 ParentCid: 44c8
  DirBase: cffb8ac0 ObjectTable: e31751e0 HandleCount: 39.
  Image: application.exe

PROCESS 85c46d88 SessionId: 1 Cid: 542c Peb: 7ffd5000 ParentCid: 4d9c
  DirBase: cffb8b00 ObjectTable: e6fbc500 HandleCount: 136.
  Image: nlapplication.exe

PROCESS 85c3c020 SessionId: 1 Cid: 5464 Peb: 7ffdc000 ParentCid: 44c8
  DirBase: cffb8b40 ObjectTable: e218b948 HandleCount: 39.
  Image: application.exe

PROCESS 85c2a020 SessionId: 1 Cid: 546c Peb: 7ffdb000 ParentCid: 44c8
  DirBase: cffb8b60 ObjectTable: e639a8d0 HandleCount: 39.
  Image: application.exe

PROCESS 85c202c8 SessionId: 1 Cid: 5474 Peb: 7ffd7000 ParentCid: 44c8
  DirBase: cffb8b80 ObjectTable: e517caa8 HandleCount: 39.
  Image: application.exe

PROCESS 85c1b020 SessionId: 1 Cid: 547c Peb: 7ffd6000 ParentCid: 44c8
  DirBase: cffb8ba0 ObjectTable: e6c0cbc0 HandleCount: 39.
  Image: application.exe

PROCESS 85c1dd88 SessionId: 1 Cid: 5484 Peb: 7ffd5000 ParentCid: 44c8
  DirBase: cffb8bc0 ObjectTable: e4a42f68 HandleCount: 39.
  Image: application.exe

PROCESS 85d3ed88 SessionId: 1 Cid: 5488 Peb: 7ffd5000 ParentCid: 44c8
  DirBase: cffb8be0 ObjectTable: e68558f0 HandleCount: 39.
  Image: application.exe
```

[...]

We see that all created processes have the same parent process with PID 44c8 and when we inspect it we see many threads inside creating *application.exe* process:

```
1: kd> .process /r /p 8607c020
Implicit process is now 8607c020
Loading User Symbols

1: kd> !process 8607c020
PROCESS 8607c020 SessionId: 1 Cid: 44c8 Peb: 7ffdc000 ParentCid: 4cf8
  DirBase: cffb7080 ObjectTable: e3c9fd38 HandleCount: 25407.
  Image: explorer.exe
  VadRoot 88efec98 Vads 3445 Clone 0 Private 30423. Modified 71292. Locked 0.
  DeviceMap e3743340
    Token                      e29be5e0
    ElapsedTime                00:54:31.359
    UserTime                   00:00:19.234
    KernelTime                 00:04:04.828
    QuotaPoolUsage[PagedPool]  1075132
    QuotaPoolUsage[NonPagedPool] 137800
    Working Set Sizes (now,min,max) (15457, 50, 345) (61828KB, 200KB, 1380KB)
    PeakWorkingSetSize         48919
    VirtualSize                585 Mb
    PeakVirtualSize            978 Mb
    PageFaultCount             123488
    MemoryPriority              BACKGROUND
    BasePriority                8
    CommitCharge                49919

[...]

THREAD 84f25300 Cid 44c8.6288 Teb: 7ff8e000 Win32Thread: bc486830 READY
IRP List:
  88699110: (0006,0220) Flags: 00000884 Mdl: 00000000
Not impersonating
DeviceMap          e3743340
Owning Process    8607c020      Image:      explorer.exe
Wait Start TickCount 1327981      Ticks: 29 (0:00:00:00.453)
Context Switch Count 145332      LargeStack
UserTime           00:00:00.000
KernelTime         00:00:00.093
Win32 Start Address SHLWAPI!SHCreateThread (0x77ec3ea5)
Start Address kernel32!BaseThreadStartThunk (0x7c8217ec)
Stack Init a98e4000 Current a98e3700 Base a98e4000 Limit a98e0000 Call 0
Priority 8 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr
a98e3718 80833ec5 nt!KiSwapContext+0x26
a98e3744 80829bc0 nt!KiSwapThread+0x2e5
a98e378c 8087e0d8 nt!KeWaitForSingleObject+0x346
a98e37c4 8087e397 nt!ExpWaitForResource+0x30
a98e37e4 badff32a nt!ExAcquireResourceExclusiveLite+0x8d
a98e3808 badffe35 driverA+0x132a
a98e3824 bae00208 driverA+0x1e35
a98e3868 bae0e45a driverA+0x2208
a98e38a0 8081e095 driverA+0x1045a
a98e38b4 b972c73b nt!IoCallDriver+0x45
```

```
[...]
a98e38e8 b9b194e1 nt!IoCallDriver+0x45
[...]
a98e3940 b85cbf08 nt!IoCallDriver+0x45
a98e3968 b85bcfcc driverB!LowerDevicePassThrough+0x48
a98e398c b85bd63d driverB+0x6fcc
a98e3a24 b85cb167 driverB+0x763d
a98e3a34 b85cb1b7 driverB+0x15167
a98e3a5c 8081e095 driverB!DispatchPassThrough+0x48
a98e3a70 808fb13b nt!IoCallDriver+0x45
a98e3b58 80939c6a nt!IopParseDevice+0xa35
a98e3bd8 80935d9e nt!ObpLookupObjectName+0x5b0
a98e3c2c 808ece57 nt!ObOpenObjectByName+0xea
a98e3ca8 808ee0f1 nt!IopCreateFile+0x447
a98e3d04 808f1e31 nt!IoCreateFile+0xa3
a98e3d44 8088ad3c nt!NtOpenFile+0x27
a98e3d44 7c9485ec nt!KiFastCallEntry+0xfc (TrapFrame @ a98e3d64)
03bbda04 7c82bdf6 ntdll!KiFastSystemCall1Ret
03bbda2c 7c82dd9a kernel32!BasepSxsCreateStreams+0xe2
03bbda9c 7c82d895 kernel32!BasepSxsCreateProcessCsrMessage+0x136
03bbe2c4 7c8024a0 kernel32!CreateProcessInternalW+0x1943
03bbe2fc 7ca36750 kernel32!CreateProcessW+0x2c
03bbbed80 7ca36b45 SHELL32!_SHCreateProcess+0x387
03bbbedd4 7ca3617b SHELL32!CShellExecute::DoExecCommand+0xb4
03bbbede0 7ca35a76 SHELL32!CShellExecute::TryInvokeApplication+0x49
03bbbedf4 7ca3599f SHELL32!CShellExecute::ExecuteNormal+0xb1
03bbe08 7ca35933 SHELL32!ShellExecuteNormal+0x30
03bbe24 7ca452ff SHELL32!ShellExecuteExW+0x8d

1: kd> .thread 84e6a600
Implicit thread is now 84e6a600

1: kd> kv 100
[...]
03bbda04 7c82bdf6 001200a9 03bbda8c 03bbdb20 ntdll!KiFastSystemCall1Ret
03bbda2c 7c82dd9a 00000000 00000003 001200a9 kernel32!BasepSxsCreateStreams+0xe2
03bbda9c 7c82d895 00000000 00000000 03bbdc38 kernel32!BasepSxsCreateProcessCsrMessage+0x136
03bbe2c4 7c8024a0 00000000 01dafb9c 01dad904 kernel32!CreateProcessInternalW+0x1943
03bbe2fc 7ca36750 01dafb9c 01dad904 00000000 kernel32!CreateProcessW+0x2c
03bbbed80 7ca36b45 00010098 00000000 01daffac SHELL32!_SHCreateProcess+0x387
[...]

1: kd> du /c 100 01dafb9c
01dafb9c "C:\Program Files\App Package\Application.exe"
```

The difference between this pattern and similar **Handle Leak** (page 526) or **Zombie Processes** (page 1158) is the fact that leaks usually happen when a process forgets to close handles but **Process Factory** creates active processes which are full resource containers and consume system resources, for example, they all have the full handle table or consume GDI resources if they are GUI processes.

## Punctuated Memory Leak

An example of this pattern is somewhat similar to a large block allocation leak for process heap<sup>176</sup>. An application has some functionality, and after each command, its committed memory was increasing by 50 - 60 Mb. Three process memory dumps were taken with one before failures and then after each failure:

// Before failures

```
0:000> !address -summary
```

	RgnCount	Total Size	%ofBusy	%ofTotal
Free	267	76c50000 ( 1.856 Gb)	92.79%	
<unclassified>	270	4d6f000 ( 77.434 Mb)	52.45%	3.78%
Image	620	31bf000 ( 49.746 Mb)	33.70%	2.43%
Stack	60	1400000 ( 20.000 Mb)	13.55%	0.98%
ActivationContextData	48	35000 ( 212.000 kb)	0.14%	0.01%
NlsTables	1	23000 ( 140.000 kb)	0.09%	0.01%
TEB	20	14000 ( 80.000 kb)	0.05%	0.00%
CsrSharedMemory	1	5000 ( 20.000 kb)	0.01%	0.00%
PEB	1	1000 ( 4.000 kb)	0.00%	0.00%
 --- Type Summary (for busy) -----				
MEM_PRIVATE	296	3bca000 ( 59.789 Mb)	40.50%	2.92%
MEM_IMAGE	647	340c000 ( 52.047 Mb)	35.26%	2.54%
MEM_MAPPED	78	23ca000 ( 35.789 Mb)	24.24%	1.75%
 --- State Summary -----				
MEM_FREE	267	76c50000 ( 1.856 Gb)	92.79%	
MEM_RESERVE	125	5006000 ( 80.023 Mb)	54.21%	3.91%
MEM_COMMIT	896	439a000 ( 67.602 Mb)	45.79%	3.30%
 --- Protect Summary (for commit) -				
PAGE_EXECUTE_READ	125	1f2c000 ( 31.172 Mb)	21.12%	1.52%
PAGE_READONLY	363	1ee5000 ( 30.895 Mb)	20.93%	1.51%
PAGE_READWRITE	309	4c2000 ( 4.758 Mb)	3.22%	0.23%
PAGE_WRITECOPY	43	6a000 ( 424.000 kb)	0.28%	0.02%
PAGE_READWRITE PAGE_GUARD	40	4b000 ( 300.000 kb)	0.20%	0.01%
PAGE_EXECUTE_READWRITE	15	11000 ( 68.000 kb)	0.04%	0.00%
PAGE_EXECUTE	1	1000 ( 4.000 kb)	0.00%	0.00%

<sup>176</sup> Models of Software Behaviour, Memory Leak (Process Heap) Pattern, Memory Dump Analysis Anthology, Volume 5, page 315

--- Largest Region by Usage -----		Base Address	Region Size -----
Free		6130000	5fb70000 ( 1.496 Gb)
<b>&lt;unclassified&gt;</b>		abf000	13d1000 ( 19.816 Mb)
Image		75141000	879000 ( 8.473 Mb)
Stack		3290000	fd000 (1012.000 kb)
ActivationContextData		50000	4000 ( 16.000 kb)
NlsTables		7efb0000	23000 ( 140.000 kb)
TEB		7ef6f000	1000 ( 4.000 kb)
CsrSharedMemory		7efe0000	5000 ( 20.000 kb)
PEB		7efde000	1000 ( 4.000 kb)

//After the 1st failure

0:000> !address -summary

--- Usage Summary -----		RgnCount	Total Size -----	%ofBusy	%ofTotal
Free		267	7388c000 ( 1.805 Gb)	90.26%	
<b>&lt;unclassified&gt;</b>		272	8133000 ( 129.199 Mb)	64.80%	6.31%
Image		614	31bf000 ( 49.746 Mb)	24.95%	2.43%
Stack		60	1400000 ( 20.000 Mb)	10.03%	0.98%
ActivationContextData		48	35000 ( 212.000 kb)	0.10%	0.01%
NlsTables		1	23000 ( 140.000 kb)	0.07%	0.01%
TEB		20	14000 ( 80.000 kb)	0.04%	0.00%
CsrSharedMemory		1	5000 ( 20.000 kb)	0.01%	0.00%
PEB		1	1000 ( 4.000 kb)	0.00%	0.00%
--- Type Summary (for busy) -----		RgnCount	Total Size -----	%ofBusy	%ofTotal
<b>MEM_PRIVATE</b>		297	6f8e000 ( 111.555 Mb)	55.95%	5.45%
MEM_IMAGE		642	340c000 ( 52.047 Mb)	26.10%	2.54%
MEM_MAPPED		78	23ca000 ( 35.789 Mb)	17.95%	1.75%
--- State Summary -----		RgnCount	Total Size -----	%ofBusy	%ofTotal
MEM_FREE		267	7388c000 ( 1.805 Gb)	90.26%	
<b>MEM_COMMIT</b>		892	775e000 ( 119.367 Mb)	59.87%	5.83%
MEM_RESERVE		125	5006000 ( 80.023 Mb)	40.13%	3.91%
--- Protect Summary (for commit) -		RgnCount	Total Size -----	%ofBusy	%ofTotal
<b>PAGE_READWRITE</b>		314	38a3000 ( 56.637 Mb)	28.40%	2.77%
PAGE_EXECUTE_READ		125	1f2c000 ( 31.172 Mb)	15.63%	1.52%
PAGE_READONLY		363	1ee5000 ( 30.895 Mb)	15.49%	1.51%
PAGE_WRITECOPY		34	4d000 ( 308.000 kb)	0.15%	0.01%
PAGE_READWRITE PAGE_GUARD		40	4b000 ( 300.000 kb)	0.15%	0.01%
PAGE_EXECUTE_READWRITE		15	11000 ( 68.000 kb)	0.03%	0.00%
PAGE_EXECUTE		1	1000 ( 4.000 kb)	0.00%	0.00%

--- Largest Region by Usage -----		Base Address	Region Size -----
Free		94f4000	5c7ac000 ( 1.445 Gb)
<u>unclassified</u>		6130000	33c4000 ( 51.766 Mb)
Image		75141000	879000 ( 8.473 Mb)
Stack		3290000	fd000 (1012.000 kb)
ActivationContextData		50000	4000 ( 16.000 kb)
NlsTables		7efb0000	23000 ( 140.000 kb)
TEB		7ef6f000	1000 ( 4.000 kb)
CsrSharedMemory		7efe0000	5000 ( 20.000 kb)
PEB		7efde000	1000 ( 4.000 kb)

0:000> !address -f:VAR

BaseAddr	EndAddr+1	RgnSize	Type	State	Protect	Usage
-----						
[...]						
5e82000	5f70000	ee000	MEM_PRIVATE	MEM_RESERVE		<unclassified>
<b>6130000</b>	<b>94f4000</b>	<b>33c4000</b>	<b>MEM_PRIVATE</b>	<b>MEM_COMMIT</b>	<b>PAGE_READWRITE</b>	<b>&lt;unclassified&gt;</b>
74220000	74221000	1000	MEM_IMAGE	MEM_COMMIT	PAGE_READONLY	<unclassified>
[...]						

0:000> ? 33c4000/0n1024

Evaluate expression: 53008 = 0000cf10

//After the 2nd failure

0:000> !address -summary

--- Usage Summary -----		RgnCount	Total Size -----	%ofBusy	%ofTotal
Free		268	704c8000 ( 1.755 Gb)		87.74%
<u>unclassified</u>		273	b4f7000 ( 180.965 Mb)	<b>72.05%</b>	<b>8.84%</b>
Image		614	31bf000 ( 49.746 Mb)	19.81%	2.43%
Stack		60	1400000 ( 20.000 Mb)	7.96%	0.98%
ActivationContextData		48	35000 ( 212.000 kb)	0.08%	0.01%
NlsTables		1	23000 ( 140.000 kb)	0.05%	0.01%
TEB		20	14000 ( 80.000 kb)	0.03%	0.00%
CsrSharedMemory		1	5000 ( 20.000 kb)	0.01%	0.00%
PEB		1	1000 ( 4.000 kb)	0.00%	0.00%

--- Type Summary (for busy) -----		RgnCount	Total Size -----	%ofBusy	%ofTotal
<u>MEM_PRIVATE</u>		298	a352000 ( 163.320 Mb)	<b>65.03%</b>	<b>7.97%</b>
MEM_IMAGE		642	340c000 ( 52.047 Mb)	20.72%	2.54%
MEM_MAPPED		78	23ca000 ( 35.789 Mb)	14.25%	1.75%

--- State Summary -----		RgnCount	Total Size -----	%ofBusy	%ofTotal
MEM_FREE		268	704c8000 ( 1.755 Gb)		87.74%
<u>MEM_COMMIT</u>		893	ab22000 ( 171.133 Mb)	<b>68.14%</b>	<b>8.36%</b>
MEM_RESERVE		125	5006000 ( 80.023 Mb)	31.86%	3.91%

	RgnCount	Total Size	%ofBusy	%ofTotal
<u>PAGE_READWRITE</u>	315	6c67000 ( 108.402 Mb)	43.16%	5.29%
PAGE_EXECUTE_READ	125	1f2c000 ( 31.172 Mb)	12.41%	1.52%
PAGE_READONLY	363	1ee5000 ( 30.895 Mb)	12.30%	1.51%
PAGE_WRITECOPY	34	4d000 ( 308.000 kb)	0.12%	0.01%
PAGE_READWRITE PAGE_GUARD	40	4b000 ( 300.000 kb)	0.12%	0.01%
PAGE_EXECUTE_READWRITE	15	11000 ( 68.000 kb)	0.03%	0.00%
PAGE_EXECUTE	1	1000 ( 4.000 kb)	0.00%	0.00%

	Base Address	Region Size
Free	c8c4000	593dc000 ( 1.394 Gb)
<u>&lt;unclassified&gt;</u>	6130000	33c4000 ( 51.766 Mb)
Image	75141000	879000 ( 8.473 Mb)
Stack	3290000	fd000 (1012.000 kb)
ActivationContextData	50000	4000 ( 16.000 kb)
NlsTables	7efb0000	23000 ( 140.000 kb)
TEB	7ef6f000	1000 ( 4.000 kb)
CsrSharedMemory	7efe0000	5000 ( 20.000 kb)
PEB	7efde000	1000 ( 4.000 kb)

```
0:000> !address -f:VAR
```

BaseAddr	EndAddr+1	RgnSize	Type	State	Protect	Usage
5e82000	5f70000	ee000	MEM_PRIVATE	MEM_RESERVE		<unclassified>
6130000	94f4000	33c4000	MEM_PRIVATE	MEM_COMMIT	PAGE_READWRITE	<unclassified>
9500000	c8c4000	33c4000	MEM_PRIVATE	MEM_COMMIT	PAGE_READWRITE	<unclassified>
74220000	74221000	1000	MEM_IMAGE	MEM_COMMIT	PAGE_READONLY	<unclassified>
[...]						

The name of this pattern comes from the process of discrete large memory allocations that happen after specific actions or events. Between them, there are no visible or substantial increases in memory usage.

**Q****Quiet Dump**

We call such a memory dump where we don't see anything abnormal or even suspicious. For example, in such a dump its **Stack Trace Collection** (page 943) would not deviate from reference stack traces (page 1164), and we would not see any **Spiking Thread** (page 885).

## Quotient Stack Trace

A long time ago we introduced the notion of **Collapsed Stack Trace**<sup>177</sup> when all functions are removed from **Stack Trace** (page 926, for example, `kc` WinDbg command) and remaining repeated modules are removed similar to **Quotient Trace**<sup>178</sup> analysis pattern. It is similar to **Stack Trace Signature** (page 955) with frame count set to 1. We originally planned to call this pattern **Compact Stack (Trace)**, and it was on our list of possible future analysis patterns. This came to our attention again while preparing *Theoretical Software Diagnostics* book<sup>179</sup> and we decided to publish it under the name **Quotient Stack Trace** as a specialization of the more general trace and log analysis pattern.

Such a pattern may be useful for the analysis of module **Wait Chains** (page 1103).

---

<sup>177</sup> Memory Dump Analysis Anthology, Volume 3, page 381

<sup>178</sup> Ibid., Volume 9b, page 62

<sup>179</sup> Theoretical Software Diagnostics, ISBN-13: 978-1-908043986

## R

## Random Object

Sometimes we observe rare events when abnormal conditions that usually result in a system crash result in a milder problem, for example, a service is unavailable and not affecting other services and users. It was reported that an application was freezing during user session logoff. A complete memory dump was saved at that time, and its **Stack Trace Collection (!stacks command, 943)** shows the following suspicious thread in a user process (all other threads were waiting as normal):

```
0: kd> !stacks
Proc.Thread .Thread Ticks ThreadState Blocker
[...]
[89cfa960 Application.exe]
ea0.001c4c 89a11db0 0499cd1 Blocked DriverA+0x69db
[...]

0: kd> !thread 89a11db0 16
THREAD 89a11db0 Cid 0ea0.1c4c Peb: 7ffffd000 Win32Thread: bc347a48 WAIT: (Unknown) KernelMode Non-
Alertable
 89b87770 Unknown
  b97004ac NotificationEvent
IRP List:
  899e2668: (0006,0244) Flags: 00000884 Mdl: 00000000
Not impersonating
DeviceMap          daf62b28
Owning Process    89cfa960   Image:      Application.exe
Attached Process   N/A        Image:      N/A
Wait Start TickCount 909331   Ticks: 4824273 (0:20:56:19.265)
Context Switch Count 186      LargeStack
UserTime           00:00:00.015
KernelTime         00:00:00.093
*** ERROR: Module load completed but symbols could not be loaded for Application.exe
Win32 Start Address Application (0x00406b2a)
Start Address kernel32!BaseProcessStartThunk (0x77e617f8)
Stack Init b60ceb30 Current b60cdf10 Base b60cf000 Limit b60cb000 Call b60ceb34
Priority 10 BasePriority 10 PriorityDecrement 0
ChildEBP RetAddr Args to Child
b60cdf28 80833485 89a11db0 00000002 00000000 nt!KiSwapContext+0x26
b60cdf54 808294b9 dc399008 89b87748 b60ce01c nt!KiSwapThread+0x2e5
b60cdf88 b96d69db 00000002 b60cdfbc 00000001 nt!KeWaitForMultipleObjects+0x3d7
WARNING: Stack unwind information not available. Following frames may be wrong.
b60cdfe8 b96d719e 89b87748 dc399008 b60ce01c DriverA+0x69db
[...]
```

We notice “89b87770 Unknown” and double check what object the thread is waiting for:

```
0: kd> dp b60cdfbc L00000002
b60cdfbc 89b87770 b97004ac
```

These are exactly the same objects that are listed in **!thread** command output. We see that the second one is normal and resides in a nonpaged area:

```
0: kd> dt _DISPATCHER_HEADER b97004ac
ntdll!_DISPATCHER_HEADER
+0x000 Type : 0 ''
+0x001 Absolute : 0 ''
+0x001 NpxIrql : 0 ''
+0x002 Size : 0x4 ''
+0x002 Hand : 0x4 ''
+0x003 Inserted : 0 ''
+0x003 DebugActive : 0 ''
+0x000 Lock : 262144
+0x004 SignalState : 0
+0x008 WaitListHead : _LIST_ENTRY [ 0x89a11e70 - 0x89a11e70 ]

0: kd> !address b97004ac
a71e3000 - 13e1d000
Usage KernelSpaceUsageNonPagedSystem
```

The other looks like an invalid **Random Object** from the free nonpaged pool entry (it even says about itself that it is bad) that used to belong to Configuration Manager in the past:

```
0: kd> !pool 89b87770
Pool page 89b87770 region is Nonpaged pool
[...]
89b87540 size: 98 previous size: 40 (Allocated) File (Protected)
*89b875d8 size: 260 previous size: 98 (Free) *CMpa
  Pooltag CMpa : registry post apcs, Binary : nt!cm
89b87838 size: 28 previous size: 260 (Allocated) FSfm
[...]

0: kd> dd 89b87770
89b87770 bad0b0b0 00000000 00000000 00000000
89b87780 8a04be01 00000000 89b87788 89b87788
89b87790 00150006 e56c6946 8993e208 89ab96b8
89b877a0 00000000 00000000 bad0b0b0 c0000800
89b877b0 02110004 63426343 88ebbf80 00001000
89b877c0 00199000 00000000 8993e238 88d0d248
89b877d0 0019a000 00000000 00000000 00000000
89b877e0 00000000 00000000 00000000 00000000
```

```
0: kd> dt _DISPATCHER_HEADER 89b87770
ntdll!_DISPATCHER_HEADER
+0x000 Type          : 0xb0 "
+0x001 Absolute      : 0xb0 "
+0x001 NpxIrql       : 0xb0 "
+0x002 Size          : 0xd0 "
+0x002 Hand          : 0xd0 "
+0x003 Inserted      : 0xba "
+0x003 DebugActive   : 0xba "
+0x000 Lock          : -1160728400
+0x004 SignalState    : 0
+0x008 WaitListHead   : _LIST_ENTRY [ 0x0 - 0x0 ]
```

Now comes some counterfactual thinking. One possible scenario for a bugcheck: after *KeWaitForMultipleObjects* was called to wait for both objects to become signalled (3rd WAIT\_TYPE parameter) the free pool slot was allocated or coalesced with *SignalState* becoming nonzero by coincidence and other members becoming random values and then the second normal object becomes signalled when another thread sets the notification event.

## Raw Pointer

This pattern is about pointers without matching symbol files. They may be in the expected module range or in some other known module range in the form of *module + offset* or can be completely out of range of any module from the loaded module list and therefore just a number. For example, usually we have certain structures or arrays (tables) where we expect pointers with matching symbols such as IAT, IDT and 32-bit SSDT where an occurrence of a raw pointer immediately triggers a suspicion such as in this Import Address Table from *ProcessA*:

```
[...]
00000001`3f8a9048 00000000`76e282d0 ntdll!RtlSizeHeap
00000001`3f8a9050 00000000`76bf9070 kernel32!GetStringTypeWStub
00000001`3f8a9058 00000000`76c03580 kernel32!WideCharToMultiByteStub
00000001`3f8a9060 00000000`76e33f20 ntdll!RtlReAllocateHeap
00000001`3f8a9068 00000000`76e533a0 ntdll!RtlAllocateHeap
00000001`3f8a9070 00000000`76bfc420 kernel32!GetCommandLineWStub
00000001`3f8a9078 00000001`3f8a1638 ProcessA+0x10ac
00000001`3f8a9080 00000000`76c2cc50 kernel32!IsProcessorFeaturePresent
00000001`3f8a9088 00000000`76c02d60 kernel32!GetLastErrorStub
00000001`3f8a9090 00000000`76c02d80 kernel32!SetLastError
00000001`3f8a9098 00000000`76bf3ee0 kernel32!GetCurrentThreadIdStub
[...]
```

Note that structures are not limited to the above and can be any OS or even application specific structure where we have symbol files. Raw pointers that are outside of expected module range are covered in the next pattern.

## Reduced Symbolic Information

Sometimes we have reduced symbolic information for modules which can range from stripped or public symbol files to exported only function names. In such cases we can use API function prototypes, structure definitions and possible **String Parameters** (page 962) to make sense of function arguments:

```
0:000:x86> kv
ChildEBP RetAddr  Args to Child
0013fe34 75a1790d 0013fe74 00000000 00000000 user32!NtUserGetMessage+0x15
0013fe50 00fc148a 0013fe74 00000000 00000000 user32!GetMessageW+0x33
0013fe90 00fc16ec 00fc0000 00000000 00354082 notepad!WinMain+0xe6
0013ff20 758233aa 7efde000 0013ff6c 77059ef2 notepad!_initterm_e+0x1a1
0013ff2c 77059ef2 7efde000 57785ae5 00000000 kernel32!BaseThreadInitThunk+0xe
0013ff6c 77059ec5 00fc3689 7efde000 00000000 ntdll_77020000!__RtlUserThreadStart+0x70
0013ff84 00000000 00fc3689 7efde000 00000000 ntdll_77020000!__RtlUserThreadStart+0x1b
```

The first parameter of *GetMessage* API is a pointer to *MSG* structure:

```
0:000:x86> dt MSG 0013fe74
Symbol MSG not found.
```

From MSDN we find this structure definition:

```
typedef struct tagMSG {
    HWND    hwnd;
    UINT    message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD   time;
    POINT   pt;
} MSG, *PMSG, *LPMMSG;

0:000:x86> dc 0013fe74 L7
0013fe74 0007149c 00000113 0038a508 7287c5d6 .....8....r
0013fe84 2079a177 00000539 000001c0           w.y 9.....
```

## Reference Leak

Objects such as processes may be referenced internally in addition to using handles. If their reference counts are unbalanced, we may have **Reference Leak** pattern. For example, we have an instance of thousands of **Zombie Processes** (page 1158), but we don't see **Handle Leaks** (page 416) from their parent processes if we analyze ParentCids:

```
0: kd> !process 0 0
[...]
PROCESS ffffffa801009a060
SessionId: 0 Cid: 2e270 Peb: 7fffffdb000 ParentCid: 032c
DirBase: 12ba37000 ObjectTable: 00000000 HandleCount: 0.
Image: conhost.exe

PROCESS ffffffa8009b7e8e0
SessionId: 1 Cid: 2e0c8 Peb: 7fffffd9000 ParentCid: 10a0
DirBase: 21653e000 ObjectTable: 00000000 HandleCount: 0.
Image: taskmgr.exe

PROCESS ffffffa8009e7a450
SessionId: 0 Cid: 2e088 Peb: 7efdf000 ParentCid: 0478
DirBase: 107f02000 ObjectTable: 00000000 HandleCount: 0.
Image: AppA.exe

PROCESS ffffffa8009e794b0
SessionId: 0 Cid: 2e394 Peb: 7fffffd3000 ParentCid: 032c
DirBase: 210ffc000 ObjectTable: 00000000 HandleCount: 0.
Image: conhost.exe

PROCESS ffffffa8009ed4060
SessionId: 0 Cid: 2dee4 Peb: 7efdf000 ParentCid: 0478
DirBase: 11b7c7000 ObjectTable: 00000000 HandleCount: 0.
Image: AppB.exe

PROCESS ffffffa800a13bb30
SessionId: 0 Cid: 2e068 Peb: 7fffffd5000 ParentCid: 032c
DirBase: 1bb8c1000 ObjectTable: 00000000 HandleCount: 0.
Image: conhost.exe

PROCESS ffffffa80096f26b0
SessionId: 0 Cid: 2e320 Peb: 7efdf000 ParentCid: 0478
DirBase: 6ad4c000 ObjectTable: 00000000 HandleCount: 0.
Image: AppC.exe

PROCESS ffffffa8009c44060
SessionId: 0 Cid: 2e300 Peb: 7fffffd000 ParentCid: 032c
DirBase: 10df06000 ObjectTable: 00000000 HandleCount: 0.
Image: conhost.exe
[...]
```

```
0: kd> !object ffffffa800a13bb30
Object: ffffffa800a13bb30 Type: (ffffffa8006cecf30) Process
ObjectHeader: ffffffa800a13bb00 (new version)
HandleCount: 0 PointerCount: 1

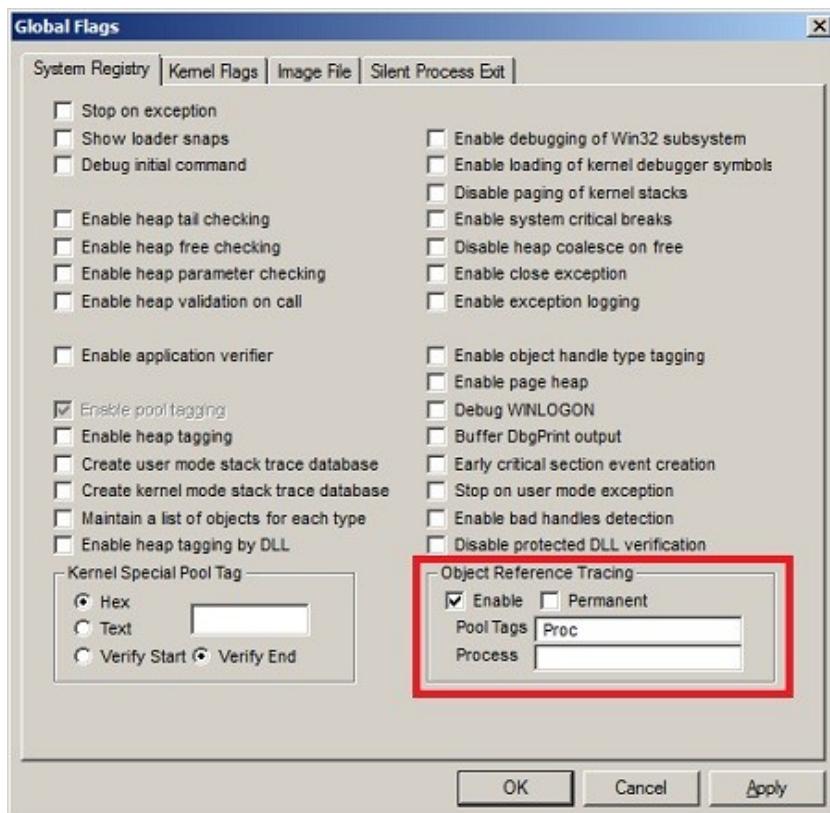
0: kd> !object ffffffa8009b7e8e0
Object: ffffffa8009b7e8e0 Type: (ffffffa8006cecf30) Process
ObjectHeader: ffffffa8009b7e8b0 (new version)
HandleCount: 0 PointerCount: 1
```

Such number of processes correlates with non-paged pool usage for process structures:

```
0: kd> !poolused 3
....
Sorting by NonPaged Pool Consumed

NonPaged Paged
Tag    Allocs   Frees     Diff   Used      Allocs   Frees   Diff   Used
Proc  55488     60      55428  80328320 0       0       0       0       0   Process objects , Binary: nt!ps
File  51733526 51708737 24789  7150416 0       0       0       0       0   File objects
[...]
```

Here we recommend enabling object reference tracing either using *gflags.exe* or directly modifying registry:



```
Key: HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\kernel
Value: ObTracePoolTags
Type: REG_SZ
Data: Proc
```

After troubleshooting or debugging we need to disable tracing because it consumes pool (another variant of **Instrumentation Side Effect** pattern (page 520) and may lead to similar **Insufficient Memory** pattern for stack trace database, page 563):

```
0: kd> !poolused 3
....
Sorting by NonPaged Pool Consumed

NonPaged Paged
Tag Allocs Frees Diff Used Allocs Frees Diff Used

ObRt 5688634 5676109 12525 4817288240 0 0 0 object reference stack tracing , Binary: nt!ob
Proc 22120 101 22019 25961168 0 0 0 Process objects , Binary: nt!ps
[...]
```

After enabling tracing we collect a complete memory dump (in the case of postmortem debugging) to analyze another variant of **Stack Trace** pattern (page 919) using **!obtrace WinDbg** command:

```
0: kd> !obtrace ffffffa800af9e220
Object: fffffa800af9e220
Image: AppD.exe
Sequence (+/-) Tag Stack
-----
ad377858 +1 Dflt nt! ?? ::NNGAKEGL::`string'+21577
nt!PspAllocateProcess+185
nt!NtCreateUserProcess+4a3
nt!KiSystemServiceCopyEnd+13

ad377882 +1 Dflt nt! ?? ::NNGAKEGL::`string'+1f9d8
nt!NtProtectVirtualMemory+119
nt!KiSystemServiceCopyEnd+13
nt!KiServiceLinkage+0
nt!RtlCreateUserStack+1e4
nt!PspAllocateThread+299
nt!NtCreateUserProcess+65d
nt!KiSystemServiceCopyEnd+13

ad377884 -1 Dflt nt! ?? ::FNODOBFM::`string'+4886e
nt!NtProtectVirtualMemory+161
nt!KiSystemServiceCopyEnd+13
nt!KiServiceLinkage+0
nt!RtlCreateUserStack+1e4
[...]
```

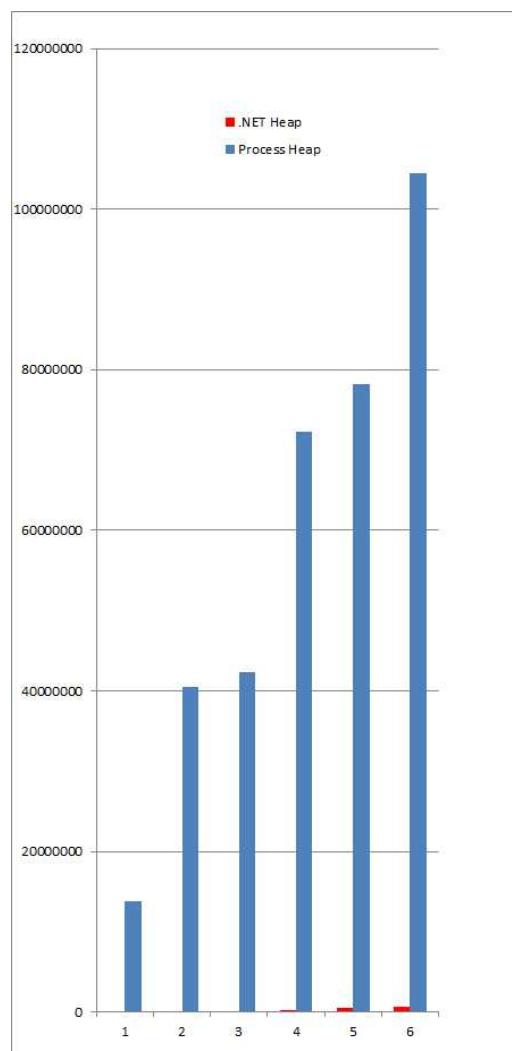
Analysis of such traces may be complicated due to **Truncated Stack Traces** (page 1015). **Origin Module** (page 771) may simplify counting in some cases.

## Regular Data

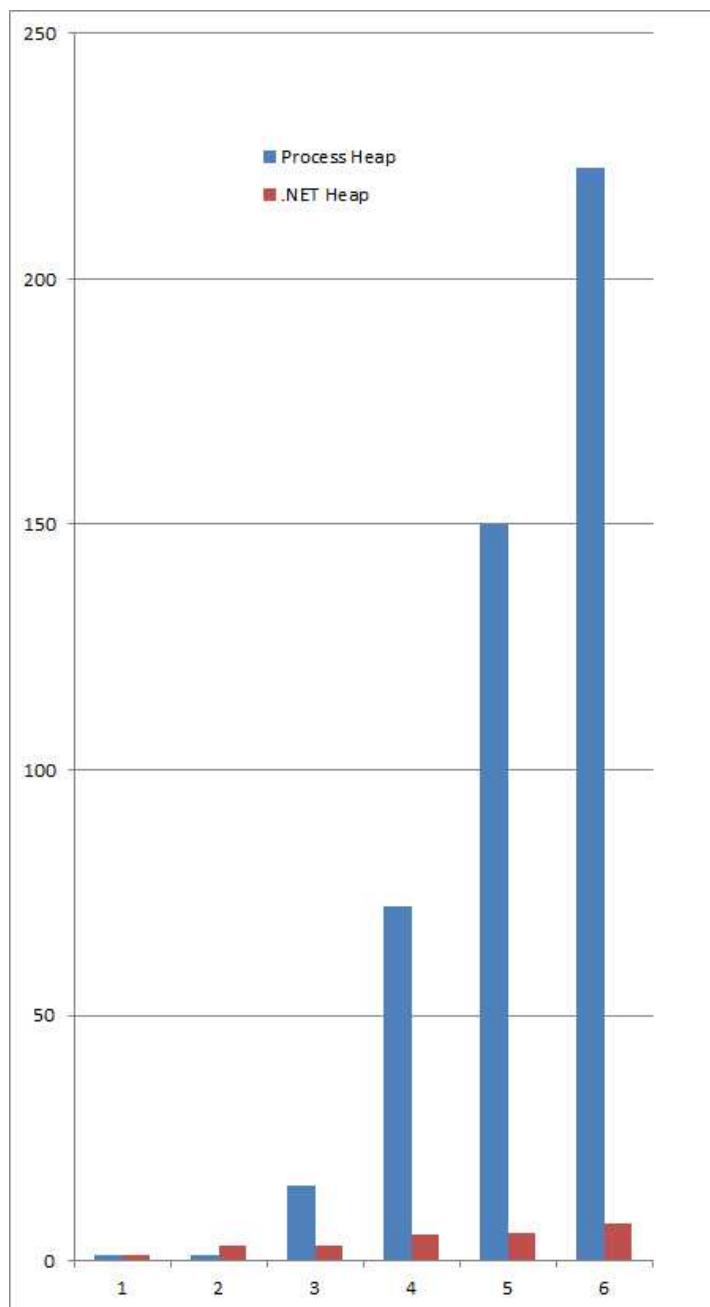
This pattern generalizes ASCII and UNICODE-type (00xx00yy) data found in memory to domain-specific data formats such as bitmaps and vector data. An example of the latter could be a sequence of ...0xxx0yyy... (xxx are triplets of hex digits). A typical usage of this pattern is an analysis of corrupt dynamic memory blocks (process heap, kernel pool) where continuity of regular data across block boundary points to a possible **Shared Buffer Overwrite** (page 868).

## Relative Memory Leak

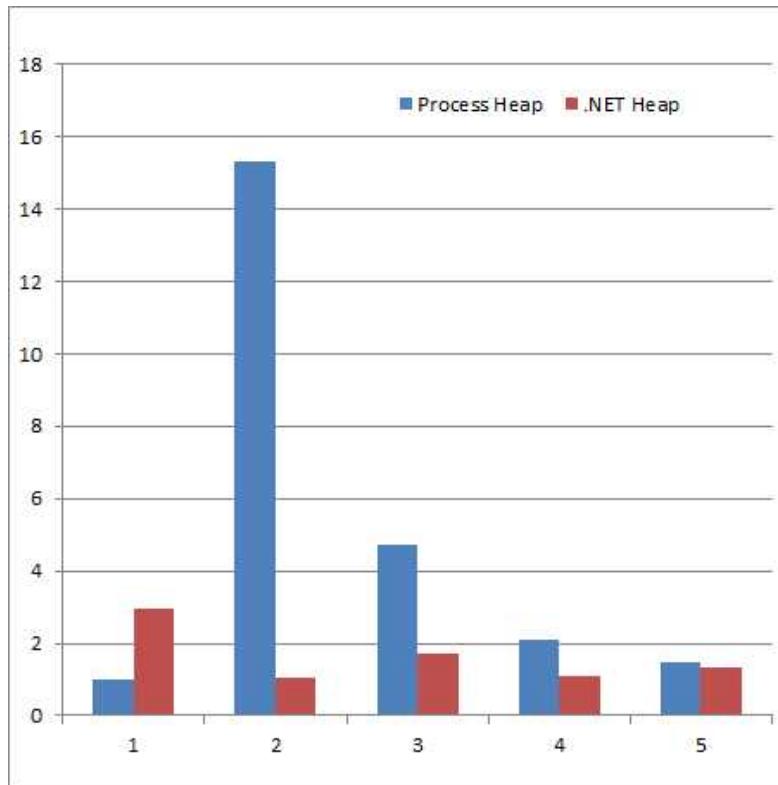
In the case of parallel **Memory Leak** for process heap (page 650) and .NET heap (page 636, or generally with several dynamic memory allocators) we are interested in relative growth to see whether they are interdependent, i.e. managed heap objects have pointers to process heap entries. When we have a set of consecutive memory dumps we can construct a table of heap sizes and plot the graph where the axes represent memory snapshot numbers and total heap size in bytes:



Unfortunately, the relative growth sizes can be disproportionate as the picture above shows. To overcome this, we can normalize size by the lowest corresponding heap size, i.e.  $S_n/S_1$ :



Still, this doesn't show the absence of correlation if there is no such. We can plot the relative growth, i.e.  $S_n/S_{n-1}$ ,  $n > 1$ :



The graph shows that there is no obvious correlation between **Relative Heap Leaks**. Similar graphs can be constructed for relative object distributions (**Object Distribution Anomaly**, .NET Heap, page 756).

## RIP Stack Trace

Injected code address may not be in address ranges of loaded modules. In such cases, in the execution call history, we would see plain EIP and RIP return addresses on stack traces. We call this pattern **RIP Stack Trace** partly because we have seen these addresses after something had gone wrong, and a process crashed:

```

0:005> k
ChildEBP RetAddr
02aec974 77655620 ntdll!KiFastSystemCallRet
02aec978 77683c62 ntdll!NtWaitForSingleObject+0xc
02aec9fc 77683d4b ntdll!RtlReportExceptionEx+0x14b
02aec9c3 7769fa87 ntdll!RtlReportException+0x3c
02aec9a50 7769fb0d ntdll!RtlpTerminateFailureFilter+0x14
02aec95c 775f9bdc ntdll!RtlReportCriticalFailure+0x6b
02aec970 775f4067 ntdll!_EH4_CallFilterFunc+0x12
02aec9a98 77655f79 ntdll!_except_handler4+0x8e
02aecabc 77655f4b ntdll!ExecuteHandler2+0x26
02aecb6c 77655dd7 ntdll!ExecuteHandler+0x24
02aecb6c 7769faf8 ntdll!KiUserExceptionDispatcher+0xf
02aece0 776a0704 ntdll!RtlReportCriticalFailure+0x5b
02aecf0 776a07f2 ntdll!RtlpReportHeapFailure+0x21
02aecf24 7766b1a5 ntdll!RtlpLogHeapFailure+0xa1
02aecf6c 7765730a ntdll!RtlpCoalesceFreeBlocks+0x4b9
02aed064 77657545 ntdll!RtlpFreeHeap+0x1e2
02aed080 75e47e4b ntdll!RtlFreeHeap+0x14e
02aed0c8 77037277 kernel32!GlobalFree+0x47
02aed0dc 774b4a1f ole32!ReleaseStgMedium+0x124
02aed0f0 77517feb urlmon!ReleaseBindInfo+0x4c
02aed100 774d9a87 urlmon!CINet::ReleaseCNetObjects+0x3d
02aed118 774d93f0 urlmon!CINetHttp::OnWininetRequestHandleClosing+0x60
02aed12c 76432078 urlmon!CINet::CINetCallback+0x2de
02aed274 76438f5d wininet!InternetIndicateStatus+0xfc
02aed2a4 7643937a wininet!HANDLE_OBJECT::~HANDLE_OBJECT+0xc9
02aed2c0 7643916b wininet!INTERNET_CONNECT_HANDLE_OBJECT::~INTERNET_CONNECT_HANDLE_OBJECT+0x209
02aed2cc 76438d5e wininet!HTTP_REQUEST_HANDLE_OBJECT::`vector deleting destructor'+0xd
02aed2dc 76434e72 wininet!HANDLE_OBJECT::Dereference+0x22
02aed2e8 76439419 wininet!DereferenceObject+0x21
02aed310 76439114 wininet!_InternetCloseHandle+0x9d
02aed330 0004aaaf wininet!InternetCloseHandle+0x11e
WARNING: Frame IP not in any known module. Following frames may be wrong.
02aed33c 774c5d25 0x4aaaf
02aed358 774c5d95 urlmon!CINet::TerminateRequest+0x82
02aed364 774c5d7c urlmon!CINet::MyUnlockRequest+0x10
02aed370 774c5d63 urlmon!CINetProtImpl::UnlockRequest+0x10
02aed37c 774c5d49 urlmon!CINetEmbdFilter::UnlockRequest+0x11
02aed388 774b743d urlmon!CINet::UnlockRequest+0x13
02aed394 774b73e1 urlmon!COInetProt::UnlockRequest+0x11
02aed3a8 774b7530 urlmon!CTransaction::UnlockRequest+0x36
02aed3b4 774b74e0 urlmon!CTransData::~CTransData+0x3a
02aed3c0 774b74c9 urlmon!CTransData::`scalar deleting destructor'+0xd
02aed3d8 774e221f urlmon!CTransData::Release+0x25
02aed3e0 774b6d0a urlmon!CReadOnlyStreamDirect::~CReadOnlyStreamDirect+0x1a
02aed3ec 774b7319 urlmon!CReadOnlyStreamDirect::`vector deleting destructor'+0xd
02aed404 774b72be urlmon!CReadOnlyStreamDirect::Release+0x25

```

```
02aed410 774b71f4 urlmon!CBinding::~CBinding+0xb9
02aed41c 774b71dd urlmon!CBinding::`scalar deleting destructor'+0xd
02aed434 6b20b0e8 urlmon!CBinding::Release+0x25
02aed448 6b20b0ba mshtml!ATL::AtlComPtrAssign+0x2b
02aed458 6b20b8de mshtml!ATL::CComPtr<IBindCallbackInternal>::operator=+0x15
02aed464 6b20b8aa mshtml!CBindingXSSFilter::TearDown+0x2b
02aed46c 6b20b887 mshtml!BindingXSSFilter_TearDown+0x19
02aed478 6b0da61a mshtml!CStreamProxy::Passivate+0x12
02aed484 6b0ddf3a mshtml!CBaseFT::Release+0x1d
02aed4ac 6b0e0b70 mshtml!CDwnBindData::TerminateBind+0x11d
02aed4b8 6b11a2a9 mshtml!CDwnBindData::TerminateOnApt+0x14
02aed4ec 6b105066 mshtml!GlobalWndOnMethodCall+0xfb
02aed50c 7742fd72 mshtml!GlobalWndProc+0x183
02aed538 7742fe4a user32!InternalCallWinProc+0x23
02aed5b0 7743018d user32!UserCallWinProcCheckWow+0x14b
02aed614 7743022b user32!DispatchMessageWorker+0x322
02aed624 6ecac1d5 user32!DispatchMessageW+0xf
02aef72c 6ec5337e ieframe!CTabWindow::_TabWindowThreadProc+0x54c
02aef7e4 760f426d ieframe!LCIETab_ThreadProc+0x2c1
02aef7f4 75e4d0e9 iertutil!CIsoScope::RegisterThread+0xab
02aef800 776319bb kernel32!BaseThreadInitThunk+0xe
02aef840 7763198e ntdll!__RtlUserThreadStart+0x23
02aef858 00000000 ntdll!_RtlUserThreadStart+0x1b
```

However, such addresses need to be checked whether they belong to .NET CLR **JIT Code** (page 591).

## Rough Stack Trace

This pattern is an example of more general **Execution Residue** (page 371) pattern or **Caller-n-Callee** (page 111) for managed space. It is just a collection of symbolic references (may also include **Coincidental Symbolic Information**, page 137) from the thread stack region or its fragment. In WinDbg, we can get it by using **dpS** command:

```
0:003> !teb
TEB at 000007fffffd6000
ExceptionList: 0000000000000000
StackBase: 0000000002450000
StackLimit: 000000000244b000
SubSystemTib: 0000000000000000
FiberData: 0000000000001e00
ArbitraryUserPointer: 0000000000000000
Self: 000007fffffd6000
EnvironmentPointer: 0000000000000000
ClientId: 00000000000047fc . 0000000000004824
RpcHandle: 0000000000000000
Tls Storage: 000007fffffd6058
PEB Address: 000007fffffd000
LastErrorValue: 0
LastStatusValue: c0000302
Count Owned Locks: 0
HardErrorMode: 0

0:003> dpS 000000000244b000 0000000002450000
000007fe`fd4a8a2e ole32!InternalVerifyStackAvailable+0x44 [d:\winmain\minio\safealloc\alloca.c @ 317]
000007fe`fd4a8a2e ole32!InternalVerifyStackAvailable+0x44 [d:\winmain\minio\safealloc\alloca.c @ 317]
000007fe`fd4a8a2e ole32!InternalVerifyStackAvailable+0x44 [d:\winmain\minio\safealloc\alloca.c @ 317]
00000000`771d5430 ntdll!RtlpInterceptorRoutines
00000000`771134d8 ntdll!RtlAllocateHeap+0x16c
00000000`770ec9c3 ntdll!RtlAppendUnicodeStringToString+0x53
00000000`76eaebef kernel32!Wow64RedirectKeyPathInternal+0xb7
00000000`770ec9c3 ntdll!RtlAppendUnicodeStringToString+0x53
00000000`771140fd ntdll!RtlFreeHeap+0x1a6
00000000`76eaec01 kernel32!ConstructKernelKeyPath+0x15f
00000000`76eaedd3 kernel32!Wow64NtOpenKey+0xee
00000000`771140fd ntdll!RtlFreeHeap+0x1a6
00000000`76ebc8aa kernel32!BaseRegOpenClassKeyFromLocation+0x3ba
00000000`76f3edf0 kernel32!\string'
00000000`771d5430 ntdll!RtlpInterceptorRoutines
00000000`76ebc9b9 kernel32!BaseReg GetUserPrefixLength+0xea
00000000`76f3ee38 kernel32!\string'
00000000`76f3edc8 kernel32!\string'
00000000`76ebc3a8 kernel32!BaseRegGetKeySemantics+0x1b8
00000000`771150d3 ntdll!RtlNtStatusToDosError+0x27
00000000`76eb36b7 kernel32!LocalBaseRegOpenKey+0x276
000007fe`fd4b6c79 ole32!GetUnquotedPath+0x29 [d:\w7rtm\com\ole32\com\objact\dllcache.cxx @ 2256]
000007fe`fd4b7019 ole32!CClassCache::CDllPathEntry::NegotiateDllInstantiationProperties2+0x145
[d:\w7rtm\com\ole32\com\objact\dllcache.cxx @ 3092]
00000000`771d5430 ntdll!RtlpInterceptorRoutines
00000000`771134d8 ntdll!RtlAllocateHeap+0x16c
00000000`77115cc4 ntdll!RtlpAllocateHeap+0xc12
000007fe`fdc10359 usp10!CUspShapingClient::AllocMem+0x49
```

```
000007fe`fdc48942 usp10!COtlsClient::AllocMem+0x12
000007fe`fdc48942 usp10!COtlsClient::AllocMem+0x12
000007fe`fdc1d4f1 usp10!UspFreeMem+0x61
000007fe`fdc4896e usp10!COtlsClient::FreeMem+0xe
000007fe`fdc6e817 usp10!ApplyFeatures+0xa17
000007fe`fdc6f2f2 usp10!ApplyLookup+0x592
000007fe`fdc48901 usp10!COtlsClient::GetDefaultGlyphs+0x131
000007fe`fdc60100 usp10!HangulEngineGetGlyphs+0x2c0
000007fe`fdc10359 usp10!CUSpShapingClient::AllocMem+0x49
000007fe`fdc48942 usp10!COtlsClient::AllocMem+0x12
000007fe`fdc10359 usp10!CUSpShapingClient::AllocMem+0x49
000007fe`fdc1d4f1 usp10!UspFreeMem+0x61
000007fe`fdc48942 usp10!COtlsClient::AllocMem+0x12
000007fe`fdc1d4f1 usp10!UspFreeMem+0x61
000007fe`fdc4896e usp10!COtlsClient::FreeMem+0xe
000007fe`fdc6e817 usp10!ApplyFeatures+0xa17
000007fe`fdc6aaa8 usp10!RePositionOtlGlyphs+0x238
000007fe`fdc48901 usp10!COtlsClient::GetDefaultGlyphs+0x131
000007fe`fdc60100 usp10!HangulEngineGetGlyphs+0x2c0
000007fe`fdc48798 usp10!COtlsClient::ReleaseOtlTable+0x78
000007fe`fdc6ae85 usp10!otlResourceMgr::detach+0xc5
00000000`7717c63e ntdll!EtwEventWriteNoRegistration+0xae
000007fe`fdc48a99 usp10!COtlsClient::Release+0x49
00000000`771150d3 ntdll!RtlNtStatusToDosError+0x27
00000000`7716bd85 ntdll!WaitForWerSvc+0x85
00000000`7717b94e ntdll!WerpAllocateAndInitializeSid+0xbe
00000000`7716bd90 ntdll! ?? ::FNODOBFM::`string'
00000000`77175dcf ntdll!WerpFreeSid+0x3f
00000000`7718123d ntdll!SendMessageToWERService+0x22d
00000000`77181260 ntdll! ?? ::FNODOBFM::`string'
00000000`77182308 ntdll!ReportExceptionInternal+0xc8
000007fe`fd061430 KERNELBASE!WaitForMultipleObjectsEx+0xe8
00000000`76ec1723 kernel32!WaitForMultipleObjectsExImplementation+0xb3
00000000`76f3b5e5 kernel32!WerpReportFaultInternal+0x215
00000000`76f3b767 kernel32!WerpReportFault+0x77
00000000`76f3b7bf kernel32!BasepReportFault+0x1f
00000000`76f3b9dc kernel32!UnhandledExceptionFilter+0x1fc
00000000`77118d7e ntdll!RtlpFindUnicodeStringInSection+0x50e
00000000`771198fc ntdll!LdrpFindLoadedD1l+0x10c
00000000`770e9caa ntdll!RtlDecodePointer+0x2a
00000000`770c0000 ntdll!RtlDeactivateActivationContext <PERF> (ntdll+0x0)
00000000`771e8180 ntdll!`string'+0xc040
00000000`771e818c ntdll!`string'+0xc04c
00000000`77153398 ntdll! ?? ::FNODOBFM::`string'+0x2365
00000000`770d85c8 ntdll!_C_specific_handler+0x8c
00000000`770c0000 ntdll!RtlDeactivateActivationContext <PERF> (ntdll+0x0)
00000000`770ec541 ntdll!RtlUserThreadStart+0x1d
00000000`770e9d2d ntdll!RtlpExecuteHandlerForException+0xd
00000000`77202dd0 ntdll!CsrPortMemoryRemoteDelta <PERF> (ntdll+0x142dd0)
00000000`770d91cf ntdll!RtlDispatchException+0x45a
00000000`76fadda0 kernel32!__PchSym_ <PERF> (kernel32+0x10dda0)
00000000`7711920a ntdll!RtlDosApplyFileIsolationRedirection_Ustr+0x3da
00000000`77202dd0 ntdll!CsrPortMemoryRemoteDelta <PERF> (ntdll+0x142dd0)
00000000`771e8180 ntdll!`string'+0xc040
00000000`770c0000 ntdll!RtlDeactivateActivationContext <PERF> (ntdll+0x0)
00000000`770ec541 ntdll!RtlUserThreadStart+0x1d
```

```

00000000`770c0000 ntdll!RtlDeactivateActivationContext <PERF> (ntdll+0x0)
00000000`77202dd0 ntdll!CsrPortMemoryRemoteDelta <PERF> (ntdll+0x142dd0)
00000000`771d7718 ntdll!LdrpDefaultExtension
00000000`770d852c ntdll!_C_specific_handler
00000000`771e8180 ntdll!`string'+0xc040
000007fe`ff3625c0 msctf!s_szCompClassName
00000000`770e7a33 ntdll!LdrpFindOrMapD1l+0x138
00000000`771192a8 ntdll!LdrpApplyFileNameRedirection+0x2d3
00000000`771d5430 ntdll!RtlpInterceptorRoutines
00000000`77113448 ntdll!RtlAllocateHeap+0xe4
00000000`76fd88b8 user32!GetPropW+0x4d
00000000`76fd7931 user32!IsWindow+0x9
00000000`770f41c8 ntdll!RtlpReAllocateHeap+0x178
000007fe`fb601381 uxtheme!CTHEMEWnd::_PreDefWindowProc+0x31
00000000`76eb59e0 kernel32!BaseThreadInitThunk
00000000`ffdbdb32 calc!CTimedCalc::Start+0xa9
00000000`ffd90000 calc!CCalculatorController::CCalculatorController <PERF> (calc+0x0)
00000000`ffe0ac64 calc!_dyn_tls_init_callback <PERF> (calc+0x7ac64)
00000000`76ea0000 kernel32!TestResourceDataMatchEntry <PERF> (kernel32+0x0)
00000000`76fadda0 kernel32!__PchSym_ <PERF> (kernel32+0x10dda0)
00000000`770c0000 ntdll!RtlDeactivateActivationContext <PERF> (ntdll+0x0)
00000000`77202dd0 ntdll!CsrPortMemoryRemoteDelta <PERF> (ntdll+0x142dd0)
00000000`76fd760e user32!RealDefWindowProcW+0x5a
000007fe`fb600037 uxtheme!operator delete <PERF> (uxtheme+0x37)
00000000`77111248 ntdll!KiUserExceptionDispatch+0x2e
000007fe`fb63fb40 uxtheme!$$VProc_ImageExportDirectory
00000000`ffdbdb27 calc!CTimedCalc::WatchDogThread+0xb2
00000000`76fe76c2 user32!DefDlgProcW+0x36
00000000`76fd9bef user32!UserCallWinProcCheckWow+0x1cb
00000000`76fd9b43 user32!UserCallWinProcCheckWow+0x99
00000000`76fd9bef user32!UserCallWinProcCheckWow+0x1cb
00000000`76fd72cb user32!DispatchClientMessage+0xc3
00000000`770e46b4 ntdll!NtDllDialogWndProc_W
00000000`ffdbdb27 calc!CTimedCalc::WatchDogThread+0xb2
00000000`77101530 ntdll!NtDllDispatchMessage_W
00000000`76fe505b user32!DialogBox2+0x2ec
00000000`ffd90000 calc!CCalculatorController::CCalculatorController <PERF> (calc+0x0)
00000000`76fe4edd user32!InternalDialogBox+0x135
00000000`ffdcedb0 calc!CTimedCalc::TimeOutDlgProc
00000000`ffd90000 calc!CCalculatorController::CCalculatorController <PERF> (calc+0x0)
00000000`76fe4f52 user32!DialogBoxIndirectParamAorW+0x58
00000000`ffdcedb0 calc!CTimedCalc::TimeOutDlgProc
00000000`ffd90000 calc!CCalculatorController::CCalculatorController <PERF> (calc+0x0)
00000000`76fdd476 user32!DialogBoxParamW+0x66
00000000`ffdcedb0 calc!CTimedCalc::TimeOutDlgProc
00000000`ffdbdaf0 calc!CTimedCalc::WatchDogThread+0x72
00000000`76eb59ed kernel32!BaseThreadInitThunk+0xd
00000000`770ec541 ntdll!RtlUserThreadStart+0x1d
00000000`76f3b7e0 kernel32!UnhandledExceptionFilter

```

The name for this pattern comes from rough sets<sup>180</sup> in mathematics.

<sup>180</sup> [http://en.wikipedia.org/wiki/Rough\\_set](http://en.wikipedia.org/wiki/Rough_set)

# S

## Same Vendor

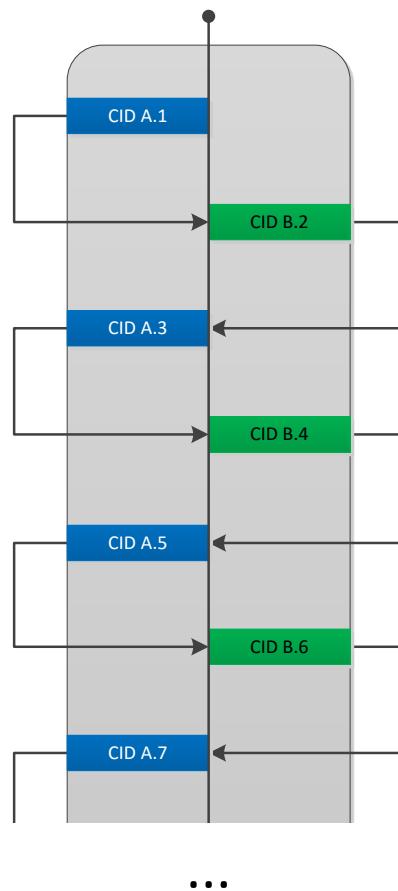
Sometimes we have very similar abnormal software behavior dispositions (like crashes with similar stack traces) for different applications or services. In such cases, we should also check application or service vendor and copyright in the output of **lmv** command. Similar to **Template Module** (page 997) **Same Vendor** pattern can be useful to relate such different incidents. Usually, in the same company, code and people reuse tends to distribute code fragments and code construction styles across different product lines, and, therefore, software defects may surface in different images. For example:

```
0:000> lmv m ApplicationA
start    end        module name
00400000 00d99000  ApplicationA  (deferred)
[...]
    Image name: ApplicationA.exe
    Timestamp:      [...]
    CheckSum:       00000000
[...]
    CompanyName:   CompanyA
    ProductName:   CompanyA Application
    LegalCopyright: Copyright (c) CompanyA
[...]

0:000> lmv m ApplicationB
start    end        module name
00400000 019d0000  ApplicationB C (no symbols)
    Image name: ApplicationB.exe
[...]
    CompanyName:   CompanyA
    ProductName:   ApplicationB
    LegalCopyright: Copyright (c) CompanyA
[...]
```

## Screwbolt Wait Chain

Here we introduce another **Wait Chain** (page 1082) pattern where a client thread makes a request and a created server thread servicing the request makes another request to the client which creates a new client thread to service the server request. The new client thread makes a request to the server again, and a new server thread is created which makes a new client request, and so on. The additional signs here may be an abnormal number of threads and possibly **Handle Leak** (page 416) pattern although the latter may be present only in a client or server process only. **Thread Age** (page 1001), **Waiting Thread Time** (page 1137), and common **Blocking Module** (page 96) patterns may be used to unwind the chain and diagnose the possible problem module and corresponding **Module Product Process** (page 697). The pattern is illustrated in this diagram:



Although we initially found this pattern related to LPC /ALPC IPC (page 1097) we think it is not limited to it and can occur in different client-server communication implementations.

## Self-Diagnosis

### Kernel Mode

This pattern is a kernel mode counterpart to **Self-Diagnosis** in user mode (page 847). It is just a collection of bugcheck codes where a problem is usually detected before corruption causes a fault, exception or trap. A typical example would be a detection of a failed assertion or **Corrupt Structures** (page 144) such as:

**BAD\_POOL\_HEADER (19)**

The pool is already corrupt at the time of the current request.

This may or may not be due to the caller.

The internal pool links must be walked to figure out a possible cause of the problem, and then special pool applied to the suspect tags or the driver verifier to a suspect driver.

Arguments:

Arg1: 00000020, a pool block header size is corrupt.

Arg2: 8b79d078, The pool entry we were looking for within the page.

Arg3: 8b79d158, The next pool entry.

Arg4: 8a1c0004, (reserved)

### Comments

Another example is this bugcheck:

**CRITICAL\_STRUCTURE\_CORRUPTION (109)**

This bugcheck is generated when the kernel detects that critical kernel code or data have been corrupted. There are generally three causes for a corruption:

- 1) A driver has inadvertently or deliberately modified critical kernel code or data. See <http://www.microsoft.com/whdc/driver/kernel/64bitPatching.mspx>
- 2) A developer attempted to set a normal kernel breakpoint using a kernel debugger that was not attached when the system was booted. Normal breakpoints, “bp”, can only be set if the debugger is attached at boot time. Hardware breakpoints, “ba”, can be set at any time.
- 3) A hardware corruption occurred, e.g. failing RAM holding kernel code or data.

Arguments:

Arg1: [...], Reserved

Arg2: [...], Reserved

Arg3: [...], Failure type dependent information

Arg4: 0000000000000002, Type of corrupted region, can be

0 : A generic data region

1 : Modification of a function or .pdata

2 : A processor IDT

3 : A processor GDT

4 : Type 1 process list corruption

5 : Type 2 process list corruption        6 : Debug routine modification

7 : Critical MSR modification

## Registry

This is a variant of **Self-Diagnosis** (kernel mode) pattern (page 844) for system configuration database (registry). Sometimes it is possible to see which part of it (hive) caused the problem. Here's an example involving possibly corrupt user profiles:

### REGISTRY\_ERROR (51)

Something has gone badly wrong with the registry. If a kernel debugger is available, get a stack trace. It can also indicate that the registry got an I/O error while trying to read one of its files, so it can be caused by hardware problems or filesystem corruption. It may occur due to a failure in a refresh operation, which is used only in by the security system, and then only when resource limits are encountered.

Arguments:

Arg1: 00000003, (reserved)

Arg2: 00000004, (reserved)

Arg3: **e82372f8**, depends on where Windows bugchecked, *may be pointer to hive*

Arg4: 00000000, depends on where Windows bugchecked, may be return code of HvCheckHive if the hive is corrupt.

0: kd> !reg hivelist

HiveAddr	Stable Length	Stable Map	Volatile Length	Volatile Map	MappedViews	PinnedViews	U(Cnt)	BaseBlock	FileName
e1008a68	13000	e1008ac8	1000	e1008c04	0	0	0	e1015000	<NONAME>
e101a4e0	901000	e1023000	40000	e101a67c	202	0	0	e101e000	SYSTEM
e1938188	d000	e19381e8	4000	e1938324	0	0	0	e193a000	<NONAME>
e1968290	8000	e19682f0	0	00000000	3	0	0	e1d39000	\SystemRoot\System32\Config\SAM
e1cab270	3d000	e1cab2d0	1000	e1cab40c	16	0	0	e1d32000	emRoot\System32\Config\SECURITY
e1c9f448	3f70000	e1e37000	1000	e1c9f5e4	256	0	0	e1d71000	temRoot\System32\Config\DEFAULT
e1d75a80	7d5d000	e1ee3000	23000	e1d75c1c	254	12	0	e1d37000	emRoot\System32\Config\SOFTWARE
e1ba30d0	37000	e1ba3130	1000	e1ba326c	17	0	0	e1b9e000	tings\NetworkService\ntuser.dat
e1ba8060	1000	e1ba80c0	0	00000000	1	0	0	e1b8e000	\Microsoft\Windows\UsrClass.dat
e1afc068	3b000	e1afc0c8	1000	e1afc204	17	0	0	e1b3d000	ettings\LocalService\ntuser.dat
e1d6e2a0	1000	e1d6e300	0	00000000	1	0	0	e1b39000	\Microsoft\Windows\UsrClass.dat
[...]									
e82372f8	106000	e8237358	0	00000000	55	4	0	e514c000	ings\User123\NTUSER.DAT
[...]									

0: kd> dt \_CMHIVE e82372f8

nt!\_CMHIVE

```
+0x000 Hive : _HHIVE
+0x2d0 FileHandles : [3] 0x80002234 Void
+0x2dc NotifyList : _LIST_ENTRY [ 0x0 - 0x0 ]
+0x2e4 HiveList : _LIST_ENTRY [ 0xe7a38d64 - 0xe4d9fc9c ]
+0x2ec HiveLock : _EX_PUSH_LOCK
+0x2f0 ViewLock : 0x877b0120 _KGUARDED_MUTEX
+0x2f4 WriterLock : _EX_PUSH_LOCK
+0x2f8 FlusherLock : _EX_PUSH_LOCK
+0xfc SecurityLock : _EX_PUSH_LOCK
+0x300 LRUViewListHead : _LIST_ENTRY [ 0xe6160170 - 0xe3d71978 ]
+0x308 PinViewListHead : _LIST_ENTRY [ 0xe2714fe0 - 0xe108d9e0 ]
+0x310 FileObject : 0x89ecf310 _FILE_OBJECT
+0x314 FilePath : _UNICODE_STRING "\Device\HarddiskVolumeX\Documents and Settings\User123\NTUSER.DAT"
+0x31c UserName : _UNICODE_STRING "\??\E:\Documents and Settings\User123\NTUSER.DAT"
+0x324 MappedViews : 0x37
+0x326 PinnedViews : 4
+0x328 UseCount : 0
```

```
+0x32c SecurityCount : 9
+0x330 SecurityCacheSize : 9
+0x334 SecurityHitHint : 0n0
+0x338 SecurityCache : 0xe74d5008 _CM_KEY_SECURITY_CACHE_ENTRY
+0x33c SecurityHash : [64] _LIST_ENTRY [ 0xe3f80228 - 0xe5901ef0 ]
+0x53c UnloadEvent : (null)
+0x540 RootKcb : (null)
+0x544 Frozen : 0 "
+0x548 UnloadWorkItem : (null)
+0x54c GrowOnlyMode : 0 "
+0x550 GrowOffset : 0
+0x554 KcbConvertListHead : _LIST_ENTRY [ 0xe823784c - 0xe823784c ]
+0x55c KnodeConvertListHead : _LIST_ENTRY [ 0xe8237854 - 0xe8237854 ]
+0x564 CellRemapArray : (null)
+0x568 Flags : 1
+0x56c TrustClassEntry : _LIST_ENTRY [ 0xe8237864 - 0xe8237864 ]
+0x574 FlushCount : 0
+0x578 CreatorOwner : (null)
```

## User Mode

Sometimes patterns like **Message Box** (page 660) and **Stack Trace** semantics (page 926) reveal another pattern that we call **Self-Diagnosis**, that may or may not result in **Self-Dump** (page 850). The diagnostic message may reveal the problem internally detected by the runtime environment.

Consider the following stack trace (shown in smaller font for visual clarity):

```
0:000> kv
ChildEBP RetAddr  Args to Child
0012e8c0 77f4bf53 77f4610a 00000000 00000000 ntdll!KiFastSystemCallRet
0012e8f8 77f3965e 000101a2 00000000 00000001 user32!NtUserWaitMessage+0xc
0012e920 77f4f762 77f30000 00151768 00000000 user32!InternalDialogBox+0xd0
0012eb0 77f4f047 0012ed3c 00000000 ffffffff user32!SoftModalMessageBox+0x94b
0012ed30 77f4eec9 0012ed3c 00000028 00000000 user32!MessageBoxWorker+0x2ba
0012ed88 77f87d0d 00000000 001511a8 0014ef50 user32!MessageBoxTimeoutW+0x7a
0012edbc 77f742c8 00000000 0012ee70 1001d7d4 user32!MessageBoxTimeoutA+0x9c
0012eddc 77f742a4 00000000 0012ee70 1001d7d4 user32!MessageBoxExA+0x1b
0012edf8 10014c9a 00000000 0012ee70 1001d7d4 user32!MessageBoxA+0x45
WARNING: Stack unwind information not available. Following frames may be wrong.
0012ee2c 10010221 0012ee70 1001d7d4 00012010 component!Error+0x7e4a
...

```

Dumping the message box message and its title shows that Visual C++ runtime detected a buffer overflow condition:

```
0:000> da 0012ee70
0012ee70 "Buffer overrun detected!..Progra"
0012ee90 "m: E:\W\program.exe..A buffer ov"
0012eeb0 "errun has been detected which ha"
0012eed0 "s corrupted the program's.intern"
0012eef0 "al state. The program cannot sa"
0012ef10 "fely continue execution and must"
0012ef30 ".now be terminated.."

0:000> da 1001d7d4
1001d7d4 "Microsoft Visual C++ Runtime Lib"
1001d7f4 "rary"
```

## Comments

---

One of the questions asked:

**Q.** Sometimes we get something like “Microsoft Visual C++ Runtime error” dialog when a program crashes. No *drwtsn32.log* file will be created followed by this dialog. How are we going to deal with this?

**A.** Here we can save the dump manually either by using MS *userdump*, *procdump*, or using Task Manager in Vista/W2K8<sup>181</sup>

An additional example is from IE:

```
0:000> kc

user32!NtUserMessageCall
user32!SendMessageWorker
user32!SendMessageW
ieframe!CTabWindow::_MakeBlockingCallToHungTabToTriggerNtUserHangDetection
ieframe!CTabWindow::MarkTabAsHung
ieframe!FrameTabWndProc
user32!InternalCallWinProc
user32!UserCallWinProcCheckWow
user32!DispatchMessageWorker
user32!DispatchMessageW
ieframe!CBrowserFrame::FrameMessagePump
ieframe!BrowserThreadProc
ieframe!BrowserNewThreadProc
ieframe!SHOpenFolderWindow
ieframe!IEWinMainEx
ieframe!IEWinMain
ieframe!LCIEStartAsFrame
iexplore!wWinMain
iexplore!_initterm_e
kernel32!BaseThreadInitThunk
ntdll_77dc0000!_RtlUserThreadStart
ntdll_77dc0000!_RtlUserThreadStart
```

Another example: runtime library *abort()*:

```
# 2 Id: acc.13b0 Suspend: 0 Teb: 7efa9000 Unfrozen
ChildEBP RetAddr
0333f4cc 768c15f7 ntdll!NtWaitForMultipleObjects+0x15
0333f568 762c19f8 KERNELBASE!WaitForMultipleObjectsEx+0x100
0333f5b0 762c4200 kernel32!WaitForMultipleObjectsExImplementation+0xe0
0333f5cc 762e80a4 kernel32!WaitForMultipleObjects+0x18
0333f638 762e7f63 kernel32!WerFaultInternal+0x186
0333f64c 762e7858 kernel32!WerFault+0x70
```

---

<sup>181</sup> Proactive Crash Dumps, Memory Dump Analysis Anthology, Volume 1, page 39

```
0333f65c 762e77d7 kernel32!BaseReportFault+0x20
0333f6e8 733f267a kernel32!UnhandledExceptionFilter+0x1af
0333fa20 747371ed msrv90!abort+0x10f
WARNING: Stack unwind information not available. Following frames may be wrong.
0333fab8 77a938aa DispatcherProxy!Singleton<_dispatcherproxyreceiver>::instance+0x5ed
0333fbcc 77a99f45 ntdll!RtlpFreeHeap+0xb7a
0333fbe4 00000000 ntdll!_RtlUserThreadStart+0x1b
```

Another example is Windows 8-style security interrupts:

```
0:112> .exr -1
ExceptionAddress: 00007ffffdb82513 (eModel!wil::details::ReportFailure+0x0000000000000ab)
ExceptionCode: c0000409 (Security check failure or stack buffer overrun)
ExceptionFlags: 00000001
NumberParameters: 1
Parameter[0]: 0000000000000007
Subcode: 0x7 FAST_FAIL_FATAL_APP_EXIT

0:112> kc 4
# Call Site
00 eModel!wil::details::ReportFailure
01 eModel!wil::details::ReportFailure_Hr
02 eModel!wil::details::in1diag3::FailFast_Hr
03 eModel!SpartanCore::LayerOwner::ConnectToLayerStateSystem

0:112> r
Last set context:
rax=0000000000000000 rbx=0000000000000000 rcx=0000000000000007
rdx=0000004d31b8421c rsi=00000000000331c rdi=0000000000000004
rip=00007ffffdb82513 rsp=0000004d32c4bff0 rbp=00000000000331c
r8=0000000000000003 r9=0000004d31b8421c r10=0000004d31b841a8
r11=0000004d32c4bf60 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0 nv up ei pl zr na po nc
cs=0033 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000246
eModel!wil::details::ReportFailure+0xab:
00007fff`fdb82513 cd29 int 29h
```

## Self-Dump

Sometimes processes dump themselves using Microsoft DbgHelp API when they encounter internal errors or for other debugging purposes. We separate that from unhandled exceptions which usually cause an external postmortem debugger process to dump memory contents as explained in “Who Calls the Postmortem Debugger?”<sup>182</sup> and “Inside Vista Error Reporting”<sup>183</sup> articles.

It is important to understand that a process can dump itself not only as a reaction to hardware and **Software Exceptions** (page 875) but for any reasons that came to mind of an application designer. In any case, it is useful to look at raw stack dump of all threads<sup>184</sup> and search for the first chance exceptions like c0000005 and **Custom Exception Handlers** (page 169).

We can consider this pattern as a specialized version of **Special Stack Trace** pattern (page 882). The typical thread stack might look like and might be **Incorrect Stack Trace** (page 499) that requires manual reconstruction<sup>185</sup>:

```
0:012> kL
ChildEBP RetAddr
0151f0c4 77e61f0c ntdll!KiFastSystemCallRet
0151f0d4 08000000 kernel32!CreateFileMappingW+0xc8
WARNING: Frame IP not in any known module. Following frames may be wrong.
0151f0f0 7c82728b 0x8000000
0151f0f4 77e63e41 ntdll!NtMapViewOfSection+0xc
0151f12c 77e6440c kernel32!MapViewOfFileEx+0x71
0151f14c 7c826d2b kernel32!MapViewOfFile+0x1b
0151f1d4 028ca67c ntdll!ZwClose+0xc
0151f2a4 028cc2f1 dbghelp!GenGetClrMemory+0xec
0151f2b4 028c8e55 dbghelp!Win32LiveSystemProvider::CloseMapping+0x11
0151f414 00000000 dbghelp!GenAllocateModuleObject+0x3c5
```

---

<sup>182</sup> Who Calls the Postmortem Debugger?, Memory Dump Analysis Anthology, Volume 1, page 113

<sup>183</sup> Inside Vista Error Reporting, Memory Dump Analysis Anthology, Volume 1, page 117

<sup>184</sup> Raw Stack Dump of All Threads (Process Dump), Memory Dump Analysis Anthology, Volume 1, page 231

<sup>185</sup> Manual Stack Trace Reconstruction, Memory Dump Analysis Anthology, Volume 1, page 157

Raw stack data should reveal DbgHelp API calls:

```

0151f944 00000000
0151f948 00000000
0151f94c 0151f9c0
0151f950 028c7662 dbghelp!MiniDumpWriteDump+0x1b2
0151f954 ffffffff
0151f958 00000cb0
0151f95c 00c21ea8
0151f960 00c21f88
0151f964 00c21e90
0151f968 00c21fa0
0151f96c 00000002
0151f970 00000000
0151f974 00000000
0151f978 00000000
0151f97c 7c829f60 ntdll!CheckHeapFillPattern+0x64
0151f980 ffffffff
0151f984 7c829f59 ntdll!RtlFreeHeap+0x70f
0151f988 7c34218a ms脆r71!free+0xc3
0151f98c 00000000
0151f990 00000000
0151f994 00c21e90
0151f998 00c21fa0
0151f99c 00c21ea8
0151f9a0 00c21f88
0151f9a4 00000002
0151f9a8 021a00da
0151f9ac 001875d8
0151f9b0 7c3416db ms脆r71!_nh_malloc+0x10
0151f9b4 0151f998
0151f9b8 001875d8
0151f9bc 00000000
0151f9c0 0151fbe4
0151f9c4 57b77d01 application!write_problem_report+0x18d1
0151f9c8 ffffffff
0151f9cc 00000cb0
0151f9d0 00000718
0151f9d4 00000002
0151f9d8 00000000
0151f9dc 00000000
0151f9e0 00000000
0151f9e4 00029722
0151f9e8 0151fc20

```

Partially reconstructed stack trace may look like this:

```

0:012> k L=0151f94c
ChildEBP RetAddr
0151f0c4 77e61f0c ntdll!KiFastSystemCallRet
0151f94c 028c7662 kernel32!CreateFileMappingW+0xc8
0151f9c0 57b77d01 dbghelp!MiniDumpWriteDump+0x1b2
0151fbe4 57b77056 application!write_problem_report+0x18d1
0151fc30 579c83af application!TerminateThread+0x18

```

## Comments

---

Sometimes, **Truncated Stack Traces** (page 1017) hide **Self-Dump** processing:

```
4 Id: 10e4.2838 Suspend: 0 Teb: 7efa6000 Unfrozen  
ChildEBP RetAddr  
026beb1c 00000000 ntdll!NtGetContextThread+0x12
```

We need to examine raw stack **Execution Residue** (page 371) to see the presence of DbgHelp API.

## Semantic Split

**Semantic Split** is the partitioning of anomalous debugger output from debugging commands into several disjoint or weakly linked classes. It is better characterized as the partition of a memory analysis pattern, for example, **Blocked Thread** pattern (page 80), into classes with different semantics, for example, blocked display threads and blocked remote share threads. Here is one short example of it found in the output of **!locks** and **!process 0 3f** WinDbg commands from a complete memory dump forced on a hanging server. The output shows several blocked threads and **Wait Chains** (page 1089) of executive resources (some shared locks have been removed for clarity):

```
0: kd> !locks
**** DUMP OF ALL RESOURCE OBJECTS ****

Resource @ 0x88eeeaf0    Exclusively owned
Contention Count = 809254
NumberOfExclusiveWaiters = 4
Threads: 88a26db0-01<*>
Threads Waiting On Exclusive Access:
    88c6b6d0      8877b020      88a7e020      88938db0

Resource @ 0x88badb20    Exclusively owned
Contention Count = 9195
NumberOfExclusiveWaiters = 4
Threads: 88842020-02<*>
Threads Waiting On Exclusive Access:
    88a8b170      89069450      88c4d020      88a26db0

Resource @ 0x88859cc0    Exclusively owned
Contention Count = 51021
NumberOfExclusiveWaiters = 8
Threads: 886f1c50-01<*>
Threads Waiting On Exclusive Access:
    88e04db0      886785c0      8851edb0      896ee890
    8869fb50      886d6498      889aa918      88c2da38

Resource @ 0x881cc138    Exclusively owned
Contention Count = 173698
NumberOfExclusiveWaiters = 4
Threads: 87e72598-01<*>
Threads Waiting On Exclusive Access:
    88392020      8918c8d8      88423020      880eba50

Resource @ 0x884ffab0    Exclusively owned
Contention Count = 3363
NumberOfExclusiveWaiters = 2
Threads: 8807c5b8-02<*>
Threads Waiting On Exclusive Access:
    87e72598      881c12a8
```

```

Resource @ 0x87cd6d48    Exclusively owned
  Contention Count = 242361
  NumberOfExclusiveWaiters = 5
  Threads: 87540718-01<*>
  Threads Waiting On Exclusive Access:
    878ceaf0      8785ac50      8884a7b8      87c4ca28
    89ab5db0

Resource @ 0x87c44d08    Exclusively owned
  Contention Count = 2560
  NumberOfExclusiveWaiters = 1
  Threads: 87540718-01<*>
  Threads Waiting On Exclusive Access:
    87c4e468

Resource @ 0x87bf51d8    Exclusively owned
  Contention Count = 3
  NumberOfSharedWaiters = 3
  Threads: 89e76db0-01<*> 8739ac50-01      86f5d1c8-01      870f4db0-01

Resource @ 0x888bfc38    Exclusively owned
  Contention Count = 3
  NumberOfSharedWaiters = 3
  Threads: 88a10db0-01<*> 86c94198-01      86dac598-01      86d85c50-01

```

The first group of locks (bold above) shows various problems with *ComponentA* module:

```

0: kd> !thread 88842020 1f
THREAD 88842020  Cid 1cf8.1b28  Teb: 7fffd8000 Win32Thread: bc25e8c0 WAIT: (Unknown) KernelMode Non-
Alertable
  88842098  NotificationTimer
Not impersonating
DeviceMap          e3813fd0
Owning Process    888c5d88      Image:        ApplicationA.exe
Attached Process   N/A          Image:        N/A
Wait Start TickCount 1163714      Ticks: 0
Context Switch Count 35781          LargeStack
UserTime           00:00:00.453
KernelTime         00:00:01.109
Win32 Start Address 0x77ec3ea5
Start Address kernel32!BaseThreadStartThunk (0x7c8217ec)
Stack Init b5cc4bd0 Current b5cc4614 Base b5cc5000 Limit b5cbf000 Call b5cc4bd8
Priority 9 BasePriority 9 PriorityDecrement 0
ChildEBP RetAddr
b5cc462c 80833ec5 nt!KiSwapContext+0x26
b5cc4658 80829069 nt!KiSwapThread+0x2e5
b5cc46a0 bf8981b3 nt!KeDelayExecutionThread+0x2ab
b5cc46c4 bf898422 ComponentA!LockGUITHandle+0x6d
[...]
b5cc49e8 80a63456 nt!KiFastCallEntry+0xcd
[...]

```

The second group of locks (in bold italics above) shows the problem with *ComponentB* module:

```
0: kd> !thread 89e76db0 1f
THREAD 89e76db0 Cid 0004.0624 Teb: 00000000 Win32Thread: 00000000 WAIT: (Unknown) KernelMode Alertable
 89e76e28 NotificationTimer
Not impersonating
DeviceMap          e1006e10
Owning Process    8b581648   Image:      System
Attached Process   N/A        Image:      N/A
Wait Start TickCount 1163714   Ticks: 0
Context Switch Count 545
UserTime           00:00:00.000
KernelTime         00:00:00.015
Start Address 0xb9003c20
Stack Init b9148000 Current b9147abc Base b9148000 Limit b9145000 Call 0
Priority 16 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr
b9147ad4 80833ec5 nt!KiSwapContext+0x26
b9147b00 80829069 nt!KiSwapThread+0x2e5
b9147b48 b8fc9353 nt!KeDelayExecutionThread+0x2ab
b9147b74 b8ff9460 ComponentB!DeleteShareConnection+0x203
[...]
b9147ddc 8088f61e nt!PspSystemThreadStartup+0x2e
00000000 00000000 nt!KiThreadStartup+0x16
```

Looking at the list of all threads we see other classes of **Blocked Threads** (page 80), one that is involving *ComponentC* module in user space:

```
0: kd> !thread 86c21db0 1f
THREAD 86c21db0 Cid 0fac.5260 Teb: 7ff6a000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Non-
Alertable
 869f2f68 SynchronizationEvent
IRP List:
 87fe3148: (0006,0220) Flags: 00000830 Mdl: 00000000
Not impersonating
DeviceMap          e1006e10
Owning Process    896ccc28   Image:      ServiceA.exe
Attached Process   N/A        Image:      N/A
Wait Start TickCount 1163714   Ticks: 0
Context Switch Count 22
UserTime           00:00:00.000
KernelTime         00:00:00.000
Win32 Start Address 0x005c1de0
LPC Server thread working on message Id 5c1de0
Start Address kernel32!BaseThreadStartThunk (0x7c8217ec)
Stack Init b9787000 Current b9786c60 Base b9787000 Limit b9784000 Call 0
Priority 9 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr
b9786c78 80833ec5 nt!KiSwapContext+0x26
b9786ca4 80829bc0 nt!KiSwapThread+0x2e5
b9786cec 8093b034 nt!KeWaitForSingleObject+0x346
b9786d50 8088ad3c nt!NtWaitForSingleObject+0x9a
b9786d50 7c9485ec nt!KiFastCallEntry+0xfc
03f8f984 7c821c8d ntdll!KiFastSystemCallRet
03f8f998 10097728 kernel32!WaitForSingleObject+0x12
```

```
03f8f9bc 10008164 ComponentC!ComponentB_Control+0x68
[...]
03f8ffec 00000000 kernel32!BaseThreadStart+0x34
```

This thread holds a mutant and blocks a dozen of other threads in *ServiceA.exe*, for example:

```
THREAD 8aa7cb40 Cid 0fac.0110 Teb: 7ffad000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Non-
Alertable
    87764550 Mutant - owning thread 86c21db0
```

From the function name, we can infer that *ComponentC* controls *ComponentB*, and this makes both blocked threads weakly connected.

Another thread in *ServiceB* involves *DriverA* module and blocks a thread *ServiceA*:

```
0: kd> !thread 8899e778 1f
THREAD 8899e778 Cid 01b0.13e0 Teb: 7ffdc000 Win32Thread: 00000000 WAIT: (Unknown) KernelMode Non-
Alertable
    8aadb6e0 SynchronizationEvent
    8899e7f0 NotificationTimer
IRP List:
    86f21de0: (0006,0220) Flags: 00000884 Mdl: 00000000
Not impersonating
DeviceMap          e1006e10
Owning Process     8ab3d020      Image:       ServiceB.exe
Attached Process   N/A          Image:       N/A
Wait Start TickCount 1163714    Ticks: 0
Context Switch Count 2
UserTime           00:00:00.000
KernelTime         00:00:00.000
Win32 Start Address 0x005c1a3c
LPC Server thread working on message Id 5c1a3c
Start Address 0x48589bb3
Stack Init aecee000 Current aeeced768 Base aecee000 Limit aeceb000 Call 0
Priority 12 BasePriority 11 PriorityDecrement 0
ChildEBP RetAddr
aeeced780 80833ec5 nt!KiSwapContext+0x26
aeeced7ac 80829bc0 nt!KiSwapThread+0x2e5
aeeced7f4 badffece nt!KeWaitForSingleObject+0x346
WARNING: Stack unwind information not available. Following frames may be wrong.
aeeced824 bae00208 DriverA+0x1ece
aeeced868 bae0e45a DriverA+0x2208
aeeced8a0 8081e095 DriverA+0x1045a
aeeced8b4 b946673b nt!IoCallDriver+0x45
[...]
```

```
0: kd> !thread 8776c220 1f
THREAD 8776c220 Cid 0fac.5714 Peb: 7ff66000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Non-
Alertable
    8776c40c Semaphore Limit 0x1
Waiting for reply to LPC MessageId 005c1a3c:
Current LPC port e213b0c8
Not impersonating
DeviceMap          e1006e10
Owning Process     896ccc28      Image:       ServiceA.exe
Attached Process   N/A          Image:       N/A
Wait Start TickCount 1163714    Ticks: 0
Context Switch Count 12
UserTime           00:00:00.000
KernelTime         00:00:00.000
Win32 Start Address 0x75fddd73
Start Address kernel32!BaseThreadStartThunk (0x7c8217ec)
Stack Init aecf2000 Current aecf1c08 Base aecf2000 Limit aecef000 Call 0
Priority 8 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr
aecf1c20 80833ec5 nt!KiSwapContext+0x26
aecf1c4c 80829bc0 nt!KiSwapThread+0x2e5
aecf1c94 80920f28 nt!KeWaitForSingleObject+0x346
aecf1d50 8088ad3c nt!NtRequestWaitReplyPort+0x776
aecf1d50 7c9485ec nt!KiFastCallEntry+0xfc
0408f594 75fdde6b5 nt!NtRequestWaitReplyPort+0x776
0408f5fc 75fdd65e ComponentD!ServiceB Request+0x1ae
[...]
0408ffec 00000000 kernel32!BaseThreadStart+0x34
```

In *ServiceA* we can also find several threads blocked by an RPC request to *ServiceC*:

```
0: kd> !thread 87397020 1f
THREAD 87397020 Cid 0fac.38cc Peb: 7ff80000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Non-
Alertable
    87397098 NotificationTimer
Not impersonating
DeviceMap          e1006e10
Owning Process     896ccc28      Image:       ServiceA.exe
Attached Process   N/A          Image:       N/A
Wait Start TickCount 1163714    Ticks: 0
Context Switch Count 7807
UserTime           00:00:00.125
KernelTime         00:00:00.109
Win32 Start Address 0x005c21a8
LPC Server thread working on message Id 5c21a8
Start Address kernel32!BaseThreadStartThunk (0x7c8217ec)
Stack Init b4ecf000 Current b4ecec80 Base b4ecf000 Limit b4ecc000 Call 0
Priority 13 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr
b4ecec98 80833ec5 nt!KiSwapContext+0x26
b4ecec4 80829069 nt!KiSwapThread+0x2e5
b4eced0c 80996d8a nt!KeDelayExecutionThread+0x2ab
b4eced54 8088ad3c nt!NtDelayExecution+0x84
b4eced54 7c9485ec nt!KiFastCallEntry+0xfc
03a1f178 7c8024ed nt!NtDelayExecution+0x84
03a1f188 77c5e51a kernel32!Sleep+0xf
```

```

03a1f198 77c36a44 RPCRT4!OSF_BINDING_HANDLE::Unbind+0x3a
03a1f1b0 77c36a08 RPCRT4!OSF_BINDING_HANDLE::~OSF_BINDING_HANDLE+0x32
03a1f1bc 77c369f1 RPCRT4!OSF_BINDING_HANDLE::`scalar deleting destructor'+0xd
03a1f1cc 77c5250a RPCRT4!OSF_BINDING_HANDLE::BindingFree+0x30
03a1f1dc 77f48c00 RPCRT4!RpcBindingFree+0x4e
03a1f1e8 77f48be2 ADVAPI32!RcpUnbindRpc+0x15
03a1f1f4 77c3688e ADVAPI32!PLSAPR_SERVER_NAME_unbind+0xd
03a1f21c 77c369bb RPCRT4!GenericHandleMgr+0xca
03a1f23c 77c36983 RPCRT4!GenericHandleUnbind+0x31
03a1f260 77cb31b2 RPCRT4!NdrpClientFinally+0x5b
03a1f26c 77cb317a RPCRT4!NdrClientCall12+0x324
03a1f64c 77f4a0a1 RPCRT4!NdrClientCall12+0x2ea
03a1f664 77f4a022 ComponentD!ServiceC_Request+0x1c
[...]
03a1f8f8 77cb33e1 RPCRT4!Invoke+0x30
03a1fcf8 77cb35c4 RPCRT4!NdrStubCall12+0x299
03a1fd14 77c4ff7a RPCRT4!NdrServerCall12+0x19
03a1fd48 77c5042d RPCRT4!DispatchToStubInCNoAvrf+0x38
03a1fd9c 77c50353 RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x11f
03a1fdc0 77c511dc RPCRT4!RPC_INTERFACE::DispatchToStub+0xa3
03a1fdfe 77c512f0 RPCRT4!LRPC_SCALL::DealWithRequestMessage+0x42c
03a1fe20 77c58678 RPCRT4!LRPC_ADDRESS::DealWithLRPCRequest+0x127
03a1ff84 77c58792 RPCRT4!LRPC_ADDRESS::ReceiveLotsaCalls+0x430
03a1ff8c 77c5872d RPCRT4!RecvLotsaCallsWrapper+0xd
03a1ffac 77c4b110 RPCRT4!BaseCachedThreadRoutine+0x9d
03a1ffb8 7c824829 RPCRT4!ThreadStartRoutine+0xb
03a1ffec 00000000 kernel32!BaseThreadStart+0x34

```

In *ServiceC*, we see several RPC processing threads blocked by *ComponentE*:

```

0: kd> !thread 873acb40 1f
THREAD 873acb40 Cid 023c.3a00 Teb: 7fff93000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Non-Alertable
    89f0aeb0 Semaphore Limit 0x1
    873acbb8 NotificationTimer
IRP List:
    89838a00: (0006,0094) Flags: 00000900 Mdl: 00000000
    8705d4a0: (0006,0094) Flags: 00000800 Mdl: 00000000
    88bc9440: (0006,0094) Flags: 00000900 Mdl: 00000000
    87674af8: (0006,0094) Flags: 00000900 Mdl: 00000000
    86f2aa48: (0006,0094) Flags: 00000900 Mdl: 00000000
    87551290: (0006,0094) Flags: 00000900 Mdl: 00000000
Not impersonating
DeviceMap          e1006e10
Owning Process     89dc0508      Image:       ServiceC.exe
Attached Process   N/A          Image:       N/A
Wait Start TickCount 1163714      Ticks: 0
Context Switch Count 16571
UserTime            00:00:00.250
KernelTime          00:00:00.703
Win32 Start Address RPCRT4!ThreadStartRoutine (0x77c4b0f5)
Start Address kernel32!BaseThreadStartThunk (0x7c8217ec)
Stack Init b2a9b000 Current b2a9ac60 Base b2a9b000 Limit b2a98000 Call 0
Priority 13 BasePriority 9 PriorityDecrement 0
ChildEBP RetAddr
b2a9ac78 80833ec5 nt!KiSwapContext+0x26

```

```

b2a9aca4 80829bc0 nt!KiSwapThread+0x2e5
b2a9acec 8093b034 nt!KeWaitForSingleObject+0x346
b2a9ad50 8088ad3c nt!NtWaitForSingleObject+0x9a
b2a9ad50 7c9485ec nt!KiFastCallEntry+0xfc
022cf8d0 7c821c8d nt!nl!KiFastSystemCallRet
022cf8e4 741269e5 kernel32!WaitForSingleObject+0x12
022cf8f8 7412cdca ComponentE!Enumerate+0x37
[...]
022cf944 77cb33e1 RPCRT4!Invoke+0x30
022cf944 77cb35c4 RPCRT4!NdrStubCall2+0x299
022cf944 77c4ff7a RPCRT4!NdrServerCall2+0x19
022cf944 77c5042d RPCRT4!DispatchToStubInCNoAvrf+0x38
022cf944 77c50353 RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x11f
022cfe0c 77c38e0d RPCRT4!RPC_INTERFACE::DispatchToStub+0xa3
022cfe40 77c38cb3 RPCRT4!OSF_SCALL::DispatchHelper+0x149
022cfe54 77c38c2b RPCRT4!OSF_SCALL::DispatchRPCCall+0x10d
022cfe84 77c38b5e RPCRT4!OSF_SCALL::ProcessReceivedPDU+0x57f
022cfea4 77c3e8db RPCRT4!OSF_SCALL::BeginRpcCall+0x194
022cff04 77c3e7b4 RPCRT4!OSF_SCONNECTION::ProcessReceiveComplete+0x435
022cff18 77c4b799 RPCRT4!ProcessConnectionServerReceivedEvent+0x21
022cff84 77c4b9b5 RPCRT4!LOADABLE_TRANSPORT::ProcessIOEvents+0x1b8
022cff8c 77c5872d RPCRT4!ProcessIOEventsWrapper+0xd
022cffac 77c4b110 RPCRT4!BaseCachedThreadRoutine+0x9d
022cffb8 7c824829 RPCRT4!ThreadStartRoutine+0x1b
022cffec 00000000 kernel32!BaseThreadStart+0x34

```

Therefore, we have 4 main groups of **Wait Chain** (page 1092) endpoints involving semantically disjoint *ComponentA*, *ComponentB*, *ComponentE* and *DriverA* modules. Although their module names do not infer disjointedness, this example was abstracted from the real incident where respective modules were having different system functions and were even from different software vendors.

## Semantic Structure

### PID.TID

This part starts the block of patterns called **Semantic Structures**. These structures are fragments of memory which have meaning helping us in troubleshooting and debugging. The first pattern in this block deals with PID.TID structures of the form DWORD : DWORD or QWORD : QWORD. Such memory fragments are useful for wait chain analysis (page 1188), for example, by looking at **Execution Residue** (page 371) left on a raw stack to find a target or an origin of RPC or (A)LPC calls. RPC target example can be found in the article: In Search of Lost CID<sup>186</sup>. Here we look at another example, this time to find the originator of an ALPC call.

*ServiceA* process was executing some undesired functionality, and a breakpoint was set on *ModuleA* code to trigger it under irreproducible conditions. Then a complete memory dump was saved for offline analysis. There we see an ALPC server thread that triggered the breakpoint, but we don't see the message information in the output of WinDbg **!thread** command that can help us finding a corresponding ALPC client thread easily:

```
THREAD ffffffa8005e6b060 Cid 0cc0.1838 Teb: 000007fffff8e000 Win32Thread: 0000000000000000 WAIT:
(Executive) KernelMode Non-Alertable
SuspendCount 1
fffff880094ad0a0 SynchronizationEvent
Not impersonating
DeviceMap      fffff8a001aba3c0
Owning Process  ffffffa8004803b30           Image: ServiceA.exe
Attached Process N/A          Image: N/A
Wait Start TickCount 1441562    Ticks: 106618 (0:00:27:43.251)
Context Switch Count 414
UserTime        00:00:00.000
KernelTime      00:00:00.031
Win32 Start Address ntdll!TppWorkerThread (0x0000000077c88f00)
Stack Init fffff880094addb0 Current fffff880094acdb0
Base fffff880094ae000 Limit fffff880094a8000 Call 0
Priority 12 BasePriority 10 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
Child-SP RetAddr Call Site
fffff880`094acdf0 fffff800`01678992 nt!KiSwapContext+0x7a
fffff880`094acf30 fffff800`0167acff nt!KiCommitThreadWait+0x1d2
fffff880`094acf0 fffff800`01a150e8 nt!KeWaitForSingleObject+0x19f
fffff880`094ad060 fffff800`01a1546c nt!DbgkpQueueMessage+0x2a8
fffff880`094ad230 fffff800`019b9116 nt!DbgkpSendApiMessage+0x5c
fffff880`094ad270 fffff800`016abb96 nt! ?? ::NNGAK EGL::`string'+0x3463d
fffff880`094ad3b0 fffff800`01670d82 nt!KiDispatchException+0x316
fffff880`094ada40 fffff800`0166ebb4 nt!KiExceptionDispatch+0xc2
fffff880`094adc20 000007fe`f79365d1 nt!KiBreakpointTrap+0xf4 (TrapFrame @ fffff880`094adc20)
00000000`035ee568 000007fe`f80670b5 ModuleA+0x38611
[...]
00000000`035ee5d0 000007fe`ff4bc7f5 ModuleB!Start+0x6e1
00000000`035ee770 000007fe`ff56b62e RPCRT4!Invoke+0x65
```

<sup>186</sup> In Search of Lost CID, Memory Dump Analysis Anthology, Volume 2, page 136

```

00000000`035ee7c0 000007fe`ff4bf1f6 RPCRT4!Ndr64StubWorker+0x61b
00000000`035eed80 000007fe`fffdf223 RPCRT4!NdrStubCall3+0xb5
00000000`035eede0 000007fe`ffedfc0d ole32!CStdStubBuffer_Invoke+0x5b
00000000`035eee10 000007fe`ffedfb83 ole32!SyncStubInvoke+0x5d
00000000`035eee80 000007fe`ffd7fd60 ole32!StubInvoke+0xdb
00000000`035ef30 000007fe`ffedfa22 ole32!CCtxComChnl::ContextInvoke+0x190
00000000`035ef0c0 000007fe`ffedf76b ole32!AppInvoke+0xc2
00000000`035ef130 000007fe`ffeeded6d ole32!ComInvokeWithLockAndIPID+0x52b
00000000`035ef2c0 000007fe`ff4b9c24 ole32!ThreadInvoke+0x30d
00000000`035ef360 000007fe`ff4b9d86 RPCRT4!DispatchToStubInCNoAvrf+0x14
00000000`035ef390 000007fe`ff4bc44b RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x146
00000000`035ef4b0 000007fe`ff4bc38b RPCRT4!RPC_INTERFACE::DispatchToStub+0x9b
00000000`035ef4f0 000007fe`ff4bc322 RPCRT4!RPC_INTERFACE::DispatchToStubWithObject+0x5b
00000000`035ef570 000007fe`ff4ba11d RPCRT4!LRPC_SCALL::DispatchRequest+0x422
00000000`035ef650 000007fe`ff4c7ddf RPCRT4!LRPC_SCALL::HandleRequest+0x20d
00000000`035ef780 000007fe`ff4c7995 RPCRT4!LRPC_ADDRESS::ProcessIO+0x3bf
00000000`035ef8c0 00000000`77c8b43b RPCRT4!LrpclIoComplete+0xa5
00000000`035ef950 00000000`77c8923f ntdll!TppAlpcpExecuteCallback+0x26b
00000000`035ef9e0 00000000`77a6f56d ntdll!TppWorkerThread+0x3f8
00000000`035efce0 00000000`77ca3281 kernel32!BaseThreadInitThunk+0xd
00000000`035efd10 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

```

We inspect the raw stack starting from the first top Child-SP value for *RPCRT4* subtrace and find NNN:NNN data there resembling a PID:TID pair:

```

1: kd> dpp 00000000`035ef360 1100
[...]
00000000`035ef698 00000000`00000000
00000000`035ef6a0 00000000`00000001
00000000`035ef6a8 00000000`00000000
00000000`035ef6b0 00000000`00000000
00000000`035ef6b8 00000000`00000118
00000000`035ef6c0 00000000`0000048c
00000000`035ef6c8 00000000`00495e50 000007fe`ff57d920 RPCRT4!LRPC_ADDRESS::`vftable'
00000000`035ef6d0 00000000`00000000
[...]

```

We find such CID in **Stack Trace Collection** (page 943) and see a wait for an ALPC message reply:

```

THREAD ffffffa8003d49b60 Cid 0118.048c Teb: 000007fffffaa000 Win32Thread: fffff900c01e4c30 WAIT:
(WrLpcReply) UserMode Non-Alertable
fffffa8003d49f20 Semaphore Limit 0x1
Waiting for reply to ALPC Message fffff8a000bdb6c0 : queued at port fffff8a80042f8090 : owned by process
fffffa8004803b30
Not impersonating
DeviceMap      fffff8a000008600
Owning Process  ffffffa8003cf15d0      Image: ServiceB.exe
Attached Process N/A      Image: N/A
Wait Start TickCount 1441554      Ticks: 106626 (0:00:27:43.376)
Context Switch Count 23180 LargeStack
UserTime        00:00:00.468
KernelTime      00:00:03.057
Win32 Start Address ntdll!TppWorkerThread (0x0000000077c88f00)
Stack Init fffff88004ffcdb0 Current fffff88004ffc620

```

```

Base fffff88004ffd000 Limit fffff88004ff7000 Call 0
Priority 6 BasePriority 6 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
Kernel stack not resident.
Child-SP RetAddr Call Site
fffff880`04ffc660 fffff800`01678992 nt!KiSwapContext+0x7a
fffff880`04ffc7a0 fffff800`0167acff nt!KiCommitThreadWait+0x1d2
fffff880`04ffc830 fffff800`0168fd1f nt!KeWaitForSingleObject+0x19f
fffff880`04ffc8d0 fffff800`01977ac6 nt!AlpcpSignalAndWait+0x8f
fffff880`04ffc980 fffff800`01975a50 nt!AlpcpReceiveSynchronousReply+0x46
fffff880`04ffc9e0 fffff800`01972fc8 nt!AlpcpProcessSynchronousRequest+0x33d
fffff880`04ffcb00 fffff800`01670993 nt!NtAlpcSendWaitReceivePort+0x1ab
fffff880`04ffccbb0 00000000`77cc070a nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ fffff880`04ffcc20)
00000000`018ce308 000007fe`ff4caa76 ntdll!ZwAlpcSendWaitReceivePort+0xa
00000000`018ce310 000007fe`ff4bf802 RPCRT4!LRPC_CCALL::SendReceive+0x156
00000000`018ce3d0 000007fe`ffee0900 RPCRT4!I_RpcSendReceive+0x42
00000000`018ce400 000007fe`ffee05ef ole32!ThreadSendReceive+0x40
00000000`018ce450 000007fe`ffee041b ole32!CRpcChannelBuffer::SwitchAptAndDispatchCall+0xa3
00000000`018ce4f0 000007fe`ffd819c6 ole32!CRpcChannelBuffer::SendReceive2+0x11b
00000000`018ce6b0 000007fe`ffd81928 ole32!CAptRpcChnl::SendReceive+0x52
00000000`018ce780 000007fe`ffedfcf5 ole32!CCtxComChnl::SendReceive+0x68
00000000`018ce830 000007fe`ff56ba3b ole32!NdrExtpProxySendReceive+0x45
00000000`018ce860 000007fe`ffee02d0 RPCRT4!NdrpClientCall3+0x2e2
00000000`018ceb20 000007fe`ffd818a2 ole32!ObjectStublessClient+0x11d
00000000`018ceeb0 00000000`ff5afe64 ole32!ObjectStubless+0x42
[...]
00000000`018cf7a0 00000000`77c8f8eb ServiceB!Worker+0x366
00000000`018cf800 00000000`77c89d9f ntdll!RtlpTpWorkCallback+0x16b
00000000`018cf8e0 00000000`77a6f56d ntdll!TppWorkerThread+0x5ff
00000000`018cfbe0 00000000`77ca3281 kernel32!BaseThreadInitThunk+0xd
00000000`018cffc10 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

```

Inspection of that message shows that it was directed to our server thread that triggered the breakpoint:

```

1: kd> !alpc /m fffff8a000bdb6c0

Message @ fffff8a000bdb6c0
MessageID      : 0x0600 (1536)
CallbackID     : 0x2D910D (2986253)
SequenceNumber : 0x0002CB50 (183120)
Type          : LPC_REQUEST
DataLength     : 0x0068 (104)
TotalLength    : 0x0090 (144)
Canceled       : No
Release        : No
ReplyWaitReply : No
Continuation   : Yes
OwnerPort      : ffffffa8004823a80 [ALPC_CLIENT_COMMUNICATION_PORT]
WaitingThread  : ffffffa8003d49b60
QueueType      : ALPC_MSGQUEUE_PENDING
QueuePort      : ffffffa80042f8090 [ALPC_CONNECTION_PORT]
QueuePortOwnerProcess : ffffffa8004803b30 (ServiceA.exe)
ServerThread   : ffffffa8005e6b060
QuotaCharged   : No
CancelQueuePort : 0000000000000000
CancelSequencePort : 0000000000000000

```

```
CancelSequenceNumber : 0x00000000 (0)
ClientContext      : 00000000020f0c0
ServerContext      : 0000000000000000
PortContext        : 0000000000416990
CancelPortContext  : 0000000000000000
SecurityData       : 0000000000000000
View               : 0000000000000000
```

## Comments

Another example is two blocked COM threads where the second one has PID.TID packed in a qword on its raw stack that clearly shows a dependency.

## Shared Buffer Overwrite

### Mac OS X

This is a Mac OS X example of **Shared Buffer Overwrite** pattern. Originally we wanted to construct a default C runtime heap corruption example using *malloc / free* functions. Unfortunately, we couldn't get a heap corrupted as easily as was possible in Windows Visual C++ environment by writing before or after allocated block. Desperately we printed allocated pointers and they all pointed to memory blocks laid out one after another without any headers in between (could be just a default Apple LLVM C runtime implementation, and we have to check that with GCC). Therefore, any subsequent reallocation didn't cause corruption either. So all this naturally fits into shared buffer overwrites or underwrites where corruption is only detectable when the overwritten data is used such as a pointer dereference.

```
int main(int argc, const char * argv[])
{
    char *p1 = (char *) malloc (1024);
    strcpy(p1, "Hello World!");
    printf("p1 = %p\n", p1);
    printf("*p1 = %s\n", p1);

    char *p2 = (char *) malloc (1024);
    strcpy(p2, "Hello World!");
    printf("p2 = %p\n", p2);
    printf("*p2 = %s\n", p2);

    char *p3 = (char *) malloc (1024);
    strcpy(p3, "Hello World!");
    printf("p3 = %p\n", p3);
    printf("*p3 = %s\n", p3);

    strcpy(p2-sizeof(p2), "Hello Crash!");
    strcpy(p3-sizeof(p3), "Hello Crash!");
    p2 = (char *)realloc(p2, 2048);
    printf("p2 = %p\n", p2);
    printf("*p2 = %s\n", p2);

    char *p4 = (char *) malloc (1024);
    strcpy(p4-sizeof(p4), "Hello Crash!");
    printf("p4 = %p\n", p4);
    printf("*p4 = %s\n", p4);

    p3 = (char *)realloc(p3, 2048);
    printf("p3 = %p\n", p3);
    printf("*p3 = %s\n", p3);

    char *p5 = NULL; // to force a core dump
    *p5 = 0;
    free (p4);
    free (p3);
    free (p2);
    free (p1);
```

```
    return 0;
}
```

When we run the program above we get this output:

```
p1 = 0x7fc6d9000000
*p1 = Hello World!
p2 = 0x7fc6d9001400
*p2 = Hello World!
p3 = 0x7fc6d9001800
*p3 = Hello World!
p2 = 0x7fc6d9001c00
*p2 = ash!
p4 = 0x7fc6d9001400
*p4 = ash!
p3 = 0x7fc6d9002400
*p3 = ash!
Segmentation fault: 11 (core dumped)
```

Now is GDB output:

```
(gdb) x/1024bc p1
0x7fc6d9000000: 72 'H' 101 'e' 108 'l' 108 'l' 111 'o' 32 ' ' 87 'W' 111 'o'
0x7fc6d9000008: 114 'r' 108 'l' 100 'd' 33 '!' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9000010: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
[...]
0x7fc6d90003e8: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d90003f0: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d90003f8: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'

(gdb) x/32bc p1+1024-sizeof(p1)
0x7fc6d90003f8: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9000400: 42 '*' 112 'p' 51 '3' 32 ' ' 61 '=' 32 ' ' 97 'a' 115 's'
0x7fc6d9000408: 104 'h' 33 '!' 10 'n' 100 'd' 57 '9' 48 '0' 48 '0' 50 '2'
0x7fc6d9000410: 52 '4' 48 '0' 48 '0' 10 '\n' 0 '\0' 0 '\0' 0 '\0' 0 '\0'

(gdb) x/2048bc p2
0x7fc6d9001c00: 97 'a' 115 's' 104 'h' 33 '!' 0 '\0' 32 ' ' 87 'W' 111 'o'
0x7fc6d9001c08: 114 'r' 108 'l' 100 'd' 33 '!' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9001c10: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
[...]
0x7fc6d9001fe8: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9001ff0: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9001ff8: 72 'H' 101 'e' 108 'l' 108 'l' 111 'o' 32 ' ' 67 'C' 114 'r'
0x7fc6d9002000: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002008: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
[...]
0x7fc6d90023e8: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d90023f0: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d90023f8: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
```

```
(gdb) x/64bc p2-sizeof(p2)
0x7fc6d9001bf8: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9001c00: 97 'a' 115 's' 104 'h' 33 '!' 0 '\0' 32 ' ' 87 'W' 111 'o'
0x7fc6d9001c08: 114 'r' 108 'l' 100 'd' 33 '!' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9001c10: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9001c18: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9001c20: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9001c28: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9001c30: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'

(gdb) x/64bc p2+2048-sizeof(p2)
0x7fc6d90023f8: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002400: 97 'a' 115 's' 104 'h' 33 '!' 0 '\0' 32 ' ' 87 'W' 111 'o'
0x7fc6d9002408: 114 'r' 108 'l' 100 'd' 33 '!' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002410: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002418: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002420: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002428: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002430: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'

(gdb) x/1024bc p3
0x7fc6d9002400: 97 'a' 115 's' 104 'h' 33 '!' 0 '\0' 32 ' ' 87 'W' 111 'o'
0x7fc6d9002408: 114 'r' 108 'l' 100 'd' 33 '!' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002410: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
[...]
0x7fc6d90027e8: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d90027f0: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d90027f8: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'

(gdb) x/64bc p3-sizeof(p3)
0x7fc6d90023f8: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002400: 97 'a' 115 's' 104 'h' 33 '!' 0 '\0' 32 ' ' 87 'W' 111 'o'
0x7fc6d9002408: 114 'r' 108 'l' 100 'd' 33 '!' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002410: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002418: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002420: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002428: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002430: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'

(gdb) x/64bc p3+1024-sizeof(p3)
0x7fc6d90027f8: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002800: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002808: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002810: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002818: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002820: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002828: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9002830: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
```

```
(gdb) x/1024bc p4
0x7fc6d9001400: 97 'a' 115 's' 104 'h' 33 '!' 0 '\0' 32 ' ' 87 'W' 111 'o'
0x7fc6d9001408: 114 'r' 108 '1' 100 'd' 33 '!' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9001410: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
[...]
0x7fc6d90017e8: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d90017f0: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d90017f8: 72 'H' 101 'e' 108 'l' 108 '1' 111 'o' 32 ' ' 67 'C' 114 'r'

(gdb) x/64bc p4-sizeof(p4)
0x7fc6d90013f8: 72 'H' 101 'e' 108 'L' 108 'L' 111 'o' 32 ' ' 67 'C' 114 'r'
0x7fc6d9001400: 97 'a' 115 's' 104 'h' 33 '!' 0 '\0' 32 ' ' 87 'W' 111 'o'
0x7fc6d9001408: 114 'r' 108 '1' 100 'd' 33 '!' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9001410: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9001418: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9001420: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9001428: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9001430: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'

(gdb) x/64bc p4+1024-sizeof(p4)
0x7fc6d90017f8: 72 'H' 101 'e' 108 'L' 108 'L' 111 'o' 32 ' ' 67 'C' 114 'r'
0x7fc6d9001800: 97 'a' 115 's' 104 'h' 33 '!' 0 '\0' 32 ' ' 87 'W' 111 'o'
0x7fc6d9001808: 114 'r' 108 '1' 100 'd' 33 '!' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9001810: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9001818: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9001820: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9001828: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fc6d9001830: 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
```

## Windows

This pattern differs from **Local Buffer Overflow** (page 611) and **Dynamic Memory Corruption** (heap, page 307, and pool, page 292) patterns in not writing over control structures situated at dynamically allocated memory or procedure frame (local call stack) boundaries. Its effect is visible when the buffer data contains pointers that become **Wild Pointers** (page 1151) after overwrite and are later dereferenced resulting in a crash. For example, when the overwriting data contains UNICODE and /or ASCII characters we see them in a pointer data:

```
1: kd> !analyze -v

[...]

SYSTEM_THREAD_EXCEPTION_NOT_HANDLED (7e)
This is a very common bugcheck. Usually the exception address pinpoints
the driver/function that caused the problem. Always note this address
as well as the link date of the driver/image that contains this address.
Arguments:
Arg1: c0000005, The exception code that was not handled
Arg2: 8086c949, The address that the exception occurred at
Arg3: f78eec54, Exception Record Address
Arg4: f78ee950, Context Record Address

[...]

EXCEPTION_RECORD: f78eec54 -- (.exr 0xffffffff78eec54)
ExceptionAddress: 8086c949 (nt!ObfDereferenceObject+0x00000023)
    ExceptionCode: c0000005 (Access violation)
    ExceptionFlags: 00000000
NumberParameters: 2
    Parameter[0]: 00000001
    Parameter[1]: 006f0058
Attempt to write to address 006f0058

CONTEXT: f78ee950 -- (.cxr 0xffffffff78ee950)
eax=f78e0001 ebx=fffffff ecx=006f0070 edx=00000000 esi=006f0058 edi=8087cd8
eip=8086c949 esp=f78eed1c ebp=f78eed2c iopl=0 nv up ei ng nz na pe nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010286
nt!ObfDereferenceObject+0x23:
8086c949 f00fc11e lock xadd dword ptr [esi],ebx ds:0023:006f0058=????????
[...]

STACK_TEXT:
f78eed2c f707212e 886e6530 f78eed80 f706e04e nt!ObfDereferenceObject+0x23
f78eed38 f706e04e e47b1258 8b2fc40 808ae5c0 DriverA!CloseConnection+0x16
f78eed80 80880475 8835f248 00000000 8b2fc40 DriverA!Resume+0x9f
f78eedac 80949c5a 8835f248 00000000 00000000 nt!ExpWorkerThread+0xeb
f78eeddc 8088e0c2 8088038a 00000000 00000000 nt!PspSystemThreadStartup+0x2e
00000000 00000000 00000000 00000000 nt!KiThreadStartup+0x16
```

```

1: kd> ub f707212e
DriverA!CloseConnection+0x2:
f707211a push    ebp
f707211b mov     ebp,esp
f707211d push    esi
f707211e mov     esi,dword ptr [ebp+8]
f7072121 mov     ecx,dword ptr [esi+14h]
f7072124 test    ecx,ecx
f7072126 je      DriverA!CloseConnection+0x1a (f7072132)
f7072128 call    dword ptr [DriverA!_imp_0bfDereferenceObject (f70610f4)]


1: kd> db e47b1258 L20
e47b1258 61 67 65 20 57 72 69 74-65 72 00 05 77 00 69 00  age Writer..w.i.
e47b1268 6e 00 73 00 70 00 6f 00-6f 00 6c 00 2c 00 4e 00  n.s.p.o.o.l.,.N.

1: kd> !pool e47b1258
Pool page e47b1258 region is Paged pool
e47b1000 size: 108 previous size: 0 (Allocated) CM39
e47b1108 size: 38 previous size: 108 (Free) CMVa
e47b1140 size: 28 previous size: 38 (Allocated) NtFs
e47b1168 size: 8 previous size: 28 (Free) CMDa
e47b1170 size: 80 previous size: 8 (Allocated) FSim
e47b11f0 size: 28 previous size: 80 (Allocated) CMNb (Protected)
*e47b1218 size: 70 previous size: 28 (Allocated) *CMDa
  Pooltag CMDa : value data cache pool tag, Binary : nt!cm
e47b1288 size: 58 previous size: 70 (Allocated) Sect (Protected)
e47b12e0 size: 18 previous size: 58 (Allocated) Ntf0
e47b12f8 size: 28 previous size: 18 (Allocated) NtFs
e47b1320 size: 20 previous size: 28 (Allocated) CMNb (Protected)
e47b1340 size: 48 previous size: 20 (Allocated) Ntfc
e47b1388 size: 68 previous size: 48 (Allocated) Sect (Protected)
e47b13f0 size: 30 previous size: 68 (Allocated) CMVa
e47b1420 size: 38 previous size: 30 (Allocated) CMVa
e47b1458 size: 8 previous size: 38 (Free) CMVa
e47b1460 size: 48 previous size: 8 (Allocated) CMVa
e47b14a8 size: d0 previous size: 48 (Allocated) Ntfo
e47b1578 size: 330 previous size: d0 (Allocated) Ntff
e47b18a8 size: 10 previous size: 330 (Free) Ntfe
e47b18b8 size: e0 previous size: 10 (Allocated) Ntfo
e47b1998 size: 40 previous size: e0 (Allocated) MmSm
e47b19d8 size: 8 previous size: 40 (Free) Ica
e47b19e0 size: 18 previous size: 8 (Allocated) Ntf0
e47b19f8 size: 68 previous size: 18 (Allocated) CMDa
e47b1a60 size: 28 previous size: 68 (Allocated) ObNm
e47b1a88 size: b8 previous size: 28 (Allocated) Port (Protected)
e47b1b40 size: 58 previous size: b8 (Allocated) Sect (Protected)
e47b1b98 size: 30 previous size: 58 (Allocated) CMVa
e47b1bc8 size: 8 previous size: 30 (Free) NtFA
e47b1bd0 size: 100 previous size: 8 (Allocated) IoNm
e47b1cd0 size: 18 previous size: 100 (Allocated) ObDi
e47b1ce8 size: 38 previous size: 18 (Allocated) CMNb Process: 88469928
e47b1d20 size: 78 previous size: 38 (Free ) NtFI
e47b1d98 size: 68 previous size: 78 (Allocated) CMDa
e47b1e00 size: 18 previous size: 68 (Allocated) PsIm (Protected)
e47b1e18 size: e8 previous size: 18 (Free ) TunP
e47b1f00 size: 100 previous size: e8 (Allocated) IoNm

```

Another example:

```
0: kd> !analyze -v

[...]

SYSTEM_THREAD_EXCEPTION_NOT_HANDLED (7e)
This is a very common bugcheck. Usually the exception address pinpoints
the driver/function that caused the problem. Always note this address
as well as the link date of the driver/image that contains this address.
Arguments:
Arg1: c0000005, The exception code that was not handled
Arg2: 8083e4d6, The address that the exception occurred at
Arg3: f78cec54, Exception Record Address
Arg4: f78ce950, Context Record Address

[...]

EXCEPTION_RECORD: f78cec54 -- (.exr 0xfffffffff78cec54)
ExceptionAddress: 8083e4d6 (nt!ObfDereferenceObject+0x00000023)
    ExceptionCode: c0000005 (Access violation)
    ExceptionFlags: 00000000
NumberParameters: 2
    Parameter[0]: 00000001
    Parameter[1]: 65696c2b
Attempt to write to address 65696c2b

CONTEXT: f78ce950 -- (.cxr 0xfffffffff78ce950)
eax=f78c0001 ebx=fffffff ecx=65696c43 edx=00000000 esi=65696c2b edi=8083e407
eip=8083e4d6 esp=f78ced1c ebp=f78ced2c iopl=0 nv up ei ng nz na pe nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010286
nt!ObfDereferenceObject+0x23:
8083e4d6 f00fc11e lock xadd dword ptr [esi],ebx ds:0023:65696c2b=????????
Resetting default scope

[...]

STACK_TEXT:
f78ced2c f71bd12e 87216470 f78ced80 f71b904e nt!ObfDereferenceObject+0x23
f78ced38 f71b904e e49afb90 8a38eb40 808b70e0 DriverA!CloseConnection+0x16
f78ced80 8082db10 868989e0 00000000 8a38eb40 DriverA!Resume+0x9f
f78cedac 809208bb 868989e0 00000000 00000000 nt!ExpWorkerThread+0xeb
f78ceddc 8083fe9f 8082da53 00000000 00000000 nt!PspSystemThreadStartup+0x2e
00000000 00000000 00000000 00000000 nt!KiThreadStartup+0x16

[...]
```

```

0: kd> .formats 65696c2b
Evaluate expression:
Hex:      65696c2b
Decimal: 1701407787
Octal:   14532266053
Binary:  01100101 01101001 01101100 00101011
Chars:   eil+
Time:    Fri Dec 01 05:16:27 2023
Float:   low 6.88942e+022 high 0
Double:  8.40607e-315

0: kd> db e49afb90 L20
e49afb90  41 41 22 00 1e 00 00 00-00 5f 07 00 01 00 00 00  AA"....._.....
e49afba0  01 00 00 00 43 6c 69 65-6e 74 41 2f 41 41 41 41  ....ClientA/AAAA

0: kd> !pool e49afb90
Pool page e49afb90 region is Paged pool
e49af000 size: 330 previous size: 0 (Allocated) Ntff
e49af330 size: 2c0 previous size: 330 (Allocated) Toke (Protected)
e49af5f0 size: 78 previous size: 2c0 (Allocated) NtFU
e49af668 size: 10 previous size: 78 (Free) CMVI
e49af678 size: a8 previous size: 10 (Allocated) Ntfo
e49af720 size: 80 previous size: a8 (Allocated) NtFU
e49af7a0 size: 78 previous size: 80 (Allocated) NtFU
e49af818 size: 18 previous size: 78 (Allocated) Ntf0
e49af830 size: 20 previous size: 18 (Allocated) ObHd
e49af850 size: 38 previous size: 20 (Allocated) MmSm
e49af888 size: 78 previous size: 38 (Allocated) NtFU
e49af900 size: 28 previous size: 78 (Allocated) NtFs
e49af928 size: 48 previous size: 28 (Allocated) Ntfc
e49af970 size: 40 previous size: 48 (Allocated) CMNb (Protected)
e49af9b0 size: 28 previous size: 40 (Allocated) NtFs
e49af9d8 size: 30 previous size: 28 (Allocated) AtmA
e49afa08 size: 108 previous size: 30 (Allocated) CM39
e49afb10 size: 18 previous size: 108 (Allocated) Ntf0
e49afb28 size: 30 previous size: 18 (Allocated) CMVw (Protected)
e49afb58 size: 28 previous size: 30 (Allocated) MPXC
*e49afb80 size: 70 previous size: 28 (Free) *CMDa
 Pooltag CMDa : value data cache pool tag, Binary : nt!cm
e49afbf0 size: b8 previous size: 70 (Allocated) Port (Protected)
e49afca8 size: 28 previous size: b8 (Allocated) CMNb (Protected)
e49afcdd0 size: 330 previous size: 28 (Allocated) Ntff

```

Notice that in the latter example the pointer references a freed pool element. If a pointer points to an overwritten buffer, the result is similar to a dangling pointer<sup>187</sup> pointing to a reallocated freed buffer. If an object was located in a shared buffer and its data becomes overwritten, we can also observe **Random Object** pattern (page 825).

---

<sup>187</sup> [http://en.wikipedia.org/wiki/Dangling\\_pointer](http://en.wikipedia.org/wiki/Dangling_pointer)

## Shared Structure

Sometimes we look at **Stack Trace Collection** (page 943) or it's predicate subset (page 943) and we recognize that one of the parameters is actually the same structure address or handle. In x64 case we may possibly see it from the return address backward disassembly (**ub** WinDbg command) but in x86 case most of the time we can spot that directly from the verbose stack trace, like in the snippet below (unless a parameter memory slot was reused, **Optimized Code**, page 767):

```
THREAD 830f9990 Cid 0428.0e94 Peb: 7ffdf000 Win32Thread: 00000000 WAIT: (UserRequest) UserMode Non-Alertable
```

```
[...]
```

```
ChildEBP RetAddr Args to Child
```

```
0031f74c 7784b071 00000000 00000000 7ffdb000 ntdll!RtlpWaitOnCriticalSection+0x154
```

```
0031f774 00a91150 00a9b7a8 00000000 00a91452 ntdll!RtlEnterCriticalSection+0x152
```

```
WARNING: Stack unwind information not available. Following frames may be wrong.
```

```
0031f7c8 76113833 7ffdb000 0031f814 7784a9bd Application+0x1150
```

```
0031f7d4 7784a9bd 7ffdb000 003114bf 00000000 kernel32!BaseThreadInitThunk+0xe
```

```
0031f814 00000000 00a914a9 7ffdb000 00000000 ntdll!_RtlUserThreadStart+0x23
```

```
THREAD 886ee030 Cid 0428.0ef4 Peb: 7ffdde000 Win32Thread: 00000000 WAIT: (UserRequest) UserMode Non-Alertable
```

```
[...]
```

```
ChildEBP RetAddr Args to Child
```

```
0098fcbb 77f881b1 00000000 00000000 001614a0 ntdll!RtlpUnWaitCriticalSection+0x1b
```

```
0098fce0 00a9102e 00a9b7a8 00000000 00000000 ntdll!RtlEnterCriticalSection+0x152
```

```
WARNING: Stack unwind information not available. Following frames may be wrong.
```

```
0098fd28 00a91275 0098fd3c 76113833 001614a0 Application+0x102e
```

```
0098fd30 76113833 001614a0 0098fd7c 7784a9bd Application+0x1275
```

```
0098fd3c 7784a9bd 001614a0 009811d7 00000000 kernel32!BaseThreadInitThunk+0xe
```

```
0098fd7c 00000000 00a911ff 001614a0 00000000 ntdll!_RtlUserThreadStart+0x23
```

In the case of **Multiple Exceptions** (page 714) or even a single exception on one thread involving invalid access to a structure field, the reference to the same structure on a different thread may point to possible synchronization problems.

## Small Value

Sometimes we see the so-called **Small Values** in memory (such as on raw stack) or in CPU registers which can be ASCII or UNICODE value, some ID or even a handle. When in aggregates they can form a certain **Semantic Structure** (page 860) such as a PID.TID example or **Regular Data** (page 833) pattern. Here we illustrate a handle example (also an example of **Wait Chain** analysis in user space, page 1092):

```
0:000> kv
Child-SP          RetAddr          : Args to
Child
00000000`0016de78 000007fe`fcf010dc : 00000000`02c79fa0 00000000`08c3faf0 00000000`021551f0
00000000`08c3fb00 : ntdll!NtWaitForSingleObject+0xa
00000000`0016de80 000007fe`f90e6d7f : 00000000`10b40010 00000000`10b40010 00000000`00000000
00000000`000007e0 : KERNELBASE!WaitForSingleObjectEx+0x79
[...]

0:000> !handle 00000000`000007e0 ff
Handle 00000000000007d0
Type      Thread
Attributes 0
GrantedAccess 0x1fffff:
    Delete,ReadControl,WriteDac,WriteOwner,Synch
    Terminate,Suspend,Alert,GetContext,SetContext,SetInfo,QueryInfo,SetToken,
Impersonate,DirectImpersonate
HandleCount 5
PointerCount 9
Name      <none>
Object specific information
    Thread Id 278c.a58
    Priority 13
    Base Priority 0

0:000> ~[a58]s
ntdll!NtWaitForMultipleObjects+0xa:
00000000`770c186a c3          ret

0:002> kv
Child-SP          RetAddr          : Args to
Child
00000000`0f6af758 000007fe`fcf01430 : 00000000`00000025 00000000`00000000 00000000`00000000
000007fe`e35a1fb0 : ntdll!NtWaitForMultipleObjects+0xa
00000000`0f6af760 00000000`76e61220 : 00000000`0f6af8a8 00000000`0f6af890 00000000`00000000
00000000`00000000 : KERNELBASE!WaitForMultipleObjectsEx+0xe8
[...]

0:002> dp 00000000`0f6af890 L4
00000000`0f6af890 00000000`00000dbc 00000000`000007c0
00000000`0f6af8a0 00000000`00000000 00000000`00000000
```

```
0:002> !handle dbc ff
Handle 000000000000dbc
  Type      Thread
  Attributes 0
  GrantedAccess 0x1fffff:
    Delete,ReadControl,WriteDac,WriteOwner,Synch
    Terminate,Suspend,Alert,GetContext,SetContext,SetInfo,QueryInfo,SetToken,
Impersonate,DirectImpersonate
  HandleCount 2
  PointerCount 4
  Name      <none>
Object specific information
  Thread Id 278c.24ac
  Priority 14
  Base Priority 0

0:002> !handle 7c0 ff
Handle 0000000000007c0
  Type      Thread
  Attributes 0
  GrantedAccess 0x1fffff:
    Delete,ReadControl,WriteDac,WriteOwner,Synch
    Terminate,Suspend,Alert,GetContext,SetContext,SetInfo,QueryInfo,SetToken,
Impersonate,DirectImpersonate
  HandleCount 2
  PointerCount 4
  Name      <none>
Object specific information
  Thread Id 278c.628
  Priority 14
  Base Priority 0
```

## Comments

Sometimes a small value can be coincidentally a valid handle value and at the same time a valid thread or process id.

## Software Exception

**Software Exception** is mentioned in **Activation Context** (page 60), **Exception Module** (page 328), **Missing Component** (static linkage, page 672), **Self-Dump** (page 850), **Stack Overflow** (software implementation, page 910), and **Translated Exception** (page 1013) patterns. A typical example of **Software Exceptions** is **C++ Exception** (page 108) pattern.

**Software Exceptions**, such as *not enough memory*, are different from the so-called *hardware exceptions* by being predictable, synchronous, and detected by software code itself. Hardware exceptions such as divide by zero, access violation, and memory protection, on the contrary, are unpredictable and detected by hardware. Of course, it is possible to do some checks before code execution, and then throw **Software Exception** or some diagnostic message for a would be hardware exception. See, for example, **Self-Diagnosis** pattern for user mode (page 847) and its corresponding equivalent for kernel mode (page 844).

In Windows memory dumps, we may see *RaiseException* call in user space stack trace, such as from **Data Correlation** pattern example (page 180):

```
0:000> kL
ChildEBP RetAddr
0012e950 78158e89 kernel32!RaiseException+0x53
0012e988 7830770c msvcr80!_CxxThrowException+0x46
0012e99c 783095bc mfc80u!AfxThrowMemoryException+0x19
0012e9b4 02afa8ca mfc80u!operator new+0x27
0012e9c8 02b0992f ModuleA!std::_Allocate<...>+0x1a
0012e9e0 02b09e7c ModuleA!std::vector<double, std::allocator >::vector<double, std::allocator >+0x3f
[...]</double, std::allocator>/double, std::allocator
```

When looking for **Multiple Exceptions** (page 706) or **Hidden Exceptions** (page 455), we may also want to check for such calls.

## Comments

Here is an example of software exceptions which are collected and then enveloped by an internal software exception, see **Multiple Exceptions (Stowed)** analysis pattern for additional references, page 714.

```
0:008> kc
#
00 combbase!RoFailFastWithErrorContextInternal2
01 combbase!RoFailFastWithErrorContextInternal
02 Windows_UI_Xaml!DirectUI::ErrorHelper::ProcessUnhandledError
03 Windows_UI_Xaml!DirectUI::FinalUnhandledErrorDetectedRegistration::OnFinalUnhandledErrorDetected
04 Windows_UI_Xaml!Microsoft::WRL::Details::InvokeHelper, Windows::Foundation::IEventHandler,
Microsoft::WRL::FtmBase, Microsoft::WRL::Details::Nil, Microsoft::WRL::Details::Nil,
Microsoft::WRL::Details::Nil, Microsoft::WRL::Details::Nil,
Microsoft::WRL::Details::Nil, Microsoft::WRL::Details::Nil, Microsoft::WRL::Details::Nil>, long
```

```

(_stdcall*)(IInspectable *,Windows::ApplicationModel::Core::IUnhandledErrorDetectedEventArgs
*),2>::Invoke
05 twinapi_appcore!Windows::Internal::Details::GitInvokeHelper,Windows::Internal::GitPtr,2>::Invoke
06 twinapi_appcore!Windows::ApplicationModel::Core::UnhandledErrorInvokeHelper::Invoke
07 twinapi_appcore!Microsoft::WRL::InvokeTraits<2>::InvokeDelegates< ,Windows::Foundation::IEventHandler
>
08 twinapi_appcore!Microsoft::WRL::EventSource,Microsoft::WRL::InvokeModeOptions<2> >::DoInvoke< >
09 twinapi_appcore!Windows::ApplicationModel::Core::CoreApplication::ForwardLocalError
0a twinapi_appcore!Windows::ApplicationModel::Core::CoreApplicationFactory::ForwardLocalError
0b combase!CallErrorForwarder
0c combase!RoReportFailedDelegate
0d twinapi_appcore!wil::ErrorHandlerHelpers::Instance'::`2'::`dynamic atexit destructor for 'wrapper"
0e twinapi_appcore!Microsoft::WRL::EventSource,Microsoft::WRL::InvokeModeOptions< -2> >::DoInvoke< >
0f twinapi_appcore!Windows::ApplicationModel::Core::CoreApplicationView::Activate
10 rpcrt4!Invoke
11 rpcrt4!NdrStubCall2
12 combase!CStdStubBuffer_Invoke
13 rpcrt4!CStdStubBuffer_Invoke
14 combase!InvokeStubWithExceptionPolicyAndTracing::__17:::operator()
15 combase!ObjectMethodExceptionHandlingAction< >
16 combase!InvokeStubWithExceptionPolicyAndTracing
17 combase!DefaultStubInvoke
18 combase!SyncStubCall::Invoke
19 combase!SyncServerCall::StubInvoke
1a combase!StubInvoke
1b combase!ServerCall::ContextInvoke
1c combase!CServerChannel::ContextInvoke
1d combase!DefaultInvokeInApartment
1e combase!ASTAInvokeInApartment
1f combase!AppInvoke
20 combase!ComInvokeWithLockAndIPID
21 combase!ComInvoke
22 combase!ThreadDispatch
23 combase!CComApartment::ASTAHandleMessage
24 combase!ASTAWaitContext::Wait
25 combase!ASTAWaitInNewContext
26 combase!ASTAThreadWaitForHandles
27 combase!CoWaitForMultipleHandles
28 twinapi_appcore!CTSimpleArray,CSimpleArrayStandardCompareHelper,CSimpleArrayStandardMergeHelper
>::RemoveAt
29 SHCore!CTSimpleArray,4294967294,CTPolicyCoTaskMem >,CSimpleArrayStandardCompareHelper
>,CSimpleArrayStandardMergeHelper > >:_Add const &>
2a kernel32!BaseThreadInitThunk
2b ntdll!_RtlUserThreadStart
2c ntdll!_RtlUserThreadStart

0:008> !error c000027b
Error code: (NTSTATUS) 0xc000027b (3221226107) - An application-internal exception has occurred.

```

## Special Process

**Special Stack Trace** pattern (page 882) is about stack traces not present in normal crash dumps. **Special Process** is a similar pattern of processes not running during normal operation or highly domain specific processes that are the sign of certain software environment, for example, OS running inside VMWare or VirtualPC<sup>188</sup>. Here we'll see one example when identifying specific process led to successful problem identification inside a complete memory dump.

Inspection of running processes shows the presence of Dr. Watson:

```
5: kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS 89d9b648 SessionId: none Cid: 0004 Peb: 00000000 ParentCid: 0000
  DirBase: 54d5d020 ObjectTable: e1000e20 HandleCount: 1711.
  Image: System

PROCESS 8979b758 SessionId: none Cid: 01b0 Peb: 7ffdd000 ParentCid: 0004
  DirBase: 54d5d040 ObjectTable: e181d8b0 HandleCount: 29.
  Image: smss.exe

PROCESS 89793cf0 SessionId: 0 Cid: 01e0 Peb: 7ffde000 ParentCid: 01b0
  DirBase: 54d5d060 ObjectTable: e13eea10 HandleCount: 1090.
  Image: csrss.exe

...
...
...

PROCESS 8797a600 SessionId: 1 Cid: 17d0 Peb: 7ffdc000 ParentCid: 1720
  DirBase: 54d5d8c0 ObjectTable: e2870af8 HandleCount: 243.
  Image: explorer.exe

PROCESS 87966d88 SessionId: 2 Cid: 0df0 Peb: 7ffd4000 ParentCid: 01b0
  DirBase: 54d5d860 ObjectTable: e284cd48 HandleCount: 53.
  Image: csrss.exe

PROCESS 879767c8 SessionId: 0 Cid: 0578 Peb: 7ffde000 ParentCid: 0ca8
  DirBase: 54d5d8a0 ObjectTable: e2c05268 HandleCount: 180.
  Image: drwtsn32.exe
```

Inspecting stack traces shows that *drwtsn32.exe* is waiting for a debugger event so there must be some debugging target (debuggee):

```
5: kd> .process /r /p 879767c8
Implicit process is now 879767c8
Loading User Symbols
```

---

<sup>188</sup> Memory Dumps from Virtual Images, Memory Dump Analysis Anthology, Volume 1, page 219

```

5: kd> !process 879767c8
PROCESS 879767c8 SessionId: 0 Cid: 0578 Peb: 7ffde000 ParentCid: 0ca8
  DirBase: 54d5d8a0 ObjectTable: e2c05268 HandleCount: 180.
  Image: drwtsn32.exe
  VadRoot 88a33cd0 Vads 59 Clone 0 Private 1737. Modified 10792. Locked 0.
  DeviceMap e10028e8
    Token e2ee2330
    ElapsedTime 00:01:12.703
    UserTime 00:00:00.203
    KernelTime 00:00:00.031
    QuotaPoolUsage[PagedPool] 52092
    QuotaPoolUsage[NonPagedPool] 2360
    Working Set Sizes (now,min,max) (2488, 50, 345) (9952KB, 200KB, 1380KB)
    PeakWorkingSetSize 2534
    VirtualSize 34 Mb
    PeakVirtualSize 38 Mb
    PageFaultCount 13685
    MemoryPriority BACKGROUND
    BasePriority 6
    CommitCharge 1927

THREAD 87976250 Cid 0578.04bc Teb: 7fffd000 Win32Thread: bc14a008 WAIT: (Unknown) UserMode Non-Alertable
  87976558 Thread
  Not impersonating
  DeviceMap e10028e8
  Owning Process 879767c8 Image: drwtsn32.exe
  Wait Start TickCount 13460 Ticks: 4651 (0:00:01:12.671)
  Context Switch Count 15 LargeStack
  UserTime 00:00:00.000
  KernelTime 00:00:00.000
  Win32 Start Address drwtsn32!mainCRTStartup (0x01007c1d)
  Start Address kernel32!BaseProcessStartThunk (0x7c8217f8)
  Stack Init f433b000 Current f433ac60 Base f433b000 Limit f4337000 Call 0
  Priority 6 BasePriority 6 PriorityDecrement 0
  ChildEBP RetAddr
  f433ac78 80833465 nt!KiSwapContext+0x26
  f433aca4 80829a62 nt!KiSwapThread+0x2e5
  f433acec 80938d0c nt!KeWaitForSingleObject+0x346
  f433ad50 8088978c nt!NtWaitForSingleObject+0x9a
  f433ad50 7c9485ec nt!KiFastCallEntry+0xfc
  0007fe98 7c821c8d ntdll!KiFastSystemCallRet
  0007feac 01005557 kernel32!WaitForSingleObject+0x12
  0007ff0c 01003ff8 drwtsn32!NotifyWinMain+0x1ba
  0007ff44 01007d4c drwtsn32!main+0x31
  0007ffc0 7c82f23b drwtsn32!mainCRTStartup+0x12f
  0007fff0 00000000 kernel32!BaseProcessStart+0x23

```

```

THREAD 87976558 Cid 0578.0454 Peb: 7ffdc000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Non-Alertable
    89de2e50 NotificationEvent
    879765d0 NotificationTimer
Not impersonating
DeviceMap          e10028e8
Owning Process     879767c8      Image:           drwtsn32.exe
Wait Start TickCount 18102       Ticks: 9 (0:00:00:00.140)
Context Switch Count 1163
UserTime           00:00:00.203
KernelTime         00:00:00.031
Win32 Start Address drwtsn32!DispatchDebugEventThread (0x01003d6d)
Start Address kernel32!BaseThreadStartThunk (0x7c8217ec)
Stack Init f4e26000 Current f4e25be8 Base f4e26000 Limit f4e23000 Call 0
Priority 6 BasePriority 6 PriorityDecrement 0
ChildEBP RetAddr
f4e25c00 80833465 nt!KiSwapContext+0x26
f4e25c2c 80829a62 nt!KiSwapThread+0x2e5
f4e25c74 809a06ab nt!KeWaitForSingleObject+0x346
f4e25d4c 8088978c nt!NtWaitForDebugEvent+0xd5
f4e25d4c 7c9485ec nt!KiFastCallEntry+0xfc
0095ed20 60846f8f nt!KiFastSystemCallRet
0095ee6c 60816ecf dbgeng!LiveUserTargetInfo::WaitForEvent+0x1fa
0095ee88 608170d3 dbgeng!WaitForAnyTarget+0x45
0095eecc 60817270 dbgeng!RawWaitForEvent+0x15f
0095eee4 01003f8d dbgeng!DebugClient::WaitForEvent+0x80
0095ffb8 7c824829 drwtsn32!DispatchDebugEventThread+0x220
0095ffec 00000000 kernel32!BaseThreadStart+0x34

```

Knowing that a debugger suspends threads in a debuggee (**Suspended Thread** pattern, page 964) we see the problem process indeed:

```

5: kd> !process 0 2
**** NT ACTIVE PROCESS DUMP ****

...
...
...

PROCESS 898285b0 SessionId: 0 Cid: 0ca8 Peb: 7ffda000 ParentCid: 022c
    DirBase: 54d5d500 ObjectTable: e2776880 HandleCount: 2.
    Image: svhost.exe

THREAD 888b8668 Cid 0ca8.1448 Peb: 00000000 Win32Thread: 00000000 WAIT: (Unknown) KernelMode Non-Alertable
SuspendCount 2
888b87f8 Semaphore Limit 0x2

```

Dumping its thread stacks shows only one system thread where we normally expect plenty of them originated from user space. There is also the presence of a debug port:

```

5: kd> .process /r /p 898285b0
Implicit process is now 898285b0
Loading User Symbols

```

```

5: kd> !process 898285b0
PROCESS 898285b0 SessionId: 0 Cid: 0ca8 Peb: 7ffda000 ParentCid: 022c
  DirBase: 54d5d500 ObjectTable: e2776880 HandleCount: 2.
  Image: svchost.exe
  VadRoot 88953220 Vads 209 Clone 0 Private 901. Modified 3. Locked 0.
  DeviceMap e10028e8
    Token                      e27395b8
    ElapsedTime                00:03:25.640
    UserTime                   00:00:00.156
    KernelTime                 00:00:00.234
    QuotaPoolUsage[PagedPool]  82988
    QuotaPoolUsage[NonPagedPool] 8824
    Working Set Sizes (now,min,max) (2745, 50, 345) (10980KB, 200KB, 1380KB)
    PeakWorkingSetSize        2819
    VirtualSize                82 Mb
    PeakVirtualSize            83 Mb
    PageFaultCount             4519
    MemoryPriority              BACKGROUND
    BasePriority                  6
    CommitCharge                  1380
    DebugPort                    89de2e50

THREAD 888b8668 Cid 0ca8.1448 Teb: 00000000 Win32Thread: 00000000 WAIT: (Unknown) KernelMode Non-
Alertable
SuspendCount 2
  888b87f8 Semaphore Limit 0x2
Not impersonating
DeviceMap          e10028e8
Owning Process     898285b0      Image:           svchost.exe
Wait Start TickCount 13456      Ticks: 4655 (0:00:01:12.734)
Context Switch Count 408
UserTime           00:00:00.000
KernelTime         00:00:00.000
Start Address driverA!DriverThread (0xf6fb8218)
Stack Init f455b000 Current f455a3ac Base f455b000 Limit f4558000 Call 0
Priority 6 BasePriority 6 PriorityDecrement 0
ChildEBP RetAddr
f455a3c4 80833465 nt!KiSwapContext+0x26
f455a3f0 80829a62 nt!KiSwapThread+0x2e5
f455a438 80833178 nt!KeWaitForSingleObject+0x346
f455a450 8082e01f nt!KiSuspendThread+0x18
f455a498 80833480 nt!KiDeliverApc+0x117
f455a4d0 80829a62 nt!KiSwapThread+0x300
f455a518 f6fb7f13 nt!KeWaitForSingleObject+0x346
f455a548 f4edd457 driverA!WaitForSingleObject+0x75
f455a55c f4edcd8 driverB!DeviceWaitForRead+0x19
f455ad90 f6fb8265 driverB!InputThread+0x17e
f455adac 80949b7c driverA!DriverThread+0x4d
f455addc 8088e062 nt!PspSystemThreadStartup+0x2e
00000000 00000000 nt!KiThreadStartup+0x16

```

The most likely scenario was that *svchost.exe* experienced an unhandled exception that triggered the launch of a postmortem debugger<sup>189</sup> such as Dr. Watson.

Other similar examples of this pattern might include the presence of *WerFault.exe* on Vista, NTSD and other JIT debuggers running.

## Comments

---

Some processes briefly appear to do a specialized task and normally don't present in memory. Their presence in the memory dump can point to abnormal conditions such as **Wait Chains** (page 1082), **Blocked Threads** (page 80), and many others. For example, multiple *LogonUI* processes in terminal session environments on W2K8.

Another error reporting example is the presence of *DW20.exe*.

---

<sup>189</sup> Who Calls the Postmortem Debugger?, Memory Dump Analysis Anthology, Volume 1, page 113

## Special Stack Trace

Sometimes we encounter thread stacks related to debugger events like *Process Exit*, *Module Load* or *Unload*. These thread stacks are not normally encountered in healthy process dumps and, statistically speaking, when a process terminates or unloads a library the chances to save a memory dump manually using process dumpers like *userdump.exe* or Task Manager in Vista are very low unless an interactive debugger was attached, or breakpoints were set in advance. Therefore, the presence of such threads in a captured crash dump usually indicates some problem or at least focuses attention on the procedure used to save a dump. Such pattern merits its own name: **Special Stack Trace**.

For example, one process dump had the following stack trace showing process termination initiated from .NET runtime:

```
STACK_TEXT:
0012fc2c 7c827c1b ntdll!KiFastSystemCallRet
0012fc30 77e668c3 ntdll!NtTerminateProcess+0xc
0012fd24 77e66905 KERNEL32!_ExitProcess+0x63
0012fd38 01256d9b KERNEL32!ExitProcess+0x14
0012ff60 01256dc7 mscorwks!SafeExitProcess+0x11a
0012ff6c 011c5fa4 mscorwks!DisableRuntime+0xd0
0012ffb0 79181b5f mscorwks!_CorExeMain+0x8c
0012ffc0 77e6f23b mscoree!_CorExeMain+0x2c
0012fff0 00000000 KERNEL32!BaseProcessStart+0x23
```

The original problem was an error message box, and the application disappeared when a user dismissed the message. How was the dump saved? Someone advised attaching NTSD to that process, hit 'g' and then save the memory dump when the process breaks into the debugger again. So the problem was already gone by that time, and the better way would have been to create the manual user dump of that process when it was displaying the error message.

## Comments

Another example is blocked *csrss.exe* during session logoff in terminal services environments:

```
ntdll!KiFastSystemCallRet
ntdll!NtWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
winsrv!W32WinStationTerminate
winsrv!TerminalServerRequestThread
```

## Special Thread

### .NET CLR

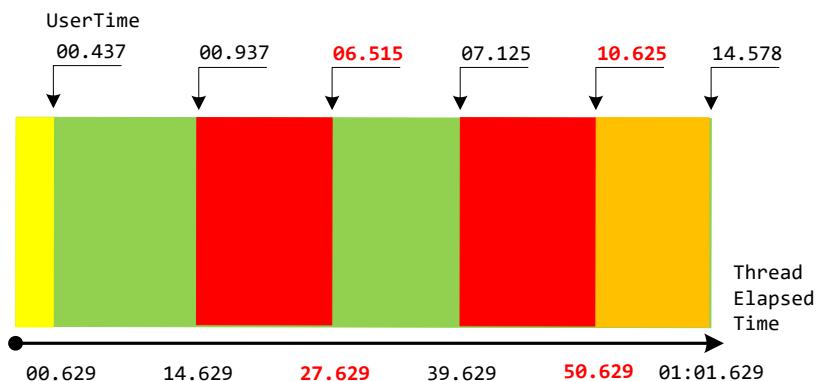
When analyzing memory dumps from specific application platforms, we see threads having definite purpose as a part of that specific platform architecture, design, and implementation. For example, in applications and services involving .NET CLR we see the following threads:

```
0:000> !Threads -special
ThreadCount:          9
UnstartedThread:      0
BackgroundThread:     7
PendingThread:        0
DeadThread:           1
Hosted Runtime:       no
PreEmptive GC Alloc Lock
 ID OSID ThreadOBJ State   GC      Context             Domain  Count APT Exception
0 1 b10 002fbe88 6020   Enabled  0acbdebc:0acbf5a4 002f17d0 0    STA
2 2 bf0 00306b18 b220   Enabled  00000000:00000000 002f17d0 0    MTA (Finalizer)
3 3 b34 0034c188 b220   Enabled  00000000:00000000 002f17d0 0    MTA
XXXX 5 0037e3e0 19820  Enabled  00000000:00000000 002f17d0 0    Ukn
5 7 700 04b606c8 200b220 Enabled  00000000:00000000 002f17d0 0    MTA
6 4 ec4 04baffa0 200b220 Enabled  00000000:00000000 002f17d0 0    MTA
8 8 10c 04bf19b8 8009220 Enabled  00000000:00000000 002f17d0 0    MTA (Threadpool Completion Port)
9 11 464 0be106d8 1220   Enabled  00000000:00000000 002f17d0 0    Ukn
10 10 da0 003c7958 7220   Disabled 00000000:00000000 0be1dd00 0    STA

OSID Special thread type
1 c08 DbgHelper
2 bf0 Finalizer
7 f54 Gate
8 10c IOCompletion
9 464 ADUnloadHelper
```

## Spike Interval

Sometimes, when an application is sluggish, periodically consumes CPU, it is possible to create a set of consecutive memory dumps of the same process to see the temporal development of any thread CPU consumption and figure out potential spike interval(s). For example, the following diagram was plotted from !runaway WinDbg command output for thread #1:



The 3rd and the 5th user process memory dumps in addition to increased CPU consumption also had corresponding non-waiting stack trace frames caught while executing some CPU instructions in *ModuleA* (not preempted with saved context). The first memory dump (the first left bar) with 437 ms user time spent out of 629 ms elapsed time also had a non-waiting stack trace, but we consider it a normal application startup CPU consumption spike.

## Spiking Thread

### Linux

This is a variant of **Spiking Thread** pattern previously described for Mac OS X (page 886) and Windows (page 888) platforms:

```
(gdb) info threads
Id Target Id Frame
6 LWP 3712 0x00000000004329d1 in nanosleep ()
5 LWP 3717 0x00000000004007a3 in isnan ()
4 LWP 3716 0x00000000004329d1 in nanosleep ()
3 LWP 3715 0x00000000004329d1 in nanosleep ()
2 LWP 3714 0x00000000004329d1 in nanosleep ()
* 1 LWP 3713 0x00000000004329d1 in nanosleep ()
```

We notice a non-waiting thread and switch to it:

```
(gdb) thread 5
[Switching to thread 5 (LWP 3717)]
#0 0x00000000004007a3 in isnan ()

(gdb) bt
#0 0x00000000004007a3 in isnan ()
#1 0x0000000000400743 in sqrt ()
#2 0x0000000000400528 in procB ()
#3 0x0000000000400639 in bar_five ()
#4 0x0000000000400649 in foo_five ()
#5 0x0000000000400661 in thread_five ()
#6 0x0000000000403e30 in start_thread ()
#7 0x0000000000435089 in clone ()
#8 0x0000000000000000 in ?? ()
```

If we disassemble the return address for *procB* function to come back from *sqrt* call we see an infinite loop:

```
(gdb) disassemble 0x400528
Dump of assembler code for function procB:
0x0000000000400500 <+0>: push    %rbp
0x0000000000400501 <+1>: mov     %rsp,%rbp
0x0000000000400504 <+4>: sub    $0x20,%rsp
0x0000000000400508 <+8>: movabs $0x3fd5555555555555,%rax
0x0000000000400512 <+18>: mov    %rax,-0x8(%rbp)
0x0000000000400516 <+22>: mov    -0x8(%rbp),%rax
0x000000000040051a <+26>: mov    %rax,-0x18(%rbp)
0x000000000040051e <+30>: movsd -0x18(%rbp),%xmm0
0x0000000000400523 <+35>: callq 0x400710 <sqrt>
0x0000000000400528 <+40>: movsd %xmm0,-0x18(%rbp)
0x000000000040052d <+45>: mov    -0x18(%rbp),%rax
0x0000000000400531 <+49>: mov    %rax,-0x8(%rbp)
0x0000000000400535 <+53>: jmp    0x400516 <procB+22>
End of assembler dump.
```

## Mac OS X

This is a Mac OS X / GDB counterpart to **Spiking Thread** pattern:

```
(gdb) info threads
4 0x00007fff85b542df in sqrt$fenv_access_off ()
3 0x00007fff8616ee42 in __semwait_signal ()
2 0x00007fff8616ee42 in __semwait_signal ()
* 1 0x00007fff8616ee42 in __semwait_signal ()
```

We notice a non-waiting thread and switch to it:

```
(gdb) thread 4
[Switching to thread 4 (core thread 3)]
0x00007fff85b542df in sqrt$fenv_access_off ()

(gdb) bt
#0 0x00007fff85b542df in sqrt$fenv_access_off ()
#1 0x000000010cc85dc9 in thread_three (arg=0x7fff6c884ac0)
#2 0x00007fff8fac68bf in _pthread_start ()
#3 0x00007fff8fac9b75 in thread_start ()
```

If we disassemble the return address for *thread\_three* function called from *SQRT* call we see an infinite loop:

```
(gdb) disass 0x000000010cc85dc9
Dump of assembler code for function thread_three:
0x000000010cc85db0 <thread_three+0>:    push %rbp
0x000000010cc85db1 <thread_three+1>:    mov %rsp,%rbp
0x000000010cc85db4 <thread_three+4>:    sub $0x10,%rsp
0x000000010cc85db8 <thread_three+8>:    mov %rdi,-0x10(%rbp)
0x000000010cc85dbc <thread_three+12>:   mov -0x10(%rbp),%ax
0x000000010cc85dc0 <thread_three+16>:   movsd (%rax),%xmm0
0x000000010cc85dc4 <thread_three+20>:   callq 0x10cc85eac <dyld_stub_sqrt>
0x000000010cc85dc9 <thread_three+25>:   mov -0x10(%rbp),%rax
0x000000010cc85dcd <thread_three+29>:   movsd %xmm0,(%rax)
0x000000010cc85dd1 <thread_three+33>:   jmpq 0x10cc85dbc <thread_three+12>
End of assembler dump.
```

Here's the source code of the modeling application:

```
void * thread_one (void *arg)
{
    while (1)
    {
        sleep (1);
    }

    return 0;
}
```

```
void * thread_two (void *arg)
{
    while (1)
    {
        sleep (2);
    }

    return 0;
}

void * thread_three (void *arg)
{
    while (1)
    {
        *(double*)arg=sqrt(*(double *)arg);
    }

    return 0;
}

int main(int argc, const char * argv[])
{
    pthread_t threadID_one, threadID_two, threadID_three;
    double result = 0xffffffff;

    pthread_create (&threadID_one, NULL, thread_one, NULL);
    pthread_create (&threadID_two, NULL, thread_two, NULL);
    pthread_create (&threadID_three, NULL, thread_three, &result);
    pthread_join(threadID_three, NULL);

    return 0;
}
```

## Windows

If we have a process dump with many threads, it is sometimes difficult to see which thread there was spiking CPU. This is why it is always good to have some screenshots or notes from QSlice or Process Explorer showing spiking thread ID and process ID. The latter ID is to make sure that the process dump was from the correct process. New process dumpers and tools from Microsoft (userdump.exe, procdump) save thread time information so we can open the dump and see the time spent in kernel and user mode for any thread by entering **!runaway** command. However, if that command shows many threads with similar CPU consumption, it will not highlight the particular thread that was spiking at the time the crash dump was saved, so screenshots are still useful in some cases.

What to do if we don't have the spiking thread ID? We need to look at all threads and find those that are not waiting. Almost all threads are waiting most of the time. So the chances to dump the normal process and see some active threads are very low. If the thread is waiting, the top function on its stack is usually:

```
ntdll!KiFastSystemCallRet
```

and below that we can see some blocking calls waiting on some synchronization object, *Sleep* API call, IO completion or for LPC reply:

```
0:085> ~*kv
...
...
64 Id: 1b0.120c Suspend: -1 Teb: 7ff69000 Unfrozen
ChildEBP RetAddr Args to Child
02defe18 7c90e399 ntdll!KiFastSystemCallRet
02defe1c 77e76703 ntdll!NtReplyWaitReceivePortEx+0xc
02deff80 77e76c22 rpct4!LRPC_ADDRESS::ReceiveLotsaCalls+0xf4
02deff88 77e76a3b rpct4!RecvLotsaCallsWrapper+0xd
02deffa8 77e76c0a rpct4!BaseCachedThreadRoutine+0x79
02deffb4 7c80b683 rpct4!ThreadStartRoutine+0x1a
02deffec 00000000 kernel32!BaseThreadStart+0x37

65 Id: 1b0.740 Suspend: -1 Teb: 7ff67000 Unfrozen
ChildEBP RetAddr Args to Child
02edff44 7c90d85c ntdll!KiFastSystemCallRet
02edff48 7c8023ed ntdll!NtDelayExecution+0xc
02edffa0 57cde2dd kernel32!SleepEx+0x61
02edffb4 7c80b683 component!foo+0x35
02edffec 00000000 kernel32!BaseThreadStart+0x37
```

```

66 Id: 1b0.131c Suspend: -1 Teb: 7ff66000 Unfrozen
ChildEBP RetAddr Args to Child
02f4ff38 7c90e9c0 ntdll!KiFastSystemCallRet
02f4ff3c 7c8025cb ntdll!ZwWaitForSingleObject+0xc
02f4ffa0 72001f65 kernel32!WaitForSingleObjectEx+0xa8
02f4ffb4 7c80b683 component!WorkerThread+0x15
02f4ffec 00000000 kernel32!BaseThreadStart+0x37

67 Id: 1b0.1320 Suspend: -1 Teb: 7ff65000 Unfrozen
ChildEBP RetAddr Args to Child
02f8fe1c 7c90e9ab ntdll!KiFastSystemCallRet
02f8fe20 7c8094e2 ntdll!ZwWaitForMultipleObjects+0xc
02f8feb0 7e4195f9 kernel32!WaitForMultipleObjectsEx+0x12c
02f8ff18 7e4196a8 user32!RealMsgWaitForMultipleObjectsEx+0x13e
02f8ff34 720019f6 user32!MsgWaitForMultipleObjects+0x1f
02f8ffa0 72001a29 component!bar+0xd9
02f8ffb4 7c80b683 component!MonitorWorkerThread+0x11
02f8ffec 00000000 kernel32!BaseThreadStart+0x37

68 Id: 1b0.1340 Suspend: -1 Teb: 7ff63000 Unfrozen
ChildEBP RetAddr Args to Child
0301ff1c 7c90e31b ntdll!KiFastSystemCallRet
0301ff20 7c80a746 ntdll!ZwRemoveIoCompletion+0xc
0301ff4c 57d46e65 kernel32!GetQueuedCompletionStatus+0x29
0301ffb4 7c80b683 component!AsyncEventsThread+0x91
0301ffec 00000000 kernel32!BaseThreadStart+0x37
...
...
...
# 85 Id: 1b0.17b4 Suspend: -1 Teb: 7ffd4000 Unfrozen
ChildEBP RetAddr Args to Child
00dafffc8 7c9507a8 ntdll!DbgBreakPoint
00dafff4 00000000 ntdll!DbgUiRemoteBreakin+0x2d

```

Therefore, if we have a different thread like this one below the chances that it was spiking are bigger:

```

58 Id: 1b0.9f4 Suspend: -1 Teb: 7ff75000 Unfrozen
ChildEBP RetAddr Args to Child
0280f64c 500af723 componentB!DoSomething+32
0280f85c 500b5391 componentB!CheckSomething+231
0280f884 500b7a3f componentB!ProcessWorkItem+9f
0301ffec 00000000 kernel32!BaseThreadStart+0x37

```

There is no *KiFastSystemCallRet* on top, and if we look at the currently executing instruction we see that it does some copy operation:

```
0:085> ~58r
eax=00000000 ebx=0280fdd4 ecx=0000005f edx=00000000 esi=03d30444 edi=0280f6dc
eip=500a4024 esp=0280f644 ebp=0280f64c iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010202
componentB!DoSomething+32:
500a4024 f3a5 rep movs dword ptr es:[edi],dword ptr [esi] es:0023:0280f6dc=00000409
ds:0023:03d30444=00000409
```

In a kernel or a complete memory dump we can see CPU spikes by checking *KernelTime* and *UserTime*:

```
0: kd> !thread 88b66768
THREAD 88b66768 Cid 01fc.1550 Teb: 7ffad000 Win32Thread: bc18f240 RUNNING on processor 1
IRP List:
89716008: (0006,0094) Flags: 00000a00 Mdl: 00000000
Impersonation token: e423a030 (Level Impersonation)
DeviceMap e3712480
Owning Process 8a0a56a0 Image: SomeSvc.exe
Wait Start TickCount 1782229 Ticks: 0
Context Switch Count 877610 LargeStack
UserTime 00:00:01.0078
KernelTime 02:23:21.0718
```

By default **!runaway** command shows only user mode time. By specifying additional flags, it is possible to see both kernel and user times:

```
0:000> !runaway 3
User Mode Time
Thread Time
8:15a4 0 days 0:12:32.812
0:1c00 0 days 0:00:00.312
9:1b50 0 days 0:00:00.296
22:2698 0 days 0:00:00.046
17:22b8 0 days 0:00:00.031
14:2034 0 days 0:00:00.031
21:21b4 0 days 0:00:00.000
20:27b0 0 days 0:00:00.000
19:278c 0 days 0:00:00.000
18:2788 0 days 0:00:00.000
16:2194 0 days 0:00:00.000
15:2064 0 days 0:00:00.000
13:2014 0 days 0:00:00.000
12:1e38 0 days 0:00:00.000
11:1c54 0 days 0:00:00.000
10:1d40 0 days 0:00:00.000
7:1994 0 days 0:00:00.000
6:1740 0 days 0:00:00.000
5:1c18 0 days 0:00:00.000
4:c10 0 days 0:00:00.000
3:1774 0 days 0:00:00.000
```

```

2:1a08 0 days 0:00:00.000
1:fb8 0 days 0:00:00.000

Kernel Mode Time
Thread Time
9:1b50 0 days 1:21:54.125
8:15a4 0 days 0:02:48.390
0:1c00 0 days 0:00:00.328
14:2034 0 days 0:00:00.234
22:2698 0 days 0:00:00.156
17:22b8 0 days 0:00:00.015
21:21b4 0 days 0:00:00.000
20:27b0 0 days 0:00:00.000
19:278c 0 days 0:00:00.000
18:2788 0 days 0:00:00.000
16:2194 0 days 0:00:00.000
15:2064 0 days 0:00:00.000
13:2014 0 days 0:00:00.000
12:1e38 0 days 0:00:00.000
11:1c54 0 days 0:00:00.000
10:1d40 0 days 0:00:00.000
7:1994 0 days 0:00:00.000
6:1740 0 days 0:00:00.000
5:1c18 0 days 0:00:00.000
4:c10 0 days 0:00:00.000
3:1774 0 days 0:00:00.000
2:1a08 0 days 0:00:00.000
1:fb8 0 days 0:00:00.000

```

We see that the thread #15a4 spikes mostly in user mode but the thread #1b50 spikes mostly in kernel mode.

In the kernel and complete memory dumps we can scan all threads with *Ticks: 0* or *Elapsed Ticks: 0* to check their kernel and user times:

```

PROCESS 8782cd60 SessionId: 52 Cid: 4a58 Peb: 7fffd000 ParentCid: 1ea4
DirBase: 0a0260c0 ObjectTable: 88ab33a8 TableSize: 486.
Image: IEXPLORE.EXE
VadRoot: 87f59ea8 Clone 0 Private 2077. Modified 123. Locked 0.
DeviceMap: 880f6828
Token: e8217cd0
ElapsedTime: 0:03:09.0765
UserTime: 0:00:00.0890
KernelTime: 0:00:10.0171
QuotaPoolUsage[PagedPool] 100320
QuotaPoolUsage[NonPagedPool] 58100
Working Set Sizes (now,min,max) (4944, 50, 345) (19776KB, 200KB, 1380KB)
PeakWorkingSetSize 4974
VirtualSize 83 Mb
PeakVirtualSize 83 Mb
PageFaultCount 8544
MemoryPriority FOREGROUND
BasePriority 8
CommitCharge 2262

```

```

THREAD 87836580 Cid 4a58.57cc Teb: 7ffde000 Win32Thread: a224f1d8 WAIT: (Executive) KernelMode Non-
Alertable
89dee788 Semaphore Limit 0x7fffffff
87836668 NotificationTimer
Not impersonating
Owning Process 8782cd60
Wait Start TickCount 123758 Elapsed Ticks: 0
Context Switch Count 97636 LargeStack
UserTime 0:00:00.0593
KernelTime 0:00:08.0359
Start Address 0x7c57b70c
Win32 Start Address 0x00401ee6
Stack Init ac154770 Current ac154320 Base ac155000 Limit ac14d000 Call ac15477c
Priority 11 BasePriority 8 PriorityDecrement 0 DecrementCount 0

ChildEBP RetAddr Args to Child
ac154338 8042d8d7 00000000 8047bd00 00000001 nt!KiSwapThread+0x1b1
ac154360 80415d61 89dee788 00000000 00000000 nt!KeWaitForSingleObject+0xa3
ac15439c 8041547c 00000000 00000001 8051c501 nt!ExpWaitForResource+0x2d
ac1543b4 8046907a 8047bd00 00000001 805225e9 nt!ExAcquireResourceSharedLite+0xc6
ac1543c0 805225e9 00000000 00000001 8051c501 nt!CmpLockRegistry+0x18
ac154430 8051c718 e7c5fd08 ac15448c 00000001 nt!CmSetValueKey+0x31
ac1544b4 8046b2a9 00000798 00125c04 00000000 nt!NtSetValueKey+0x196
ac1544b4 77f88de7 00000798 00125c04 00000000 nt!KiSystemService+0xc9
00125bb0 00000000 00000000 00000000 00000000 +0x77f88de7

```

For complete and kernel dumps we can also pay attention to the output of **!running** command and to the output of **!stacks** command (*Ticks* and *ThreadState* columns).

## Comments

---

Is some legacy dumps with missing **!runaway** information we can guess spiking threads if they do some sort of a peek message loop:

```
0:012> !runaway
ERROR: !runaway: extension exception 0x80004002.
"Unable to get thread times - dumps may not have time information"

6 Id: 136c.13bc Suspend: 1 Teb: 7ffd8000 Unfrozen
ChildEBP RetAddr Args to Child
0198f914 5b2d887c 0198f938 00000000 00000000 USER32!PeekMessageA+0xfb
0198fa20 5b2d3b87 059d6704 00000001 00000000 ModuleA!Choose+0x177
[...]

0:012> ~6s
eax=000abbdd ebx=00000400 ecx=00000200 edx=00000abb esi=7ffd8000 edi=0066b3e0
eip=7e42a3e5 esp=0198f908 ebp=0198f914 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00040206
USER32!PeekMessageA+0xfb:
7e42a3e5 2b470c sub eax,dword ptr [edi+0Ch] ds:0023:0066b3ec=000abbdd
```

It is also useful to combine it with **Thread Age** pattern (page 1001) to see the magnitude of spike:

```
Process Uptime: 0 days 17:35:06.000
```

If we look at thread CPU consumption we see spike values relatively small compared to the overall process uptime:

```
0:000> !runaway f
User Mode Time
Thread Time
4:1560 0 days 0:00:08.034
7:7e0 0 days 0:00:03.338
11:1244 0 days 0:00:03.213
8:1510 0 days 0:00:02.324
3:86c 0 days 0:00:00.015
[...]
```

However, compared to **Thread Ages** we see them much bigger:

```
Elapsed Time
Thread Time
[...]
3:86c 0 days 17:35:05.213
4:1560 0 days 0:02:12.350
[...]
```

```
7:7e0 0 days 0:00:22.429  
11:1244 0 days 0:00:17.655
```

How to use WinDbg scripts to get spiking threads from the kernel and complete memory dumps<sup>190</sup>.

In kernel dumps, we can see all running thread stacks with this command:

```
!running -t
```

When executing that command for the complete memory dumps, the user portion is invalid if the thread process contexts are different from the current process on the current CPU, so manual thread inspection via **!thread 1f** is recommended.

---

<sup>190</sup> Two WinDbg Scripts That Changed the World, Memory Dump Analysis Anthology, Volume 7, page 32

## Stack Overflow

### Linux

This is a Linux variant of **Stack Overflow** (user mode) pattern previously described for Mac OS X (page 897) and Windows (page 912) platforms:

```
(gdb) bt
#0  0x000000000004004fb in procF ()
#1  0x0000000000040054b in procF ()
#2  0x0000000000040054b in procF ()
#3  0x0000000000040054b in procF ()
#4  0x0000000000040054b in procF ()
#5  0x0000000000040054b in procF ()
#6  0x0000000000040054b in procF ()
#7  0x0000000000040054b in procF ()
#8  0x0000000000040054b in procF ()
#9  0x0000000000040054b in procF ()
#10 0x0000000000040054b in procF ()
#11 0x0000000000040054b in procF ()
#12 0x0000000000040054b in procF ()
[...]

(gdb) bt -10
#15409 0x0000000000040054b in procF ()
#15410 0x0000000000040054b in procF ()
#15411 0x0000000000040054b in procF ()
#15412 0x0000000000040055b in procE ()
#15413 0x00000000000400575 in bar_one ()
#15414 0x00000000000400585 in foo_one ()
#15415 0x0000000000040059d in thread_one ()
#15416 0x00000000000401690 in start_thread (arg=<optimized out>
at pthread_create.c:304
#15417 0x0000000000432549 in clone ()
#15418 0x0000000000000000 in ?? ()
```

In the case of a stack overflow, the stack pointer is decremented beyond the stack region boundary into a non-accessible region, so any stack memory access triggers an access violation:

```
(gdb) x $rsp
0x7eff46109ec0: 0x0

(gdb) frame 1
#1  0x0000000000040054b in procF ()

(gdb) x $rsp
0x7eff4610a0e0: 0x0
```

```
(gdb) maintenance info sections
[...]
Core file:
[...]
0x7eff46109000->0x7eff4610a000 at 0x02034000: Load13 ALLOC LOAD READONLY HAS_CONTENTS
0x7eff4610a000->0x7eff4690a000 at 0x02035000: load14 ALLOC LOAD HAS_CONTENTS
[...]
```

## Mac OS X

This is a Mac OS X / GDB counterpart to **Stack Overflow** (user mode) pattern:

```
(gdb) bt 10
#0 0x0000000105dafea8 in bar (i=0)
#1 0x0000000105dafec9 in bar (i=262102)
#2 0x0000000105dafec9 in bar (i=262101)
#3 0x0000000105dafec9 in bar (i=262100)
#4 0x0000000105dafec9 in bar (i=262099)
#5 0x0000000105dafec9 in bar (i=262098)
#6 0x0000000105dafec9 in bar (i=262097)
#7 0x0000000105dafec9 in bar (i=262096)
#8 0x0000000105dafec9 in bar (i=262095)
#9 0x0000000105dafec9 in bar (i=262094)
(More stack frames follow...)
```

There are at least 262,102 frames, so we don't attempt to list them all. What we'd like to do is to get stack trace boundaries from the list of sections based on the current stack pointer address and dump the upper part of it (the stack grows from higher addresses to the lower ones) to get bottom initial stack traces:

```
(gdb) x $rsp
0x7fff651aeff0: 0x00000000
```

Because this is a stack overflow, we expect that RSP went out of page bounds, and we expect the lowest address to be 0x7fff651af000.

```
(gdb) maint info sections
[...]
Core file:
`/cores/core.2763', file type mach-o-le.
[...]
0x0000000105e0000->0x0000000105f0000 at 0x00035000: LC_SEGMENT. ALLOC LOAD CODE HAS_CONTENTS
0x00007ffff619af000->0x00007ffff651af000 at 0x00135000: LC_SEGMENT. ALLOC LOAD CODE HAS_CONTENTS
0x00007ffff651af000->0x00007ffff659af000 at 0x03935000: LC_SEGMENT. ALLOC LOAD CODE HAS_CONTENTS
0x00007ffff659af000->0x00007ffff659e4000 at 0x04135000: LC_SEGMENT. ALLOC LOAD CODE HAS_CONTENTS
0x00007ffff659e4000->0x00007ffff659e6000 at 0x0416a000: LC_SEGMENT. ALLOC LOAD CODE HAS_CONTENTS
[...]

(gdb) x/250a 0x00007ffff659af000-2000
0x7fff659ae830: 0x0 0x1500000000
0x7fff659ae840: 0x7fff659ae860 0x105dafeb9 <bar+25>
0x7fff659ae850: 0x0 0x1400000000
0x7fff659ae860: 0x7fff659ae880 0x105dafeb9 <bar+25>
0x7fff659ae870: 0x0 0x1300000000
0x7fff659ae880: 0x7fff659ae8a0 0x105dafeb9 <bar+25>
0x7fff659ae890: 0x0 0x1200000000
0x7fff659ae8a0: 0x7fff659ae8c0 0x105dafeb9 <bar+25>
0x7fff659ae8b0: 0x0 0x1100000000
0x7fff659ae8c0: 0x7fff659ae8e0 0x105dafeb9 <bar+25>
0x7fff659ae8d0: 0x0 0x1000000000
0x7fff659ae8e0: 0x7fff659ae900 0x105dafeb9 <bar+25>
```

```

0x7fff659ae8f0: 0x0 0xf00000000
0x7fff659ae900: 0x7fff659ae920 0x105dafec9 <bar+25>
0x7fff659ae910: 0x0 0xe00000000
0x7fff659ae920: 0x7fff659ae940 0x105dafec9 <bar+25>
0x7fff659ae930: 0x0 0xd00000000
0x7fff659ae940: 0x7fff659ae960 0x105dafec9 <bar+25>
0x7fff659ae950: 0x0 0xc00000000
0x7fff659ae960: 0x7fff659ae980 0x105dafec9 <bar+25>
0x7fff659ae970: 0x0 0xb00000000
0x7fff659ae980: 0x7fff659ae9a0 0x105dafec9 <bar+25>
0x7fff659ae990: 0x0 0xa00000000
0x7fff659ae9a0: 0x7fff659ae9c0 0x105dafec9 <bar+25>
0x7fff659ae9b0: 0x0 0x900000000
0x7fff659ae9c0: 0x7fff659ae9e0 0x105dafec9 <bar+25>
0x7fff659ae9d0: 0x0 0x800000000
0x7fff659ae9e0: 0x7fff659aea00 0x105dafec9 <bar+25>
0x7fff659ae9f0: 0x0 0x700000000
0x7fff659aea00: 0x7fff659aea20 0x105dafec9 <bar+25>
0x7fff659aea10: 0x0 0x600000000
0x7fff659aea20: 0x7fff659aea40 0x105dafec9 <bar+25>
0x7fff659aea30: 0x0 0x5659b9fe0
0x7fff659aea40: 0x7fff659aea60 0x105dafec9 <bar+25>
0x7fff659aea50: 0x7fff659aea70 0x4659bd31f
0x7fff659aea60: 0x7fff659aea80 0x105dafec9 <bar+25>
0x7fff659aea70: 0x7fff659aeaf0 0x3659b031a
0x7fff659aea80: 0x7fff659aeaa0 0x105dafec9 <bar+25>
0x7fff659aea90: 0x7fff659af5c0 0x200000000
0x7fff659aeaa0: 0x7fff659aeac0 0x105dafec9 <bar+25>
0x7fff659aeab0: 0x100000000 0x1659aeb18
0x7fff659aeac0: 0x7fff659aeado 0x105dafecce <foo+14>
0x7fff659aeado: 0x7fff659aeaf0 0x105dafeeb <main+27>
0x7fff659aeae0: 0x7fff659aeb18 0x1
-Type to continue, or q to quit-
0x7fff659aeaf0: 0x7fff659aeb08 0x105dafec94 <start+52>
0x7fff659aeb00: 0x0 0x0
[...]
0x7fff659aeff0: 0x3139336561303363 0x316235

```

Interesting, if we set the lowest frame down and try to get register info GDB core dumps:

```

(gdb) frame 262102
#262102 0x0000000105dafec9 in bar (i=1)
13 bar(i+1);
(gdb) info r
Segmentation fault: 11 (core dumped)

```

Looking at its core dump shows that it also experienced **Stack Overflow**:

```
(gdb) bt
#0 0x000007ffff8c1bacf0 in __sfvwrite ()
#1 0x000007ffff8c189947 in __vfprintf ()
#2 0x000007ffff8c184edb in vsnprintf_l ()
#3 0x000007ffff8c1566be in __sprintf_chk ()
#4 0x0000000010bd14d15 in print_displacement ()
#5 0x0000000010bd10ddf in OP_E ()
#6 0x0000000010bd13f9b in print_insn ()
#7 0x0000000010bc164ce in length_of_this_instruction ()
#8 0x0000000010bc9e296 in x86_analyze_prologue ()
#9 0x0000000010bc9f1f3 in x86_frame_prev_register ()
#10 0x0000000010bc91d70 in frame_register_unwind ()
#11 0x0000000010bc92015 in frame_unwind_register ()
#12 0x0000000010bc91d70 in frame_register_unwind ()
#13 0x0000000010bc92015 in frame_unwind_register ()
#14 0x0000000010bc91d70 in frame_register_unwind ()
#15 0x0000000010bc92015 in frame_unwind_register ()
#16 0x0000000010bc91d70 in frame_register_unwind ()
#17 0x0000000010bc92015 in frame_unwind_register ()
#18 0x0000000010bc91d70 in frame_register_unwind ()
#19 0x0000000010bc92015 in frame_unwind_register ()
#20 0x0000000010bc91d70 in frame_register_unwind ()
#21 0x0000000010bc92015 in frame_unwind_register ()
#22 0x0000000010bc91d70 in frame_register_unwind ()
#23 0x0000000010bc92015 in frame_unwind_register ()
#24 0x0000000010bc91d70 in frame_register_unwind ()
#25 0x0000000010bc92015 in frame_unwind_register ()
#26 0x0000000010bc91d70 in frame_register_unwind ()
#27 0x0000000010bc92015 in frame_unwind_register ()
#28 0x0000000010bc91d70 in frame_register_unwind ()
#29 0x0000000010bc92015 in frame_unwind_register ()
#30 0x0000000010bc91d70 in frame_register_unwind ()
#31 0x0000000010bc92015 in frame_unwind_register ()
[...]
```

The source code for our modeling application:

```
void bar(int i)
{
    bar(i+1);
}

void foo()
{
    bar(1);
}

int main(int argc, const char * argv[])
{
    foo();
    return 0;
}
```

## Windows

### Kernel Mode

Here is an example of **Stack Overflow** pattern in x86 Windows kernel. When it happens in kernel mode we usually have bugcheck 7F with the first argument being EXCEPTION\_DOUBLEFAULT (8):

```
UNEXPECTED_KERNEL_MODE_TRAP (7f)
```

This means a trap occurred in kernel mode, and it's a trap of a kind that the kernel isn't allowed to have/catch (bound trap) or that is always instant death (double fault). The first number in the bugcheck params is the number of the trap (8 = double fault, etc). Consult an Intel x86 family manual to learn more about what these traps are. Here is a \*portion\* of those codes:

If kv shows a taskGate

use .tss on the part before the colon, then kv.

Else if kv shows a trapframe

use .trap on that value

Else

.trap on the appropriate frame will show where the trap was taken (on x86, this will be the ebp that goes with the procedure KiTrap)

Endif

kb will then show the corrected stack.

Arguments:

Arg1: 00000008, EXCEPTION\_DOUBLE\_FAULT

Arg2: f7747fe0

Arg3: 00000000

Arg4: 00000000

The kernel stack size for a thread is limited to 12Kb on x86 platforms (24kb on x64 platforms) and is guarded by an invalid page. Therefore when we hit an invalid address on that page, the processor generates a page fault, tries to push registers and gets a second page fault. This is what "double fault" means. In this scenario, the processor switches to another stack via TSS (task state segment) task switching mechanism because IDT entry for trap 8 contains not an interrupt handler address but the so-called TSS segment selector. This selector points to a memory segment that contains a new kernel stack pointer. The difference between normal IDT entry and double fault entry can be seen by inspecting IDT:

```
5: kd> !pcr 5
KPCR for Processor 5 at f7747000:
    Major 1 Minor 1
    NtTib.ExceptionList: b044e0b8
        NtTib.StackBase: 00000000
        NtTib.StackLimit: 00000000
    NtTib.SubSystemTib: f7747fe0
        NtTib.Version: 00ae1064
    NtTib.UserPointer: 00000020
    NtTib.SelfTib: 7ffdf000
        SelfPcr: f7747000
        Prcb: f7747120
        Irql: 00000000
        IRR: 00000000
        IDR: ffffffff
    InterruptMode: 00000000
```

```

IDT: f774d800
GDT: f774d400
TSS: f774a2e0
CurrentThread: 8834c020
NextThread: 00000000
IdleThread: f774a090

5: kd> dt _KIDTENTRY f774d800
+0x000 Offset : 0x97e8
+0x002 Selector : 8
+0x004 Access : 0x8e00
+0x006 ExtendedOffset : 0x8088

5: kd> ln 0x808897e8
(808897e8) nt!KiTrap00 | (808898c0) nt!Dr_kit1_a
Exact matches:
    nt!KiTrap00

5: kd> dt _KIDTENTRY f774d800+7*8
+0x000 Offset : 0xa880
+0x002 Selector : 8
+0x004 Access : 0x8e00
+0x006 ExtendedOffset : 0x8088

5: kd> ln 8088a880
(8088a880) nt!KiTrap07 | (8088ab72) nt!KiTrap08
Exact matches:
    nt!KiTrap07

5: kd> dt _KIDTENTRY f774d800+8*8
+0x000 Offset : 0x1238
+0x002 Selector : 0x50
+0x004 Access : 0x8500
+0x006 ExtendedOffset : 0

5: kd> dt _KIDTENTRY f774d800+9*8
+0x000 Offset : 0xac94
+0x002 Selector : 8
+0x004 Access : 0x8e00
+0x006 ExtendedOffset : 0x8088

5: kd> ln 8088ac94
(8088ac94) nt!KiTrap09 | (8088ad10) nt!Dr_kita_a
Exact matches:
    nt!KiTrap09

```

If we switch to selector 50 explicitly, we see *nt!KiTrap08* function which does our bugcheck and saves a crash dump in *KeBugCheck2* function:

```

5: kd> .tss 50
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
eip=8088ab72 esp=f774d3c0 ebp=00000000 iopl=0 nv up di pl nz na po nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00000000

```

```
nt!KiTrap08:  
8088ab72 fa           cli  
  
5: kd> .asm no_code_bytes  
Assembly options: no_code_bytes  
  
5: kd> uf nt!KiTrap08  
nt!KiTrap08:  
8088ab72 cli  
8088ab73 mov    eax,dword ptr fs:[00000040h]  
8088ab79 mov    ecx,dword ptr fs:[124h]  
8088ab80 mov    edi,dword ptr [ecx+38h]  
8088ab83 mov    ecx,dword ptr [edi+18h]  
8088ab86 mov    dword ptr [eax+1Ch],ecx  
8088ab89 mov    cx,word ptr [edi+30h]  
8088ab8d mov    word ptr [eax+66h],cx  
8088ab91 mov    ecx,dword ptr [edi+20h]  
8088ab94 test   ecx,ecx  
8088ab96 je     nt!KiTrap08+0x2a (8088ab9c)  
  
nt!KiTrap08+0x26:  
8088ab98 mov    cx,48h  
  
nt!KiTrap08+0x2a:  
8088ab9c mov    word ptr [eax+60h],cx  
8088aba0 mov    ecx,dword ptr fs:[3Ch]  
8088aba7 lea    eax,[ecx+50h]  
8088abaa mov    byte ptr [eax+5],89h  
8088abae pushfd  
8088abaf and   dword ptr [esp],0FFFFBFFFh  
8088abb6 popfd  
8088abb7 mov    eax,dword ptr fs:[0000003Ch]  
8088abbd mov    ch,byte ptr [eax+57h]  
8088abc0 mov    cl,byte ptr [eax+54h]  
8088abc3 shl    ecx,10h  
8088abc6 mov    cx,word ptr [eax+52h]  
8088abca mov    eax,dword ptr fs:[00000040h]  
8088abd0 mov    dword ptr fs:[40h],ecx  
  
nt!KiTrap08+0x65:  
8088abd7 push   0  
8088abd9 push   0  
8088abdb push   0  
8088abdd push   eax  
8088abde push   8  
8088abe0 push   7Fh  
8088abe2 call   nt!KeBugCheck2 (80826a92)  
8088abe7 jmp    nt!KiTrap08+0x65 (8088abd7)
```

We can inspect the TSS address shown in the `!pcr` command output above:

```
5: kd> dt _KTSS f774a2e0
+0x000 Backlink      : 0x28
+0x002 Reserved0    : 0
+0x004 Esp0          : 0xf774d3c0
+0x008 Ss0           : 0x10
+0x00a Reserved1    : 0
+0x00c NotUsed1     : [4] 0
+0x01c CR3           : 0x646000
+0x020 Eip           : 0x8088ab72
+0x024 EFlags        : 0
+0x028 Eax           : 0
+0x02c Ecx           : 0
+0x030 Edx           : 0
+0x034 Ebx           : 0
+0x038 Esp           : 0xf774d3c0
+0x03c Ebp           : 0
+0x040 Esi           : 0
+0x044 Edi           : 0
+0x048 Es            : 0x23
+0x04a Reserved2    : 0
+0x04c Cs             : 8
+0x04e Reserved3    : 0
+0x050 Ss             : 0x10
+0x052 Reserved4    : 0
+0x054 Ds             : 0x23
+0x056 Reserved5    : 0
+0x058 Fs             : 0x30
+0x05a Reserved6    : 0
+0x05c Gs             : 0
+0x05e Reserved7    : 0
+0x060 LDT            : 0
+0x062 Reserved8    : 0
+0x064 Flags          : 0
+0x066 IoMapBase     : 0x20ac
+0x068 IoMaps         : [1] _KiIoAccessMap
+0x208c IntDirectionMap : [32] "???"
```

We see that EIP points to `nt!KiTrap08` and we see that `Backlink` value is 28 which is the previous TSS selector value that was before the double fault trap:

```
5: kd> .tss 28
eax=00000020 ebx=8bef5100 ecx=01404800 edx=8bee4aa8 esi=01404400 edi=00000000
eip=80882e4b esp=b044e000 ebp=b044e034 iopl=0 nv up ei ng nz na po nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010282
nt!_SEH_prolog+0x1b:
80882e4b push    esi
```

```

5: kd> k 100
ChildEBP RetAddr
b044e034 f7b840ac nt!_SEH_prolog+0x1b
b044e054 f7b846e6 Ntfs!NtfsMapStream+0x4b
b044e0c8 f7b84045 Ntfs!NtfsReadMftRecord+0x86
b044e100 f7b840f4 Ntfs!NtfsReadFileRecord+0x7a
b044e138 f7b7cdb5 Ntfs!NtfsLookupInFileRecord+0x37
b044e210 f7b6fefef Ntfs!NtfsWriteFileSizes+0x76
b044e260 f7b6eedad Ntfs!NtfsFlushAndPurgeScb+0xd4
b044e464 f7b7e302 Ntfs!NtfsCommonCleanup+0x1ca8
b044e5d4 8081dce5 Ntfs!NtfsFsdCleanup+0xcf
b044e5e8 f70fac53 nt!IofCallDriver+0x45
b044e610 8081dce5 FltpMgr!FltpDispatch+0x6f
b044e624 f420576a nt!IofCallDriver+0x45
b044e634 f4202621 component2!DispatchEx+0xa4
b044e640 8081dce5 component2!Dispatch+0x53
b044e654 f4e998c7 nt!IofCallDriver+0x45
b044e67c f4e9997c component!PassThrough+0xbb
b044e688 8081dce5 component!Dispatch+0x78
b044e69c f41e72ff nt!IofCallDriver+0x45
WARNING: Stack unwind information not available. Following frames may be wrong.
b044e6c0 f41e71ed driver+0xc2ff
00000000 00000000 driver+0xc1ed

```

This is what **!analyze -v** does for this crash dump:

```
STACK_COMMAND: .tss 0x28 ; kb
```

In our case, NTFS tries to process an exception, and SEH exception handler causes a double fault when trying to save registers on the stack. Let's look at the stack trace and crash point. We see that ESP points to the beginning of the valid stack page but the push decrements ESP before memory access, and the previous page is clearly invalid:

```

TSS: 00000028 -- (.tss 28)
eax=00000020 ebx=8bef5100 ecx=01404800 edx=8bee4aa8 esi=01404400 edi=00000000
eip=80882e4b esp=b044e000 ebp=b044e034 iopl=0 nv up ei ng nz na po nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010282
nt!_SEH_prolog+0x1b:
80882e4b 56          push    esi

5: kd> dd b044e000-4
b044dffc ??????? 8bef5100 00000000 00000000
b044e00c 00000000 00000000 00000000 00000000
b044e01c 00000000 00000000 b044e0b8 80880c80
b044e02c 808b6426 80801300 b044e054 f7b840ac
b044e03c 8bece5e0 b044e064 00000400 00000001
b044e04c b044e134 b044e164 b044e0c8 f7b846e6
b044e05c b044e480 8bee4aa8 01404400 00000000
b044e06c 00000400 b044e134 b044e164 e143db08

```

```
5: kd> !pte b044e000-4
    VA b044dffc
PDE at 00000000C0602C10      PTE at 00000000C0582268
contains 000000010AA3C863  contains 0000000000000000
pfn 10aa3c -DA-KWEV
```

WinDbg was unable to get all stack frames, and we don't see big frame values ("Memory" column below):

```
5: kd> knf 100
*** Stack trace for last set context - .thread/.cxr resets it
#   Memory  ChildEBP RetAddr
00      b044e034 f7b840ac nt!_SEH_prolog+0x1b
01      20 b044e054 f7b846e6 Ntfs!NtfsMapStream+0x4b
02      74 b044e0c8 f7b84045 Ntfs!NtfsReadMftRecord+0x86
03      38 b044e100 f7b840f4 Ntfs!NtfsReadFileRecord+0x7a
04      38 b044e138 f7b7cdb5 Ntfs!NtfsLookupInFileRecord+0x37
05      d8 b044e210 f7b6efef Ntfs!NtfsWriteFileSizes+0x76
06      50 b044e260 f7b6eedd Ntfs!NtfsFlushAndPurgeScb+0xd4
07      204 b044e464 f7b7e302 Ntfs!NtfsCommonCleanup+0x1ca8
08      170 b044e5d4 8081dce5 Ntfs!NtfsFsdCleanup+0xcf
09      14 b044e5e8 f70fac53 nt!IofCallDriver+0x45
0a      28 b044e610 8081dce5 fltMgr!FltpDispatch+0x6f
0b      14 b044e624 f420576a nt!IofCallDriver+0x45
0c      10 b044e634 f4202621 component2!DispatchEx+0xa4
0d      c b044e640 8081dce5 component2!Dispatch+0x53
0e      14 b044e654 f4e998c7 nt!IofCallDriver+0x45
0f      28 b044e67c f4e9997c component!PassThrough+0xbb
10      c b044e688 8081dce5 component!Dispatch+0x78
11      14 b044e69c f41e72ff nt!IofCallDriver+0x45
WARNING: Stack unwind information not available. Following frames may be wrong.
12      24 b044e6c0 f41e71ed driver+0xc2ff
13      00000000 00000000 driver+0xc1ed
```

To see all components involved, we need to dump raw stack data (12Kb is 0x3000). There we can also see some **Software Exceptions** (page 875) processed and get some partial stack traces for them. Some caution is required because stack traces might be incomplete and misleading due to overwritten stack data.

```
5: kd> dds b044e000 b044e000+3000
...
...
...
b044ebc4  b044ec74
b044ebc8  b044ec50
b044ebcc  f41f9458 driver+0x1e458
b044ebd0  b044f140
b044ebd4  b044ef44
b044ebd8  b044f138
b044ebdc  80877290 nt!RtlDispatchException+0x8c
b044ebe0  b044ef44
b044ebe4  b044f138
b044ebe8  b044ec74
b044ebec  b044ec50
b044ebf0  f41f9458 driver+0x1e458
b044ebf4  8a7668c0
```

```
b044ebf8  e16c2e80
b044ebfc  00000000
b044ec00  00000000
b044ec04  00000002
b044ec08  01000000
b044ec0c  00000000
b044ec10  00000000
...
...
...
b044ec60  00000000
b044ec64  b044ef94
b044ec68  8088e13f nt!RtlRaiseStatus+0x47
b044ec6c  b044ef44
b044ec70  b044ec74
b044ec74  00010007
...
...
...
b0450fe8  00000000
b0450fec  00000000
b0450ff0  00000000
b0450ff4  00000000
b0450ff8  00000000
b0450ffc  00000000
b0451000  ????????
5: kd> .exr b044ef44
ExceptionAddress: f41dde6d (driver+0x00002e6d)
    ExceptionCode: c0000043
    ExceptionFlags: 00000001
NumberParameters: 0

5: kd> .cxr b044ec74
eax=c0000043 ebx=00000000 ecx=89fe1bc0 edx=b044f084 esi=e16c2e80 edi=8a7668c0
eip=f41dde6d esp=b044efa0 ebp=b044f010 iopl=0 nv up ei pl zr na pe nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00000246
driver+0x2e6d:
f41dde6d e92f010000      jmp     driver+0x2fa1 (f41ddfa1)
```

```

5: kd> knf
*** Stack trace for last set context - .thread/.cxr resets it
#   Memory  ChildEBP RetAddr
WARNING: Stack unwind information not available. Following frames may be wrong.
00      b044f010 f41ddce6 driver+0x2e6d
01      b0 b044f0c0 f41dd930 driver+0x2ce6
02      38 b044f0f8 f41e88eb driver+0x2930
03      2c b044f124 f6598eba driver+0xd8eb
04      24 b044f148 f41dc40 driver2!AllocData+0x84da
05      18 b044f160 8081dce5 driver+0x1d40
06      14 b044f174 f6596741 nt!IoCallDriver+0x45
07      28 b044f19c f659dd70 driver2!AllocData+0x5d61
08      1c b044f1b8 f65967b9 driver2!EventObjectCreate+0xa60
09      40 b044f1f8 8081dce5 driver2!AllocData+0x5dd9
0a      14 b044f20c 808f8255 nt!IoCallDriver+0x45
0b      e8 b044f2f4 80936af5 nt!IopParseDevice+0xa35
0c      80 b044f374 80932de6 nt!ObpLookupObjectName+0x5a9
0d      54 b044f3c8 808ea211 nt!ObOpenObjectByName+0xea
0e      7c b044f444 808eb4ab nt!IopCreateFile+0x447
0f      5c b044f4a0 808edf2a nt!IoCreateFile+0xa3
10      40 b044f4e0 80888c6c nt!NtCreateFile+0x30
11      0 b044f4e0 8082e105 nt!KiFastCallEntry+0xfc
12      a4 b044f584 f657f20d nt!ZwCreateFile+0x11
13      54 b044f5d8 f65570f6 driver3+0x2e20d

```

Therefore, the following components found on raw stack look suspicious: *driver.sys*, *driver2.sys*, and *driver3.sys*.

We should check their timestamps using **lmv** command and contact their vendors for any existing updates. The workaround would be to remove these products. The rest are Microsoft modules and drivers *component.sys* and *component2.sys*. For the latter two, we don't have significant local variable usage in their functions.

OSR NT Insider article provides another example<sup>191</sup>.

---

<sup>191</sup> <http://www.osronline.com/article.cfm?article=254>

## Comments

There is another example from NT Debugging blog<sup>192</sup>.

Another example:

```
1: kd> !analyze -v

UNEXPECTED_KERNEL_MODE_TRAP (7f)
This means a trap occurred in kernel mode, and it's a trap of a kind
that the kernel isn't allowed to have/catch (bound trap) or that
is always instant death (double fault). The first number in the
bugcheck params is the number of the trap (8 = double fault, etc)
Consult an Intel x86 family manual to learn more about what these
traps are. Here is a *portion* of those codes:
If kv shows a taskGate
use .tss on the part before the colon, then kv.
Else if kv shows a trapframe
use .trap on that value
Else
.trap on the appropriate frame will show where the trap was taken
(on x86, this will be the ebp that goes with the procedure KiTrap)
Endif
kb will then show the corrected stack.
Arguments:
Arg1: 00000008, EXCEPTION_DOUBLE_FAULT
Arg2: f7727fe0
Arg3: 00000000
Arg4: 00000000
```

Debugging Details:

WARNING: Process directory table base AFFB7740 doesn't match CR3 00545000  
Unable to get PEB pointer

WARNING: Process directory table base AFFB7740 doesn't match CR3 00545000  
Unable to get PEB pointer

BUGCHECK\_STR: 0x7f\_8

---

<sup>192</sup> <http://blogs.msdn.com/ntdebugging/archive/2008/02/01/kernel-stack-overflows.aspx>

```

TSS: 00000028 - (.tss 0x28)
eax=00000000 ebx=f78dd100 ecx=a53a5b40 edx=007ffff8 esi=80000000 edi=c0603018
eip=8085e1d0 esp=f78dcfb8 ebp=f78dd008 iopl=0 nv up ei ng nz na pe nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 ef1=00010286
nt!MmAccessFault+0x8:
8085e1d0 and dword ptr [ebp-1Ch],0 ss:0010:f78dcfec=00000000
Resetting default scope

DEFAULT_BUCKET_ID: DRIVER_FAULT

PROCESS_NAME: drwtsn32.exe

CURRENT_IRQL: 2

TRAP_FRAME: f78dd308 - (.trap 0xfffffffff78dd308)
ErrCode = 00000000
eax=40000000 ebx=c0400000 ecx=c0603018 edx=007ffff8 esi=80000000 edi=00000001
eip=8084d509 esp=f78dd37c ebp=f78dd3e8 iopl=0 nv up ei pl nz na po cy
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 ef1=00010203
nt!MiCheckPdeForPagedPool+0x73:
8084d509 mov eax,dword ptr [ecx] ds:0023:c0603018=00549063
Resetting default scope

LAST_CONTROL_TRANSFER: from 8088c798 to 8085e1d0

STACK_TEXT:
f78dd008 8088c798 00000000 c0603018 00000000 nt!MmAccessFault+0x8

```

There were a few questions asked:

**Q.** According to the result of: dds b044e000 b044e000+3000. May I know why did you choose “b044ef44” & “b044ec74” to inspect? Do we need to choose the value of “*RtlRaiseStatus+0x47*” after? How about if I can’t find “*RtlRaiseStatus+0x47*”?

**A.** We chose b044e000 because we wanted to inspect the raw stack data from the top of the stack (ESP value in bold above). By choosing other ranges, we can miss something.

**Q.** I have the same question. May I know why to choose “b044ef44” & “b044ec74” to inspect?

**A.** Certain functions like *RtlDispatchException*, *RtlRaiseStatus*, and others have function parameters pointing to an exception record and context. See, for example, this source code file<sup>193</sup>.

---

<sup>193</sup> <http://source.winehq.org/source/dlls/kernel32/except.c#L84>

## Software Implementation

**Stack Overflow** pattern variants in user (page 912) and kernel mode (page 900) are ISA (Instruction Set Architecture) and processor architecture oriented. Another pattern variant is a software stack implementation where push and pop operations check a stack ADT precondition and throw **Software Exception** (page 875, overflow or underflow) or call an assertion mechanism to display an error message. For the latter example, we look at a bugcheck for the specific stack implementation on Windows: IRP stack locations array. For a graphical reminder on how driver-to-driver communication is implemented by an IRP stack corresponding to a driver stack, please refer to a UML diagram<sup>194</sup>. The following WinDbg command output is from a kernel memory dump:

```
0: kd> !analyze -v
[...]
NO_MORE_IRP_STACK_LOCATIONS (35)
A higher level driver has attempted to call a lower level driver through the IoCallDriver() interface,
but there are no more stack locations in the packet, hence, the lower level driver would not be able to
access its parameters, as there are no parameters for it. This is a disastrous situation, since the
higher level driver "thinks" it has filled in the parameters for the lower level driver (something it
MUST do before it calls it), but since there is no stack location for the latter driver, the former has
written off of the end of the packet. This means that some other memory has probably been trashed at this
point.
Arguments:
Arg1: ffffffa800500c9e0, Address of the IRP
Arg2: 0000000000000000
Arg3: 0000000000000000
Arg4: 0000000000000000
[...]

0: kd> kL 100
Child-SP RetAddr Call Site
fffff880`01fe2338 fffff800`016d7732 nt!KeBugCheckEx
fffff880`01fe2340 fffff800`01754f27 nt!KiBugCheck3+0x12
fffff880`01fe2370 fffff880`0177e271 nt! ?? ::FNODOBFM::`string'+0x3f31b
fffff880`01fe23a0 fffff880`0177c138 DriverA!CallProvider+0x161
[...]
fffff880`01fe2cb0 fffff800`0197a7c6 nt!ExpWorkerThread+0x111
fffff880`01fe2d40 fffff800`016b5c26 nt!PspSystemThreadStartup+0x5a
fffff880`01fe2d80 00000000`00000000 nt!KxStartSystemThread+0x16

0: kd> !irp ffffffa800500c9e0
Irp is active with 1 stacks 0 is current (= 0xfffffa8006c2e960)
No Mdl: No System Buffer: Thread 00000000: Irp stack trace.
cmd flg cl Device File Completion-Context
[ 4, 0] 0 e0 ffffffa8004045c50 ffffffa8006c2e960 fffff88005a04460-fffffa8005b9c370 Success Error Cancel
\DriverA DriverB!CompleteRoutine
Args: 00000008 00000000 00000000 00000000
```

<sup>194</sup> UML and Device Drivers, Memory Dump Analysis Anthology, Volume 1, page 701

```
0: kd> ub fffff880`0177e271
DriverA!CallProvider+0x13e:
fffff880`0177e24e mov qword ptr [r11-10h],rax
fffff880`0177e252 mov qword ptr [r11-8],r12
fffff880`0177e256 mov byte ptr [r11-45h],0E0h
fffff880`0177e25b mov rcx,qword ptr [rdi+40h]
fffff880`0177e25f call qword ptr [DriverA!_imp_IoGetAttachedDevice (fffff880`017790b0)]
fffff880`0177e265 mov rdx,rbp
fffff880`0177e268 mov rcx,rax
fffff880`0177e26b call qword ptr [DriverA!_imp_IofCallDriver (fffff880`01779068)]
```

## User Mode

**Stack Overflow** pattern in user mode has its own characteristic exception code and stack trace:

```
FAULTING_IP:
StackOverflow!SoFunction+27
00401317 6a00      push    0

EXCEPTION_RECORD: ffffffff -- (.exr 0xffffffffffff)
ExceptionAddress: 00401300 (StackOverflow!SoFunction+0x00000010)
ExceptionCode: c00000fd (Stack overflow)
ExceptionFlags: 00000000
NumberParameters: 2
Parameter[0]: 00000001
Parameter[1]: 00082ffc

0:000> kL
ChildEBP RetAddr
00083000 00401317 StackOverflow!SoFunction+0x10
00083010 00401317 StackOverflow!SoFunction+0x27
00083020 00401317 StackOverflow!SoFunction+0x27
00083030 00401317 StackOverflow!SoFunction+0x27
00083040 00401317 StackOverflow!SoFunction+0x27
00083050 00401317 StackOverflow!SoFunction+0x27
00083060 00401317 StackOverflow!SoFunction+0x27
00083070 00401317 StackOverflow!SoFunction+0x27
00083080 00401317 StackOverflow!SoFunction+0x27
00083090 00401317 StackOverflow!SoFunction+0x27
000830a0 00401317 StackOverflow!SoFunction+0x27
000830b0 00401317 StackOverflow!SoFunction+0x27
000830c0 00401317 StackOverflow!SoFunction+0x27
000830d0 00401317 StackOverflow!SoFunction+0x27
000830e0 00401317 StackOverflow!SoFunction+0x27
000830f0 00401317 StackOverflow!SoFunction+0x27
00083100 00401317 StackOverflow!SoFunction+0x27
00083110 00401317 StackOverflow!SoFunction+0x27
00083120 00401317 StackOverflow!SoFunction+0x27
00083130 00401317 StackOverflow!SoFunction+0x27
```

There could be thousands of stack frames:

```
0:000> kL 2000
...
000a2fa0 00401317 StackOverflow!SoFunction+0x27
000a2fb0 00401317 StackOverflow!SoFunction+0x27
000a2fc0 00401317 StackOverflow!SoFunction+0x27
000a2fd0 00401317 StackOverflow!SoFunction+0x27
000a2fe0 00401317 StackOverflow!SoFunction+0x27
000a2ff0 00401317 StackOverflow!SoFunction+0x27
```

To reach the bottom and avoid overscrolling, we can dump the raw stack data, search for the end of the repeating pattern of *StackOverflow!SoFunction+0x27* and try to manually reconstruct the bottom of the stack trace:

```
0:000> !teb
TEB at 7efdd000
ExceptionList: 0017fdf0
StackBase: 00180000
StackLimit: 00081000
SubSystemTib: 00000000
FiberData: 00001e00
ArbitraryUserPointer: 00000000
Self: 7efdd000
EnvironmentPointer: 00000000
ClientId: 00001dc4 . 00001b74
RpcHandle: 00000000
Tls Storage: 7efdd02c
PEB Address: 7efde000
LastErrorValue: 0
LastStatusValue: c0000034
Count Owned Locks: 0
HardErrorMode: 0

0:000> dds 00081000 00180000
...
0017fc74 00401317 StackOverflow!SoFunction+0x27
0017fc78 00000000
0017fc7c a3a8ea65
0017fc80 0017fc90
0017fc84 00401317 StackOverflow!SoFunction+0x27
0017fc88 10001843
0017fc8c a3a8ea95
0017fc90 0017fca0
0017fc94 00401317 StackOverflow!SoFunction+0x27
0017fc98 0017fcbb
0017fc9c a3a8ea85
0017fca0 0017fcb0
0017fca4 00401317 StackOverflow!SoFunction+0x27
0017fca8 00000003
0017fcac a3a8eab5
0017fcb0 0017fcc0
0017fcb4 00401317 StackOverflow!SoFunction+0x27
0017fcb8 76c68738 user32!_EndUserApiHook+0x11
0017fcbe a3a8eaa5
0017fcc0 0017fcdb
0017fcc4 00401317 StackOverflow!SoFunction+0x27
0017fcc8 76c6a6cc user32!DefWindowProcW+0x94
0017fccc a3a8ead5
0017fcdb 0017fce0
0017fcdb4 00401317 StackOverflow!SoFunction+0x27
0017fcdb8 0037311e
0017fcdbd a3a8eac5
0017fce0 0017fcf0
0017fce4 00401317 StackOverflow!SoFunction+0x27
0017fce8 0017fcdb
0017fcdb4 a3a8eaf5
```

```

0017fcf0 0017fd00
0017fcf4 00401317 StackOverflow!SoFunction+0x27
0017fcf8 76c6ad0f user32!NtUserBeginPaint+0x15
0017fcfc a3a8ea5
0017fd00 0017fd5c
0017fd04 00401272 StackOverflow!WndProc+0xe2
0017fd08 00401190 StackOverflow!WndProc
0017fd0c 00000003
0017fd10 cf017ada
...

```

We use the extended version of **k** WinDbg command and supply EBP, ESP, and EIP to see the function it started from:

```

0:000> r
eax=a3b739e5 ebx=00000000 ecx=ac430000 edx=ffefd944 esi=0037311e edi=00000000
eip=00401300 esp=00082ff8 ebp=00083000 iopl=0 nv up ei ng nz na po nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010282
StackOverflow!SoFunction+0x10:
00401300 89442404 mov     dword ptr [esp+4],eax ss:002b:00082ffc=00000000

0:000> k L=0017fcf0 00082ff8 00401300
ChildEBP RetAddr
0017fcf0 00401317 StackOverflow!SoFunction+0x10
0017fd00 00401272 StackOverflow!SoFunction+0x27
0017fd5c 76c687af StackOverflow!WndProc+0xe2
0017fd88 76c68936 user32!InternalCallWinProc+0x23
0017fe00 76c6a571 user32!UserCallWinProcCheckWow+0x109
0017fe5c 76c6a5dd user32!DispatchClientMessage+0xe0
0017fe98 77ccee2e user32!__fnDWORD+0x2b
0017fedc 0040107d ntdll!KiUserCallbackDispatcher+0x2e
0017ff08 0040151e StackOverflow!wWinMain+0x7d
00402ba0 20245c8b StackOverflow!_tmainCRTStartup+0x176

```

We see that it started in WndProc.

## Comments

One of the readers sent us this program and corresponding WinDbg output:

```
void func1(void);

void __cdecl main(void)
{
    func1();
}

void func1(void)
{
    func1();
}

0:000> .lastevent
Last event: d0c.e94: Stack overflow - code c00000fd (first chance)

0:000> k L50
ChildEBP RetAddr
00103000 002a18e8 test!func1+0x3
00103008 002a18e8 test!func1+0x8
00103010 002a18e8 test!func1+0x8
00103018 002a18e8 test!func1+0x8
...
0:000> !teb
TEB at 7ffd़f000
ExceptionList: 001ffa1c
StackBase: 00200000
StackLimit: 00101000
...
0:000>dds 101000 200000
...[ebp][ret addr]... mainframe...
001ff9d4 002a18e8 test!func1+0x8
001ff9d8 001ff9e0
001ff9dc 002a18e8 test!func1+0x8
001ff9e0 001ff9e8
001ff9e4 002a18f8 test!main+0x8
001ff9e8 001ffa2c
001ff9ec 002a1174 test!__tmainCRTStartup+0x122
001ff9f0 00000001
001ff9f4 00651388
001ff9f8 00651928
001ff9fc bc2792b1
001ffa00 00000000
001ffa04 00000000
001ffa08 7ffd़c000
001ffa0c 00000000
001ffa10 00000000
001ffa14 001ff9fc
```

```

001ffa18 b33a09b6
001ffa1c 001ffa68
001ffa20 002a1619 test!_except_handler4
001ffa24 bc124925
001ffa28 00000000
001ffa2c 001ffa38
001ffa30 75911194 kernel32!BaseThreadInitThunk+0xe
001ffa34 7ffdc000
001ffa38 001ffa78
001ffa3c 7747b495 ntdll!__RtlUserThreadStart+0x70
001ffa40 7ffdc000
001ffa44 774f7154 ntdll!RtlpSecMemListHead
001ffa48 00000000
001ffa4c 00000000
001ffa50 7ffdc000
001ffa54 00000000
001ffa58 00000000
001ffa5c 00000000
001ffa60 001ffa44
001ffa64 00000000
001ffa68 ffffffff
001ffa6c 7743d75d ntdll!_except_handler4
001ffa70 00178d24
001ffa74 00000000
001ffa78 001ffa90
001ffa7c 7747b468 ntdll!_RtlUserThreadStart+0x1b
001ffa80 002a12dc test!mainCRTStartup
001ffa84 7ffdc000
001ffa88 00000000
001ffa8c 00000000
001ffa90 00000000
001ffa94 00000000
001ffa98 002a12dc test!mainCRTStartup
...
0:000> r
eax=00651928 ebx=00000000 ecx=6ca33714 edx=00000000 esi=00000001 edi=002a3378
eip=002a18e3 esp=00103000 ebp=00103000 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 ef1=00010246
test!func1+0x3:
002a18e3 e8f8ffff call test!func1 (002a18e0)

0:000> k 1ff9d8 103000 2a18e3
Requested number of stack frames (0x1ff9d8) is too large! The maximum number is 0xffff.
^ Range error in 'k 1ff9d8 103000 2a18e3'

0:000> k L=1ff9d8 103000 2a18e3
ChildEBP RetAddr
001ff9d8 002a18e8 test!func1+0x3
001ff9e0 002a18f8 test!func1+0x8
001ff9e8 002a1174 test!main+0x8
001ffa2c 75911194 test!__tmainCRTStartup+0x122
001ffa38 7747b495 kernel32!BaseThreadInitThunk+0xe
001ffa78 7747b468 ntdll!__RtlUserThreadStart+0x70
001ffa90 00000000 ntdll!_RtlUserThreadStart+0x1b

```

## Stack Trace

### Linux

This is a Linux variant of **Stack Trace** pattern previously described for Mac OS X (page 918) and Windows (page 926) platforms. Here we show a stack trace when debug symbols are not available (stripped executable) and also how to apply debug symbols from the executable where they were preserved:

```
(gdb) bt
#0 0x000000000043e4f1 in nanosleep ()
#1 0x000000000043e3c0 in sleep ()
#2 0x0000000000400789 in main ()

(gdb) symbol-file ./App/App.debug
Reading symbols from /home/Apps/App/App.debug...done.

(gdb) bt
#0 0x000000000043e4f1 in nanosleep ()
#1 0x000000000043e3c0 in sleep ()
#2 0x0000000000400789 in main (argc=1, argv=0x7fff5d1572d8) at main.cpp:85
```

## Mac OS X

This is a Mac OS X / GDB counterpart to **Stack Trace** pattern. Here we first show a stack trace when symbols are not available and then show how to apply symbols:

```
(gdb) bt
#0 0x000000010d3b0e90 in ?? ()
#1 0x000000010d3b0ea9 in ?? ()
#2 0x000000010d3b0ec4 in ?? ()
#3 0x000000010d3b0e74 in ?? ()

(gdb) maintenance info sections
Exec file:
[...]
Core file:
`/cores/core.262', file type mach-o-le.
0x000000010d3b0000->0x000000010d3b1000 at 0x00001000: LC_SEGMENT. ALLOC LOAD CODE HAS_CONTENTS
0x000000010d3b1000->0x000000010d3b2000 at 0x00002000: LC_SEGMENT. ALLOC LOAD CODE HAS_CONTENTS
0x000000010d3b2000->0x000000010d3b3000 at 0x00003000: LC_SEGMENT. ALLOC LOAD CODE HAS_CONTENTS
0x000000010d3b3000->0x000000010d3b4000 at 0x00004000: LC_SEGMENT. ALLOC LOAD CODE HAS_CONTENTS
0x000000010d3b4000->0x000000010d3b5000 at 0x00005000: LC_SEGMENT. ALLOC LOAD CODE HAS_CONTENTS
0x000000010d3b5000->0x000000010d3b6000 at 0x00006000: LC_SEGMENT. ALLOC LOAD CODE HAS_CONTENTS
0x000000010d3b6000->0x000000010d3cb000 at 0x00007000: LC_SEGMENT. ALLOC LOAD CODE HAS_CONTENTS
0x000000010d3cb000->0x000000010d3cc000 at 0x0001c000: LC_SEGMENT. ALLOC LOAD CODE HAS_CONTENTS
0x000000010d3cc000->0x000000010d3cd000 at 0x0001d000: LC_SEGMENT. ALLOC LOAD CODE HAS_CONTENTS
0x000000010d3cd000->0x000000010d3e2000 at 0x0001e000: LC_SEGMENT. ALLOC LOAD CODE HAS_CONTENTS
0x000000010d3e2000->0x000000010d3e3000 at 0x00033000: LC_SEGMENT. ALLOC LOAD CODE HAS_CONTENTS
0x000000010d3e3000->0x000000010d3e4000 at 0x00034000: LC_SEGMENT. ALLOC LOAD CODE HAS_CONTENTS
0x000000010d400000->0x000000010d500000 at 0x00035000: LC_SEGMENT. ALLOC LOAD CODE HAS_CONTENTS
[...]

(gdb) add-symbol-file ~/Documents/Work/Test.sym 0x000000010d3b0000
add symbol table from file "/Users/DumpAnalysis/Documents/Work/Test.sym" at
LC_SEGMENT.__TEXT = 0x10d3b0000
(y or n) y
Reading symbols from /Users/DumpAnalysis/Documents/Work/Test.sym...done.

(gdb) bt
#0 0x000000010d3b0e90 in bar () at main.c:15
#1 0x000000010d3b0ea9 in foo () at main.c:20
#2 0x000000010d3b0ec4 in main (argc=1,
argv=0x7fff6cfafbf8) at main.c:25
```

## Windows

### Database

Some troubleshooting and debugging techniques involve saving in some region in memory, called stack trace database, every **Stack Trace** (page 926) that leads to a specific action such as a memory allocation of the opening of a resource handle. Typical pattern usage examples include process heap **Memory Leak** (page 650), **Insufficient Memory** (page 526) due to **Handle Leak** (page 416). A typical entry in such a database consists of return addresses saved during function calls (that may be **Truncated Stack Trace**, page 1015):

```
00000000`00325da0 000007fe`fd5e37aa KERNELBASE!InitializeCriticalSectionAndSpinCount+0xa
00000000`00325da8 00000001`3fd72239 AllocFree!_ioinit+0x2cd
00000000`00325db0 00000001`3fd71115 AllocFree!__tmainCRTStartup+0xc5
00000000`00325db8 00000000`773759ed kernel32!BaseThreadInitThunk+0xd
00000000`00325dc0 00000000`774ac541 ntdll!RtlUserThreadStart+0x1d

0:001> ub 00000001`3fd72239
AllocFree!_ioinit+0x2af:
00000001`3fd7221b cmp    eax,3
00000001`3fd7221e jne    AllocFree!_ioinit+0x2be (00000001`3fd7222a)
00000001`3fd72220 movsx  eax,byte ptr [rbx+8]
00000001`3fd72224 or     eax,8
00000001`3fd72227 mov    byte ptr [rbx+8],al
00000001`3fd7222a lea    rcx,[rbx+10h]
00000001`3fd7222e mov    edx,0FA0h
00000001`3fd72233 call   qword ptr [AllocFree!_imp_InitializeCriticalSectionAndSpinCount
(00000001`3fd78090)
```

This slightly differs from ‘k’-style stack trace format where the return address belongs to the function on the next line if moving downwards:

```
0:000> k
Child-SP RetAddr Call Site
00000000`002ff9f8 000007fe`fd5e1203 ntdll!ZwDelayExecution+0xa
00000000`002ffa00 00000001`3fd71018 KERNELBASE!SleepEx+0xab
00000000`002ffaa0 00000001`3fd71194 AllocFree!wmain+0x18
00000000`002ffad0 00000000`773759ed AllocFree!__tmainCRTStartup+0x144
00000000`002ffb10 00000000`774ac541 kernel32!BaseThreadInitThunk+0xd
00000000`002ffb40 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

0:000> ub 00000001`3fd71194
AllocFree!__tmainCRTStartup+0x11b:
00000001`3fd7116b je    AllocFree!__tmainCRTStartup+0x124 (00000001`3fd71174)
00000001`3fd7116d mov   ecx, eax
00000001`3fd7116f call  AllocFree!_amsg_exit (00000001`3fd718ec)
00000001`3fd71174 mov   r8,qword ptr [AllocFree!_wenviron (00000001`3fd80868)]
00000001`3fd7117b mov   qword ptr [AllocFree!__winitenv (00000001`3fd80890)],r8
00000001`3fd71182 mov   rdx,qword ptr [AllocFree!__wargv (00000001`3fd80858)]
00000001`3fd71189 mov   ecx,dword ptr [AllocFree!__argc (00000001`3fd8084c)]
00000001`3fd7118f call  AllocFree!wmain (00000001`3fd71000)
```

Sometimes we can see such traces as **Execution Residue** (page 371) inside a stack or some other region. If user mode stack trace database is enabled in *gflags.exe* we might be able to dump the specific database region:

```
0:001> !gflag
Current NtGlobalFlag contents: 0x00001000
ust - Create user mode stack trace database

0:001> !address
[...]
BaseAddress EndAddress+1 RegionSize Type State Protect Usage
-----
-----
[...]
+ 0`00300000 0`00326000 0`00026000 MEM_PRIVATE MEM_COMMIT PAGE_READWRITE Other [Stack Trace Database]
 0`00326000 0`01aff000 0`017d9000 MEM_PRIVATE MEM_RESERVE Other [Stack Trace Database]
 0`01aff000 0`01b00000 0`00001000 MEM_PRIVATE MEM_COMMIT PAGE_READWRITE Other [Stack Trace Database]
[...]

0:001> dps 0`00326000-1000 0`00326000
[...]
00000000`003257e0 00000000`00000000
00000000`003257e8 00030001`00001801
00000000`003257f0 00000000`774c34eb ntdll!LdrpInitializeProcess+0x7e6
00000000`003257f8 00000000`774c1937 ntdll! ?? ::FNODOBFM::`string'+0x28ff0
00000000`00325800 00000000`774ac34e ntdll!LdrInitializeThunk+0xe
00000000`00325808 00000000`00000000
00000000`00325810 00000000`00000000
00000000`00325818 00030002`00001801
00000000`00325820 00000000`774c3511 ntdll!LdrpInitializeProcess+0x80c
00000000`00325828 00000000`774c1937 ntdll! ?? ::FNODOBFM::`string'+0x28ff0
00000000`00325830 00000000`774ac34e ntdll!LdrInitializeThunk+0xe
00000000`00325838 00000000`00000000
00000000`00325840 00000000`00000000
00000000`00325848 00040003`00001801
00000000`00325850 00000000`774bda86 ntdll!RtlCreateHeap+0x506
00000000`00325858 00000000`774c3557 ntdll!LdrpInitializeProcess+0x851
00000000`00325860 00000000`774c1937 ntdll! ?? ::FNODOBFM::`string'+0x28ff0
00000000`00325868 00000000`774ac34e ntdll!LdrInitializeThunk+0xe
00000000`00325870 00000000`00000000
00000000`00325878 00050004`00002801
00000000`00325880 00000000`7751998a ntdll! ?? ::FNODOBFM::`string'+0xdc1a
00000000`00325888 00000000`774bdaee ntdll!RtlCreateHeap+0x56e
00000000`00325890 00000000`774c3557 ntdll!LdrpInitializeProcess+0x851
00000000`00325898 00000000`774c1937 ntdll! ?? ::FNODOBFM::`string'+0x28ff0
00000000`003258a0 00000000`774ac34e ntdll!LdrInitializeThunk+0xe
00000000`003258a8 00000000`00000000
00000000`003258b0 00000000`00000000
00000000`003258b8 00030005`00001801
00000000`003258c0 00000000`774c359e ntdll!LdrpInitializeProcess+0x902
00000000`003258c8 00000000`774c1937 ntdll! ?? ::FNODOBFM::`string'+0x28ff0
00000000`003258d0 00000000`774ac34e ntdll!LdrInitializeThunk+0xe
00000000`003258d8 00000000`00000000
00000000`003258e0 00000000`00000000
00000000`003258e8 00030006`00001801
00000000`003258f0 00000000`774c35af ntdll!LdrpInitializeProcess+0x913
```

```
00000000`003258f8 00000000`774c1937 ntdll! ?? ::FNODOBFM::`string'+0x28ff0
00000000`00325900 00000000`774ac34e ntdll!LdrInitializeThunk+0xe
00000000`00325908 00000000`00000000
00000000`00325910 00000000`00000000
00000000`00325918 00090007`00004801
00000000`00325920 00000000`774bda86 ntdll!RtlCreateHeap+0x506
00000000`00325928 00000000`774c47ff ntdll!CsrpConnectToServer+0x41f
00000000`00325930 00000000`774c43c5 ntdll!CsrClientConnectToServer+0x230
00000000`00325938 000007fe`fd5ee232 KERNELBASE!KernelBaseDllInitialize+0x148
00000000`00325940 00000000`774bb108 ntdll!LdrpRunInitializeRoutines+0x1fe
00000000`00325948 00000000`774c42fd ntdll!LdrGetProcedureAddressEx+0x2aa
00000000`00325950 00000000`774c1ddc ntdll!LdrpInitializeProcess+0x1a0b
00000000`00325958 00000000`774c1937 ntdll! ?? ::FNODOBFM::`string'+0x28ff0
00000000`00325960 00000000`774ac34e ntdll!LdrInitializeThunk+0xe
00000000`00325968 00000000`00000000
00000000`00325970 00000000`00000000
00000000`00325978 000a0008`00004801
00000000`00325980 00000000`7751998a ntdll! ?? ::FNODOBFM::`string'+0xdc1a
00000000`00325988 00000000`774bdaee ntdll!RtlCreateHeap+0x56e
00000000`00325990 00000000`774c47ff ntdll!CsrpConnectToServer+0x41f
00000000`00325998 00000000`774c43c5 ntdll!CsrClientConnectToServer+0x230
00000000`003259a0 000007fe`fd5ee232 KERNELBASE!KernelBaseDllInitialize+0x148
00000000`003259a8 00000000`774bb108 ntdll!LdrpRunInitializeRoutines+0x1fe
00000000`003259b0 00000000`774c42fd ntdll!LdrGetProcedureAddressEx+0x2aa
00000000`003259b8 00000000`774c1ddc ntdll!LdrpInitializeProcess+0x1a0b
00000000`003259c0 00000000`774c1937 ntdll! ?? ::FNODOBFM::`string'+0x28ff0
00000000`003259c8 00000000`774ac34e ntdll!LdrInitializeThunk+0xe
00000000`003259d0 00000000`00000000
00000000`003259d8 00080009`00003801
00000000`003259e0 000007fe`fd5edf81 KERNELBASE!NlsProcessInitialize+0x11
00000000`003259e8 000007fe`fd604439 KERNELBASE!BaseNlsDllInitialize+0x29
00000000`003259f0 000007fe`fd5ee446 KERNELBASE!KernelBaseDllInitialize+0x40c
00000000`003259f8 00000000`774bb108 ntdll!LdrpRunInitializeRoutines+0x1fe
00000000`00325a00 00000000`774c42fd ntdll!LdrGetProcedureAddressEx+0x2aa
00000000`00325a08 00000000`774c1ddc ntdll!LdrpInitializeProcess+0x1a0b
00000000`00325a10 00000000`774c1937 ntdll! ?? ::FNODOBFM::`string'+0x28ff0
00000000`00325a18 00000000`774ac34e ntdll!LdrInitializeThunk+0xe
00000000`00325a20 00000000`00000000
00000000`00325a28 0008000a`00003801
00000000`00325a30 000007fe`fd5edfa0 KERNELBASE!NlsProcessInitialize+0x30
00000000`00325a38 000007fe`fd604439 KERNELBASE!BaseNlsDllInitialize+0x29
00000000`00325a40 000007fe`fd5ee446 KERNELBASE!KernelBaseDllInitialize+0x40c
00000000`00325a48 00000000`774bb108 ntdll!LdrpRunInitializeRoutines+0x1fe
00000000`00325a50 00000000`774c42fd ntdll!LdrGetProcedureAddressEx+0x2aa
00000000`00325a58 00000000`774c1ddc ntdll!LdrpInitializeProcess+0x1a0b
00000000`00325a60 00000000`774c1937 ntdll! ?? ::FNODOBFM::`string'+0x28ff0
00000000`00325a68 00000000`774ac34e ntdll!LdrInitializeThunk+0xe
00000000`00325a70 00000000`00000000
00000000`00325a78 0007000b`00003801
00000000`00325a80 000007fe`fd604a21 KERNELBASE!BasepInitComputerNameCache+0x11
00000000`00325a88 000007fe`fd603d20 KERNELBASE!KernelBaseDllInitialize+0x419
00000000`00325a90 00000000`774bb108 ntdll!LdrpRunInitializeRoutines+0x1fe
00000000`00325a98 00000000`774c42fd ntdll!LdrGetProcedureAddressEx+0x2aa
00000000`00325aa0 00000000`774c1ddc ntdll!LdrpInitializeProcess+0x1a0b
00000000`00325aa8 00000000`774c1937 ntdll! ?? ::FNODOBFM::`string'+0x28ff0
00000000`00325ab0 00000000`774ac34e ntdll!LdrInitializeThunk+0xe
```

```
00000000`00325ab8 00000000`00000000
00000000`00325ac0 00000000`00000000
00000000`00325ac8 0006000c`00002801
00000000`00325ad0 00000000`77375699 kernel32!BaseDllInitialize+0x2f9
00000000`00325ad8 00000000`774bb108 ntdll!LdrpRunInitializeRoutines+0x1fe
00000000`00325ae0 00000000`774c42fd ntdll!LdrGetProcedureAddressEx+0x2aa
00000000`00325ae8 00000000`774c1ddc ntdll!LdrpInitializeProcess+0x1a0b
00000000`00325af0 00000000`774c1937 ntdll! ?? ::FNODOBFM::`string'+0x28ff0
00000000`00325af8 00000000`774ac34e ntdll!LdrInitializeThunk+0xe
00000000`00325b00 00000000`00000000
00000000`00325b08 0007000d`00003801
00000000`00325b10 00000000`773771f7 kernel32!InitializeConsoleConnectionInfo+0xe7
00000000`00325b18 00000000`773756ae kernel32!BaseDllInitialize+0x30e
00000000`00325b20 00000000`774bb108 ntdll!LdrpRunInitializeRoutines+0x1fe
00000000`00325b28 00000000`774c42fd ntdll!LdrGetProcedureAddressEx+0x2aa
00000000`00325b30 00000000`774c1ddc ntdll!LdrpInitializeProcess+0x1a0b
00000000`00325b38 00000000`774c1937 ntdll! ?? ::FNODOBFM::`string'+0x28ff0
00000000`00325b40 00000000`774ac34e ntdll!LdrInitializeThunk+0xe
00000000`00325b48 00000000`00000000
00000000`00325b50 00000000`00000000
00000000`00325b58 0009000e`00004801
00000000`00325b60 00000000`774bda86 ntdll!RtlCreateHeap+0x506
00000000`00325b68 00000000`773787f7 kernel32!ConsoleConnect+0x1d7
00000000`00325b70 00000000`773770de kernel32!ConnectConsoleInternal+0x147
00000000`00325b78 00000000`773756fe kernel32!BaseDllInitialize+0x35e
00000000`00325b80 00000000`774bb108 ntdll!LdrpRunInitializeRoutines+0x1fe
00000000`00325b88 00000000`774c42fd ntdll!LdrGetProcedureAddressEx+0x2aa
00000000`00325b90 00000000`774c1ddc ntdll!LdrpInitializeProcess+0x1a0b
00000000`00325b98 00000000`774c1937 ntdll! ?? ::FNODOBFM::`string'+0x28ff0
00000000`00325ba0 00000000`774ac34e ntdll!LdrInitializeThunk+0xe
00000000`00325ba8 00000000`00000000
00000000`00325bb0 00000000`00000000
00000000`00325bb8 000a000f`00004801
00000000`00325bc0 00000000`7751998a ntdll! ?? ::FNODOBFM::`string'+0xdc1a
00000000`00325bc8 00000000`774bdae ntdll!RtlCreateHeap+0x56e
00000000`00325bd0 00000000`773787f7 kernel32!ConsoleConnect+0x1d7
00000000`00325bd8 00000000`773770de kernel32!ConnectConsoleInternal+0x147
00000000`00325be0 00000000`773756fe kernel32!BaseDllInitialize+0x35e
00000000`00325be8 00000000`774bb108 ntdll!LdrpRunInitializeRoutines+0x1fe
00000000`00325bf0 00000000`774c42fd ntdll!LdrGetProcedureAddressEx+0x2aa
00000000`00325bf8 00000000`774c1ddc ntdll!LdrpInitializeProcess+0x1a0b
00000000`00325c00 00000000`774c1937 ntdll! ?? ::FNODOBFM::`string'+0x28ff0
00000000`00325c08 00000000`774ac34e ntdll!LdrInitializeThunk+0xe
00000000`00325c10 00000000`00000000
00000000`00325c18 00060010`00002801
00000000`00325c20 00000000`773757dc kernel32!BaseDllInitialize+0x43c
00000000`00325c28 00000000`774bb108 ntdll!LdrpRunInitializeRoutines+0x1fe
00000000`00325c30 00000000`774c42fd ntdll!LdrGetProcedureAddressEx+0x2aa
00000000`00325c38 00000000`774c1ddc ntdll!LdrpInitializeProcess+0x1a0b
00000000`00325c40 00000000`774c1937 ntdll! ?? ::FNODOBFM::`string'+0x28ff0
00000000`00325c48 00000000`774ac34e ntdll!LdrInitializeThunk+0xe
00000000`00325c50 00000000`00000000
00000000`00325c58 00060011`00002801
00000000`00325c60 00000000`7737582c kernel32!BaseDllInitialize+0x48c
00000000`00325c68 00000000`774bb108 ntdll!LdrpRunInitializeRoutines+0x1fe
00000000`00325c70 00000000`774c42fd ntdll!LdrGetProcedureAddressEx+0x2aa
```

```

00000000`00325c78 00000000`774c1ddc ntdll!LdrpInitializeProcess+0x1a0b
00000000`00325c80 00000000`774c1937 ntdll! ?? ::FNODOBFM::`string'+0x28ff0
00000000`00325c88 00000000`774ac34e ntdll!LdrInitializeThunk+0xe
00000000`00325c90 00000000`00000000
00000000`00325c98 00060012`0000280e
00000000`00325ca0 000007fe`fd5e37aa KERNELBASE!InitializeCriticalSectionAndSpinCount+0xa
00000000`00325ca8 00000001`3fd7319f AllocFree!_mtinitlocks+0x43
00000000`00325cb0 00000001`3fd717fc AllocFree!_mtinit+0x10
00000000`00325cb8 00000001`3fd710e4 AllocFree!__tmainCRTStartup+0x94
00000000`00325cc0 00000000`773759ed kernel32!BaseThreadInitThunk+0xd
00000000`00325cc8 00000000`774ac541 ntdll!RtlUserThreadStart+0x1d
00000000`00325cd0 00000000`00000000
00000000`00325cd8 000b0013`00005801
00000000`00325ce0 00000000`774c1131 ntdll!RtlpActivateLowFragmentationHeap+0x181
00000000`00325ce8 00000000`774c0f97 ntdll!RtlpPerformHeapMaintenance+0x27
00000000`00325cf0 00000000`774c0f5b ntdll!RtlpAllocateHeap+0x1819
00000000`00325cf8 00000000`774d34d8 ntdll!RtlAllocateHeap+0x16c
00000000`00325d00 00000000`774a9300 ntdll!RtlInitializeCriticalSectionAndSpinCount+0x183
00000000`00325d08 000007fe`fd5e37aa KERNELBASE!InitializeCriticalSectionAndSpinCount+0xa
00000000`00325d10 00000001`3fd7319f AllocFree!_mtinitlocks+0x43
00000000`00325d18 00000001`3fd717fc AllocFree!_mtinit+0x10
00000000`00325d20 00000001`3fd710e4 AllocFree!__tmainCRTStartup+0x94
00000000`00325d28 00000000`773759ed kernel32!BaseThreadInitThunk+0xd
00000000`00325d30 00000000`774ac541 ntdll!RtlUserThreadStart+0x1d
00000000`00325d38 00000000`00000000
00000000`00325d40 00000000`00000000
00000000`00325d48 00070014`00003801
00000000`00325d50 000007fe`fd5e37aa KERNELBASE!InitializeCriticalSectionAndSpinCount+0xa
00000000`00325d58 00000001`3fd7312f AllocFree!_mtinitlocknum+0x8f
00000000`00325d60 00000001`3fd72ff7 AllocFree!_lock+0x23
00000000`00325d68 00000001`3fd71f9b AllocFree!_ioinit+0x2f
00000000`00325d70 00000001`3fd71115 AllocFree!__tmainCRTStartup+0xc5
00000000`00325d78 00000000`773759ed kernel32!BaseThreadInitThunk+0xd
00000000`00325d80 00000000`774ac541 ntdll!RtlUserThreadStart+0x1d
00000000`00325d88 00000000`00000000
00000000`00325d90 00000000`00000000
00000000`00325d98 00050015`00002803
00000000`00325da0 000007fe`fd5e37aa KERNELBASE!InitializeCriticalSectionAndSpinCount+0xa
00000000`00325da8 00000001`3fd72239 AllocFree!_ioinit+0x2cd
00000000`00325db0 00000001`3fd71115 AllocFree!__tmainCRTStartup+0xc5
00000000`00325db8 00000000`773759ed kernel32!BaseThreadInitThunk+0xd
00000000`00325dc0 00000000`774ac541 ntdll!RtlUserThreadStart+0x1d
00000000`00325dc8 00000000`00000000
[...]

```

This database corresponds to this simple program:

```

int _tmain(int argc, _TCHAR* argv[])
{
    free(malloc(256));
    Sleep(-1);
    return 0;
}

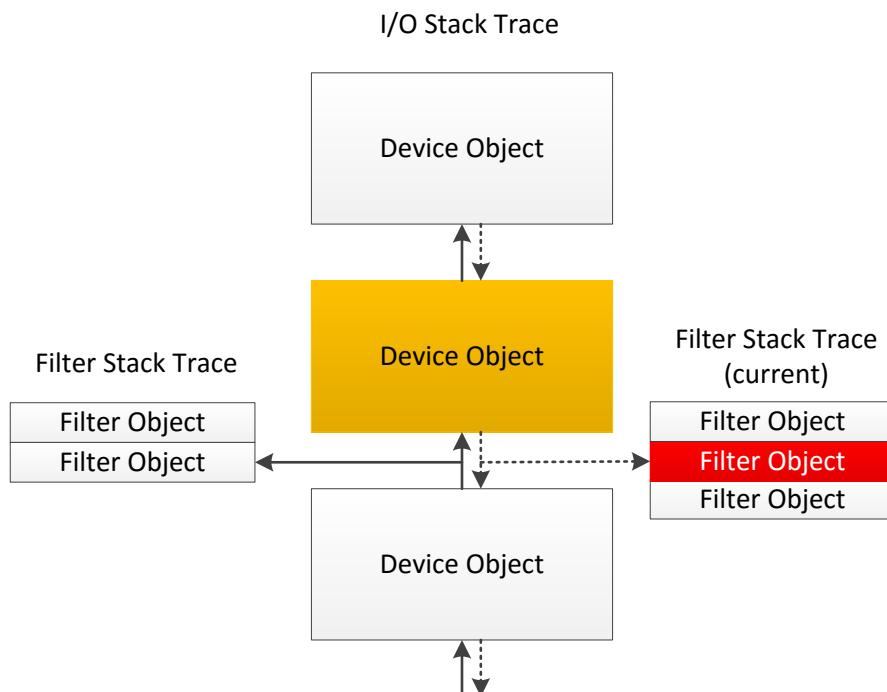
```

## File System Filters

Sometimes threads related to file system operations may be **Blocked Threads** (page 82) with not easily recognizable 3rd-party **Top Module** (page 1012) with only OS vendor modules such as *NTFS* or *futmgr* present:

```
nt!KiSwapContext+0x7a
nt!KiCommitThreadWait+0x1d2
nt!KeWaitForSingleObject+0x19f
nt!FsRtlCancellableWaitForMultipleObjects+0x5e
nt!FsRtlCancellableWaitForSingleObject+0x27
futmgr! ?? ::FNODOBFM::`string'+0x2bfa
futmgr!FltpCreate+0x2a9
nt!IopParseDevice+0x14d3
nt!ObpLookupObjectName+0x588
nt!ObOpenObjectByName+0x306
nt!IopCreateFile+0x2bc
nt!NtCreateFile+0x78
nt!KiSystemServiceCopyEnd+0x13
ntdll!NtCreateFile+0xa
[...]
```

We see the same modules in I/O Request **Stack Trace** (page 926) from the thread IRP. However, because we see filter manager involved, there may be some 3rd-party file system filters involved. Such filters are called before a device processes a request and also upon the completion of the request. There may be different filter callbacks registered for each case, and they form a similar structure like I/O stack locations:



If one of such filters is blocked in a wait chain, this may not be visible on I/O request or thread stacks because of possible asynchronous processing. However, we may use **!fltkd.irpcctl** debugging extension command to examine the IRP context:

```
0: kd> !irp ffffffa80162aa230
cmd      flg cl Device          File          Completion-Context
[...]
[ 0, 0]  0  0  ffffffa800cb28030 00000000  fffff880012048f0-fffffa8016f64010
\FileSystem\Ntfs fltmgr!FltpSynchronizedOperationCompletion
Args: 00000000 00000000 00000000 00000000
> [ 0, 0] 0  1  ffffffa800ca00890 ffffffa801060d070 00000000-00000000 pending
\FileSystem\FltMgr
Args: fffff88014450868 02000060 00000006 00000000

0: kd> !fltkd.irpcctl ffffffa8016f64010
[...]
Cmd      IrpFl    OpFl   CmpFl Instance FileObjt Completion-Context Node Adr
----- -----
[0,0] 00000884 00 0000 ffffffa800d29c010 ffffffa801060d070 fffff8800518b474-0000000000000000
fffffa8016f641e0
("luafv","luafv") luafv!LuafvPostCreate
Args: fffff88014450868 000000002000060 0000000000000006 0000000000000000 0000000000000000
0000000000000000
>[0,0] 00000884 00 0000 ffffffa800e8051d0 ffffffa801060d070 fffff88006808440-0000000000000000
fffffa8016f64160
("3rdPartyFilter","3rdPartyFilter Instance") FilterA!FltDriver_PostOperationCallback
Args: fffff88014450868 000000002000060 0000000000000006 0000000000000000 0000000000000000
0000000000000000
[...]
```

So we see that *FilterA* module may be involved in blocking the thread. We may consider this as **Blocking Module** (page 96) pattern extended to I/O request and filter stack traces.

## General

The most important pattern that is used for problem identification and resolution is **Stack Trace**. Consider the following fragment of `!analyze -v` output from `w3wp.exe` crash dump:

```
STACK_TEXT:
WARNING: Frame IP not in any known module. Following frames may be wrong.
1824f90c 5a39f97e 01057b48 01057bd0 5a3215b4 0x0
1824fa50 5a32cf7c 01057b48 00000000 79e651c0 w3core!ISAPI_REQUEST::SendResponseHeaders+0x5d
1824fa78 5a3218ad 01057bd0 79e651c0 79e64d9c w3isapi!SSFSendResponseHeader+0xe0
1824fae8 79e76127 01057bd0 00000003 79e651c0 w3isapi!ServerSupportFunction+0x351
1824fb0c 79e763a3 80000411 00000000 00000000 aspnet_isapi!HttpCompletion::ReportHttpError+0x3a
1824fd50 79e761c3 34df6cf8 79e8e42f 79e8e442
aspnet_isapi!HttpCompletion::ProcessRequestInManagedCode+0x1d1
1824fd5c 79e8e442 34df6cf8 00000000 00000000 aspnet_isapi!HttpCompletion::ProcessCompletion+0x24
1824fd70 791d6211 34df6cf8 18e60110 793ee0d8 aspnet_isapi!CorThreadPoolWorkitemCallback+0x13
1824fd84 791d616a 18e60110 00000000 791d60fa mscorsvr!ThreadpoolMgr::ExecuteWorkRequest+0x19
1824fda4 791fe95c 00000000 8083d5c7 00000000 msCorsvr!ThreadpoolMgr::WorkerThreadStart+0x129
1824ffb8 77e64829 17bb9c18 00000000 00000000 msCorsvr!ThreadpoolMgr::intermediateThreadProc+0x44
1824ffec 00000000 791fe91b 17bb9c18 00000000 kernel32!BaseThreadStart+0x34
```

Ignoring the first 5 numeric columns gives us the following trace:

```
0x0
w3core!ISAPI_REQUEST::SendResponseHeaders+0x5d
w3isapi!SSFSendResponseHeader+0xe0
w3isapi!ServerSupportFunction+0x351
aspnet_isapi!HttpCompletion::ReportHttpError+0x3a
aspnet_isapi!HttpCompletion::ProcessRequestInManagedCode+0x1d1
aspnet_isapi!HttpCompletion::ProcessCompletion+0x24
aspnet_isapi!CorThreadPoolWorkitemCallback+0x13
mscorsvr!ThreadpoolMgr::ExecuteWorkRequest+0x19
mscorsvr!ThreadpoolMgr::WorkerThreadStart+0x129
mscorsvr!ThreadpoolMgr::intermediateThreadProc+0x44
kernel32!BaseThreadStart+0x34
```

In general we have something like this:

```
moduleA!functionX+offsetN
moduleB!functionY+offsetM
...
...
...
```

Sometimes function names are not available, or offsets are very big like `0x2380`. If this is the case, then we probably don't have symbol files for `moduleA` and `moduleB`:

```
moduleA+offsetN
moduleB+offsetM
...
...
...
```

Usually, there is some kind of a database of previous issues we can use to match *moduleA!functionX + offsetN* against. If there is no such match we can try *functionX + offsetN*, *moduleA!functionX* or just *functionX*. If there is no such match again, we can try the next signature, *moduleB!functionY+offSetM*, and *moduleB!functionY*. Usually, the further in the trace the less useful the signature is for problem resolution. For example, *mscorsvr!ThreadpoolMgr::WorkerThreadStart+0x129* will probably match many issues because this signature is common for many ASP.NET applications.

If there is no match in internal databases, we can try Internet search engines. For our example, Google search for *SendResponseHeaders+0x5d* gives the following search results:

Google search results for "SendResponseHeaders+0x5d":

- w3wp.exe Hanging - IIS State log**  
ISAPI\_REQUEST::SendResponseHeaders+0x5d 02 020afb78 5a3218b5  
SSFSendResponseHeader+0xe0 03 020afbe8 79e76100 w3isapi!  
ServerSupportFunction+0x351 ...  
[www.associate.de/board/post/216102/w3wp.exe\\_Hanging\\_-\\_IIS\\_State\\_log.h](http://www.associate.de/board/post/216102/w3wp.exe_Hanging_-_IIS_State_log.h)  
[Cached](#) - [Similar pages](#)
- IIS 6 w3wp.exe crashes - help please reading the stack**  
ISAPI\_REQUEST::SendResponseHeaders+0x5d 02 019afb78 5a  
SSFSendResponseHeader+0xe0 03 019afbe8 79e76100 w3isapi!  
ServerSupportFunction+0x351 ...  
[www.associate.de/board/post/257974/IIS\\_6\\_w3wp.exe\\_crashes\\_-\\_helpPleaseReadingTheStackDump.html](http://www.associate.de/board/post/257974/IIS_6_w3wp.exe_crashes_-_helpPleaseReadingTheStackDump.html) - 9k - [Cached](#) - [Similar](#)

Browsing search results reveals the discussion<sup>195</sup> which can be found directly by searching Google groups:

Google Groups search results for "SendResponseHeaders+0x5d":

- w3wp.exe Hanging - IIS State log** Group: [microsoft.public.inetserver.iis](#)  
ISAPI\_REQUEST::SendResponseHeaders+0x5d 02 020afb78 5a3218b5  
w3isapi!SSFSendResponseHeader+0xe0 03 020afbe8 79e76100 w3isapi!  
04 020afc0c 79e7637c aspnet\_isapi!HttpCompletion::ReportHttpError+0x3  
79e7619c aspnet\_isapi!HttpCompletion::ProcessRequestInManagedCode+  
[25 May 2005 by Pat \[MSFT\]](#) - 7 messages - 2 authors
- IIS 6 w3wp.exe crashes - help please reading the stack dump**  
ISAPI\_REQUEST::SendResponseHeaders+0x5d 02 019afb78 5a3218b5  
w3isapi!SSFSendResponseHeader+0xe0 03 019afbe8 79e76100 w3isapi!  
04 019afc0c 79e7637c aspnet\_isapi!HttpCompletion::ReportHttpError+0x3  
79e7619c aspnet\_isapi!HttpCompletion::ProcessRequestInManagedCode+  
[27 Sep 2005 by John Crim](#) - 2 messages - 1 author

<sup>195</sup> [http://groups.google.com/group/microsoft.public.inetserver.iis/browse\\_frm/thread/34bc2be635b26531?tvc=1](http://groups.google.com/group/microsoft.public.inetserver.iis/browse_frm/thread/34bc2be635b26531?tvc=1)

Another example is from BSOD complete memory dump. Analysis command has the following output:

```
MODE_EXCEPTION_NOT_HANDLED (1e)
This is a very common bugcheck. Usually the exception address pinpoints the driver/function that caused
the problem. Always note this address as well as the link date of the driver/image that contains this
address.

Arguments:
Arg1: c0000005, The exception code that was not handled
Arg2: bff90ca3, The address that the exception occurred at
Arg3: 00000000, Parameter 0 of the exception
Arg4: 00000000, Parameter 1 of the exception

TRAP_FRAME: bdf80834 -- (.trap ffffffffbd80834)
ErrCode = 00000000
eax=00000000 ebx=bdf80c34 ecx=89031870 edx=88096928 esi=88096928 edi=8905e7f0
eip=bff90ca3 esp=bdf808a8 ebp=bdf80a44 iopl=0 nv up ei ng nz na po nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010282
tsmlvs+0xfcfa3:
bff90ca3 8b08 mov ecx,dword ptr [eax] ds:0023:00000000=?????????
Resetting default scope

STACK_TEXT:
bdf807c4 80467a15 bdf807e0 00000000 bdf80834 nt!KiDispatchException+0x30e
bdf8082c 804679c6 00000000 bdf80860 804d9f69 nt!CommonDispatchException+0x4d
bdf80838 804d9f69 00000000 00000005 e56c6946 nt!KiUnexpectedInterruptTail+0x207
00000000 00000000 00000000 00000000 nt!ObpAllocateObject+0xe1
```

Because the crash point *tsmlvs+0xfcfa3* is not on the stack trace we use **.trap** command:

```
1: kd> .trap ffffffffbd80834
ErrCode = 00000000
eax=00000000 ebx=bdf80c34 ecx=89031870 edx=88096928 esi=88096928 edi=8905e7f0
eip=bff90ca3 esp=bdf808a8 ebp=bdf80a44 iopl=0 nv up ei ng nz na po nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010282
tsmlvs+0xfcfa3:
bff90ca3 8b08 mov ecx,dword ptr [eax] ds:0023:00000000=?????????

1: kd> k
*** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr
WARNING: Stack unwind information not available. Following frames may be wrong.
00000000 bdf80afc tsmlvs+0xfcfa3
89080c00 00000040 nt!ObpLookupObjectName+0x504
00000000 00000001 nt!ObOpenObjectByName+0xc5
c0100080 0012b8d8 nt!IopCreateFile+0x407
c0100080 0012b8d8 nt!IoCreateFile+0x36
c0100080 0012b8d8 nt!NtCreateFile+0x2e
c0100080 0012b8d8 nt!KiSystemService+0xc9
c0100080 0012b8d8 ntdll!NtCreateFile+0xb
00000000 00000000 KERNEL32!CreateFileW+0x343
```

```
1: kd> lmv m tsmlvsa
bff81000 bff987c0 tsmlvsa (no symbols)
Loaded symbol image file: tsmlvsa.sys
Image path: tsmlvsa.sys
Image name: tsmlvsa.sys
Timestamp: Thu Mar 18 06:18:51 2004 (40593F4B)
CheckSum: 0002D102
ImageSize: 000177C0
Translations: 0000.04b0 0000.04e0 0409.04b0 0409.04e0
```

Google search for *tsmlvsa+0xfcfa3* doesn't show anything, but if we search just for *tsmlvsa*, we may get some links towards problem resolution.

## I/O Request

If a thread has an associated I/O Request Packet (IRP), we may see another type of a stack trace. It also grows bottom-up as can be seen from the diagram<sup>196</sup>. We can see this stack trace by using **!irp** WinDbg command:

```
0: kd> !thread ffffffa801827a4c0 1f
THREAD ffffffa801827a4c0 Cid 06c0.50cc Teb: 000007ffffec8000 Win32Thread: fffff900c1c64010 WAIT:
(Executive) KernelMode Alertable
fffffa8016f64028 SynchronizationEvent
IRP List:
fffffa80162aa230: (0006,03a0) Flags: 00000884 Mdl: 00000000
[...]
nt!KiSwapContext+0x7a
nt!KiCommitThreadWait+0x1d2
nt!KeWaitForSingleObject+0x19f
nt!FsRtlCancellableWaitForMultipleObjects+0x5e
nt!FsRtlCancellableWaitForSingleObject+0x27
fltmgr! ?? ::FNODOBFM::`string'+0x2bfa
fltmgr!FltpCreate+0x2a9
nt!IopParseDevice+0x14d3
nt!ObpLookupObjectName+0x588
nt!ObOpenObjectByName+0x306
nt!IopCreateFile+0x2bc
nt!NtCreateFile+0x78
nt!KiSystemServiceCopyEnd+0x13
ntdll!NtCreateFile+0xa
[...]
```

---

<sup>196</sup> UML and Device Drivers, Memory Dump Analysis Anthology, Volume 1, page 701

```
0: kd> !irp ffffffa80162aa230
Irp is active with 10 stacks 10 is current (= 0xffffffa80162aa588)
No Mdl: No System Buffer: Thread ffffffa801827a4c0: Irp stack trace.
cmd flg cl Device File Completion-Context
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 fffffa800cb28030 00000000 fffff880012048f0-fffffa8016f64010
\FileSystem\Ntfs fltmgr!FltpSynchronizedOperationCompletion
Args: 00000000 00000000 00000000 00000000
>[ 0, 0] 0 1 fffffa800ca00890 fffffa801060d070 00000000-00000000 pending
\FileSystem\FltMgr
Args: fffff88014450868 02000060 00000006 00000000
```

We see the current stack trace pointer points to the bottom I/O stack location. Non-empty top locations are analogous to **Past Stack Trace** (page 800). Further exploration of *Device* and *File* column information may point to further troubleshooting directions such as the **Blocking File** (page 93) pattern example.

By analogy with **Stack Trace Collection** (page 943) pattern that dumps stack traces from all threads based on memory dump type, there is also I/O Requests **Stack Trace Collection** (page 933) pattern that dumps I/O request stack traces from all IRPs that were possible to find.

## Stack Trace Change

This is an important pattern for differential memory dump analysis, for example, when memory dumps were generated before and after a problem such as a CPU spike or hang. In the example below, we have a normally expected thread stack trace from a memory dump saved before an application was reported unresponsive and another different thread stack trace after:

```
3 Id: 24b8.24e4 Suspend: 0 Teb: 7efa1000 Unfrozen
ChildEBP RetAddr
037dfadc 75210bdd ntdll!ZwWaitForMultipleObjects+0x15
037dfb78 75791a2c KERNELBASE!WaitForMultipleObjectsEx+0x100
037dfbc0 7511086a kernel32!WaitForMultipleObjectsExImplementation+0xe0
037dfc14 00d17c1d user32!RealMsgWaitForMultipleObjectsEx+0x14d
037dfc3c 00ce161d ApplicationA!MsgWaitForMultipleObjects+0x2d
037dfc60 00cdc757 ApplicationA!WaitForSignal+0x1d
037dfc80 00cdaaf6 ApplicationA!WorkLoop+0x57
037dfca4 7579339a ApplicationA!ThreadStart+0x26
037dfcb0 77699ef2 kernel32!BaseThreadInitThunk+0xe
037dfcf0 77699ec5 ntdll!__RtlUserThreadStart+0x70
037dfd08 00000000 ntdll!_RtlUserThreadStart+0x1b
```

```
3 Id: 24b8.24e4 Suspend: 0 Teb: 7efa1000 Unfrozen
ChildEBP RetAddr
037df38c 752131bb ntdll!ZwDelayExecution+0x15
037df3f4 75213a8b KERNELBASE!SleepEx+0x65
037df404 00d1670b KERNELBASE!Sleep+0xf
037df40c 00d350ef ApplicationA!Sleep+0xb
037df430 6a868aab ApplicationA!PutData+0xbff
037df444 6a8662ec ModuleA!OutputData+0x1b
037df464 00d351de ModuleA!ProcessData+0x16c
037df4a4 00ca8cb4 ApplicationA!SendData+0xbe
[...]
```

## Stack Trace Collection

### CPUs

This is another variant of **Stack Trace Collection** pattern that shows stack traces from threads currently execution on all CPUs. Although we can see the non-idle running threads from the stack traces corresponding to all processes and their threads (page 943) we may also want to see idle thread stack traces too. Also, the corresponding WinDbg command (`!running -t -i`) is faster if we want to double check the output of `!analyze -v` command in case of BSOD. The latter command may show the stack trace from the current CPU instead of the stack trace from the thread running on a different CPU that caused a bugcheck. Here's an example from one of the memory dumps for which `!analyze -v` command shows an incorrect stack trace in the output when we open the dump file. It reports the stack trace from CPU 0, but the bugcheck happened on CPU 1:

```
0: kd> !running -t -i

System Processors:  (00000000000000ff)
Idle Processors:  (00000000000000fd)

Prcbs          Current          (pri) Next          (pri) Idle
0  ffffff801e5d85180  ffffff801e5dde00 ( 0)          ffffff801e5dde00  .....
.
.
.

Child-SP          RetAddr          Call Site
fffff801`e8c9eb60  ffffff801`e5c69b74 hal!KeQueryPerformanceCounter+0x75
fffff801`e8c9eba0  ffffff801`e5c69e01 nt!KiCheckStall+0x2c
fffff801`e8c9ebd0  ffffff801`e5c6aa8f nt!KiFreezeTargetExecution+0x231
fffff801`e8c9ece0  ffffff801`e5bdbec2 nt!KiProcessNMI+0x3b
fffff801`e8c9ed30  ffffff801`e5bdbd36 nt!KxNmiInterrupt+0x82
fffff801`e8c9ee70  ffffff801`e5a2d82f nt!KiNmiInterrupt+0x176
fffff801`e8c8c8e8  ffffff801`e5bb91a2 hal!HalProcessorIdle+0xf
fffff801`e8c8c8f0  ffffff801`e5ad7848 nt!PpmIdleDefaultExecute+0xa
fffff801`e8c8c920  ffffff801`e5ad72a6 nt!PpmIdleExecuteTransition+0x3e8
fffff801`e8c8cb10  ffffff801`e5bd64bc nt!PoIdle+0x2f6
fffff801`e8c8cc60  00000000`00000000 nt!KiIdleLoop+0x2c

1  fffffd000f0975180  fffffe0000d726880 (12)          fffffd000f09813c0  .....
.
.

Child-SP          RetAddr          Call Site
fffffd000`202cb618  ffffff801`e5a1cc3c hal!HalpAcpiPmRegisterReadPort+0x1b
fffffd000`202cb620  ffffff801`e5a417e7 hal!HalpAcpiPmRegisterRead+0x30
fffffd000`202cb650  ffffff801`e5c66af5 hal!HalHaltSystem+0x53
fffffd000`202cb690  ffffff801`e5c66741 nt!KiBugCheckDebugBreak+0x99
fffffd000`202cb6f0  ffffff801`e5bd2aa4 nt!KeBugCheck2+0xc6d
fffffd000`202cbe00  ffffff801`e5bde4e9 nt!KeBugCheckEx+0x104
fffffd000`202cbe40  ffffff801`e5bdcda nt!KiBugCheckDispatch+0x69
fffffd000`202cbf80  ffffff800`913601da nt!KiPageFault+0x23a
fffffd000`202cc118  ffffff800`91363710 DriverA!memcpy+0x21a
[...]
```

2	fffffd000f09ee180 fffffd000f09fa3c0 ( 0)	fffffd000f09fa3c0 .....  Child-SP RetAddr Call Site ffffd000`f09f9f88 fffff801`e5c69e01 nt!KiCheckStall+0xa ffffd000`f09f9f90 fffff801`e5c6aa8f nt!KiFreezeTargetExecution+0x231 ffffd000`f09fa0a0 fffff801`e5bdbec2 nt!KiProcessNMI+0x3b ffffd000`f09fa0f0 fffff801`e5bdbd36 nt!KxNmiInterrupt+0x82 ffffd000`f09fa230 fffff801`e5a2d82f nt!KiNmiInterrupt+0x176 ffffd000`eb5938e8 fffff801`e5bb91a2 hal!HalProcessorIdle+0xf ffffd000`eb5938f0 fffff801`e5ad7848 nt!PpmIdleDefaultExecute+0xa ffffd000`eb593920 fffff801`e5ad72a6 nt!PpmIdleExecuteTransition+0x3e8 ffffd000`eb593b10 fffff801`e5bd64bc nt!PoIdle+0x2f6 ffffd000`eb593c60 00000000`00000000 nt!KiIdleLoop+0x2c
3	fffffd000eb5e5180 fffffd000eb5f13c0 ( 0)	fffffd000eb5f13c0 .....  Child-SP RetAddr Call Site ffffd000`eb5f0f60 fffff801`e5c69e01 nt!KiCheckStall+0x5f ffffd000`eb5f0f90 fffff801`e5c6aa8f nt!KiFreezeTargetExecution+0x231 ffffd000`eb5f10a0 fffff801`e5bdbec2 nt!KiProcessNMI+0x3b ffffd000`eb5f10f0 fffff801`e5bdbd36 nt!KxNmiInterrupt+0x82 ffffd000`eb5f1230 fffff801`e5a2d82f nt!KiNmiInterrupt+0x176 ffffd000`eb5fa8e8 fffff801`e5bb91a2 hal!HalProcessorIdle+0xf ffffd000`eb5fa8f0 fffff801`e5ad7848 nt!PpmIdleDefaultExecute+0xa ffffd000`eb5fa920 fffff801`e5ad72a6 nt!PpmIdleExecuteTransition+0x3e8 ffffd000`eb5fab10 fffff801`e5bd64bc nt!PoIdle+0x2f6 ffffd000`eb5fac60 00000000`00000000 nt!KiIdleLoop+0x2c
4	fffffd000f08d1180 fffffd000f08dd3c0 ( 0)	fffffd000f08dd3c0 .....  Child-SP RetAddr Call Site ffffd000`f08dcf90 fffff801`e5c6aa8f nt!KiFreezeTargetExecution+0x227 ffffd000`f08dd0a0 fffff801`e5bdbec2 nt!KiProcessNMI+0x3b ffffd000`f08dd0f0 fffff801`e5bdbd36 nt!KxNmiInterrupt+0x82 ffffd000`f08dd230 fffff801`e5a2d82f nt!KiNmiInterrupt+0x176 ffffd000`eb85b8e8 fffff801`e5bb91a2 hal!HalProcessorIdle+0xf ffffd000`eb85b8f0 fffff801`e5ad7848 nt!PpmIdleDefaultExecute+0xa ffffd000`eb85b920 fffff801`e5ad72a6 nt!PpmIdleExecuteTransition+0x3e8 ffffd000`eb85bb10 fffff801`e5bd64bc nt!PoIdle+0x2f6 ffffd000`eb85bc60 00000000`00000000 nt!KiIdleLoop+0x2c
5	fffffd000eb8ad180 fffffd000eb8b93c0 ( 0)	fffffd000eb8b93c0 .....  Child-SP RetAddr Call Site ffffd000`eb8b8f60 fffff801`e5c69e01 nt!KiCheckStall+0x75 ffffd000`eb8b8f90 fffff801`e5c6aa8f nt!KiFreezeTargetExecution+0x231 ffffd000`eb8b90a0 fffff801`e5bdbec2 nt!KiProcessNMI+0x3b ffffd000`eb8b90f0 fffff801`e5bdbd36 nt!KxNmiInterrupt+0x82 ffffd000`eb8b9230 fffff801`e5a2d82f nt!KiNmiInterrupt+0x176 ffffd000`eb8db8e8 fffff801`e5bb91a2 hal!HalProcessorIdle+0xf ffffd000`eb8db8f0 fffff801`e5ad7848 nt!PpmIdleDefaultExecute+0xa ffffd000`eb8db920 fffff801`e5ad72a6 nt!PpmIdleExecuteTransition+0x3e8 ffffd000`eb8dbb10 fffff801`e5bd64bc nt!PoIdle+0x2f6 ffffd000`eb8dbc60 00000000`00000000 nt!KiIdleLoop+0x2c

6      fffffd000eb92a180    fffffd000eb9363c0 ( 0)	fffffd000eb9363c0 .....  Child-SP                RetAddr                Call Site ffffd000`eb935f60 ffffff801`e5c69e01 nt!KiCheckStall+0x75 ffffd000`eb935f90 ffffff801`e5c6aa8f nt!KiFreezeTargetExecution+0x231 ffffd000`eb9360a0 ffffff801`e5bdbec2 nt!KiProcessNMI+0x3b ffffd000`eb9360f0 ffffff801`e5bdbd36 nt!KxNmiInterrupt+0x82 ffffd000`eb936230 ffffff801`e5a2d82f nt!KiNmiInterrupt+0x176 ffffd000`eb93f8e8 ffffff801`e5bb91a2 hal!HalProcessorIdle+0xf ffffd000`eb93f8f0 ffffff801`e5ad7848 nt!PpmIdleDefaultExecute+0xa ffffd000`eb93f920 ffffff801`e5ad72a6 nt!PpmIdleExecuteTransition+0x3e8 ffffd000`eb93fb10 ffffff801`e5bd64bc nt!PoIdle+0x2f6 ffffd000`eb93fc60 00000000`00000000 nt!KiIdleLoop+0x2c
7      fffffd000eb967180    fffffd000eb9733c0 ( 0)	fffffd000eb9733c0 .....  Child-SP                RetAddr                Call Site ffffd000`eb972f60 ffffff801`e5c69e01 nt!KiCheckStall+0x75 ffffd000`eb972f90 ffffff801`e5c6aa8f nt!KiFreezeTargetExecution+0x231 ffffd000`eb9730a0 ffffff801`e5bdbec2 nt!KiProcessNMI+0x3b ffffd000`eb9730f0 ffffff801`e5bdbd36 nt!KxNmiInterrupt+0x82 ffffd000`eb973230 ffffff801`e5a2d82f nt!KiNmiInterrupt+0x176 ffffd000`eb97c8e8 ffffff801`e5bb91a2 hal!HalProcessorIdle+0xf ffffd000`eb97c8f0 ffffff801`e5ad7848 nt!PpmIdleDefaultExecute+0xa ffffd000`eb97c920 ffffff801`e5ad72a6 nt!PpmIdleExecuteTransition+0x3e8 ffffd000`eb97cb10 ffffff801`e5bd64bc nt!PoIdle+0x2f6 ffffd000`eb97cc60 00000000`00000000 nt!KiIdleLoop+0x2c

This command is obviously faster than repeatedly switching to subsequent CPUs using `s` command and then checking the corresponding stack trace (`k`). It also helps in diagnosing **Spiking Threads** (page 888) in the kernel and complete memory dumps.

## I/O Requests

Such requests are implemented via the so-called I/O request packets (IRP) that “travel” from a device driver to a device driver similar to a C++ class method to another C++ class method (where a device object address is similar to a C++ object instance address). An IRP stack is used to keep track of the current driver processing an IRP that is reused between device drivers. It is basically an array of structures describing how a particular driver function was called with appropriate parameters similar to a call frame on an execution thread stack. A long time ago we created a UML diagram depicting the flow of an IRP through the driver (device) stack<sup>197</sup>. An I/O stack location pointer is decremented (from the bottom to the top) as a thread stack pointer (ESP or RSP). We can list active and completed I/O requests with their stack traces using `!irpfind -v` WinDbg command:

```
1: kd> !irpfind -v

Scanning large pool allocation table for Tag: Irp? (832c7000 : 833c7000)

Irp      [ Thread ] irpStack: (Mj,Mn)    DevObj  [Driver]          MDL Process
8883dc18: Irp is active with 1 stacks 1 is current (= 0x8883dc88)
  No Mdl: No System Buffer: Thread 888f8950:  Irp stack trace.
    cmd  flg cl Device   File   Completion-Context
  > [ d, 0]  5 1 88515ae8 888f82f0 00000000-00000000  pending
    \FileSystem\Npfs
      Args: 00000000 00000000 00110008 00000000

891204c8: Irp is active with 1 stacks 1 is current (= 0x89120538)
  No Mdl: No System Buffer: Thread 889635b0:  Irp stack trace.
    cmd  flg cl Device   File   Completion-Context
  > [ 3, 0]  0 1 88515ae8 84752028 00000000-00000000  pending
    \FileSystem\Npfs
      Args: 0000022a 00000000 00000000 00000000

89120ce8: Irp is active with 1 stacks 1 is current (= 0x89120d58)
  No Mdl: No System Buffer: Thread 89212030:  Irp stack trace.
    cmd  flg cl Device   File   Completion-Context
  > [ 3, 0]  0 1 88515ae8 8921be00 00000000-00000000  pending
    \FileSystem\Npfs
      Args: 0000022a 00000000 00000000 00000000

Searching NonPaged pool (80000000 : ffc00000) for Tag: Irp?

[...]

892cbe48: Irp is active with 9 stacks 9 is current (= 0x892cbfd8)
  No Mdl: No System Buffer: Thread 892add78:  Irp stack trace.
    cmd  flg cl Device   File   Completion-Context
  [ 0, 0]  0 0 00000000 00000000 00000000-00000000
```

<sup>197</sup> UML and Device Drivers, Memory Dump Analysis Anthology, Volume 1, page 701

```
          Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
> [ c, 2] 0 1 8474a020 892c8c80 00000000-00000000 pending
    \FileSystem\Ntfs
          Args: 00000800 00000002 00000000 00000000

892daa88: Irp is active with 4 stacks 4 is current (= 0x892dab64
No Mdl: System buffer=831559c8: Thread 8322c8e8: Irp stack tra
    cmd flg cl Device   File     Completion-Context
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
> [ e,2d] 5 1 884ba750 83190c40 00000000-00000000 pending
    \Driver\AFD
          Args: 890cbc44 890cbc44 88e55297 8943b6c8
```

```

892ea4e8: Irp is active with 4 stacks 4 is current (= 0x892ea5c4)
No Mdl: No System Buffer: Thread 0000000: Irp stack trace. Pending has been returned
  cmd flg cl Device File Completion-Context
[ 0, 0] 0 2 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 c0000185
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
[ f, 0] 0 2 83a34bb0 00000000 84d779ed-88958050
  \Driver\atapi CLASSPNP!ClasspMediaChangeDetectionCompletion
    Args: 88958050 00000000 00000000 83992d10
> [ 0, 0] 2 0 891ee030 00000000 00000000-00000000
  \Driver\cdrom
    Args: 00000000 00000000 00000000 00000000

8933fcb0: Irp is active with 1 stacks 1 is current (= 0x8933fd20)
No Mdl: No System Buffer: Thread 84753d78: Irp stack trace.
  cmd flg cl Device File Completion-Context
> [ 3, 0] 0 1 88515ae8 84759f40 00000000-00000000 pending
  \FileSystem\Npfs
    Args: 0000022a 00000000 00000000 00000000

893cf550: Irp is active with 1 stacks 1 is current (= 0x893cf5c0)
No Mdl: No System Buffer: Thread 888fd3b8: Irp stack trace.
  cmd flg cl Device File Completion-Context
> [ 3, 0] 0 1 88515ae8 834d30d0 00000000-00000000 pending
  \FileSystem\Npfs
    Args: 00000400 00000000 00000000 00000000

893da468: Irp is active with 6 stacks 7 is current (= 0x893da5b0)
Mdl=892878f0: No System Buffer: Thread 0000000: Irp is completed. Pending has been returned
  cmd flg cl Device File Completion-Context
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 84b3e028 00000000 9747fc00-00000000
  \Driver\usbehci USBSTOR!USBSTOR_CswCompletion
    Args: 00000000 00000000 00000000 00000000
[ f, 0] 0 0 892ba8f8 00000000 84d780ce-8328e0f0
  \Driver\USBSTOR CLASSPNP!TransferPktComplete
    Args: 00000000 00000000 00000000 00000000

```

```
893efb00: Irp is active with 10 stacks 11 is current (= 0x893efcd8)
Mdl=83159378: No System Buffer: Thread 82b7f828: Irp is completed. Pending has been returned
  cmd flg cl Device File Completion-Context
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

          Args: 00000000 00000000 00000000 00000000
[ 3, 0] 0 0 885a55b8 00000000 81614138-00000000
          \Driver\disk partmgr!PmReadWriteCompletion
          Args: 00000000 00000000 00000000 00000000
[ 3, 0] 0 0 89257c90 00000000 8042e4d4-831caab0
          \Driver\partmgr volmgr!VmpReadWriteCompletionRoutine
          Args: 00000000 00000000 00000000 00000000
[ 3, 0] 0 0 831ca9f8 00000000 84dad0be-00000000
          \Driver\volmgr ecache!EcDispatchReadWriteCompletion
          Args: 00000000 00000000 00000000 00000000
[ 3, 0] 0 0 8319c020 00000000 84dcc4d4-8576f8ac
          \Driver\Ecache volsnap!VspSignalCompletion
          Args: 00000000 00000000 00000000 00000000
```

## Managed Space

Here we have a managed space counterpart to unmanaged **Stack Trace Collection** (page 943) pattern. When looking at crash dumps from a different than a postmortem analysis machine we might need the appropriate **Version-Specific SOS Extension** (page 1058). To list all managed stack traces we use this combined command:

```
0:000> ~*e !CLRStack
OS Thread Id: 0x36f0 (0)
Child SP IP          Call Site
0031cf84 779b0f34 [HelperMethodFrame: 0031cf84]
0031cf8 6b65665e System.Collections.ArrayList.GetEnumerator()
0031cfe4 059f4c92 ActiproSoftware.SyntaxEditor.EditorView.a(System.Windows.Forms.PaintEventArgs,
System.Drawing.Rectangle, ActiproSoftware.SyntaxEditor.DocumentLine,
ActiproSoftware.SyntaxEditor.DisplayLine, ActiproSoftware.SyntaxEditor.EditPositionRange, Int32 ByRef)
0031e158 05a01798 ActiproSoftware.SyntaxEditor.EditorView.OnRender(System.Windows.Forms.PaintEventArgs)
0031e748 04c5f888 ActiproSoftware.WinUICore.UIElement.Render(System.Windows.Forms.PaintEventArgs)
0031e758 04c5f602
ActiproSoftware.WinUICore.UIControl.OnRenderChildElements(System.Windows.Forms.PaintEventArgs)
0031e80c 04c5f1ac ActiproSoftware.WinUICore.UIControl.Render(System.Windows.Forms.PaintEventArgs)
0031e81c 04c5e6fe ActiproSoftware.WinUICore.UIControl.a(System.Windows.Forms.PaintEventArgs)
0031e9a4 04c5e415 ActiproSoftware.WinUICore.UIControl.OnPaint(System.Windows.Forms.PaintEventArgs)
0031e9b4 69f156f5
System.Windows.Forms.Control.PaintWithErrorHandling(System.Windows.Forms.PaintEventArgs, Int16)
0031e9e8 69f1809e System.Windows.Forms.Control.WmPaint(System.Windows.Forms.Message ByRef)
0031ead4 69f073b1 System.Windows.Forms.Control.WndProc(System.Windows.Forms.Message ByRef)
0031ead8 69f102aa [InlinedCallFrame: 0031ead8]
0031eb2c 69f102aa System.Windows.Forms.ScrollableControl.WndProc(System.Windows.Forms.Message ByRef)
0031eb38 048269c3 ActiproSoftware.SyntaxEditor.SyntaxEditor.WndProc(System.Windows.Forms.Message ByRef)
0031eb8 69f070f3 System.Windows.Forms.Control+ControlNativeWindow.OnMessage(System.Windows.Forms.Message
ByRef)
0031ebf0 69f07071 System.Windows.Forms.Control+ControlNativeWindow.WndProc(System.Windows.Forms.Message
ByRef)
0031ec04 69f06fb6 System.Windows.Forms.NativeWindow.Callback(IntPtr, Int32, IntPtr, IntPtr)
0031ee44 00e71365 [InlinedCallFrame: 0031ee44]
0031ee40 69f22eec DomainNeutralILStubClass.IL_STUB_PInvoke(MSG ByRef)
0031ee44 69f171ff [InlinedCallFrame: 0031ee44]
System.Windows.Forms.UnsafeNativeMethods.DispatchMessageW(MSG ByRef)
0031ee88 69f171ff System.Windows.Forms.Application+ComponentManager.System.Windows.Forms.
UnsafeNativeMethods.IMsoComponentManager.FPushMessageLoop(IntPtr, Int32, Int32)
0031ee8c 69f16e2c [InlinedCallFrame: 0031ee8c]
0031ef24 69f16e2c System.Windows.Forms.Application+ThreadContext.RunMessageLoopInner(Int32,
System.Windows.Forms.ApplicationContext)
0031ef7c 69f16c81 System.Windows.Forms.Application+ThreadContext.RunMessageLoop(Int32,
System.Windows.Forms.ApplicationContext)
0031efac 69ea366d System.Windows.Forms.Application.Run(System.Windows.Forms.Form)
0031efc0 003c3f0d LINQPad.Program.Run(System.String, Boolean, System.String, Boolean, Boolean,
System.String)
0031f0b4 003c1515 LINQPad.Program.Go(System.String[])
0031f2e4 003c0584 LINQPad.Program.Start(System.String[])
0031f324 003c034f LINQPad.ProgramStarter.Run(System.String[])
0031f330 003c00f5 LINQPad.Loader.Main(System.String[])
0031f570 6c1721db [GCFrame: 0031f570]
OS Thread Id: 0x36f8 (1)
Unable to walk the managed stack. The current thread is likely not a
```

managed thread. You can run !threads to get a list of managed threads in the process

**OS Thread Id: 0x36fc (2)**  
Child SP IP Call Site  
03c2fb78 779b0f34 [DebuggerU2MCatchHandlerFrame: 03c2fb78]

**OS Thread Id: 0x3700 (3)**  
Child SP IP Call Site  
0470f0cc 779b0f34 [InlinedCallFrame: 0470f0cc]  
0470f0c8 699380b7 DomainNeutralILStubClass.IL\_STUB\_PInvoke(Microsoft.Win32.SafeHandles.SafePipeHandle, IntPtr)  
0470f0cc 699cc07c [InlinedCallFrame: 0470f0cc]  
Microsoft.Win32.UnsafeNativeMethods.ConnectNamedPipe(Microsoft.Win32.SafeHandles.SafePipeHandle, IntPtr)  
0470f130 699cc07c System.IO.Pipes.NamedPipeServerStream.WaitForConnection()  
0470f140 003c31ed LINQPad.Program.Listen()  
0470f1d8 6b64ae5b System.Threading.ThreadHelper.ThreadStart\_Context(System.Object)  
0470f1e8 6b5d7ff4 System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext, System.Threading.ContextCallback, System.Object, Boolean)  
0470f20c 6b5d7f34 System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext, System.Threading.ContextCallback, System.Object)  
0470f228 6b64ade8 System.Threading.ThreadHelper.ThreadStart()  
0470f44c 6c1721db [GCFrame: 0470f44c]  
0470f710 6c1721db [DebuggerU2MCatchHandlerFrame: 0470f710]

**OS Thread Id: 0x3704 (4)**  
Unable to walk the managed stack. The current thread is likely not a managed thread. You can run !threads to get a list of managed threads in the process

**OS Thread Id: 0x3710 (5)**  
Unable to walk the managed stack. The current thread is likely not a managed thread. You can run !threads to get a list of managed threads in the process

**OS Thread Id: 0x3714 (6)**  
Unable to walk the managed stack. The current thread is likely not a managed thread. You can run !threads to get a list of managed threads in the process

**OS Thread Id: 0x3718 (7)**  
Child SP IP Call Site  
0596ee40 779b0f34 [GCFrame: 0596ee40]  
0596ef34 779b0f34 [HelperMethodFrame\_10BJ: 0596ef34] System.Threading.Monitor.ObjWait(Boolean, Int32, System.Object)  
0596ef90 6b5d9140 System.Threading.Monitor.Wait(System.Object, Int32, Boolean)  
0596efa0 6bb9428a System.Threading.Monitor.Wait(System.Object)  
0596efa4 003cb55a ActiproSoftware.SyntaxEditor.SemanticParserService.c()  
0596f068 6b64ae5b System.Threading.ThreadHelper.ThreadStart\_Context(System.Object)  
0596f078 6b5d7ff4 System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext, System.Threading.ContextCallback, System.Object, Boolean)  
0596f09c 6b5d7f34 System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext, System.Threading.ContextCallback, System.Object)  
0596f0b8 6b64ade8 System.Threading.ThreadHelper.ThreadStart()  
0596f2dc 6c1721db [GCFrame: 0596f2dc]  
0596f5a0 6c1721db [DebuggerU2MCatchHandlerFrame: 0596f5a0]

**OS Thread Id: 0x3764 (8)**  
Child SP IP Call Site  
0564f148 779b0f34 [HelperMethodFrame\_10BJ: 0564f148]  
System.Threading.WaitHandle.WaitOneNative(System.Runtime.InteropServices.SafeHandle, UInt32, Boolean, Boolean)  
0564f1f0 6b64b5ef System.Threading.InternalWaitOne(System.Runtime.InteropServices.SafeHandle,

```
Int64, Boolean, Boolean)
0564f20c 6b64b5ad System.Threading.WaitHandle.WaitOne(Int32, Boolean)
0564f224 6b64b570 System.Threading.WaitHandle.WaitOne()
0564f22c 04b607d4 ActiproBridge.ReferenceManager+?15?.<StartAdvanceFeeder>b__4()
0564f268 6b64ae5b System.Threading.ThreadHelper.ThreadStart_Context(System.Object)
0564f278 6b5d7ff4 System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object, Boolean)
0564f29c 6b5d7f34 System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
0564f2b8 6b64ade8 System.Threading.ThreadHelper.ThreadStart()
0564f4dc 6c1721db [GCFrame: 0564f4dc]
0564f7a0 6c1721db [DebuggerU2MCatchHandlerFrame: 0564f7a0]
OS Thread Id: 0x3768 (9)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
OS Thread Id: 0x376c (10)
Child SP IP           Call Site
GetFrameContext failed: 1
OS Thread Id: 0x3798 (11)
Child SP IP           Call Site
GetFrameContext failed: 1
OS Thread Id: 0x37e0 (12)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
OS Thread Id: 0x37f8 (13)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
```

## Predicate

Sometimes we need to narrow general stack trace collection to a few threads that satisfy some predicate<sup>198</sup>, for example, all threads with kernel time spent greater than some value, or all suspended threads, or all threads that wait for a specific synchronization object type. This can be implemented using WinDbg scripts<sup>199</sup> and debugger extensions.

## Comments

An example here can be service tags in TEB.SubProcessTag to identify threads from specific service in *svchost.exe*.

To list 32-bit stack traces from the specific WOW64 process:

```
!for_each_thread ".thread @#Thread; r $t0 = @#Thread; .if (@@c+(((nt!_KTHREAD *)@$t0)->Process) == ProcessAddress) {.thread /w @#Thread; .reload; kv 256; .effmach AMD64 }"
```

<sup>198</sup> [http://en.wikipedia.org/wiki/Predicate\\_\(mathematical\\_logic\)](http://en.wikipedia.org/wiki/Predicate_(mathematical_logic))

<sup>199</sup> Two WinDbg Scripts That Changed the World, Memory Dump Analysis Anthology, Volume 7, page 32

## Unmanaged Space

Sometimes a problem can be identified not from a single **Stack Trace** pattern but from **Stack Trace Collection**.

These include **Coupled Processes** (page 149), **Procedure Call Chains**<sup>200</sup> and **Blocked Threads** (page 80). Here we only discuss various methods to list stack traces.

- Process dumps including various process minidumps:

`~*kv` command lists all process threads.

`!findstack module[!symbol] 2` command filters all stack traces to show ones containing *module* or *module!symbol*.

`!uniqstack` command.

- Kernel minidumps:

have only one problem thread. `kv` command or its variant is sufficed.

- Kernel and complete memory dumps:

`!process 0 3f` command lists all processes and their threads including user space process thread stacks for complete memory dumps. This command is valid for Windows XP and later. For older systems, we can use WinDbg scripts.

`!stacks 2 [module[!symbol]]` command shows kernel mode stack traces, and we can filter the output based on *module* or *module!symbol*. Filtering is valid only for crash dumps from Windows XP and later systems.

`~[ProcessorN]s;.reload /user;kv` command sequence shows the stack trace for the running thread on the specified processor.

The processor change command is illustrated in this example:

```
0: kd> ~2s

2: kd> k
ChildEBP RetAddr
eb42bd58 00000000 nt!KiIdleLoop+0x14
```

---

<sup>200</sup> This is a pattern we may add in the future

```
2: kd> ~1s;.reload /user;k
Loading User Symbols
...
ChildEBP RetAddr
be4f8c30 eb091f43 i8042prt!I8xProcessCrashDump+0x53
be4f8c8c 8046bfe2 i8042prt!I8042KeyboardInterruptService+0x15d
be4f8c8c 8049470f nt!KiInterruptDispatch+0x32
be4f8d54 80468389 nt!NtSetEvent+0x71
be4f8d54 77f8290a nt!KiSystemService+0xc9
081cffec 77f88266 ntdll!ZwSetEvent+0xb
081cff0c 77f881b1 ntdll!RtlpUnWaitCriticalSection+0x1b
081cff14 1b00c7d1 ntdll!RtlLeaveCriticalSection+0x1d
081cff4c 1b0034da msjet40!Database::ReadPages+0x81
081cffb4 7c57b3bc msjet40!System::WorkerThread+0x115
081cffec 00000000 KERNEL32!BaseThreadStart+0x52
```

Example of **!findstack** command (process dump):

```
0:000> !findstack kernel32!RaiseException 2
Thread 000, 1 frame(s) match
* 00 0013b3f8 72e8d3ef kernel32!RaiseException+0x53
  01 0013b418 72e9a26b msxml3!Exception::raiseException+0x5f
  02 0013b424 72e8ff00 msxml3!Exception::_throwError+0x22
  03 0013b46c 72e6abaa msxml3!COMSafeControlRoot::getBaseUrl+0x3d
  04 0013b4bc 72e6a888 msxml3!Document::loadXML+0x82
  05 0013b510 64b73a9b msxml3!DOMDocumentWrapper::loadXML+0x5a
  06 0013b538 64b74eb6 iepeers!CPersistUserData::initXMLCache+0xa6
  07 0013b560 77d0516e iepeers!CPersistUserData::load+0xfc
  08 0013b57c 77d14abf oleaut32!DispCallFunc+0x16a
...
...
...
  66 0013fec8 0040243d shdocvw!IEWinMain+0x129
  67 0013ff1c 00402744 iexplore!WinMain+0x316
  68 0013ffc0 77e6f23b iexplore!WinMainCRTStartup+0x182
  69 0013fff0 00000000 kernel32!BaseProcessStart+0x23
```

Example of **!stacks** command (kernel dump):

```
2: kd> !stacks 2 nt!PspExitThread
Proc.Thread .Thread Ticks ThreadState Blocker
[8a390818 System]

[8a1bbbf8 smss.exe]

[8a16cbf8 csrss.exe]

[89c14bf0 winlogon.exe]

[89ddaa630 services.exe]

[89c23af0 lsass.exe]
```

[8a227470 svchost.exe]  
[89f03bb8 svchost.exe]  
[89de3820 svchost.exe]  
[89d09b60 svchost.exe]  
[89c03530 ccEvtMgr.exe]  
[89b8f4f0 ccSetMgr.exe]  
[89dfe8c0 SPBBCSvc.exe]  
[89c9db18 svchost.exe]  
[89dfa268 spoolsv.exe]  
[89dfa6b8 msdtc.exe]  
[89df38f0 CpSvc.exe]  
[89d97d88 DefWatch.exe]  
[89e04020 IBMSPSVC.EXE]  
[89b54710 IBMSPREM.EXE]  
[89d9e4b0 IBMSPREM.EXE]  
[89c2c4e8 svchost.exe]  
[89d307c0 SavRoam.exe]  
[89bfcd88 Rtvscan.exe]  
[89b53b60 uphclean.exe]  
[89c24020 AgentSVC.exe]  
[89d75b60 sAgainst.exe]  
[89cf0d88 CdfSvc.exe]  
[89d87020 cdmsvc.exe]  
[89dafd88 ctxxmlss.exe]  
[89d8dd88 encsvc.exe]

[89d06d88 ImaSrv.exe]  
[89d37b60 mfcom.exe]  
[89c8bb18 SmaService.exe]  
[89d2ba80 svchost.exe]  
[89ce8630 XTE.exe]  
[89b64b60 XTE.exe]  
[89b7c680 ctxcpusched.exe]  
[88d94a88 ctxcpuusync.exe]  
[89ba5418 unsecapp.exe]  
[89d846e0 w miprvse.exe]  
[89cda9d8 ctxwmisvc.exe]  
[88d6cb78 logon.scr]  
[88ba0a70 csrss.exe]  
[88961968 winlogon.exe]  
[8865f740 rdclip.exe]  
[8858db20 wfshell.exe]  
[88754020 explorer.exe]  
[88846d88 BacsTray.exe]  
[886b6180 ccApp.exe]  
[884bc020 fppdis3a.exe]  
[885cb350 ctfmon.exe]  
[888bb918 cscript.exe]  
[8880b3c8 cscript.exe]  
[88ad2950 csrss.exe]  
**b68.00215c 88930020 0000000 RUNNING nt!KeBugCheckEx+0x1b  
nt!MiCheckSessionPoolAllocations+0xe3  
nt!MiDereferenceSessionFinal+0x183**

```
nt!MmCleanProcessAddressSpace+0x6b
nt!PspExitThread+0x5f1
nt!PspTerminateThreadByPointer+0x4b
nt!PspSystemThreadStartup+0x3c
nt!KiThreadStartup+0x16

[88629310 winlogon.exe]

[88a4d9b0 csrss.exe]

[88d9f8b0 winlogon.exe]

[88cd5840 wfshell.exe]

[8a252440 OUTLOOK.EXE]

[8a194bf8 WINWORD.EXE]

[88aabbd20 ctfmon.exe]

[889ef440 EXCEL.EXE]

[88bec838 HogiaGUI2.exe]

[88692020 csrss.exe]

[884dd508 winlogon.exe]

[88be1d88 wfshell.exe]

[886a7d88 OUTLOOK.EXE]

[889baa70 WINWORD.EXE]

[8861e3d0 ctfmon.exe]

[887bbb68 EXCEL.EXE]

[884e4020 csrss.exe]

[8889d218 winlogon.exe]

[887c8020 wfshell.exe]
```

Threads Processed: 1101

What if we have a list of processes from a complete memory dump by using **!process 0 0** command and we want to interrogate the specific process? In this case, we need to switch to that process and reload user space symbol files (**.process /r /p address**).

There is also a separate command to reload user space symbol files any time (**.reload /user**).

After switching, we can list threads (**!process address**), dump or search process virtual memory. For example:

```
1: kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS 890a3320 SessionId: 0 Cid: 0008 Peb: 00000000 ParentCid: 0000
  DirBase: 00030000 ObjectTable: 890a3e08 TableSize: 405.
  Image: System

PROCESS 889dfdf60 SessionId: 0 Cid: 0144 Peb: 7ffdf000 ParentCid: 0008
  DirBase: 0b9e7000 ObjectTable: 889fdb48 TableSize: 212.
  Image: SMSS.EXE

PROCESS 890af020 SessionId: 0 Cid: 0160 Peb: 7ffdf000 ParentCid: 0144
  DirBase: 0ce36000 ObjectTable: 8898e308 TableSize: 747.
  Image: CSRSS.EXE

PROCESS 8893d020 SessionId: 0 Cid: 0178 Peb: 7ffdf000 ParentCid: 0144
  DirBase: 0d33b000 ObjectTable: 890ab4c8 TableSize: 364.
  Image: WINLOGON.EXE

PROCESS 88936020 SessionId: 0 Cid: 0194 Peb: 7ffdf000 ParentCid: 0178
  DirBase: 0d7d5000 ObjectTable: 88980528 TableSize: 872.
  Image: SERVICES.EXE

PROCESS 8897f020 SessionId: 0 Cid: 01a0 Peb: 7ffdf000 ParentCid: 0178
  DirBase: 0d89d000 ObjectTable: 889367c8 TableSize: 623.
  Image: LSASS.EXE

1: kd> .process /r /p 8893d020
Implicit process is now 8893d020
Loading User Symbols
...

1: kd> !process 8893d020
PROCESS 8893d020 SessionId: 0 Cid: 0178 Peb: 7ffdf000 ParentCid: 0144
  DirBase: 0d33b000 ObjectTable: 890ab4c8 TableSize: 364.
  Image: WINLOGON.EXE
  VadRoot 8893a508 Clone 0 Private 1320. Modified 45178. Locked 0.
  DeviceMap 89072448
    Token e392f8d0
    ElapsedTime 9:54:06.0882
    UserTime 0:00:00.0071
    KernelTime 0:00:00.0382
    QuotaPoolUsage[PagedPool] 34828
    QuotaPoolUsage[NonPagedPool] 43440
    Working Set Sizes (now,min,max) (737, 50, 345) (2948KB, 200KB, 1380KB)
    PeakWorkingSetSize 2764
    VirtualSize 46 Mb
    PeakVirtualSize 52 Mb
    PageFaultCount 117462
    MemoryPriority FOREGROUND
```

BasePriority	13
CommitCharge	1861

THREAD 8893dda0 Cid 178.15c Teb: 7ffde000 Win32Thread: a2034908 WAIT: (WrUserRequest) UserMode Non-Alertable

8893bee0 SynchronizationEvent  
Not impersonating  
Owning Process 8893d020

Wait Start TickCount 29932455 Elapsed Ticks: 7  
Context Switch Count 28087 LargeStack  
UserTime 0:00:00.0023  
KernelTime 0:00:00.0084  
Start Address winlogon!WinMainCRTStartup (0x0101cbb0)  
Stack Init eb1b0000 Current eb1afcc8 Base eb1b0000 Limit eb1ac000 Call 0  
Priority 15 BasePriority 15 PriorityDecrement 0 DecrementCount 0

ChildEBP RetAddr

```

eb1afce0 8042d893 nt!KiSwapThread+0x1b1
eb1afd08 a00019c2 nt!KeWaitForSingleObject+0xa3
eb1afd44 a0013993 win32k!xxxSleepThread+0x18a
eb1afd54 a001399f win32k!xxxWaitMessage+0xe
eb1afd5c 80468389 win32k!NtUserWaitMessage+0xb
eb1afd5c 77e58b53 nt!KiSystemService+0xc9
0006fdd0 77e33630 USER32!NtUserWaitMessage+0xb
0006fe04 77e44327 USER32!DialogBox2+0x216
0006fe28 77e38d37 USER32!InternalDialogBox+0xd1
0006fe48 77e39eba USER32!DialogBoxIndirectParamAorW+0x34
0006fe6c 01011749 USER32!DialogBoxParamW+0x3d
0006fea8 01018bd3 winlogon!TimeoutDialogBoxParam+0x27
0006fee0 76b93701 winlogon!WlxDialogBoxParam+0x7b
0006ff08 010164c6 3rdPartyGINA!WlxDisplaySASNotice+0x43
0006ff20 01014960 winlogon!MainLoop+0x96
0006ff58 0101cd06 winlogon!WinMain+0x37a
0006fff4 00000000 winlogon!WinMainCRTStartup+0x156

```

THREAD 88980020 Cid 178.188 Teb: 7ffdc000 Win32Thread: 00000000 WAIT: (DelayExecution) UserMode Alertable

88980108 NotificationTimer  
Not impersonating  
Owning Process 8893d020

Wait Start TickCount 29930810 Elapsed Ticks: 1652  
Context Switch Count 15638  
UserTime 0:00:00.0000  
KernelTime 0:00:00.0000  
Start Address KERNEL32!BaseThreadStartThunk (0x7c57b740)  
Win32 Start Address ntdll!RtlpTimerThread (0x77faa02d)  
Stack Init bf6f7000 Current bf6f6cc4 Base bf6f7000 Limit bf6f4000 Call 0  
Priority 13 BasePriority 13 PriorityDecrement 0 DecrementCount 0

ChildEBP RetAddr

```

bf6f6cdc 8042d340 nt!KiSwapThread+0x1b1
bf6f6d04 8052aac9 nt!KeDelayExecutionThread+0x182
bf6f6d54 80468389 nt!NtDelayExecution+0x7f
bf6f6d54 77f82831 nt!KiSystemService+0xc9
00bfff9c 77f842c4 ntdll!NtDelayExecution+0xb

```

```

00bffffb4 7c57b3bc ntdll!RtlpTimerThread+0x42
00bfffec 00000000 KERNEL32!BaseThreadStart+0x52

1: kd> dds 0006fee0
0006fee0 0006ff08
0006fee4 76b93701 3rdPartyGINA!WlxDisplaySASNotice+0x43
0006fee8 000755e8
0006fec0 76b90000 3rdParty
0006fef0 00000578
0006fef4 00000000
0006fef8 76b9370b 3rdParty!WlxDisplaySASNotice+0x4d
0006fefc 0008d0e0
0006ff00 00000008
0006ff04 00000080
0006ff08 0006ff20
0006ff0c 010164c6 winlogon!MainLoop+0x96
0006ff10 0008d0e0
0006ff14 5ffa0000
0006ff18 000755e8
0006ff1c 00000000
0006ff20 0006ff58
0006ff24 01014960 winlogon!WinMain+0x37a
0006ff28 000755e8
0006ff2c 00000005
0006ff30 00072c9c
0006ff34 00000001
0006ff38 000001bc
0006ff3c 00000005
0006ff40 00000001
0006ff44 0000000d
0006ff48 00000000
0006ff4c 00000000
0006ff50 00000000
0006ff54 0000ffe4
0006ff58 0006ffff
0006ff5c 0101cd06 winlogon!WinMainCRTStartup+0x156

```

We can also filter stacks that belong to processes having the same module name, for example, *svchost.exe*<sup>201</sup>.

Sometimes the collection of all stack traces from all threads in the system can disprove or decrease the plausibility of the hypothesis that some module is involved. In one case, the customer claimed that the specific driver was involved in the server freeze. However, there was no such module found in all thread stacks.

## Comments

---

There are also commands **!sprocess** and **!for\_each\_thread**.

---

<sup>201</sup> Filtering Processes, Memory Dump Analysis Anthology, Volume 1, page 220

## Stack Trace Set

If **Stack Trace Collection** (page 943) pattern covers all thread stack traces from a memory dump, this pattern covers only unique non-duplicated thread stack traces differing, for example, in stack frame modules and function names. In user process memory dumps it is **!uniqstack** WinDbg command (don't forget that command has optional parameters, for example, **-v** to simulate verbose **~\*kv** output):

```
0:000> ~
. 0 Id: f00.f04 Suspend: 0 Teb: 7efdd000 Unfrozen
1 Id: f00.f18 Suspend: 1 Teb: 7efda000 Unfrozen
2 Id: f00.f1c Suspend: 1 Teb: 7efd7000 Unfrozen

0:000> ~*kc

. 0 Id: f00.f04 Suspend: 0 Teb: 7efdd000 Unfrozen

ntdll!NtWaitForMultipleObjects
KERNELBASE!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjectsExImplementation
kernel32!WaitForMultipleObjects
kernel32!WerFaultInternal
kernel32!WerReportFault
kernel32!BaseReportFault
kernel32!UnhandledExceptionFilter
ntdll!__RtlUserThreadStart
ntdll!_EH4_CallFilterFunc
ntdll!_except_handler4
ntdll!ExecuteHandler2
ntdll!ExecuteHandler
ntdll!KiUserExceptionDispatcher
KERNELBASE!DebugBreak
ApplicationK!main
ApplicationK!__tmainCRTStartup
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

1 Id: f00.f18 Suspend: 1 Teb: 7efda000 Unfrozen

ntdll!NtDelayExecution
KERNELBASE!SleepEx
KERNELBASE!Sleep
kernel32!WerFault
kernel32!BaseReportFault
kernel32!UnhandledExceptionFilter
ntdll!__RtlUserThreadStart
ntdll!_EH4_CallFilterFunc
ntdll!_except_handler4
ntdll!ExecuteHandler2
ntdll!ExecuteHandler
ntdll!KiUserExceptionDispatcher
ApplicationK!thread_two
```

```
Application!_callthreadstart
Application!_threadstart
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

2 Id: f00.f1c Suspend: 1 Teb: 7efd7000 Unfrozen

ntdll!NtDelayExecution
KERNELBASE!SleepEx
KERNELBASE!Sleep
kernel32!WerFault
kernel32!BaseFault
kernel32!UnhandledExceptionFilter
ntdll!__RtlUserThreadStart
ntdll!_EH4_CallFilterFunc
ntdll!_except_handler4
ntdll!ExecuteHandler2
ntdll!ExecuteHandler
ntdll!KiUserExceptionDispatcher
Application!thread_two
Application!_callthreadstart
Application!_threadstart
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

0:000> !uniqstack
Processing 3 threads, please wait
```

```
. 0 Id: f00.f04 Suspend: 0 Teb: 7efdd000 Unfrozen
Start: Application!mainCRTStartup (013a137c)
Priority: 0 Priority class: 32 Affinity: 3
ChildEBP RetAddr
0037f1a4 770d0bdd ntdll!NtWaitForMultipleObjects+0x15
0037f240 7529162d KERNELBASE!WaitForMultipleObjectsEx+0x100
0037f288 75291921 kernel32!WaitForMultipleObjectsExImplementation+0xe0
0037f2a4 752b9b2d kernel32!WaitForMultipleObjects+0x18
0037f310 752b9bc4 kernel32!WerFaultInternal+0x186
0037f324 752b98f8 kernel32!WerReportFault+0x70
0037f334 752b9875 kernel32!BaseFault+0x20
0037f3c0 77b10df7 kernel32!UnhandledExceptionFilter+0x1af
0037f3c8 77b10cd4 ntdll!__RtlUserThreadStart+0x62
0037f3dc 77b10b71 ntdll!_EH4_CallFilterFunc+0x12
0037f404 77ae6ac9 ntdll!_except_handler4+0x8e
0037f428 77ae6a9b ntdll!ExecuteHandler2+0x26
0037f4d8 77ab010f ntdll!ExecuteHandler+0x24
0037f4d8 770d280c ntdll!KiUserExceptionDispatcher+0xf
0037f824 013a1035 KERNELBASE!DebugBreak+0x2
0037f828 013a1325 Application!main+0x25
0037f870 75293677 Application!_tmainCRTStartup+0xfb
0037f87c 77ad9f02 kernel32!BaseThreadInitThunk+0xe
0037f8bc 77ad9ed5 ntdll!__RtlUserThreadStart+0x70
0037f8d4 00000000 ntdll!_RtlUserThreadStart+0x1b
```

```
. 1 Id: f00.f18 Suspend: 1 Peb: 7efda000 Unfrozen
Start: ApplicationK!_threadstart (013a10d1)
Priority: 0 Priority class: 32 Affinity: 3
ChildEBP RetAddr
0080f9ac 770d31bb ntdll!NtDelayExecution+0x15
0080fa14 770d3a8b KERNELBASE!SleepEx+0x65
0080fa24 752d28dd KERNELBASE!Sleep+0xf
0080fa38 752b98f8 kernel32!WerFault+0x3f
0080fa48 752b9875 kernel32!BaseReportFault+0x20
0080fad4 77b10df7 kernel32!UnhandledExceptionFilter+0x1af
0080fadc 77b10cd4 ntdll!__RtlUserThreadStart+0x62
0080faf0 77b10b71 ntdll!_EH4_CallFilterFunc+0x12
0080fb18 77ae6ac9 ntdll!_except_handler4+0x8e
0080fb3c 77ae6a9b ntdll!ExecuteHandler2+0x26
0080fbec 77ab010f ntdll!ExecuteHandler+0x24
0080fbec 013a1000 ntdll!KiUserExceptionDispatcher+0xf
0080ff38 013a10ab ApplicationK!thread_two
0080ff70 013a1147 ApplicationK!_callthreadstart+0x1b
0080ff78 75293677 ApplicationK!_threadstart+0x76
0080ff84 77ad9f02 kernel32!BaseThreadInitThunk+0xe
0080ffc4 77ad9ed5 ntdll!__RtlUserThreadStart+0x70
0080ffdc 00000000 ntdll!_RtlUserThreadStart+0x1b
```

Total threads: 3

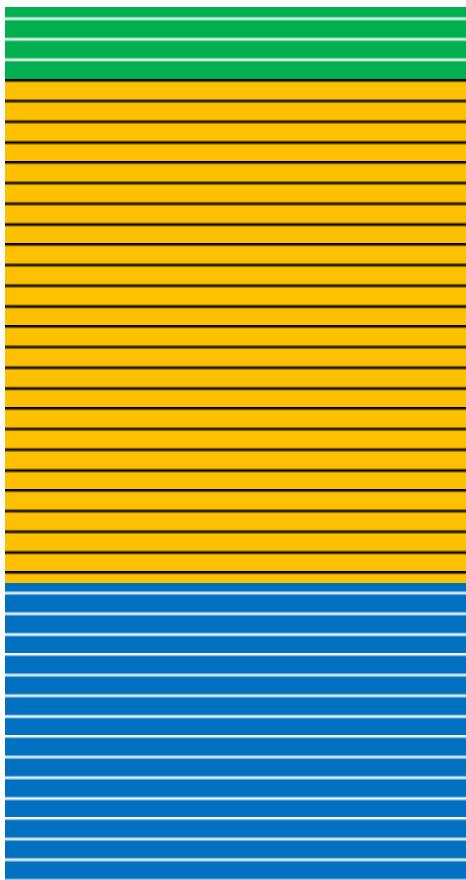
Duplicate callstacks: [1](#)(windbg thread #s follow):

2

Generally, any property can be chosen to form such a set from a collection of stack traces.

## Stack Trace Signature

Stack traces resemble functions: they have prolog, body, and epilog. Frame trace is also similar to trace **Partition** analysis pattern<sup>202</sup>. Bottom stack subtrace plays the role of prolog, for example, thread initialization and RPC call stub dispatch. Middle stack subtrace plays the role of body or core, for example, application specific function calls invoked by RPC. Top stack subtrace plays the role of epilog, for example, system calls. Such stack trace partition is useful for stack trace matching, especially when symbols are not available. In such a case **Stack Trace Signature** of module names and their frame counts may help (together with **Crash Signature** where appropriate, page 153):



T<name<sub>i</sub>, p>

M<name<sub>i</sub>, p>

M<name<sub>j</sub>, r>

M<name<sub>k</sub>, s>

...

B<name<sub>i</sub>, p>

B<name<sub>j</sub>, r>

...

<sup>202</sup> Memory Dump Analysis Anthology, Volume 5, page 299

The following stack trace may be split into TMB (pronounced TOMB):

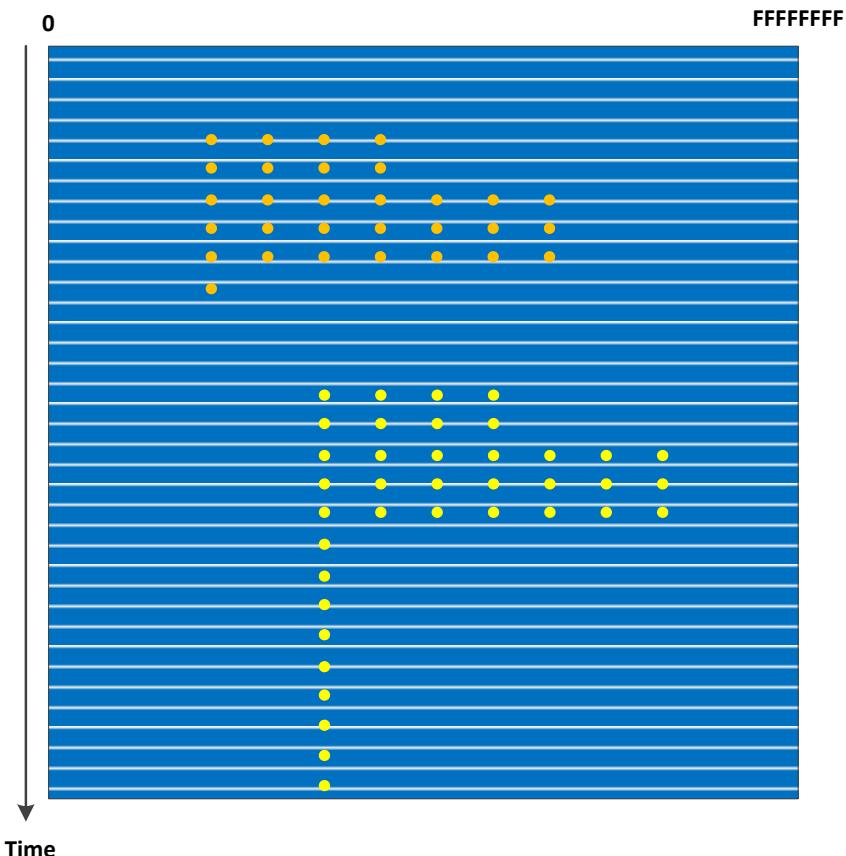
```
0:001> kc
# Call Site
00 ntdll!RtlEnterCriticalSection
01 ModuleA
02 ModuleA
03 ModuleA
04 ModuleA
05 ModuleA
06 ModuleA
07 ModuleA
08 ModuleA
09 ModuleA
0a ModuleA
0b ModuleA
0c ModuleA
0d ModuleA
0e ModuleA
0f ModuleA
10 rpcrt4!Invoke
11 rpcrt4!NdrStubCall12
12 rpcrt4!NdrServerCall12
13 rpcrt4!DispatchToStubInCNoAvrf
14 rpcrt4!RPC_INTERFACE::DispatchToStubWorker
15 rpcrt4!RPC_INTERFACE::DispatchToStub
16 rpcrt4!RPC_INTERFACE::DispatchToStubWithObject
17 rpcrt4!LRPC_SCALL::DispatchRequest
18 rpcrt4!LRPC_SCALL::HandleRequest
19 rpcrt4!LRPC_SASSOCIATION::HandleRequest
1a rpcrt4!LRPC_ADDRESS::HandleRequest
1b rpcrt4!LRPC_ADDRESS::ProcessIO
1c rpcrt4!LrpclIoComplete
1d ntdll!TpAlpcpExecuteCallback
1e ntdll!TpWorkerThread
1f kernel32!BaseThreadInitThunk
20 ntdll!RtlUserThreadStart
```

It has the following signature:

```
T<ntdll,1>M<ModuleA,15>B<rpcrt4,13>B<ntdll,2>B<kernel32,1>B<ntdll,1>
```

## Stack Trace Surface

The advent of virtual machines, the possibility of saving complete memory snapshots without interruption, and the ability to quickly convert such snapshots into a debugger readable memory dump format such as in the case of VMware allows to study how **Stack Trace Collections** (page 943) and **Wait Chains** (page 1092) change over time in complex problem scenarios. Such **Stack Trace Surface** may also show service restarts if PID changes for processes of interest. We call this pattern by analogy with a memory dump surface where each line corresponds to an individual memory snapshot with coordinates from 0 to the highest address:



In the case of orbifold memory space<sup>203</sup>, we have a 3D volume (we may call it 3D orbifold).

---

<sup>203</sup> Dictionary of Debugging: Orbifold Memory Space <http://www.dumpanalysis.org/blog/index.php/2011/02/16/dictionary-of-debugging-orbifold-memory-space/>

## Step Dumps

It is common to get dozens of process memory dumps saved sequentially, for example, after each second. Then we can first analyze memory dumps corresponding to changes in their file sizes ignoring plateaus to save analysis time. This pattern is called by analogy with step functions<sup>204</sup>. For example, we have this dump set with comments from WinDbg analysis sessions (it was reported that an application was freezing for some time until its disappearance from a user screen):

```
C:\MemoryDumps>dir
[...]
12/30/2012  8:33 PM  218,252,862 AppA-1.dmp // normal
12/30/2012  8:34 PM  218,541,762 AppA-2.dmp // slightly increased CPU consumption for thread #11
12/30/2012  8:37 PM  218,735,848 AppA-3.dmp // spiking thread #11
12/30/2012  8:38 PM  218,735,848 AppA-4.dmp
12/30/2012  8:38 PM  218,735,848 AppA-5.dmp
12/30/2012  8:39 PM  218,735,848 AppA-6.dmp
12/30/2012  8:39 PM  218,735,848 AppA-7.dmp
12/30/2012  8:39 PM  218,735,848 AppA-8.dmp
12/30/2012  8:40 PM  218,735,848 AppA-9.dmp
12/30/2012  8:40 PM  218,735,848 AppA-10.dmp
12/30/2012  8:41 PM  218,735,848 AppA-11.dmp
12/30/2012  8:41 PM  218,735,848 AppA-12.dmp // spiking thread #11
12/30/2012  8:42 PM  219,749,040 AppA-13.dmp // spiking thread #11, another thread blocked in ALPC
12/30/2012  8:42 PM  219,048,842 AppA-14.dmp // only one thread left
[...]
```

---

<sup>204</sup> [http://en.wikipedia.org/wiki/Step\\_function](http://en.wikipedia.org/wiki/Step_function)

## Stored Exception

This pattern is mostly useful when an exception thread is not present in the dump like in this rare example:

```
ERROR: Unable to find system thread 9B7E
ERROR: The thread being debugged has either exited or cannot be accessed
ERROR: Many commands will not work properly
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
ERROR: Exception C0000005 occurred on unknown thread 9B7E
(95f4.9b7e): Access violation - code c0000005 (first/second chance not available)
```

.ecxr command will not work here, but the exception record is available via .exr command:

```
0:???> .exr -1
ExceptionAddress: 08a9ae18 (DllB.dll+0x001cae18)
ExceptionCode:    c0000005 (Access violation)
ExceptionFlags:   00000001
NumberParameters: 1
Parameter[0]:     00000008
```

## String Hint

This pattern covers traces of ASCII and UNICODE strings that look suspicious such as website, password and HTTP forms or strange names that intuitively shouldn't be present according to the purpose of a module or its container process (the example is taken from Victimware presentation case study<sup>205</sup>):

```
0:005> s-sa 00040000 L1d000
0004004d  "!This program cannot be run in D"
0004006d  "OS mode."
00040081  "3y@"
000400b8  "Rich"
000401d0  ".text"
000401f7  "` .rdata"
0004021f  "@.data"
00040248  ".reloc"
[...]
00054018  "GET /stat?uptime=%d&downlink=%d&"
00054038  "uplink=%d&id=%s&statpass=%s&comm"
00054058  "ent=%s HTTP/1.0"
000540d8  "ftp://%s:%s@%s:%d"
000540fc  "Accept-Encoding:"
00054118  "Accept-Encoding:"
00054130  "0123456789ABCDEF"
00054144  "://"
00054160  "POST %s HTTP/1.0"
00054172  "Host: %s"
0005417c  "User-Agent: %s"
0005418c  "Accept: text/html"
0005419f  "Connection: Close"
000541b2  "Content-Type: application/x-www-"
000541d2  "form-urlencoded"
000541e3  "Content-Length: %d"
000541fc  "id="
00054208  "POST %s HTTP/1.1"
0005421a  "Host: %s"
00054224  "User-Agent: %s"
00054234  "Accept: text/html"
00054247  "Connection: Close"
0005425a  "Content-Type: application/x-www-"
0005427a  "form-urlencoded"
0005428b  "Content-Length: %d"
000542a4  "id=%s&base="
000542b8  "id=%s&brw=%d&type=%d&data="
000542d8  "POST %s HTTP/1.1"
000542ea  "Host: %s"
000542f4  "User-Agent: %s"
00054304  "Accept: text/html"
00054317  "Connection: Close"
0005432a  "Content-Type: application/x-www-"
```

---

<sup>205</sup> <http://www.patterndiagnostics.com/Victimware-materials>

0005434a "form-urlencoded"  
0005435b "Content-Length: %d"  
00054378 "id=%s&os=%s&plist="  
00054390 "POST %s HTTP/1.1"  
000543a2 "Host: %s"  
000543ac "User-Agent: %s"  
000543bc "Accept: text/html"  
000543cf "Connection: Close"  
000543e2 "Content-Type: application/x-www-"  
00054402 "form-urlencoded"  
00054413 "Content-Length: %d"  
00054430 "id=%s&data=%s"  
00054440 "POST %s HTTP/1.1"  
00054452 "Host: %s"  
0005445c "User-Agent: %s"  
0005446c "Accept: text/html"  
0005447f "Connection: Close"  
00054492 "Content-Type: application/x-www-"  
000544b2 "form-urlencoded"  
000544c3 "Content-Length: %d"  
000544e0 "GET %s HTTP/1.0"  
000544f1 "Host: %s"  
000544fb "User-Agent: %s"  
0005450b "Connection: close"  
00054528 "POST /get/scr.html HTTP/1.0"  
00054545 "Host: %s"  
0005454f "User-Agent: %s"  
0005455f "Connection: close"  
00054572 "Content-Length: %d"  
00054586 "Content-Type: multipart/form-data"  
000545a6 "a; boundary=-----"  
000545c6 "-----%d"  
000545d4 "-----%d"  
000545f8 "%sContent-Disposition: form-data"  
00054618 "; name='id'"  
00054630 "%sContent-Disposition: form-data"  
00054650 "; name='screen'; filename='%d'"  
00054670 "Content-Type: application/octet-"  
00054690 "stream"  
000546ac "%s failed with error %d: %s"  
000546d8 "BlackwoodPRO"  
000546e8 "FinamDirect"  
000546f4 "GrayBox"  
000546fc "MbtPRO"  
00054704 "Laser"  
0005470c "LightSpeed"  
00054718 "LTGroup"  
00054720 "Mbt"  
00054724 "ScotTrader"  
00054730 "SaxoTrader"  
00054740 "Program: %s"  
0005474f "Username: %s"  
0005475e "Password: %s"  
0005476d "AccountNO: %s"  
[ ... ]

## String Parameter

This analysis pattern is frequently useful when found on a function call stack. The trivial case is when a function parameter is a pointer to an ASCII or a Unicode string (**da** and **du** WinDbg commands). The more interesting case is when we have a function that takes pointers to a structure that has string fields (**dpa** and **dpu** commands), for example:

```
0:018> kv 100
ChildEBP RetAddr Args to Child
00de8c7c 7739bf53 7739610a 07750056 00000000 ntdll!KiFastSystemCallRet
00de8cb4 7738965e 00080126 07750056 00000001 user32!NtUserWaitMessage+0xc
00de8cdc 7739f762 77380000 0012b238 07750056 user32!InternalDialogBox+0xd0
00de8f9c 7739f047 00de90f8 00000000 ffffffff user32!SoftModalMessageBox+0x94b
00de90ec 7739eec9 00de90f8 00000028 07750056 user32!MessageBoxWorker+0x2ba
00de9144 773d7d0d 07750056 0015cd68 00132a60 user32!MessageBoxTimeoutW+0x7a
00de9178 773c42c8 07750056 00de923f 00de91ec user32!MessageBoxTimeoutA+0x9c
00de9198 773c42a4 07750056 00de923f 00de91ec user32!MessageBoxExA+0x1b
00de91b4 6dfcf8c2 07750056 00de923f 00de91ec user32!MessageBoxA+0x45
00de99f0 6dfcfad2 00de9285 00de9a1c 77bc6cd5 compstui!FilterException+0x174
00dead94 7739b6e3 0038010e 00000110 00000000 compstui!CPSUIPageDlgProc+0xf3
00deadc0 77395f82 6dfcf9df 0038010e 00000110 user32!InternalCallWinProc+0x28
00deae3c 77395e22 0015d384 6dfcf9df 0038010e user32!UserCallDlgProcCheckWow+0x147
00deaee84 7738aaa4 00000000 00000110 00000000 user32!DefDlgProcWorker+0xa8
00deaeb4 77388c01 004673d0 00461130 00000000 user32!SendMessageWorker+0x43e
00deaf6c 77387910 6dfc0000 004673d0 00000404 user32!InternalCreateDialog+0x9cf
00deaf90 7739fb5b 6dfc0000 001621d0 07750056 user32!CreateDialogIndirectParamAorW+0x33
00deafb0 774279a5 6dfc0000 001621d0 07750056 user32!CreateDialogIndirectParamW+0x1b
00deb000 77427abc 02192c78 000ddd08 07750056 comctl32!_CreatePageDialog+0x79
00deb028 77429d12 02192c78 6dff5c30 07750056 comctl32!_CreatePage+0xb1
00deb244 7742b8b6 02192c78 00000001 00290110 comctl32!PageChange+0xcc
00deb604 7742c446 07750056 02192c78 00deb6ec comctl32!InitPropSheetDlg+0xbb8
00deb674 7739b6e3 07750056 00000110 00290110 comctl32!PropSheetDlgProc+0x4cb
00deb6a0 77395f82 7742bf7b 07750056 00000110 user32!InternalCallWinProc+0x28
00deb71c 77395e22 0008c33c 7742bf7b 07750056 user32!UserCallDlgProcCheckWow+0x147
00deb764 7738aaa4 00000000 00000110 00290110 user32!DefDlgProcWorker+0xa8
00deb794 77388c01 004652e0 00461130 00290110 user32!SendMessageWorker+0x43e
00deb84c 77387910 77420000 004652e0 00000100 user32!InternalCreateDialog+0x9cf
00deb870 7739fb5b 77420000 02184be8 00000000 user32!CreateDialogIndirectParamAorW+0x33
00deb890 774ab1c5 77420000 02184be8 00000000 user32!CreateDialogIndirectParamW+0x1b
00deb8d8 7742ca78 77420000 02184be8 00000000 comctl32!SHFusionCreateDialogIndirectParam+0x36
00deb93c 7742cce4 00000000 000000a0 00000000 comctl32!_RealPropertySheet+0x242
00deb954 7742cd05 00deb9b4 00000000 00deb99c comctl32!_PropertySheet+0x146
00deb964 6dfd1178 00deb9b4 000000a0 00deba30 comctl32!PropertySheetW+0xf
00deb99c 6dfcf49b 00deb9b4 0256b3f8 0013fbe0 compstui!PropertySheetW+0x4b
00deba14 6dfd0718 00000000 00134da4 00debae8 compstui!DoComPropSheet+0x2ef
00deba44 6dfd0799 00000000 7307c8da 00debad0 compstui!DoCommonPropertySheetUI+0xe9
00deba5c 730801c5 00000000 7307c8da 00debad0 compstui!CommonPropertySheetUIW+0x17
00debaa4 73080f5d 00000000 7307c8da 00debad0 winspool!CallCommonPropertySheetUI+0x43
00debeec 4f49cdfe 00000000 0218bd84 02277fe8 winspool!PrinterPropertiesNative+0x10c
WARNING: Stack unwind information not available. Following frames may be wrong.
00debef2c 4f4950a5 00deeaa8 00000002 02277fe8 PrintDriverA!DllGetClassObject+0xdb7e
00deee18 4f4904fb 00ca6ee0 00000003 00000001 PrintDriverA!DllGetClassObject+0x5e25
00deee30 18f60282 02277fe8 00ca6ee0 00000003 PrintDriverA!DllGetClassObject+0x127b
00deee58 18f5abce 001042e4 00ca6ee0 00000003 ps5ui!HComOEMPrinterEvent+0x33
```

```

00deee9c 7308218c 00ca6ee0 00000003 00000001 ps5ui!DrvPrinterEvent+0x22e
00deeee8 761543c8 00ca6ee0 00000003 00000001 winspool!SpoolerPrinterEventNative+0x57
00def04 761560d2 00ca6ee0 00000003 00000000 localspl!SplDriverEvent+0x21
00def28 761447f9 00cb2160 00000003 00000000 localspl!PrinterDriverEvent+0x46
00def3f0 76144b12 00000000 00000002 00d12020 localspl!SplAddPrinter+0x5f3
00def41c 74070193 00000000 00000002 00d12020 localspl!LocalAddPrinterEx+0x2e
00def86c 7407025c 00000000 00000002 00d12020 spoolss!AddPrinterExW+0x151
00def888 01007a93 00000000 00000002 00d12020 spoolss!AddPrinterW+0x17
00def8a4 01006772 00000000 00ce74b0 021b6278 spoolsV!YAddPrinter+0x75
00def8c8 77c80355 00000000 00ce74b0 021b6278 spoolsV!RpcAddPrinter+0x37
00def8f0 77ce43e1 0100673b 00defae0 00000005 rpcrt4!Invoke+0x30
00defcf8 77ce45c4 00000000 00000000 000e8584 rpcrt4!NdrStubCall2+0x299
00defd14 77c8013a 000e8584 000d63d8 000e8584 rpcrt4!NdrServerCall2+0x19
00defd48 77c805ef 01002c57 000e8584 00defdec rpcrt4!DispatchToStubInCNoAvrf+0x38
00defd9c 77c80515 00000005 00000000 0100d228 rpcrt4!RPC_INTERFACE::DispatchToStubWorker+0x11f
00defdc0 77c8139e 000e8584 00000000 0100d228 rpcrt4!RPC_INTERFACE::DispatchToStub+0xa3
00defdfc 77c814b2 000e1c48 000d85b8 02154180 rpcrt4!LRPC_SCALL::DealWithRequestMessage+0x42c
00defe20 77c88848 000d85f0 00defe38 000e1c48 rpcrt4!LRPC_ADDRESS::DealWithLRPCRequest+0x127
00eff84 77c88962 00efffac 77c888fd 000d85b8 rpcrt4!LRPC_ADDRESS::ReceiveLotsaCalls+0x430
00eff8c 77c888fd 000d85b8 00000000 00000000 rpcrt4!RecvLotsaCallsWrapper+0xd
00effac 77c7b293 0008b038 00efffec 77e6482f rpcrt4!BaseCachedThreadRoutine+0x9d
00efffb8 77e6482f 000bdba8 00000000 00000000 rpcrt4!ThreadStartRoutine+0x1b
00efffec 00000000 77c7b278 000bdba8 00000000 kernel32!BaseThreadStart+0x34

```

0:018> da **00de923f**  
00de923f “Function address 0x77481456 caus”  
00de925f “ed a protection fault. (exceptio”  
00de927f “n code 0xc0000005).The applicati”  
00de929f “on property sheet page(s) may no”  
00de92bf “t function properly.”

0:018> dpu **00d12020**  
00d12020 00000000  
00d12024 021b6088 “Printer A User B Server C”  
00d12028 00000000  
00d1202c 021b6124 “Remote Printer Address for User C”  
00d12030 021b6190 “Printer Name and Family”  
00d12034 021b61c4 “Printer Client Name”  
00d12038 021b6228 “Printer Location”  
00d1203c 00000000  
00d12040 00000000  
00d12044 021b6264 “Printer Module Name”  
00d12048 00000000  
00d1204c 00000000  
00d12050 021b628c  
00d12054 00008841  
00d12058 00000000  
00d1205c 00000000  
00d12060 00000000  
00d12064 00000000  
[...]

## Suspended Thread

Most of the time threads are not suspended explicitly. If we look at active and waiting threads in the kernel and complete memory dumps their *SuspendCount* member is 0:

```
THREAD 88951bc8 Cid 03a4.0d24 Peb: 7fffaa000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Non-Alertable
  889d6a78 Semaphore Limit 0xffffffff
  88951c40 NotificationTimer
Not impersonating
DeviceMap          e1b80b98
Owning Process    888a9d88 Image: svchost.exe
Wait Start TickCount 12669 Ticks: 5442 (0:00:01:25.031)
Context Switch Count 3
UserTime           00:00:00.000
KernelTime         00:00:00.000
Win32 Start Address RPCRT4!ThreadStartRoutine (0x77c4b0f5)
Start Address kernel32!BaseThreadStartThunk (0x7c8217ec)
Stack Init f482f000 Current f482ec0c Base f482f000 Limit f482c000 Call 0
Priority 10 BasePriority 10 PriorityDecrement 0
ChildEBP RetAddr
f482ec24 80833465 nt!KiSwapContext+0x26
f482ec50 80829a62 nt!KiSwapThread+0x2e5
f482ec98 809226bd nt!KeWaitForSingleObject+0x346
f482ed48 8088978c nt!NtReplyWaitReceivePortEx+0x521
f482ed48 7c9485ec nt!KiFastCallEntry+0xfc (TrapFrame @ f482ed64)
00efff84 77c58792 ntdll!KiFastSystemCallRet
00efff8c 77c5872d RPCRT4!RecvLotsaCallsWrapper+0xd
00efffac 77c4b110 RPCRT4!BaseCachedThreadRoutine+0x9d
00efffb8 7c824829 RPCRT4!ThreadStartRoutine+0x1b
00efffec 00000000 kernel32!BaseThreadStart+0x34

5: kd> dt _KTHREAD 88951bc8
ntdll!_KTHREAD
+0x000 Header      : _DISPATCHER_HEADER
+0x010 MutantListHead : _LIST_ENTRY [ 0x88951bd8 - 0x88951bd8 ]
+0x018 InitialStack : 0xf482f000
+0x01c StackLimit   : 0xf482c000
+0x020 KernelStack   : 0xf482ec0c
+0x024 ThreadLock    : 0
+0x028 ApcState     : _KAPC_STATE
...
...
...
+0x14f FreezeCount  : 0 ''
+0x150 SuspendCount : 0 "
```

We won't find *SuspendCount* in reference stack traces (page 1164). Only when some other thread explicitly suspends another thread, the latter has non-zero suspend count. **Suspended Threads** are excluded from thread scheduling, and, therefore, can be considered as blocked. This might be the sign of a debugger present, for example, where all threads in a process are suspended when a user debugger is processing a debugger event like a breakpoint or access violation exception. In this case **!process 0 3f** command output shows *SuspendCount* value:

```

THREAD 888b8668 Cid 0ca8.1448 Teb: 00000000 Win32Thread: 00000000 WAIT: (Unknown) KernelMode Non-Alertable
SuspendCount 2
  888b87f8 Semaphore Limit 0x2
Not impersonating
DeviceMap          e10028e8
Owning Process    898285b0      Image:       processA.exe
Wait Start TickCount 13456      Ticks: 4655 (0:00:01:12.734)
Context Switch Count 408
UserTime           00:00:00.000
KernelTime         00:00:00.000
Start Address driver!DriverThread (0xf6fb8218)
Stack Init f455b000 Current f455a3ac Base f455b000 Limit f4558000 Call 0
Priority 6 BasePriority 6 PriorityDecrement 0
ChildEBP RetAddr
f455a3c4 80833465 nt!KiSwapContext+0x26
f455a3f0 80829a62 nt!KiSwapThread+0x2e5
f455a438 80833178 nt!KeWaitForSingleObject+0x346
f455a450 8082e01f nt!KiSuspendThread+0x18
f455a498 80833480 nt!KiDeliverApc+0x117
f455a4d0 80829a62 nt!KiSwapThread+0x300
f455a518 f6fb7f13 nt!KeWaitForSingleObject+0x346
f455a548 f4edd457 driver!WaitForSingleObject+0x75
f455a55c f4edcdd8 driver!DeviceWaitForRead+0x19
f455ad90 f6fb8265 driver!InputThread+0x17e
f455adac 80949b7c driver!DriverThread+0x4d
f455addc 8088e062 nt!PspSystemThreadStartup+0x2e
00000000 00000000 nt!KiThreadStartup+0x16

```

```

5: kd> dt _KTHREAD 888b8668
ntdll!_KTHREAD
+0x000 Header        : _DISPATCHER_HEADER
+0x010 MutantListHead : _LIST_ENTRY [ 0x888b8678 - 0x888b8678 ]
+0x018 InitialStack   : 0xf455b000
+0x01c StackLimit     : 0xf4558000
+0x020 KernelStack     : 0xf455a3ac
+0x024 ThreadLock      : 0
...
...
...
+0x14F FreezeCount    : 0 ''
+0x150 SuspendCount    : 2 "

```

This pattern should raise suspicion bar and in some cases coupled with **Special Process** pattern (page 877) can lead to immediate problem identification.

## Swarm of Shared Locks

Sometimes there are so many shared locks on the system that it might point to some problems in subsystems that own them. For example, there are two large swarms of them in this memory dump from a system running 90 user sessions:

```
0: kd> !session
Sessions on machine: 90

0: kd> !locks
**** DUMP OF ALL RESOURCE OBJECTS ****
KD: Scanning for held locks.....

Resource @ nt!CmpRegistryLock (0x808ad4c0)      Shared 210 owning threads
Contention Count = 1432
Threads: 88bf1590-01<*> 8a78a660-01<*> 8a787660-01<*> 8825a3a8-01<*>
         89003358-01<*> 86723b90-01<*> 865bb00-01<*> 89634638-01<*>
         888d9508-01<*> 88da6b48-01<*> 87db9db0-01<*> 86a9e610-01<*>
         89ff7410-01<*> 87450db0-01<*> 86bdedb0-01<*> 86d604c8-01<*>
         88d465d8-01<*> 86c3b6a0-01<*> 87c89020-01<*> 88e73db0-01<*>
         865fe5b0-01<*> 88450020-01<*> 86bd9db0-01<*> 8a73e838-01<*>
         88dc3db0-01<*> 88035708-01<*> 8833a2f0-01<*> 88608350-01<*>
         87aca020-01<*> 87e007c0-01<*> 86ec39b8-01<*> 893be1b8-01<*>
         8671ddb0-01<*> 8679a718-01<*> 89fe34c8-01<*> 86cccd720-01<*>
         881b1db0-01<*> 86771b20-01<*> 86d71db0-01<*> 89574db0-01<*>
         87dfac50-01<*> 86597020-01<*> 874b3488-01<*> 873b59b0-01<*>
         88e792f8-01<*> 878d2430-01<*> 8853d480-01<*> 889e2020-01<*>
         88c36db0-01<*> 8824f990-01<*> 8719b830-01<*> 884ba020-01<*>
         88e1d768-01<*> 89523db0-01<*> 896529f8-01<*> 887e2870-01<*>
         8a022db0-01<*> 867253a0-01<*> 865f0448-01<*> 87d35640-01<*>
         8715d968-01<*> 87ce0c50-01<*> 87d44730-01<*> 86d69aa8-01<*>
         88e5b020-01<*> 88734410-01<*> 898f2b40-01<*> 8a00a510-01<*>
         87e69db0-01<*> 8722b860-01<*> 86d8e308-01<*> 87263c50-01<*>
         8706ddb0-01<*> 892136e8-01<*> 8875b020-01<*> 8833ca48-01<*>
         8a100db0-01<*> 86b77590-01<*> 888bc020-01<*> 865c3db0-01<*>
         89fba910-01<*> 8a789660-01<*> 8670b2a8-01<*> 868737a8-01<*>
         868326d0-01<*> 871cdfaf0-01<*> 8852edb0-01<*> 882b23b8-01<*>
         877e29e0-01<*> 8774f558-01<*> 876aa020-01<*> 89187518-01<*>
         8664b8e0-01<*> 865b4478-01<*> 88135020-01<*> 8686f020-01<*>
         866a0190-01<*> 87316758-01<*> 894dab18-01<*> 87938560-01<*>
         8658f5f0-01<*> 88e54020-01<*> 867f6350-01<*> 89246af8-01<*>
         86801430-01<*> 86db2af0-01<*> 865cf588-01<*> 86ab64f8-01<*>
         8a4a61e8-01<*> 885f3020-01<*> 86ea9af0-01<*> 8a4a7ba8-01<*>
         8a746b08-01<*> 89fc4790-01<*> 87093b10-01<*> 8659bc50-01<*>
         86681db0-01<*> 87102228-01<*> 866145a0-01<*> 866dddb0-01<*>
         86bda990-01<*> 88257db0-01<*> 8687d590-01<*> 867a9db0-01<*>
         89898848-01<*> 8a49b920-01<*> 86596db0-01<*> 8a0f7db0-01<*>
         866c1b40-01<*> 8754e020-01<*> 87fc1428-01<*> 8658c870-01<*>
         880d6a90-01<*> 88be6c50-01<*> 86bbcdb0-01<*> 8a37b8f8-01<*>
         866a13e0-01<*> 873e33d0-01<*> 87d43db0-01<*> 88a5adb0-01<*>
         884a5440-01<*> 883646f0-01<*> 87128020-01<*> 88e1d020-01<*>
         888e6418-01<*> 875c7c50-01<*> 871dd020-01<*> 890d5838-01<*>
         88d061f0-01<*> 88a09428-01<*> 8972f780-01<*> 87325b08-01<*>
         86deb020-01<*> 878b31b8-01<*> 891ac8a8-01<*> 86b234c0-01<*>
```

```

86dd2190-01<*> 875f9db0-01<*> 87bbf200-01<*> 8a1a9c40-01<*>
88628020-01<*> 87919020-01<*> 87c2a660-01<*> 877dc7c0-01<*>
8a08adb0-01<*> 87c0f628-01<*> 87ca9a28-01<*> 8880a210-01<*>
86ec0020-01<*> 88571020-01<*> 8a01edb0-01<*> 88115db0-01<*>
87a9adb0-01<*> 879ecdb0-01<*> 8868ddb0-01<*> 872bcb58-01<*>
884a0100-01<*> 8929f020-01<*> 87087020-01<*> 886e75a8-01<*>
885a5908-01<*> 8762c020-01<*> 89550db0-01<*> 8a554768-01<*>
89f10680-01<*> 87b322e8-01<*> 87cc74d0-01<*> 883ee2d0-01<*>
8956caf8-01<*> 8788f330-01<*> 87d5c320-01<*> 86b99db0-01<*>
876f42e0-01<*> 88e812d0-01<*> 8687cdb0-01<*> 8677a310-01<*>
89711b40-01<*> 89b013a8-01<*> 86abcd0-01<*> 89fd7bb0-01<*>
877c22b0-01<*> 883fc850-01<*> 889e11f8-01<*> 892ff0e0-01<*>
878ac490-01<*> 86de5c50-01<*> 87741db0-01<*> 8679f020-01<*>
880ac6d0-01<*> 86d8fb00-01<*>

```

KD: Scanning for held locks....

Resource @ Ntfs!NtfsData (0xf71665b0) Shared 1 owning threads

Threads: 8a78d660-01<\*>

KD: Scanning for held locks.

Resource @ 0x8a5c7734 Shared 1 owning threads

Contention Count = 507565

NumberOfSharedWaiters = 128

NumberOfExclusiveWaiters = 1

Threads: 894b4db0-01	87c773e0-01	88de7020-01	891c9db0-01
894d2020-01	865af5f8-01	87867340-01	<b>88c964a0-01&lt;*&gt;</b>
88e57c98-01	87ae3020-01	86dbe730-01	88343790-01
871102e8-01	8855f020-01	87c99920-01	8796a318-01
88028db0-01	88ad6610-01	88b73db0-01	89fba3f0-01
87d8bc00-01	86f4c5c8-01	8a028608-01	88c783f0-01
88c138e0-01	89236910-01	896fb78-01	88523600-01
8926f3b0-01	88a49a48-01	87c19750-01	86c88c50-01
88adfad8-01	872b0020-01	87ecab18-01	88b02020-01
875f9b10-01	8755e020-01	86f9fdb0-01	86a1cab8-01
86816858-01	881eedb0-01	894a99f0-01	87c97740-01
8a3bf4b0-01	867765a8-01	8a787660-01	86810330-01
876ad268-01	87af3320-01	865fdb0-01	88eb8230-01
86b0c438-01	881c0230-01	888b67c8-01	883e3210-01
87acbc50-01	873d6648-01	86ed0db0-01	88e2d020-01
89fdadb0-01	8934e830-01	870f89f0-01	8756c5e0-01
878c88d0-01	86fec608-01	88fdb420-01	87fa0628-01
87cad8d8-01	88ee3978-01	86fc49a0-01	875d5020-01
871a5020-01	89667a60-01	87170db0-01	88254ae0-01
8775e408-01	88204db0-01	87989890-01	873b89a8-01
888e6bf8-01	88cc3db0-01	88bf1590-01	879565a0-01
86773db0-01	8731a020-01	88aa7a78-01	8759cdb0-01
87e555f8-01	86de5678-01	86e28020-01	86ec9320-01
86871af0-01	8719cba0-01	8723f820-01	884dac20-01
89249020-01	889da168-01	8900b810-01	8a78d660-01
88cac758-01	892984c8-01	87d0c020-01	87ec050-01
87ad8c90-01	88109aa8-01	86ef5bf0-01	8a78d3f0-01
88d2b020-01	88640db0-01	86fec878-01	895b12d8-01
86dd6708-01	87386930-01	888e34e0-01	86a56c50-01
8815f768-01	886c42a0-01	898f2020-01	87ca3610-01
886dd448-01	86ada210-01	8a37adb0-01	8896c940-01
8800e898-01	8733d4b8-01	865fa358-01	88ae1af0-01

```
868dd020-01
Threads Waiting On Exclusive Access:
 8a78b020
```

Both swarms are grouped around NTFS structures as can be seen from thread stack traces but also have another module in common: *PGP sdk*

```
0: kd> !thread 88bf1590 1f
THREAD 88bf1590 Cid 4354.2338 Peb: 7fffd000 Win32Thread: bc3e88f8 WAIT: (Unknown) KernelMode Non-
Alertable
 8a7a73d8 Semaphore Limit 0x7fffffff
 88bf1608 NotificationTimer
IRP List:
 86fb39d0: (0006,0268) Flags: 00000004 Mdl: 00000000
Not impersonating
DeviceMap          e13c9ca0
Owning Process    869a6d88      Image:       ApplicationA.exe
Wait Start TickCount 15423469      Ticks: 30 (0:00:00:00.468)
Context Switch Count 6465           LargeStack
UserTime           00:00:00.343
KernelTime         00:00:01.062
Win32 Start Address 0x0056f122
Start Address 0x77e617f8
Stack Init 97e9d000 Current 97e9c788 Base 97e9d000 Limit 97e98000 Call 0
Priority 14 BasePriority 8 PriorityDecrement 6
ChildEBP RetAddr
97e9c7a0 8083d5b1 nt!KiSwapContext+0x26
97e9c7cc 8083df9e nt!KiSwapThread+0x2e5
97e9c814 8081e05b nt!KeWaitForSingleObject+0x346
97e9c850 80824ba8 nt!ExpWaitForResource+0xd5
97e9c870 f718a07d nt!ExAcquireResourceSharedLite+0xf5
97e9c884 f717b2eb Ntfs!NtfsAcquireSharedVcb+0x23
97e9c8f0 f717a2e2 Ntfs!NtfsCommonFlushBuffers+0xf5
97e9c954 80840153 Ntfs!NtfsFsdFlushBuffers+0x92
97e9c968 f7272c45 nt!IofCallDriver+0x45
97e9c990 80840153 fltmgr!FltpDispatch+0x6f
97e9c9a4 f6fb1835 nt!IofCallDriver+0x45
WARNING: Stack unwind information not available. Following frames may be wrong.
97e9c9b8 f6fad69a PGP sdk+0x5835
97e9c9c4 80840153 PGP sdk+0x169a
86fb39d0 00000000 nt!IofCallDriver+0x45
```

```

0: kd> !thread 88c964a0 1f
THREAD 88c964a0 Cid 323c.43f0 Teb: 7ffad000 Win32Thread: bc2ceea8 WAIT: (Unknown) KernelMode Non-
Alertable
    88268338 SynchronizationEvent
    88c96518 NotificationTimer
IRP List:
    86dad430: (0006,0268) Flags: 00000404 Mdl: 00000000
Not impersonating
DeviceMap          e16c8eb0
Owning Process     8886ac88      Image: ApplicationB.EXE
Wait Start TickCount 15423352      Ticks: 147 (0:00:00:02.296)
Context Switch Count 1660           LargeStack
UserTime            00:00:00.078
KernelTime          00:00:00.109
Win32 Start Address 0x14225c34
Start Address 0x77e617ec
Stack Init 96835000 Current 96834640 Base 96835000 Limit 96832000 Call 0
Priority 14 BasePriority 8 PriorityDecrement 6
ChildEBP RetAddr
96834658 8083d5b1 nt!KiSwapContext+0x26
96834684 8083df9e nt!KiSwapThread+0x2e5
968346cc 8081e05b nt!KeWaitForSingleObject+0x346
96834708 8082e012 nt!ExpWaitForResource+0xd5
96834728 f714b89b nt!ExAcquireResourceExclusiveLite+0x8d
96834738 f718b194 Ntfs!NtfsAcquirePagingResourceExclusive+0x20
9683493c f718b8d9 Ntfs!NtfsCommonCleanup+0x193
96834aac 80840153 Ntfs!NtfsFsdCleanup+0xcf
96834ac0 f7272c45 nt!IofCallDriver+0x45
96834ae8 80840153 fltmgr!FltpDispatch+0x6f
96834afc f6fb196c nt!IofCallDriver+0x45
WARNING: Stack unwind information not available. Following frames may be wrong.
96834b10 f6fad69a PGPsdk+0x596c
96834b1c 80840153 PGPsdk+0x169a
86dad430 00000000 nt!IofCallDriver+0x45

```

Because no processors are busy except the one that processes the crash dump request via NMI interrupt and there are no ready threads it would be natural to assume that the problem with paging started some time ago, and some checks for 3rd-party volume encryption software are necessary as PGP name of the module suggests:

```

0: kd> lmv m PGPsdk
start   end     module name
f6fac000 f6fb7000  PGPsdk      (no symbols)
Loaded symbol image file: PGPsdk.sys
Image path: \SystemRoot\System32\Drivers\PGPsdk.sys
Image name: PGPsdk.sys
Timestamp:       Wed Jun 09 11:44:04 2004 (40C6E9F4)
CheckSum:        00010F72
ImageSize:       0000B000
Translations:    0000.04b0 0000.04e0 0409.04b0 0409.04e0

```

```
0: kd> !running

System Processors f (affinity mask)
Idle Processors e

    Prcb      Current   Next
0  ffdff120  808a68c0  86841588  .....

0: kd> !thread 808a68c0 1f
THREAD 808a68c0  Cid 0000.0000  Teb: 00000000 Win32Thread: 00000000 RUNNING on processor 0
Not impersonating
Owning Process          808a6b40      Image:           Idle
Wait Start TickCount     0              Ticks: 15423499 (2:18:56:32.171)
Context Switch Count     100782385
UserTime                 00:00:00.000
KernelTime                2 Days 12:18:49.343
Stack Init 808a38b0 Current 808a35fc Base 808a38b0 Limit 808a08b0 Call 0
Priority 0 BasePriority 0 PriorityDecrement 0
ChildEBP RetAddr
808a07bc 80a84df7 nt!KeBugCheckEx+0x1b
808a080c 80834b83 hal!HalHandleNMI+0x1a5
808a080c 80a80853 nt!KiTrap02+0x136 (TrapFrame @ 808a0820)
808a3570 f7659ca2 hal!HalpClockInterrupt+0xff (TrapFrame @ 808a3570)
808a3600 80839b12 intelppm!AcpiC1Idle+0x12
808a3604 00000000 nt!KiIdleLoop+0xa

0: kd> !ready
Processor 0: No threads in READY state
Processor 1: No threads in READY state
Processor 2: No threads in READY state
Processor 3: No threads in READY state
```

## System Object

Certain **System Objects** can be found in object directory and can be useful to see additional system and other product activity. For example, in a complete memory dump we see that *LowCommitCondition* event is signaled:

```
1: kd> !object \KernelObjects
Object: 85a08030 Type: (82b38ed0) Directory
ObjectHeader: 85a08018 (old version)
HandleCount: 0 PointerCount: 19
Directory Object: 85a074c0 Name: KernelObjects

Hash Address Type Name
---- ----- ----
02 82b7b0b8 Event HighCommitCondition
04 82b7b780 Event HighMemoryCondition
10 82b7b178 Event LowNonPagedPoolCondition
11 82b7b138 Event HighNonPagedPoolCondition
17 82b7b0f8 Event LowCommitCondition
20 82b78d08 Event SuperfetchParametersChanged
     82b6eb58 Event BootLoaderTraceReady
23 84bfd58 Session Session0
     82b78c88 Event PrefetchTracesReady
24 84b7d1f8 Session Session1
25 82b78cc8 Event SuperfetchScenarioNotify
     82b7b740 Event LowPagedPoolCondition
26 82b7b1b8 Event HighPagedPoolCondition
     82b7a030 Event MemoryErrors
28 82b78c48 Event SuperfetchTracesReady
32 82b7b7c0 Event LowMemoryCondition
     85a09d00 KeyedEvent CritSecOutOfMemoryEvent
34 82b7b078 Event MaximumCommitCondition

1: kd> dt _DISPATCHER_HEADER 82b7b0f8
ntdll!_DISPATCHER_HEADER
+0x000 Type : 0 ''
+0x001 Abandoned : 0 ''
+0x001 Absolute : 0 ''
+0x001 NpxIrql : 0 ''
+0x001 Signalling : 0 ''
+0x002 Size : 0x4 ''
+0x002 Hand : 0x4 ''
+0x003 Inserted : 0 ''
+0x003 DebugActive : 0 ''
+0x003 DpcActive : 0 ''
+0x000 Lock : 0n262144
+0x004 SignalState : 0n1
+0x008 WaitListHead : _LIST_ENTRY [ 0x82b7b100 - 0x82b7b100 ]
```

If we check virtual memory statistics we see a lot of free space for the current physical memory and page file:

```
1: kd> !vm

*** Virtual Memory Usage ***
Physical Memory: 261872 ( 1047488 Kb)
Page File: \??\C:\pagefile.sys
Current: 1354688 Kb      Free Space: 53120 Kb
Minimum: 1354688 Kb      Maximum: 4194304 Kb
Available Pages:         180984 ( 723936 Kb)
ResAvail Pages:          216475 ( 865900 Kb)
Locked IO Pages:         0 (      0 Kb)
Free System PTEs:        352925 ( 1411700 Kb)
Modified Pages:          129 (   516 Kb)
Modified PF Pages:       94 (   376 Kb)
NonPagedPool Usage:      0 (      0 Kb)
NonPagedPoolNx Usage:    16894 (   67576 Kb)
NonPagedPool Max:        192350 ( 769400 Kb)
PagedPool 0 Usage:       5957 (   23828 Kb)
PagedPool 1 Usage:       3218 (   12872 Kb)
PagedPool 2 Usage:       965 (   3860 Kb)
PagedPool 3 Usage:       1311 (   5244 Kb)
PagedPool 4 Usage:       1064 (   4256 Kb)
PagedPool Usage:          12515 (   50060 Kb)
PagedPool Maximum:       523264 ( 2093056 Kb)
Session Commit:          5021 (   20084 Kb)
Shared Commit:           15023 (   60092 Kb)
Special Pool:            0 (      0 Kb)
Shared Process:          1938 (   7752 Kb)
PagedPool Commit:        12523 (   50092 Kb)
Driver Commit:           2592 (   10368 Kb)
Committed pages:         402494 ( 1609976 Kb)
Commit limit:            589254 ( 2357016 Kb)
[...]
```

Another example is from Windows 7 memory dump. Here we can find WER reporting mutant in session 1 object directory and get problem PID from its name:

```
0: kd> !object \Sessions\1\BaseNamedObjects\
Object: fffff8a0016eb290 Type: (fffffa800426df30) Directory
ObjectHeader: fffff8a0016eb260 (new version)
HandleCount: 57 PointerCount: 217
Directory Object: fffff8a0016e9220 Name: BaseNamedObjects

Hash Address      Type      Name
---- -----      ----      --
00  fffffa8008437670 Event    STOP_HOOKING64
[...]
08  fffffa80044baa40 Mutant   WERReportingForProcess1788
[...]
```

```
0: kd> !process 0n1788 1
Searching for Process with Cid == 6fc
Cid handle table at fffff8a00180b000 with 21248 entries in use

PROCESS ffffffa8004364060
SessionId: 1 Cid: 06fc Peb: 7fffffd4000 ParentCid: 0840
DirBase: 5fbc2000 ObjectTable: fffff8a004c8e930 HandleCount: 16.
Image: ApplicationD.exe
VadRoot ffffffa8009d85170 Vads 34 Clone 0 Private 206. Modified 0. Locked 0.
DeviceMap      fffff8a001ce6b90
Token          fffff8a003eab060
ElapsedTime    00:01:51.543
UserTime       00:00:00.000
KernelTime     00:00:00.000
QuotaPoolUsage[PagedPool]      0
QuotaPoolUsage[NonPagedPool]    0
Working Set Sizes (now,min,max) (483, 50, 345) (1932KB, 200KB, 1380KB)
PeakWorkingSetSize 483
VirtualSize     13 Mb
PeakVirtualSize 13 Mb
PageFaultCount 481
MemoryPriority BACKGROUND
BasePriority    8
CommitCharge   231
```

## T

## Tampered Dump

The availability of direct dump modification raises the possibility of such memory dumps specifically modified to alter structural and behavioral diagnostic patterns. For example, to suppress certain module involvement or introduce fictitious past objects and interaction traces such as **Execution Residue** (page 365) and **Module Hints** (page 696). There can be two types of such artifacts: *strong tampering* with new or altered information completely integrated into memory fabric and *weak tampering* to confuse inexperienced software support engineers and memory forensics analysts.

For example, in one such experimental process memory dump we see **Exception Stack Trace** (page 363) pointing to a problem in *calc* module:

```
0:003> k
Child-SP RetAddr Call Site
00000000`0244e858 000007fe`fd061430 ntdll!NtWaitForMultipleObjects+0xa
00000000`0244e860 00000000`76ec1723 KERNELBASE!WaitForMultipleObjectsEx+0xe8
00000000`0244e960 00000000`76f3b5e5 kernel32!WaitForMultipleObjectsExImplementation+0xb3
00000000`0244e9f0 00000000`76f3b767 kernel32!WerFaultInternal+0x215
00000000`0244ea90 00000000`76f3b7bf kernel32!WerReportFault+0x77
00000000`0244eac0 00000000`76f3b9dc kernel32!BaseReportFault+0x1f
00000000`0244eaf0 00000000`77153398 kernel32!UnhandledExceptionFilter+0x1fc
00000000`0244ebd0 00000000`770d85c8 ntdll! ?? ::FNODOBFM::`string'+0x2365
00000000`0244ec00 00000000`770e9d2d ntdll!_C_specific_handler+0x8c
00000000`0244ec70 00000000`770d91cf ntdll!RtlpExecuteHandlerForException+0xd
00000000`0244eca0 00000000`77111248 ntdll!RtlDispatchException+0x45a
00000000`0244f380 00000000`ffdbdb27 ntDLL!KiUserExceptionDispatch+0x2e
00000000`0244fab0 00000000`76eb59ed calc!CTimedCalc::WatchDogThread+0xb2
00000000`0244faf0 00000000`770ec541 kernel32!BaseThreadInitThunk+0xd
00000000`0244fb20 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

The default analysis command (**!analyze -v**) diagnoses “*stack corruption*”:

```
FAULTING_IP:
kernel32!UnhandledExceptionFilter+1fc
00000000 76f3b9dc 448bf0 mov r14d,eax

EXCEPTION_RECORD: ffffffff`ffff -- (.exr 0xffffffff`ffff)
ExceptionAddress: 000000076f3b9dc (kernel32!UnhandledExceptionFilter+0x00000000000001fc)
ExceptionCode: 0244e9f0
ExceptionFlags: 00000000
NumberParameters: 0

DEFAULT_BUCKET_ID: STACK_CORRUPTION

PRIMARY_PROBLEM_CLASS: STACK_CORRUPTION

BUGCHECK_STR: APPLICATION_FAULT_STACK_CORRUPTION
```

```
IP_ON_HEAP: 8d483674c33bffffa
```

The fault address is not in any loaded module, please check your build's rebase log at <releasedir>\bin\build\_logs\timebuild\ntrebase.log for module which may contain the address if it were loaded.

```
UNALIGNED_STACK_POINTER: 0000000076f3b767
```

```
STACK_TEXT:
```

```
00000000`00000000 00000000`00000000 calc!CTimedCalc::WatchDogThread+0x0
```

```
FOLLOWUP_IP:
```

```
calc!CTimedCalc::WatchDogThread+0  
00000000`ffd92254 48895c2408 mov qword ptr [rsp+8],rbx
```

**Stored Exception** (page 959) resembles signs of **Local Buffer Overflow** (page 611): segment register values and CPU flags have suspiciously invalid values, possibly from **Lateral Damage** (page 602):

```
0:003> .ecxr
rax=0000000000000000 rbx=0000000000000001 rcx=00000000244ec30
rdx=00000000244ec30 rsi=0100000000000080 rdi=0000000000000158
rip=000000076f3b9dc rsp=0000000076f3b767 rbp=0000000000000000
r8=0000000000000000 r9=fffffffffffffff r10=0000000076f3b7bf
r11=00000000244ec30 r12=0000000000000001 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0 nv up di pl nz na pe nc
cs=0000 ss=0000 ds=0266 es=0000 fs=0000 gs=0154 ef1=00000000
kernel32!UnhandledExceptionFilter+0x1fc:
00000000`76f3b9dc 448bf0 mov r14d,eax
```

```
0:003> k
```

```
*** Stack trace for last set context - .thread/.cxr resets it
Child-SP RetAddr Call Site
00000000`76f3b767 8d483674`c33bffffa kernel32!UnhandledExceptionFilter+0x1fc
00000000`76f3b847 5aa3e800`05bfa0d 0x8d483674`c33bffffa
00000000`76f3b84f ebffcf83`48ccffff 0x5aa3e800`05bfa0d
00000000`76f3b857 8348c000`0409ba27 0xebffcf83`48ccffff
00000000`76f3b85f 54dfe8cf`8b48ffcf 0x8348c000`0409ba27
00000000`76f3b867 4c02778d`db33ffff 0x54dfe8cf`8b48ffcf
00000000`76f3b86f 4c000000`e024a48b 0x4c02778d`db33ffff
00000000`76f3b877 fffc8348`04ebeb8b 0x4c000000`e024a48b
00000000`76f3b87f fffc59e9`e8cc8b49 0xfffc8348`04ebeb8b
00000000`76f3b887 42e9c78b`0775c73b 0xffffc59e9`e8cc8b49
00000000`76f3b88f fffa6fa9`e8000003 0x42e9c78b`0775c73b
00000000`76f3b897 32e9c033`0774c33b 0xffffa6fa9`e8000003
00000000`76f3b89f fa7f3d8d`4c000003 0x32e9c033`0774c33b
00000000`76f3b8a7 de15ffcf`8b490006 0xfa7f3d8d`4c000003
00000000`76f3b8af f9370d8b`4800000e 0xde15ffcf`8b490006
00000000`76f3b8b7 000014a1`15ff0006 0xf9370d8b`4800000e
00000000`76f3b8bf 840fc33b`48f08b4c 0x000014a1`15ff0006
00000000`76f3b8c7 f6158b48`00000099 0x840fc33b`48f08b4c
00000000`76f3b8cf 0238c281`480006f3 0xf6158b48`00000099
00000000`76f3b8d7 48cfe8c8`8b480000 0x0238c281`480006f3
00000000`76f3b8df 8b4c7f74`c33bffff 0x48cfe8c8`8b480000
00000000`76f3b8e7 888b4900`06f3dc05 0x8b4c7f74`c33bffff
```

00000000`76f3b8ef 75083949`00000238 0x888b4900`06f3dc05  
00000000`76f3b8f7 00000240`808b496c 0x75083949`00000238  
00000000`76f3b8ff 8b415f75`08403949 0x00000240`808b496c  
00000000`76f3b907 00024880`3b411040 0x8b415f75`08403949  
00000000`76f3b90f 01040000`a9527500 0x00024880`3b411040  
00000000`76f3b917 00025090`8d491874 0x01040000`a9527500  
00000000`76f3b91f c68a4418`488d4900 0x00025090`8d491874  
00000000`76f3b927 c33a0000`117315ff 0xc68a4418`488d4900  
00000000`76f3b92f 4e15ffcf`8b493374 0xc33a0000`117315ff  
00000000`76f3b937 ff41cc8b`4900000e 0x4e15ffcf`8b493374  
00000000`76f3b93f 00028c84`0fc63bd6 0xff41cc8b`4900000e  
00000000`76f3b947 00028484`0fc73b00 0x00028c84`0fc63bd6  
00000000`76f3b94f 6ee7e819`75c33b00 0x00028484`0fc73b00  
00000000`76f3b957 c0331074`c33bffffa 0x6ee7e819`75c33b00  
00000000`76f3b95f cf8b4900`000270e9 0xc0331074`c33bffffa  
00000000`76f3b967 8b490000`0e1b15ff 0xcf8b4900`000270e9  
00000000`76f3b96f 3b000013`e215ffcc 0x8b490000`0e1b15ff  
00000000`76f3b977 0253e9c7`8b0775c7 0x3b000013`e215ffcc  
00000000`76f3b97f 41fff959`4ae80000 0x0253e9c7`8b0775c7  
00000000`76f3b987 c6844100`000002be 0x41fff959`4ae80000  
00000000`76f3b98f 15ff0000`023d850f 0xc6844100`000002be  
00000000`76f3b997 850f20a8`00000f65 0x15ff0000`023d850f  
00000000`76f3b99f 245c8948`0000022f 0x850f20a8`00000f65  
00000000`76f3b9a7 448d4c3e`4e8d4520 0x245c8948`0000022f  
00000000`76f3b9af ffc933d6`8b416024 0x448d4c3e`4e8d4520  
00000000`76f3b9b7 7cc33b00`0009f415 0xffffc933d6`8b416024  
00000000`76f3b9bf 730a7024`64ba0f0f 0x7cc33b00`0009f415  
00000000`76f3b9c7 00000205`e9c68b07 0x730a7024`64ba0f0f  
00000000`76f3b9cf cc8b49d6`8bf8b44 0x00000205`e9c68b07  
00000000`76f3b9d7 f08b44ff`ffffdc4e8 0xcc8b49d6`8bf8b44  
00000000`76f3b9df e9c03307`7508f883 0xf08b44ff`ffffdc4e8  
00000000`76f3b9e7 7506f883`000001e9 0xe9c03307`7508f883  
00000000`76f3b9ef c33bffffa`6e4be810 0x7506f883`000001e9  
00000000`76f3b9f7 0001d4e9`c0330774 0xc33bffffa`6e4be810  
00000000`76f3b9ff 86850f04`fe834100 0x0001d4e9`c0330774  
00000000`76f3ba07 0000024a`ba000001 0x86850f04`fe834100  
00000000`76f3ba0f 00b841ce`8b45c933 0x0000024a`ba000001  
00000000`76f3ba17 fff7a249`e8000010 0x00b841ce`8b45c933  
00000000`76f3ba1f 0775c33b`48e88b4c 0xffff7a249`e8000010  
00000000`76f3ba27 48000001`a6e9c033 0x0775c33b`48e88b4c  
00000000`76f3ba2f 24448948`3024448d 0x48000001`a6e9c033  
00000000`76f3ba37 0000f024`8c8d4c20 0x24448948`3024448d  
00000000`76f3ba3f 49000001`25b84100 0x0000f024`8c8d4c20  
00000000`76f3ba47 8a0fe8cf`8b48d58b 0x49000001`25b84100  
00000000`76f3ba4f 4166097c`c33bffffe 0x8a0fe8cf`8b48d58b  
00000000`76f3ba57 39fe450f`44005d39 0x4166097c`c33bffffe  
00000000`76f3ba5f 850f0000`00f0249c 0x39fe450f`44005d39  
00000000`76f3ba67 240c8b49`000000bc 0x850f0000`00f0249c  
00000000`76f3ba6f 40244489`48016348 0x240c8b49`000000bc  
00000000`76f3ba77 24448948`10418b48 0x40244489`48016348  
00000000`76f3ba7f 75c00000`06398148 0x24448948`10418b48  
00000000`76f3ba87 480b7203`18798318 0x75c00000`06398148  
00000000`76f3ba8f 50244489`4830418b 0x480b7203`18798318  
00000000`76f3ba97 eb50245c`89481ceb 0x50244489`4830418b  
00000000`76f3ba9f 8b480b72`18713915 0xeb50245c`89481ceb  
00000000`76f3baa7 eb502444`89482041 0x8b480b72`18713915

```

00000000`76f3baaf 02ba5024`5c894805 0xeb502444`89482041
00000000`76f3bab7 0b721851`39000000 0x02ba5024`5c894805
00000000`76f3babf 24448948`28418b48 0x0b721851`39000000
00000000`76f3bac7 58245c89`4805eb58 0x24448948`28418b48
00000000`76f3bacf ba1d3808`74fb3b44 0x58245c89`4805eb58
00000000`76f3bad7 48d68b02`740006fd 0xba1d3808`74fb3b44
00000000`76f3badf 48000000`e824848d 0x48d68b02`740006fd
00000000`76f3bae7 20245489`28244489 0x48000000`e824848d
00000000`76f3baef c0334540`244c8d4c 0x20245489`28244489
00000000`76f3baf7 000144b9`04508d41 0xc0334540`244c8d4c
00000000`76f3baff ba00000d`7215ffd0 0x000144b9`04508d41
00000000`76f3bb07 8c8bc223`c0000000 0xba00000d`7215ffd0
00000000`76f3bb0f b8c23b00`0000e824 0x8c8bc223`c0000000
00000000`76f3bb17 89c8440f`00000006 0xb8c23b00`0000e824
00000000`76f3bb1f 07eb0000`00e8248c 0x89c8440f`00000006
00000000`76f3bb27 44000000`e8248c8b 0x07eb0000`00e8248c
00000000`76f3bb2f 7403f983`5d74fb3b 0x44000000`e8248c8b
00000000`76f3bb37 000000f0`249c3909 0x7403f983`5d74fb3b
00000000`76f3bb3f 0006fd4d`058a4f74 0x000000f0`249c3909
00000000`76f3bb47 f85f5ce8`4b75c33a 0x0006fd4d`058a4f74
00000000`76f3bb4f 448b3b75`5c5838ff 0xf85f5ce8`4b75c33a
00000000`76f3bb57 894c2824`44893024 0x448b3b75`5c5838ff
00000000`76f3bb5f 08244c8b`4d20246c 0x894c2824`44893024
00000000`76f3bb67 fec2c748`24048b4d 0x08244c8b`4d20246c
00000000`76f3bb6f b6e8cf8b`4fffffff 0xfc2c748`24048b4d
00000000`76f3bb77 fd130db6`0fffffea 0xb6e8cf8b`4fffffff
00000000`76f3bb7f 88ce4c0f`c33b0006 0xfd130db6`0fffffea
00000000`76f3bb87 ebfb8b00`06fd080d 0x88ce4c0f`c33b0006
00000000`76f3bb8f 3a0006fc`fe058a29 0xebfb8b00`06fd080d
00000000`76f3bb97 8b240c8b`491874c3 0x3a0006fc`fe058a29
00000000`76f3bb9f 060f15ff`cf8b4811 0x8b240c8b`491874c3
00000000`76f3bba7 0000f824`bc8b0000 0x060f15ff`cf8b4811
00000000`76f3bbaf 00f824bc`8b07eb00 0x0000f824`bc8b0000
00000000`76f3bbb7 331074eb`3b4c0000 0x00f824bc`8b07eb00
00000000`76f3bbbf 49000080`00b841d2 0x331074eb`3b4c0000
00000000`76f3bbc7 8bfff74b`5ae8cd8b 0x49000080`00b841d2
00000000`76f3bbcf c48148c6`8b02ebc7 0x8bfff74b`5ae8cd8b
00000000`76f3bbd7 5e415f41`000000a0 0xc48148c6`8b02ebc7
00000000`76f3bbdf c35b5e5f`5c415d41 0x5e415f41`000000a0
00000000`76f3bbe7 158ead00`00000090 0xc35b5e5f`5c415d41
00000000`76f3bbef 00000200`00000053 0x158ead00`00000090
00000000`76f3bbf7 09bc2400`00002500 0x00000200`00000053
00000000`76f3bbff 00000000`09b42400 0x09bc2400`00002500
00000000`76f3bc07 7e023553`158ead00 0x9b42400
00000000`76f3bc0f 00000400`00000a19 0x7e023553`158ead00
00000000`76f3bc17 09b42000`09bc2000 0x00000400`00000a19
00000000`76f3bc1f 445352bb`03197e00 0x09b42000`09bc2000
00000000`76f3bc27 4c886225`48e28953 0x445352bb`03197e00
00000000`76f3bc2f 4fb29af4`dfbb8344 0x4c886225`48e28953
00000000`76f3bc37 72656b00`0000020e 0x4fb29af4`dfbb8344
00000000`76f3bc3f 64702e32`336c656e 0x72656b00`0000020e
00000000`76f3bc47 00000000`00000062 0x64702e32`336c656e

```

We check for any **Hidden Exceptions** (page 457) and find it was **NULL Data Pointer** (page 752):

```
0:003> .cxr
Resetting default scope

0:003> k
Child-SP RetAddr Call Site
00000000`0244e858 000007fe`fd061430 ntdll!NtWaitForMultipleObjects+0xa
00000000`0244e860 00000000`76ec1723 KERNELBASE!WaitForMultipleObjectsEx+0xe8
00000000`0244e960 00000000`76f3b5e5 kernel32!WaitForMultipleObjectsExImplementation+0xb3
00000000`0244e9f0 00000000`76f3b767 kernel32!WerpReportFaultInternal+0x215
00000000`0244ea90 00000000`76f3b7bf kernel32!WerpReportFault+0x77
00000000`0244eac0 00000000`76f3b9dc kernel32!BasepReportFault+0x1f
00000000`0244eaf0 00000000`77153398 kernel32!UnhandledExceptionFilter+0xfc
00000000`0244ebd0 00000000`770d85c8 ntdll! ?? ::FNODOBFM::`string'+0x2365
00000000`0244ec00 00000000`770e9d2d ntdll!_C_specific_handler+0x8c
00000000`0244ec70 00000000`770d91cf ntdll!RtlpExecuteHandlerForException+0xd
00000000`0244eca0 00000000`77111248 ntdll!RtlDispatchException+0x45a
00000000`0244f380 00000000`ffdbdb27 ntdll!KiUserExceptionDispatch+0x2e
00000000`0244fab0 00000000`76eb59ed calc!CTimedCalc::WatchDogThread+0xb2
00000000`0244faf0 00000000`770ec541 kernel32!BaseThreadInitThunk+0xd
00000000`0244fb20 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

0:003> dps 00000000`0244eca0 00000000`0244fab0
00000000`0244eca0 00000000`02450000
00000000`0244eca8 00000000`76fadda0 kernel32!__PchSym_ <PERF> (kernel32+0x10dda0)
00000000`0244ecb0 00000000`00012f00
00000000`0244ecb8 00000000`7711920a ntdll!RtlDosApplyFileIsolationRedirection_Ustr+0x3da
00000000`0244ecc0 00000000`00000005
00000000`0244ecc8 00000000`00000000
00000000`0244ecd0 00000000`00000000
00000000`0244ecd8 00000000`00000000
00000000`0244ece0 00000000`0244fb20
00000000`0244ece8 00000000`00000000
00000000`0244ecf0 00000000`77202dd0 ntdll!CsrPortMemoryRemoteDelta <PERF> (ntdll+0x142dd0)
00000000`0244ecf8 00000000`00000000
00000000`0244ed00 00000000`00000000
00000000`0244ed08 00000000`02450000
00000000`0244ed10 00000000`771e8180 ntdll!`string'+0xc040
00000000`0244ed18 00000000`0244b000
00000000`0244ed20 00000000`0244f250
00000000`0244ed28 00000000`770c0000 ntdll!RtlDeactivateActivationContext <PERF> (ntdll+0x0)
00000000`0244ed30 00000000`770ec541 ntdll!RtlUserThreadStart+0x1d
00000000`0244ed38 00000000`770c0000 ntdll!RtlDeactivateActivationContext <PERF> (ntdll+0x0)
00000000`0244ed40 00000000`77202dd0 ntdll!CsrPortMemoryRemoteDelta <PERF> (ntdll+0x142dd0)
00000000`0244ed48 00000000`0244fb20
00000000`0244ed50 00000000`771d7718 ntdll!LdrpDefaultExtension
00000000`0244ed58 00000000`0244ed80
00000000`0244ed60 00000000`770d852c ntdll!_C_specific_handler
00000000`0244ed68 00000000`771e8180 ntdll!`string'+0xc040
00000000`0244ed70 00000000`0244f250
00000000`0244ed78 00000000`00000000
00000000`0244ed80 00000000`00000000
00000000`0244ed88 00000000`00000000
00000000`0244ed90 00000000`00000000
```

```
00000000`0244ed98 00000000`00000000
00000000`0244eda0 00000000`00000000
00000000`0244eda8 00000000`00000000
00000000`0244edb0 00001f80`00000000
00000000`0244edb8 00000000`00000033
00000000`0244edc0 00010246`002b0000
00000000`0244edc8 00000000`00000000
00000000`0244edd0 00000000`00000000
00000000`0244edd8 00000000`00000000
00000000`0244ede0 00000000`00000000
00000000`0244ede8 000007fe`ff3625c0 msctf!s_szCompClassName
00000000`0244edf0 00000000`002b0000
00000000`0244edf8 00000000`0244ee40
00000000`0244ee00 00000000`0244ee40
00000000`0244ee08 00000000`0244ee40
00000000`0244ee10 00000000`00000000
00000000`0244ee18 00000000`0244fb70
00000000`0244ee20 00000000`00000000
00000000`0244ee28 00000000`00000000
00000000`0244ee30 00000000`00000000
00000000`0244ee38 000007fe`fd602790 ole32!`string'
00000000`0244ee40 00000000`00292170
00000000`0244ee48 00000000`770e7a33 ntdll!LdrpFindOrMapD1l+0x138
00000000`0244ee50 00000000`0244ef68
00000000`0244ee58 00000000`00000000
00000000`0244ee60 00000000`00000000
00000000`0244ee68 00000000`00000000
00000000`0244ee70 00000000`00000000
00000000`0244ee78 00000000`00000000
00000000`0244ee80 00000000`0000027f
00000000`0244ee88 00000000`00000000
00000000`0244ee90 00000000`00000000
00000000`0244ee98 0000ffff`00001f80
00000000`0244eea0 00000000`00000000
00000000`0244eea8 00000000`00000000
00000000`0244eeb0 00000000`00000000
00000000`0244eeb8 00000000`00000000
00000000`0244eec0 00000000`00000000
00000000`0244eec8 00000000`00000000
00000000`0244eed0 00000000`00000000
00000000`0244eed8 00000000`00000000
00000000`0244eee0 00000000`00000000
00000000`0244eee8 00000000`00000000
00000000`0244eef0 00000000`00000000
00000000`0244eef8 00000000`00000000
00000000`0244ef00 00000000`00000000
00000000`0244ef08 00000000`00000000
00000000`0244ef10 00000000`00000000
00000000`0244ef18 00000000`00000000
00000000`0244ef20 00000000`00000000
00000000`0244ef28 00000000`771192a8 ntdll!LdrpApplyFileNameRedirection+0x2d3
00000000`0244ef30 00000000`00000000
00000000`0244ef38 00000000`00000000
00000000`0244ef40 00000000`00000000
00000000`0244ef48 00000000`02080000
00000000`0244ef50 00000000`0244f028
```

```
00000000`0244ef58 00000000`0244f020
00000000`0244ef60 00000000`00000000
00000000`0244ef68 00000000`00000000
00000000`0244ef70 00000000`00000000
00000000`0244ef78 000007fe`fd602848 ole32!`string'
00000000`0244ef80 00000000`00000000
00000000`0244ef88 00000000`00000000
00000000`0244ef90 00000000`00000000
00000000`0244ef98 00000000`00000000
00000000`0244efa0 00000000`00000000
00000000`0244efa8 00000000`00000000
00000000`0244efb0 00000000`00000000
00000000`0244efb8 00000000`00000000
00000000`0244efc0 00000000`00000000
00000000`0244efc8 00000000`00000000
00000000`0244efd0 00000000`00000000
00000000`0244efd8 00000000`00000000
00000000`0244efe0 00000000`00000000
00000000`0244efe8 00000000`00000000
00000000`0244eff0 00000000`00000000
00000000`0244eff8 00000000`00000000
00000000`0244f000 00000000`00000000
00000000`0244f008 00000000`00000000
00000000`0244f010 00000000`00000000
00000000`0244f018 00000000`00000000
00000000`0244f020 00000000`0244f038
00000000`0244f028 00000000`0000011b
00000000`0244f030 00000000`024d0000
00000000`0244f038 00000080`001a024d
00000000`0244f040 00000000`01c0c8a0
00000000`0244f048 00000000`002f0101
00000000`0244f050 00000000`00000000
00000000`0244f058 00000000`00000022
00000000`0244f060 00000000`002f9b00
00000000`0244f068 00000000`01bd5390
00000000`0244f070 00000000`002f7c00
00000000`0244f078 00000000`01bd5580
00000000`0244f080 00000000`01bd57b0
00000000`0244f088 00000000`002f9b00
00000000`0244f090 00000000`00000000
00000000`0244f098 00000024`00000003
00000000`0244f0a0 00000000`002e91b0
00000000`0244f0a8 00000000`00000022
00000000`0244f0b0 00000000`771d5430 ntdll!RtlpInterceptorRoutines
00000000`0244f0b8 00000000`00000000
00000000`0244f0c0 00000000`00000010
00000000`0244f0c8 00000000`01bd0000
00000000`0244f0d0 00000000`00000008
00000000`0244f0d8 00000000`00000001
00000000`0244f0e0 00000000`01bd0288
00000000`0244f0e8 00000000`77113448 ntdll!RtlAllocateHeap+0xe4
00000000`0244f0f0 00000000`00000000
00000000`0244f0f8 00000000`00000001
00000000`0244f100 000002b2`000f002f
00000000`0244f108 00000000`01bd5780
00000000`0244f110 00000000`00250230
```

```

00000000`0244f118 00000000`000000df
00000000`0244f120 00000000`002551a0
00000000`0244f128 00000000`00255210
00000000`0244f130 00000000`002f9b00
00000000`0244f138 00000000`002551a0
00000000`0244f140 00000000`000000df
00000000`0244f148 00000000`10000010
00000000`0244f150 00000000`00250230
00000000`0244f158 00000000`00000000
00000000`0244f160 00000000`00250498
00000000`0244f168 00000000`0025026c
00000000`0244f170 00000000`002f9b00
00000000`0244f178 00000000`002551a0
00000000`0244f180 00000000`00000022
00000000`0244f188 00000000`76fd88b8 user32!GetPropW+0x4d
00000000`0244f190 00000000`00002974
00000000`0244f198 00000000`76fd88b8 user32!GetPropW+0x4d
00000000`0244f1a0 00000000`00250230
00000000`0244f1a8 00000000`76fd7931 user32!IsWindow+0x9
00000000`0244f1b0 00000000`002ed6d0
00000000`0244f1b8 00000000`76fd7931 user32!IsWindow+0x9
00000000`0244f1c0 00000000`00000000
00000000`0244f1c8 00000000`01c0c8d0
00000000`0244f1d0 00000000`01c0c8a0
00000000`0244f1d8 00000000`00000000
00000000`0244f1e0 00000000`00000008
00000000`0244f1e8 00000000`01bd0000
00000000`0244f1f0 00000000`00000000
00000000`0244f1f8 00000000`770f41c8 ntdll!RtlpReAllocateHeap+0x178
00000000`0244f200 00000000`00000002
00000000`0244f208 00000000`00000002
00000000`0244f210 00000000`00000000
00000000`0244f218 000007fe`4f00024d
00000000`0244f220 00000000`00000000
00000000`0244f228 000007fe`fb601381 uxtheme!CTHEMEWND::_PreDefWindowProc+0x31
00000000`0244f230 00000000`00000082
00000000`0244f238 00000000`00000000
00000000`0244f240 00000000`7a337100
00000000`0244f248 00000000`01c0c8c0
00000000`0244f250 00000000`00000003
00000000`0244f258 00000000`76eb59e0 kernel32!BaseThreadInitThunk
00000000`0244f260 00000000`ffdedb32 calc!CTimedCalc::Start+0xa9
00000000`0244f268 00000000`ffd90000 calc!CCalculatorController <PERF> (calc+0x0)
00000000`0244f270 00000000`ffe0ac64 calc!_dyn_tls_init_callback <PERF> (calc+0x7ac64)
00000000`0244f278 00000000`76ea0000 kernel32!TestResourceDataMatchEntry <PERF> (kernel32+0x0)
00000000`0244f280 00000000`76fadda0 kernel32!__PchSym_ <PERF> (kernel32+0x10dda0)
00000000`0244f288 00000000`770c0000 ntdll!RtlDeactivateActivationContext <PERF> (ntdll+0x0)
00000000`0244f290 00000000`77202dd0 ntdll!CsrPortMemoryRemoteDelta <PERF> (ntdll+0x142dd0)
00000000`0244f298 00000000`76fd760e user32!RealDefWindowProcW+0x5a
00000000`0244f2a0 00000000`00000001
00000000`0244f2a8 000007fe`fb600037 uxtheme!operator delete <PERF> (uxtheme+0x37)
00000000`0244f2b0 00000000`01bd0158
00000000`0244f2b8 00000000`00000082
00000000`0244f2c0 00000000`00000000
00000000`0244f2c8 00000000`00000003
00000000`0244f2d0 00000000`000111f2

```

00000000`0244f2d8 00000000`00000054  
00000000`0244f2e0 00000000`00000000  
00000000`0244f2e8 00000000`00000000  
00000000`0244f2f0 00000000`00000001  
00000000`0244f2f8 00000000`01c11c60  
00000000`0244f300 00000000`0244f462  
00000000`0244f308 00000000`01bd0230  
00000000`0244f310 00000000`00000000  
00000000`0244f318 00000000`00000000  
00000000`0244f320 00000000`00000000  
00000000`0244f328 00000000`14010015  
00000000`0244f330 00000000`01c11570  
00000000`0244f338 00000000`00000000  
00000000`0244f340 00000000`00000000  
00000000`0244f348 00000000`00000000  
00000000`0244f350 00000000`00009c40  
00000000`0244f358 00000000`00000000  
00000000`0244f360 00000000`00000000  
00000000`0244f368 00000000`00000000  
00000000`0244f370 00000000`00002710  
00000000`0244f378 00000000`77111248 *ntdll!KiUserExceptionDispatch+0x2e*  
00000000`0244f380 00000000`0244f870  
00000000`0244f388 00000000`0244f380  
00000000`0244f390 00000000`00000000  
00000000`0244f398 00000000`00000000  
00000000`0244f3a0 000007fe`fb63fb40 uxtheme!\$\$VProc\_ImageExportDirectory  
00000000`0244f3a8 00000000`0000ad5  
00000000`0244f3b0 00001f80`0010005f  
00000000`0244f3b8 0053002b`002b0033  
00000000`0244f3c0 00010246`002b002b  
00000000`0244f3c8 00000000`00000000  
00000000`0244f3d0 00000000`00000000  
00000000`0244f3d8 00000000`00000000  
00000000`0244f3e0 00000000`00000000  
00000000`0244f3e8 00000000`00000000  
00000000`0244f3f0 00000000`00000000  
00000000`0244f3f8 00000000`0012c770  
00000000`0244f400 00000000`00000000  
00000000`0244f408 00000000`00000000  
00000000`0244f410 00000000`00002710  
00000000`0244f418 00000000`0244fab0  
00000000`0244f420 00000000`00000000  
00000000`0244f428 00000000`00000000  
00000000`0244f430 00000000`00000000  
00000000`0244f438 00000000`0244f938  
00000000`0244f440 00000000`00962210  
00000000`0244f448 00000000`00000000  
00000000`0244f450 00000000`0244f9a0  
00000000`0244f458 00000000`00009c40  
00000000`0244f460 00000000`00000000  
00000000`0244f468 00000000`00000000  
00000000`0244f470 00000000`00000000  
00000000`0244f478 00000000`ffdedb27 calc!CTimedCalc::WatchDogThread+0xb2  
00000000`0244f480 00000000`000027f  
00000000`0244f488 00000000`00000000  
00000000`0244f490 00000000`00000000

00000000`0244f498 0000ffff`00001f80  
00000000`0244f4a0 00000000`00000000  
00000000`0244f4a8 00000000`00000000  
00000000`0244f4b0 00000000`00000000  
00000000`0244f4b8 00000000`00000000  
00000000`0244f4c0 00000000`00000000  
00000000`0244f4c8 00000000`00000000  
00000000`0244f4d0 00000000`00000000  
00000000`0244f4d8 00000000`00000000  
00000000`0244f4e0 00000000`00000000  
00000000`0244f4e8 00000000`00000000  
00000000`0244f4f0 00000000`00000000  
00000000`0244f4f8 00000000`00000000  
00000000`0244f500 00000000`00000000  
00000000`0244f508 00000000`00000000  
00000000`0244f510 00000000`00000000  
00000000`0244f518 00000000`00000000  
00000000`0244f520 00000000`00000000  
00000000`0244f528 00000000`00000000  
00000000`0244f530 00000000`00000000  
00000000`0244f538 00000000`00000000  
00000000`0244f540 00000000`00000000  
00000000`0244f548 00000000`00000000  
00000000`0244f550 00000000`00000000  
00000000`0244f558 00000000`00000000  
00000000`0244f560 00000000`00000000  
00000000`0244f568 00000000`00000000  
00000000`0244f570 00000000`00000000  
00000000`0244f578 00000000`00000000  
00000000`0244f580 00000000`00000000  
00000000`0244f588 00000000`00000000  
00000000`0244f590 00000000`00000000  
00000000`0244f598 00000000`00000000  
00000000`0244f5a0 00000000`00000000  
00000000`0244f5a8 00000000`00000000  
00000000`0244f5b0 00000000`00000000  
00000000`0244f5b8 00000000`00000000  
00000000`0244f5c0 00000000`00000000  
00000000`0244f5c8 00000000`00000000  
00000000`0244f5d0 00000000`00000000  
00000000`0244f5d8 00000000`00000000  
00000000`0244f5e0 00000000`00000000  
00000000`0244f5e8 00000000`00000000  
00000000`0244f5f0 00000000`00000000  
00000000`0244f5f8 00000000`00000000  
00000000`0244f600 00000000`00000000  
00000000`0244f608 00000000`00000000  
00000000`0244f610 00000000`00000000  
00000000`0244f618 00000000`00000000  
00000000`0244f620 00000000`00000000  
00000000`0244f628 00000000`00000000  
00000000`0244f630 00000000`00000000  
00000000`0244f638 00000000`00000000  
00000000`0244f640 00000000`00000000  
00000000`0244f648 00000000`00000000  
00000000`0244f650 00000000`00000000

00000000`0244f658 00000000`00000000  
00000000`0244f660 00000000`00000000  
00000000`0244f668 fffff800`032d5e53  
00000000`0244f670 00000000`00000002  
00000000`0244f678 00000000`00000000  
00000000`0244f680 00000000`01c11580  
00000000`0244f688 00000000`00000082  
00000000`0244f690 00000000`00000082  
00000000`0244f698 00000000`000111e4  
00000000`0244f6a0 00000000`00000002  
00000000`0244f6a8 00000000`0244f6f0  
00000000`0244f6b0 00000000`00000002  
00000000`0244f6b8 00000000`00000000  
00000000`0244f6c0 00000000`000111e4  
00000000`0244f6c8 00000000`00000000  
00000000`0244f6d0 00000000`00000082  
00000000`0244f6d8 00000000`00000000  
00000000`0244f6e0 00000000`00000000  
00000000`0244f6e8 00000000`76fe76c2 user32!DefDlgProcW+0x36  
00000000`0244f6f0 00000000`00000000  
00000000`0244f6f8 00000000`00000000  
00000000`0244f700 00000000`000111e4  
00000000`0244f708 00000000`00000000  
00000000`0244f710 00000000`00000082  
00000000`0244f718 00000000`00000000  
00000000`0244f720 00000000`0244f908  
00000000`0244f728 00000000`76fd9bef user32!UserCallWinProcCheckWow+0x1cb  
00000000`0244f730 00000000`00962210  
00000000`0244f738 00000000`00000001  
00000000`0244f740 00000000`00000000  
00000000`0244f748 00000000`00000000  
00000000`0244f750 00000000`0244f768  
00000000`0244f758 00000000`0244f778  
00000000`0244f760 00000000`00000001  
00000000`0244f768 00000000`00000000  
00000000`0244f770 00000000`00000000  
00000000`0244f778 00000000`00000000  
00000000`0244f780 00000000`00000048  
00000000`0244f788 00000000`00000001  
00000000`0244f790 00000000`00000000  
00000000`0244f798 00000000`00000000  
00000000`0244f7a0 00000000`00000070  
00000000`0244f7a8 ffffffff`ffffffff  
00000000`0244f7b0 ffffffff`ffffffff  
00000000`0244f7b8 00000000`76fd9b43 user32!UserCallWinProcCheckWow+0x99  
00000000`0244f7c0 00000000`76fd9bef user32!UserCallWinProcCheckWow+0x1cb  
00000000`0244f7c8 00000000`00000000  
00000000`0244f7d0 00000000`00000000  
00000000`0244f7d8 00000000`00000000  
00000000`0244f7e0 00000000`00000000  
00000000`0244f7e8 00000000`76fd72cb user32!DispatchClientMessage+0xc3  
00000000`0244f7f0 00000000`00000000  
00000000`0244f7f8 00000000`770e46b4 ntdll!NtDllDialogWndProc\_W  
00000000`0244f800 00000000`00000000  
00000000`0244f808 00000000`00000000  
00000000`0244f810 00000000`00000000

```
00000000`0244f818 00000000`00000000
00000000`0244f820 00000000`00962238
00000000`0244f828 00000000`00000001
00000000`0244f830 00000000`00000000
00000000`0244f838 00000000`00000000
00000000`0244f840 00000000`00000000
00000000`0244f848 00000000`00000000
00000000`0244f850 00000730`fffffb30
00000000`0244f858 000004d0`fffffb30
00000000`0244f860 00000170`000000f0
00000000`0244f868 0000002c`00000001
00000000`0244f870 00000000`c0000005
00000000`0244f878 00000000`00000000
00000000`0244f880 00000000`ffdedb27 calc!CTimedCalc::WatchDogThread+0xb2
00000000`0244f888 00000000`00000002
00000000`0244f890 00000000`00000000
00000000`0244f898 00000000`00000000
00000000`0244f8a0 00000000`00000000
00000000`0244f8a8 00000000`00000000
00000000`0244f8b0 00000000`00000000
00000000`0244f8b8 00000000`00000000
00000000`0244f8c0 00000000`00000000
00000000`0244f8c8 00000000`00000000
00000000`0244f8d0 00000000`00000000
00000000`0244f8d8 00000000`00000000
00000000`0244f8e0 00000000`00000000
00000000`0244f8e8 00000000`00000000
00000000`0244f8f0 00000000`00000000
00000000`0244f8f8 00000000`00000000
00000000`0244f900 00000000`00000000
00000000`0244f908 00000000`00962210
00000000`0244f910 00000000`ffdedb27 calc!CTimedCalc::WatchDogThread+0xb2
00000000`0244f918 00000000`00000000
00000000`0244f920 00000000`00000000
00000000`0244f928 00000000`0244fab0
00000000`0244f930 00000000`77101530 ntdll!NtDllDispatchMessage_W
00000000`0244f938 00000000`76fe505b user32!DialogBox2+0x2ec
00000000`0244f940 00000000`00000000
00000000`0244f948 00000000`00000000
00000000`0244f950 00000000`00000000
00000000`0244f958 00000000`00000000
00000000`0244f960 00000000`00000000
00000000`0244f968 00000000`00000000
00000000`0244f970 00000000`00000000
00000000`0244f978 00000000`00000000
00000000`0244f980 00000000`00000002
00000000`0244f988 00000000`00011f0
00000000`0244f990 00000271`0f689359
00000000`0244f998 00000000`00000030
00000000`0244f9a0 00000000`00000000
00000000`0244f9a8 00000000`00000000
00000000`0244f9b0 00000000`ffd90000 calc!CCalculatorController::CCalculatorController <PERF> (calc+0x0)
00000000`0244f9b8 00000000`001a17e0
00000000`0244f9c0 00000000`00000000
00000000`0244f9c8 00000000`76fe4edd user32!InternalDialogBox+0x135
00000000`0244f9d0 00000000`00000000
```

```

00000000`0244f9d8 00000000`ffdcedb0 calc!CTimedCalc::TimeOutDlgProc
00000000`0244f9e0 00000000`00000000
00000000`0244f9e8 00000000`00000000
00000000`0244f9f0 00000000`ffdcedb0 calc!CTimedCalc::TimeOutDlgProc
00000000`0244f9f8 00000000`00000000
00000000`0244fa00 00000000`00000001
00000000`0244fa08 00000000`00000000
00000000`0244fa10 00000000`00000000
00000000`0244fa18 00000000`00009c40
00000000`0244fa20 00000000`ffd90000 calc!CCalculatorController::CCalculatorController <PERF> (calc+0x0)
00000000`0244fa28 00000000`76fe4f52 user32!DialogBoxIndirectParamAorW+0x58
00000000`0244fa30 00000000`001a17e0
00000000`0244fa38 00000000`00000000
00000000`0244fa40 00000000`ffdcedb0 calc!CTimedCalc::TimeOutDlgProc
00000000`0244fa48 00000000`ffdcedb0 calc!CTimedCalc::TimeOutDlgProc
00000000`0244fa50 00000000`00000000
00000000`0244fa58 00000000`00000001
00000000`0244fa60 00000000`ffd90000 calc!CCalculatorController::CCalculatorController <PERF> (calc+0x0)
00000000`0244fa68 00000000`76fdd476 user32!DialogBoxParamW+0x66
00000000`0244fa70 ffffffff`fffffff
00000000`0244fa78 00000000`00000000
00000000`0244fa80 00000000`ffdcedb0 calc!CTimedCalc::TimeOutDlgProc
00000000`0244fa88 00000000`00000000
00000000`0244fa90 00000000`00000000
00000000`0244fa98 00000000`00000000
00000000`0244faa0 00000000`00000000
00000000`0244faa8 00000000`ffdbdfa calc!CTimedCalc::WatchDogThread+0x72
00000000`0244fab0 00000000`00002710

```

Segment registers and flags look normal now:

```

0:003> .cxr 00000000`0244f380
rax=000000000012c770 rbx=0000000000002710 rcx=0000000000000000
rdx=0000000000000000 rsi=0000000000000000 rdi=0000000000000000
rip=00000000ffdbdb27 rsp=00000000244fab0 rbp=0000000000000000
r8=00000000244f938 r9=0000000000962210 r10=0000000000000000
r11=00000000244f9a0 r12=000000000009c40 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0 nv up ei pl zr na po nc
cs=0033 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010246
calc!CTimedCalc::WatchDogThread+0xb2:
00000000`ffdbdb27 488b01 mov rax,qword ptr [rcx] ds:00000000`00000000=??????????????????
0:003> k
*** Stack trace for last set context - .thread/.cxr resets it
Child-SP RetAddr Call Site
00000000`0244fab0 00000000`76eb59ed calc!CTimedCalc::WatchDogThread+0xb2
00000000`0244faf0 00000000`770ec541 kernel32!BaseThreadInitThunk+0xd
00000000`0244fb20 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

```

## Technology-Specific Subtrace

### COM Client Call

Here we add **Technology-Specific Subtrace** pattern for **COM client calls** (as compared to **COM interface invocation** for servers, page 988). We recently got a complete memory dump where we had to find the destination server process, and we used the old technique described in the article **In Search of Lost CID**<sup>206</sup>. We reprint the 32-bit stack subtrace here:

```
[...]
00faf828 7778c38b ole32!CRpcChannelBuffer::SwitchAptAndDispatchCall+0x112
00faf908 776c0565 ole32!CRpcChannelBuffer::SendReceive2+0xd3
00faf974 776c04fa ole32!CAptRpcChnl::SendReceive+0xab
00faf9c8 77ce247f ole32!CCtxComChnl::SendReceive+0x1a9
00faf9e4 77ce252f RPCRT4!NdrProxySendReceive+0x43
00fafadcc 77ce25a6 RPCRT4!NdrClientCall2+0x206
[...]
```

Here's also an x64 fragment from **Semantic Structures** (PID.TID) pattern (page 860):

```
[...]
00000000`018ce450 000007fe`ffee041b ole32!CRpcChannelBuffer::SwitchAptAndDispatchCall+0xa3
00000000`018ce4f0 000007fe`ffd819c6 ole32!CRpcChannelBuffer::SendReceive2+0x11b
00000000`018ce6b0 000007fe`ffd81928 ole32!CAptRpcChnl::SendReceive+0x52
00000000`018ce780 000007fe`ffedfcf5 ole32!CCtxComChnl::SendReceive+0x68
00000000`018ce830 000007fe`ff56ba3b ole32!NdrExtpProxySendReceive+0x45
00000000`018ce860 000007fe`ffee02d0 RPCRT4!NdrpClientCall3+0x2e2
[...]
```

If we have the call over ALPC, it is easy to find the server process and thread (**Wait Chain**, page 1097). In the case of a modal loop, we can use the raw stack analysis technique mentioned above (see also the case study<sup>207</sup>).

Other subtrace examples can be found in pattern examples for **High Contention** (.NET CLR monitors, page 472), **Wait Chain** (RTL\_RESOURCE, page 1122), and in the case study<sup>208</sup>.

---

<sup>206</sup> Memory Dump Analysis Anthology, Volume 2, page 136

<sup>207</sup> Memory Dump Analysis Anthology, Volume 3, page 205

<sup>208</sup> Memory Dump Analysis Anthology, Volume 4, page 182

## COM Interface Invocation

**Stack Trace** (page 926) is a general pattern, and there can always be found fine-grained patterns in stack traces as well. Here we discuss the general category of such stack trace patterns (TSST) and give examples related to COM technology.

Consider this trace:

```
1: kd> k250
ChildEBP RetAddr
8d5d2808 82a7eb15 nt!KiSwapContext+0x26
8d5d2840 82a7d403 nt!KiSwapThread+0x266
8d5d2868 82a772cf nt!KiCommitThreadWait+0x1df
8d5d28e0 82550d75 nt!KeWaitForSingleObject+0x393
8d5d293c 82550e10 win32k!xxxRealSleepThread+0x1d7
8d5d2958 824ff4b0 win32k!xxxSleepThread+0x2d
8d5d29cc 825547e8 win32k!xxxInterSendMsgEx+0xb1c
8d5d2a1c 825546a4 win32k!xxxSendMessageTimeout+0x13b
8d5d2a44 82533843 win32k!xxxSendMessage+0x28
8d5d2b08 824fd865 win32k!xxxCalcValidRects+0xf7
8d5d2b64 82502c98 win32k!xxxEndDeferWindowPosEx+0x100
8d5d2b84 825170c9 win32k!xxxSetWindowPos+0xf6
8d5d2c08 82517701 win32k!xxxActivateThisWindow+0x2b1
8d5d2c38 82517537 win32k!xxxActivateWindow+0x144
8d5d2c4c 824fd9dd win32k!xxxSwpActivate+0x44
8d5d2ca4 82502c98 win32k!xxxEndDeferWindowPosEx+0x278
8d5d2cc4 824ffff82 win32k!xxxSetWindowPos+0xf6
8d5d2d10 82a5342a win32k!NtUserSetWindowPos+0x140
8d5d2d10 76ee64f4 nt!KiFastCallEntry+0x12a (TrapFrame @ 8d5d2d34)
01e2cea0 7621358d nt!NtUserSetWindowPos+0xc
01e2cea4 6a8fa0eb USER32!NtUserSetWindowPos+0xc
01e2cf14 6a894b13 IEFRA  
ME!SHToggleDialogExpando+0x15a
01e2cf28 6a894d5d IEFRA  
ME!EleDlg::ToggleExpando+0x20
01e2d74c 6a895254 IEFRA  
ME!EleDlg::OnInitDlg+0x229
01e2d7b8 762186ef IEFRA  
ME!EleDlg::DlgProcEx+0x189
01e2d7e4 76209eb2 USER32!InternalCallWinProc+0x23
01e2d860 7620bb98b USER32!UserCallDlgProcCheckWow+0xd6
01e2d8a8 7620bb7b USER32!DefDlgProcWorker+0xa8
01e2d8c4 762186ef USER32!DefDlgProcW+0x22
01e2d8f0 76218876 USER32!InternalCallWinProc+0x23
01e2d968 76217631 USER32!UserCallWinProcCheckWow+0x14b
01e2d9a8 76209b1d USER32!SendMessageWorker+0x4d0
01e2da64 76235500 USER32!InternalCreateDialog+0xb0d
01e2da94 76235553 USER32!InternalDialogBox+0xa7
01e2dab4 76235689 USER32!DialogBoxIndirectParamAorW+0x37
01e2dad8 6a5d4952 USER32!DialogBoxParamW+0x3f
01e2db00 6a5d5024 IEFRA  
ME!Detour_DialogBoxParamW+0x47
01e2db24 6a8956df IEFRA  
ME!SHFusionDialogBoxParam+0x32
01e2db58 6a8957bb IEFRA  
ME!EleDlg::ShowDialog+0x398
01e2e638 6a8959d3 IEFRA  
ME!ShowDialogBox+0xb6
01e2eb9c 6a9013ed IEFRA  
ME!ShowElevationPrompt+0x1dd
01e2f010 7669fc8f IEFRA  
ME!CIEUserBrokerObject::BrokerCoCreateInstance+0x202
01e2f040 76704c53 RPCRT4!Invoke+0x2a
01e2f448 76d9d936 RPCRT4!NdrStubCallL2+0x2d6
```

```

01e2f490 76d9d9c6 ole32!CStdStubBuffer_Invoke+0xb6
01e2f4d8 76d9df1f ole32!SyncStubInvoke+0x3c
01e2f524 76cb213c ole32!StubInvoke+0xb9
01e2f600 76cb2031 ole32!CCtxComChnl::ContextInvoke+0xfa
01e2f61c 76d9a754 ole32!MTAInvoke+0x1a
01e2f64c 76d9dcbb ole32!AppInvoke+0xab
01e2f72c 76d9a773 ole32!ComInvokeWithLockAndIPID+0x372
01e2f778 7669f34a ole32!ThreadInvoke+0x302
01e2f7b4 7669f4da RPCRT4!DispatchToStubInCNoAvrf+0x4a
01e2f80c 7669f3c6 RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x16c
01e2f834 766a0cef RPCRT4!RPC_INTERFACE::DispatchToStub+0x8b
01e2f86c 7669f882 RPCRT4!RPC_INTERFACE::DispatchToStubWithObject+0xb2
01e2f8b8 7669f7a4 RPCRT4!LRPC_CALL::DispatchRequest+0x23b
01e2f8d8 7669f763 RPCRT4!LRPC_CALL::QueueOrDispatchCall+0xbd
01e2f8f4 7669f5ff RPCRT4!LRPC_CALL::HandleRequest+0x34f
01e2f928 7669f573 RPCRT4!LRPC_SASSOCIATION::HandleRequest+0x144
01e2f960 7669ee4f RPCRT4!LRPC_ADDRESS::HandleRequest+0xbd
01e2f9dc 7669ece7 RPCRT4!LRPC_ADDRESS::ProcessIO+0x50a
01e2f9e8 766a1357 RPCRT4!LrpclServerIoHandler+0x16
01e2f9f8 76ecd3a3 RPCRT4!LrpclIoComplete+0x16
01e2fa20 76ed0748 ntdll!TppAlpcpExecuteCallback+0x1c5
01e2fb88 76e11174 ntdll!TppWorkerThread+0x5a4
01e2fb94 76efb3f5 kernel32!BaseThreadInitThunk+0xe
01e2fbfd4 76efb3c8 ntdll!_RtlUserThreadStart+0x70
01e2fbec 00000000 ntdll!_RtlUserThreadStart+0x1b

```

In the middle of the stack trace, we see COM interface invocation in *IEFRAME* module. The similar stack trace fragment can be found in the following stack trace where COM IRemUnknown interface implementation resides in .NET CLR *mscorwks* module:

```

0:000> kL
ChildEBP RetAddr
0018a924 68b5f8f0 mscorewks!SafeReleaseHelper+0x77
0018a958 68b04a99 mscorewks!SafeRelease+0x2f
0018a98c 68b04860 mscorewks!IUnkEntry::Free+0x68
0018a9a0 68b049b5 mscorewks!RCW::ReleaseAllInterfaces+0x18
0018a9d0 68b049e1 mscorewks!RCW::ReleaseAllInterfacesCallBack+0xbd
0018aa00 68c0a108 mscorewks!RCW::Cleanup+0x22
0018aa0c 68c0a570 mscorewks!RCWCleanupList::ReleaseRCWListRaw+0x16
0018aa3c 68bd4b3d mscorewks!RCWCleanupList::ReleaseRCWListInCorrectCtx+0xdf
0018aa4c 75dd8c2e mscorewks!CtxEntry::EnterContextCallback+0x89
0018aa68 763c586c ole32!CRemoteUnknown::DoCallback+0x7a
0018aa84 764405f1 rpcrt4!Invoke+0x2a
0018ae88 75efd936 rpcrt4!NdrStubCall2+0x2ea
0018aed0 75efd9c6 ole32!CStdStubBuffer_Invoke+0xb6
0018af18 75efdf1f ole32!SyncStubInvoke+0x3c
0018af64 75e1223c ole32!StubInvoke+0xb9
0018b040 75e12131 ole32!CCtxComChnl::ContextInvoke+0xfa
0018b05c 75e130fa ole32!MTAInvoke+0x1a
0018b088 75efde47 ole32!STAInvoke+0x46
0018b0bc 75efdccb ole32!AppInvoke+0xab
0018b19c 75efe34c ole32!ComInvokeWithLockAndIPID+0x372
0018b1c4 75e12ed2 ole32!ComInvoke+0xc5
0018b1d8 75e12e91 ole32!ThreadDispatch+0x23
0018b21c 75a06238 ole32!ThreadWndProc+0x161

```

```

0018b248 75a068ea user32!InternalCallWinProc+0x23
0018b2c0 75a07d31 user32!UserCallWinProcCheckWow+0x109
0018b320 75a07dfa user32!DispatchMessageWorker+0x3bc
0018b330 75ddd6be user32!DispatchMessageW+0xf
0018b360 75ddd66d ole32!CCliModalLoop::PeekRPCAndDDEMessage+0x4c
0018b390 75ddd57e ole32!CCliModalLoop::FindMessage+0x30
0018b3f0 75ddd633 ole32!CCliModalLoop::HandleWakeForMsg+0x41
0018b408 75dd1117 ole32!CCliModalLoop::BlockFn+0xc3
0018b488 68a6c905 ole32!CoWaitForMultipleHandles+0xcd
0018b4a8 68a6c866 mscorewks!NT5WaitRoutine+0x51
0018b514 68a6c7ca mscorewks!MsgWaitHelper+0xa5
0018b534 68b5fbe4 mscorewks!Thread::DoAppropriateAptStateWait+0x28
0018b5b8 68b5fc79 mscorewks!Thread::DoAppropriateWaitWorker+0x13c
0018b608 68b5fdf9 mscorewks!Thread::DoAppropriateWait+0x40
0018b664 68a1c5b6 mscorewks!CLREvent::WaitEx+0xf7
0018b678 68b1adb4 mscorewks!CLREvent::Wait+0x17
0018b6c8 68b1ab2a mscorewks!WKS::GCHeap::FinalizerThreadWait+0xfb
0018b764 08fa12c1 mscorewks!GCInterface::RunFinalizers+0x99
[...]

```

A TSST usually spans several modules. In any stack trace, we can also find several TSST that may be overlapping. For example, in the first stack trace above we can discern fragments of COM, RPC, LPC, GUI Dialog, Window Management, and Window Messaging subtraces. In the second trace, we can also see GC, Modal Loop, COM Wrapper, and Interface Management stack frames.

The closest software trace analysis pattern here is **Implementation Discourse**<sup>209</sup>.

---

<sup>209</sup> Implementation Discourse, Memory Dump Analysis Anthology, Volume 6, page 245

## Comments

---

Another example:

```
App!foo
rpcrt4!Invoke
rpcrt4!NdrStubCall12
combase!CStdStubBuffer_Invoke
oleaut32!CUnivStubWrapper::Invoke
combase!SyncStubInvoke
combase!StubInvoke
combase!CCTxComChnl::ContextInvoke
combase!DefaultInvokeInApartment
combase!ClassicSTAInvokeInApartment
combase!AppInvoke
combase!ComInvokeWithLockAndIPID
combase!ComInvoke
combase!ThreadDispatch
combase!ThreadWndProc
user32!_InternalCallWinProc
user32!UserCallWinProcCheckWow
user32!DispatchMessageWorker
user32!DispatchMessageA
App!bar
kernel32!BaseThreadInitThunk
```

## Dynamic Memory

We consider dynamic memory allocation example in kernel space (kernel pool). Usually, the pool corruption is detected during pool memory allocation or release with a special bugcheck code, for example:

```
BAD_POOL_HEADER (19)
The pool is already corrupt at the time of the current request.
This may or may not be due to the caller.
The internal pool links must be walked to figure out a possible cause of the problem, and then special
pool applied to the suspect tags or the driver verifier to a suspect driver.
Arguments:
Arg1: 00000020, a pool block header size is corrupt.
Arg2: 8b79d078, The pool entry we were looking for within the page.
Arg3: 8b79d158, The next pool entry.
Arg4: 8a1c0004, (reserved)
```

However, pool corruption might be deeper enough to trigger an access violation even before **Self-Diagnosis** (page 844). In such cases, stack subtraces with functions like *ExFreePoolWithTag* might point to troubleshooting and debugging directions:

```
ATTEMPTED_WRITE_TO_READONLY_MEMORY (be)
An attempt was made to write to readonly memory. The guilty driver is on the stack trace (and is
typically the current instruction pointer).
When possible, the guilty driver's name (Unicode string) is printed on the bugcheck screen and saved in
KiBugCheckDriver.
Arguments:
Arg1: 00470044, Virtual address for the attempted write.
Arg2: 06d39025, PTE contents.
Arg3: aec0fb30, (reserved)
Arg4: 0000000a, (reserved)

TRAP_FRAME: aec0fb30 -- (.trap 0xfffffffffaec0fb30)
ErrCode = 00000003
eax=8ac12d38 ebx=8b700040 ecx=000001ff edx=00470040 esi=8ac12db8 edi=808b0b40
eip=808949e7 esp=aec0fba4 ebp=aec0fb0 iopl=0 nv up ei pl nz na po nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010202
nt!ExFreePoolWithTag+0x6a3:
808949e7 895a04 mov dword ptr [edx+4],ebx ds:0023:00470044=?????????

STACK_TEXT:
aec0faa0 80860121 000000be 00470044 06d39025 nt!KeBugCheckEx+0x1b
aec0fb18 8088e490 00000001 00470044 00000000 nt!MmAccessFault+0xb25
aec0fb18 808949e7 00000001 00470044 00000000 nt!KiTrap0E+0xdc
aec0fb0 808d93b5 8ac12dc0 00000000 00000000 nt!ExFreePoolWithTag+0x6a3
aec0fc08 808cd304 e5ae5770 8ac12dc0 8aa77db0 nt!CmpFreePostBlock+0x4d
aec0fc3c 8082ea53 8ac12dc0 aec0fc88 aec0fc7c nt!CmpPostApc+0xde
aec0fc8c 80833eec 00000000 00000000 00000000 nt!KiDeliverApc+0xf9
aec0fcc4 808290bd aec0fd64 8099781c 0160fd44 nt!KiSwapThread+0x300
aec0fd0c 809978a0 00000001 00000000 f77275e0 nt!KeDelayExecutionThread+0x2ab
aec0fd54 8088b45c 00000000 0160fd74 0160fd9c nt!NtDelayExecution+0x84
aec0fd54 7c82847c 00000000 0160fd74 0160fd9c nt!KiFastCallEntry+0xfc
```

WARNING: Frame IP not in any known module. Following frames may be wrong.

0160fd9c 00000000 00000000 00000000 00000000 0x7c82847c

```
1: kd> !pool 8ac12dc0
Pool page 8ac12dc0 region is Nonpaged pool
8ac12000 size: 858 previous size: 0 (Allocated) TWPG
8ac12858 size: 8 previous size: 858 (Free) ....
8ac12860 size: 20 previous size: 8 (Allocated) VadS
8ac12880 size: 8 previous size: 20 (Free) NtFs
8ac12888 size: 20 previous size: 8 (Allocated) VadS
8ac128a8 size: 28 previous size: 20 (Allocated) Ntfn
8ac128d0 size: 30 previous size: 28 (Allocated) Vad
8ac12900 size: 40 previous size: 30 (Allocated) Muta (Protected)
8ac12940 size: 38 previous size: 40 (Allocated) Sema (Protected)
8ac12978 size: 40 previous size: 38 (Allocated) Muta (Protected)
8ac129b8 size: 270 previous size: 40 (Allocated) Thre (Protected)
8ac12c28 size: 40 previous size: 270 (Allocated) Ntfr
8ac12c68 size: d0 previous size: 40 (Allocated) DRIV
8ac12d38 is not a valid large pool allocation, checking large session pool...
8ac12d38 is freed (or corrupt) pool
Bad previous allocation size @8ac12d38, last size was 1a
```

```
***  
*** An error (or corruption) in the pool was detected;  
*** Attempting to diagnose the problem.  
***  
*** Use !poolval 8ac12000 for more details.  
***
```

Pool page [ 8ac12000 ] is \_\_inVALID.

```
Analyzing linked list...
[ 8ac12c68 --> 8ac12db8 (size = 0x150 bytes)]: Corrupt region
Scanning for single bit errors...
```

None found

## JIT .NET Code

When looking at process memory dumps and seeing **CLR Threads** (page 124), we can find fragments of JIT-ed code return addresses on the unmanaged stack trace:

```
0:011> kL
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
0b73e120 057223e2 0x572240f
0b73e134 6af44a2a 0x57223e2
0b73e1b0 6af44bcc clr!CallDescrWorkerWithHandler+0x8e
0b73e2f0 6af44c01 clr!MethodDesc::CallDescr+0x194
0b73e30c 6af44c21 clr!MethodDesc::CallTargetWorker+0x21
0b73e324 6afb7856 clr!MethodDescCallSite::Call+0x1c
0b73e4e8 6afb7ba3 clr!CallWithValueTypes_RetArgSlotWrapper+0x5c
0b73e7b4 6afb7d65 clr!InvokeImpl+0x621
0b73e880 6963d689 clr!RuntimeMethodHandle::InvokeMethodFast+0x180
0b73e8d4 6963d3d0 mscorelib_ni+0x2bd689
0b73e90c 6963bfed mscorelib_ni+0x2bd3d0
0b73e934 69643284 mscorelib_ni+0x2bbfed
0b73e958 6af3de7e mscorelib_ni+0x2c3284
0b73eb64 05720988 clr!ListLockEntry::Release+0x68
0b73ebc0 6962ae5b 0x5720988
0b73ebd0 695b7ff4 mscorelib_ni+0x2aae5b
0b73ebec 695b7f34 mscorelib_ni+0x237ff4
0b73ec0c 6962ade8 mscorelib_ni+0x237f34
0b73ec24 6af221db mscorelib_ni+0x2aad8
0b73ec34 6af44a2a clr!CallDescrWorker+0x33
0b73ecb0 6af44bcc clr!CallDescrWorkerWithHandler+0x8e
0b73ede8 6af44c01 clr!MethodDesc::CallDescr+0x194
0b73ee04 6b0bb512 clr!MethodDesc::CallTargetWorker+0x21
0b73f010 6afd5c05 clr!ThreadNative::KickOffThread_Worker+0x1e1
0b73f024 6afd5c87 clr!Thread::DoExtraWorkForFinalizer+0x114
0b73f0d4 6afd5d42 clr!Thread::ShouldChangeAbortToUnload+0x101
0b73f134 6afc37a2 clr!Thread::ShouldChangeAbortToUnload+0x399
0b73f140 6b0a6465 clr!Thread::RaiseCrossContextException+0x3f8
0b73f220 6afc37cf clr!Thread::DoADCallBack+0xf0
0b73f240 6afd5c87 clr!Thread::DoExtraWorkForFinalizer+0xfa
0b73f2f0 6afd5d42 clr!Thread::ShouldChangeAbortToUnload+0x101
0b73f350 6afd5dd9 clr!Thread::ShouldChangeAbortToUnload+0x399
0b73f374 6b0bb3e5 clr!Thread::ShouldChangeAbortToUnload+0x43a
0b73f38c 6b0bb2e0 clr!ManagedThreadBase::KickOff+0x15
0b73f424 6afd5a08 clr!ThreadNative::KickOffThread+0x23e
0b73fb44 76573833 clr!Thread::intermediateThreadProc+0x4b
0b73fb50 77c1a9bd kernel32!BaseThreadInitThunk+0xe
```

With the correct loaded CLR **Version-Specific Extension** (page 1058) we can inspect these addresses and get their method names, module and class addresses using **!IP2MD** WinDbg SOS extension command:

```

0:011> !IP2MD 0x572240f
MethodDesc: 057420e8
Method Name: UserQuery+ClassMain.Main()
Class: 057341d8
MethodTable: 05742108
mdToken: 06000004
Module: 05741048
IsJitted: yes
CodeAddr: 05722400
Transparency: Critical

0:011> !IP2MD 0x57223e2
MethodDesc: 0574204c
Method Name: UserQuery.RunUserAuthoredQuery()
Class: 057340a4
MethodTable: 0574206c
mdToken: 06000001
Module: 05741048
IsJitted: yes
CodeAddr: 057223d0
Transparency: Critical

0:011> !IP2MD 0x5720988
MethodDesc: 056e601c
Method Name: LINQPad.ExecutionModel.Server.StartClrQuery()
Class: 0571f6e4
MethodTable: 056e60e4
mdToken: 06000c59
Module: 056e336c
IsJitted: yes
CodeAddr: 05720910
Transparency: Critical

```

These method calls can also be seen on **Managed Stack Trace** (page 624):

```

0:011> !CLRStack
OS Thread Id: 0xac (11)
Child SP IP Call Site
0b73e120 0572240f UserQuery+ClassMain.Main()
0b73e128 057223e2 UserQuery.RunUserAuthoredQuery()
0b73e674 6af221db [DebuggerU2MCatchHandlerFrame: 0b73e674]
0b73e640 6af221db [CustomGCFrame: 0b73e640]
0b73e614 6af221db [GCFrame: 0b73e614]
0b73e5f8 6af221db [GCFrame: 0b73e5f8]
0b73e81c 6af221db [HelperMethodFrame_PROTECTOBJ: 0b73e81c]
System.RuntimeMethodHandle._InvokeMethodFast(System.IRuntimeMethodInfo, System.Object, System.Object[], System.SignatureStruct ByRef, System.Reflection.MethodAttributes, System.RuntimeType)
0b73e898 6963d689 System.RuntimeMethodHandle.InvokeMethodFast(System.IRuntimeMethodInfo, System.Object, System.Object[], System.Signature, System.Reflection.MethodAttributes, System.RuntimeType)
0b73e8ec 6963d3d0 System.Reflection.RuntimeMethodInfo.Invoke(System.Object, System.Reflection.BindingFlags, System.Reflection.Binder, System.Object[], System.Globalization.CultureInfo, Boolean)
0b73e928 6963bfed System.Reflection.RuntimeMethodInfo.Invoke(System.Object, System.Reflection.BindingFlags, System.Reflection.Binder, System.Object[], System.Globalization.CultureInfo)

```

```
0b73e94c 69643284 System.Reflection.MethodBase.Invoke(System.Object, System.Object[])
0b73e958 0572134c LINQPad.ExecutionModel.Server.RunClrQuery()
0b73eb6c 05720988 LINQPad.ExecutionModel.Server.StartClrQuery()
0b73ebc8 6962ae5b System.Threading.ThreadHelper.ThreadStart_Context(System.Object)
0b73ebd8 695b7ff4 System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object, Boolean)
0b73ebfc 695b7f34 System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
0b73ec18 6962ade8 System.Threading.ThreadHelper.ThreadStart()
0b73ee30 6af221db [GCFrame: 0b73ee30]
0b73f0f4 6af221db [DebuggerU2MCatchHandlerFrame: 0b73f0f4]
0b73f18c 6af221db [ContextTransitionFrame: 0b73f18c]
0b73f310 6af221db [DebuggerU2MCatchHandlerFrame: 0b73f310]
```

## Template Module

It is a common technique in driver development to copy/paste an entire driver sample from WDK and modify it for a specific device or a filter value-adding functionality in a specific driver stack or framework. The problem here is that developers sometimes forget to change module resources and a certain amount of detective work is required to find out the module vendor. Here is an example. In a spooler service there were many **Blocked Threads** (page 82) in displaying **Dialog Box** (page 230):

```
0:000> ~34KL 100
Child-SP      RetAddr          Call Site
00000000`08a49368 00000000`774c5118 user32!NtUserWaitMessage+0xa
00000000`08a49370 00000000`774c5770 user32!DialogBox2+0x261
00000000`08a493f0 00000000`774c57e6 user32!InternalDialogBox+0x134
00000000`08a49450 00000000`774c3e36 user32!DialogBoxIndirectParamAorW+0x58
00000000`08a49490 000007fe`fa27cc97 user32!DialogBoxParamW+0x66
00000000`08a494d0 000007fe`fa28832b unidrvui!ICheckConstraintsDlg+0xbff
00000000`08a49950 000007fe`fa29423d unidrvui!BUpdateUISettingForOEM+0x2f
00000000`08a49980 00000000`50036d2c unidrvui!CPrintOemDriverUI::DrvUpdateUISetting+0x1d
00000000`08a499b0 00000000`50038a1d ModuleZ!DLLGetClassObject+0x1fe74
[...]
00000000`08a4b250 000007fe`f759546b unidrvui!OEMDevicePropertySheets+0x56
00000000`08a4b280 000007fe`f759653e compstui!CallpfnPSUI+0x137
00000000`08a4b330 000007fe`f7596b84 compstui!InsertPSUIPage+0x24a
00000000`08a4b5f0 000007fe`fa2880e9 compstui!CPSUICallBack+0x3ec
00000000`08a4b6a0 000007fe`fa2836c4 unidrvui!BAddOemPluginPages+0x12d
00000000`08a4b6d0 000007fe`f759546b unidrvui!DrvDevicePropertySheets+0x2c8
00000000`08a4bb60 000007fe`f759653e compstui!CallpfnPSUI+0x137
00000000`08a4bc10 000007fe`f7596b84 compstui!InsertPSUIPage+0x24a
00000000`08a4bed0 000007fe`fb452838 compstui!CPSUICallBack+0x3ec
00000000`08a4bf80 000007fe`f759546b winspool!DevicePropertySheets+0x108
00000000`08a4fbf0 000007fe`f759653e compstui!CallpfnPSUI+0x137
00000000`08a4c060 000007fe`f7596b84 compstui!InsertPSUIPage+0x24a
00000000`08a4c320 000007fe`f759758e compstui!CPSUICallBack+0x3ec
00000000`08a4c3d0 000007fe`f75976b2 compstui!DoCommonPropertySheetUI+0xbe
00000000`08a4c430 000007fe`fb446339 compstui!CommonPropertySheetUIW+0xe
00000000`08a4c470 000007fe`fb44b425 winspool!CallCommonPropertySheetUI+0x65
00000000`08a4c4c0 00000000`5003623c winspool!PrinterPropertiesNative+0x121
00000000`08a4c950 00000000`50035d16 ModuleZ!DLLGetClassObject+0x1f384
[...]
00000000`08a4dd70 000007fe`fb4472d8 unidrvui!DrvPrinterEvent+0x419
00000000`08a4de00 000007fe`fb44737f winspool!SpoolerPrinterEventNative+0x84
00000000`08a4de60 000007fe`faedc957 winspool!SpoolerPrinterEvent+0x13
00000000`08a4dea0 000007fe`faedc8c7 localspl!SplDriverEvent+0x4f
00000000`08a4def0 000007fe`faec3d74 localspl!PrinterDriverEvent+0xcf
00000000`08a4df30 000007fe`fa771f20 localspl!SplAddPrinter+0xae0
00000000`08a4e4e0 000007fe`fa7491d8 win32spl!NCSRCommon::TLocalPrinter::AddPrinterW+0xb4
00000000`08a4e5b0 000007fe`fa747511 win32spl!TPrintOpen::AddLocalPrinter+0xb8
00000000`08a4e6b0 000007fe`fa746dfb win32spl!TPrintOpen::AddAndInstallLocalPrinter+0x34d
00000000`08a4e830 000007fe`fa746bb0 win32spl!TPrintOpen::ReEstablishCacheConnectionNoGuidPrinter+0x157
00000000`08a4e900 000007fe`fa7467d1 win32spl!TPrintOpen::ReEstablishCacheConnection+0x178
00000000`08a4e980 000007fe`fa7465c1 win32spl!TPrintOpen::ReEstablishPrinterConnection+0x16d
00000000`08a4ea30 000007fe`fa73e5ad win32spl!TPrintOpen::ReEstablishConnectionFromKey+0x1fd
00000000`08a4eb30 000007fe`fa733492 win32spl!TPrintOpen::RediscoverPrinterConnections+0xd7
```

```
00000000`08a4ebe0 000007fe`fb3f2332 win32spl!TPrintProviderTable::forwardEnumPrinters+0x47
00000000`08a4ec70 00000000`ff3414c8 spoolss!EnumPrintersW+0x176
00000000`08a4ed20 00000000`ff3413cc spools!YEnumPrinters+0x112
00000000`08a4eda0 000007fe`fe225ec5 spools!RpcEnumPrinters+0x30
00000000`08a4edf0 000007fe`fe2beb6d rpcrt4!Invoke+0x65
00000000`08a4ee70 000007fe`fe1f5df0 rpcrt4!Ndr64StubWorker+0x5a9
00000000`08a4f440 000007fe`fe2268d4 rpcrt4!NdrServerCallAll+0x40
00000000`08a4f490 000007fe`fe2269f0 rpcrt4!DispatchToStubInCNoAvrf+0x14
00000000`08a4f4c0 000007fe`fe227402 rpcrt4!RPC_INTERFACE::DispatchToStubWorker+0x100
00000000`08a4f5b0 000007fe`fe227080 rpcrt4!LRPC_SCALL::DispatchRequest+0x1c2
00000000`08a4f620 000007fe`fe2262bb rpcrt4!LRPC_SCALL::HandleRequest+0x200
00000000`08a4f740 000007fe`fe225e1a rpcrt4!LRPC_ADDRESS::ProcessIO+0x44a
00000000`08a4f860 000007fe`fe207769 rpcrt4!LOADABLE_TRANSPORT::ProcessIOEvents+0x24a
00000000`08a4f910 000007fe`fe207714 rpcrt4!ProcessIOEventsWrapper+0x9
00000000`08a4f940 000007fe`fe2077a4 rpcrt4!BaseCachedThreadRoutine+0x94
00000000`08a4f980 00000000`7758be3d rpcrt4!ThreadStartRoutine+0x24
00000000`08a4f9b0 00000000`776c6a51 kernel32!BaseThreadInitThunk+0xd
00000000`08a4f9e0 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

We suspect *ModuleZ* but its module information points to Microsoft:

```
0:000> lmv m ModuleZ
start           end             module name
00000000`50000000 00000000`500a4000  ModuleZ    (export symbols)      ModuleZ.DLL
Loaded symbol image file: ModuleZ.dll
Image path: C:\Windows\System32\spool\drivers\x64\3\ModuleZ.dll
Image name: ModuleZ.dll
Timestamp:      Feb [...] 2010
[...]
File version:   2.5.0.0
Product version: 2.5.0.0
File flags:     0 (Mask 3F)
File OS:        40004 NT Win32
File type:      2.0 Dll
File date:      00000000.00000000
Translations:   0407.04b0
CompanyName:    Microsoft Corp.
ProductName:    Microsoft PS UI Replacement Sample
InternalName:   PSUIREP
OriginalFilename: PSUIREP.dll
ProductVersion: 2.5
FileVersion:    2.5.0.0
FileDescription: PS UI Replacement Sample
LegalCopyright: Copyright © 1998 - 2009 Microsoft Corp.
LegalTrademarks: Microsoft® is a registered trademark of Microsoft Corporation. Windows(TM) is a
trademark of Microsoft Corporation
Comments:       Written by Windows Printing & Imaging Team
```

Having never seen *ModuleZ* in Microsoft module lists and suspecting the word “Sample” in a file and product description we did an Internet search and found the module name on various “DLL fixing” websites but still pointing to Microsoft in module description. However, in a full module list (**!lmt** WinDbg command) we found more modules having Module\* name structure:

```

0:000> lmv m ModuleC
start           end             module name
00000000`10000000 00000000`100b7000  ModuleC  (deferred)
Image path: C:\Windows\System32\spool\drivers\x64\3\ModuleC.DLL
Image name: ModuleC.DLL
Timestamp:      Feb [...] 2010
[...]
File version:   20.1.0.0
Product version: 20.1.0.0
File flags:     0 (Mask 17)
File OS:        4 Unknown Win32
File type:      2.0 Dll
File date:      00000000.00000000
Translations:   0409.04b0
CompanyName:
ProductName:    Printer Driver
InternalName:   MC.dll
OriginalFilename: MC.dll
ProductVersion: 20.1.0.0
FileVersion:    20.1.0.0
FileDescription: Printer Driver
LegalCopyright:

0:000> lmv m ModuleO
start           end             module name
00000000`6f280000 00000000`6f2e2000  ModuleO  (deferred)
Image path: C:\Windows\System32\spool\drivers\x64\3\ModuleO.DLL
Image name: ModuleO.DLL
Timestamp:      Feb [...] 2010
[...]
File version:   2.4.0.0
Product version: 2.4.0.0
File flags:     8 (Mask 3F) Private
File OS:        40004 NT Win32
File type:      3.1 Driver
File date:      00000000.00000000
Translations:   0409.04b0
CompanyName:    CompanyA
ProductName:    CompanyA Printer driver
InternalName:   ModuleO.dll
OriginalFilename: ModuleO.dll
ProductVersion: 2.4
FileVersion:    2.4.0.0
FileDescription: CompanyA Printer driver
LegalCopyright: Copyright © CompanyA
Comments:

```

We see that both module names and time stamps follow the same pattern, so our “Microsoft” *ModuleZ* is definitely from *CompanyA* instead. We also check more detailed information:

```
0:000> !lmi 00000000`50000000
Loaded Module Info: [00000000`50000000]
    Module: ModuleZ
[...]
    Pdb: N:\ServerQ\ [...]
[...]

0:000> !lmi 00000000`10000000
Loaded Module Info: [00000000`10000000]
    Module: ModuleC
[...]
    Pdb: N:\ServerQ\ [...]
[...]

0:000> !lmi 00000000`6f280000
Loaded Module Info: [00000000`6f280000]
    Module: ModuleO
[...]
    Pdb: N:\ServerQ\ [...]
[...]
```

All three modules have the same build server in their PDB file name path. We advise contacting *CompanyA* for updates.

## Thread Age

Sometimes, it is useful to know how much time ago a thread was created in order to understand when other behavioral patterns possibly started to appear. For example, in the user process memory dumps with saved thread time information we can see the latter using **!runaway** WinDbg extension command or using **.ttime** command. Looking at **Stack Trace Collection** (page 943) we notice a thread blocked in LPC call (**Wait Chain**, page 1097):

```
0:000> ~40 kc
Call Site
ntdll!NtAlpcSendWaitReceivePort
rpcrt4!LRPC_CCALL::SendReceive
rpcrt4!NdrpClientCall13
rpcrt4!NdrClientCall13
[...]
kernel32!BaseThreadInitThunk
ntdll!RtlUserThreadStart
```

We are interested when all this started because we want to compare with other **Coupled Process** (page 149) memory dumps saved at different times:

```
0:000> !runaway f
User Mode Time
[...]
Kernel Mode Time
Thread Time
[...]
Elapsed Time
Thread Time
0:8ac 4 days 11:42:14.484
1:8b4 4 days 11:42:14.296
[...]
35:868 4 days 10:18:48.255
36:73ec 0 days 15:55:31.938
37:c0bc 0 days 10:36:53.447
38:782c 0 days 0:02:01.683
39:5864 0 days 0:00:52.236
40:5ffc 0 days 0:00:02.565

0:000> ~40s
ntdll!NtAlpcSendWaitReceivePort+0xa:
00000000`76d3ff0a c3 ret

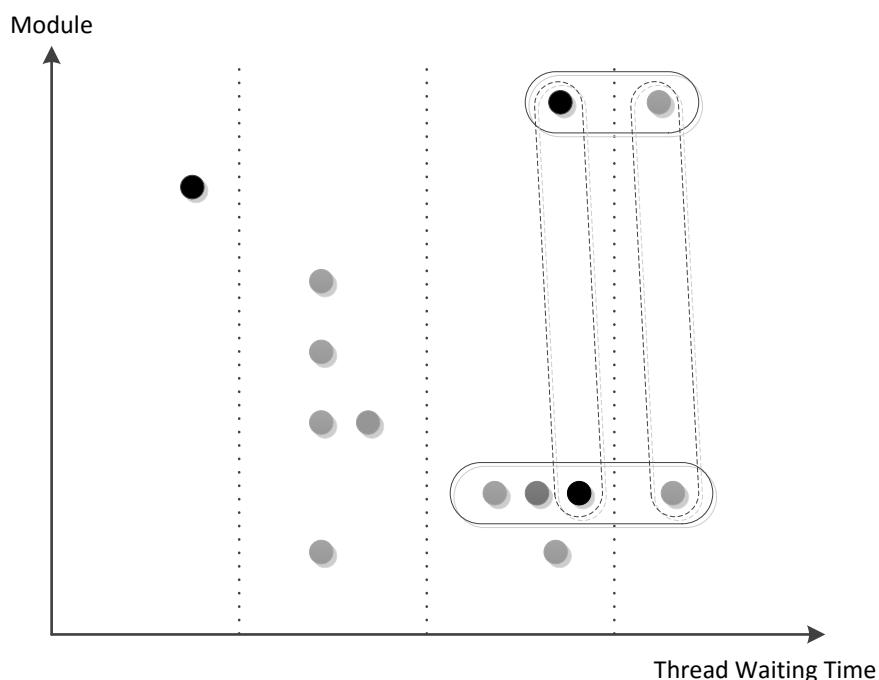
0:040> .ttime
Created: Tue Jun 14 15:15:28.444 2011
Kernel: 0 days 0:00:00.000
User: 0 days 0:00:00.000
```

```
0:040> .time
Debug session time: Tue Jun 14 15:15:31.000 2011
System Uptime: 4 days 11:43:21.389
Process Uptime: 4 days 11:42:15.000
Kernel time: 0 days 0:00:10.000
User time: 0 days 0:04:22.000
```

We see that our blocked thread had only recently started compared to other threads and actually started after other dumps were saved when we look at their debug session time.

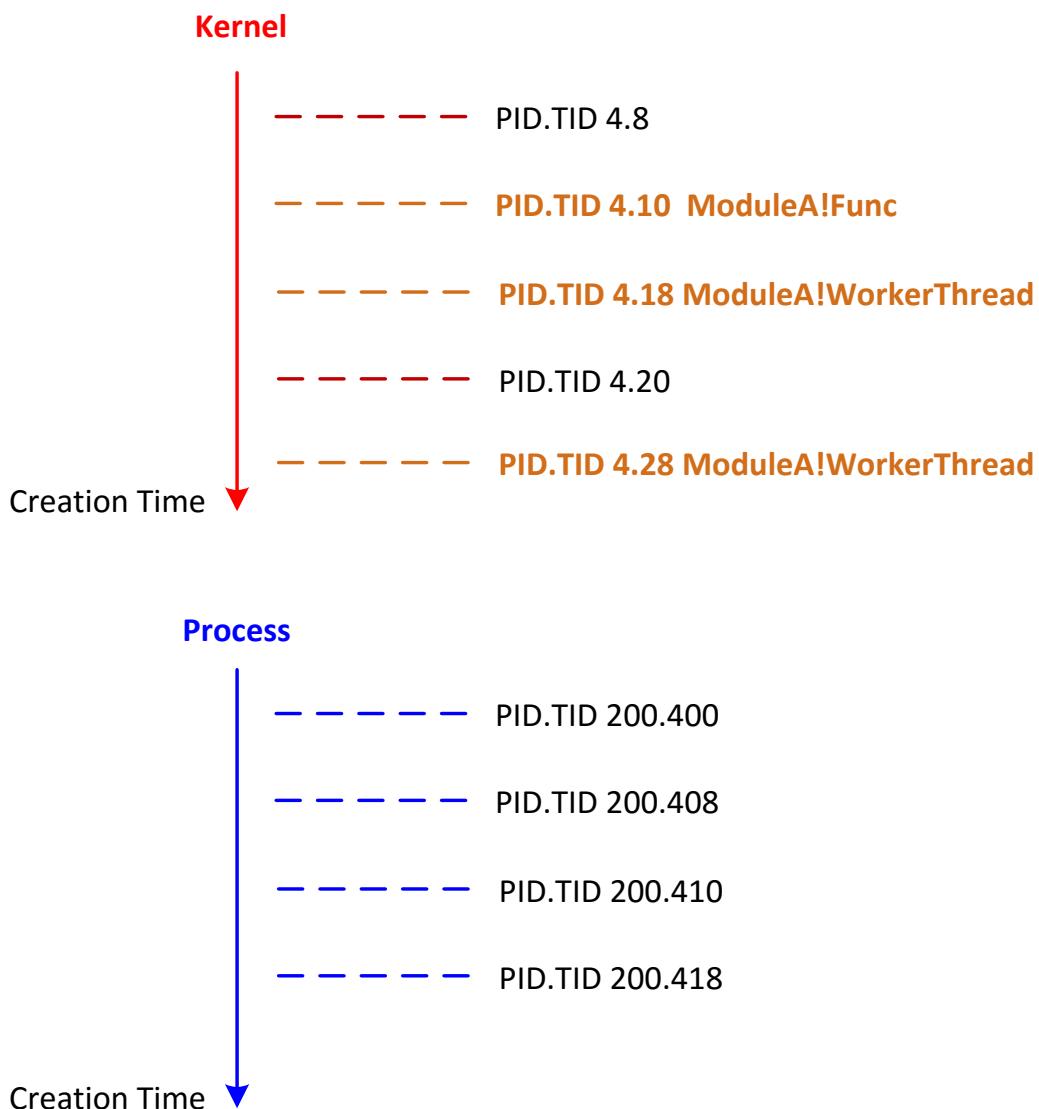
## Thread Cluster

One of the useful patterns for analysis of system hangs is **Waiting Thread Time** (page 1137). If there are many such threads of interest, they can be partitioned by waiting time and modules of interest (page 1175) from their **Stack Traces** (page 926). Modules of interest may include **Directing Modules** (page 235), **Coupled Modules** (page 147), **Blocking Modules** (page 96), **Top Modules** (page 1012), and **Problem Modules** (page 812) depending on the problem description. Extra-dimensional information can also be added such as the number of threads having the same or similar waiting time and other attributes by using different colors. For example, on the following diagram illustrating a real system hang we see clustering of threads running through one 3rd-party module of interest and having the longest waiting time. Also, we are able to identify possibly coupled (semantically related) threads running through another module of interest:



## Thread Poset

**Predicate Stack Trace Collections** (page 943) allow us to get a subset of stack traces, for example, by showing only stack traces where a specific module is used (for example, `!stacks 2 module` WinDbg command). From a diagnostic analysis perspective, the order in which threads from the subset appear is also important, especially when the output is sorted by thread creation time or simply the order is given by a global thread linked list. We call this analysis pattern **Thread Poset** by analogy with a mathematical concept of poset (partially ordered set<sup>210</sup>):



<sup>210</sup> [http://en.wikipedia.org/wiki/Partially\\_ordered\\_set](http://en.wikipedia.org/wiki/Partially_ordered_set)

Such an analysis pattern is mostly useful when we compare stack traces for differences or when we don't have symbols for some problem version and want to map threads to some other previous normal run where symbol files are available. Any discrepancies may point in the direction of further diagnostic analysis. For example, we got this fragment of **Stack Trace Collection** (page 943):

```
4.000188 ffffffa800d3d3b50 ffd0780f Blocked ModuleA+0x1ac1
4.00018c ffffffa800d3f9950 ffd07b53 Blocked ModuleA+0xd802
4.000190 ffffffa800d4161b0 fffffdfa6 Blocked ModuleA+0x9ce4
4.000194 ffffffa800d418b50 fffffdfa6 Blocked ModuleA+0x9ce4
4.000198 ffffffa800d418660 fffffdfa6 Blocked ModuleA+0x9ce4
4.0001ac ffffffa800d41eb50 ffd078d2 Blocked ModuleA+0xa7cf
4.0001b0 ffffffa800d41e660 ffd0780f Blocked ModuleA+0x9ce4
4.0001c0 ffffffa800d48f300 ffd0e5c0 Blocked ModuleA+0x7ee5
```

We didn't have symbols, and, therefore, didn't know whether there was anything wrong with those threads. Fortunately, we had **Thread Poset** from an earlier 32-bit version with available symbol files:

```
4.0000ec 85d8dc58 000068c Blocked ModuleA!FuncA+0x9b
4.0000f0 85d9fc78 001375a Blocked ModuleA!FuncB+0x67
4.0000fc 85db8a58 000068c Blocked ModuleA!WorkerThread+0xa2
4.000104 85cdbd48 000ff44 Blocked ModuleA!WorkerThread+0xa2
4.000108 85da2788 000ff47 Blocked ModuleA!WorkerThread+0xa2
4.000110 857862e0 0013758 Blocked ModuleA!FuncC+0xe4
4.000114 85dda250 000ff44 Blocked ModuleA!FuncD+0xf2
```

If we map worker threads to the middle section of the x64 version we see just one more worker thread, but the overall order is the same:

```
4.000188 ffffffa800d3d3b50 ffd0780f Blocked ModuleA+0x1ac1
4.00018c ffffffa800d3f9950 ffd07b53 Blocked ModuleA+0xd802
4.000190 ffffffa800d4161b0 fffffdfa6 Blocked ModuleA+0x9ce4
4.000194 ffffffa800d418b50 fffffdfa6 Blocked ModuleA+0x9ce4
4.000198 ffffffa800d418660 fffffdfa6 Blocked ModuleA+0x9ce4
4.0001ac ffffffa800d41eb50 ffd078d2 Blocked ModuleA+0xa7cf
4.0001b0 ffffffa800d41e660 ffd0780f Blocked ModuleA+0x9ce4
4.0001c0 ffffffa800d48f300 ffd0e5c0 Blocked ModuleA+0x7ee5
```

So we may think of x64 **Thread Poset** as normal if x86 **Thread Poset** is normal too. Of course, only initially, then to continue looking for other patterns of abnormal behavior. If necessary, we may need to inspect stack traces deeper because individual threads from two **Thread Posets** may differ in their stack trace depth, subtraces, and in the usage of other components. Despite the same order, some threads may actually be abnormal.

## Thread Starvation

### Normal Priority

Here we show the possible signs of the classical **Thread Starvation**. Suppose we have two running threads with a priority 8:

```
0: kd> !running

System Processors 3 (affinity mask)
  Idle Processors 0

Prcbs Current Next
 0  ffdff120  89a92020      0.....
 1  f7737120  89275020      W.....
[...]
Priority 8 BasePriority 8 PriorityDecrement 0

0: kd> !thread 89a92020
THREAD 89a92020 Cid 11d8.27d8 Teb: 7ffd9000 Win32Thread: bc1e6860 RUNNING on processor 0
[...]
Priority 8 BasePriority 8 PriorityDecrement 0

0: kd> !thread 89275020
THREAD 89275020 Cid 1cd0.2510 Teb: 7ffa9000 Win32Thread: bc343180 RUNNING on processor 1
[...]
Priority 8 BasePriority 8 PriorityDecrement 0
```

If we have other threads ready with the same priority contending for the same processors (**High Contention**, page 480) other threads with less priority might starve (shown in bold italicics):

```
0: kd> !ready
Processor 0: Ready Threads at priority 8
  THREAD 894a1db0 Cid 1a98.25c0 Teb: 7ffde000 Win32Thread: bc19cea8 READY
  THREAD 897c4818 Cid 11d8.1c5c Teb: 7ffa2000 Win32Thread: bc2c5ba8 READY
  THREAD 8911fd18 Cid 2730.03f4 Teb: 7ffd9000 Win32Thread: bc305830 READY
Processor 0: Ready Threads at priority 7
  THREAD 8a9e5ab0 Cid 0250.0470 Teb: 7ff9f000 Win32Thread: 00000000 READY
  THREAD 8a086838 Cid 0250.0654 Teb: 7ff93000 Win32Thread: 00000000 READY
  THREAD 8984b8b8 Cid 0250.1dc4 Teb: 7ff99000 Win32Thread: 00000000 READY
  THREAD 8912a4c0 Cid 0f4c.2410 Teb: 7ff81000 Win32Thread: 00000000 READY
  THREAD 89e5c570 Cid 0f4c.01c8 Teb: 00000000 Win32Thread: 00000000 READY
Processor 0: Ready Threads at priority 6
  THREAD 8a9353b0 Cid 1584.1598 Teb: 7ff8b000 Win32Thread: bc057698 READY
  THREAD 8aba2020 Cid 1584.15f0 Teb: 7ff9f000 Win32Thread: bc2a0ea8 READY
  THREAD 8aab17a0 Cid 1584.01a8 Teb: 7ff92000 Win32Thread: bc316ea8 READY
  THREAD 8a457020 Cid 1584.0634 Teb: 7ff8d000 Win32Thread: bc30fea8 READY
  THREAD 8a3d4020 Cid 1584.1510 Teb: 7ff8f000 Win32Thread: bc15b8a0 READY
  THREAD 8a5f5db0 Cid 1584.165c Teb: 7ff9d000 Win32Thread: bc171be8 READY
  THREAD 8a297020 Cid 0f4c.0f54 Teb: 7ffd000 Win32Thread: bc20fda0 READY
  THREAD 8a126020 Cid 1584.175c Teb: 7ffa9000 Win32Thread: 00000000 READY
  THREAD 8a548478 Cid 0250.07b0 Teb: 7ff9a000 Win32Thread: 00000000 READY
  THREAD 8a478020 Cid 0944.0988 Teb: 7ffd9000 Win32Thread: 00000000 READY
```

```

THREAD 8986ad08 Cid 1d2c.1cf0 Teb: 7ffa8000 Win32Thread: bc285800 READY
THREAD 897f4db0 Cid 1d2c.2554 Teb: 7ffdb000 Win32Thread: bc238e80 READY
THREAD 89a2e618 Cid 1d2c.1de4 Teb: 7ffd0000 Win32Thread: bc203908 READY
Processor 0: Ready Threads at priority 0
THREAD 8b184db0 Cid 0004.0008 Teb: 00000000 Win32Thread: 00000000 READY
Processor 1: Ready Threads at priority 8
THREAD 89d89db0 Cid 1b10.20ac Teb: 7ffd7000 Win32Thread: bc16e680 READY
THREAD 891f24a8 Cid 1e2c.20d0 Teb: 7ffda000 Win32Thread: bc1b9ea8 READY
THREAD 89214db0 Cid 1e2c.24d4 Teb: 7ffd7000 Win32Thread: bc24ed48 READY
THREAD 89a28020 Cid 1b10.21b4 Teb: 7ffa7000 Win32Thread: bc25b3b8 READY
THREAD 891e03b0 Cid 1a98.05c4 Teb: 7ffdb000 Win32Thread: bc228bb0 READY
THREAD 891b0020 Cid 1cd0.0144 Teb: 7ffde000 Win32Thread: bc205ea8 READY
Processor 1: Ready Threads at priority 7
THREAD 898367a0 Cid 0f4c.1cd4 Teb: 00000000 Win32Thread: 00000000 READY
THREAD 8a1ac020 Cid 0f4c.1450 Teb: 00000000 Win32Thread: 00000000 READY
THREAD 8aa1ab90 Cid 0f4c.11b0 Teb: 00000000 Win32Thread: 00000000 READY
THREAD 89cc92e0 Cid 0f4c.1b34 Teb: 00000000 Win32Thread: 00000000 READY
THREAD 89579020 Cid 0f4c.2220 Teb: 00000000 Win32Thread: 00000000 READY
Processor 1: Ready Threads at priority 6
THREAD 8a487db0 Cid 1584.14bc Teb: 7ffa2000 Win32Thread: bc304ea8 READY
THREAD 8a3ce020 Cid 1584.0630 Teb: 7ff8e000 Win32Thread: bc293c20 READY
THREAD 8a1b6db0 Cid 1584.1590 Teb: 7ff8c000 Win32Thread: bc310ea8 READY
THREAD 8a1fe6e0 Cid 1584.15ec Teb: 7ffa1000 Win32Thread: bc15bea8 READY
THREAD 8ac0adb0 Cid 1584.156c Teb: 7ff8a000 Win32Thread: bc153be8 READY
THREAD 8b1e35a0 Cid 1584.15f4 Teb: 7ff9e000 Win32Thread: bc0567e8 READY
THREAD 8a3288e8 Cid 1584.14b8 Teb: 7ff9a000 Win32Thread: bc2fbea8 READY
THREAD 8a5056a0 Cid 1584.1518 Teb: 7ff91000 Win32Thread: bc337ea8 READY
THREAD 891afdb0 Cid 1d2c.27e8 Teb: 7ffaf000 Win32Thread: bc217c18 READY
THREAD 8a07d308 Cid 1d2c.2548 Teb: 7ffae000 Win32Thread: bc235750 READY
THREAD 8a055d18 Cid 1584.17d0 Teb: 7ffd5000 Win32Thread: 00000000 READY
THREAD 8ac0b770 Cid 0250.0268 Teb: 7ffde000 Win32Thread: bc2349d8 READY
THREAD 8a0eeeb40 Cid 1584.1560 Teb: 7ffdc000 Win32Thread: 00000000 READY

```

Here we should also analyze **Stack Traces** (page 926) for running and ready threads with the priority 8 and check kernel and user times (**Spiking Thread**, page 888). If we find anything common between them, we should also check ready threads with lower priority to see if that commonality is unique to threads with the priority 8. See also the similar pattern **Busy System** (page 99), and the similar starvation pattern resulted from realtime priority threads (page 1008).

## Realtime Priority

This pattern happens when some threads or processes have higher priority and are favored by OS thread dispatcher effectively starving other threads. If prioritized threads are CPU-bound, we also see CPU spikes (**Spiking Thread** pattern, page 888). However, if their thread priorities were normal they would have been preempted by other threads, and latter threads would not be starved. Here is one example where two threads from two different applications but from one user session are spiking on two processors (threads running on other processors have above normal and normal priority):

```
System Uptime: 0 days 6:40:41.143

0: kd> !running

System Processors f (affinity mask)
Idle Processors None

    Prcb      Current      Next
0 f773a120  89864c86 .....
1 f773d120  89f243d2 .....
2 f7737120  89f61398 .....
3 f773f120  897228a0 .....
```

0: kd> .thread /r /p 89f61398  
Implicit thread is now 89f61398  
Implicit process is now 88bcc2e0  
Loading User Symbols

0: kd> !thread 89f61398 1f  
THREAD 89f61398 Cid 16f8.2058 Teb: 7fffd000 Win32Thread: bc41aea8 RUNNING on processor 2  
Not impersonating  
DeviceMap e48a6508  
Owning Process 88bcc2e0 Image: application.exe  
Wait Start TickCount 1569737 Ticks: 0  
Context Switch Count 7201654 LargeStack  
UserTime 01:24:06.687  
KernelTime 00:14:53.828  
Win32 Start Address application (0x0040a52c)  
Start Address kernel32!BaseProcessStartThunk (0x77e617f8)  
Stack Init ba336000 Current ba335d00 Base ba336000 Limit ba330000 Call 0  
Priority 24 BasePriority 24 PriorityDecrement 0  
ChildEBP RetAddr  
0012e09c 762c3b7d USER32!IsWindowVisible  
0012e0c4 762d61bb MSVBVM50!RbyCountVisibleDesks+0x3c  
0012e0d0 004831f6 MSVBVM50!rtcDoEvents+0x7  
0012e348 0046d1ae application+0x831f6  
0012e3a0 762ce5a9 application+0x6d1ae  
0012e3dc 762ce583 MSVBVM50!CallProcWithArgs+0x20  
0012e3f8 762db781 MSVBVM50!InvokeVtblEvent+0x33  
0012e434 762cfbc2 MSVBVM50!InvokeEvent+0x32  
0012e514 762cfa4a MSVBVM50!EvtErrFireWorker+0x175  
0012e55c 762b1aa3 MSVBVM50!EvtErrFire+0x18  
0012e5ac 7739bffa MSVBVM50!CThreadPool::GetThreadData+0xf  
0012e58c 762cd13b USER32!CallHookWithSEH+0x21  
0012e5ac 7739bffa MSVBVM50!VBDefControlProc\_2787+0xad

```

0012e618 762d3348 USER32!CallHookWithSEH+0x21
0012e640 762cda44 MSVBVM50!PushCtlProc+0x2e
0012e668 762cd564 MSVBVM50!CommonGizWndProc+0x4e
0012e6b8 7739b6e3 MSVBVM50!StdCtlWndProc+0x171
0012e6e4 7739b874 USER32!InternalCallWinProc+0x28
0012e75c 7739ba92 USER32!UserCallWinProcCheckWow+0x151
0012e7c4 773a16e5 USER32!DispatchMessageWorker+0x327
0012e7d4 762d616e USER32!DispatchMessageA+0xf
0012e828 762d6054 MSVBVM50!ThunderMsgLoop+0x97
0012e874 762d5f55 MSVBVM50!SCM::FPushMessageLoop+0xaf
0012e8b4 004831f6 MSVBVM50!CMsoComponent::PushMsgLoop+0x24
0012e8c0 00d3b3c8 application+0x831f6
00184110 00000000 0xd3b3c8

0: kd> .thread /r /p 897228a0
Implicit thread is now 897228a0
Implicit process is now 897348a8
Loading User Symbols

0: kd> !thread 897228a0 1f 100
THREAD 897228a0 Cid 2984.2988 Teb: 7fffd000 Win32Thread: bc381488 RUNNING on processor 3
IRP List:
 89794bb8: (0006,0220) Flags: 00000000 Mdl: 8a145878
Not impersonating
DeviceMap           e3ec0360
Owning Process     897348a8      Image:       application2.exe
Wait Start TickCount 1569737      Ticks: 0
Context Switch Count 10239625      LargeStack
UserTime            02:38:18.890
KernelTime          00:29:36.187
Win32 Start Address application2 (0x00442e4c)
Start Address kernel32!BaseProcessStartThunk (0x77e617f8)
Stack Init f1d90000 Current f1d8fd00 Base f1d90000 Limit f1d88000 Call 0
Priority 24 BasePriority 24 PriorityDecrement 0
ChildEBP RetAddr
0012f66c 762d61bb USER32!_SEH_prolog+0x5
0012f678 00fdb0b9 MSVBVM50!rtcDoEvents+0x7
0012f92c 00fcfa760 application2+0xbdb0b9
0012fa20 762ce5a9 application2+0xbcfa760
0012fa40 762ce583 MSVBVM50!CallProcWithArgs+0x20
0012fa5c 762db781 MSVBVM50!InvokeVtblEvent+0x33
0012fa98 762cfbc2 MSVBVM50!InvokeEvent+0x32
0012fb78 762cf4a MSVBVM50!EvtErrFireWorker+0x175
0012fb90 76330b2b MSVBVM50!EvtErrFire+0x18
0012fbf0 762cd13b MSVBVM50!ErrDefMouse_100+0x16d
0012fcfa4 762cda44 MSVBVM50!VBDefControlProc_2787+0xad
0012fcc4 7631c826 MSVBVM50!CommonGizWndProc+0x4e
0012fd08 762cd523 MSVBVM50!StdCtlPreFilter_50+0x9e
0012fd5c 7739b6e3 MSVBVM50!StdCtlWndProc+0x130
0012fd88 7739b874 USER32!InternalCallWinProc+0x28
0012fe00 7739ba92 USER32!UserCallWinProcCheckWow+0x151
0012fe68 773a16e5 USER32!DispatchMessageWorker+0x327
0012fe78 762d616e USER32!DispatchMessageA+0xf
0012fea8 762bb78f MSVBVM50!ThunderMsgLoop+0x97
0012feb8 762d60cb MSVBVM50!MsоФInitPx+0x39
0012fec4 762d6054 MSVBVM50!CMsoCMHandler::FPushMessageLoop+0x1a
0012ff18 762d5f55 MSVBVM50!SCM::FPushMessageLoop+0xaf

```

```
0012ffa0 8082ea41 MSVBVM50!CMsoComponent::PushMsgLoop+0x24
0012fef8 762d5f8e nt!KiDeliverApc+0x11f
0012ff18 762d5f55 MSVBVM50!SCM_MsoCompMgr::FPushMessageLoop+0x2f
0012ffa0 8082ea41 MSVBVM50!CMsoComponent::PushMsgLoop+0x24
0012ff18 762d5f55 nt!KiDeliverApc+0x11f
0012ffa0 8082ea41 MSVBVM50!CMsoComponent::PushMsgLoop+0x24
0012ffd4 0012ffc8 nt!KiDeliverApc+0x11f
00000000 00000000 0x12ffc8
```

What we see here is unusually high Priority and BasePriority values (24 and 24). This means that the base priority of these processes was most likely artificially increased to Realtime. Most processes have base priority 8 (Normal):

```
0: kd> !thread 88780db0 1f
THREAD 88780db0 Cid 44a8.1b8c Teb: 7ffaf000 Win32Thread: bc315d20 WAIT: (Unknown) UserMode Non-
Alertable
    887b8650 Semaphore Limit 0xffffffff
    88780e28 NotificationTimer
Not impersonating
DeviceMap           e1085298
Owning Process     889263a0      Image:       explorer.exe
Wait Start TickCount 1565543      Ticks: 4194 (0:00:01:05.531)
Context Switch Count 7            LargeStack
UserTime            00:00:00.000
KernelTime          00:00:00.000
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init b6754000 Current b6753c0c Base b6754000 Limit b6750000 Call 0
Priority 9 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr
b6753c24 80833e8d nt!KiSwapContext+0x26
b6753c50 80829b74 nt!KiSwapThread+0x2e5
b6753c98 809249cd nt!KeWaitForSingleObject+0x346
b6753d48 8088ac4c nt!NtReplyWaitReceivePortEx+0x521
b6753d48 7c8285ec nt!KiFastCallEntry+0xfc
01a2fe18 7c82783b ntdll!KiFastSystemCall1Ret
01a2fe1c 77c885ac ntdll!NtReplyWaitReceivePortEx+0xc
01a2ff84 77c88792 RPCRT4!LRPC_ADDRESS::ReceiveLotsaCalls+0x198
01a2ff8c 77c8872d RPCRT4!RecvLotsaCallsWrapper+0xd
01a2ffac 77c7b110 RPCRT4!BaseCachedThreadRoutine+0x9d
01a2ffb8 77e64829 RPCRT4!ThreadStartRoutine+0x1b
01a2ffec 00000000 kernel32!BaseThreadStart+0x34
```

Some important system processes like *csrss.exe* have base priority 13 (High), but their threads wait most of the time, and this doesn't create any problems:

```
0: kd> !thread 887eb3d0 1f
THREAD 887eb3d0  Cid 4cf4.2bd4  Teb: 7fffaf000 Win32Thread: bc141cc0 WAIT: (Unknown) UserMode Non-
Alertable
 888769b0  SynchronizationEvent
Not impersonating
DeviceMap          e1000930
Owning Process    8883f7c0      Image:        csrss.exe
Wait Start TickCount 1540456      Ticks: 29281 (0:00:07:37.515)
Context Switch Count 40           LargeStack
UserTime           00:00:00.000
KernelTime         00:00:00.000
Start Address winsrv!ConsoleInputThread (0x75a81b18)
Stack Init b5c5a000 Current b5c59bac Base b5c5a000 Limit b5c55000 Call 0
Priority 15 BasePriority 13 PriorityDecrement 0
Kernel stack not resident.
ChildEBP RetAddr
b5c59bc4 80833e8d nt!KiSwapContext+0x26
b5c59bf0 80829b74 nt!KiSwapThread+0x2e5
b5c59c38 bf89b1c3 nt!KeWaitForSingleObject+0x346
b5c59c94 bf89b986 win32k!xxxSleepThread+0x1be
b5c59cec bf89da22 win32k!xxxRealInternalGetMessage+0x46a
b5c59d4c 8088ac4c win32k!NtUserGetMessage+0x3f
b5c59d4c 7c8285ec nt!KiFastCallEntry+0xfc
00ffff64 7739c811 ntdll!KiFastSystemCallRet
00ffff84 75a81c47 USER32!NtUserGetMessage+0xc
00fffff4 00000000 winsrv!ConsoleInputThread+0x16c
```

It is very unusual for a process to have Realtime base priority. We can speculate what had really happened before the system crash was forced. The system administrator noticed two applications consuming CPU over the long period of time and decided to intervene. Unfortunately, the hand slipped when browsing Task Manager Set Priority menu, and Realtime was selected instead of Low. This may have been a *human error*.

## Top Module

This pattern is similar to **Blocking Module** pattern (page 96) with the difference in stack trace syntax only. A top module is any module we choose that is simply on top of a stack trace. Most of the time, it is likely to be a non-OS vendor module. Whether the stack trace is well-formed and semantically sound or **Incorrect Stack Trace** (page 499) is irrelevant:

```
0:005> kL  
ChildEBP RetAddr  
01abc4d8 6efba23d ntdll!KiFastSystemCallRet  
WARNING: Stack unwind information not available. Following frames may be wrong.  
01abc988 7c820833 ModuleB+0x2a23d  
01abcbe4 7c8207f6 kernel32!GetVolumeNameForRoot+0x26  
01abcc0c 7c82e6de kernel32!BasepGetVolumeNameForVolumeMountPoint+0x75  
01abcc54 6efaf70b kernel32!GetVolumePathNameW+0x18a  
01abccdc 6efbd1a6 ModuleB+0x1f70b  
01abcce0 00000000 ModuleB+0x2d1a6
```

Here we can also check the validity of *ModuleB* code by backward disassembly of 6efba23d return address (**ub** command). If we have **Abridged Dump** (page 49) file (minidump), we need to specify the image file path in WinDbg.

Why is **Top Module** important? In various troubleshooting scenarios, we can check the module timestamp (**Not My Version** pattern, page 744) and other useful information (**!lmv** and **!lmi** WinDbg commands). If we suspect the module as **Hooksware** (page 1175), we can also recommend removing it or its software vendor package for testing purposes.

## Translated Exception

Runtime **Software Exceptions** (page 875, such as C++ exceptions, page 108) can be translated by **Custom Exception Handlers** (page 171) into other exceptions by changing exception data. This is different from **Nested Exceptions** (page 726) where another exception is thrown. One example of such possible translation we recently encountered when looked at raw stack data (**!teb -> dps**) having signs of **Hidden Exceptions** (multiple *RaiseException* calls, page 457) and also CLR **Execution Residue** (valid return addresses of *clr* module, page 371). In addition to final **Invalid Handle** exception (page 574) and one hidden access violation, there were many exception codes c0000027. Google search pointed to the article about skipped C++ destructors<sup>211</sup> that prompted us to introduce this pattern.

---

<sup>211</sup> <http://www.codeproject.com/Articles/46294/When-a-C-destructor-did-not-run-Part-2>

## Truncated Dump

### Mac OS X

This is a Mac OS X / GDB counterpart to **Truncated Dump** pattern:

```
(gdb) info threads
Cannot access memory at address 0x7fff885e9e42
4 0x00007fff885e9e42 in ?? ()  
3 0x00007fff885e9e42 in ?? ()  
2 0x00007fff885e9e42 in ?? ()  
* 1 0x00007fff885e9e42 in ?? ()  
warning: Couldn't restore frame in current thread, at frame 0  
0x00007fff885e9e42 in ?? ()  
  
(gdb) disass 0x00007fff885e9e42
No function contains specified address.  
  
(gdb) info r rsp
rsp 0x7fff67fe8a18 0x7fff67fe8a18  
  
(gdb) x/100a 0x7fff67fe8a18
0x7fff67fe8a18: Cannot access memory at address 0x7fff67fe8a18
```

This often happens if there is no space to save a full core dump.

## Windows

Sometimes the page file size is less than the amount of physical memory. If this is the case and we have configured “Complete memory dump” in Startup and Recovery settings in Control Panel, we get **Truncated Memory Dumps**. WinDbg prints a warning when we open such a dump file:

```
*****
WARNING: Dump file has been truncated. Data may be missing.
*****
```

We can double check this with **!vm** command:

```
kd> !vm

*** Virtual Memory Usage ***
Physical Memory: 511859 ( 2047436 Kb)
Paging File Name paged out
  Current: 1536000 Kb  Free Space: 1522732 Kb
  Minimum: 1536000 Kb  Maximum: 1536000 Kb
```

We see that the page file size is 1.5Gb, but the amount of physical memory is 2Gb. When BSOD happens, the physical memory contents will be saved to the page file, and the dump file size will be no more than 1.5Gb effectively truncating the data needed for crash dump analysis.

Sometimes we can still access some data in truncated dumps, but we need to pay attention to what WinDbg says. For example, in **Truncated Dump** shown above the stack and driver code are not available:

```
kd> kv
ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be wrong.
f408b004 00000000 00000000 00000000 00000000 driver+0x19237

kd> r
Last set context:
eax=89d55230 ebx=89d21130 ecx=89d21130 edx=89c8cc20 esi=89e24ac0 edi=89c8cc20
eip=f7242237 esp=f408afec ebp=f408b004 iopl=0 nv up ei ng nz ac po nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010292
driver+0x19237:
f7242237 ??          ???

kd> dds esp
f408afec ????????
f408aff0 ????????
f408aff4 ????????
f408aff8 ????????
f408affc ????????
f408b000 ????????
f408b004 ????????
f408b008 ????????
f408b00c ????????
```

```
f408b010 ????????
f408b014 ????????
f408b018 ????????
f408b01c ????????
f408b020 ????????
f408b024 ????????
f408b028 ????????
f408b02c ????????
f408b030 ????????
f408b034 ????????
f408b038 ????????
f408b03c ????????
f408b040 ????????
f408b044 ????????
f408b048 ????????
f408b04c ????????
f408b050 ????????
f408b054 ????????
f408b058 ????????
f408b05c ????????
f408b060 ????????
f408b064 ????????
f408b068 ????????

kd> lmv m driver
start end module name
f7229000 f725f000 driver T (no symbols)
  Loaded symbol image file: driver.sys
  Image path: driver.sys
  Image name: driver.sys
  Timestamp: unavailable (FFFFFFFE)
  CheckSum: missing
  ImageSize: 00036000

kd> dd f7229000
f7229000 ????????
f7229010 ????????
f7229020 ????????
f7229030 ????????
f7229040 ????????
f7229050 ????????
f7229060 ????????
f7229070 ????????
```

If due to some reason we cannot increase the size of our page file we should configure “Kernel memory dump” in Startup and Recovery. For most all bugchecks, kernel memory dump is sufficient except for **Manual Dumps** (page 625) when we need to inspect user process space.

## Truncated Stack Trace

Sometimes we see this pattern with missing stack frames. For example, in one incident, after enabling user mode stack trace database for **Memory Leaking** (page 650) application we got these entries from the growing heap segment (other segments had non-truncated saved stack traces):

```
0bdc1350: 40010 . 40010 [101] - busy (3fff8) Internal

7702fb02: ntdll!RtlAllocateHeap+0x0000021d
77005eef: ntdll!RtlpAllocateUserBlock+0x000000a2
77026a65: ntdll!RtlpLowFragHeapAllocFromContext+0x00000785
7702661f: ntdll!RtlAllocateHeap+0x0000017c

0be01360: 40010 . 40010 [101] - busy (3fff8) Internal

7702fb02: ntdll!RtlAllocateHeap+0x0000021d
77005eef: ntdll!RtlpAllocateUserBlock+0x000000a2
77026a65: ntdll!RtlpLowFragHeapAllocFromContext+0x00000785
7702661f: ntdll!RtlAllocateHeap+0x0000017c

0be41370: 40010 . 40010 [101] - busy (3fff8) Internal

7702fb02: ntdll!RtlAllocateHeap+0x0000021d
77005eef: ntdll!RtlpAllocateUserBlock+0x000000a2
77026a65: ntdll!RtlpLowFragHeapAllocFromContext+0x00000785
7702661f: ntdll!RtlAllocateHeap+0x0000017c
```

**Truncated Stack Traces** are different from **Incorrect Stack Traces** (page 499) because their surviving part is correct. How can we find the rest of such stack traces? Here we can suggest looking at other heap segments and see allocations of the same size. If **Truncated Stack Trace** comes from **Stack Trace Collection** (page 943), we can compare it with a non-truncated thread stack from another process instance having the same thread position.

## Comments

Another example is when a debugger truncates the output. In WinDbg case, there is a command **.kframes** that sets the default number of stack trace frames.

Here's another example of **Truncated Stack Trace** from x64 Windows 10 after **NULL Code Pointer** (page 750) was called.

```
0:003> kL
# Child-SP RetAddr Call Site
00 0000005b`056ae1f8 00007ffa`e2f2916f ntdll!NtWaitForMultipleObjects+0xa
01 0000005b`056ae200 00007ffa`e2f2906e KERNELBASE!WaitForMultipleObjectsEx+0xef
02 0000005b`056ae500 00007ffa`e3b5155c KERNELBASE!WaitForMultipleObjects+0xe
03 0000005b`056ae540 00007ffa`e3b51088 kernel32!WerReportFaultInternal+0x494
04 0000005b`056aeab0 00007ffa`e2f503ad kernel32!WerReportFault+0x48
05 0000005b`056aeae0 00007ffa`e5c58cd2 KERNELBASE!UnhandledExceptionFilter+0x1fd
06 0000005b`056aebe0 00007ffa`e5c44296 ntdll!RtlUserThreadStart$filt$0+0x3e
07 0000005b`056aec20 00007ffa`e5c5666d ntdll!_C_specific_handler+0x96
08 0000005b`056aec90 00007ffa`e5bd3c00 ntdll!RtlpExecuteHandlerForException+0xd
09 0000005b`056aec00 00007ffa`e5c5577a ntdll!RtlDispatchException+0x370
0a 0000005b`056af3c0 00000000`00000000 ntdll!KiUserExceptionDispatch+0x3a
```

We can look at the trap frame RSP address and recover the return address:

```
0:003> dt _KTRAP_FRAME RSP 0000005b`056af7e8
ntdll!_KTRAP_FRAME
+0x180 Rsp : 0x0000005b`056afad8

0:003> dps 0x0000005b`056afad8 L1
0000005b`056afad8 00007ff7`f6381266 ApplicationM!thread_two+0x16

0:003> ub 00007ff7`f6381266
[...]
ApplicationM!thread_two:
00007ff7`f6381250 48894c2408      mov qword ptr [rsp+8],rcx
00007ff7`f6381255 4883ec38      sub rsp,38h
00007ff7`f6381259 48c74424200000000  mov qword ptr [rsp+20h],0
00007ff7`f6381262 ff542420      call qword ptr [rsp+20h]
```

and reconstruct the bottom of the stack trace:

```
0:003> k L=0x0000005b`056afad8
# Child-SP RetAddr Call Site
00 0000005b`056afad8 00007ff7`f6381266 ntdll!NtWaitForMultipleObjects+0xa
01 0000005b`056afae0 00007ff7`f6382bd1 ApplicationM!thread_two+0x16
02 (Inline Function) --- ApplicationM!invoke_thread_procedure+0xe
03 0000005b`056afb20 00007ffa`e3b42d92 ApplicationM!thread_start+0x5d
04 0000005b`056afb50 00007ffa`e5bc9f64 kernel32!BaseThreadInitThunk+0x22
05 0000005b`056afb80 00000000`00000000 ntdll!RtlUserThreadStart+0x34
```

and then construct **Glued Stack Trace** (page 413):

```
0000005b`056ae1f8 00007ffa`e2f2916f ntdll!NtWaitForMultipleObjects+0xa
0000005b`056ae200 00007ffa`e2f2906e KERNELBASE!WaitForMultipleObjectsEx+0xef
0000005b`056ae500 00007ffa`e3b5155c KERNELBASE!WaitForMultipleObjects+0xe
0000005b`056ae540 00007ffa`e3b51088 kernel32!WerReportFaultInternal+0x494
0000005b`056aeab0 00007ffa`e2f503ad kernel32!WerReportFault+0x48
0000005b`056aeae0 00007ffa`e5c58cd2 KERNELBASE!UnhandledExceptionFilter+0x1fd
0000005b`056aebe0 00007ffa`e5c44296 ntdll!RtlUserThreadStart$filt$0+0x3e
0000005b`056aec20 00007ffa`e5c5666d ntdll!_C_specific_handler+0x96
0000005b`056aec90 00007ffa`e5bd3c00 ntdll!RtlpExecuteHandlerForException+0xd
0000005b`056aec00 00007ffa`e5c5577a ntdll!RtlDispatchException+0x370
```

```
0000005b`056af3c0 00000000`00000000 ntdll!KiUserExceptionDispatch+0x3a
0000005b`056afae0 00007ff7`f6382bd1 ApplicationM!thread_two+0x16
(Inline Function) --- ApplicationM!invoke_thread_procedure+0xe
0000005b`056afb20 00007ffa`e3b42d92 ApplicationM!thread_start+0x5d
0000005b`056afb50 00007ffa`e5bc9f64 kernel32!BaseThreadInitThunk+0x22
0000005b`056afb80 00000000`00000000 ntdll!RtlUserThreadStart+0x34
```

Some truncated stack traces originate from **JIT Code** (page 591):

```
00007ffa86d9952c: ntdll!RtlAllocateHeap+0x000000000000788fc
0000000058248d17: msvcr100!malloc+0x000000000000005b
0000000058248ddb: msvcr100!operator new+0x0000000000000001f
00007ffa1f1a77f8: +0x00007ffa1f1a77f8

0:000> ub 0x00007ffa1f1a77f8
00007ffa`1f1a77d5 488b4320      mov rax,qword ptr [rbx+20h]
00007ffa`1f1a77d9 48895d18      mov qword ptr [rbp+18h],rbx
00007ffa`1f1a77dd 488d1514000000 lea rdx,[00007ffa`1f1a77f8]
00007ffa`1f1a77e4 48895530      mov qword ptr [rbp+30h],rdx
00007ffa`1f1a77e8 c7410c00000000 mov dword ptr [rcx+0Ch],0
00007ffa`1f1a77ef 488b4d40      mov rcx,qword ptr [rbp+40h]
00007ffa`1f1a77f3 4533db       xor r11d,r11d
00007ffa`1f1a77f6 ff10         call qword ptr [rax]

0:000> !IP2MD 0x00007ffa1f1a77f8
MethodDesc:    00007ffa1f292678
Method Name:   ModuleA.Foo(UInt64)
Class:          00007ffa1f204b90
MethodTable:   00007ffa1f204c08
mdToken:        000000006000000
Module:         00007ffa1f203760
IsJitted:      yes
CodeAddr:       00007ffa1f1a7790
Transparency:  Safe critical
*** ERROR: Module load completed but symbols could not be loaded for Company.Subsystem.Component.dll
```

Sometimes WinDbg commands show only limited number of frames. For example, **!cs -l -o -s** command may not show heap corruption as the cause of heap-loader deadlock if unhandled exception processing is too long. We need to double check stack traces with **k** commands. See also Common Mistakes<sup>212</sup>.

---

<sup>212</sup> Memory Dump Analysis Anthology, Volume 2, page 39

## U

## Ubiquitous Component

## Kernel Space

This is a kernel space counterpart of **Ubiquitous Component** pattern. Such a component (especially when it is **Top Module**, page 1012) can be a sign of **Wait Chain(s)** (page 1092) and **Blocking Module** (page 96), and if it is present in the same process names - a sign of **Distributed Wait Chain** (page 253).

```
0: kd> !stacks 0 ModuleA
Proc.Thread .Thread Ticks ThreadState Blocker

[fffffa800e673b30 svchost.exe]
534.006240 fffffa801388f5f0 fffd41d9 Blocked ModuleA+0x12468

[fffffa800e705b30 svchost.exe]
630.000e14 fffffa800edacb50 ffdcf7a Blocked ModuleA+0x12468
630.000f04 fffffa8012c2fb50 ffdcf49 Blocked ModuleA+0x12468
630.006610 fffffa80134f5b50 ffdcf46 Blocked ModuleA+0x12468
630.001fcfc fffffa800f55a2d0 ffdcf44 Blocked ModuleA+0x12468
630.003db8 fffffa80121f1540 ffdcf43 Blocked ModuleA+0x12468
630.000b9c fffffa80133d1780 ffdcf3c Blocked ModuleA+0x12468
630.0041c4 fffffa8013c77b50 ffdcf43 Blocked ModuleA+0x12468
630.00641c fffffa8012476b50 ffdcf43 Blocked ModuleA+0x12468
630.006424 fffffa8013207b50 ffdcf40 Blocked ModuleA+0x12468
630.002fcc fffffa80128f9060 ffdcf3e Blocked ModuleA+0x12468
630.003de8 fffffa80139edb50 ffdcf3d Blocked ModuleA+0x12468
630.0062c4 fffffa800f5ff2d0 ffdcf3c Blocked ModuleA+0x12468
630.0065e8 fffffa80139dc50 ffdcf3b Blocked ModuleA+0x12468
630.004524 fffffa8011e51b50 ffdcf3a Blocked ModuleA+0x12468
630.004570 fffffa801346b060 ffdcf39 Blocked ModuleA+0x12468
630.00173c fffffa8010b99b50 ffdcf39 Blocked ModuleA+0x12468

[fffffa800f63db30 iexplore.exe]
24c4.0024c8 fffffa800fe854e0 fffcb6cf Blocked ModuleA+0x12468

[fffffa8010b9ab30 explorer.exe]
2b64.0043d0 fffffa8012e8ab00 ffd9095 Blocked ModuleA+0x12468

[fffffa800fe55060 explorer.exe]
2c80.002e58 fffffa8012e75060 fffba7af Blocked ModuleA+0x12468

[fffffa8010c54b30 iexplore.exe]
2e3c.002e98 fffffa8010c75620 fffcbb7f Blocked ModuleA+0x12468

[fffffa80111c3720 iexplore.exe]
32d8.003230 fffffa80111b1b00 ffd41d9 Blocked ModuleA+0x12468
```

[fffffa80110cb690 iexplore.exe]  
2e74.002854 ffffffa8011121b00 fffbe8a4 Blocked ModuleA+0x12468

[fffffa801146cb30 OUTLOOK.EXE]  
35cc.0035e8 ffffffa8013831b00 ffffaf33a Blocked ModuleA+0x12468

[fffffa80105a5640 OUTLOOK.EXE]  
3858.00385c ffffffa801133ab00 fffd3691 Blocked ModuleA+0x12468

[fffffa8011998060 explorer.exe]  
3d70.004a0c ffffffa80139ddb00 fffd0482 Blocked ModuleA+0x12468

[fffffa8010ff5850 OUTLOOK.EXE]  
3540.000458 ffffffa8011052b00 fffbd007 Blocked ModuleA+0x12468

[fffffa8011d3d060 OUTLOOK.EXE]  
49f8.0049fc ffffffa8011c78060 fffdbbf9 Blocked ModuleA+0x12468

[fffffa801241b060 OUTLOOK.EXE]  
4888.005af0 ffffffa8012e8eab0 fffae442 Blocked ModuleA+0x12468  
4888.003d24 ffffffa800eca7b00 fffae443 Blocked ModuleA+0x12468

[fffffa8012687b30 explorer.exe]  
5048.0051fc ffffffa801129cb00 fffca8bf Blocked ModuleA+0x12468

[fffffa8011c1e060 OUTLOOK.EXE]  
52c4.00117c ffffffa80130f8710 fffaa157 Blocked ModuleA+0x12468  
52c4.0045fc ffffffa801374f060 fffaa15e Blocked ModuleA+0x12468

[fffffa8011c42b30 explorer.exe]  
5898.0001ec ffffffa80137a1b00 fffd8da0 Blocked ModuleA+0x12468

[fffffa8012e04b30 OUTLOOK.EXE]  
5a74.004954 ffffffa8012e05060 fffa9ff8 Blocked ModuleA+0x12468

[fffffa8010908b30 spoolsv.exe]  
2724.004190 ffffffa8011ea1060 ffdcaf8 Blocked ModuleA+0x12468

[fffffa801206eb30 WerFault.exe]  
3e50.005424 ffffffa8013c5eb00 ffdcf39 Blocked ModuleA+0x12468

[fffffa800f8cf2a0 WerFault.exe]  
9f4.00570c ffffffa8013c8ab00 ffdca9f Blocked ModuleA+0x12468

[fffffa8013af1060 WerFault.exe]  
3c74.002b80 ffffffa8013c5c060 ffd9dc8 Blocked ModuleA+0x12468

[fffffa800f8053a0 WINWORD.EXE]  
3dd0.0066a8 ffffffa800ce618c0 ffd7c02 Blocked ModuleA+0x12468

[fffffa8010b66b30 WINWORD.EXE]  
62a4.001934 ffffffa801368c430 ffd7ce7 Blocked ModuleA+0x12468

```
[fffffa80141dc060 WerFault.exe]
17d0.0052e4  fffffa801347a060 fffd57b8 Blocked    ModuleA+0x12468

[fffffa8012629760 WerFault.exe]
621c.005b64  fffffa8011e395d0 fffc8dc2 Blocked    ModuleA+0x12468

[fffffa80131a75d0 explorer.exe]
4884.002b34  fffffa8013dc3b00 fffd67bc Blocked    ModuleA+0x12468

[...]

Threads Processed: 5948
```

## User Space

Sometimes we look at **Stack Trace Collection** (page 943), and we see dozens of threads running through some 3rd-party component. We do not normally expect this component to appear, or expect that only one or two threads are running through it. Here is an example from IE after installing a 3rd-party toolbar that becomes a ubiquitous component:

```
0:087> *kc

# 0 Id: 136c.fa0 Suspend: 0 Teb: 7ffd000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
user32!RealMsgWaitForMultipleObjectsEx
ieui!CoreSC::Wait
ieui!CoreSC::WaitMessage
ieui!WaitMessageEx
ieframe!CBrowserFrame::FrameMessagePump
ieframe!BrowserThreadProc
ieframe!BrowserNewThreadProc
ieframe!SHOpenFolderWindow
ieframe!IEWinMain
iexplore!wWinMain
iexplore!_initterm_e
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

1 Id: 136c.12c0 Suspend: 0 Teb: 7ffdc000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
ntdll!TppWaiterpThread
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

2 Id: 136c.1120 Suspend: 0 Teb: 7ffda000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
user32!RealMsgWaitForMultipleObjectsEx
ieui!CoreSC::Wait
ieui!CoreSC::xwProcessNL
ieui!GetMessageExA
ieui!ResourceManager::SharedThreadProc
msvcrt!_endthreadex
msvcrt!_endthreadex
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

```
3 Id: 136c.1588 Suspend: 0 Teb: 7ffd9000 Unfrozen

ntdll!KiFastSystemCallRet
user32!NtUserWaitMessage
ieframe!CTabWindow::_TabWindowThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

4 Id: 136c.15a4 Suspend: 0 Teb: 7ffd8000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!DllGetClassObject
ToolbarA!ToolBarMain
ToolbarA!DllCanUnloadNow
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

5 Id: 136c.111c Suspend: 0 Teb: 7ffd6000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwWaitForWorkViaWorkerFactory
ntdll!TppWorkerThread
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

6 Id: 136c.10a8 Suspend: 0 Teb: 7ffd5000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwWaitForSingleObject
kernel32!WaitForSingleObjectEx
kernel32!WaitForSingleObject
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!DllGetClassObject
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

7 Id: 136c.fb0 Suspend: 0 Teb: 7ffd3000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
```

```

WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!DllGetClassObject
ToolbarA!ToolBarMain
ToolbarA!DllCanUnloadNow
ntdll!_RtlUserThreadStart
ntdll!_RtlUserThreadStart

8  Id: 136c.1728 Suspend: 0 Teb: 7ffae000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!DllCanUnloadNow
ToolbarA!DllCanUnloadNow
ToolbarA!DllCanUnloadNow
ntdll!_RtlUserThreadStart
ntdll!_RtlUserThreadStart

9  Id: 136c.918 Suspend: 0 Teb: 7ffad000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!DllCanUnloadNow
ToolbarA!DllCanUnloadNow
ToolbarA!DllCanUnloadNow
ntdll!_RtlUserThreadStart
ntdll!_RtlUserThreadStart

10 Id: 136c.11c4 Suspend: 0 Teb: 7ffd4000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwWaitForSingleObject
mswsock!SockWaitForSingleObject
mswsock!WSPSelect
ws2_32!select
wininet!ICASyncThread::SelectThread
wininet!ICASyncThread::SelectThreadWrapper
kernel32!BaseThreadInitThunk
ntdll!_RtlUserThreadStart
ntdll!_RtlUserThreadStart

11 Id: 136c.13bc Suspend: 0 Teb: 7ffa8000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwWaitForSingleObject
kernel32!WaitForSingleObjectEx
kernel32!WaitForSingleObject
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!DllGetClassObject
ToolbarA!ToolBarMain
ToolbarA!DllCanUnloadNow

```

```

ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

12 Id: 136c.178 Suspend: 0 Teb: 7ffab000 Unfrozen

ntdll!KiFastSystemCallRet
user32!NtUserGetMessage
user32!GetMessageA
winmm!mciwindow
kernel32!BaseThreadInitThunk
ntdll!_RtlUserThreadStart
ntdll!_RtlUserThreadStart

13 Id: 136c.1594 Suspend: 0 Teb: 7ffa9000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
wdmaud!CWorker::_ThreadProc
wdmaud!CWorker::_StaticThreadProc
kernel32!BaseThreadInitThunk
ntdll!_RtlUserThreadStart
ntdll!_RtlUserThreadStart

14 Id: 136c.b50 Suspend: 0 Teb: 7ffa0000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!_RtlUserThreadStart
ntdll!_RtlUserThreadStart

15 Id: 136c.eec Suspend: 0 Teb: 7ff9e000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!_RtlUserThreadStart
ntdll!_RtlUserThreadStart

16 Id: 136c.664 Suspend: 0 Teb: 7ff9d000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!_RtlUserThreadStart
ntdll!_RtlUserThreadStart

17 Id: 136c.2cc Suspend: 0 Teb: 7ff9b000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus

```

```

dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

18 Id: 136c.e00 Suspend: 0 Teb: 7ff9c000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwWaitForSingleObject
kernel32!WaitForSingleObjectEx
kernel32!WaitForSingleObject
mshtml!CTimerMan::ThreadExec
mshtml!CExecFT::ThreadProc
mshtml!CExecFT::StaticThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

19 Id: 136c.620 Suspend: 0 Teb: 7ff95000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

20 Id: 136c.1158 Suspend: 0 Teb: 7ff91000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

21 Id: 136c.10cc Suspend: 0 Teb: 7ff90000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

22 Id: 136c.1264 Suspend: 0 Teb: 7ff8f000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

```

```

23 Id: 136c.13fc Suspend: 0 Teb: 7ffd0000 Unfrozen

ntdll!KiFastSystemCallRet
user32!NtUserWaitMessage
ieframe!CTabWindow::_TabWindowThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

24 Id: 136c.914 Suspend: 0 Teb: 7ffdb000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!D11GetClassObject
ToolbarA!ToolBarMain
ToolbarA!D11CanUnloadNow
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

25 Id: 136c.1194 Suspend: 0 Teb: 7ffac000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!D11CanUnloadNow
ToolbarA!D11CanUnloadNow
ToolbarA!D11CanUnloadNow
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

26 Id: 136c.1548 Suspend: 0 Teb: 7ffa6000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!D11CanUnloadNow
ToolbarA!D11CanUnloadNow
ToolbarA!D11CanUnloadNow
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

27 Id: 136c.a00 Suspend: 0 Teb: 7ff99000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

```

28 Id: 136c.1360 Suspend: 0 Teb: 7ff97000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

29 Id: 136c.dc0 Suspend: 0 Teb: 7ff96000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

30 Id: 136c.a80 Suspend: 0 Teb: 7ff94000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

31 Id: 136c.1390 Suspend: 0 Teb: 7ffa1000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwWaitForSingleObject
kernel32!WaitForSingleObjectEx
kernel32!WaitForSingleObject
WARNING: Stack unwind information not available.
Following frames may be wrong.
Flash10b!DLLUnregisterServer
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

32 Id: 136c.123c Suspend: 0 Teb: 7ff9f000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwWaitForSingleObject
kernel32!WaitForSingleObjectEx
kernel32!WaitForSingleObject
WARNING: Stack unwind information not available.
Following frames may be wrong.
Flash10b!DLLUnregisterServer
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

33 Id: 136c.1398 Suspend: 0 Teb: 7ff82000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
winmm!timeThread
```

```
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

34 Id: 136c.ca0 Suspend: 0 Teb: 7ff7d000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
user32!RealMsgWaitForMultipleObjectsEx
ieui!CoreSC::Wait
ieui!CoreSC::WaitMessage
ieui!WaitMessageEx
ieframe!CBrowserFrame::FrameMessagePump
ieframe!BrowserThreadProc
ieframe!BrowserNewThreadProc
ieframe!SHOpenFolderWindow
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

35 Id: 136c.135c Suspend: 0 Teb: 7ff7c000 Unfrozen

```
ntdll!KiFastSystemCallRet
user32!NtUserWaitMessage
ieframe!CTabWindow::_TabWindowThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

36 Id: 136c.c34 Suspend: 0 Teb: 7ff7b000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!D1lGetObject
ToolbarA!ToolBarMain
ToolbarA!D1lCanUnloadNow
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

37 Id: 136c.a08 Suspend: 0 Teb: 7ff7a000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!D1lCanUnloadNow
ToolbarA!D1lCanUnloadNow
ToolbarA!D1lCanUnloadNow
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

<pre>38  Id: 136c.1108 Suspend: 0 Teb: 7ff79000 Unfrozen  ntdll!KiFastSystemCallRet ntdll!ZwWaitForMultipleObjects kernel32!WaitForMultipleObjectsEx kernel32!WaitForMultipleObjects WARNING: Stack unwind information not available. Following frames may be wrong. ToolbarA!DllCanUnloadNow ToolbarA!DllCanUnloadNow ToolbarA!DllCanUnloadNow ntdll!__RtlUserThreadStart ntdll!_RtlUserThreadStart  39  Id: 136c.c20 Suspend: 0 Teb: 7ff77000 Unfrozen  ntdll!KiFastSystemCallRet ntdll!ZwRemoveIoCompletion kernel32!GetQueuedCompletionStatus dxtrans!TMThreadProc kernel32!BaseThreadInitThunk ntdll!__RtlUserThreadStart ntdll!_RtlUserThreadStart  40  Id: 136c.e48 Suspend: 0 Teb: 7ff74000 Unfrozen  ntdll!KiFastSystemCallRet ntdll!ZwRemoveIoCompletion kernel32!GetQueuedCompletionStatus dxtrans!TMThreadProc kernel32!BaseThreadInitThunk ntdll!__RtlUserThreadStart ntdll!_RtlUserThreadStart  41  Id: 136c.298 Suspend: 0 Teb: 7ff73000 Unfrozen  ntdll!KiFastSystemCallRet ntdll!ZwRemoveIoCompletion kernel32!GetQueuedCompletionStatus dxtrans!TMThreadProc kernel32!BaseThreadInitThunk ntdll!__RtlUserThreadStart ntdll!_RtlUserThreadStart  42  Id: 136c.11c8 Suspend: 0 Teb: 7ff72000 Unfrozen  ntdll!KiFastSystemCallRet ntdll!ZwRemoveIoCompletion kernel32!GetQueuedCompletionStatus dxtrans!TMThreadProc kernel32!BaseThreadInitThunk ntdll!__RtlUserThreadStart ntdll!_RtlUserThreadStart  43  Id: 136c.1180 Suspend: 0 Teb: 7ff68000 Unfrozen  ntdll!KiFastSystemCallRet ntdll!ZwRemoveIoCompletion kernel32!GetQueuedCompletionStatus</pre>	<pre>dtrans!TMThreadProc kernel32!BaseThreadInitThunk ntdll!__RtlUserThreadStart ntdll!_RtlUserThreadStart  44  Id: 136c.1750 Suspend: 0 Teb: 7ff66000 Unfrozen  ntdll!KiFastSystemCallRet ntdll!ZwRemoveIoCompletion kernel32!GetQueuedCompletionStatus dxtrans!TMThreadProc kernel32!BaseThreadInitThunk ntdll!__RtlUserThreadStart ntdll!_RtlUserThreadStart  45  Id: 136c.16d8 Suspend: 0 Teb: 7ff65000 Unfrozen  ntdll!KiFastSystemCallRet ntdll!ZwRemoveIoCompletion kernel32!GetQueuedCompletionStatus dxtrans!TMThreadProc kernel32!BaseThreadInitThunk ntdll!__RtlUserThreadStart ntdll!_RtlUserThreadStart  46  Id: 136c.15b8 Suspend: 0 Teb: 7ff64000 Unfrozen  ntdll!KiFastSystemCallRet ntdll!ZwRemoveIoCompletion kernel32!GetQueuedCompletionStatus dxtrans!TMThreadProc kernel32!BaseThreadInitThunk ntdll!__RtlUserThreadStart ntdll!_RtlUserThreadStart  47  Id: 136c.b88 Suspend: 0 Teb: 7ff6f000 Unfrozen  ntdll!KiFastSystemCallRet ntdll!ZwRemoveIoCompletion kernel32!GetQueuedCompletionStatus dxtrans!TMThreadProc kernel32!BaseThreadInitThunk ntdll!__RtlUserThreadStart ntdll!_RtlUserThreadStart  48  Id: 136c.434 Suspend: 0 Teb: 7ff6e000 Unfrozen  ntdll!KiFastSystemCallRet ntdll!ZwRemoveIoCompletion kernel32!GetQueuedCompletionStatus dxtrans!TMThreadProc kernel32!BaseThreadInitThunk ntdll!__RtlUserThreadStart ntdll!_RtlUserThreadStart  49  Id: 136c.16e8 Suspend: 0 Teb: 7ff6d000 Unfrozen  ntdll!KiFastSystemCallRet ntdll!ZwRemoveIoCompletion</pre>
---	--

```

kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

50  Id: 136c.f04 Suspend: 0 Teb: 7ff6c000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

51  Id: 136c.128c Suspend: 0 Teb: 7ff67000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

52  Id: 136c.1074 Suspend: 0 Teb: 7ff63000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

53  Id: 136c.2dc Suspend: 0 Teb: 7ff62000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

54  Id: 136c.172c Suspend: 0 Teb: 7ff60000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

```

```

55  Id: 136c.1240 Suspend: 0 Teb: 7ff69000 Unfrozen

ntdll!KiFastSystemCallRet
user32!NtUserWaitMessage
ieframe!CTabWindow::_TabWindowThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

56  Id: 136c.1604 Suspend: 0 Teb: 7ff61000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!D11GetClassObject
ToolbarA!ToolBarMain
ToolbarA!D11CanUnloadNow
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

57  Id: 136c.6a4 Suspend: 0 Teb: 7ff5f000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!D11CanUnloadNow
ToolbarA!D11CanUnloadNow
ToolbarA!D11CanUnloadNow
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

58  Id: 136c.1258 Suspend: 0 Teb: 7ff5e000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!D11CanUnloadNow
ToolbarA!D11CanUnloadNow
ToolbarA!D11CanUnloadNow
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

59  Id: 136c.8c0 Suspend: 0 Teb: 7ff5b000 Unfrozen

ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart

```

60 Id: 136c.868 Suspend: 0 Teb: 7ff58000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

61 Id: 136c.a54 Suspend: 0 Teb: 7ff57000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

62 Id: 136c.77c Suspend: 0 Teb: 7ff56000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

63 Id: 136c.1290 Suspend: 0 Teb: 7ff59000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

64 Id: 136c.1480 Suspend: 0 Teb: 7ff55000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

65 Id: 136c.1270 Suspend: 0 Teb: 7ff54000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

66 Id: 136c.b8c Suspend: 0 Teb: 7ff53000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

67 Id: 136c.167c Suspend: 0 Teb: 7ff92000 Unfrozen

```
ntdll!KiFastSystemCallRet
user32!NtUserWaitMessage
ieframe!CTabWindow::_TabWindowThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

68 Id: 136c.176c Suspend: 0 Teb: 7ff8e000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!DllGetClassObject
ToolbarA!ToolBarMain
ToolbarA!DllCanUnloadNow
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

69 Id: 136c.80c Suspend: 0 Teb: 7ff8b000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!DllCanUnloadNow
ToolbarA!DllCanUnloadNow
ToolbarA!DllCanUnloadNow
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

70 Id: 136c.1570 Suspend: 0 Teb: 7ff8a000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!DllCanUnloadNow
ToolbarA!DllCanUnloadNow
ToolbarA!DllCanUnloadNow
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

71 Id: 136c.e74 Suspend: 0 Teb: 7ff78000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

72 Id: 136c.1490 Suspend: 0 Teb: 7ff76000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

73 Id: 136c.d28 Suspend: 0 Teb: 7ff6a000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

74 Id: 136c.8d8 Suspend: 0 Teb: 7ff5c000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

75 Id: 136c.1064 Suspend: 0 Teb: 7ff4a000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

76 Id: 136c.1478 Suspend: 0 Teb: 7ff47000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

77 Id: 136c.1470 Suspend: 0 Teb: 7ff44000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

78 Id: 136c.aa0 Suspend: 0 Teb: 7ff43000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

79 Id: 136c.1210 Suspend: 0 Teb: 7ff48000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

80 Id: 136c.954 Suspend: 0 Teb: 7ff46000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

81 Id: 136c.9d4 Suspend: 0 Teb: 7ff45000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

82 Id: 136c.f30 Suspend: 0 Teb: 7ff42000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

83 Id: 136c.cc4 Suspend: 0 Teb: 7ff34000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

84 Id: 136c.1018 Suspend: 0 Teb: 7ff33000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

85 Id: 136c.940 Suspend: 0 Teb: 7ff32000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

86 Id: 136c.bd8 Suspend: 0 Teb: 7ff31000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

87 Id: 136c.1714 Suspend: 0 Teb: 7ff84000 Unfrozen

```
ntdll!KiFastSystemCallRet
user32!NtUserWaitMessage
user32!DialogBox2
user32!InternalDialogBox
user32!SoftModalMessageBox
user32!MessageBoxWorker
user32!MessageBoxTimeoutW
user32!MessageBoxTimeoutA
user32!MessageBoxExA
ieframe!Detour_MessageBoxExA
user32!MessageBoxA
WARNING: Stack unwind information not available.
Following frames may be wrong.
MathMLMimer!DllUnregisterServer
MathMLMimer!DllUnregisterServer
MathMLMimer!DllUnregisterServer
```

88 Id: 136c.1744 Suspend: 0 Teb: 7ff83000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!DllGetClassObject
ToolbarA!ToolBarMain
ToolbarA!DllCanUnloadNow
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

89 Id: 136c.9cc Suspend: 0 Teb: 7ff81000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!DllCanUnloadNow
ToolbarA!DllCanUnloadNow
ToolbarA!DllCanUnloadNow
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

90 Id: 136c.f68 Suspend: 0 Teb: 7ff5a000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!DllCanUnloadNow
ToolbarA!DllCanUnloadNow
ToolbarA!DllCanUnloadNow
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

91 Id: 136c.17f4 Suspend: 0 Teb: 7ff49000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

92 Id: 136c.13c Suspend: 0 Teb: 7ff41000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
```

```
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

93 Id: 136c.110 Suspend: 0 Teb: 7ff3f000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

94 Id: 136c.8e4 Suspend: 0 Teb: 7ff3e000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

95 Id: 136c.fcc Suspend: 0 Teb: 7ff4c000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
user32!RealMsgWaitForMultipleObjectsEx
ieui!CoreSC::Wait
ieui!CoreSC::WaitMessage
ieui!WaitMessageEx
ieframe!CBrowserFrame::FrameMessagePump
ieframe!BrowserThreadProc
ieframe!BrowserNewThreadProc
ieframe!SHOpenFolderWindow
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

96 Id: 136c.1378 Suspend: 0 Teb: 7ff4b000 Unfrozen

```
ntdll!KiFastSystemCallRet
user32!NtUserWaitMessage
ieframe!CTabWindow::_TabWindowThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

97 Id: 136c.8ec Suspend: 0 Teb: 7ff40000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!DllGetClassObject
ToolbarA!ToolBarMain
ToolbarA!DllCanUnloadNow
```

```
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

98 Id: 136c.ac Suspend: 0 Teb: 7ff3d000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!DllCanUnloadNow
ToolbarA!DllCanUnloadNow
ToolbarA!DllCanUnloadNow
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

99 Id: 136c.79c Suspend: 0 Teb: 7ff3c000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
WARNING: Stack unwind information not available.
Following frames may be wrong.
ToolbarA!DllCanUnloadNow
ToolbarA!DllCanUnloadNow
ToolbarA!DllCanUnloadNow
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

100 Id: 136c.bd0 Suspend: 0 Teb: 7ff3a000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwWaitForMultipleObjects
kernel32!WaitForMultipleObjectsEx
kernel32!WaitForMultipleObjects
msiltcfg!WorkerThread
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

101 Id: 136c.1504 Suspend: 0 Teb: 7ff36000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

102 Id: 136c.5c0 Suspend: 0 Teb: 7ff35000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
```

```
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

103 Id: 136c.17a0 Suspend: 0 Peb: 7ff2d000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

104 Id: 136c.17c4 Suspend: 0 Peb: 7ff2c000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

105 Id: 136c.f08 Suspend: 0 Peb: 7ffaa000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

106 Id: 136c.1268 Suspend: 0 Peb: 7ff28000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

107 Id: 136c.12c8 Suspend: 0 Peb: 7ff27000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

108 Id: 136c.1634 Suspend: 0 Peb: 7ff26000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
```

```
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

109 Id: 136c.120c Suspend: 0 Peb: 7ff21000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

110 Id: 136c.13f4 Suspend: 0 Peb: 7ff20000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

111 Id: 136c.1494 Suspend: 0 Peb: 7ff1f000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

112 Id: 136c.16ec Suspend: 0 Peb: 7ff1e000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
dxtrans!TMThreadProc
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

113 Id: 136c.d68 Suspend: 0 Peb: 7ffa4000 Unfrozen

```
ntdll!KiFastSystemCallRet
ntdll!ZwRemoveIoCompletion
kernel32!GetQueuedCompletionStatus
rpclt4!COMMON_ProcessCalls
rpclt4!LOADABLE_TRANSPORT::ProcessIOEvents
rpclt4!ProcessIOEventsWrapper
rpclt4!BaseCachedThreadRoutine
rpclt4!ThreadStartRoutine
kernel32!BaseThreadInitThunk
ntdll!__RtlUserThreadStart
ntdll!_RtlUserThreadStart
```

```
114 Id: 136c.5fc Suspend: 0 Teb: 7ffdd000 Unfrozen
```

```
ntdll!KiFastSystemCallRet  
ntdll!ZwWaitForWorkViaWorkerFactory
```

```
ntdll!TppWorkerThread  
kernel32!BaseThreadInitThunk  
ntdll!__RtlUserThreadStart  
ntdll!_RtlUserThreadStart
```

In contrast, *Flash10b* module (shown in bold italics above) is not ubiquitous; it is present on just two threads (#31 and #32).

## Unified Stack Trace

We got the idea of **Unified Stack Trace** analysis pattern from Flame Graphs<sup>213</sup>. Like the latter, we combine **Stack Trace Collection** (page 933) into one aggregate trace, but we may use the same length for repeated frames and may use different color intensities to present multiplicities. Different frame height may also be used to unify top frames such as waiting API. Different collections may be used in addition to database-like (page 919) stack traces (**Unmanaged Space**, page 943, **Managed Space**, page 940, **Predicate**, page 943, **I/O Requests**, page 936, **CPUs**, page 933). The collections may be composed of different varieties of stack traces, such as **General**, page 926, **Managed**, page 624, **Module**, page 699, **Quotient**, page 824, **Filters**, page 924).

As a very simple example, consider this **Stack Trace Collection** from Notepad:

```
0:003> ~*kc

0 Id: 984.994 Suspend: 1 Teb: 00007ff6`f411d000 Unfrozen
# Call Site
00 USER32!NtUserGetMessage
01 USER32!GetMessageW
02 notepad!WinMain
03 notepad!WinMainCRTStartup
04 KERNEL32!BaseThreadInitThunk
05 ntdll!RtlUserThreadStart

1 Id: 984.eb8 Suspend: 1 Teb: 00007ff6`f411b000 Unfrozen
# Call Site
00 ntdll!NtWaitForWorkViaWorkerFactory
01 ntdll!TppWorkerThread
02 KERNEL32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

2 Id: 984.1a8c Suspend: 1 Teb: 00007ff6`f4119000 Unfrozen
# Call Site
00 ntdll!NtWaitForWorkViaWorkerFactory
01 ntdll!TppWorkerThread
02 KERNEL32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart

# 3 Id: 984.11b0 Suspend: 1 Teb: 00007ff6`f4117000 Unfrozen
# Call Site
00 ntdll!DbgBreakPoint
01 ntdll!DbgUiRemoteBreakin
02 KERNEL32!BaseThreadInitThunk
03 ntdll!RtlUserThreadStart
```

---

<sup>213</sup> <http://www.brendangregg.com/flamegraphs.html>

The collection can be represented in a more compact form with multiplicities:

```
USER32!NtUserGetMessage
USER32!GetMessageW
notepad!WinMain      | 2* ntdll!NtWaitForWorkViaWorkerFactory | ntdll!DbgBreakPoint
notepad!WinMainCRTStartup | 2* ntdll!TppWorkerThread           | ntdll!DbgUiRemoteBreakin
4* KERNEL32!BaseThreadInitThunk
4* ntdll!RtlUserThreadStart
```

It can also be illustrated in the following diagram:



USER32!NtUserGetMessage	1	2	3	4
USER32!GetMessageW				
notepad!WinMain	ntdll!NtWaitForWorkViaWorkerFactory	ntdll!DbgBreakPoint		
notepad!WinMainCRTStartup	ntdll!TppWorkerThread	ntdll!DbgUiRemoteBreakin		
KERNEL32!BaseThreadInitThunk				
ntdll!RtlUserThreadStart				

Such diagrams may help to spot **Ubiquitous Components** (page 1020) quickly.

**Unified Stack Trace** is also a generalization of **Stack Trace Set** (page 952) where the latter only excludes fully duplicated stack traces, but the former takes into account **Constant Subtraces** (page 141).

## Unknown Component

Sometimes we suspect that a problem was caused by some module but WinDbg **!mv** command doesn't show the company name and other verbose information for it, and Google search has no results for the file name. We call this pattern **Unknown Component**.

In such cases, additional information can be obtained by dumping the module resource section or the whole module address range and looking for ASCII and UNICODE strings. For example (byte values in **db** output are omitted for clarity):

```
2: kd> !mv m driver
start    end      module name
f5022000 f503e400  driver  (deferred)
Image path: \SystemRoot\System32\drivers\driver.sys
Image name: driver.sys
Timestamp:      Tue Jun 12 11:33:16 2007 (466E766C)
CheckSum:        00021A2C
ImageSize:       0001C400
Translations:    0000.04b0 0000.04e0 0409.04b0 0409.04e0

2: kd> db f5022000 f503e400
f5022000  MZ.....
f5022010  ....@....
f5022020  .....
f5022030  .....
f5022040  .....!..L.!Th
f5022050  is program canno
f5022060  t be run in DOS
f5022070  mode....$.....
f5022080  .g,._.B._.B._.B.
f5022090  _..C.=.B.%Q.X.B.
f50220a0  _..B.].B.Y%H.|.B.
f50220b0  ..D.^B.Rich_.B.
f50220c0  .....PE..L...
f50220d0  lvnF.....
...
...
...
f503ce30  .....
f503ce40  .....
f503ce50  .....
f503ce60  .....0...
f503ce70  .....
f503ce80  ....H.....
f503ce90  .....4...V.
f503cea0  S._.V.E.R.S.I.O.
f503ceb0  N._.I.N.F.O.....
f503cec0  .....
f503ced0  .....?
f503cee0  .....
f503cef0  ....P.....S.t.r.
f503cf00  i.n.g.F.i.l.e.I.
```

```
f503cf10 n.f.o.,....0.
f503cf20 4.0.9.0.4.b.0...
f503cf30 4.....C.o.m.p.a.
f503cf40 n.y.N.a.m.e....
f503cf50 M.y.C.o.m.p. .A.
f503cf60 G...p.$...F.i.l.
f503cf70 e.D.e.s.c.r.i.p.
f503cf80 t.i.o.n....M.y.
f503cf90 .B.i.g. .P.r.o.
f503cfa0 d.u.c.t. .H.o.o.
f503cfb0 k...........
f503cff0 .....4....F.i.l.
f503cfe0 e.V.e.r.s.i.o.n.
f503cff0 ....5...1...0...
f503d000 ????????????????
f503d010 ????????????????
f503d020 ????????????????
f503d030 ????????????????
...
...
...
```

We see that *CompanyName* is *MyComp AG*, *FileDescription* is *My Big Product Hook*, and *FileVersion* is 5.0.1.

In our example the same information can be retrieved by dumping the image file header and then finding and dumping the resource section:

```
2: kd> lmv m driver
start   end     module name
f5022000 f503e400   driver  (deferred)
  Image path: \SystemRoot\System32\drivers\driver.sys
  Image name: driver.sys
  Timestamp:      Tue Jun 12 11:33:16 2007 (466E766C)
  CheckSum:       00021A2C
  ImageSize:      0001C400
  Translations:   0000.04b0 0000.04e0 0409.04b0 0409.04e0

2: kd> !dh f5022000 -f

File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
  14C machine (i386)
    6 number of sections
466E766C time date stamp Tue Jun 12 11:33:16 2007

  0 file pointer to symbol table
  0 number of symbols
  E0 size of optional header
  10E characteristics
    Executable
    Line numbers stripped
    Symbols stripped
    32 bit word machine
```

## OPTIONAL HEADER VALUES

```

10B magic #
6.00 linker version
190A0 size of code
30A0 size of initialized data
0 size of uninitialized data
1A340 address of entry point
2C0 base of code
----- new -----
00010000 image base
20 section alignment
20 file alignment
1 subsystem (Native)
4.00 operating system version
0.00 image version
4.00 subsystem version
1C400 size of image
2C0 size of headers
21A2C checksum
00100000 size of stack reserve
00001000 size of stack commit
00100000 size of heap reserve
00001000 size of heap commit
0 [     0] address [size] of Export Directory
1A580 [     50] address [size] of Import Directory
1AE40 [   348] address [size] of Resource Directory
0 [     0] address [size] of Exception Directory
0 [     0] address [size] of Security Directory
1B1A0 [ 1084] address [size] of Base Relocation Directory
420 [     1C] address [size] of Debug Directory
0 [     0] address [size] of Description Directory
0 [     0] address [size] of Special Directory
0 [     0] address [size] of Thread Storage Directory
0 [     0] address [size] of Load Configuration Directory
0 [     0] address [size] of Bound Import Directory
2C0 [    15C] address [size] of Import Address Table Directory
0 [     0] address [size] of Delay Import Directory
0 [     0] address [size] of COR20 Header Directory
0 [     0] address [size] of Reserved Directory

```

```

2: kd> db f5022000+1AE40 f5022000+1AE40+348
f503ce40 .....
f503ce50 .....
f503ce60 .....0...
f503ce70 .....
f503ce80 ....H.....
f503ce90 .....4..V.
f503cea0 S._.V.E.R.S.I.O.
f503ceb0 N._.I.N.F.O....
f503cec0 .....
f503ced0 .....?.....
f503cee0 .....
f503cef0 ....P.....S.t.r.
f503cf00 i.n.g.F.i.l.e.I.
f503cf10 n.f.o...,....0.
f503cf20 4.0.9.0.4.b.0...

```

```
f503cf30 4.....C.o.m.p.a.  
f503cf40 n.y.N.a.m.e.....  
f503cf50 M.y.C.o.m.p. .A.  
f503cf60 G...p.$...F.i.l.  
f503cf70 e.D.e.s.c.r.i.p.  
f503cf80 t.i.o.n.....M.y.  
f503cf90 .B.i.g. .P.r.o.  
f503cfa0 d.u.c.t. .H.o.o.  
f503cfb0 k.....  
f503cffc0 .....  
f503cfcd0 ....4.....F.i.l.  
f503cfe0 e.V.e.r.s.i.o.n.  
f503cff0 ....5...1...0...  
f503d000 ??????????????????  
f503d010 ??????????????????  
...  
...  
...
```

## Unloaded Module

One of the frequent problems is an access violation at an address that belongs to **Unloaded Module**. Here's an example that recently happened on our machine during an auto-update of the popular software package, so we immediately attached a debugger after seeing a WER dialog box:

```
0:000> ~*k

. 0 Id: bc8.bcc Suspend: 1 Teb: 7efdd000 Unfrozen
ChildEBP RetAddr
0035f1c4 771a0bdd ntdll!ZwWaitForMultipleObjects+0x15
0035f260 75771a2c KERNELBASE!WaitForMultipleObjectsEx+0x100
0035f2a8 75774208 kernel32!WaitForMultipleObjectsExImplementation+0xe0
0035f2c4 757980a4 kernel32!WaitForMultipleObjects+0x18
0035f330 75797f63 kernel32!WerReportFaultInternal+0x186
0035f344 75797858 kernel32!WerReportFault+0x70
0035f354 757977d7 kernel32!BaseReportFault+0x20
0035f3e0 77ec74df kernel32!UnhandledExceptionFilter+0x1af
0035f3e8 77ec73bc ntdll!__RtlUserThreadStart+0x62
0035f3fc 77ec7261 ntdll!_EH4_CallFilterFunc+0x12
0035f424 77eab459 ntdll!_except_handler4+0x8e
0035f448 77eab42b ntdll!ExecuteHandler2+0x26
0035f46c 77eab3ce ntdll!ExecuteHandler+0x24
0035f4f8 77e60133 ntdll!RtlDispatchException+0x127
0035f4f8 73eb2200 ntdll!KiUserExceptionDispatcher+0xf
WARNING: Frame IP not in any known module. Following frames may be wrong.
0035f844 76e462fa <Unloaded_fpb.tmp>+0x12200
0035f870 76e46d3a USER32!InternalCallWinProc+0x23
0035f8e8 76e4965e USER32!UserCallWinProcCheckWow+0x109
0035f92c 76e496c5 USER32!SendMessageWorker+0x581
0035f950 7269c05c USER32!SendMessageW+0x7f
0035f9ec 7270be62 comctl32!CCSendNotify+0xc19
0035fa28 75f6f52a comctl32!SendNotify+0x36
0035fa4c 75f61d66 SHELL32!SetAppStartingCursor+0x6d
0035fa64 75f61ee2 SHELL32!CShellExecute::ExecuteNormal+0x16
0035fa78 75f61e70 SHELL32!ShellExecuteNormal+0x33
0035fa90 75f53cd0 SHELL32!ShellExecuteExW+0x62
0035fae4 003e2211 SHELL32!ShellExecuteW+0x77
0035fbc4 77e838be Install1FlashPlayer+0x2211
0035fcb4 77e83492 ntdll!RtlpFreeHeap+0xbb1
0035fc4d 757714dd ntdll!RtlFreeHeap+0x142
0035fce8 003f0324 kernel32!HeapFree+0x14
0035fd80 003f0241 Install1FlashPlayer+0x10324
0035fe10 7577339a Install1FlashPlayer+0x10241
0035fe1c 77e89ef2 kernel32!BaseThreadInitThunk+0xe
0035fe5c 77e89ec5 ntdll!__RtlUserThreadStart+0x70
0035fe74 00000000 ntdll!_RtlUserThreadStart+0x1b

1 Id: bc8.6b0 Suspend: 2 Teb: 7efda000 Unfrozen
ChildEBP RetAddr
03e1f9e0 77ea2f51 ntdll!ZwWaitForMultipleObjects+0x15
03e1fb74 7577339a ntdll!TppWaiterpThread+0x33d
03e1fb80 77e89ef2 kernel32!BaseThreadInitThunk+0xe
```

```

03e1fbc0 77e89ec5 ntdll!_RtlUserThreadStart+0x70
03e1fb08 00000000 ntdll!_RtlUserThreadStart+0x1b

2 Id: bc8.8dc Suspend: 2 Teb: 7efd7000 Unfrozen
ChildEBP RetAddr
03f5fd50 77ea3352 ntdll!NtWaitForWorkViaWorkerFactory+0x12
03f5feb0 7577339a ntdll!TppWorkerThread+0x216
03f5febc 77e89ef2 kernel32!BaseThreadInitThunk+0xe
03f5fefc 77e89ec5 ntdll!_RtlUserThreadStart+0x70
03f5ff14 00000000 ntdll!_RtlUserThreadStart+0x1b

3 Id: bc8.944 Suspend: 2 Teb: 7efaf000 Unfrozen
ChildEBP RetAddr
0416f8b4 77ea3352 ntdll!NtWaitForWorkViaWorkerFactory+0x12
0416fa14 7577339a ntdll!TppWorkerThread+0x216
0416fa20 77e89ef2 kernel32!BaseThreadInitThunk+0xe
0416fa60 77e89ec5 ntdll!_RtlUserThreadStart+0x70
0416fa78 00000000 ntdll!_RtlUserThreadStart+0x1b

```

The exception thread shows *fpb.tmp* module as unloaded:

```

0:000> lmv m fpb.tmp
start      end          module name

Unloaded modules:
00cb0000 00d5a000  fpb.tmp
Timestamp: Fri Jun 01 02:56:00 2012 (4FC82130)
Checksum: 000B0CD5
ImageSize: 000AA000
73ea0000 73f15000  fpb.tmp
Timestamp: Fri Jun 01 02:49:25 2012 (4FC81FA5)
Checksum: 0007A7CE
ImageSize: 00075000

```

We change the exception thread context to get registers at the time of the exception:

```

0:000> kv
ChildEBP RetAddr Args to Child
0035f1c4 771a0bdd 00000002 0035f214 00000001 ntdll!ZwWaitForMultipleObjects+0x15
0035f260 75771a2c 0035f214 0035f288 00000000 KERNELBASE!WaitForMultipleObjectsEx+0x100
0035f2a8 75774208 00000002 7efde000 00000000 kernel32!WaitForMultipleObjectsExImplementation+0xe0
0035f2c4 757980a4 00000002 0035f2f8 00000000 kernel32!WaitForMultipleObjects+0x18
0035f330 75797f63 0035f410 00000001 00000001 kernel32!WerFaultInternal+0x186
0035f344 75797858 0035f410 00000001 0035f3e0 kernel32!WerFault+0x70
0035f354 757977d7 0035f410 00000001 658587c7 kernel32!BaseReportFault+0x20
0035f3e0 77ec74df 00000000 77ec73bc 00000000 kernel32!UnhandledExceptionFilter+0x1af
0035f3e8 77ec73bc 00000000 0035fe5c 77e7c530 ntdll!_RtlUserThreadStart+0x62
0035f3fc 77ec7261 00000000 00000000 00000000 ntdll!_EH4_CallFilterFunc+0x12
0035f424 77eab459 fffffffe 0035fe4c 0035f560 ntdll!_except_handler4+0x8e
0035f448 77eab42b 0035f510 0035fe4c 0035f560 ntdll!ExecuteHandler2+0x26
0035f46c 77eab3ce 0035f510 0035fe4c 0035f560 ntdll!ExecuteHandler+0x24
0035f4f8 77e60133 0135f510 0035f560 0035f510 ntdll!RtlDispatchException+0x127
0035f4f8 73eb2200 0135f510 0035f560 0035f510 ntdll!KiUserExceptionDispatcher+0xf (CONTEXT @ 0035f560)
WARNING: Frame IP not in any known module. Following frames may be wrong.

```

```

0035f844 76e462fa 000201ce 0000004e 00000000 <Unloaded_fpb.tmp>+0x12200
0035f870 76e46d3a 73eb2200 000201ce 0000004e USER32!InternalCallWinProc+0x23
0035f8e8 76e4965e 00000000 73eb2200 000201ce USER32!UserCallWinProcCheckWow+0x109
0035f92c 76e496c5 013907f0 00000000 73eb2200 USER32!SendMessageWorker+0x581
0035f950 7269c05c 000201ce 0000004e 00000000 USER32!SendMessageW+0x7f
0035f9ec 7270be62 0035fa00 ffffff7 00000000 comctl32!CCSendNotify+0xc19
0035fa28 75f6f52a 000201ce 00000000 ffffff7 comctl32!SendNotify+0x36
0035fa4c 75f61d66 000201ce 00000001 00001500 SHELL32!SetAppStartingCursor+0x6d
0035fa64 75f61ee2 0035faa4 00001500 0035faa4 SHELL32!CShellExecute::ExecuteNormal+0x16
0035fa78 75f61e70 0035faa4 00001500 00000200 SHELL32!ShellExecuteNormal+0x33
0035fa90 75f53cd0 0035faa4 003fb654 003fa554 SHELL32!ShellExecuteExW+0x62
0035fae4 003e2211 000201ce 003fa554 0035fb14 SHELL32!ShellExecuteW+0x77
0035fbc4 77e838be 00da0138 77e8389a 77c467ad InstallFlashPlayer+0x2211
0035fcbb 77e83492 00000000 00da2320 00da2320 ntdll!RtlpFreeHeap+0xbb1
0035fcdb 757714dd 00da0000 00000000 00da2320 ntdll!RtlFreeHeap+0x142
0035fce8 003f0324 00da0000 00000000 003f0343 kernel32!HeapFree+0x14
0035fd80 003f0241 003e0000 00000000 010d3135 InstallFlashPlayer+0x10324
0035fe10 7577339a 7efde000 0035fe5c 77e89ef2 InstallFlashPlayer+0x10241
0035fe1c 77e89ef2 7efde000 77c46545 00000000 kernel32!BaseThreadInitThunk+0xe
0035fe5c 77e89ec5 003f02ac 7efde000 ffffffff ntdll!_RtlUserThreadStart+0x70
0035fe74 00000000 003f02ac 7efde000 00000000 ntdll!_RtlUserThreadStart+0x1b

```

```

0:000> .cxr 0035f560
eax=73eb2200 ebx=00000000 ecx=01080d68 edx=00000000 esi=73eb2200 edi=00000000
eip=73eb2200 esp=0035f848 ebp=0035f870 iopl=0 nv up ei pl nz na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00210206
<Unloaded_fpb.tmp>+0x12200:
73eb2200 ?? ???

```

Then we double check that a window procedure was indeed called from that module range:

```

0:000> kv
*** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr Args to Child
WARNING: Frame IP not in any known module. Following frames may be wrong.
0035f844 76e462fa 000201ce 0000004e 00000000 <Unloaded_fpb.tmp>+0x12200
0035f870 76e46d3a 73eb2200 000201ce 0000004e USER32!InternalCallWinProc+0x23
0035f8e8 76e4965e 00000000 73eb2200 000201ce USER32!UserCallWinProcCheckWow+0x109
0035f92c 76e496c5 013907f0 00000000 73eb2200 USER32!SendMessageWorker+0x581
0035f950 7269c05c 000201ce 0000004e 00000000 USER32!SendMessageW+0x7f
0035f9ec 7270be62 0035fa00 ffffff7 00000000 comctl32!CCSendNotify+0xc19
0035fa28 75f6f52a 000201ce 00000000 ffffff7 comctl32!SendNotify+0x36
0035fa4c 75f61d66 000201ce 00000001 00001500 SHELL32!SetAppStartingCursor+0x6d
0035fa64 75f61ee2 0035faa4 00001500 0035faa4 SHELL32!CShellExecute::ExecuteNormal+0x16
0035fa78 75f61e70 0035faa4 00001500 00000200 SHELL32!ShellExecuteNormal+0x33
0035fa90 75f53cd0 0035faa4 003fb654 003fa554 SHELL32!ShellExecuteExW+0x62
0035fae4 003e2211 000201ce 003fa554 0035fb14 SHELL32!ShellExecuteW+0x77
0035fbc4 77e838be 00da0138 77e8389a 77c467ad InstallFlashPlayer+0x2211
0035fcbb 77e83492 00000000 00da2320 00da2320 ntdll!RtlpFreeHeap+0xbb1
00da15a0 00000000 00da1780 02971450 003e1000 ntdll!RtlFreeHeap+0x142

```

```
0:000> ub 76e462fa
USER32!InternalCallWinProc+0x6:
76e462dd 68cdabbd push 0DCBAABCDh
76e462e2 56 push esi
76e462e3 ff7518 push dword ptr [ebp+18h]
76e462e6 ff7514 push dword ptr [ebp+14h]
76e462e9 ff7510 push dword ptr [ebp+10h]
76e462ec ff750c push dword ptr [ebp+0Ch]
76e462ef 64800dca0f000001 or byte ptr fs:[0FCAh],1
76e462f7 ff5508 call dword ptr [ebp+8]
```

We now get a memory value pointed to by an *EBP+8* address:

```
0:000> r
Last set context:
eax=73eb2200 ebx=00000000 ecx=01080d68 edx=00000000 esi=73eb2200 edi=00000000
eip=73eb2200 esp=0035f848 ebp=0035f870 iopl=0 nv up ei pl nz na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00210206
<Unloaded_fpb.tmp>+0x12200:
73eb2200 ?? ???
```

```
0:000> dp 0035f870+8 11
0035f878 73eb2200
```

```
0:000> dd 73eb2200
73eb2200 ?????????? ?????????? ?????????? ??????????
73eb2210 ?????????? ?????????? ?????????? ??????????
73eb2220 ?????????? ?????????? ?????????? ??????????
73eb2230 ?????????? ?????????? ?????????? ??????????
73eb2240 ?????????? ?????????? ?????????? ??????????
73eb2250 ?????????? ?????????? ?????????? ??????????
73eb2260 ?????????? ?????????? ?????????? ??????????
73eb2270 ?????????? ?????????? ?????????? ??????????
```

The value indeed belongs to unloaded *fpb.tmp* module address range:

```
0:000> ln 73eb2200
(73eb2200) <UnLoaded_fpb.tmp>+0x12200
```

## Unrecognizable Symbolic Information

Sometimes debugging information is absent from module information in memory dumps and a debugger can't recognize and automatically load symbol files. For example, we see this stack trace without loaded component symbols:

```

THREAD 8a17c6d8 Cid 02ec.02f0 Teb: 7ffffd000 Win32Thread: e17b4420 WAIT: (UserRequest) UserMode Non-
Alertable
 89873d00 SynchronizationEvent
IRP List:
  89d9fd20: (0006,0094) Flags: 00000800 Mdl: 00000000
Not impersonating
DeviceMap          e10086c8
Owning Process     0      Image:      <Unknown>
Attached Process   8a17cda0    Image:      ApplicationA.exe
Wait Start TickCount 8164394    Ticks: 2884 (0:00:00:45.062)
Context Switch Count 1769160           LargeStack
UserTime            00:00:55.250
KernelTime          00:01:56.109
Start Address 0x0103e5e1
Stack Init ba390000 Current ba38fca0 Base ba390000 Limit ba38b000 Call 0
Priority 15 BasePriority 15 PriorityDecrement 0 DecrementCount 0
*** ERROR: Module load completed but symbols could not be loaded for ModuleA.dll
ChildEBP RetAddr
ba38fcb8 80503836 nt!KiSwapContext+0x2f
ba38fcc4 804fb068 nt!KiSwapThread+0x8a
ba38fce4 805c0750 nt!KeWaitForSingleObject+0x1c2
ba38fd50 8054161c nt!NtWaitForSingleObject+0x9a
ba38fd50 7c90e4f4 nt!KiFastCallEntry+0xfc (TrapFrame @ ba38fd64)
0006f648 7c90df3c ntdll!KiFastSystemCallRet
0006f64c 7c91b22b ntdll!NtWaitForSingleObject+0xc
0006f6d4 7c901046 ntdll!RtlpWaitForCriticalSection+0x132
0006f6dc 01373df7 ntdll!RtlEnterCriticalSection+0x46
WARNING: Stack unwind information not available. Following frames may be wrong.
0006f7a4 0132b785 ModuleA+0x53df7
0006f7cc 0132c728 ModuleA+0xb785
0006f7e4 01346426 ModuleA+0xc728
0006f848 7e418734 ModuleA+0x26426
0006f874 7e418816 USER32!InternalCallWinProc+0x28
0006f8dc 7e4189cd USER32!UserCallWinProcCheckWow+0x150
0006f93c 7e418a10 USER32!DispatchMessageWorker+0x306
0006f94c 0084367e USER32!DispatchMessageW+0xf

0: kd> .process /r /p 8a17cda0
Implicit process is now 8a17cda0
Loading User Symbols

```

```

0: kd> lmv m ModuleA
start      end          module name
01320000 013bb000  ModuleA  (deferred)
Image path: C:\Program Files\VendorA\ModuleA.dll
Image name: ModuleA.dll
Timestamp:    Thu Aug 11 21:42:08 2011 (4E4484F0)
CheckSum:     000A9C8B
ImageSize:    0009B000
Translations: 0000.04b0 0000.04e4 0409.04b0 0409.04e4

0: kd> !lmi ModuleA
Loaded Module Info: [ModuleA]
    Module: ModuleA
    Base Address: 01320000
    Image Name: ModuleA.dll
    Machine Type: 332 (I386)
    Time Stamp: 4e4484f0 Thu Aug 11 21:42:08 2011
    Size: 9b000
    CheckSum: a9c8b
Characteristics: 2102
Debug Data Dirs: Type   Size   VA   Pointer
                  CODEVIEW 5e, 830a0, 830a0 [Debug data not mapped] - can't validate symbols, if present.
Symbol Type: DEFERRED - No error - symbol load deferred
Load Report: no symbols loaded

```

However, in **Stack Trace Collection** (page 943, **!process 0 3f** WinDbg command) we find another stack trace from a different process but with loaded symbol files for *ModuleA*:

```

THREAD 89703020  Cid 1068.1430  Teb: 7fffd000 Win32Thread: e34d43a8 WAIT: (UserRequest) UserMode Non-
Alertable
89a3ac58  NotificationEvent
89703110  NotificationTimer
IRP List:
899ab488: (0006,0094) Flags: 00000900  Mdl: 00000000
Not impersonating
DeviceMap           e10086c8
Owning Process      0      Image:       <Unknown>
Attached Process    89825020  Image:       ApplicationB.exe
Wait Start TickCount 8164457   Ticks: 2821 (0:00:00:44.078)
Context Switch Count 552        LargeStack
UserTime             00:00:00.296
KernelTime           00:00:00.890
Start Address 0x0103e5e1
Stack Init b8796000 Current b8795ca0 Base b8796000 Limit b8791000 Call 0
Priority 15 BasePriority 15 PriorityDecrement 0 DecrementCount 0
ChildEBP RetAddr
b8795cb8 80503836 nt!KiSwapContext+0x2f
b8795cc4 804fb068 nt!KiSwapThread+0x8a
b8795cec 805c0750 nt!KeWaitForSingleObject+0x1c2
b8795d50 8054161c nt!NtWaitForSingleObject+0x9a
b8795d50 7c90e4f4 nt!KiFastCallEntry+0xfc (TrapFrame @ b8795d64)
0006fa1c 7c90df3c ntdll!KiFastSystemCallRet
0006fa20 7c8025db ntdll!NtWaitForSingleObject+0xc
0006fa84 010ae96a kernel32!WaitForSingleObjectEx+0xa8
0006fafc 010aeaaf ModuleA!Wait+0xaa

```

```
0006fb38 010b84ce ModuleA!Read+0x6f
[...]

0: kd> !lmi ModuleA
Loaded Module Info: [ModuleA]
Module: ModuleA
Base Address: 01090000
Image Name: ModuleA.dll
Machine Type: 332 (I386)
Time Stamp: 4e4484f0 Thu Aug 11 21:42:08 2011
Size: 9b000
CheckSum: a9c8b
Characteristics: 2102
Debug Data Dirs: Type Size VA Pointer
CODEVIEW 5e, 830a0, 830a0 RSDS - GUID: {C14E734A-367F-4DD0-974D-FA47C1194F28}
Age: 1, Pdb: Y:\src\...\ModuleA.pdb
Symbol Type: DEFERRED - No error - symbol load deferred
Load Report: no symbols loaded

0: kd> lmv m ModuleA
start end module name
01090000 0112b000 ModuleA (deferred)
Image path: C:\Program Files\VendorA\ModuleA.dll
Image name: ModuleA.dll
Timestamp: Thu Aug 11 21:42:08 2011 (4E4484F0)
CheckSum: 000A9C8B
ImageSize: 0009B000
File version: 1.3.0.0
Product version: 1.3.0.0
File flags: 8 (Mask 3F) Private
File OS: 40004 NT Win32
File type: 2.0 Dll
File date: 00000000.00000000
Translations: 0409.04b0
CompanyName: VendorA
ProductName: VendorA
InternalName: ModuleA.dll
OriginalFilename: ModuleA.dll
ProductVersion: 1.3
FileVersion: 1.3.0.0
FileDescription: ModuleA GUI
LegalCopyright: Copyright VendorA
```

So we switch to that thread (with the new process context) to get the needed symbol path:

```
0: kd> .thread /r /p 89703020
Implicit thread is now 89703020
Implicit process is now 89825020
Loading User Symbols
```

```
0: kd> kL
*** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr
b8795cb8 80503836 nt!KiSwapContext+0x2f
b8795cc4 804fb068 nt!KiSwapThread+0x8a
b8795cec 805c0750 nt!KeWaitForSingleObject+0x1c2
b8795d50 8054161c nt!NtWaitForSingleObject+0x9a
b8795d50 7c90e4f4 nt!KiFastCallEntry+0xfc
0006fa1c 7c90df3c ntdll!KiFastSystemCallRet
0006fa20 7c8025db ntdll!NtWaitForSingleObject+0xc
0006fa84 010ae96a kernel32!WaitForSingleObjectEx+0xa8
0006fafc 010aeaaf ModuleA!Wait+0xaa
0006fb38 010b84ce ModuleA!Read+0x6f
[...]

0: kd> lmv m ModuleA
start      end          module name
01090000 0112b000  ModuleA    (private pdb
symbols)  c:\sym\ModuleA.pdb\C14E734A367F4DD0974DFA47C1194F281\ModuleA.pdb
Loaded symbol image file: ModuleA.dll
[...]
```

Now we switch back to our problem stack trace and set the found symbol path explicitly:

```
0: kd> .thread /r /p 8a17c6d8
Implicit thread is now 8a17c6d8
Implicit process is now 8a17cd0
Loading User Symbols

0: kd> kL
*** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr
ba38fcb8 80503836 nt!KiSwapContext+0x2f
ba38fcc4 804fb068 nt!KiSwapThread+0x8a
ba38fce4 805c0750 nt!KeWaitForSingleObject+0x1c2
ba38fd50 8054161c nt!NtWaitForSingleObject+0x9a
ba38fd50 7c90e4f4 nt!KiFastCallEntry+0xfc
0006f648 7c90df3c ntdll!KiFastSystemCallRet
0006f64c 7c91b22b ntdll!NtWaitForSingleObject+0xc
0006f6d4 7c901046 ntdll!RtlpWaitForCriticalSection+0x132
*** ERROR: Module load completed but symbols could not be loaded for ModuleA.dll
0006f6dc 01373df7 ntdll!RtlEnterCriticalSection+0x46
WARNING: Stack unwind information not available. Following frames may be wrong.
0006f7a4 0132b785 ModuleA+0x53df7
0006f7cc 0132c728 ModuleA+0xb785
0006f7e4 01346426 ModuleA+0xc728
0006f848 7e418734 ModuleA+0x26426
0006f874 7e418816 USER32!InternalCallWinProc+0x28
0006f8dc 7e4189cd USER32!UserCallWinProcCheckWow+0x150
0006f93c 7e418a10 USER32!DispatchMessageWorker+0x306
0006f94c 0084367e USER32!DispatchMessageW+0xf
[...]
```

```
0: kd> .sympath+ c:\sym\ModuleA.pdb\C14E734A367F4DD0974DFA47C1194F281
Symbol search path is: SRV*c:\mss*http://msdl.microsoft.com/download/symbols;
c:\sym\ModuleA.pdb\C14E734A367F4DD0974DFA47C1194F281
[...]

0: kd> .reload
Loading Kernel Symbols
Loading User Symbols
Loading unloaded module list

0: kd> kL
*** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr
ba38fcb8 80503836 nt!KiSwapContext+0x2f
ba38fcc4 804fb068 nt!KiSwapThread+0x8a
ba38fce4 805c0750 nt!KeWaitForSingleObject+0x1c2
ba38fd50 8054161c nt!NtWaitForSingleObject+0x9a
ba38fd50 7c90e4f4 nt!KiFastCallEntry+0xfc
0006f648 7c900df3c ntdll!KiFastSystemCallRet
0006f64c 7c91b22b ntdll!NtWaitForSingleObject+0xc
0006f6d4 7c901046 ntdll!RtlpWaitForCriticalSection+0x132
0006f6dc 01373df7 ntdll!RtlEnterCriticalSection+0x46
0006f6e4 0132b22e ModuleA!CSLock+0x7
0006f7a4 0132b785 ModuleA!SignalEvent+0x5e
[...]
0006f848 7e418734 ModuleA!WindowProc+0x136
0006f874 7e418816 USER32!InternalCallWinProc+0x28
0006f8dc 7e4189cd USER32!UserCallWinProcCheckWow+0x150
0006f93c 7e418a10 USER32!DispatchMessageWorker+0x306
0006f94c 0084367e USER32!DispatchMessageW+0xf
[...]
```

## Unsynchronized Dumps

For the analysis of memory dumps from **Coupled Processes** (page 149) or, in general, memory fibers from fiber bundle memory spaces<sup>214</sup> we need to know their creation times (called debug session time). In some cases, we need to know their time sequence: which process memory dump was saved first and how much time had passed before the second process memory dump was saved. Besides an initial output when we open a dump, `.time` and `version` WinDbg commands can be used to check this information at any time during the analysis.

In one example involving printing, we see **Blocking Thread** (82) trying to contact print spooler service using LPC. Its **Thread Age** (page 1001) is no more than 3 seconds. We also have the print spooler service process memory dump supposedly taken at the same time. However, when we check it, we see it was saved 2 minutes before. Moreover, *PrintIsolationHost.exe* process memory dump was saved even earlier. So the whole sequence was reversed because the printing application calls the spooler, and it calls the appropriate driver, not the way around.

---

<sup>214</sup> Fiber Bundle of Memory Space, Memory Dump Analysis Anthology, Volume 4, page 357

## User Space Evidence

One of the questions asked was what can we do if we got a kernel memory dump instead of the requested complete memory dump? Can it be useful? Of course, if we requested a complete memory dump after analyzing a kernel memory dump then the second kernel dump may be useful for double checking. Therefore, we assume that we just got a kernel memory dump for the first time, and the issue is some performance issue or system freeze and not a bugcheck. If we have a bugcheck, then kernel memory dumps are sufficient most of the time, and we do not consider them for this pattern.

Such a kernel memory dump is still useful because of user space diagnostic indicators pointing to possible patterns in user space or “interspace”. We call this pattern **User Space Evidence**. It is a collective super-pattern like **Historical Information** (page 483).

We can see patterns in kernel memory dumps such as **Wait Chains** (for example, **ALPC**, page 1097, or **Process Objects**, page 1108), **Deadlocks** (for example **ALPC**, page 197), kernel stack traces corresponding to specific **Dual Stack Traces** (page 282, for example, exception processing), **Handle Leaks** (page 416), **Missing Threads** (page 683), **Module Product Process** (page 698), **One-Thread Processes** (page 765), **Spiking Thread** (page 885), **Process Factory** (page 814, for example, Parent PID for **Zombie Processes**, page 1158), and others.

Found evidence may point to specific processes and process groups (**Coupled Processes**, page 148, and session processes) and suggest process memory dump collection (especially forcing further complete memory dumps is problematic) or troubleshooting steps for diagnosed processes.

## V

## Value Adding Process

This is a frequently observed pattern in terminal services environments when we see one or several process names listed in each session but not necessarily required. They are usually running to provide some user experience enhancements. In such cases, if observed functional problems correspond to the purpose of running additional processes we might want to eliminate them for testing and troubleshooting purposes.

```
0: kd> !sprocess 12
Dumping Session 12

_MM_SESSION_SPACE ffffff8800e5d5000
_MMSESSION ffffff8800e5d5b40
PROCESS ffffffa8008d50b30
SessionId: 12 Cid: 0b04 Peb: 7fffffff000 ParentCid: 1478
DirBase: 6bb77000 ObjectTable: fffff8a003f280b0 HandleCount: 158.
Image: csrss.exe

PROCESS ffffffa80030c7060
SessionId: 12 Cid: 1a48 Peb: 7fffffff8000 ParentCid: 1478
DirBase: 0a33c000 ObjectTable: fffff8a003c46c00 HandleCount: 179.
Image: winlogon.exe

PROCESS ffffffa8008250b30
SessionId: 12 Cid: 18c8 Peb: 7fffffff000 ParentCid: 1a48
DirBase: 0350d000 ObjectTable: fffff8a0025b6840 HandleCount: 226.
Image: LogonUI.exe

PROCESS ffffffa8008b00530
SessionId: 12 Cid: 1508 Peb: 7fffffff000 ParentCid: 02f0
DirBase: 02f65000 ObjectTable: fffff8a003b7e530 HandleCount: 197.
Image: ExcitingFeatureX.exe

[...]
```

## Value Deviation

### Stack Trace

Memory dump analysis is all about deviations, and one of them is **Value Deviation** (a super pattern), be it a number of open handles (**Insufficient Memory**, page 526), heap size (**Memory Leak**, page 650), a number of contended locks (**Swarm of Shared Locks**, page 966), time spent in kernel (**Spiking Thread**, page 888), and many others. Every system or process property has its average and mean values, and large deviations are noticeable as the so-called anomalies. Here we provide an example of a stack trace size (depth) deviation. The average number of frames for most stack traces is dependent on the type of a memory dump: user, kernel and complete but considerably longer or shorter stack traces are clearly visible in **Stack Trace Collections** (page 943). We originally planned to call this pattern **Black Swan** but after a moment of thought dismissed that idea because such deviations are not really rare, after all. Here is an example of **Stack Trace Collection** from a CPU spiking process with a number of identical stack traces with just only three frames:

```
0:000> ~*kL

[...]

19  Id: 1054.1430 Suspend: 1 Teb: 7ff9c000 Unfrozen
ChildEBP RetAddr
1ac6ff50 7739bf53 ntdll!KiFastSystemCall1Ret
1ac6ffb8 77e6482f user32!NtUserWaitMessage+0xc
1ac6ffec 00000000 kernel32!BaseThreadStart+0x34

20  Id: 1054.c90 Suspend: 1 Teb: 7ffaf000 Unfrozen
ChildEBP RetAddr
1b30ff50 7739bf53 ntdll!KiFastSystemCall1Ret
1b30ffb8 77e6482f user32!NtUserWaitMessage+0xc
1b30ffec 00000000 kernel32!BaseThreadStart+0x34

21  Id: 1054.a34 Suspend: 1 Teb: 7ff9a000 Unfrozen
ChildEBP RetAddr
1b63ff50 7739bf53 ntdll!KiFastSystemCall1Ret
1b63ffb8 77e6482f user32!NtUserWaitMessage+0xc
1b63ffec 00000000 kernel32!BaseThreadStart+0x34

22  Id: 1054.1584 Suspend: 1 Teb: 7ff99000 Unfrozen
ChildEBP RetAddr
1ba9ff50 7739bf53 ntdll!KiFastSystemCall1Ret
1ba9ffb8 77e6482f user32!NtUserWaitMessage+0xc
1ba9ffec 00000000 kernel32!BaseThreadStart+0x34

[...]
```

These stack traces are correct from *RetAddr* analysis perspective:

```
0:000> ub 7739bf53
user32!PeekMessageW+0x11e:
7739bf42 nop
7739bf43 nop
7739bf44 nop
7739bf45 nop
7739bf46 nop
user32!NtUserWaitMessage:
7739bf47 mov     eax,124Ah
7739bf4c mov     edx,offset SharedUserData!SystemCallStub (7ffe0300)
7739bf51 call    dword ptr [edx]

0:000> ub 77e6482f
kernel32!BaseThreadStart+0x10:
77e6480b mov     eax,dword ptr fs:[00000018h]
77e64811 cmp     dword ptr [eax+10h],1E00h
77e64818 jne     kernel32!BaseThreadStart+0x2e (77e64829)
77e6481a cmp     byte ptr [kernel32!BaseRunningInServerProcess (77ecb008)],0
77e64821 jne     kernel32!BaseThreadStart+0x2e (77e64829)
77e64823 call    dword ptr [kernel32!_imp__CsrNewThread (77e4132c)]
77e64829 push   dword ptr [ebp+0Ch]
77e6482c call    dword ptr [ebp+8]
```

Looking at their thread times reveals that they were the most spikers:

```
0:000> !runaway
User Mode Time
Thread      Time
19:1430      0 days 0:01:34.109
22:1584      0 days 0:01:28.140
21:a34       0 days 0:01:26.765
20:c90       0 days 0:01:24.218
0:e78        0 days 0:00:01.687
10:398       0 days 0:00:01.062
7:14e8        0 days 0:00:00.250
4:1258       0 days 0:00:00.093
6:2e8         0 days 0:00:00.015
1:11c0        0 days 0:00:00.015
26:1328      0 days 0:00:00.000
25:7ec        0 days 0:00:00.000
[...]
```

In order to hypothesize about a possible culprit component, we look at **Execution Residue** (page 371) left on their raw stack data. Indeed, we see a lot of non-**Coincidental Symbolic Information** references (page 137) to *3rdPartyExtension* module:

```
0:000> ~22s
eax=00000000 ebx=00000000 ecx=1ba9f488 edx=00000001 esi=1952bd40 edi=00000000
eip=7c82860c esp=1ba9ff54 ebp=1ba9ffb8 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00240246
ntdll!KiFastSystemCallRet:
7c82860c ret

0:022> !teb
TEB at 7ff99000
ExceptionList: 1ba9ffdcc
StackBase: 1baa0000
StackLimit: 1ba8f000
SubSystemTib: 00000000
FiberData: 00001e00
ArbitraryUserPointer: 00000000
Self: 7ff99000
EnvironmentPointer: 00000000
ClientId: 00001054 . 00001584
RpcHandle: 00000000
Tls Storage: 00000000
PEB Address: 7ffd5000
LastErrorValue: 0
LastStatusValue: c0000034
Count Owned Locks: 0
HardErrorMode: 0

0:022> dds 1ba8f000 1baa0000
1ba8f000 00000000
1ba8f004 00000000
[...]
1ba939e8 00000000
1ba939ec 00000000
1ba939f0 00000037
1ba939f4 1906e6c0
1ba939f8 064e1112 3rdPartyExtension!DllUnregisterServer+0xe1f1f
1ba939fc 1a042678
1ba93a00 034d2918
1ba93a04 00000000
1ba93a08 1a042660
1ba93a0c 00000008
1ba93a10 064e18ea 3rdPartyExtension!DllUnregisterServer+0xe26f7
1ba93a14 1a042678
1ba93a18 00000001
1ba93a1c 034d2870
1ba93a20 034d2b78
1ba93a24 0000001f
1ba93a28 00000007
1ba93a2c 034d2870
1ba93a30 1a01fc68
1ba93a34 00000001
1ba93a38 1ba93a54
1ba93a3c 064e1b45 3rdPartyExtension!DllUnregisterServer+0xe2952
1ba93a40 034d2b78
1ba93a44 00000000
1ba93a48 00000000
1ba93a4c 06e7b498
```

## 1056 | Value Deviation

```
1ba93a50 00000212
1ba93a54 1ba93c00
1ba93a58 064e3bce 3rdPartyExtension!DllUnregisterServer+0xe49db
1ba93a5c 00000001
1ba93a60 00000001
1ba93a64 00000000
1ba93a68 115d7fbc
1ba93a6c 06e7b498
1ba93a70 062de91d 3rdPartyExtension+0xe91d
1ba93a74 0000020c
1ba93a78 1ba93b78
1ba93a7c 06363797 3rdPartyExtension+0x93797
1ba93a80 00000024
1ba93a84 00000000
1ba93a88 00000000
1ba93a8c 1ba93ee0
[...]

0:022> ub 064e1112
3rdPartyExtension!DllUnregisterServer+0xe1f0d:
064e1100 jge    3rdPartyExtension!DllUnregisterServer+0xe1f16 (064e1109)
064e1102 mov    ecx,dword ptr [ecx+10h]
064e1105 cmp    ecx,eax
064e1107 jne    3rdPartyExtension!DllUnregisterServer+0xe1f0a (064e10fd)
064e1109 push   ecx
064e110a push   ebx
064e110b mov    ecx,edi
064e110d call   3rdPartyExtension!DllUnregisterServer+0xe1d17 (064e0f0a)
```

## Value References

Sometimes we have a value or a pointer or a handle and would like to know all memory addresses that reference it. This can be done by virtual memory search (`s` WinDbg command). If we look for references in the code (for example, for pool tags, please see this case study<sup>215</sup>) we can combine search with `!for_each_module` WinDbg extension command. There is also `!search` command for physical pages. We cover this **Value References** pattern in Advanced Windows Memory Dump Analysis training<sup>216</sup> with a step-by-step complete memory dump analysis exercise. For object references there is also recently added `!objtrace` command with good examples in WinDbg help.

## Comments

A useful command to find pointers to a value in the whole virtual address space: `!heap -x -v`.

If we have an object with handle references to it we can search for its process handle container, for example, for zombie processes, we can dump all handle tables from all processes: `!handle 0 3 0 Process`.

If we have an object address we can use `!findhandle` to find its process container:

```
002c: Object: ffffffa80a95cb610 GrantedAccess: 001fffff Entry: fffff8a0000030b0
Object: ffffffa80a95cb610 Type: (fffffa80a943bf30) Thread
ObjectHeader: ffffffa80a95cb5e0 (new version)
HandleCount: 1 PointerCount: 2

0: kd> !findhandle ffffffa80a95cb610
Now checking process ffffffa80a943b6d0...
[fffffa80a943b6d0 System]
2c: Entry fffff8a0000030b0 Granted Access 1fffff
```

<sup>215</sup> The Search for Tags, Memory Dump Analysis Anthology, Volume 1, page 206

<sup>216</sup> <http://www.patterndiagnostics.com/advanced-windows-memory-dump-analysis>

## Variable Subtrace

When analyzing **Spiking Threads** (page 885) across **Snapshot Collection**<sup>217</sup> we are interested in finding a module (or a function) that was most likely responsible (for example, “looping” inside). Here we can compare the same thread stack trace from different memory dumps and find their **Variable Subtrace**. For such subtraces, we have changes in the **kv**-style output: in return addresses, stack frame values, and possible arguments. The call site that starts the variable subtrace is the most likely candidate (subject to the number of snapshots). For example, consider the following pseudo code:

```
ModuleA!start()
{
    ModuleA!func1();
}
ModuleA!func1()
{
    ModuleB!func2();
}
ModuleB!func2()
{
    while (... )
    {
        ModuleB!func3();
    }
}
ModuleB!func3()
{
    ModuleB!func4();
}
ModuleB!func4()
{
    ModuleB!func5();
}
ModuleB!func5()
{
    // ...
}
```

Here, the variable stack trace part will correspond to *ModuleB* frames. The memory dump can be saved anywhere inside the “while” loop and down the calls, and the last variable return address down the stack trace will belong to *ModuleB!func2* address range. The non-variable part will start with *ModuleA!func1* address range:

---

<sup>217</sup> Snapshot Collection, Structural Memory Patterns, Memory Dump Analysis Anthology, Volume 5, page 346

```
// snapshot 1

RetAddr
ModuleB!func4+0x20
ModuleB!func3+0x10
ModuleB!func2+0x40
ModuleA!func1+0x10
ModuleA!start+0x300

// snapshot 2

RetAddr
ModuleB!func2+0x20
ModuleA!func1+0x10
ModuleA!start+0x300

// snapshot 3

RetAddr
ModuleB!func3+0x20
ModuleB!func2+0x40
ModuleA!func1+0x10
ModuleA!start+0x300
```

To illustrate this analysis pattern we adopted Memory Cell Diagram (MCD) approach from Accelerated Disassembly, Reconstruction and Reversing<sup>218</sup> training and introduce here Abstract Stack Trace Notation (ASTN) diagrams where different colors are used for different modules, and changes are highlighted with different fill patterns. The following three ASTN diagrams from subsequently saved process memory dumps illustrate real stack traces we analyzed some time ago. We see that the variable subtrace contains only the 3rd-party *ModuleB* calls. Moreover, the loop is possibly contained inside *ModuleB* because all *ModuleA* frames are non-variable including *Child-SP* and *Args* column values.

<sup>218</sup> <http://www.dumpanalysis.org/accelerated-disassembly-reconstruction-reversing-book>

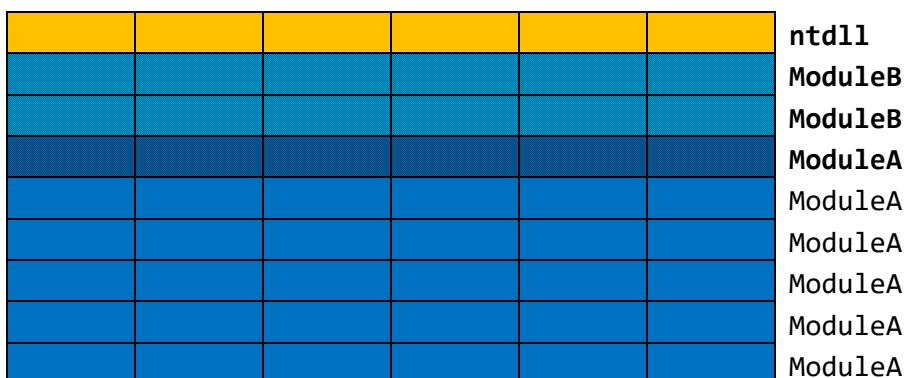
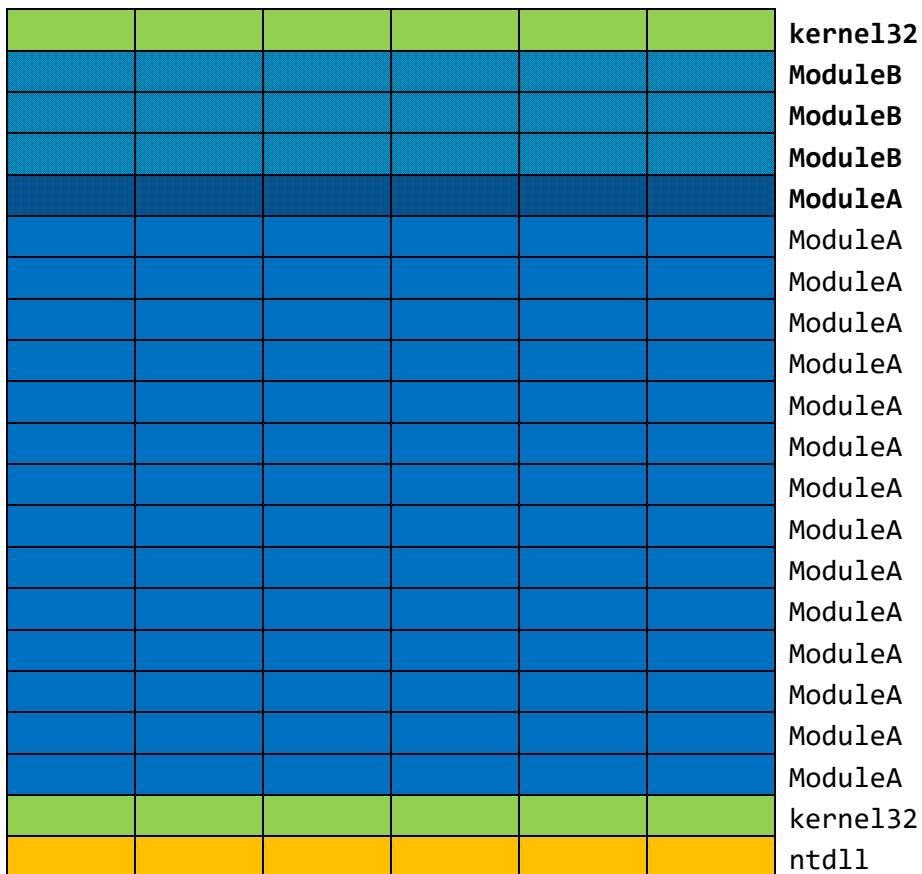
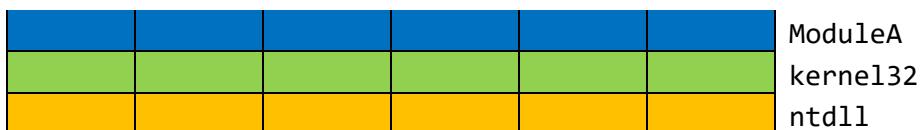
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						kernel32
						ntdll

						kernel32
						ModuleB
						ModuleB
						ModuleB
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						kernel32
						ntdll

						ntdll
						ModuleB

						ModuleB
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						kernel32
						ntdll

If we had ASTN diagrams below instead we would have concluded that the loop was in *ModuleA* with changes in *ModuleB* columns as an execution side effect:



						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						ModuleA
						kernel32
						ntdll

## Version-Specific Extension

This is a pattern similar to **Platform-Specific Debugger** pattern (page 804) by suggesting the course of the further debugging actions. Similar instructions are given when a debugger depends on specialized modules differing from platform (or application) version. We consider here a .NET example where opening a dump shows only that it was perhaps saved manually (**Manual Dump**, page 630) with possible **Hidden Exceptions** (page 457) that need to be dug out:

```
0:000> !analyze -v

FAULTING_IP:
+0
00000000`00000000 ?? ???

EXCEPTION_RECORD: ffffffff`fffffff -- (.exr 0xffffffff`fffffff)
ExceptionAddress: 00000000`00000000
ExceptionCode: 80000003 (Break instruction exception)
ExceptionFlags: 00000000
NumberParameters: 0
```

We notice a failed attempt for .NET analysis and the following instructions on how to correct it:

```
MANAGED_STACK: !dumpstack -EE
Failed to Load data access DLL, 0x80004005
Verify that 1) you have a recent build of the debugger (6.2.14 or newer)
2) the file mscordacwks.dll that matches your version of mscorewks.dll is in the version directory
3) or, if you are debugging a dump file, verify that the file mscordacwks_<arch>_<arch>_<version>.dll is
on your symbol path.
4) you are debugging on the same architecture as the dump file. For example, an IA64 dump file must be
debugged on an IA64 machine.
```

You can also run the debugger command .cordll to control the debugger's load of mscorewks.dll. .cordll -ve -u -l will do a verbose reload. If that succeeds, the SOS command should work on retry.

If you are debugging a minidump, you need to make sure that your executable path is pointing to mscorewks.dll as well.

Because we know that we have .NET framework installed on a postmortem debugging machine we check the target module version:

```
0:000> lm v m mscorewks
start end module name
000007fe`ee380000 000007fe`eed1d000 mscorewks (pdb symbols)
Loaded symbol image file: mscorewks.dll
Image path: C:\Windows\Microsoft.NET\Framework64\v2.0.50727\mscorewks.dll
Image name: mscorewks.dll
Timestamp: Sun Feb 06 20:53:54 2011 (4D4F0A62)
CheckSum: 00990593
ImageSize: 0099D000
File version: 2.0.50727.5444
```

```

Product version: 2.0.50727.5444
File flags:      0 (Mask 3F)
File OS:        4 Unknown Win32
File type:      2.0 Dll
File date:      00000000.00000000
Translations:   0409.04b0
CompanyName:   Microsoft Corporation
ProductName:   Microsoft® .NET Framework
InternalName:  mscorewks.dll
OriginalFilename: mscorewks.dll
ProductVersion: 2.0.50727.5444
FileVersion:    2.0.50727.5444 (Win7SP1GDR.050727-5400)
FileDescription: Microsoft .NET Runtime Common Language Runtime - WorkStation
LegalCopyright: © Microsoft Corporation. All rights reserved.
Comments: Flavor=Retail

```

It is slightly newer (.5444) than we have installed (.3619). The customer also sent their framework version together with the memory dump file. So we unload the current SOS extension (for details, please see **Managed Code Exception** pattern, page 617):

```

0:000> .chain
Extension DLL chain:
C:\Windows\Microsoft.NET\Framework64\v2.0.50727\sos: image 2.0.50727.3619, API 1.0.0, built Mon Oct 25
06:52:09 2010
[path: C:\Windows\Microsoft.NET\Framework64\v2.0.50727\sos.dll]
dbghelp: image 6.11.0001.404, API 6.1.6, built Thu Feb 26 02:10:27 2009
[path: C:\Program Files\Debugging Tools for Windows (x64)\dbghelp.dll]
ext: image 6.11.0001.404, API 1.0.0, built Thu Feb 26 02:10:26 2009
[path: C:\Program Files\Debugging Tools for Windows (x64)\winext\ext.dll]
exts: image 6.11.0001.404, API 1.0.0, built Thu Feb 26 02:10:17 2009
[path: C:\Program Files\Debugging Tools for Windows (x64)\WINXP\exts.dll]
uext: image 6.11.0001.404, API 1.0.0, built Thu Feb 26 02:10:20 2009
[path: C:\Program Files\Debugging Tools for Windows (x64)\winext\uext.dll]
ntsdexts: image 6.1.7015.0, API 1.0.0, built Thu Feb 26 02:09:22 2009
[path: C:\Program Files\Debugging Tools for Windows (x64)\WINXP\ntsdexts.dll]

0:000> .unload C:\Windows\Microsoft.NET\Framework64\v2.0.50727\sos
Unloading C:\Windows\Microsoft.NET\Framework64\v2.0.50727\sos extension DLL

```

and load the customer version:

```

0:000> .load \MyData\sos.dll

0:000> .chain
Extension DLL chain:
\MyDatasas.dll: image 2.0.50727.5444, API 1.0.0, built Sun Feb 06 21:14:12 2011
[path: \MyData\sos.dll]
dbghelp: image 6.11.0001.404, API 6.1.6, built Thu Feb 26 02:10:27 2009
[path: C:\Program Files\Debugging Tools for Windows (x64)\dbghelp.dll]
ext: image 6.11.0001.404, API 1.0.0, built Thu Feb 26 02:10:26 2009
[path: C:\Program Files\Debugging Tools for Windows (x64)\winext\ext.dll]
exts: image 6.11.0001.404, API 1.0.0, built Thu Feb 26 02:10:17 2009
[path: C:\Program Files\Debugging Tools for Windows (x64)\WINXP\exts.dll]

```

```
uext: image 6.11.0001.404, API 1.0.0, built Thu Feb 26 02:10:20 2009
[path: C:\Program Files\Debugging Tools for Windows (x64)\winext\uext.dll]
ntsdexts: image 6.1.7015.0, API 1.0.0, built Thu Feb 26 02:09:22 2009
[path: C:\Program Files\Debugging Tools for Windows (x64)\WINXP\ntsdexts.dll]

0:000> .cordll -ve -u -l
CLR DLL status: No load attempts
```

Then we do a load attempt:

```
0:000> !CLRStack
CLRDLL: C:\Windows\Microsoft.NET\Framework64\v2.0.50727\mscordacwks.dll:2.0.50727.3619 f:0
doesn't match desired version 2.0.50727.5444 f:0
CLRDLL: Unable to find mscordacwks_AMD64_AMD64_2.0.50727.5444.dll by mscorewks search
CLRDLL: Unable to find 'mscordacwks_AMD64_AMD64_2.0.50727.5444.dll' on the path
CLRDLL: Unable to get version info for
'c:\mss\mscorwks.dll\4D4F0A6299d000\mscordacwks_AMD64_AMD64_2.0.50727.5444.dll', Win32 error 0n87
CLRDLL: ERROR: Unable to Load DLL mscordacwks_AMD64_AMD64_2.0.50727.5444.dll, Win32 error 0n87
Failed to Load data access DLL, 0x80004005
Verify that 1) you have a recent build of the debugger (6.2.14 or newer)
2) the file mscordacwks.dll that matches your version of mscorewks.dll is in the version directory
3) or, if you are debugging a dump file, verify that the file mscordacwks_<arch>_<arch>_<version>.dll is
on your symbol path.
4) you are debugging on the same architecture as the dump file. For example, an IA64 dump file must be
debugged on an IA64 machine.
```

You can also run the debugger command .cordll to control the debugger's load of mscordacwks.dll. .cordll -ve -u -l will do a verbose reload. If that succeeds, the SOS command should work on retry.

If you are debugging a minidump, you need to make sure that your executable path is pointing to mscorewks.dll as well.

Following instructions we rename *mscordacwks.dll* to *mscordacwks\_AMD64\_AMD64\_2.0.50727.5444.dll* and retry:

```
0:000> .cordll -ve -u -l
CLR DLL status: No load attempts

0:000> !CLRStack
CLRDLL: C:\Windows\Microsoft.NET\Framework64\v2.0.50727\mscordacwks.dll:2.0.50727.3619 f:0
doesn't match desired version 2.0.50727.5444 f:0
CLRDLL: Loaded DLL \MyData\mscordacwks_AMD64_AMD64_2.0.50727.5444.dll
OS Thread Id: 0x16e8 (0)
Child-SP RetAddr Call Site
0000000002fe570 000007feeaf8e378 System.Windows.Forms.Application+ComponentManager.System.Windows.Forms.UnsafeNativeMethods.IMsoComponentManager.FPushMessageLoop(Int32, Int32, Int32)
0000000002fe7c0 000007feeaf8dde5
System.Windows.Forms.Application+ThreadContext.RunMessageLoopInner(Int32,
System.Windows.Forms.ApplicationContext)
0000000002fe910 000007ff002364b6 System.Windows.Forms.Application+ThreadContext.RunMessageLoop(Int32,
System.Windows.Forms.ApplicationContext)
0000000002fe970 000007fee6414c2 MyApplication.Main(System.String[])
```

```
0:000> !pe
Exception object: 0000000034a13f8
Exception type: System.IO.FileNotFoundException
Message: Could not load file or assembly 'System.Windows.Forms.XmlSerializers, Version=2.0.0.0,
Culture=neutral, PublicKeyToken= ...' or one of its dependencies. The system cannot find the file
specified.
InnerException: System.IO.FileNotFoundException, use !PrintException 0000000034a1b28 to see more
StackTrace (generated):
SP IP Function
0000000002FD0A0 0000000000000001
mscorlib_ni!System.Reflection.Assembly._nLoad(System.Reflection.AssemblyName, System.String,
System.Security.Policy.Evidence, System.Reflection.Assembly, System.Threading.StackCrawlMark ByRef,
Boolean, Boolean)+0x2
0000000002FD0A0 000007FEED7ABF61
mscorlib_ni!System.Reflection.Assembly.InternalLoad(System.Reflection.AssemblyName,
System.Security.Policy.Evidence, System.Threading.StackCrawlMark ByRef, Boolean)+0x1a1
0000000002FD130 000007FEED7E4804
mscorlib_ni!System.Reflection.Assembly.Load(System.Reflection.AssemblyName)+0x24
0000000002FD170 000007FEE7855C0A
System_Xml_ni!System.Xml.Serialization.TempAssembly.LoadGeneratedAssembly(System.Type, System.String,
System.Xml.Serialization.XmlSerializerImplementation ByRef)+0x11a

StackTraceString: <none>
HRESULT: 80070002

0:000> !PrintException 0000000034a1b28
Exception object: 0000000034a1b28
Exception type: System.IO.FileNotFoundException
Message: Could not load file or assembly 'System.Windows.Forms.XmlSerializers, Version=2.0.0.0,
Culture=neutral, PublicKeyToken= ...' or one of its dependencies. The system cannot find the file
specified.
InnerException: <none>
StackTrace (generated):
<none>
StackTraceString: <none>
HRESULT: 80070002
```

## Virtualized Process

### WOW64

Sometimes we get a process dump from x64 Windows, and when we load it into WinDbg, we get the output telling us that an exception or a breakpoint comes from *wow64.dll*. For example:

```
Loading Dump File [X:\ppid2088.dmp]
User Mini Dump File with Full Memory: Only application data is available

Comment: 'Userdump generated complete user-mode minidump with Exception Monitor function on SERVER01'
Symbol search path is: srv*c:\mss*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows Server 2003 Version 3790 (Service Pack 2) MP (4 procs) Free x64
Product: Server, suite: TerminalServer
Debug session time: Tue Sep 4 13:36:14.000 2007 (GMT+2)
System Uptime: 6 days 3:32:26.081
Process Uptime: 0 days 0:01:54.000
WARNING: tsappcmp overlaps ws2_32
WARNING: msvcp60 overlaps oleacc
WARNING: tapi32 overlaps rasapi32
WARNING: rtutils overlaps rasman
WARNING: dnsapi overlaps rasapi32
WARNING: wldap32 overlaps dnsapi
WARNING: ntshrui overlaps userenv
WARNING: wtsapi32 overlaps dnsapi
WARNING: winsta overlaps setupapi
WARNING: activeds overlaps rtutils
WARNING: activeds overlaps rasman
WARNING: adsldpc overlaps activeds
WARNING: drprov overlaps apphelp
WARNING: netui1 overlaps netui0
WARNING: davclnt overlaps apphelp
...
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
(2088.2fe4): Unknown exception - code 000006d9 (first/second chance not available)
wow64!Wow64NotifyDebugger+0x9:
0000000`6b006369 b001          mov     al,1
```

The analysis shows that the run-time exception (**Software Exception**, page 875) was raised, but the stack trace shows only WOW64 CPU simulation code in all process threads:

```
0:000> !analyze -v
*****
*                                         *
*             Exception Analysis          *
*                                         *
*****
```

FAULTING\_IP:  
kernel32!RaiseException+53  
0000000`7d4e2366 5e                          pop        rsi

EXCEPTION\_RECORD: ffffffffffffffff -- (.exr 0xfffffffffffff)

ExceptionAddress: 000000007d4e2366 (kernel32!RaiseException+0x0000000000000053)  
ExceptionCode: 000006d9  
ExceptionFlags: 00000001  
NumberParameters: 0

DEFAULT\_BUCKET\_ID: STACK\_CORRUPTION

PROCESS\_NAME: App.exe

ERROR\_CODE: (NTSTATUS) 0x6d9 - There are no more endpoints available from the endpoint mapper.

NTGLOBALFLAG: 0

APPLICATION\_VERIFIER\_FLAGS: 0

LAST\_CONTROL\_TRANSFER: from 00000006b0064f2 to 00000006b006369

FOLLOWUP\_IP:  
wow64!Wow64NotifyDebugger+9  
00000000 6b006369 b001                      mov        al,1

SYMBOL\_STACK\_INDEX: 0

SYMBOL\_NAME: wow64!Wow64NotifyDebugger+9

FOLLOWUP\_NAME: MachineOwner

MODULE\_NAME: wow64

IMAGE\_NAME: wow64.dll

DEBUG\_FLR\_IMAGE\_TIMESTAMP: 45d6943d

FAULTING\_THREAD: 000000000002fe4

PRIMARY\_PROBLEM\_CLASS: STACK\_CORRUPTION

BUGCHECK\_STR: APPLICATION\_FAULT\_STACK\_CORRUPTION

STACK\_COMMAND: ~0s; .ecxr ; dt ntdll!LdrpLastDllInitializer BaseDllName ; dt ntdll!LdrpFailureData ; kb

FAILURE\_BUCKET\_ID: X64\_APPLICATION\_FAULT\_STACK\_CORRUPTION\_wow64!Wow64NotifyDebugger+9

BUCKET\_ID: X64\_APPLICATION\_FAULT\_STACK\_CORRUPTION\_wow64!Wow64NotifyDebugger+9

```
Followup: MachineOwner
-----
```

```
0:000> ~*k
```

Child-SP	RetAddr	Call Site
00000000`0016e190	00000000`6b0064f2	wow64!Wow64NotifyDebugger+0x9
00000000`0016e1c0	00000000`6b006866	wow64!Wow64KiRaiseException+0x172
00000000`0016e530	00000000`78b83c7d	wow64!Wow64SystemServiceEx+0xd6
00000000`0016edf0	00000000`6b006a5a	wow64cpu!ServiceNoTurbo+0x28
00000000`0016ee80	00000000`6b005e0d	wow64!RunCpuSimulation+0xa
00000000`0016eef0	00000000`77ed8030	wow64!Wow64LdrpInitialize+0x2ed
00000000`0016f3f0	00000000`77ed582f	ntdll!LdrpInitializeProcess+0x1538
00000000`0016f6f0	00000000`77ef30a5	ntdll!LdrpInitialize+0x18f
00000000`0016f7d0	00000000`7d4d1510	ntdll!KiUserApcDispatcher+0x15
00000000`0016fcc8	00000000`00000000	kernel32!BaseProcessStartThunk
00000000`0016fcf0	00000000`00000000	0x0
00000000`0016fcf8	00000000`00000000	0x0
00000000`0016fd00	00010007`00000000	0x0
00000000`0016fd08	00000000`00000000	0x10007`00000000
00000000`0016fd10	00000000`00000000	0x0
00000000`0016fd18	00000000`00000000	0x0

Child-SP	RetAddr	Call Site
00000000`0200f0d8	00000000`6b006a5a	wow64cpu!WaitForMultipleObjects32+0x3a
00000000`0200f180	00000000`6b005e0d	wow64!RunCpuSimulation+0xa
00000000`0200f1b0	00000000`77f109f0	wow64!Wow64LdrpInitialize+0x2ed
00000000`0200f6f0	00000000`77ef30a5	ntdll!LdrpInitialize+0x2aa
00000000`0200f7d0	00000000`7d4d1504	ntdll!KiUserApcDispatcher+0x15
00000000`0200fcc8	00000000`00000000	kernel32!BaseThreadStartThunk
00000000`0200fcf0	00000000`00000000	0x0
00000000`0200fcf8	00000000`00000000	0x0
00000000`0200fd00	0001002f`00000000	0x0
00000000`0200fd08	00000000`00000000	0x1002f`00000000
00000000`0200fd10	00000000`00000000	0x0
00000000`0200fd18	00000000`00000000	0x0
00000000`0200fd20	00000000`00000000	0x0
00000000`0200fd28	00000000`00000000	0x0
00000000`0200fd30	00000000`00000000	0x0
00000000`0200fd38	00000000`00000000	0x0

2 Id: 2088.1160 Suspend: 1 Teb: 00000000`7efd5000 Unfrozen

Child-SP	RetAddr	Call Site
00000000`0272e7c8	00000000`6b29c464	wow64win!ZwUserGetMessage+0xa
00000000`0272e7d0	00000000`6b006866	wow64win!whNtUserGetMessage+0x34
00000000`0272e830	00000000`78b83c7d	wow64!Wow64SystemServiceEx+0xd6
00000000`0272f0f0	00000000`6b006a5a	wow64cpu!ServiceNoTurbo+0x28
00000000`0272f180	00000000`6b005e0d	wow64!RunCpuSimulation+0xa
00000000`0272f1b0	00000000`77f109f0	wow64!Wow64LdrpInitialize+0x2ed
00000000`0272f6f0	00000000`77ef30a5	ntdll!LdrpInitialize+0x2aa
00000000`0272f7d0	00000000`7d4d1504	ntdll!KiUserApcDispatcher+0x15
00000000`0272fcc8	00000000`00000000	kernel32!BaseThreadStartThunk
00000000`0272fc00	00000000`00000000	0x0
00000000`0272fc08	00000000`00000000	0x0
00000000`0272fce0	00000000`00000000	0x0
00000000`0272fce8	00000000`00000000	0x0
00000000`0272fcf0	00000000`00000000	0x0
00000000`0272fcf8	00000000`00000000	0x0
00000000`0272fd00	00010003`00000000	0x0
00000000`0272fd08	00000000`00000000	0x10003`00000000
00000000`0272fd10	00000000`00000000	0x0
00000000`0272fd18	00000000`00000000	0x0
00000000`0272fd20	00000000`00000000	0x0

3 Id: 2088.2d04 Suspend: 1 Teb: 00000000`7efad000 Unfrozen

Child-SP	RetAddr	Call Site
00000000`0289f108	00000000`78b84191	wow64cpu!CpuPsyscallStub+0x9
00000000`0289f110	00000000`6b006a5a	wow64cpu!Thunk2ArgNSpNsPReloadState+0x21
00000000`0289f180	00000000`6b005e0d	wow64!RunCpuSimulation+0xa
00000000`0289f1b0	00000000`77f109f0	wow64!Wow64LdrpInitialize+0x2ed
00000000`0289f6f0	00000000`77ef30a5	ntdll!LdrpInitialize+0x2aa
00000000`0289f7d0	00000000`7d4d1504	ntdll!KiUserApcDispatcher+0x15
00000000`0289fcc8	00000000`00000000	kernel32!BaseThreadStartThunk
00000000`0289fc00	00000000`00000000	0x0
00000000`0289fc08	00000000`00000000	0x0
00000000`0289fce0	00000000`00000000	0x0
00000000`0289fce8	00000000`00000000	0x0
00000000`0289fcf0	00000000`00000000	0x0
00000000`0289fcf8	00000000`00000000	0x0
00000000`0289fd00	0001002f`00000000	0x0
00000000`0289fd08	00000000`00000000	0x1002f`00000000
00000000`0289fd10	00000000`00000000	0x0
00000000`0289fd18	00000000`00000000	0x0
00000000`0289fd20	00000000`00000000	0x0
00000000`0289fd28	00000000`00000000	0x0
00000000`0289fd30	00000000`00000000	0x0

4 Id: 2088.15c4 Suspend: 1 Teb: 00000000`7efa4000 Unfrozen

Child-SP	RetAddr	Call Site
00000000`02def0a8	00000000`6b006a5a	wow64cpu!RemoveIoCompletionFault+0x41
00000000`02def180	00000000`6b005e0d	wow64!RunCpuSimulation+0xa
00000000`02def1b0	00000000`77f109f0	wow64!Wow64LdrpInitialize+0x2ed
00000000`02def6f0	00000000`77ef30a5	ntdll!LdrpInitialize+0x2aa
00000000`02def7d0	00000000`7d4d1504	ntdll!KiUserApcDispatcher+0x15
00000000`02defcc8	00000000`00000000	kernel32!BaseThreadStartThunk
00000000`02defc00	00000000`00000000	0x0
00000000`02defc08	00000000`00000000	0x0

```
00000000`02defce0 00000000`00000000 0x0
00000000`02defce8 00000000`00000000 0x0
00000000`02defcf0 00000000`00000000 0x0
00000000`02defcf8 00000000`00000000 0x0
00000000`02defd00 0001002f`00000000 0x0
00000000`02defd08 00000000`00000000 0x1002f`00000000
00000000`02defd10 00000000`00000000 0x0
00000000`02defd18 00000000`00000000 0x0
00000000`02defd20 00000000`00000000 0x0
00000000`02defd28 00000000`00000000 0x0
00000000`02defd30 00000000`00000000 0x0
00000000`02defd38 00000000`00000000 0x0
```

This is a clear indication that the process was, in fact, 32-bit, but the dump is 64-bit. This situation is depicted in the article about memory dumps, debuggers, and virtualization<sup>219</sup>, and we need a debugger plug-in to understand virtualized CPU architecture.

This crash dump pattern can be called **Virtualized Process**. In our case, we need to load `wow64exts.dll` WinDbg extension and set the target processor mode to x86 by using `.effmach` command

```
0:000> .load wow64exts
0:000> .effmach x86
Effective machine: x86 compatible (x86)
```

Then analysis gives us more meaningful results:

```
0:000:x86> !analyze -v
*****
*          *
*      Exception Analysis      *
*          *
*****
FAULTING_IP:
kernel32!RaiseException+53
00000000`7d4e2366 5e          pop     esi

EXCEPTION_RECORD: ffffffffffffff -- (.exr 0xffffffffffff)
ExceptionAddress: 000000007d4e2366 (kernel32!RaiseException+0x0000000000000053)
    ExceptionCode: 000006d9
    ExceptionFlags: 00000001
NumberParameters: 0

BUGCHECK_STR: 6d9

DEFAULT_BUCKET_ID: APPLICATION_FAULT
```

---

<sup>219</sup> Dumps, Debuggers and Virtualization, Memory Dump Analysis Anthology, Volume 1, page 516

PROCESS\_NAME: App.exe

ERROR\_CODE: (NTSTATUS) 0x6d9 - There are no more endpoints available from the endpoint mapper.

NTGLOBALFLAG: 0

APPLICATION\_VERIFIER\_FLAGS: 0

LAST\_CONTROL\_TRANSFER: from 000000007da4a631 to 000000007d4e2366

STACK\_TEXT:

0012d98c 7da4a631 kernel32!RaiseException+0x53  
0012d9a4 7da4a5f7 rpct4!RcpRaiseException+0x24  
0012d9b4 7dac0140 rpct4!NdrGetBuffer+0x46  
0012ddaa 5f2a2fba rpct4!NdrClientCall12+0x197  
0012ddbc 5f29c6a6 hnetcfg!FwOpenDynamicFwPort+0x1d  
0012de68 7db4291f hnetcfg!IcfOpenDynamicFwPort+0x6a  
0012df00 71c043db mssock!WSPBind+0x2e3  
WARNING: Frame IP not in any known module. Following frames may be wrong.  
0012df24 76ed91c8 ws2\_32+0x43db  
0012df6c 76ed9128 rasapi32+0x491c8  
0012df98 76ed997c rasapi32+0x49128  
0012dfc0 76ed8ac2 rasapi32+0x4997c  
0012dfd4 76ed89cd rasapi32+0x48ac2  
0012dff0 76ed82e5 rasapi32+0x489cd  
0012e010 76ed827f rasapi32+0x482e5  
0012e044 76ed8bf0 rasapi32+0x4827f  
0012e0c8 76ed844d rasapi32+0x48bf0  
0012e170 76ed74b5 rasapi32+0x4844d  
0012e200 76ed544f rasapi32+0x474b5  
0012e22c 76ed944d rasapi32+0x4544f  
0012e24c 76ed93a4 rasapi32+0x4944d  
0012e298 76ed505f rasapi32+0x493a4  
0012e2bc 7db442bf rasapi32+0x4505f  
0012e2ec 7db4418b mssock!SaBlob\_Query+0x2d  
0012e330 7db4407c mssock!Rnr\_DoDnsLookup+0xf0  
0012e5c8 71c06dc0 mssock!Dns\_NSPLookupServiceNext+0x24b  
0012e5e0 71c06da0 ws2\_32+0x6dc0  
0012e5fc 71c06d6a ws2\_32+0x6da0  
0012e628 71c06d08 ws2\_32+0x6d6a  
0012e648 71c08282 ws2\_32+0x6d08  
0012ef00 71c07f68 ws2\_32+0x8282  
0012ef34 71c08433 ws2\_32+0x7f68  
0012efa0 71c03236 ws2\_32+0x8433  
0012f094 71c03340 ws2\_32+0x3236  
0012f0bc 7dab22fb ws2\_32+0x3340  
0012f11c 7dab3a0e rpct4!IP\_ADDRESS\_RESOLVER::NextAddress+0x13e  
0012f238 7dab3c11 rpct4!TCPOrHTTP\_Open+0xdb  
0012f270 7da44c85 rpct4!TCP\_Open+0x55  
0012f2b8 7da44b53 rpct4!OSF\_CCONNECTION::TransOpen+0x5e  
0012f31c 7da447d7 rpct4!OSF\_CCONNECTION::OpenConnectionAndBind+0xbe  
0012f360 7da44720 rpct4!OSF\_CCALL::BindToServer+0xfa  
0012f378 7da3a9df rpct4!OSF\_BINDING\_HANDLE::InitCCallWithAssociation+0x63  
0012f3f4 7da3a8dd rpct4!OSF\_BINDING\_HANDLE::AllocateCCall+0x49d  
0012f428 7da37a1c rpct4!OSF\_BINDING\_HANDLE::NegotiateTransferSyntax+0x2e

```
0012f440 7da3642c rpcrt4!I_RpcGetBufferWithObject+0x5b
0012f450 7da37bff rpcrt4!I_RpcGetBuffer+0xf
0012f460 7dac0140 rpcrt4!NdrGetBuffer+0x2e
0012f84c 766f41f1 rpcrt4!NdrClientCall12+0x197
0012f864 766f40b8 ntdsapi!_IDL_DRSBind+0x1c
0012f930 7d8ecaa2 ntdsapi!DsBindWithSpnExW+0x223
0012f9b0 7d8ed028 secur32!SecpTranslateName+0x1f3
0012f9d0 00434aa0 secur32!TranslateNameW+0x2d
0012fab4 00419a7f App+0x34aa0
0012fb0c 0041a61b App+0x19a7f
0012fb0c 0045a293 App+0x1a61b
0012fb0c 0043682f App+0x5a293
0012fbcc 004188f3 App+0x3682f
0043682f 00000000 App+0x188f3
```

STACK\_COMMAND: kb

FOLLOWUP\_IP:

```
hnetcfg!FwOpenDynamicFwPort+1d
00000000`5f2a2fba 83c40c          add     esp,0Ch
```

SYMBOL\_STACK\_INDEX: 4

SYMBOL\_NAME: hnetcfg!FwOpenDynamicFwPort+1d

FOLLOWUP\_NAME: MachineOwner

MODULE\_NAME: hnetcfg

IMAGE\_NAME: hnetcfg.dll

DEBUG\_FLR\_IMAGE\_TIMESTAMP: 45d6cc2a

FAULTING\_THREAD: 000000000002fe4

FAILURE\_BUCKET\_ID: X64\_6d9\_hnetcfg!FwOpenDynamicFwPort+1d

BUCKET\_ID: X64\_6d9\_hnetcfg!FwOpenDynamicFwPort+1d

Followup: MachineOwner

-----

0:000:x86> ~\*k

```
. 0 Id: 2088.2fe4 Suspend: 1 Teb: 00000000`7efdb000 Unfrozen
ChildEBP      RetAddr
0012d98c 7da4a631 kernel32!RaiseException+0x53
0012d9a4 7da4a5f7 rpcrt4!RpcpRaiseException+0x24
0012d9b4 7dac0140 rpcrt4!NdrGetBuffer+0x46
0012dd0 5f2a2fba rpcrt4!NdrClientCall12+0x197
0012ddbc 5f29c6a6 hnetcfg!FwOpenDynamicFwPort+0x1d
0012de68 7db4291f hnetcfg!IcfOpenDynamicFwPort+0x6a
```

```

0012df00 71c043db mssock!WSPBind+0x2e3
WARNING: Frame IP not in any known module. Following frames may be wrong.
0012df24 76ed91c8 ws2_32+0x43db
0012df6c 76ed9128 rasapi32+0x491c8
0012df98 76ed997c rasapi32+0x49128
0012dfc0 76ed8ac2 rasapi32+0x4997c
0012dfd4 76ed89cd rasapi32+0x48ac2
0012dff0 76ed82e5 rasapi32+0x489cd
0012e010 76ed827f rasapi32+0x482e5
0012e044 76ed8bf0 rasapi32+0x4827f
0012e0c8 76ed844d rasapi32+0x48bf0

 1 Id: 2088.280c Suspend: 1 Teb: 00000000`7efd8000 Unfrozen
ChildEBP      RetAddr
01fcfea4 7d63f501 nt!NtWaitForMultipleObjects+0x15
01fcff48 7d63f988 nt!EtwpWaitForMultipleObjectsEx+0xf7
01fcffb8 7d4dfe21 nt!EtwpEventPump+0x27f
01fcffec 00000000 kernel32!BaseThreadStart+0x34

 2 Id: 2088.1160 Suspend: 1 Teb: 00000000`7efd5000 Unfrozen
ChildEBP      RetAddr
026eff50 0042f13b user32!NtUserGetMessage+0x15
WARNING: Stack unwind information not available. Following frames may be wrong.
026effb8 7d4dfe21 App+0x2f13b
026effec 00000000 kernel32!BaseThreadStart+0x34

 3 Id: 2088.2d04 Suspend: 1 Teb: 00000000`7efad000 Unfrozen
ChildEBP      RetAddr
0285ffa0 7d634d69 nt!ZwDelayExecution+0x15
0285ffb8 7d4dfe21 nt!RtlpTimerThread+0x47
0285ffec 00000000 kernel32!BaseThreadStart+0x34

 4 Id: 2088.15c4 Suspend: 1 Teb: 00000000`7efa4000 Unfrozen
ChildEBP      RetAddr
02daff80 7db4b6c6 nt!NtRemoveIoCompletion+0x15
02daffb8 7d4dfe21 mssock!SockAsyncThread+0x69
02daffec 00000000 kernel32!BaseThreadStart+0x34

```

## Comments

We can also use this command for switching between modes: **!wow64exts.sw.**

Use the right tool to capture 32-bit process memory dumps on x64 Windows<sup>220</sup>.

---

<sup>220</sup> <https://blogs.msdn.microsoft.com/tess/2010/09/29/capturing-memory-dumps-for-32-bit-processes-on-an-x64-machine/>

## Virtualized System

Sometimes we get rare or hardware-related bugchecks, and these might have been influenced by hardware virtualization (that is also software and could have its own defects). Therefore, it is beneficial to recognize when a system is running under a VM:

- Memory dumps from Xen-virtualized Windows<sup>221</sup>
- Memory dumps from VMware images<sup>222</sup>
- Memory dumps from Hyper-V guests<sup>223</sup>

For example, we get the following bugcheck and stack trace for the first processor:

```
0: kd> !analyze -v
[...]
CLOCK_WATCHDOG_TIMEOUT (101)
An expected clock interrupt was not received on a secondary processor in an MP system within the
allocated interval. This indicates that the specified processor is hung and not processing interrupts.
Arguments:
Arg1: 00000060, Clock interrupt time out interval in nominal clock ticks.
Arg2: 00000000, 0.
Arg3: 805d8120, The PRCB address of the hung processor.
Arg4: 00000001, 0.

CURRENT_IRQL:  1c

STACK_TEXT:
8d283694 816df8a5 nt!KeBugCheckEx+0x1e
8d2836c8 816e163d nt!KeUpdateRunTime+0xd5
8d2836c8 84617008 nt!KeUpdateSystemTime+0xed
WARNING: Frame IP not in any known module. Following frames may be wrong.
8d283748 816f46a1 0x84617008
8d283758 816fa6aa nt!HvlpInitiateHypercall+0x21
8d283784 8166aca5 nt!HvlNotifyLongSpinWait+0x2b
8d2837a0 816cce7e nt!KeFlushSingleTb+0xc4
8d283808 81681db4 nt!MmAccessFault+0xc1d
8d283808 816dd033 nt!KiTrap0E+0xdc
8d28389c 8168ed58 nt!memcpy+0x33
8d283954 816712bf nt!MmCopyToCachedPage+0x1193
8d2839ec 81663053 nt!CcMapAndCopy+0x210
8d283a74 8c688218 nt!CcFastCopyWrite+0x283
8d283b98 8c40badc Ntfs!NtfsCopyWriteA+0x23e
8d283bcc 8c40bcab fltmgr!FltpPerformFastIoCall+0x22e
```

<sup>221</sup> Memory Dumps from Xen-virtualized Windows, Memory Dump Analysis Anthology, Volume 2, page 401

<sup>222</sup> Memory Dumps from Virtual Images, Memory Dump Analysis Anthology, Volume 1, page 219

<sup>223</sup> Memory Dumps from Hyper-Virtualized Windows, Memory Dump Analysis Anthology, Volume 4, page 354

```
8d283bf8 8c41dc30 fltmgr!FltpPassThroughFastIo+0x7d
8d283c3c 818471cd fltmgr!FltpFastIoWrite+0x146
8d283d38 8167ec7a nt!NtWriteFile+0x34c
8d283d38 77115e74 nt!KiFastCallEntry+0x12a
01cf80 00000000 0x77115e74
```

The thread was servicing a page fault. The gap between *KeUpdateSystemTime* and *HvlpInitiateHypercall* is normal and has consistent code if we look closer:

```
0: kd> .asm no_code_bytes
Assembly options: no_code_bytes

0: kd> uf HvlpInitiateHypercall
nt!HvlpInitiateHypercall:
816f4680 push    edi
816f4681 push    esi
816f4682 push    ebx
816f4683 mov     eax,dword ptr [esp+10h]
816f4687 mov     edx,dword ptr [esp+14h]
816f468b mov     ecx,dword ptr [esp+18h]
816f468f mov     ebx,dword ptr [esp+1Ch]
816f4693 mov     esi,dword ptr [esp+20h]
816f4697 mov     edi,dword ptr [esp+24h]
816f469b call    dword ptr [nt!HvlpHypercallCodeVa (8176bb8c)]
816f46a1 pop    ebx
816f46a2 pop    esi
816f46a3 pop    edi
816f46a4 ret     18h

0: kd> dp 8176bb8c 11
8176bb8c 84617000

0: kd> uf 84617000
84617000 or      eax,80000000h
84617005 vmcall
84617008 ret
```

We have the address of RET instruction (84617008) on the stack trace:

```
0: kd> kv
ChildEBP RetAddr  Args to Child
8d283694 816df8a5 00000101 00000060 00000000 nt!KeBugCheckEx+0x1e
8d2836c8 816e163d 84e1521b 000000d1 8d283784 nt!KeUpdateRunTime+0xd5
8d2836c8 84617008 84e1521b 000000d1 8d283784 nt!KeUpdateSystemTime+0xed (TrapFrame @ 8d2836d8)
WARNING: Frame IP not in any known module. Following frames may be wrong.
8d283748 816f46a1 84e6c900 22728000 8172e28c 0x84617008
8d283758 816fa6aa 00010008 00000000 22728000 nt!HvlpInitiateHypercall+0x21
8d283784 8166aca5 22728000 00000000 00000000 nt!HvlNotifyLongSpinWait+0x2b
[...]
```

The second processor is busy too:

```
0: kd> !running

System Processors 3 (affinity mask)
  Idle Processors 0

Prcbs  Current   Next
  0     8172c920  84e6c900      .....
  1     805d8120  85138030  85a50d78  .....

0: kd> !thread 85138030
THREAD 85138030  Cid 0564.11c8  Teb: 7fff9f000 Win32Thread: 00000000 RUNNING on processor 1
IRP List:
  85ab5d00: (0006,01fc) Flags: 00000884  Mdl: 00000000
  85445ab8: (0006,0094) Flags: 00060000  Mdl: 00000000
Not impersonating
DeviceMap          98a7d558
Owning Process    84f0d938      Image:       Application.exe
Attached Process  N/A          Image:       N/A
Wait Start TickCount 695643      Ticks: 224 (0:00:00:03.500)
Context Switch Count 20
UserTime           00:00:00.000
KernelTime         00:00:00.000
Win32 Start Address 0x705e2679
Stack Init a1d13000 Current a1d10a70 Base a1d13000 Limit a1d10000 Call 0
Priority 9 BasePriority 8 PriorityDecrement 0 IoPriority 2 PagePriority 5
ChildEBP RetAddr  Args to Child
00000000 00000000 00000000 00000000 00000000 0x0
```

Because we have (*Limit*, *Current*, *Base*) triple for our thread, we check its **Execution Residue** (page 371) on the kernel raw stack. We find traces of a hypercall too:

```
0: kd> dds a1d10000 a1d13000
[...]
a1d1215c 816fa6da nt!HvlEndSystemInterrupt+0x20
a1d12160 40000070
a1d12164 00000000
a1d12168 81608838 hal!HalEndSystemInterrupt+0x7a
a1d1216c 805d8000
a1d12170 a1d12180
a1d12174 81618cc9 hal!HalpIpiHandler+0x189
a1d12178 84f4841b
a1d1217c 000000e1
a1d12180 a1d1222c
a1d12184 84617008
a1d12188 badb0d00
a1d1218c 00000000
a1d12190 81665699 nt!RtlWalkFrameChain+0x58
a1d12194 816659c4 nt!RtlWalkFrameChain+0x377
a1d12198 11c4649e
a1d1219c 00000002
a1d121a0 00000003
a1d121a4 85478444
```

```
a1d121a8 0000001f
a1d121ac 00000000
a1d121b0 00000000
a1d121b4 816f46a1 nt!HvLpInitiateHypercall+0x21
a1d121b8 8172c800 nt!KiInitialPCR
a1d121bc 85138030
a1d121c0 85a72ac0
a1d121c4 8171b437 nt!HvlSwitchVirtualAddressSpace+0x28
a1d121c8 00010001
a1d121cc 00000000
a1d121d0 85a72ac0
a1d121d4 85138030
a1d121d8 8172c802 nt!KiInitialPCR+0x2
a1d121dc 00000000
a1d121e0 85138030
a1d121e4 816fa6da nt!HvlEndSystemInterrupt+0x20
a1d121e8 40000070
a1d121ec 00000000
a1d121f0 81608838 hal!HalEndSystemInterrupt+0x7a
a1d121f4 816f46a1 nt!HvLpInitiateHypercall+0x21
a1d121f8 805d8000
a1d121fc 85138030
a1d12200 805dc1e0
a1d12204 8171b437 nt!HvlSwitchVirtualAddressSpace+0x28
a1d12208 00010001
a1d1220c 00000000
a1d12210 00000000
a1d12214 81608468 hal!HalpDispatchSoftwareInterrupt+0x5e
a1d12218 00000000
a1d1221c 00000000
a1d12220 00000206
a1d12224 a1d12240
a1d12228 81608668 hal!HalpCheckForSoftwareInterrupt+0x64
a1d1222c 00000002
a1d12230 00000000
a1d12234 816086a8 hal!KfLowerIrql
a1d12238 00000000
a1d1223c 00000002
a1d12240 a1d12250
a1d12244 8160870c hal!KfLowerIrql+0x64
a1d12248 00000000
a1d1224c 00000000
a1d12250 a1d12294
a1d12254 816e035a nt!KiSwapThread+0x477
a1d12258 85138030
a1d1225c 851380b8
a1d12260 805d8120
a1d12264 0014d1f8
[ ... ]
```

Looking at the raw stack further we can even see that it was processing a page fault too and manually reconstruct its stack trace<sup>224</sup>:

```
[...]
a1d1074c 85aef510
a1d10750 a1d10768
a1d10754 81678976 nt!IoCallDriver+0x63
a1d10758 84c87d50
a1d1075c 85aef510
a1d10760 00000000
a1d10764 84c87d50
a1d10768 a1d10784
a1d1076c 8166d74e nt!IoPageRead+0x172
a1d10770 85138030
a1d10774 84a1352c
a1d10778 84a134f8
a1d1077c 84a13538
a1d10780 84c87d50
a1d10784 a1d10840
a1d10788 816abf07 nt!MiDispatchFault+0xd14
a1d1078c 00000043
a1d10790 85138030
a1d10794 84a13538
a1d10798 84a1350c
a1d1079c 84a1352c
a1d107a0 8174c800 nt!MmSystemCacheWs
a1d107a4 00000000
a1d107a8 85138030
a1d107ac a5397bf8
a1d107b0 85b01c48
a1d107b4 00000000
a1d107b8 00000000
a1d107bc a5397bf8
a1d107c0 84a1358c
a1d107c4 a1d10864
a1d107c8 00000000
a1d107cc 8463a590
a1d107d0 84a134f8
a1d107d4 c0518000
a1d107d8 00000000
a1d107dc 00000000
a1d107e0 00000028
a1d107e4 a1d107f4
a1d107e8 00000000
a1d107ec 00000038
a1d107f0 859f5640
a1d107f4 a4bfa390
a1d107f8 00000000
a1d107fc 00000000
a1d10800 00000000
```

---

<sup>224</sup> Manual Stack Trace Reconstruction, Memory Dump Analysis Anthology, Volume 1, page 157

```
a1d10804 a1d10938
a1d10808 818652bd nt!MmCreateSection+0x98f
a1d1080c 00000000
a1d10810 846652e8
a1d10814 00000000
a1d10818 00000000
a1d1081c 00000000
a1d10820 00000028
a1d10824 00000000
a1d10828 00000080
a1d1082c 0000000a
a1d10830 85ae1c98
a1d10834 85ae1c20
a1d10838 00000000
a1d1083c 00000000
a1d10840 a1d108b8
a1d10844 816cd325 nt!MmAccessFault+0x10c6
a1d10848 a3000000
a1d1084c a5397bf8
a1d10850 00000000
a1d10854 8174c800 nt!MmSystemCacheWs
a1d10858 00000000
a1d1085c 00000000
a1d10860 a5397bf8
a1d10864 00000000
[...]
```

```
0: kd> k L=a1d10750 a1d10750 a1d10750
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
a1d10750 81678976 0xa1d10750
a1d10768 8166d74e nt!IofCallDriver+0x63
a1d10784 816abf07 nt!IoPageRead+0x172
a1d10840 816cd325 nt!MiDispatchFault+0xd14
a1d108b8 816f0957 nt!MmAccessFault+0x10c6
a1d10924 8181c952 nt!MmCheckCachedPageState+0x801
a1d109b0 8c60f850 nt!CcCopyRead+0x435
a1d109dc 8c613c52 Ntfs!NtfsCachedRead+0x13b
a1d10abc 8c612b6f Ntfs!NtfsCommonRead+0x105a
a1d10b2c 81678976 Ntfs!NtfsFsdRead+0x273
a1d10b44 8c40cba7 nt!IofCallDriver+0x63
a1d10b68 8c40d7c7 fltmgr!FltpLegacyProcessingAfterPreCallbacksCompleted+0x251
a1d10ba0 8c40dbe7 fltmgr!FltPerformSynchronousIo+0xb9
a1d10c10 9ca680e5 fltmgr!FltReadFile+0x2ed
[...]
```

We also see that the absence of a trap means that the cache processing NTFS code reuses page fault handling code.

## W

## Wait Chain

## C++11, Condition Variable

This is C++ condition variable<sup>225</sup> **Wait Chain** (page 1092) pattern variant. When we have a waiting **Blocked Thread** (page 82) stack trace that shows conditional variable implementation functions we are interested in the owner thread:

```
0:012> kL
# ChildEBP RetAddr
00 03a6f684 748d8ee9 ntdll!NtWaitForSingleObject+0xc
01 03a6f6f8 5f5fcba5 KERNELBASE!WaitForSingleObjectEx+0x99
02 03a6f70c 5f5fb506 msvcr120!Concurrency::details::ExternalContextBase::Block+0x37
03 03a6f778 639cea79 msvcr120!Concurrency::details::_Condition_variable::wait+0xab
04 03a6f7ac 639ceb58 msvcp120!do_wait+0x42
05 03a6f7c0 5c8c5a43 msvcp120!_Cnd_wait+0x10
WARNING: Stack unwind information not available. Following frames may be wrong.
06 03a6f7d0 5c8c4ee6 AppA!foo+0x48883
07 03a6f804 5c8c4bde AppA!foo+0x47d26
08 03a6f834 5c8c4b9c AppA!foo+0x47a1e
09 03a6f848 5c8c4a27 AppA!foo+0x479dc
0a 03a6f854 00dcc4e9 AppA!foo+0x47867
0b 03a6f86c 75823744 AppA+0x1c4e9
0c 03a6f880 76ef9e54 kernel32!BaseThreadInitThunk+0x24
0d 03a6f8c8 76ef9e1f ntdll!_RtlUserThreadStart+0x2f
0e 03a6f8d8 00000000 ntdll!_RtlUserThreadStart+0x1b
```

We see the thread is waiting for an event 4ac:

```
0:012> kv 2
# ChildEBP RetAddr Args to Child
00 03a6f684 748d8ee9 000004ac 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
01 03a6f6f8 5f5fcba5 000004ac ffffffff 00000000 KERNELBASE!WaitForSingleObjectEx+0x99

0:012> !handle 4ac
Handle 000004ac
Type Event
```

Instead of digging into implementation internals we show a different approach. We can use **Constant Subtrace** (page 137) analysis pattern to find out possible owner thread candidates, and we can also check raw stack region **Execution Residue** (page 371) of different threads for **Place Trace** (page 804) of synchronization

<sup>225</sup> [http://en.cppreference.com/w/cpp/thread/condition\\_variable](http://en.cppreference.com/w/cpp/thread/condition_variable)

primitives and associated symbolic references (beware of **Coincidental Symbolic Information** though, page 137), and, if possible, **Past Stack Traces** (page 800) involving synchronization.

If we look at **Stack Trace Collection** (page 943) we can match the following thread that has the same **Constant Subtrace** as our original waiting thread above:

```
14 Id: 17a0.39d0 Suspend: 1 Teb: fee4f000 Unfrozen
# ChildEBP RetAddr
00 0679f9f0 748d9edc ntdll!NtReadFile+0xc
01 0679fa54 5c8f38f2 KERNELBASE!ReadFile+0xec
WARNING: Stack unwind information not available. Following frames may be wrong.
02 0679fa84 5c8f3853 AppA!foo+0x76732
03 0679fac8 5c8f37cd AppA!foo +0x76693
04 0679fae0 5c8c4a27 AppA!foo +0x7660d
05 0679faec 00dcc4e9 AppA!foo +0x47867
06 0679fb04 75823744 AppA+0x1c4e9
07 0679fb18 76ef9e54 kernel32!BaseThreadInitThunk+0x24
08 0679fb60 76ef9e1f ntdll!_RtlUserThreadStart+0x2f
09 0679fb70 00000000 ntdll!_RtlUserThreadStart+0x1b
```

When we dump raw stack data from all threads using WinDbg script<sup>226</sup> and search for 000004ac we find its occurrences in the raw stack corresponding to the thread #14 we already found:

```
[...]
TEB at fee4f000
ExceptionList: 0679fa44
StackBase: 067a0000
StackLimit: 0679e000
[...]
0679f90c 0679f918
0679f910 5f5a4894 msrvr120!Concurrency::details::SchedulerBase::CurrentContext+0x1e
0679f914 00000033
0679f918 0679f950
0679f91c 5f5a48ca msrvr120!Concurrency::details::LockQueueNode::LockQueueNode+0x2a
0679f920 0679f998
0679f924 071cf9ac
0679f928 0679f94c
0679f92c 5f5ff57e msrvr120!Concurrency::critical_section::_Acquire_lock+0x2e
0679f930 071cf9ac
0679f934 0679f994
0679f938 0679f998
0679f93c 00000001
0679f940 070f4bfc
0679f944 0679f970
0679f948 5f5ff41f msrvr120!Concurrency::critical_section::lock+0x31
0679f94c 0679f980
0679f950 5f5ff6dc msrvr120!Concurrency::critical_section::scoped_lock::scoped_lock+0x3b
0679f954 0679f998
```

---

<sup>226</sup> Memory Dump Analysis Anthology, Volume 1, page 231

## 1084 | Wait Chain

---

```
0679f958 76f08bcc ntdll!NtSetEvent+0xc
0679f95c 748e30b0 KERNELBASE!SetEvent+0x10
0679f960 000004ac
0679f964 00000000
0679f968 0679f984
0679f96c 5f5fcbe6 msvcr120!Concurrency::details::ExternalContextBase::Unblock+0x3f
0679f970 000004ac
0679f974 03a6f750
0679f978 03a6f750
0679f97c 0679f9b4
0679f980 5f5fb70d msvcr120!Concurrency::details::_Condition_variable::notify_all+0x3f
0679f984 0679f9b4
0679f988 5f5fb722 msvcr120!Concurrency::details::_Condition_variable::notify_all+0x54
0679f98c 07481380
0679f990 05ba6f60
0679f994 071cf9ac
[...]
```

Both methods point to the same possible owner thread which is also blocked in reading a file.

## CLR Monitors

This is a variation of a general **Wait Chain** (page 1092) pattern related to **CLR Threads** (page 124). When looking at **Stack Trace Collection** (page 943) from a complete memory dump, we may find threads using a monitor synchronization mechanism:

```
[... 32-bit ...]  
09d2e908 6ba4d409 clr!CLREvent::WaitEx+0x106  
09d2e91c 6bb90160 clr!CLREvent::Wait+0x19  
09d2e9ac 6bb90256 clr!AwareLock::EnterEpilogHelper+0xa8  
09d2e9ec 6bb9029b clr!AwareLock::EnterEpilog+0x42  
09d2ea0c 6ba90f78 clr!AwareLock::Enter+0x5f  
09d2eaa8 05952499 clr!JIT_MonEnterWorker_Portable+0xf8  
[...]
```

or

```
[... 64-bit ...]  
00000000`2094e230 000007fe`eedc3e3a clr!CLREvent::WaitEx+0xc1  
00000000`2094e2d0 000007fe`eedc3d43 clr!AwareLock::EnterEpilogHelper+0xca  
00000000`2094e3a0 000007fe`eee3e613 clr!AwareLock::EnterEpilog+0x63  
00000000`2094e400 000007ff`007f4c38 clr!JIT_MonEnterWorker_Portable+0x14f  
[...]
```

When seeing such threads, we may ask for a process memory dump to perform .NET memory dump analysis using SOS or other WinDbg extensions such as in **Deadlock** (page 202) pattern example for CLR 2 (*mscorwks*).

## Critical Sections

Here is another example of general **Wait Chain** pattern (page 1092) where objects are critical sections.

WinDbg can detect them if we use **!analyze -v -hang** command but it detects only one and not necessarily the longest or the widest chain in cases with multiple wait chains:

DERIVED\_WAIT\_CHAIN:

Dl	Eid	Cid	WaitType	
2	8d8.90c	Critical Section	-->	
4	8d8.914	Critical Section	-->	
66	8d8.f9c	Unknown		

Looking at threads we see this chain, and we also see that the final thread is blocked waiting for a socket.

```
0:167> ~~[90c]kvL
ChildEBP RetAddr  Args to Child
00bbfd9c 7c942124 7c95970f 00000ea0 00000000 ntdll!KiFastSystemCallRet
00bbfd9d 7c95970f 00000ea0 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
00bbfd9e 7c959620 00000000 00000004 00000000 ntdll!RtlpWaitOnCriticalSection+0x19c
00bbfd9f 6748d2f9 060118b50 00000000 00000000 ntdll!RtlEnterCriticalSection+0xa8
...
00bbfffb8 7c82608b 00315218 00000000 00000000 msvcrt!_endthreadex+0xa3
00bbffec 00000000 77b9b4bc 00315218 00000000 kernel32!BaseThreadStart+0x34

0:167> ~~[914]kvL 100
ChildEBP RetAddr  Args to Child
00dbf1cc 7c942124 7c95970f 000004b0 00000000 ntdll!KiFastSystemCallRet
00dbf1d0 7c95970f 000004b0 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
00dbf20c 7c959620 00000000 00000004 0031abcc ntdll!RtlpWaitOnCriticalSection+0x19c
00dbf22c 6748d244 00311abd8 003174e0 00dbf254 ntdll!RtlEnterCriticalSection+0xa8
...
00dbfffb8 7c82608b 00315218 00000000 00000000 msvcrt!_endthreadex+0xa3
00dbffec 00000000 77b9b4bc 00315218 00000000 kernel32!BaseThreadStart+0x34

0:167> ~~[f9c]kvL 100
ChildEBP RetAddr  Args to Child
0fe2a09c 7c942124 71933a09 00000b50 00000001 ntdll!KiFastSystemCallRet
0fe2a0a0 71933a09 00000b50 00000001 0fe2a0c8 ntdll!NtWaitForSingleObject+0xc
0fe2a0dc 7194576e 00000b50 00000234 00000000 mswock!SockWaitForSingleObject+0x19d
0fe2a154 71a12679 00000234 0fe2a1b4 00000001 mswock!WSPRecv+0x203
0fe2a190 62985408 00000234 0fe2a1b4 00000001 WS2_32!WSARecv+0x77
0fe2a1d0 6298326b 00000234 0274ebc6 00000810 component!wait+0x338
...
0fe2fffb8 7c82608b 060cf70 00000000 00000000 msvcrt!_endthreadex+0xa3
0fe2ffec 00000000 77b9b4bc 060cf70 00000000 kernel32!BaseThreadStart+0x34
```

If we look at all held critical sections we see another thread that blocked more than 125 other threads:

```
0:167> !locks

CriticalSection +31abd8 at 0031abd8
WaiterWoken      No
LockCount        6
RecursionCount   1
OwningThread     f9c
EntryCount       0
ContentionCount  17
*** Locked

CriticalSection +51e4bd8 at 051e4bd8
WaiterWoken      No
LockCount        125
RecursionCount   1
OwningThread     830
EntryCount       0
ContentionCount  7d
*** Locked

CriticalSection +5f40620 at 05f40620
WaiterWoken      No
LockCount        0
RecursionCount   1
OwningThread     920
EntryCount       0
ContentionCount  0
*** Locked

CriticalSection +60b6320 at 060b6320
WaiterWoken      No
LockCount        1
RecursionCount   1
OwningThread     8a8
EntryCount       0
ContentionCount  1
*** Locked

CriticalSection +6017c60 at 06017c60
WaiterWoken      No
LockCount        0
RecursionCount   1
OwningThread     914
EntryCount       0
ContentionCount  0
*** Locked
```

```
CritSec +6018b50 at 06018b50
```

```
WaiterWoken      No
LockCount        3
RecursionCount   1
OwningThread    914
EntryCount       0
ContentionCount  3
*** Locked
```

```
CritSec +6014658 at 06014658
```

```
WaiterWoken      No
LockCount        2
RecursionCount   1
OwningThread    928
EntryCount       0
ContentionCount  2
*** Locked
```

0:167> ~~[830]kvL 100

```
ChildEBP RetAddr  Args to Child
0ff2f300 7c942124 7c95970f 000004b0 00000000 ntdll!KiFastSystemCallRet
0ff2f304 7c95970f 000004b0 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
0ff2f340 7c959620 00000000 00000004 0031abcc ntdll!RtlpWaitOnCriticalSection+0x19c
0ff2f360 6748d244 0031abd8 003174e0 0ff2f388 ntdll!RtlEnterCriticalSection+0xa8
...
0ff2ffb8 7c82608b 060cf9a0 00000000 00000000 msvcrt!_endthreadex+0xa3
0ff2ffec 00000000 77b9b4bc 060cf9a0 00000000 kernel32!BaseThreadStart+0x34
```

Searching for any thread waiting for critical section 051e4bd8 gives us:

```
8  Id: 8d8.924 Suspend: 1 Teb: 7ffd5000 Unfrozen
ChildEBP RetAddr  Args to Child
011ef8e0 7c942124 7c95970f 00000770 00000000 ntdll!KiFastSystemCallRet
011ef8e4 7c95970f 00000770 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
011ef920 7c959620 00000000 00000004 00000000 ntdll!RtlpWaitOnCriticalSection+0x19c
011ef940 677b209d 051e4bd8 011efa0c 057bd36c ntdll!RtlEnterCriticalSection+0xa8
...
011effb8 7c82608b 00315510 00000000 00000000 msvcrt!_endthreadex+0xa3
011effec 00000000 77b9b4bc 00315510 00000000 kernel32!BaseThreadStart+0x34
```

and we can construct yet another **Wait Chain**:

```
8  8d8.924 Critical Section      -->
67 8d8.830 Critical Section      -->
66 8d8.f9c Unknown
```

## Executive Resources

The most common and easily detectable example of **Wait Chain** pattern (page 1092) in the kernel and complete memory dumps is when objects are executive resources (**Deadlock**, page 194). In some complex cases, we can even have multiple wait chains. For example, in the output of **!locks** WinDbg command below we can find at least three wait chains marked in bold, italics and bold italics:

```
883db310 -> 8967d020 -> 889fa230
89a74228 -> 883ad4e0 -> 88d7a3e0
88e13990 -> 899da538 -> 8805fac8
```

The manual procedure to figure chains is simple. Pick up a thread marked with <\*>. This is one that currently holds the resource. See what threads are waiting on exclusive access to that resource. Then search for other occurrences of <\*> thread to see whether it is waiting exclusively for another resource blocked by some other thread and so on.

```
1: kd> !locks
**** DUMP OF ALL RESOURCE OBJECTS ****

Resource @ 0x8b4425c8      Shared 1 owning threads
  Contention Count = 45
  NumberOfSharedWaiters = 43
  NumberOfExclusiveWaiters = 2
    Threads: 8967d020-01<*> 88748b10-01    88b8b020-01    8b779ca0-01
              88673248-01    8b7797c0-01    889c8358-01    8b777b40-01
              8b7763f0-01    8b776b40-01    8b778b40-01    8841b020-01
              8b7788d0-01    88cc7b00-01    8b776020-01    8b775020-01
[...]
              8b778020-01    8b779a30-01    8b778660-01    8943a020-01
              88758020-01    8b777db0-01    88ee3590-01    896f3020-01
              89fc4b98-01    89317938-01    8867f1e0-01    89414118-01
              88e989a8-01    88de5b70-01    88c4b588-01    8907dd48-01
  Threads Waiting On Exclusive Access:
    883db310        8907d280

Resource @ 0x899e27ac      Exclusively owned
  Contention Count = 1
  NumberOfExclusiveWaiters = 1
    Threads: 889fa230-01<*>
  Threads Waiting On Exclusive Access:
    8967d020

Resource @ 0x881a38f8      Exclusively owned
  Contention Count = 915554
  NumberOfExclusiveWaiters = 18
    Threads: 883ad4e0-01<*>
  Threads Waiting On Exclusive Access:
    89a74228        8844c630        8955e020        891aa440
    8946a270        898b7ab0        89470d20        881e5760
    8b594af8        88dce020        899df328        8aa86900
    897ff020        8920adb0        8972b1c0        89657c70
    88bcc868        88cb0cb0
```

```

Resource @ 0x88a8d5b0    Exclusively owned
Contention Count = 39614
NumberOfExclusiveWaiters = 3
Threads: 88d7a3e0-01<*>
Threads Waiting On Exclusive Access:
 883ad4e0      89a5f020      87d00020

Resource @ 0x89523658    Exclusively owned
Contention Count = 799193
NumberOfExclusiveWaiters = 18
Threads: 899da538-01<*>
Threads Waiting On Exclusive Access:
 88e13990      89a11cc0      88f4b2f8      898faab8
 8b3200c0      88758468      88b289f0      89fa4a58
 88bf2510      8911a020      87feb548      8b030db0
 887ad2c8      8872e758      89665020      89129810
 886be480      898a6020

Resource @ 0x897274b0    Exclusively owned
Contention Count = 37652
NumberOfExclusiveWaiters = 2
Threads: 8805fac8-01<*>
Threads Waiting On Exclusive Access:
 899da538      88210db0

Resource @ 0x8903db88    Exclusively owned
Contention Count = 1127998
NumberOfExclusiveWaiters = 17
Threads: 882c9b68-01<*>
Threads Waiting On Exclusive Access:
 8926fdb0      8918fd18      88036430      89bc2c18
 88fca478      8856d710      882778f0      887c3240
 88ee15e0      889d3640      89324c68      8887b020
 88d826a0      8912ca08      894edb10      87e518f0
 89896688

Resource @ 0x89351430    Exclusively owned
Contention Count = 51202
NumberOfExclusiveWaiters = 1
Threads: 882c9b68-01<*>
Threads Waiting On Exclusive Access:
 88d01378

Resource @ 0x87d6b220    Exclusively owned
Contention Count = 303813
NumberOfExclusiveWaiters = 14
Threads: 8835d020-01<*>
Threads Waiting On Exclusive Access:
 8b4c52a0      891e2ae0      89416888      897968e0
 886e58a0      89b327d8      894ba4c0      8868d648
 88a10968      89a1da88      8985a1d0      88f58a30
 89499020      89661220

Resource @ 0x88332cc8    Exclusively owned
Contention Count = 21214
NumberOfExclusiveWaiters = 3
Threads: 88d15b50-01<*>

```

```

Threads Waiting On Exclusive Access:
    88648020      8835d020      88a20ab8

Resource @ 0x8986ab80    Exclusively owned
Contention Count = 753246
NumberOfExclusiveWaiters = 13
Threads: 88e6ea60-01<*>
Threads Waiting On Exclusive Access:
    89249020      87e01d50      889fb6c8      89742cd0
    8803b6a8      888015e0      88a89ba0      88c09020
    8874d470      88d97db0      8919a2d0      882732c0
    89a9eb28

Resource @ 0x88c331c0    Exclusively owned
Contention Count = 16940
NumberOfExclusiveWaiters = 2
Threads: 8b31c748-01<*>
Threads Waiting On Exclusive Access:
    896b3390      88e6ea60

33816 total locks, 20 locks currently held

```

Now we can dump the top thread from selected **Wait Chain** to see its call stack and why it is stuck holding other threads. For example,

```

883db310 -> 8967d020 -> 889fa230

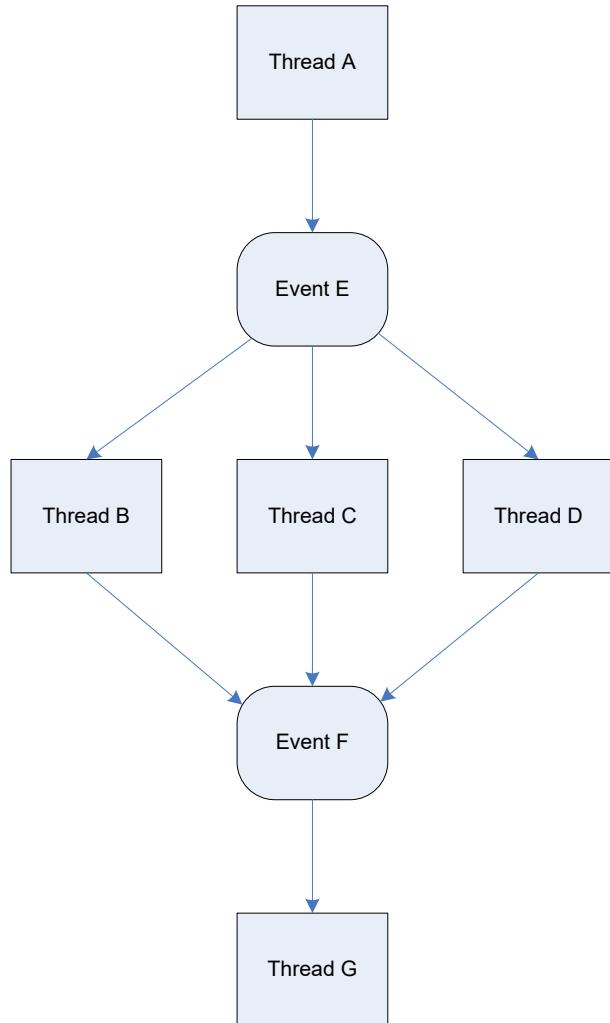
1: kd> !thread 889fa230
THREAD 889fa230 Cid 01e8.4054 Teb: 7ffac000 Win32Thread: 00000000 RUNNING on processor 3
IRP List:
    889fc008: (0006,0094) Flags: 00000a00 Mdl: 00000000
Impersonation token: e29c1630 (Level Impersonation)
DeviceMap           d7030620
Owning Process     89bcb480     Image:       MyService.exe
Wait Start TickCount 113612852   Ticks: 6 (0:00:00:00.093)
Context Switch Count 19326191
UserTime            00:00:11.0765
KernelTime          18:52:35.0750
Win32 Start Address 0x0818f190
LPC Server thread working on message Id 818f190
Start Address 0x77e6b5f3
Stack Init 8ca60000 Current 8ca5fb04 Base 8ca60000 Limit 8ca5d000 Call 0
Priority 8 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr Args to Child
8ca5fbb8 b86c1c93 c00a0006 00000003 8ca5fbfc driver!foo5+0x67
8ca5fbdc b86bdb8a 8b4742e0 00000003 8ca5fbfc driver!foo4+0x71
8ca5fc34 b86bf682 8b4742e0 889fc008 889fc078 driver!foo3+0xd8
8ca5fc50 b86bfc74 889fc008 889fc078 889fc008 driver!foo2+0x40
8ca5fc68 8081dcdf 8b45a3c8 889fc008 889fa438 driver!foo+0xd0
8ca5fc7c 808f47b7 889fc078 00000001 889fc008 nt!IoFastCallEntry+0x45
8ca5fc90 808f24ee 8b45a3c8 889fc008 89507e90 nt!IoFastCallEntry+0x10b
8ca5fd38 80888c7c 0000025c 0000029d 00000000 nt!NtWriteFile+0x65a
8ca5fd38 7c82ed54 0000025c 0000029d 00000000 nt!KiFastCallEntry+0xfc

```

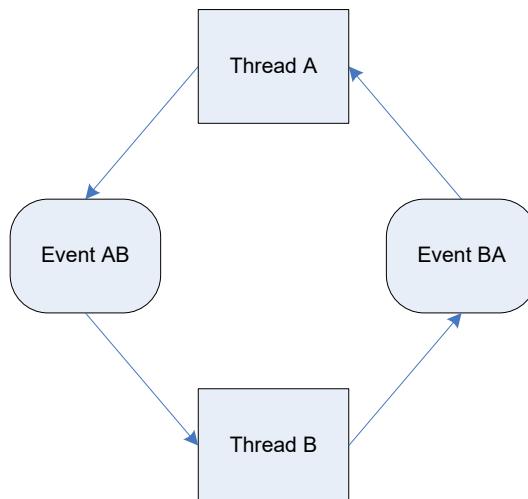
Because of huge kernel time, contention count and RUNNING status it is most probably the instance of **Spiking Thread** pattern (page 888) involving *driver.sys* called in the context of *MyService.exe* process.

## General

**Wait Chain** pattern is simply a sequence of causal relations between events: *Thread A* is waiting for *Event E* to happen that *Threads B, C or D* are supposed to signal at some time in the future, but they are all waiting for *Event F* to happen that *Thread G* is about to signal as soon as it finishes processing some critical task:

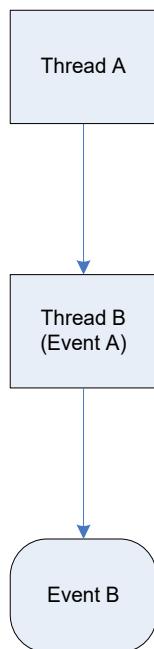


This subsumes various **Deadlock** patterns (page 1169) too which are causal loops where *Thread A* is waiting for *Event AB* that *Thread B* will signal as soon as *Thread A* signals *Event BA*. *Thread B* is waiting for:



In this context “Event” means not only a Win32 event object or kernel \_KEVENT but any type of the synchronization object, a critical section, LPC/RPC reply or data arrival through some IPC channel.

As the first example of **Wait Chain** pattern, we show a process being terminated and waiting for another thread to finish or, in other words, considering thread termination as an event itself, the main process thread is waiting for the second thread object to be signaled. The second thread tries to cancel previous I/O request directed to some device. However, that IRP is not cancellable, and process hangs. This can be depicted in the following diagram:



where *Thread A* is our main thread waiting for *Event A*, which is *Thread B* itself waiting for I/O cancellation (*Event B*). Their stack traces are:

```

THREAD 8a3178d0 Cid 04bc.01cc Teb: 7fffd000 Win32Thread: bc1b6e70 WAIT: (Unknown) KernelMode Non-Alertable
 8af2c920 Thread
Not impersonating
DeviceMap          e1032530
Owning Process     89ff8d88      Image:       processA.exe
Wait Start Ticks   80444        Ticks: 873 (0:00:00:13.640)
Context Switch Count 122           LargeStack
UserTime            00:00:00.015
KernelTime          00:00:00.156
Win32 Start Address 0x010148a4
Start Address 0x77e617f8
Stack Init f3f29000 Current f3f28be8 Base f3f29000 Limit f3f25000 Call 0
Priority 15 BasePriority 13 PriorityDecrement 0
ChildEBP RetAddr
f3f28c00 80833465 nt!KiSwapContext+0x26
f3f28c2c 80829a62 nt!KiSwapThread+0x2e5
f3f28c74 8094c0ea nt!KeWaitForSingleObject+0x346 ; stack trace with arguments shows the first parameter
as 8af2c920
f3f28d0c 8094c63f nt!PspExitThread+0x1f0
f3f28d24 8094c839 nt!PspTerminateThreadByPointer+0x4b
f3f28d54 8088978c nt!NtTerminateProcess+0x125
f3f28d54 7c8285ec nt!KiFastCallEntry+0xfc
  
```

```

THREAD 8af2c920 Cid 04bc.079c Teb: 7ffd7000 Win32Thread: 00000000 WAIT: (Unknown) KernelMode Non-Alertable
    8af2c998 NotificationTimer
IRP List:
    8ad26260: (0006,0220) Flags: 00000000 Mdl: 00000000
Not impersonating
DeviceMap          e1032530
Owning Process     89ff8d88      Image:       processA.exe
Wait Start TickCount 81312       Ticks: 5 (0:00:00:00.078)
Context Switch Count 169         LargeStack
UserTime           00:00:00.000
KernelTime         00:00:00.000
Win32 Start Address 0x77da3ea5
Start Address 0x77e617ec
Stack Init f3e09000 Current f3e08bac Base f3e09000 Limit f3e05000 Call 0
Priority 13 BasePriority 13 PriorityDecrement 0
ChildEBP RetAddr
f3e08bc4 80833465 nt!KiSwapContext+0x26
f3e08bf0 80828f0b nt!KiSwapThread+0x2e5
f3e08c38 808ea7a4 nt!KeDelayExecutionThread+0x2ab
f3e08c68 8094c360 nt!IoCancelThreadIo+0x62
f3e08cf0 8094c569 nt!PspExitThread+0x466
f3e08cfc 8082e0b6 nt!PsExitSpecialApc+0x1d
f3e08d4c 80889837 nt!KiDeliverApc+0x1ae
f3e08d4c 7c8285ec nt!KiServiceExit+0x56

```

By inspecting IRP we can see a device it was directed to, see that it has the cancel bit but doesn't have a cancel routine:

```

0: kd> !irp 8ad26260 1
Irp is active with 5 stacks 4 is current (= 0x8ad2633c)
No Mdl: No System Buffer: Thread 8af2c920: Irp stack trace.
Flags = 00000000
ThreadListEntry.Flink = 8af2cb28
ThreadListEntry.Blink = 8af2cb28
IoStatus.Status = 00000000
IoStatus.Information = 00000000
RequestorMode = 00000001
Cancel = 01
CancelIrql = 0
ApcEnvironment = 00
UserIosb = 77ecb700
UserEvent = 00000000
Overlay.AsyncParameters.UserApcRoutine = 00000000
Overlay.AsyncParameters.UserApcContext = 00000000
Overlay.AllocationSize = 00000000 - 00000000
CancelRoutine = 00000000
UserBuffer = 77ecb720
&Tail.Overlay.DeviceQueueEntry = 8ad262a0
Tail.Overlay.Thread = 8af2c920
Tail.Overlay.AuxiliaryBuffer = 00000000
Tail.Overlay.ListEntry.Flink = 00000000
Tail.Overlay.ListEntry.Blink = 00000000
Tail.Overlay.CurrentStackLocation = 8ad2633c
Tail.Overlay.OriginalFileObject = 89ff8920

```

```
Tail.Apc = 00000000
Tail.CompletionKey = 00000000
    cmd flg cl Device File Completion-Context
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

    Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

    Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

    Args: 00000000 00000000 00000000 00000000
>[ c, 2] 0 1 8ab20388 89ff8920 00000000-00000000 pending
    \Device\DeviceA
    Args: 00000020 00000017 00000000 00000000
[ c, 2] 0 0 8affa4b8 89ff8920 00000000-00000000
    \Device\DeviceB
    Args: 00000020 00000017 00000000 00000000
```

---

## Comments

---

Wait Chain Traversal API in Windows Server 2008<sup>227</sup>.

---

<sup>227</sup> <http://msdn2.microsoft.com/en-us/library/cc308564.aspx>

## LPC/ALPC

**Wait Chains** (page 1092) involving LPC calls are easily identified by searching for “Waiting for reply” in the output of **!process 0 3f** command, or if we know that some specific process is hanging and see that message in its thread information output. For example, in one kernel memory dump file saved when *AppA* was hanging we see this example of **Blocked Thread** pattern (page 82):

```
7: kd> !process 88556778 3f
PROCESS 88556778 SessionId: 0 Cid: 1f88 Peb: 7fffdc000 ParentCid: 0f74
DirBase: 96460000 ObjectTable: e65a5348 HandleCount: 80.
Image: AppA.exe
VadRoot 870d2208 Vads 54 Clone 0 Private 234. Modified 0. Locked 0.
DeviceMap e22ba7c0
Token e5e47cf0
ElapsedTime 00:04:44.017
UserTime 00:00:00.000
KernelTime 00:00:00.000
QuotaPoolUsage[PagedPool] 20092
QuotaPoolUsage[NonPagedPool] 2160
Working Set Sizes (now,min,max) (748, 50, 345) (2992KB, 200KB, 1380KB)
PeakWorkingSetSize 748
VirtualSize 16 Mb
PeakVirtualSize 16 Mb
PageFaultCount 810
MemoryPriority BACKGROUND
BasePriority 8
CommitCharge 252

PEB NULL...
```

```
THREAD 8861aab8 Cid 1f88.1bd4 Peb: 7ffffdf000 Win32Thread: bc161ea8 WAIT: (Unknown) UserMode Non-
Alertable
8861aca4 Semaphore Limit 0x1
Waiting for reply to LPC MessageId 00037bb2:
Current LPC port e625bbd0
Not impersonating
DeviceMap e22ba7c0
Owning Process 88556778 Image: AppA.exe
Wait Start TickCount 426549 Ticks: 18176 (0:00:04:44.000)
Context Switch Count 76 LargeStack
UserTime 00:00:00.000
KernelTime 00:00:00.000
Win32 Start Address 0x010043ab
Start Address 0x77e617f8
Stack Init bab4b000 Current bab4ac08 Base bab4b000 Limit bab47000 Call 0
Priority 11 BasePriority 8 PriorityDecrement 2
Kernel stack not resident.
ChildEBP RetAddr
bab4ac20 8083d5b1 nt!KiSwapContext+0x26
bab4ac4c 8083df9e nt!KiSwapThread+0x2e5
bab4ac94 8093eda1 nt!KeWaitForSingleObject+0x346
bab4ad50 80833bef nt!NtRequestWaitReplyPort+0x776
bab4ad50 7c8285ec nt!KiFastCallEntry+0xfc
```

Now we look for a server thread processing the message 00037bb2:

```
7: kd> !lpc message 00037bb2
Searching message 37bb2 in threads ...
Server thread 89815db0 is working on message 37bb2
Client thread 8861aab8 waiting a reply from 37bb2
Searching thread 8861aab8 in port rundown queues ...

Server communication port 0xe1216be8
Handles: 1 References: 1
The LpcDataInfoChainHead queue is empty
Connected port: 0xe625bbd0 Server connection port: 0xe1323f68

Client communication port 0xe625bbd0
Handles: 1 References: 2
The LpcDataInfoChainHead queue is empty

Server connection port e1323f68 Name: ApiABC
Handles: 1 References: 43
Server process : 887d32d0 (svchost.exe)
Queue semaphore : 884df210
Semaphore state 0 (0x0)
The message queue is empty

Messages in LpcDataInfoChainHead:
0000 e6067418 - Busy Id=00020695 From: 0224.134c Context=80050003 [e1323fe8 . e65fa5a8]
Length=0044002c Type=00380001 (LPC_REQUEST)
Data: 00000001 00050242 00000000 00000000 00000000 00000000
0000 e65fa5a8 - Busy Id=0002e1dd From: 0fd8.0fe0 Context=80110002 [e6067418 . e5f6a360]
Length=0044002c Type=00380001 (LPC_REQUEST)
Data: 00000001 00050242 c03007fc c01ffff7c 00000000 80a84456
0000 e5f6a360 - Busy Id=00037bb2 From: 1f88.1bd4 Context=8017000f [e65fa5a8 . e1323fe8]
Length=0044002c Type=00380001 (LPC_REQUEST)
Data: 00000001 00050242 88572278 88572290 8a386990 000015e7
The LpcDataInfoChainHead queue contains 3 messages
Threads in RunDown queue : 0xe6067258 0xe65fa3e8 0xe5f6a1a0
Done.

7: kd> !thread 89815db0
THREAD 89815db0 Cid 1218.0c00 Peb: 7ff8f000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Non-Alertable
88603e40 Mutant - owning thread 884d7db0
Not impersonating
DeviceMap e10018b8
Owning Process 887d32d0 Image: svchost.exe
Wait Start TickCount 426549 Ticks: 18176 (0:00:04:44.000)
Context Switch Count 42
UserTime 00:00:00.000
KernelTime 00:00:00.000
Win32 Start Address 0x00037bb2
LPC Server thread working on message Id 37bb2
Start Address 0x77e617ec
Stack Init f60e0000 Current f60dfc60 Base f60e0000 Limit f60dd000 Call 0
Priority 9 BasePriority 8 PriorityDecrement 0
```

```

Kernel stack not resident.
ChildEBP RetAddr  Args to Child
f60dfc78 8083d5b1 89815db0 89815e58 00000006 nt!KiSwapContext+0x26
f60dfca4 8083df9e 00000000 00000000 00000000 nt!KiSwapThread+0x2e5
f60dfcec 8092ae57 88603e40 00000006 00000001 nt!KeWaitForSingleObject+0x346
f60dfd50 80833bef 000004fc 00000000 00000000 nt!NtWaitForSingleObject+0x9a
f60dfd50 7c8285ec 000004fc 00000000 00000000 nt!KiFastCallEntry+0xfc

```

We see that it is blocked waiting for a synchronization object (mutant, shown in bold italicics above), and we check the thread 884d7db0 that owns it:

```

7: kd> !thread 884d7db0
THREAD 884d7db0  Cid 1218.12ec  Teb: 7fffd000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Non-
Alertable
    884d7f9c Semaphore Limit 0x1
Waiting for reply to LPC MessageId 0000fa9e:
Current LPC port e121fdb8
Not impersonating
DeviceMap          e10018b8
Owning Process     887d32d0      Image:           svchost.exe
Wait Start TickCount 11800        Ticks: 432925 (0:01:52:44.453)
Context Switch Count 111
UserTime            00:00:00.000
KernelTime          00:00:00.000
Win32 Start Address 0x0000fa9b
LPC Server thread working on message Id fa9b
Start Address 0x77e617ec
Stack Init f4598000 Current f4597c08 Base f4598000 Limit f4595000 Call 0
Priority 9 BasePriority 8 PriorityDecrement 0
Kernel stack not resident.
ChildEBP RetAddr  Args to Child
f4597c20 8083d5b1 884d7db0 884d7e58 00000007 nt!KiSwapContext+0x26
f4597c4c 8083df9e 884d7f9c 884d7f70 884d7db0 nt!KiSwapThread+0x2e5
f4597c94 8093eda1 884d7f9c 00000011 80930901 nt!KeWaitForSingleObject+0x346
f4597d50 80833bef 00000560 000ebfe0 000ebfe0 nt!NtRequestWaitReplyPort+0x776
f4597d50 7c8285ec 00000560 000ebfe0 000ebfe0 nt!KiFastCallEntry+0xfc

```

The thread is waiting for the LPC message 0000fa9e, and we look for a server thread processing it:

```

7: kd> !thread 898c6db0
THREAD 898c6db0  Cid 0b38.188c  Teb: 7ff4d000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Non-
Alertable
    884362c8 SynchronizationEvent
Not impersonating
DeviceMap          e11399e8
Owning Process     88340758      Image:           AppB.exe
Wait Start TickCount 11801        Ticks: 432924 (0:01:52:44.437)
Context Switch Count 7
UserTime            00:00:00.000
KernelTime          00:00:00.000
Win32 Start Address 0x0000fa9e
LPC Server thread working on message Id fa9e
Start Address 0x77e617ec
Stack Init f5138000 Current f5137c60 Base f5138000 Limit f5135000 Call 0

```

```

Priority 9 BasePriority 8 PriorityDecrement 0
Kernel stack not resident.
ChildEBP RetAddr Args to Child
f5137c78 8083d5b1 898c6db0 898c6e58 00000006 nt!KiSwapContext+0x26
f5137ca4 8083df9e 00000000 00000000 00000000 nt!KiSwapThread+0x2e5
f5137cec 8092ae57 884362c8 00000006 00000001 nt!KeWaitForSingleObject+0x346
f5137d50 80833bef 0000056c 00000000 00000000 nt!NtWaitForSingleObject+0x9a
f5137d50 7c8285ec 0000056c 00000000 00000000 nt!KiFastCallEntry+0xfc

```

We also see that the thread 884d7db0 was working on the message fa9b (shown in underlined bold above), and therefore we can find its client thread:

```

7: kd> !lpc message fa9b
Searching message fa9b in threads ...
    Server thread 884d7db0 is working on message fa9b
Client thread 871ab9a0 waiting a reply from fa9b
Searching thread 871ab9a0 in port rundown queues ...

Server communication port 0xe23f68b8
    Handles: 1    References: 1
    The LpcDataInfoChainHead queue is empty
        Connected port: 0xe1325c10      Server connection port: 0xe1323f68

Client communication port 0xe1325c10
    Handles: 1    References: 2
    The LpcDataInfoChainHead queue is empty

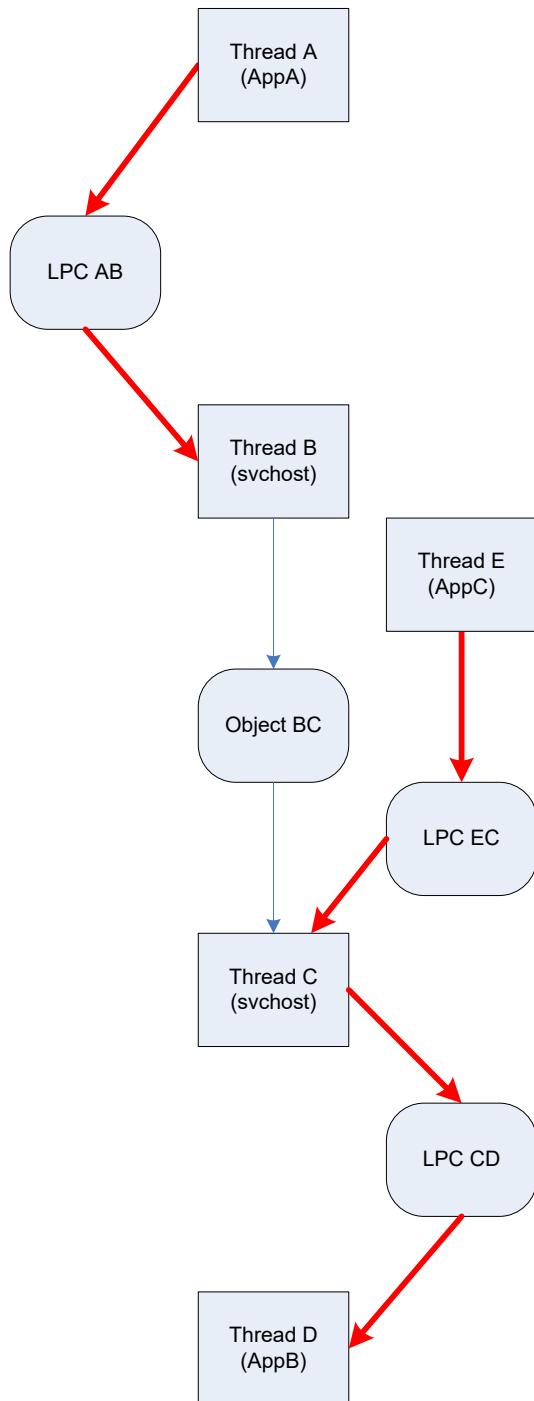
Server connection port e1323f68  Name: ApiABC
    Handles: 1    References: 43
    Server process : 887d32d0 (svchost.exe)
    Queue semaphore : 884df210
    Semaphore state 0 (0x0)
    The message queue is empty

Messages in LpcDataInfoChainHead:
    0000 e6067418 - Busy Id=00020695 From: 0224.134c Context=80050003 [e1323fe8 . e65fa5a8]
        Length=0044002c Type=00380001 (LPC_REQUEST)
        Data: 00000001 00050242 00000000 00000000 00000000 00000000
    0000 e65fa5a8 - Busy Id=0002e1dd From: 0fd8.0fe0 Context=80110002 [e6067418 . e5f6a360]
        Length=0044002c Type=00380001 (LPC_REQUEST)
        Data: 00000001 00050242 c03007fc c01ffff7c 00000000 80a84456
    0000 e5f6a360 - Busy Id=00037bb2 From: 1f88.1bd4 Context=8017000f [e65fa5a8 . e1323fe8]
        Length=0044002c Type=00380001 (LPC_REQUEST)
        Data: 00000001 00050242 88572278 88572290 8a386990 000015e7
The LpcDataInfoChainHead queue contains 3 messages
Threads in RunDown queue :      0xe6067258      0xe65fa3e8      0xe5f6a1a0
Done.

```

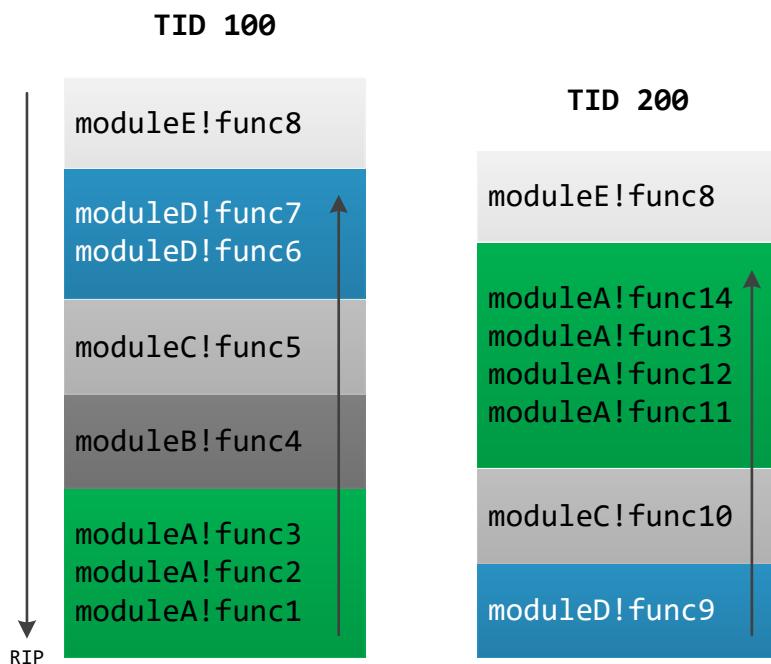
```
7: kd> !thread 871ab9a0
THREAD 871ab9a0  Cid 180c.1810  Teb: 7fffd000 Win32Thread: bc011008 WAIT: (Unknown) UserMode Non-
Alertable
    871abb8c  Semaphore Limit 0x1
Waiting for reply to LPC MessageId 0000fa9b:
Current LPC port e1325c10
Not impersonating
DeviceMap          e10018b8
Owning Process    8963c388      Image:        AppC.exe
Wait Start TickCount 11796       Ticks: 432929 (0:01:52:44.515)
Context Switch Count 540         LargeStack
UserTime           00:00:00.046
KernelTime         00:00:00.062
Start Address 0x0103e1b0
Stack Init f68a4000 Current f68a3c08 Base f68a4000 Limit f689f000 Call 0
Priority 15 BasePriority 15 PriorityDecrement 0
Kernel stack not resident.
ChildEBP RetAddr  Args to Child
f68a3c20 8083d5b1 871ab9a0 871aba48 00000004 nt!KiSwapContext+0x26
f68a3c4c 8083df9e 871abb8c 871abb60 871ab9a0 nt!KiSwapThread+0x2e5
f68a3c94 8093eda1 871abb8c 00000011 e24f0401 nt!KeWaitForSingleObject+0x346
f68a3d50 80833bef 00000150 0007fc70 0007fc70 nt!NtRequestWaitReplyPort+0x776
f68a3d50 7c8285ec 00000150 0007fc70 0007fc70 nt!KiFastCallEntry+0xfc
```

Finally, we can draw this **Wait Chain** diagram where LPC calls are shown as bold arrows:



## Modules

Most if not all **Wait Chain** patterns (page 1188) are about waiting for some synchronization objects. **Stack Traces** (page 926) of involved threads may point to **Blocking Module** (page 96) and **Top Module** (page 1012). In some situations, we may consider a module (which functions were called) itself as a pseudo-synchronization object where a module (who called it) is waiting for it to return back (to become “signaled”). All this is problem and context dependent where some intermediate modules may be **Pass Through** (page 787) or **Well-Tested** (page 1147). When we see module inversion, such as in the case of callbacks we may provisionally suspect some kind of a deadlock and then investigate these threads in terms of synchronization objects or their corresponding source code:



## Comments

See also **Quotient Stack Trace** (page 824).

## Mutex Objects

This is an additional variation of the general **Wait Chain** (page 1092) pattern where mutexes (mutants) are involved in thread wait chains, for example:

```
THREAD ffffffa8019388b60 Cid 02e8.cfd0 Teb: 000007fffffa2000 Win32Thread: 0000000000000000 WAIT:
(UserRequest) UserMode Non-Alertable
fffffa800d75daf0 Mutant - owning thread ffffffa800ea2ab60
[...]
```

```
THREAD ffffffa8016abab60 Cid 02e8.ec34 Teb: 000007fffffae000 Win32Thread: 0000000000000000 WAIT:
(UserRequest) UserMode Non-Alertable
fffffa800d75daf0 Mutant - owning thread ffffffa800ea2ab60
[...]
```

We have seen such dependencies in various previous pattern interaction case studies such as:

- Inconsistent Dump, Stack Trace Collection, LPC, Thread, Process, Executive Resource Wait Chains, Missing Threads and Waiting Thread Time<sup>228</sup>
- Inconsistent Dump, Blocked Threads, Wait Chains, Incorrect Stack Trace and Process Factory<sup>229</sup>
- **Semantic Split** pattern example (page 853)
- Insufficient Memory, Handle Leak, Wait Chain, Deadlock, Inconsistent Dump and Overaged System<sup>230</sup>
- Blocked GUI Thread, Wait Chain and Virtualized Process<sup>231</sup>
- LPC/ALPC **Wait Chain** pattern example (page 1097)
- Mixed object **Deadlock** pattern example (page 205)
- LPC **Deadlock** pattern example (page 197)

Another example we show here is an unusual number of mutant dependencies in one complete memory dump from a hung system:

```
AppA(KTHREAD-1) -> AppB(KTHREAD-2)
AppB(KTHREAD-3) -> ServiceA(KTHREAD-4)
AppA(KTHREAD-5) -> ServiceA(KTHREAD-4)
AppB(KTHREAD-6) -> AppA(KTHREAD-7)
AppB(KTHREAD-6) -> AppC(KTHREAD-8)
AppC(KTHREAD-9) -> ServiceA(KTHREAD-4)
AppC(KTHREAD-10) -> AppB(KTHREAD-11)
```

<sup>228</sup> Memory Dump Analysis Anthology, Volume 5, page 133

<sup>229</sup> Memory Dump Analysis Anthology, Volume 3, page 279

<sup>230</sup> Memory Dump Analysis Anthology, Volume 3, page 175

<sup>231</sup> Memory Dump Analysis Anthology, Volume 3, page 170

Here the notation  $\text{AppX}(N) \rightarrow \text{AppY}(M)$  means that *Thread N* from *AppX* process is waiting for a mutant that is owned by *Thread M* from *AppY* process. Because *AppB*, *AppC*, and *ServiceA* belonged to **Same Vendor** (page 842) we advised checking with that ISV.

## Named Pipes

This is a variant of the general **Wait Chain** pattern (page 1092) where threads are waiting for named pipes. This is visible when we examine a pending IRP from **Blocked Thread** (page 82):

```
THREAD 88ec9020 Cid 17a0.2034 Teb: 7ffad000 Win32Thread: bc28c6e8 WAIT: (Unknown) UserMode Non-Alertable
89095f48 Semaphore Limit 0x10000
IRP List:
89a5a370: (0006,0094) Flags: 00000900 Mdl: 00000000
Not impersonating
DeviceMap d6c30c48
Owning Process 88ffffd88 Image: ApplicationA.exe
Attached Process N/A Image: N/A
Wait Start TickCount 5632994 Ticks: 2980 (0:00:00:46.562)
Context Switch Count 2269 LargeStack
UserTime 00:00:00.000
KernelTime 00:00:00.000
Win32 Start Address 0x00a262d0
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init b204c000 Current b204bc60 Base b204c000 Limit b2048000 Call 0
Priority 8 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr
b204bc78 80833ec5 nt!KiSwapContext+0x26
b204bca4 80829c14 nt!KiSwapThread+0x2e5
b204bcec 8093b174 nt!KeWaitForSingleObject+0x346
b204bd50 8088b41c nt!NtWaitForSingleObject+0x9a
b204bd50 7c82860c nt!KiFastCallEntry+0xfc (TrapFrame @ b204bd64)
058fcabc 7c827d29 ntdll!KiFastSystemCall1Ret
058fcac0 77e61d1e ntdll!ZwWaitForSingleObject+0xc
058fcb30 77e61c8d kernel32!WaitForSingleObjectEx+0xac
058fcb44 00f98b4a kernel32!WaitForSingleObject+0x12
[...]
058ffffec 00000000 kernel32!BaseThreadStart+0x34

0: kd> !irp 89a5a370
Irp is active with 1 stacks 1 is current (= 0x89a5a3e0)
No Mdl: No System Buffer: Thread 88ec9020: Irp stack trace.
cmd flg cl Device File Completion-Context
>[ 3, 0] 0 1 89eb00 891d4f90 00000000-00000000 pending
\FileSystem\Npfs
Args: 00000100 00000000 00000000 00000000
```

```
0: kd> !fileobj 891d4f90

\ServiceB\SVC

Device Object: 0x89eb0090 \FileSystem\Npfs
Vpb is NULL

Flags: 0x40080
Named Pipe
Handle Created

FsContext: 0xdaeca230 FsContext2: 0x8949bdb0
Private Cache Map: 0x00000001
CurrentByteOffset: 0
```

The pipe chain can also extend from a thread to a thread and even cross machine boundary.

## Nonstandard Synchronization

Sometimes developers introduce their own variants of synchronization code instead of using synchronization API provided by language runtime and OS. If we are lucky we can spot it in function and class method names and then use **Constant Subtrace** (page 137) analysis pattern:

```
0: kd> kc
*** Stack trace for last set context - .thread/.cxr resets it
# Call Site
00 nt!KiSwapContext
01 nt!KiCommitThreadWait
02 nt!KeWaitForSingleObject
03 nt!NtWaitForSingleObject
04 nt!KiSystemServiceCopyEnd
05 ntdll!ZwWaitForSingleObject
06 KERNELBASE!WaitForSingleObjectEx
07 wbemcomm!CwbemCriticalSection::Enter
08 wbemcore!EnsureInitialized
09 wbemcore!InitAndwaitForClient
0a wbemcore!CwbemLevel1Login::ConnectorLogin
0b wbemcore!CwbemLevel1Login::NTLMLogin
0c RPCRT4!Invoke
0d RPCRT4!NdrStubCall2
0e ole32!CStdStubBuffer_Invoke
0f ole32!SyncStubInvoke
10 ole32!StubInvoke
11 ole32!CCTxComChnl::ContextInvoke
12 ole32!AppInvoke
13 ole32!ComInvokeWithLockAndIPID
14 ole32!ThreadInvoke
15 RPCRT4!DispatchToStubInCNoAvrf
16 RPCRT4!RPC_INTERFACE::DispatchToStubWorker
17 RPCRT4!RPC_INTERFACE::DispatchToStub
18 RPCRT4!RPC_INTERFACE::DispatchToStubWithObject
19 RPCRT4!LRPC_SCALL::DispatchRequest
1a RPCRT4!LRPC_SCALL::HandleRequest
1b RPCRT4!LRPC_SASSOCIATION::HandleRequest
1c RPCRT4!LRPC_ADDRESS::HandleRequest
1d RPCRT4!LRPC_ADDRESS::ProcessIO
1e RPCRT4!LrpclIoComplete
1f ntdll!TppAlpcpExecuteCallback
20 ntdll!TppWorkerThread
21 kernel32!BaseThreadInitThunk
22 ntdll!RtlUserThreadStart
```

```

0: kd> kc
*** Stack trace for last set context - .thread/.cxr resets it
# Call Site
00 repdrvfs!SCachePage::operator=
01 repdrvfs!std::vector<scachepage,wbem_allocator<scachepage> >::erase
02 repdrvfs!CPageCache::Read
03 repdrvfs!CPageFile::GetPage
04 repdrvfs!ValidateBTreeAgainstObjHeap
05 repdrvfs!PerformAllValidations
06 repdrvfs!VerifyRepositoryOnline
07 repdrvfs!VerifyRepository
08 repdrvfs!CPageSource::Startup
09 repdrvfs!CPageSource::Init
0a repdrvfs!CFileCache::InnerInitialize
0b repdrvfs!CFileCache::Initialize
0c repdrvfs!CRepository::Initialize
0d repdrvfs!CRepository::Logon
0e wbemcore!CRepository::Init
0f wbemcore!InitSubsystems
10 wbemcore!ConfigMgr::InitSystem
11 wbemcore!EnsureInitialized
12 wbemcore!InitAndWaitForClient
13 wbemcore!CWbemLevel1Login::ConnectorLogin
14 wbemcore!CWbemLevel1Login::NTLMLogin
15 RPCRT4!Invoke
16 RPCRT4!NdrStubCall12
17 ole32!CStdStubBuffer_Invoke
18 ole32!SyncStubInvoke
19 ole32!StubInvoke
1a ole32!CCTxComChnl::ContextInvoke
1b ole32!AppInvoke
1c ole32!ComInvokeWithLockAndIPID
1d ole32!ThreadInvoke
1e RPCRT4!DispatchToStubInCNoAvrf
1f RPCRT4!RPC_INTERFACE::DispatchToStubWorker
20 RPCRT4!RPC_INTERFACE::DispatchToStub
21 RPCRT4!RPC_INTERFACE::DispatchToStubWithObject
22 RPCRT4!LRPC_CALL::DispatchRequest
23 RPCRT4!LRPC_CALL::HandleRequest
24 RPCRT4!LRPC_SASSOCIATION::HandleRequest
25 RPCRT4!LRPC_ADDRESS::HandleRequest
26 RPCRT4!LRPC_ADDRESS::ProcessIO
27 RPCRT4!LrpclIoComplete
28 ntdll!TppAlpcpExecuteCallback
29 ntdll!TppWorkerThread
2a kernel32!BaseThreadInitThunk
2b ntdll!RtlUserThreadStart

```

These two thread stack traces were spotted from a complete memory dump **Stack Trace Collection** (page 943) as the part of a larger **ALPC Wait Chain** (page 1097). We switched to these threads using **.thread /r /p** WinDbg command to get the stripped stack trace via **kc** command for better illustration.

We see **Constant Subtrace** (page 137) until *wbemcore!EnsureInitialized* function which serves as a bifurcation stack frame. The first stack trace has *CriticalSection::Enter* after the bifurcation stack frame compared to the second stack trace which looks like **Spiking Thread** (page 885) in user space.

There is no hidden critical section associated with that process except the one which is probably related to the spiking **Variable Subtrace** (page 1058) since it doesn't have any *LockCount*:

```
0: kd> !cs -l -o -s
DebugInfo      = 0x0000000004a774d0
CriticalSection = 0x0000000000308d690 (+0x308D690)
LOCKED
LockCount      = 0x0
WaiterWoken    = No
OwningThread   = 0x000000000000000928
RecursionCount = 0x1
LockSemaphore  = 0x0
SpinCount       = 0x0000000000000000
OwningThread   = .thread ffffffa806ebd8060
ntdll!RtlpStackTraceDataBase is NULL. Probably the stack traces are not enabled
```

We can also disassemble *wbemcomm!CWbemCriticalSection::Enter* and find out that it calls *WaitForSingleObject* once and no other synchronization API indeed:

```
0: kd> uf wbemcomm!CWbemCriticalSection::Enter
[...]
wbemcomm!CWbemCriticalSection::Enter+0x41:
000007fe`f78ad1c0 mov    rcx,qword ptr [rbx+10h]
000007fe`f78ad1c4 mov    edx,r12d
000007fe`f78ad1c7 call   qword ptr [wbemcomm!_imp_WaitForSingleObject (000007fe`f78ee4f0)]
000007fe`f78ad1cd cmp    eax,esi
000007fe`f78ad1cf je    wbemcomm!CWbemCriticalSection::Enter+0x52 (000007fe`f78a62a3) Branch
```

We add this **Nonstandard Synchronization** memory analysis pattern to **Wait Chain** analysis pattern category.

## Process Objects

Here we show an example of **Wait Chain** (page 1092) involving process objects. This variation is similar to threads waiting for thread objects (page 1128). When looking at **Stack Trace Collection** (page 943) from a complete memory dump file, we see that several threads in a set of processes are blocked in ALPC **Wait Chain** (page 1097):

```

THREAD ffffffa80110b8700 Cid 12f8.1328 Teb: 000000007ef9a000 Win32Thread: 0000000000000000 WAIT:
(WrLpcReply) UserMode Non-Alertable
    ffffffa80110b8a90 Semaphore Limit 0x1
Waiting for reply to ALPC Message fffff8801c7096e0 : queued at port ffffffa8010c9d9a0 : owned by process
ffffffa80109c8c10
Not impersonating
DeviceMap          fffff880097ce5e0
Owning Process    ffffffa80110ad510      Image:       ProcessA.exe
Attached Process   N/A           Image:       N/A
Wait Start TickCount 14004580      Ticks: 62149 (0:00:16:09.530)
Context Switch Count 25100
UserTime           00:00:00.421
KernelTime         00:00:00.218
Win32 Start Address 0x0000000074ca29e1
Stack Init ffffffa6003bc4db0 Current ffffffa6003bc4670
Base ffffffa6003bc5000 Limit ffffffa6003bbf000 Call 0
Priority 8 BasePriority 8 PriorityDecrement 0 IoPriority 2 PagePriority 5
Child-SP          RetAddr        Call Site
ffffffa60`03bc46b0 ffffff800`01cba0fa nt!KiSwapContext+0x7f
ffffffa60`03bc47f0 ffffff800`01caeab nt!KiSwapThread+0x13a
ffffffa60`03bc4860 ffffff800`01ce4e72 nt!KeWaitForSingleObject+0x2cb
ffffffa60`03bc48f0 ffffff800`01f32f34 nt!AlpcpSignalAndWait+0x92
ffffffa60`03bc4980 ffffff800`01f2f9c6 nt!AlpcpReceiveSynchronousReply+0x44
ffffffa60`03bc49e0 ffffff800`01f1f52f nt!AlpcpProcessSynchronousRequest+0x24f
ffffffa60`03bc4b00 ffffff800`01cb7973 nt!NtAlpcSendWaitReceivePort+0x19f
ffffffa60`03bc4bb0 00000000`7713756a nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ ffffffa60`03bc4c20)
00000000`016ee5b8 00000000`74f9993f ntdll!ZwAlpcSendWaitReceivePort+0xa
00000000`016ee5c0 00000000`74f8a996 wow64!whNtAlpcSendWaitReceivePort+0x5f
00000000`016ee610 00000000`75183688 wow64!Wow64SystemServiceEx+0xca
00000000`016eeeec0 00000000`74f8ab46 wow64cpu!ServiceNoTurbo+0x28
00000000`016eef50 00000000`74f8a14c wow64!RunCpuSimulation+0xa
00000000`016eef80 00000000`771605a8 wow64!Wow64LdrpInitialize+0x4b4
00000000`016ef4e0 00000000`771168de ntdll! ?? ::FNODOBFM::`string'+0x2aa1
00000000`016ef590 00000000`00000000 ntdll!LdrInitializeThunk+0xe

1: kd> !alpc /m fffff8801c7096e0

Message @ fffff8801c7096e0
  MessageID      : 0x263C (9788)
  CallbackID     : 0x29F2A02 (43985410)
  SequenceNumber : 0x000009FE (2558)
  Type          : LPC_REQUEST
  DataLength     : 0x0058 (88)
  TotalLength    : 0x0080 (128)
  Canceled      : No
  Release        : No
  ReplyWaitReply : No
  Continuation   : Yes

```

OwnerPort	:	fffffa8015128040 [ALPC_CLIENT_COMMUNICATION_PORT]
WaitingThread	:	fffffa80110b8700
QueueType	:	ALPC_MSGQUEUE_PENDING
QueuePort	:	fffffa8010c9d9a0 [ALPC_CONNECTION_PORT]
QueuePortOwnerProcess	:	fffffa80109c8c10 (ProcessB.exe)
ServerThread	:	fffffa8013b87bb0
QuotaCharged	:	No
CancelQueuePort	:	0000000000000000
CancelSequencePort	:	0000000000000000
CancelSequenceNumber	:	0x00000000 (0)
ClientContext	:	000000009b49208
ServerContext	:	0000000000000000
PortContext	:	00000000280f0d0
CancelPortContext	:	0000000000000000
SecurityData	:	0000000000000000
View	:	0000000000000000

If we look at a process fffffa80109c8c10 and its thread fffffa8013b87bb0 we would see that it is blocked by some kind of a lock as well:

```

THREAD fffffa8013b87bb0 Cid 0358.2c60 Peb: 000007fffff7e000 Win32Thread: 0000000000000000 WAIT:
(UserRequest) UserMode Non-Alertable
    fffffa8010bca370 Semaphore Limit 0x7fffffff
    fffffa8013b87c68 NotificationTimer
Impersonation token: ffffff8801e614060 (Level Impersonation)
DeviceMap          ffffff880097ce5e0
Owning Process     fffffa80109c8c10      Image:      ProcessB.exe
Attached Process   N/A           Image:      N/A
Wait Start TickCount 14004580      Ticks: 62149 (0:00:16:09.530)
Context Switch Count 134
UserTime           00:00:00.000
KernelTime         00:00:00.000
Win32 Start Address RPCRT4!ThreadStartRoutine (0x000007feff267780)
Stack Init fffffa6035a1fdb0 Current fffffa6035a1f940
Base fffffa6035a20000 Limit fffffa6035a1a000 Call 0
Priority 11 BasePriority 10 PriorityDecrement 0 IoPriority 2 PagePriority 5
Child-SP           RetAddr        Call Site
fffffa60`35a1f980 ffffff800`01cba0fa nt!KiSwapContext+0x7f
fffffa60`35a1fac0 ffffff800`01caedab nt!KiSwapThread+0x13a
fffffa60`35a1fb30 ffffff800`01f1d608 nt!KeWaitForSingleObject+0x2cb
fffffa60`35a1fbco ffffff800`01cb7973 nt!NtWaitForSingleObject+0x98
fffffa60`35a1fc20 00000000`77136d5a nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ fffffa60`35a1fc20)
00000000`0486ec28 00000000`770f559f ntdll!ZwWaitForSingleObject+0xa
00000000`0486ec30 00000000`ff77d4e9 ntdll!RtlAcquireResourceShared+0xd1
00000000`0486ec70 00000000`ff77fb4d ProcessB!Clock::Clock+0x61
[...]
00000000`0486eee0 000007fe`ff261f46 RPCRT4!Invoke+0x65
00000000`0486ef40 000007fe`ff26254d RPCRT4!NdrStubCall12+0x348
00000000`0486f520 000007fe`ff2868d4 RPCRT4!NdrServerCall12+0x1d
00000000`0486f550 000007fe`ff2869f0 RPCRT4!DispatchToStubInCNoAvrf+0x14
00000000`0486f580 000007fe`ff287402 RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x100
00000000`0486f670 000007fe`ff287080 RPCRT4!LRPC_SCALL::DispatchRequest+0x1c2
00000000`0486f6e0 000007fe`ff2862bb RPCRT4!LRPC_SCALL::HandleRequest+0x200
00000000`0486f800 000007fe`ff285e1a RPCRT4!LRPC_ADDRESS::ProcessIO+0x44a
00000000`0486f920 000007fe`ff267769 RPCRT4!LOADABLE_TRANSPORT::ProcessIOEvents+0x24a

```

```
00000000`0486f9d0 000007fe`ff267714 RPCRT4!ProcessIOEventsWrapper+0x9
00000000`0486fa00 000007fe`ff2677a4 RPCRT4!BaseCachedThreadRoutine+0x94
00000000`0486fa40 00000000`76fdbbe3d RPCRT4!ThreadStartRoutine+0x24
00000000`0486fa70 00000000`77116a51 kernel32!BaseThreadInitThunk+0xd
00000000`0486faa0 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

There are many such threads, and inspection of all threads in the process ffffffa80109c8c10 reveals another thread waiting for an ALPC reply:

```
THREAD ffffffa8010c9b060 Cid 0358.02ac Peb: 000007fffffd3000 Win32Thread: 0000000000000000 WAIT:
(WrLpcReply) UserMode Non-Alertable
fffffa8010c9b3f0 Semaphore Limit 0x1
Waiting for reply to ALPC Message fffff88011994cf0 : queued at port ffffffa8010840360 : owned by
process ffffffa801083e120
Not impersonating
DeviceMap          fffff880000073d0
Owning Process    ffffffa80109c8c10      Image:       ProcessB.exe
Attached Process   N/A           Image:       N/A
Wait Start TickCount 13986969      Ticks: 79760 (0:00:20:44.263)
Context Switch Count 712
UserTime           00:00:00.000
KernelTime         00:00:00.000
Win32 Start Address ntdll!TppWorkerThread (0x0000000077107cb0)
Stack Init ffffffa6004bfbdb0 Current ffffffa6004bf670
Base ffffffa6004bfc000 Limit ffffffa6004bf6000 Call 0
Priority 10 BasePriority 10 PriorityDecrement 0 IoPriority 2 PagePriority 5
Kernel stack not resident.
Child-SP          RetAddr        Call Site
fffffa60`04bfb6b0 fffff800`01cba0fa nt!KiSwapContext+0x7f
fffffa60`04bfb7f0 fffff800`01caedab nt!KiSwapThread+0x13a
fffffa60`04bfb860 fffff800`01ce4e72 nt!KeWaitForSingleObject+0x2cb
fffffa60`04bfb8f0 fffff800`01f32f34 nt!AlpcpSignalAndWait+0x92
fffffa60`04bfb980 fffff800`01f2f9c6 nt!AlpcpReceiveSynchronousReply+0x44
fffffa60`04bfb9e0 fffff800`01f1f52f nt!AlpcpProcessSynchronousRequest+0x24f
fffffa60`04bfb9e0 fffff800`01cb7973 nt!NtAlpcSendWaitReceivePort+0x19f
fffffa60`04bfb9e0 00000000`7713756a nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ ffffffa60`04bfb9e0)
00000000`00c3f2f8 00000000`771872c9 ntdll!ZwAlpcSendWaitReceivePort+0xa
[...]
00000000`00c3f810 00000000`77107fd0 ntdll!RtlTpWorkCallback+0xf2
00000000`00c3f8c0 00000000`76fdbbe3d ntdll!TppWorkerThread+0x3d6
00000000`00c3fb40 00000000`77116a51 kernel32!BaseThreadInitThunk+0xd
00000000`00c3fb70 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

```
1: kd> !alpc /m fffff88011994cf0
```

```
Message @ fffff88011994cf0
MessageID          : 0x033C (828)
CallbackID         : 0x29CEF57 (43839319)
SequenceNumber     : 0x000000D8 (216)
Type               : LPC_REQUEST
DataLength         : 0x000C (12)
TotalLength        : 0x0034 (52)
Canceled           : No
Release             : No
ReplyWaitReply    : No
```

Continuation	:	Yes
OwnerPort	:	fffffa8010c99040 [ALPC_CLIENT_COMMUNICATION_PORT]
WaitingThread	:	fffffa8010c9b060
QueueType	:	ALPC_MSGQUEUE_PENDING
QueuePort	:	fffffa8010840360 [ALPC_CONNECTION_PORT]
QueuePortOwnerProcess	:	fffffa801083e120 (ProcessC.exe)
ServerThread	:	fffffa80109837d0
QuotaCharged	:	No
CancelQueuePort	:	0000000000000000
CancelSequencePort	:	0000000000000000
CancelSequenceNumber	:	0x00000000 (0)
ClientContext	:	0000000000000000
ServerContext	:	0000000000000000
PortContext	:	00000000005f3400
CancelPortContext	:	0000000000000000
SecurityData	:	0000000000000000
View	:	0000000000000000

We see that ProcessC thread fffffa80109837d0 is waiting for a process object fffffa801434cb40:

```

THREAD fffffa80109837d0 Cid 027c.02b0 Peb: 000007fffffdb000 Win32Thread: 0000000000000000 WAIT:
(UserRequest) UserMode Non-Alertable
    fffffa801434cb40 ProcessObject
Not impersonating
DeviceMap          fffff880000073d0
Owning Process    fffffa801083e120      Image:      ProcessC.exe
Attached Process  N/A           Image:      N/A
Wait Start TickCount 13986969      Ticks: 79760 (0:00:20:44.263)
Context Switch Count 520
UserTime           00:00:00.000
KernelTime         00:00:00.062
Win32 Start Address 0x000000004826dcf4
Stack Init fffffa6002547db0 Current fffffa6002547940
Base fffffa6002548000 Limit fffffa6002542000 Call 0
Priority 13 BasePriority 11 PriorityDecrement 0 IoPriority 2 PagePriority 5
Kernel stack not resident.
Child-SP          RetAddr        Call Site
fffffa60`02547980 fffff800`01cba0fa nt!KiSwapContext+0x7f
fffffa60`02547ac0 fffff800`01caedab nt!KiSwapThread+0x13a
fffffa60`02547b30 fffff800`01f1d608 nt!KeWaitForSingleObject+0x2cb
fffffa60`02547bc0 fffff800`01cb7973 nt!NtWaitForSingleObject+0x98
fffffa60`02547c20 00000000`77136d5a nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ fffffa60`02547c20)
00000000`0024f7c8 00000000`4826ea97 ntdll!ZwWaitForSingleObject+0xa
00000000`0024f7d0 00000000`4826ef44 ProcessC!TerminatePID+0xa3
[...]
00000000`0024fc90 00000000`00000000 ntdll!RtlUserThreadStart+0x29

```

When we inspect the process fffffa801434cb40, we see that it has only one thread with many usual **Missing Threads** (page 683). **Blocked Thread** (page 82) stack trace has *DriverA* module code waiting for an event:

```
1: kd> !process ffffffa801434cb40 3f
PROCESS ffffffa801434cb40
SessionId: 1 Cid: a0c8 Peb: 7fffffdc000 ParentCid: 1c08
DirBase: 19c6cc000 ObjectTable: ffffff8801767ee00 HandleCount: 287.
Image: ProcessD.exe
VadRoot ffffffa8021be17d0 Vads 71 Clone 0 Private 955. Modified 1245. Locked 0.
DeviceMap ffffff8800000073d0
Token ffffff880187cb3c0
ElapsedTime 00:49:23.432
UserTime 00:00:00.686
KernelTime 00:00:00.904
QuotaPoolUsage[PagedPool] 208080
QuotaPoolUsage[NonPagedPool] 6720
Working Set Sizes (now,min,max) (2620, 50, 345) (10480KB, 200KB, 1380KB)
PeakWorkingSetSize 3136
VirtualSize 101 Mb
PeakVirtualSize 222 Mb
PageFaultCount 13495
MemoryPriority BACKGROUND
BasePriority 13
CommitCharge 1154
```

[...]

```
THREAD ffffffa8012249b30 Cid a0c8.31b4 Peb: 0000000000000000 Win32Thread: 0000000000000000 WAIT:
(Executive) KernelMode Non-Alertable
ffffffa801180a6a0 SynchronizationEvent
Not impersonating
DeviceMap ffffff8800000073d0
Owning Process ffffffa801434cb40 Image: ProcessD.exe
Attached Process N/A Image: N/A
Wait Start TickCount 13986969 Ticks: 79760 (0:00:20:44.263)
Context Switch Count 97
UserTime 00:00:00.000
KernelTime 00:00:00.000
Win32 Start Address DllA (0xfffffff96000eeada0)
Stack Init ffffffa601b841db0 Current ffffffa601b841960
Base ffffffa601b842000 Limit ffffffa601b83c000 Call 0
Priority 13 BasePriority 13 PriorityDecrement 0 IoPriority 2 PagePriority 5
Child-SP RetAddr Call Site
ffffffa60`1b8419a0 ffffff800`01cba0fa nt!KiSwapContext+0x7f
ffffffa60`1b841ae0 ffffff800`01caedab nt!KiSwapThread+0x13a
ffffffa60`1b841b50 ffffff960`00eeeb281 nt!KeWaitForSingleObject+0x2cb
ffffffa60`1b841c20 ffffff800`01ec7bc7 DriverA+0x4b281
ffffffa60`1b841d50 ffffff800`01cf65a6 nt!PspSystemThreadStartup+0x57
ffffffa60`1b841d80 00000000`00000000 nt!KiStartSystemThread+0x16
```

We, therefore, recommend contacting the vendor of *DriverA* component.

## Pushlocks

Here we provide examples of threads waiting for pushlocks<sup>232</sup> as they are not normally seen in crash dumps:

```

THREAD ffffffa80033b5b50 Cid 0004.0030 Teb: 0000000000000000 Win32Thread: 0000000000000000 WAIT:
(WrPushLock) KernelMode Non-Alertable
fffff880021d9750 SynchronizationEvent
Not impersonating
DeviceMap          fffff8a0000088f0
Owning Process    ffffffa80033879e0      Image:       System
Attached Process   ffffffa800439c620      Image:       AppA.exe
Wait Start TickCount 30819           Ticks: 14746574 (2:15:54:08.028)
Context Switch Count 2800
UserTime           00:00:00.000
KernelTime         00:00:00.374
Win32 Start Address nt!ExpWorkerThread (0xfffffff8000189e530)
Stack Init fffff880021d9db0 Current fffff880021d9470
Base fffff880021da000 Limit fffff880021d4000 Call 0
Priority 12 BasePriority 12 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
Child-SP          RetAddr          Call Site
fffff880`021d94b0 fffff800`0188aa32 nt!KiSwapContext+0x7a
fffff880`021d95f0 fffff800`0189bd8f nt!KiCommitThreadWait+0x1d2
fffff880`021d9680 fffff800`018c4bf8 nt!KeWaitForSingleObject+0x19f
fffff880`021d9720 fffff800`01c2915d nt!ExfAcquirePushLockShared+0x138
fffff880`021d97a0 fffff800`01c6da31 nt!MmEnumerateAndReferenceImages+0x6d
[...]
fffff880`021d9cb0 fffff800`01b2be5a nt!ExpWorkerThread+0x111
fffff880`021d9d40 fffff800`01885d26 nt!PspSystemThreadStartup+0x5a
fffff880`021d9d80 00000000`00000000 nt!KxStartSystemThread+0x16

```

<sup>232</sup> <http://blogs.msdn.com/b/ntdebugging/archive/2009/09/02/push-locks-what-are-they.aspx>

```
THREAD ffffffa8003c9d600 Cid 0004.00ac Teb: 0000000000000000 Win32Thread: 0000000000000000 WAIT:  
(WrPushLock) KernelMode Non-Alertable  
fffff880023d1b30 SynchronizationEvent  
Not impersonating  
DeviceMap fffff8a0000088f0  
Owning Process ffffffa80033879e0 Image: System  
Attached Process N/A Image: N/A  
Wait Start TickCount 177686 Ticks: 14599707 (2:15:15:56.888)  
Context Switch Count 1590  
UserTime 00:00:00.000  
KernelTime 00:00:00.124  
Win32 Start Address 0xfffffff80001bac754  
Stack Init fffff880023d1db0 Current fffff880023d1850  
Base fffff880023d2000 Limit fffff880023cc000 Call 0  
Priority 15 BasePriority 15 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5  
Child-SP RetAddr Call Site  
fffff880`023d1890 fffff800`0188aa32 nt!KiSwapContext+0x7a  
fffff880`023d19d0 fffff800`0189bd8f nt!KiCommitThreadWait+0x1d2  
fffff880`023d1a60 fffff800`01886183 nt!KeWaitForSingleObject+0x19f  
fffff880`023d1b00 fffff800`01cd9982 nt!ExfAcquirePushLockExclusive+0x188  
[...]  
fffff880`023d1d40 fffff800`01885d26 nt!PspSystemThreadStartup+0x5a  
fffff880`023d1d80 00000000`00000000 nt!KxStartSystemThread+0x16
```

## RPC

In addition to LPC / ALPC **Wait Chains** (page 1097), we can also see RPC chains in complete memory dumps and even mixed (A)LPC / RPC chains. How to distinguish RPC from (A)LPC (and RPC over LPC) threads? Here's a fragment from an RPC over LPC thread (they also have "waiting for ..." or "working on ..." strings in THREAD output):

```
f50e4c20 80833491 nt!KiSwapContext+0x26
f50e4c4c 80829a82 nt!KiSwapThread+0x2e5
f50e4c94 8091ecf2 nt!KeWaitForSingleObject+0x346
f50e4d50 808897cc nt!NtRequestWaitReplyPort+0x776
f50e4d50 7c82860c nt!KiFastCallEntry+0xfc (TrapFrame @ f50e4d64)
00e5f2b8 7c827899 ntdll!KiFastSystemCallRet
00e5f2bc 77c80a6e ntdll!ZwRequestWaitReplyPort+0xc
00e5f308 77c7fcf0 RPCRT4!LRPC__CALL::SendReceive+0x230
00e5f314 77c80673 RPCRT4!I_RpcSendReceive+0x24
00e5f328 77ce315a RPCRT4!NdrSendReceive+0x2b
00e5f710 771f40c4 RPCRT4!NdrClientCall2+0x22e
```

Here's the thread stack of an RPC waiting thread (the connection was over a pipe):

```
THREAD 8a4b7320 Cid 0208.0220 Peb: 7ffa4000 Win32Thread: bc3eaea8 WAIT: (Unknown) UserMode Non-Alertable
      8bc379f8 NotificationEvent
IRP List:
      891879a8: (0006,0094) Flags: 00000800 Mdl: 00000000
Not impersonating
DeviceMap          e1002270
Owning Process    8a5c8828   Image:       ApplicationA.exe
Attached Process   N/A        Image:       N/A
Wait Start TickCount 3044574   Ticks: 37746 (0:00:09:49.781)
Context Switch Count 54673     LargeStack
UserTime           00:00:00.015
KernelTime         00:00:00.046
Win32 Start Address MSVCR90!_threadstartex (0x7854345e)
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init f473b000 Current f473ac60 Base f473b000 Limit f4737000 Call 0
Priority 11 BasePriority 10 PriorityDecrement 0
ChildEBP RetAddr
f473ac78 80833491 nt!KiSwapContext+0x26
f473aca4 80829a82 nt!KiSwapThread+0x2e5
f473acec 80938dea nt!KeWaitForSingleObject+0x346
f473ad50 808897cc nt!NtWaitForSingleObject+0x9a
f473ad50 7c82860c nt!KiFastCallEntry+0xfc (TrapFrame @ f473ad64)
0155f0f4 7c827d29 ntdll!KiFastSystemCallRet
0155f0f8 77e61d1e ntdll!ZwWaitForSingleObject+0xc
0155f168 77c6a85f kernel32!WaitForSingleObjectEx+0xac
0155f184 77c69bf7 RPCRT4!UTIL_WaitForSyncIO+0x20
0155f1a8 77c6a935 RPCRT4!UTIL_GetOverlappedResultEx+0x1d
0155f1c4 77c6a8f3 RPCRT4!UTIL_GetOverlappedResult+0x17
0155f1e4 77c6944f RPCRT4!NMP_SyncSendRecv+0x73
0155f20c 77c69667 RPCRT4!OSF_CCONNECTION::TransSendReceive+0x7d
0155f294 77c695d4 RPCRT4!OSF_CCONNECTION::SendFragment+0x2ae
```

```

0155f2ec 77c6977a RPCRT4!OSF_CCALL::SendNextFragment+0x1e2
0155f334 77c699f2 RPCRT4!OSF_CCALL::FastSendReceive+0x148
0155f350 77c69975 RPCRT4!OSF_CCALL::SendReceiveHelper+0x5b
0155f380 77c7fcf0 RPCRT4!OSF_CCALL::SendReceive+0x41
0155f38c 77c80673 RPCRT4!I_RpcSendReceive+0x24
0155f3a0 77ce315a RPCRT4!NdrSendReceive+0x2b
0155f788 7d1fa0b1 RPCRT4!NdrClientCall12+0x22e
[...]
0155ffac 785434c7 MSVCR90!_callthreadstartex+0x1b
0155ffb8 77e6482f MSVCR90!_threadstartex+0x69
0155ffec 00000000 kernel32!BaseThreadStart+0x34

```

Here's the endpoint thread stack in the RPC service processing the client call:

```

THREAD 8a631d80 Cid 0244.0290 Peb: 7ffd4000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Non-
Alertable
 8a6354d8 SynchronizationEvent
IRP List:
 882d0008: (0006,0094) Flags: 00000900 Mdl: 00000000
 8811c450: (0006,0094) Flags: 00000900 Mdl: 00000000
 8a4d1b28: (0006,0190) Flags: 00000000 Mdl: 8a4d2e40
 8a634188: (0006,0094) Flags: 00000800 Mdl: 00000000
Not impersonating
DeviceMap          e1002270
Owning Process     8a5b3ac8    Image:       ServiceB.exe
Attached Process   N/A         Image:       N/A
Wait Start TickCount 3041752   Ticks: 40568 (0:00:10:33.875)
Context Switch Count 36194
UserTime           00:00:00.562
KernelTime         00:00:01.093
Win32 Start Address RPCRT4!ThreadStartRoutine (0x77c7b0f5)
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init f5f3e000 Current f5f3dc60 Base f5f3e000 Limit f5f3b000 Call 0
Priority 12 BasePriority 10 PriorityDecrement 0
ChildEBP RetAddr
f5f3dc78 80833491 nt!KiSwapContext+0x26
f5f3dca4 80829a82 nt!KiSwapThread+0x2e5
f5f3dcec 80938dea nt!KeWaitForSingleObject+0x346
f5f3dd50 808897cc nt!NtWaitForSingleObject+0x9a
f5f3dd50 7c82860c nt!KiFastCallEntry+0xfc (TrapFrame @ f5f3dd64)
00def83c 7c827d29 ntdll!KiFastSystemCallRet
00def840 7c83d266 ntdll!ZwWaitForSingleObject+0xc
00def87c 7c83d2b1 ntdll!RtlpWaitOnCriticalSection+0x1a3
00def89c 4ab773ea ntdll!RtlEnterCriticalSection+0xa8
00def8ac 4ab78726 ServiceB!AcquireLock+0x7c
[...]
00def944 77ce33e1 RPCRT4!Invoke+0x30
00defd44 77ce35c4 RPCRT4!NdrStubCall12+0x299
00defd60 77c7ff7a RPCRT4!NdrServerCall12+0x19
00defd94 77c8042d RPCRT4!DispatchToStubInCNoAvrf+0x38
00defde8 77c80353 RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x11f
00defe0c 77c68e0d RPCRT4!RPC_INTERFACE::DispatchToStub+0xa3
00defe40 77c68cb3 RPCRT4!OSF_SCALL::DispatchHelper+0x149
00defe54 77c68c2b RPCRT4!OSF_SCALL::DispatchRpCCall+0x10d
00defe84 77c68b5e RPCRT4!OSF_SCALL::ProcessReceivedPDU+0x57f
00defea4 77c6e8db RPCRT4!OSF_SCALL::BeginRpcCall+0x194

```

```
00deff04 77c6e7b4 RPCRT4!OSF_SCONNECTION::ProcessReceiveComplete+0x435
00deff18 77c7b799 RPCRT4!ProcessConnectionServerReceivedEvent+0x21
00deff84 77c7b9b5 RPCRT4!LOADABLE_TRANSPORT::ProcessIOEvents+0x1b8
00deff8c 77c8872d RPCRT4!ProcessIOEventsWrapper+0xd
00deffac 77c7b110 RPCRT4!BaseCachedThreadRoutine+0x9d
00deffb8 77e6482f RPCRT4!ThreadStartRoutine+0x1b
00deffec 00000000 kernel32!BaseThreadStart+0x34
```

We also see that the latter thread is waiting for a critical section, so we have an example of a mixed wait chain here as well. Another example is an RPC over LPC server thread that is also an RPC client thread:

```
THREAD 8989f020 Cid 0170.1cfcc Teb: 7ff8c000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Non-Alertable
89a1c368 NotificationEvent
IRP List:
887aac68: (0006,0094) Flags: 00000800 Mdl: 00000000
Not impersonating
DeviceMap e1002270
Owning Process 8a056b80 Image: ServiceC.exe
Attached Process N/A Image: N/A
Wait Start TickCount 3075354 Ticks: 6966 (0:00:01:48.843)
Context Switch Count 2521
UserTime 00:00:00.031
KernelTime 00:00:00.015
Win32 Start Address 0x00750d91
LPC Server thread working on message Id 750d91
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init f26da000 Current f26d9c60 Base f26da000 Limit f26d7000 Call 0
Priority 8 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr
f26d9c78 80833491 nt!KiSwapContext+0x26
f26d9ca4 80829a82 nt!KiSwapThread+0x2e5
f26d9cec 80938dea nt!KeWaitForSingleObject+0x346
f26d9d50 808897cc nt!NtWaitForSingleObject+0x9a
f26d9d50 7c82860c nt!KiFastCallEntry+0xfc (TrapFrame @ f26d9d64)
03e9efa8 7c827d29 nt!NtWaitForSingleObject+0x9a
03e9efac 77e61d1e nt!ZwWaitForSingleObject+0xc
03e9f01c 77c6a85f kernel32!WaitForSingleObjectEx+0xac
03e9f038 77c69bf7 RPCRT4!UTIL_WaitForSyncIO+0x20
03e9f05c 77c6a935 RPCRT4!UTIL_GetOverlappedResultEx+0x1d
03e9f078 77c6a8f3 RPCRT4!UTIL_GetOverlappedResult+0x17
03e9f098 77c6944f RPCRT4!NMP_SyncSendRecv+0x73
03e9f0c0 77c69667 RPCRT4!OSF_CCONNECTION::TransSendReceive+0x7d
03e9f148 77c695d4 RPCRT4!OSF_CCONNECTION::SendFragment+0x2ae
03e9f1a0 77c6977a RPCRT4!OSF_CCALL::SendNextFragment+0x1e2
03e9f1f8 77c699f2 RPCRT4!OSF_CCALL::FastSendReceive+0x148
03e9f214 77c69975 RPCRT4!OSF_CCALL::SendReceiveHelper+0x5b
03e9f244 77c7fcf0 RPCRT4!OSF_CCALL::SendReceive+0x41
03e9f250 77c80673 RPCRT4!I_RpcSendReceive+0x24
03e9f264 77ce315a RPCRT4!NdrSendReceive+0x2b
03e9f64c 7d1fa0b1 RPCRT4!NdrClientCall12+0x22e
03e9f8ac 7654fa50 ServiceC!QueryInformation+0x801
[...]
03e9f8f8 77ce33e1 RPCRT4!Invoke+0x30
03e9fcf8 77ce35c4 RPCRT4!NdrStubCall12+0x299
```

```
03e9fd14 77c7ff7a RPCRT4!NdrServerCall12+0x19
03e9fd48 77c8042d RPCRT4!DispatchToStubInCNoAvrf+0x38
03e9fd9c 77c80353 RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x11f
03e9fdc0 77c811dc RPCRT4!RPC_INTERFACE::DispatchToStub+0xa3
03e9fdfc 77c812f0 RPCRT4!LRPC_SCALL::DealWithRequestMessage+0x42c
03e9fe20 77c88678 RPCRT4!LRPC_ADDRESS::DealWithLRPCRequest+0x127
03e9ff84 77c88792 RPCRT4!LRPC_ADDRESS::ReceiveLotsaCalls+0x430
03e9ff8c 77c8872d RPCRT4!RecvLotsaCallsWrapper+0xd
03e9ffac 77c7b110 RPCRT4!BaseCachedThreadRoutine+0x9d
03e9ffb8 77e6482f RPCRT4!ThreadStartRoutine+0x1b
03e9ffec 00000000 kernel32!BaseThreadStart+0x34
```

## RTL\_RESOURCE

Here we provide another variant of **Wait Chain** pattern (page 1092) related to *RtlAcquireResourceShared* and *RtlAcquireResourceExclusive* calls:

```
THREAD ffffffa8052d66060 Cid 03c0.3240 Tcb: 0000007fffff90000 Win32Thread: 0000000000000000 WAIT:
(UserRequest) UserMode Non-Alertable
ffffffa804a79ad50 Semaphore Limit 0x7fffffff
Impersonation token: fffff8a01b19d060 (Level Impersonation)
DeviceMap fffff8a0035276c0
Owning Process ffffffa804a16b260 Image: lsm.exe
Attached Process N/A Image: N/A
Wait Start TickCount 73343513 Ticks: 1460259 (0:06:20:16.546)
Context Switch Count 17 IdealProcessor: 1
UserTime 00:00:00.000
KernelTime 00:00:00.000
Win32 Start Address ntdll!TppWorkerThread (0x000000007735fbf0)
Stack Init fffff8800e870db0 Current fffff8800e870900
Base fffff8800e871000 Limit fffff8800e86b000 Call 0
Priority 9 BasePriority 8 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
Kernel stack not resident.

Child-SP RetAddr Call Site
fffff880`0e870940 fffff800`01c76972 nt!KiSwapContext+0x7a
fffff880`0e870a80 fffff800`01c87d8f nt!KiCommitThreadWait+0x1d2
fffff880`0e870b10 fffff800`01f7b2be nt!KeWaitForSingleObject+0x19f
fffff880`0e870bb0 fffff800`01c801d3 nt!NtWaitForSingleObject+0xde
fffff880`0e870c20 00000000`773912fa nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ fffff880`0e870c20)
00000000`022ae6c8 00000000`773470b4 ntdll!NtWaitForSingleObject+0xa
00000000`022ae6d0 00000000`ff4013a3 ntdll!RtlAcquireResourceShared+0xd0
00000000`022ae710 00000000`ff401675 lsm!CAutoSharedLock::CAutoSharedLock+0x61
00000000`022ae7e0 00000000`ff402c68 lsm!CTSSession::getTerminal+0x21
00000000`022ae820 000007fe`fd8bff85 lsm!RpcGetEnumResult+0x202
00000000`022ae980 000007fe`fd8b4de2 RPCRT4!Invoke+0x65
00000000`022ae9e0 000007fe`fd8b17bd RPCRT4!NdrStubCall12+0x32a
00000000`022af000 000007fe`fd8b3254 RPCRT4!NdrServerCall12+0x1d
00000000`022af030 000007fe`fd8b33b6 RPCRT4!DispatchToStubInCNoAvrf+0x14
00000000`022af060 000007fe`fd8b3aa9 RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x146
00000000`022af180 000007fe`fd8b375d RPCRT4!LRPC_SCALL::DispatchRequest+0x149
00000000`022af260 000007fe`fd8d09ff RPCRT4!LRPC_SCALL::HandleRequest+0x20d
00000000`022af390 000007fe`fd8d05b5 RPCRT4!LRPC_ADDRESS::ProcessIO+0x3bf
00000000`022af4d0 00000000`7735b6bb RPCRT4!LrpclIoComplete+0xa5
00000000`022af560 00000000`7735ff2f ntdll!TppAlpcpExecuteCallback+0x26b
00000000`022af5f0 00000000`7713652d ntdll!TppWorkerThread+0x3f8
00000000`022af8f0 00000000`7736c541 kernel32!BaseThreadInitThunk+0xd
00000000`022af920 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

These functions are undocumented, but ReactOS source code shows they all take a pointer to *RTL\_RESOURCE* structure<sup>233</sup> that has handles to shared and exclusive semaphores:

<sup>233</sup> [https://doxygen.reactos.org/df/d19/struct\\_\\_RTL\\_\\_RESOURCE.html](https://doxygen.reactos.org/df/d19/struct__RTL__RESOURCE.html)

RTL_CRITICAL_SECTION	Lock
HANDLE	SharedSemaphore
ULONG	SharedWaiters
HANDLE	ExclusiveSemaphore
ULONG	ExclusiveWaiters
LONG	NumberActive
HANDLE	OwningThread
ULONG	TimeoutBoost
PVOID	DebugInfo

To double check that we disassemble *RtlAcquireResourceShared* and check the return address from *NtWaitForSingleObject* call (00000000`773470b4):

```
0: kd> .thread /r /p ffffffa8052d66060
Implicit thread is now ffffffa80`52d66060
Implicit process is now ffffffa80`4a16b260
Loading User Symbols
.....
0: kd> uf ntdll!RtlAcquireResourceShared
[...]
ntdll!RtlAcquireResourceShared+0xc2:
00000000`773470a6 488b4b28      mov rcx,qword ptr [rbx+28h]
00000000`773470aa 4c8bc6      mov r8,rsi
00000000`773470ad 33d2      xor edx,edx
00000000`773470af e83ca20400    call ntdll!NtWaitForSingleObject (00000000`773912f0)
00000000`773470b4 3d02010000    cmp eax,102h
00000000`773470b9 0f8402800600 je ntdll! ?? ::FNODOBFM::`string'+0x12629 (00000000`773af0c1)
[...]
ntdll!RtlAcquireResourceShared:
00000000`77352af0 48895c2420    mov qword ptr [rsp+20h],rbx
00000000`77352af5 57      push rdi
00000000`77352af6 4883ec30    sub rsp,30h
00000000`77352afa 448b4944    mov r9d,dword ptr [rcx+44h]
00000000`77352afe 0fb6fa      movzx edi,dl
00000000`77352b01 488bd9      mov rbx,rcx
00000000`77352b04 4585c9      test r9d,r9d
00000000`77352b07 0f88a7000000 js ntdll!RtlAcquireResourceShared+0x65 (00000000`77352bb4)
[...]
```

We see the handle is taken from [RBX+28], and we see that RBX was saved at the function prolog, and then the value of RCX was assigned to RBX. RCX as the first calling convention parameter should be a pointer to RTL\_RESOURCE that has RTL\_CRITICAL\_SECTION as the first member, and its size is 0x28:

```
0: kd> dt ntdll!_RTL_CRITICAL_SECTION
ntdll!_RTL_CRITICAL_SECTION
+0x000 DebugInfo      : Ptr64 _RTL_CRITICAL_SECTION_DEBUG
+0x008 LockCount      : Int4B
+0x00c RecursionCount : Int4B
+0x010 OwningThread   : Ptr64 Void
+0x018 LockSemaphore  : Ptr64 Void
+0x020 SpinCount       : Uint8B
```

Therefore [RBX+28] contains *SharedSemaphore* field that is assigned to RCX as a first parameter to *NtWaitForSingleObject*. The similar fragment of *RtlAcquireResourceExclusive* has [RBX+38] which 0x10 further than 0x28 and corresponds to *ExclusiveSemaphore* handle field:

```
ntdll!RtlAcquireResourceExclusive+0xd2:
00000000`770c2a12 488b4b38      mov    rcx,qword ptr [rbx+38h]
00000000`770c2a16 4c8bc6      mov    r8,rsi
00000000`770c2a19 33d2      xor    edx,edx
00000000`770c2a1b e8d0e80400    call   ntdll!NtWaitForSingleObject (00000000`771112f0)
00000000`770c2a20 3d02010000    cmp    eax,102h
00000000`770c2a25 0f8401c60600    je     ntdll! ?? ::FNODOBFM::`string'+0x12591 (00000000`7712f02c)
```

So we just need to know the value of RBX and dump the structure to find *OwningThread* field. We can either calculate it from RSP or use /c switch with .frame command:

```
0: kd> kn
*** Stack trace for last set context - .thread/.cxr resets it
# Child-SP          RetAddr           Call Site
00 ffffff880`0e870940 ffffff800`01c76972 nt!KiSwapContext+0x7a
01 ffffff880`0e870a80 ffffff800`01c87d8f nt!KiCommitThreadWait+0x1d2
02 ffffff880`0e870b10 ffffff800`01f7b2be nt!KeWaitForSingleObject+0x19f
03 ffffff880`0e870bb0 ffffff800`01c801d3 nt!NtWaitForSingleObject+0xde
04 ffffff880`0e870c20 00000000`773912fa nt!KiSystemServiceCopyEnd+0x13
05 00000000`022ae6c8 00000000`773470b4 ntdll!NtWaitForSingleObject+0xa
06 00000000`022ae6d0 00000000`ff4013a3 ntdll!RtlAcquireResourceShared+0xd0
07 00000000`022ae710 00000000`ff401675 1sm!CAutoSharedLock::CAutoSharedLock+0x61
08 00000000`022ae7e0 00000000`ff402c68 1sm!CTSSession::getTerminal+0x21
09 00000000`022ae820 000007fe`fd8bfff85 1sm!RpcGetEnumResult+0x202
0a 00000000`022ae980 000007fe`fd8b4de2 RPCRT4!Invoke+0x65
0b 00000000`022ae9e0 000007fe`fd8b17bd RPCRT4!NdrStubCall12+0x32a
0c 00000000`022af000 000007fe`fd8b3254 RPCRT4!NdrServerCall12+0x1d
0d 00000000`022af030 000007fe`fd8b33b6 RPCRT4!DispatchToStubInCNoAvrf+0x14
0e 00000000`022af060 000007fe`fd8b3aa9 RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x146
0f 00000000`022af180 000007fe`fd8b375d RPCRT4!LRPC_SCALL::DispatchRequest+0x149
10 00000000`022af260 000007fe`fd8d09ff RPCRT4!LRPC_SCALL::HandleRequest+0x20d
11 00000000`022af390 000007fe`fd8d05b5 RPCRT4!LRPC_ADDRESS::ProcessIO+0x3bf
12 00000000`022af4d0 00000000`7735b6bb RPCRT4!LrpclIoComplete+0xa5
13 00000000`022af560 00000000`7735ff2f ntdll!TppAlpcpExecuteCallback+0x26b
14 00000000`022af5f0 00000000`7713652d ntdll!TppWorkerThread+0x3f8
15 00000000`022af8f0 00000000`7736c541 kernel32!BaseThreadInitThunk+0xd
16 00000000`022af920 00000000`00000000 ntdll!RtlUserThreadStart+0x1d

0: kd> .frame /c 6
06 00000000`022ae6d0 00000000`ff4013a3 ntdll!RtlAcquireResourceShared+0xd0
rax=0000000000000000  rbx=00000000023ac128 rcx=0000000000000000
rdx=0000000000000000  rsi=00000000077472410 rdi=0000000000000001
rip=000000000773470b4  rsp=00000000022ae6d0 rbp=0000000000000000
r8=0000000000000000  r9=0000000000000000 r10=0000000000000000
r11=0000000000000000  r12=29406b2a1a85bd43 r13=0000000000000009
r14=000000000000000c  r15=00000000022aef20
iopl=0             nv up di pl nz na pe nc
cs=0000  ss=0000  ds=0000  es=0000  fs=0000  gs=0000          efl=00000000
ntdll!RtlAcquireResourceShared+0xd0:
00000000`773470b4 3d02010000    cmp    eax,102h
```

```
0: kd> dp rbx+28 L10
00000000`023ac150  00000000`00001244 00000000`000001b5
00000000`023ac160  00000000`00000f3c  ffffffff`00000000
00000000`023ac170  00000000`000021a0  00000000`00000000
00000000`023ac180  00000000`02735fc0  00000000`00000001
00000000`023ac190  00000000`00000000  01cf07ac`9fa06d27
00000000`023ac1a0  00000000`00000000  00000000`00000000
00000000`023ac1b0  ffffffff`fffffff  00000000`00000000
00000000`023ac1c0  00000000`00000000  00000000`00000000
```

We check all these handles (*OwnerThread* seems comes earlier with *NumberActive* field missing, but that could just differences between the old x86 structure implemented in ReactOS and x64 Windows):

```
0: kd> !handle 00000000`00001244

PROCESS ffffffa804a16b260
SessionId: 0 Cid: 03c0 Peb: 7fffffdc000 ParentCid: 0350
DirBase: 195950000 ObjectTable: fffff8a0032424e0 HandleCount: 5252.
Image: lsm.exe
```

Handle table at fffff8a0032424e0 with 5252 entries in use

```
1244: Object: ffffffa804a79ad50 GrantedAccess: 00100003 Entry: fffff8a022b39910
Object: ffffffa804a79ad50 Type: (fffffa8048fc8790) Semaphore
ObjectHeader: ffffffa804a79ad20 (new version)
HandleCount: 1 PointerCount: 438
```

```
0: kd> !handle 00000000`00000f3c
```

```
PROCESS ffffffa804a16b260
SessionId: 0 Cid: 03c0 Peb: 7fffffdc000 ParentCid: 0350
DirBase: 195950000 ObjectTable: fffff8a0032424e0 HandleCount: 5252.
Image: lsm.exe
```

Handle table at fffff8a0032424e0 with 5252 entries in use

```
0f3c: Object: ffffffa804fa81f60 GrantedAccess: 00100003 Entry: fffff8a02cd3ecf0
Object: ffffffa804fa81f60 Type: (fffffa8048fc8790) Semaphore
ObjectHeader: ffffffa804fa81f30 (new version)
HandleCount: 1 PointerCount: 1
```

```

0: kd> !thread -t 00000000`000021a0 1f
THREAD ffffffa804d5d51b0 Cid 03c0.21a0 Peb: 000007fffff9c000 Win32Thread: 0000000000000000 WAIT:
(WrLpcReply) UserMode Non-Alertable
fffffa804d5d5578 Semaphore Limit 0x1
Waiting for reply to ALPC Message fffff8a02c9a9500 : queued at port fffffa804ac4e7d0 : owned by process
fffffa804adc8730
Not impersonating
DeviceMap fffff8a0000088c0
Owning Process ffffffa804a16b260 Image: lsm.exe
Attached Process N/A Image: N/A
Wait Start TickCount 73337319 Ticks: 1466453 (0:06:21:53.328)
Context Switch Count 69 IdealProcessor: 1
UserTime 00:00:00.000
KernelTime 00:00:00.000
Win32 Start Address ntdll!TppWorkerThread (0x000000007735fbf0)
Stack Init fffff8800aa1fdb0 Current fffff8800aa1f600
Base fffff8800aa20000 Limit fffff8800aa1a000 Call 0
Priority 9 BasePriority 8 UnusualBoost 0 ForegroundBoost 0 IoPriority 2 PagePriority 5
Kernel stack not resident.

Child-SP RetAddr Call Site
fffffa880`0aa1f640 fffff800`01c76972 nt!KiSwapContext+0x7a
fffffa880`0aa1f780 fffff800`01c87d8f nt!KiCommitThreadWait+0x1d2
fffffa880`0aa1f810 fffff800`01ca25af nt!KeWaitForSingleObject+0x19f
fffffa880`0aa1f8b0 fffff800`01f968b6 nt!AlpcpSignalAndWait+0x8f
fffffa880`0aa1f960 fffff800`01f95fb0 nt!AlpcpReceiveSynchronousReply+0x46
fffffa880`0aa1f9c0 fffff800`01f93dab nt!AlpcpProcessSynchronousRequest+0x33d
fffffa880`0aa1fb00 fffff800`01c801d3 nt!NtAlpcSendWaitReceivePort+0x1ab
fffffa880`0aa1fb00 00000000`77391b0a nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ fffff880`0aa1fc20)
00000000`01dddb48 000007fe`fd8c8306 ntdll!ZwAlpcSendWaitReceivePort+0xa
00000000`01dddb50 000007fe`fd8c2a02 RPCRT4!RPC_CCALL::SendReceive+0x156
00000000`01dddc10 000007fe`ff5b28c0 RPCRT4!I_RpcSendReceive+0x42
00000000`01dddc40 000007fe`ff5b282f ole32!ThreadSendReceive+0x40
[d:\w7rtm\com\ole32\com\dcomrem\channelb.cxx @ 5003]
00000000`01dddc90 000007fe`ff5b265b ole32!CRpcChannelBuffer::SwitchAptAndDispatchCall+0xa3
[d:\w7rtm\com\ole32\com\dcomrem\channelb.cxx @ 4454]
00000000`01dddc30 000007fe`ff46daaa ole32!CRpcChannelBuffer::SendReceive2+0x11b
[d:\w7rtm\com\ole32\com\dcomrem\channelb.cxx @ 4074]
00000000`01dddef0 000007fe`ff46da0c ole32!CAptRcpChnl::SendReceive+0x52
[d:\w7rtm\com\ole32\com\dcomrem\callctrl.cxx @ 603]
00000000`01dddfc0 000007fe`ff5b205d ole32!CCtxComChnl::SendReceive+0x68
[d:\w7rtm\com\ole32\com\dcomrem\ctxchnl.cxx @ 734]
00000000`01dde070 000007fe`fd96b949 ole32!NdrExtpProxySendReceive+0x45 [d:\w7rtm\com\rpc\ndrole\proxy.cxx
@ 1932]
00000000`01dde0a0 000007fe`ff5b21d0 RPCRT4!NdrpClientCall3+0x2e2
00000000`01dde360 000007fe`ff46d8a2 ole32!ObjectStublessClient+0x11d
[d:\w7rtm\com\rpc\ndrole\amd64\stblsclt.cxx @ 621]
00000000`01dde6f0 00000000`ff417d26 ole32!ObjectStubless+0x42 [d:\w7rtm\com\rpc\ndrole\amd64\stubless.asm
@ 117]
00000000`01dde740 00000000`ff4186ba lsm!CTSSession::Disconnect+0x3a5
00000000`01dde810 000007fe`fd8bff85 lsm!RpcDisconnect+0x15e
00000000`01dde850 000007fe`fd96b68e RPCRT4!Invoke+0x65
00000000`01dde8a0 000007fe`fd8a92e0 RPCRT4!Ndr64StubWorker+0x61b
00000000`01dee60 000007fe`fd8b3254 RPCRT4!NdrServerCallAll+0x40
00000000`01deeb0 000007fe`fd8b33b6 RPCRT4!DispatchToStubInCNoAvrf+0x14
00000000`01deee0 000007fe`fd8b3aa9 RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x146

```

```
00000000`01ddf000 000007fe`fd8b375d RPCRT4!LRPC_SCALL::DispatchRequest+0x149
00000000`01ddf0e0 000007fe`fd8d09ff RPCRT4!LRPC_SCALL::HandleRequest+0x20d
00000000`01ddf210 000007fe`fd8d05b5 RPCRT4!LRPC_ADDRESS::ProcessIO+0x3bf
00000000`01ddf350 00000000`7735b6bb RPCRT4!LrpcIoComplete+0xa5
00000000`01ddf3e0 00000000`7735ff2f ntdll!TppAlpcpExecuteCallback+0x26b
00000000`01ddf470 00000000`7713652d ntdll!TppWorkerThread+0x3f8
00000000`01ddf770 00000000`7736c541 kernel32!BaseThreadInitThunk+0xd
00000000`01ddf7a0 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

We see the wait chain continues with waiting for an ALPC request.

## Thread Objects

Another example of **Wait Chain** pattern (page 1092) for objects with ownership semantics is seen in the kernel and complete memory dumps where threads are waiting for thread objects. A thread object is a synchronization object whose owner is a thread so they can be easily identified. For example, the main application thread (**Main Thread**, page 614) is waiting for such an object:

```
1: kd> !thread 8818e660 16
THREAD 8818e660 Cid 1890.1c50 Teb: 7ffd000 Win32Thread: b8411008 WAIT: (Unknown) UserMode Non-
Alertable
  87d569d8 Thread
    8818e6d8 NotificationTimer
  Not impersonating
  DeviceMap          e10008d8
  Owning Process     87db5d88      Image:       App.exe
  Wait Start TickCount 299006      Ticks: 255 (0:00:00:03.984)
  Context Switch Count 1208        LargeStack
  UserTime           00:00:00.203
  KernelTime         00:00:00.203
  Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
  Stack Init b42c3000 Current b42c2c60 Base b42c3000 Limit b42be000 Call 0
  Priority 15 BasePriority 15 PriorityDecrement 0
  ChildEBP RetAddr  Args to Child
  b42c2c78 80833e8d 8818e660 8818e708 00000003 nt!KiSwapContext+0x26
  b42c2ca4 80829b74 00000000 b42c2d14 00000000 nt!KiSwapThread+0x2e5
  b42c2cec 8093b034 87d569d8 00000006 00804c01 nt!KeWaitForSingleObject+0x346
  b42c2d50 8088ac4c 000001ec 00000000 b42c2d14 nt!NtWaitForSingleObject+0x9a
  b42c2d50 7c8285ec 000001ec 00000000 b42c2d14 nt!KiFastCallEntry+0xfc
  0006fde4 7c827d0b 77e61d1e 000001ec 00000000 ntdll!KiFastSystemCallRet
  0006fde8 77e61d1e 000001ec 00000000 0006fe2c ntdll!NtWaitForSingleObject+0xc
  0006fe58 77e61c8d 000001ec 00001388 00000000 kernel32!WaitForSingleObjectEx+0xac
  0006fe6c 01039308 000001ec 00001388 00000000 kernel32!WaitForSingleObject+0x12
  0006fe94 010204ac 0007cc00 00000001 00000002 App!WaitForNotifyList+0xf2
  [...]
```

That object is a thread too:

```
THREAD 87d569d8 Cid 1890.1ec0 Teb: 7ffd9000 Win32Thread: b869ba48 WAIT: (Unknown) UserMode Non-
Alertable
  8a1f8870 Thread
```

Therefore, we see that the thread 8818e660 is waiting for another thread 87d569d8 which belongs to the same process with PID 1890 and the thread 87d569d8 itself is waiting for the thread 8a1f8870 that has the following stack trace:

```

1: kd> !thread 8a1f8870 16
THREAD 8a1f8870 Cid 1890.07d8 Peb: 7ff95000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Non-
Alertable
    8a0ce4c0 NotificationEvent
    886f1870 NotificationEvent
Not impersonating
DeviceMap          e10008d8
Owning Process     87db5d88      Image:       App.exe
Wait Start TickCount 292599      Ticks: 6662 (0:00:01:44.093)
Context Switch Count 17
UserTime           00:00:00.000
KernelTime         00:00:00.000
Win32 Start Address Dll!StartMonitoring (0x758217b8)
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init b6d4f000 Current b6d4e900 Base b6d4f000 Limit b6d4c000 Call 0
Priority 14 BasePriority 13 PriorityDecrement 0
ChildEBP RetAddr Args to Child
b6d4e918 80833e8d 8a1f8870 00000002 00140000 nt!KiSwapContext+0x26
b6d4e944 808295ab 8a1f8870 00000002 00000000 nt!KiSwapThread+0x2e5
b6d4e978 8093b290 00000002 b6d4eaac 00000001 nt!KeWaitForMultipleObjects+0x3d7
b6d4ebf4 8093b3f2 00000002 b6d4ec1c 00000001 nt!ObpWaitForMultipleObjects+0x202
b6d4ed48 8088ac4c 00000002 026bfc08 00000001 nt!NtWaitForMultipleObjects+0xc8
b6d4ed48 7c8285ec 00000002 026bfc08 00000001 nt!KiFastCallEntry+0xfc
026bfbb8 7c827cfb 77e6202c 00000002 026bfc08 ntdll!KiFastSystemCallRet
026bfbbc 77e6202c 00000002 026bfc08 00000001 ntdll!NtWaitForMultipleObjects+0xc
026bfc64 77e62fbe 00000002 026bfca4 00000000 kernel32!WaitForMultipleObjectsEx+0x11a
026bfc80 6554a01f 00000002 026bfca4 00000000 kernel32!WaitForMultipleObjects+0x18
026bfcfc 758237a3 cd050002 ffffffff 026bfd4c Dll!GetStatusChange+0x7bf
026bffb8 77e64829 75833120 00000000 00000000 Dll!StartMonitoring+0x14b
026bffc 00000000 758217b8 75833120 00000000 kernel32!BaseThreadStart+0x34

```

The thread 8a1f8870 is waiting for two disjoint notification events, and this is confirmed by dumping *WaitForMultipleObjects* function arguments. Neither of them is in a signaled state<sup>234</sup>, and one is a named event “MyEventObject”:

```

1: kd> dd 026bfc08 12
026bfc08 0000008c 00000084

1: kd> !handle 0000008c
processor number 1, process 87db5d88
PROCESS 87db5d88 SessionId: 4 Cid: 1890 Peb: 7ffdc000 ParentCid: 01d0
    DirBase: cfe438e0 ObjectTable: e178c228 HandleCount: 439.
    Image: App.exe

Handle table at e50d2000 with 439 Entries in use
008c: Object: 8a0ce4c0 GrantedAccess: 001f0003 Entry: e50d2118
Object: 8a0ce4c0 Type: (8b26ec00) Event
    ObjectHeader: 8a0ce4a8 (old version)
    HandleCount: 1 PointerCount: 3

```

<sup>234</sup> Sighaled Objects, Memory Dump Analysis Anthology, Volume 2, page 80

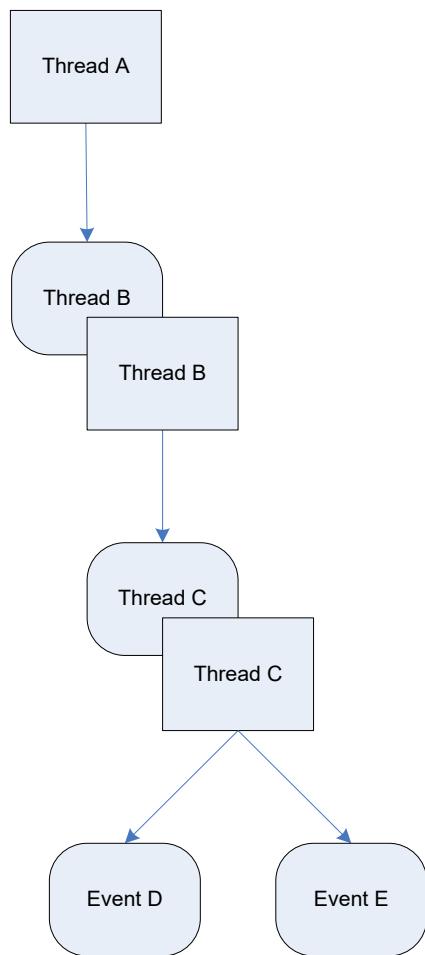
```
1: kd> !handle 00000084
processor number 1, process 87db5d88
PROCESS 87db5d88 SessionId: 4 Cid: 1890 Peb: 7ffdc000 ParentCid: 01d0
    DirBase: cfe438e0 ObjectTable: e178c228 HandleCount: 439.
    Image: App.exe

Handle table at e50d2000 with 439 Entries in use
0084: Object: 886f1870 GrantedAccess: 001f0003 (Inherit) Entry: e50d2108
Object: 886f1870 Type: (8b26ec00) Event
    ObjectHeader: 886f1858 (old version)
        HandleCount: 1 PointerCount: 4
        Directory Object: e43ee320 Name: MyEventObject

1: kd> dt _DISPATCHER_HEADER 8a0ce4c0
+0x000 Type : 0 "
+0x001 Absolute : 0 "
+0x002 Size : 0x4 "
+0x003 Inserted : 0 "
+0x003 DebugActive : 0 "
+0x000 Lock : 262144
+0x004 SignalState : 0
+0x008 WaitListHead : _LIST_ENTRY [ 0x88519d18 - 0x8a1f8918 ]

1: kd> dt _DISPATCHER_HEADER 886f1870
+0x000 Type : 0 "
+0x001 Absolute : 0 "
+0x002 Size : 0x4 "
+0x003 Inserted : 0 "
+0x003 DebugActive : 0 "
+0x000 Lock : 262144
+0x004 SignalState : 0
+0x008 WaitListHead : _LIST_ENTRY [ 0x88519d30 - 0x8a1f8930 ]
```

Here is a diagram showing that **Wait Chain**:



## Window Messaging

This is another variant of the general **Wait Chain** (page 1092) pattern where **Blocked Threads** (page 82) are waiting for synchronous window message calls (*sent* messages). For example, here three threads from different processes are blocked in such a chain (*hWnd* parameters for *SendMessage* calls and associated window procedures are marked with bold italics, bold and underlined):

```
0:000> ~*kbl

. 0 Id: 116c.1174 Suspend: 1 Teb: 7efdd000 Unfrozen
ChildEBP RetAddr Args to Child
0034f83c 76261c01 000a0e54 00000111 00000068 USER32!NtUserMessageCall+0x15
0034f87c 7625cd81 011114d0 00000000 00d41190 USER32!SendMessageWorker+0x5e9
0034f8a0 00fa1256 000a0e54 00000111 00000068 USER32!SendMessageW+0x7f
0034f90c 76256238 00040eb0 00000111 00000068 WCM_A!WndProc+0xc6
0034f938 762568ea 00fa1190 00040eb0 00000111 USER32!InternalCallWinProc+0x23
0034f9b0 76257d31 00000000 00fa1190 00040eb0 USER32!UserCallWinProcCheckWow+0x109
0034fa10 76257dfa 00fa1190 00000000 76257d79 USER32!DispatchMessageWorker+0x3bc
0034fa20 00fa10d3 0034fa3c 0034fae8 00000000 USER32!DispatchMessageW+0xf
0034fa54 00fa14b6 00fa0000 00000000 00571bee WCM_A!wWinMain+0xd3
0034fae8 76493677 7efde000 0034fb34 77399d72 WCM_A!_tmainCRTStartup+0x150
0034faf4 77399d72 7efde000 72afcb2e 00000000 kernel32!BaseThreadInitThunk+0xe
0034fb34 77399d45 00fa1625 7efde000 ffffffff ntdll!_RtlUserThreadStart+0x70
0034fb4c 00000000 00fa1625 7efde000 00000000 ntdll!_RtlUserThreadStart+0x1b

0:000> ~*kbl

. 0 Id: 10dc.e14 Suspend: 1 Teb: 7efdd000 Unfrozen
ChildEBP RetAddr Args to Child
0017f7c4 76261c01 000c0ea4 00000111 00000068 USER32!NtUserMessageCall+0x15
0017f804 7625cd81 00ec3ec0 00000000 012e1190 USER32!SendMessageWorker+0x5e9
0017f828 00d41256 000c0ea4 00000111 00000068 USER32!SendMessageW+0x7f
0017f890 76256238 000a0e54 00000111 00000068 WCM_B!WndProc+0xc6
0017f8bc 762568ea 00d41190 000a0e54 00000111 USER32!InternalCallWinProc+0x23
0017f934 76257177 00000000 00d41190 000a0e54 USER32!UserCallWinProcCheckWow+0x109
0017f990 762572f1 00eb14d0 00000000 00000111 USER32!DispatchClientMessage+0xe0
0017f9cc 773700e6 0017f9e4 00000000 0017fae4 USER32!_fnDWORD+0x2b
0017f9e0 00eb14d0 00000000 00000111 00000068 ntdll!KiUserCallbackDispatcher+0x2e
WARNING: Frame IP not in any known module. Following frames may be wrong.
0017fa20 00d410e0 0017fa48 00000000 00000000 0xeb14d0
0017fa60 00d414b6 00d40000 00000000 00601bee WCM_B!wWinMain+0xe0
0017faf4 76493677 7efde000 0017fb40 77399d72 WCM_B!_tmainCRTStartup+0x150
0017fb00 77399d72 7efde000 728cf6de 00000000 kernel32!BaseThreadInitThunk+0xe
0017fb40 77399d45 00d41625 7efde000 ffffffff ntdll!_RtlUserThreadStart+0x70
0017fb58 00000000 00d41625 7efde000 00000000 ntdll!_RtlUserThreadStart+0x1b
```

```
0:000> ~*kbL
```

```
. 0 Id: e68.fbc Suspend: 1 Teb: 7efdd000 Unfrozen
ChildEBP RetAddr Args to Child
0017f4c8 76272674 000c0ea4 00000000 00000000 USER32!NtUserWaitMessage+0x15
0017f504 7627288a 00070ee6 000c0ea4 00000000 USER32!DialogBox2+0x222
0017f530 762727b8 012e0000 012efc54 000c0ea4 USER32!InternalDialogBox+0xe5
0017f550 76272aa1 012e0000 012efc54 000c0ea4 USER32!DialogBoxIndirectParamAorW+0x37
0017f574 012e124d 012e0000 00000067 000c0ea4 USER32!DialogBoxParamW+0x3f
0017f5e4 76256238 000c0ea4 00000111 00000068 WCM_C!WndProc+0xbd
0017f610 762568ea 012e1190 000c0ea4 00000111 USER32!InternalCallWinProc+0x23
0017f688 76257177 00000000 012e1190 000c0ea4 USER32!UserCallWinProcCheckWow+0x109
0017f6e4 762572f1 01463ec0 00000000 00000111 USER32!DispatchClientMessage+0xe0
0017f720 773700e6 0017f738 00000000 0017f838 USER32!__fnDWORD+0x2b
0017f734 01463ec0 00000000 00000111 00000068 ntdll!KiUserCallbackDispatcher+0x2e
WARNING: Frame IP not in any known module. Following frames may be wrong.
0017f774 012e10e0 0017f79c 00000000 00000000 0x1463ec0
0017f7b4 012e1496 012e0000 00000000 00471bee WCM_C!wWinMain+0xe0
0017f848 76493677 7efde000 0017f894 77399d72 WCM_C!__tmainCRTStartup+0x150
0017f854 77399d72 7efde000 728ca9cf 00000000 kernel32!BaseThreadInitThunk+0xe
0017f894 77399d45 012e1605 7efde000 ffffffff ntdll!_RtlUserThreadStart+0x70
0017f8ac 00000000 012e1605 7efde000 00000000 ntdll!_RtlUserThreadStart+0x1b
```

Send message calls can also be directed to the same thread:

```
0: kd> kv 100
ChildEBP RetAddr Args to Child
aa839180 80833ed1 8c446b60 8c446c08 00000002 nt!KiSwapContext+0x26
aa8391ac 80829c14 8c446d4c 8c446d20 8c446b60 nt!KiSwapThread+0xe5
aa8391f4 80921102 8c446d4c 00000011 8c4a8c01 nt!KeWaitForSingleObject+0x346
aa8392b0 8088b41c 000006a8 00172e58 00172e58 nt!NtRequestWaitReplyPort+0x776
aa8392b0 7c82860c 000006a8 00172e58 00172e58 nt!KiFastCallEntry+0xfc
0012f194 7c827899 77c80a6e 000006a8 00172e58 ntdll!KiFastSystemCallRet
0012f198 77c80a6e 000006a8 00172e58 00172e58 ntdll!ZwRequestWaitReplyPort+0xc
0012f1e4 77c7fcf0 0012f220 0012f204 77c80673 RPCRT4!LRPC_CCALL::SendReceive+0x230
0012f1f0 77c80673 0012f220 771f2918 0012f60c RPCRT4!I_RpcSendReceive+0x24
0012f204 77ce315a 0012f24c 00172ea8 77e63e5f RPCRT4!NdrSendReceive+0x2b
0012f5ec 771f4fb0 771f2918 771f1858 0012f60c RPCRT4!NdrClientCall2+0x22e
[...]
0012f698 7739b6e3 0004001a 00000016 00000001 ApplicationA!WndProc+0xcc
0012f6c4 7739b874 00407440 0004001a 00000016 USER32!InternalCallWinProc+0x28
0012f73c 7739c8b8 00000000 00407440 0004001a USER32!UserCallWinProcCheckWow+0x151
0012f798 7739c9c6 00607890 00000016 00000001 USER32!DispatchClientMessage+0xd9
0012f7c0 7c828556 0012f7d8 00000018 0012f894 USER32!__fnDWORD+0x24
0012f7c0 80831378 0012f7d8 00000018 0012f894 ntdll!KiUserCallbackDispatcher+0x2e
aa83957c 8091fb00 aa839634 aa839638 aa839608 nt!KiCallUserMode+0x4
aa8395d4 bf8a2492 00000002 aa839618 00000018 nt!KeUserModeCallback+0x8f
aa839658 bf8a229d be487890 00000016 00000001 win32k!SfnDWORD+0xb4
aa8396a0 bf8a1249 02487890 00000016 00000001 win32k!xxxSendMessageToClient+0x176
aa8396ec bf8a115e be487890 00000016 00000001 win32k!xxxSendMessageTimeout+0x1a6
aa839710 bf926e0d be487890 00000016 00000001 win32k!xxxSendMessage+0x1b
aa83974c bf926eb5 bc18cbc8 00000016 00000001 win32k!xxxClientShutdown2+0x87
aa839768 bf8ad9fa be487890 80000009 0000029e win32k!xxxClientShutdown+0x47
aa8397c4 bf8845d4 be487890 000003b 80000009 win32k!xxxRealDefWindowProc+0x364
aa8397dc bf884604 be487890 000003b 80000009 win32k!xxxWrapRealDefWindowProc+0x16
```

```

aa8397f8 bf8c1259 be487890 0000003b 80000009 win32k!NtUserFnNCDESTROY+0x27
aa839830 8088b41c 0004001a 0000003b 80000009 win32k!NtUserMessageCall+0xc0
aa839830 7c82860c 0004001a 0000003b 80000009 nt!KiFastCallEntry+0xfc (TrapFrame @ aa839854)
0012f7c0 7c828556 0012f7d8 00000018 0012f894 ntdll!KiFastSystemCallRet
0012f7c0 80831378 0012f7d8 00000018 0012f894 ntdll!KiUserCallbackDispatcher+0x2e
aa839b08 8091fb9b aa839bc0 aa839bc4 aa839b94 nt!KiCallUserMode+0x4
aa839b60 bf8a2492 00000002 aa839ba4 00000018 nt!KeUserModeCallback+0x8f
aa839be4 bf8a229d be487890 0000003b 80000009 win32k!SfnDWORD+0xb4
aa839c2c bf8c3f77 02487890 0000003b 80000009 win32k!xxxSendMessageToClient+0x176
aa839c9c bf89b88e bc18e838 aa839d64 0012fa38 win32k!xxxReceiveMessage+0x2b5
aa839cec bf89d201 aa839d18 0004001a 00000000 win32k!xxxRealInternalGetMessage+0x2d7
aa839d4c 8088b41c 0012fa5c 0004001a 00000000 win32k!NtUserGetMessage+0x3f
aa839d4c 7c82860c 0012fa5c 0004001a 00000000 nt!KiFastCallEntry+0xfc (TrapFrame @ aa839d64)
0012f9f0 7c828556 0012fa08 00000018 0012ffb0 ntdll!KiFastSystemCallRet
0012fa1c 7739c811 7739c844 0012fa5c 0004001a ntdll!KiUserCallbackDispatcher+0x2e
0012fa3c 0040634e 0012fa5c 0004001a 00000000 USER32!NtUserGetMessage+0xc
0012ff18 00408d9d 00000032 00000000 00142546 ApplicationA!WinMain+0x80f
0012ffc0 77e6f22b 00000000 00000000 7ffd000 ApplicationA!WinMainCRTStartup+0x185
0012fff0 00000000 00408c18 00000000 78746341 kernel32!BaseProcessStart+0x23

```

Blocked sent message calls can also be manifested in kernel space and mixed with patterns like **Message Box** (page 660) and **Main Thread** (page 614), for example:

```

1: kd> k250
ChildEBP RetAddr
8d5d2808 82a7eb15 nt!KiSwapContext+0x26
8d5d2840 82a7d403 nt!KiSwapThread+0x266
8d5d2868 82a772cf nt!KiCommitThreadWait+0x1df
8d5d28e0 82550d75 nt!KeWaitForSingleObject+0x393
8d5d293c 82550e10 win32k!xxxRealSleepThread+0x1d7
8d5d2958 824ff4b0 win32k!xxxSleepThread+0x2d
8d5d29cc 825547e8 win32k!xxxInterSendMsgEx+0xb1c
8d5d2a1c 825546a4 win32k!xxxSendMessageTimeout+0x13b
8d5d2a44 82533843 win32k!xxxSendMessage+0x28
8d5d2b08 824fd865 win32k!xxxCalcValidRects+0xf7
8d5d2b64 82502c98 win32k!xxxEndDeferWindowPosEx+0x100
8d5d2b84 825170c9 win32k!xxxSetWindowPos+0xf6
8d5d2c08 82517701 win32k!xxxActivateThisWindow+0x2b1
8d5d2c38 82517537 win32k!xxxActivateWindow+0x144
8d5d2c4c 824fd9dd win32k!xxxSwpActivate+0x44
8d5d2ca4 82502c98 win32k!xxxEndDeferWindowPosEx+0x278
8d5d2cc4 824ffff82 win32k!xxxSetWindowPos+0xf6
8d5d2d10 82a5342a win32k!NtUserSetWindowPos+0x140
8d5d2d10 76ee64f4 nt!KiFastCallEntry+0x12a (TrapFrame @ 8d5d2d34)
01e2cea0 7621358d ntdll!KiFastSystemCallRet
01e2cea4 6a8fa0eb USER32!NtUserSetWindowPos+0xc
01e2cf14 6a894b13 IEFRAME!SHToggleDialogExpando+0x15a
01e2cf28 6a894d5d IEFRAME!EleDlg::ToggleExpando+0x20
01e2d74c 6a895254 IEFRAME!EleDlg::OnInitDlg+0x229
01e2d7b8 762186ef IEFRAME!EleDlg::DlgProcEx+0x189
01e2d7e4 76209eb2 USER32!InternalCallWinProc+0x23
01e2d860 7620b98b USER32!UserCallDlgProcCheckWow+0xd6
01e2d8a8 7620bb7b USER32!DefDlgProcWorker+0xa8
01e2d8c4 762186ef USER32!DefDlgProcW+0x22
01e2d8f0 76218876 USER32!InternalCallWinProc+0x23

```

```

01e2d968 76217631 USER32!UserCallWinProcCheckWow+0x14b
01e2d9a8 76209b1d USER32!SendMessageWorker+0x4d0
01e2da64 76235500 USER32!InternalCreateDialog+0xb0d
01e2da94 76235553 USER32!InternalDialogBox+0xa7
01e2dab4 76235689 USER32!DialogBoxIndirectParamAorW+0x37
01e2dad8 6a5d4952 USER32!DialogBoxParamW+0x3f
01e2db00 6a5d5024 IEFRAME!Detour_DialogBoxParamW+0x47
01e2db24 6a8956df IEFRAME!SHFusionDialogBoxParam+0x32
01e2db58 6a8957bb IEFRAME!EleDlg::ShowDialog+0x398
01e2e638 6a8959d3 IEFRAME!ShowDialogBox+0xb6
01e2eb9c 6a9013ed IEFRAME!ShowElevationPrompt+0x1dd
01e2f010 7669fc8f IEFRAME!CIEUserBrokerObject::BrokerCoCreateInstance+0x202
01e2f040 76704c53 RPCRT4!Invoke+0x2a
01e2f448 76d9d936 RPCRT4!NdrStubCall2+0x2d6
01e2f490 76d9d9c6 ole32!CStdStubBuffer_Invoke+0xb6
01e2f4d8 76d9df1f ole32!SyncStubInvoke+0x3c
01e2f524 76cb213c ole32!StubInvoke+0xb9
01e2f600 76cb2031 ole32!CCtxComChnl::ContextInvoke+0xfa
01e2f61c 76d9a754 ole32!MTAInvoke+0x1a
01e2f64c 76d9dcbb ole32!AppInvoke+0xab
01e2f72c 76d9a773 ole32!ComInvokeWithLockAndIPID+0x372
01e2f778 7669f34a ole32!ThreadInvoke+0x302
01e2f7b4 7669f4da RPCRT4!DispatchToStubInCNoAvrf+0x4a
01e2f80c 7669f3c6 RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x16c
01e2f834 766a0cef RPCRT4!RPC_INTERFACE::DispatchToStub+0x8b
01e2f86c 7669f882 RPCRT4!RPC_INTERFACE::DispatchToStubWithObject+0xb2
01e2f8b8 7669f7a4 RPCRT4!LRPC_SCALL::DispatchRequest+0x23b
01e2f8d8 7669f763 RPCRT4!LRPC_SCALL::QueueOrDispatchCall+0xbd
01e2f8f4 7669f5ff RPCRT4!LRPC_SCALL::HandleRequest+0x34f
01e2f928 7669f573 RPCRT4!LRPC_SASSOCIATION::HandleRequest+0x144
01e2f960 7669ee4f RPCRT4!LRPC_ADDRESS::HandleRequest+0xbd
01e2f9dc 7669ece7 RPCRT4!LRPC_ADDRESS::ProcessIO+0x50a
01e2f9e8 766a1357 RPCRT4!LrpServerIoHandler+0x16
01e2f9f8 76ecd3a3 RPCRT4!LrpCiComplete+0x16
01e2fa20 76ed0748 nt!TppAlpcpExecuteCallback+0x1c5
01e2fb88 76e11174 nt!TppWorkerThread+0x5a4
01e2fb94 76efb3f5 kernel32!BaseThreadInitThunk+0xe
01e2fb9d 76efb3c8 nt!_RtlUserThreadStart+0x70
01e2fbec 00000000 nt!_RtlUserThreadStart+0x1b

```

```

2: kd> !process 890ff430 1f
PROCESS 890ff430 SessionId: 1 Cid: 18a4 Peb: 7ffd000 ParentCid: 1fdc
DirBase: 7fbf04e0 ObjectTable: da89fb80 HandleCount: 852.
Image: iexplore.exe

```

```

THREAD 89141db0 Cid 18a4.19c8 Teb: 7ffdf000 Win32Thread: bc373d18 WAIT: (Unknown) UserMode Non-Alertable
8915b020 SynchronizationEvent
Not impersonating
DeviceMap      da7f9680
Owning Process 890ff430 Image: iexplore.exe
Attached Process N/A Image: N/A
Wait Start TickCount 56879 Ticks: 1634 (0:00:00:25.531)
Context Switch Count 12410 NoStackSwap LargeStack
UserTime        00:00:00.078
KernelTime      00:00:01.234
Win32 Start Address iexplore!wWinMainCRTStartup (0x004031b9)

```

```
Start Address kernel32!BaseProcessStartThunk (0x77e617f8)
Stack Init b5672000 Current b56717c4 Base b5672000 Limit b566c000 Call 0
Priority 4 BasePriority 4 PriorityDecrement 0
ChildEBP RetAddr
b56717dc 80833ec5 nt!KiSwapContext+0x26
b5671808 80829c14 nt!KiSwapThread+0x2e5
b5671850 bf89ab73 nt!KeWaitForSingleObject+0x346
b56718ac bf8c4ba6 win32k!xxxSleepThread+0x1be
b5671948 bf8a13e0 win32k!xxxInterSendMsgEx+0x798
b5671994 bf8a132f win32k!xxxSendMessageTimeout+0x1f3
b56719b8 bf85ca01 win32k!xxxSendMessage+0x1b
b5671a7c bf85da04 win32k!xxxCalcValidRects+0xea
b5671ad8 bf85de2e win32k!xxxEndDeferWindowPosEx+0xf2
b5671af4 bf861cf2 win32k!xxxSetWindowPos+0xb1
b5671b3c bf882098 win32k!xxxProcessEventMessage+0x232
b5671c7c bf89b89e win32k!xxxScanSysQueue+0x21e
b5671ce4 bf89c529 win32k!xxxRealInternalGetMessage+0x2aa
b5671d48 8088b41c win32k!NtUserPeekMessage+0x42
b5671d48 7c82860c nt!KiFastCallEntry+0xfc (TrapFrame @ b5671d64)
0012e6e8 7739bde5 ntdll!KiFastSystemCallRet
0012e714 7739be5e USER32!NtUserPeekMessage+0xc
0012e740 02935f8c USER32!PeekMessageW+0xab
0012e7b4 02936150 IEUI!DUserRegisterSuper+0x920
0012e7d4 40d2ee98 IEUI!PeekMessageExW+0x42
0012e818 40d2abf4 IEFRA  
ME!CBrowserFrame::FrameMessagePump+0x23
0012e824 40d2bc63 IEFRA  
ME!BrowserThreadProc+0x3f
0012e848 40d2bbb1 IEFRA  
ME!BrowserNewThreadProc+0x7b
0012f8b8 40d2ba61 IEFRA  
ME!SHOpenFolderWindow+0x188
0012fae8 00401484 IEFRA  
ME!IEWinMain+0x2d9
0012ff2c 0040131f iexplore!wWinMain+0x2c6
0012ffc0 77e6f23b iexplore!_initterm_e+0x1b1
0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

## Waiting Thread Time

### Kernel Dumps

Almost all threads in any system are waiting for resources or waiting in ready-to-run queues to be scheduled. At any moment of time, the number of running threads is equal to the number of processors. The rest, hundreds and thousands of threads, are waiting. Looking at their waiting times in the kernel and complete memory dumps provides some interesting observations that worth their own pattern name: **Waiting Thread Time**.

When a thread starts waiting that time is recorded in *WaitTime* field of *\_KTHREAD* structure:

```
1: kd> dt _KTHREAD 8728a020
+0x000 Header      : _DISPATCHER_HEADER
+0x010 MutantListHead : _LIST_ENTRY [ 0x8728a030 - 0x8728a030 ]
+0x018 InitialStack : 0xa3a1f000
+0x01c StackLimit   : 0xa3a1a000
+0x020 KernelStack   : 0xa3a1ec08
+0x024 ThreadLock    : 0
+0x028 ApcState     : _KAPC_STATE
+0x028 ApcStateFill  : [23] "H???""
+0x03f ApcQueueable  : 0x1 ''
+0x040 NextProcessor  : 0x3 ''
+0x041 DeferredProcessor : 0x3 ''
+0x042 AdjustReason   : 0 ''
+0x043 AdjustIncrement : 1 ''
+0x044 ApcQueueLock   : 0
+0x048 ContextSwitches : 0x6b7
+0x04c State         : 0x5 ''
+0x04d NpxState      : 0xa ''
+0x04e WaitIrql       : 0 ''
+0x04f WaitMode       : 1 ''
+0x050 WaitStatus     : 0
+0x054 WaitBlockList  : 0x8728a0c8 _KWAIT_BLOCK
+0x054 GateObject     : 0x8728a0c8 _KGATE
+0x058 Alertable      : 0 ''
+0x059 WaitNext        : 0 ''
+0x05a WaitReason      : 0x11 ''
+0x05b Priority        : 12 ''
+0x05c EnableStackSwap : 0x1 ''
+0x05d SwapBusy        : 0 ''
+0x05e Alerted         : [2] ""
+0x060 WaitListEntry   : _LIST_ENTRY [ 0x88091e10 - 0x88029ce0 ]
+0x060 SwapListEntry   : _SINGLE_LIST_ENTRY
+0x068 Queue           : (null)
+0x06c WaitTime         : 0x82de9b
+0x070 KernelApcDisable : 0
...
```

This value is also displayed in the decimal format as *Wait Start TickCount* when we list threads or use **!thread** command:

```
0: kd> ? 0x82de9b
Evaluate expression: 8576667 = 0082de9b

1: kd> !thread 8728a020
THREAD 8728a020 Cid 4c9c.59a4 Teb: 7fffd000 Win32Thread: bc012008 WAIT: (Unknown) UserMode Non-
Alertable
8728a20c Semaphore Limit 0x1
Waiting for reply to LPC MessageId 017db413:
Current LPC port e5fcff68
Impersonation token: e2b07028 (Level Impersonation)
DeviceMap           e1da6518
Owning Process     89d20740      Image:          winlogon.exe
Wait Start TickCount    8576667      Ticks: 7256 (0:00:01:53.375)
Context Switch Count 1719          LargeStack
UserTime            00:00:00.0359
KernelTime          00:00:00.0375
```

*Tick* is a system unit of time and *KeTimeIncrement* double-word value contains its equivalent as the number of 100-nanosecond units:

```
0: kd> dd KeTimeIncrement 11
808a6304 0002625a

0: kd> ? 0002625a
Evaluate expression: 156250 = 0002625a

0: kd> ?? 156250.0/10000000.0
double 0.015625
```

Therefore on that system, one tick is 0.015625 of a second.

The current tick count is available via *KeTickCount* variable:

```
0: kd> dd KeTickCount 11
8089c180 0082faf3
```

If we subtract the recorded start wait time from the current tick count we get the number of ticks passed since the thread began waiting:

```
0: kd> ? 0082faf3-82de9b
Evaluate expression: 7256 = 00001c58
```

Using our previously calculated constant of the number of seconds per tick (0.015625) we get the number of seconds passed:

```
0: kd> ?? 7256.0 * 0.015625
double 113.37499999999999
```

113.375 seconds is 1 minute 53 seconds and 375 milliseconds:

```
0: kd> ?? 113.375-60.0
double 53.37499999999986
```

We can see that this value corresponds to *Ticks* value that WinDbg shows for the thread:

```
Wait Start TickCount 8576667 Ticks: 7256 (0:00:01:53.375)
```

Why do we need to concern ourselves with these ticks? If we know that some activity was frozen for 15 minutes, we can filter out threads from our search space because threads with significantly less number of ticks were running at some time and were not waiting for 15 minutes.

Threads with a low number of ticks were running recently:

```
THREAD 86ced020 Cid 0004.3908 Teb: 00000000 Win32Thread: 00000000 WAIT: (Unknown) KernelMode Non-Alertable
b99cb7d0 QueueObject
86ced098 NotificationTimer
Not impersonating
DeviceMap e10038e0
Owning Process 8ad842a8 Image: System
Wait Start TickCount 8583871 Ticks: 52 (0:00:00:00.812)
Context Switch Count 208
UserTime 00:00:00.0000
KernelTime 00:00:00.0000
Start Address rdbss!RxWorkerThreadDispatcher (0xb99cdc2e)
Stack Init ad21d000 Current ad21cccd8 Base ad21d000 Limit ad21a000 Call 0
Priority 8 BasePriority 8 PriorityDecrement
ChildEBP RetAddr
ad21ccf0 808330c6 nt!KiSwapContext+0x26
ad21cd1c 8082af7f nt!KiSwapThread+0x284
ad21cd64 b99c00e9 nt!KeRemoveQueue+0x417
ad21cd9c b99cdc48 rdbss!RxpWorkerThreadDispatcher+0x4b
ad21cdac 80948e74 rdbss!RxWorkerThreadDispatcher+0x1a
ad21cddc 8088d632 nt!PspSystemThreadStartup+0x2e
00000000 00000000 nt!KiThreadStartup+0x16
```

Another application would be to find all threads from different processes whose wait time roughly corresponds to 15 minutes, and, therefore, they might be related to the same frozen activity. For example, these RPC threads below from different processes are most likely related because one is the RPC client thread, the other is the RPC server thread waiting for some object, and their common *Ticks* value is the same: 15131.

## 1140 | Waiting Thread Time

THREAD 89cc9750 Cid 0f1c.0f60 Peb: 7ffd6000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Non-Alertable  
89cc993c Semaphore Limit 0x1  
Waiting for reply to LPC MessageId 0000a7e7:  
Current LPC port e18fcae8  
Not impersonating  
DeviceMap e10018a8  
Owning Process 88d3b938 Image: svchost.exe  
Wait Start TickCount 29614 Ticks: 15131 (0:00:03:56.421)  
Context Switch Count 45  
UserTime 00:00:00.0000  
KernelTime 00:00:00.0000  
Win32 Start Address 0x0000a7e6  
LPC Server thread working on message Id a7e6  
Start Address kernel32!BaseThreadStartThunk (0x7c82b5bb)  
Stack Init f29a6000 Current f29a5c08 Base f29a6000 Limit f29a3000 Call 0  
Priority 11 BasePriority 10 PriorityDecrement 0  
Kernel stack not resident.  
ChildEBP RetAddr  
f29a5c20 80832f7a nt!KiSwapContext+0x26  
f29a5c4c 8082927a nt!KiSwapThread+0x284  
f29a5c94 8091df86 nt!KeWaitForSingleObject+0x346  
f29a5d50 80888c6c nt!NtRequestWaitReplyPort+0x776  
f29a5d50 7c94ed54 nt!KiFastCallEntry+0xfc  
0090f6b8 7c941c94 nt!NtRequestWaitReplyPort+0xc  
0090f6bc 77c42700 nt!NtRequestWaitReplyPort+0xc  
0090f708 77c413ba RPCRT4!LRPC\_CCALL::SendReceive+0x230  
0090f714 77c42c7f RPCRT4!I\_RpcSendReceive+0x24  
0090f728 77cb5d63 RPCRT4!NdrSendReceive+0x2b  
0090f9cc 67b610ca RPCRT4!NdrClientCall+0x334  
0090f9dc 67b61c07 component!NotifyOfEvent+0x14  
...  
0090ffec 00000000 kernel32!BaseThreadStart+0x34

THREAD 89b49590 Cid 098c.01dc Peb: 7ff92000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Non-Alertable  
88c4e020 Thread  
89b49608 NotificationTimer  
Not impersonating  
DeviceMap e10018a8  
Owning Process 89d399c0 Image: MyService.exe  
Wait Start TickCount 29614 Ticks: 15131 (0:00:03:56.421)  
Context Switch Count 310  
UserTime 00:00:00.0015  
KernelTime 00:00:00.0000  
Win32 Start Address 0x0000a7e7  
LPC Server thread working on message Id a7e7  
Start Address kernel32!BaseThreadStartThunk (0x7c82b5bb)  
Stack Init f2862000 Current f2861c60 Base f2862000 Limit f285f000 Call 0  
Priority 11 BasePriority 10 PriorityDecrement 0  
Kernel stack not resident.  
ChildEBP RetAddr  
f2861c78 80832f7a nt!KiSwapContext+0x26  
f2861ca4 8082927a nt!KiSwapThread+0x284  
f2861cec 80937e4c nt!KeWaitForSingleObject+0x346  
f2861d50 80888c6c nt!NtWaitForSingleObject+0x9a

```
f2861d50 7c94ed54 nt!KiFastCallEntry+0xfc
0a6cf590 7c942124 ntdll!KiFastSystemCallRet
0a6cf594 7c82baa8 ntdll!NtWaitForSingleObject+0xc
0a6cf604 7c82ba12 kernel32!WaitForSingleObjectEx+0xac
0a6cf618 3f691c11 kernel32!WaitForSingleObject+0x12
0a6cf658 09734436 component2!WaitForResponse+0x75
...
0a6cf8b4 77cb23f7 RPCRT4!Invoke+0x30
0a6cfcb4 77cb26ed RPCRT4!NdrStubCall2+0x299
0a6cfcd0 77c409be RPCRT4!NdrServerCall2+0x19
0a6cf04 77c4093f RPCRT4!DispatchToStubInCNoAvrf+0x38
0a6cf58 77c40865 RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x117
0a6cf7c 77c357eb RPCRT4!RPC_INTERFACE::DispatchToStub+0xa3
0a6cfdbc 77c41e26 RPCRT4!RPC_INTERFACE::DispatchToStubWithObject+0xc0
0a6cfdfc 77c41bb3 RPCRT4!LRPC_CALL::DealWithRequestMessage+0x42c
0a6cfe20 77c45458 RPCRT4!LRPC_ADDRESS::DealWithLRPCRequest+0x127
0a6cff84 77c2778f RPCRT4!LRPC_ADDRESS::ReceiveLotsaCalls+0x430
0a6cff8c 77c2f7dd RPCRT4!RecvLotsaCallsWrapper+0xd
0a6cffac 77c2de88 RPCRT4!BaseCachedThreadRoutine+0x9d
0a6cffb8 7c826063 RPCRT4!ThreadStartRoutine+0x1b
0a6cffec 00000000 kernel32!BaseThreadStart+0x34
```

To convert ticks to the time interval we can use **!whattime** command:

```
3: kd> !whattime 0n7256
7256 Ticks in Standard Time: 01:53.375s
```

Also **!stacks** command shows *Ticks* data for threads.

## Comments

Another example: a client LPC thread was waiting, but the corresponding server thread value was missing in the output of `!lpc` extension. Because the server process name was known from the output, the server thread was found by the same waiting value:

```
THREAD 894e13a8 Cid 02d4.0e28 Teb: 7ffdf000 Win32Thread: e6e7cea8 WAIT: (Unknown) KernelMode Non-Alertable
894e1594 Semaphore Limit 0x1
Waiting for reply to LPC MessageId 3786440d:
Current LPC port e6edd680
IRP List:
894cb818: (0006,01d8) Flags: 00000884 Mdl: 00000000
Impersonation token: e76aa028 (Level Impersonation)
DeviceMap e558a008
Owning Process 895497d0 Image: ApplicationA.exe
Attached Process N/A Image: N/A
Wait Start TickCount 9426151 Ticks: 162368 (0:00:42:17.000)
Context Switch Count 1297 LargeStack
UserTime      00:00:00.031
KernelTime    00:00:00.359
Start Address 0x0103e1b0
Stack Init b7ad8000 Current b7ad7174 Base b7ad8000 Limit b7ad3000 Call 0
Priority 15 BasePriority 15 PriorityDecrement 0

0: kd> !lpc message 3786440d
Searching message 3786440d in threads ...
Client thread 894e13a8 waiting a reply from 3786440d
Searching thread 894e13a8 in port rundown queues ...

Server communication port 0xe71e7f08
Handles: 1 References: 1
The LpcDataInfoChainHead queue is empty
Connected port: 0xe6edd680 Server connection port: 0xe4f348e0

Client communication port 0xe6edd680
Handles: 0 References: 1
The LpcDataInfoChainHead queue is empty

Server connection port e4f348e0 Name: ServiceBPort
Handles: 1 References: 156
Server process : 8a75f438 (ServiceB.exe)
Queue semaphore : 8a6c4908
Semaphore state 0 (0x0)
The message queue is empty
The LpcDataInfoChainHead queue is empty
Done.
```

```
THREAD 8a6439d8 Cid 0234.02b0 Teb: 7ffa5000 Win32Thread: 00000000 WAIT: (Unknown) UserMode Alertable  
8a63b7f0 NotificationEvent  
IRP List:  
8a229a50: (0006,0094) Flags: 00000900 Mdl: 00000000  
Not impersonating  
DeviceMap e1001840  
Owning Process 8a75f438 Image: ServiceB.exe  
Attached Process N/A Image: N/A  
Wait Start TickCount 9426151 Ticks: 162368 (0:00:42:17.000)  
Context Switch Count 31433  
UserTime      00:00:00.375  
KernelTime    00:00:02.625  
Win32 Start Address 0x4ab8f3f1  
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)  
Stack Init ba466000 Current ba465c60 Base ba466000 Limit ba463000 Call 0  
Priority 13 BasePriority 9 PriorityDecrement 0  
ChildEBP RetAddr  
ba465c78 80833491 nt!KiSwapContext+0x26  
ba465ca4 80829a82 nt!KiSwapThread+0x2e5  
ba465cec 80938e0a nt!KeWaitForSingleObject+0x346  
ba465d50 808897ec nt!NtWaitForSingleObject+0x9a  
ba465d50 7c82845c nt!KiFastCallEntry+0xfc (TrapFrame @ ba465d64)  
00ebf250 7c827b79 nt!KFastSystemCallRet  
00ebf254 77e61d1e nt!ZwWaitForSingleObject+0xc  
00ebf2c4 77c6a927 kernel32!WaitForSingleObjectEx+0xac  
00ebf2e0 77c69cbf RPCRT4!UTIL_WaitForSyncIO+0x20  
00ebf304 77c6cef0 RPCRT4!UTIL_GetOverlappedResultEx+0x1d  
00ebf32c 77c6b02a RPCRT4!CO_SyncRecv+0x71  
00ebf354 77c6972f RPCRT4!OSF_CCONNECTION::TransSendReceive+0xb8  
00ebf3dc 77c6969c RPCRT4!OSF_CCONNECTION::SendFragment+0x2ae  
00ebf434 77c69842 RPCRT4!OSF_CCALL::SendNextFragment+0x1e2  
00ebf47c 77c69aba RPCRT4!OSF_CCALL::FastSendReceive+0x148  
00ebf498 77c69a3d RPCRT4!OSF_CCALL::SendReceiveHelper+0x5b  
00ebf4c8 77c7feb0 RPCRT4!OSF_CCALL::SendReceive+0x41  
00ebf4d4 77c80845 RPCRT4!I_RpcSendReceive+0x24  
00ebf4e8 77ce415a RPCRT4!NdrSendReceive+0x2b  
00ebf8d8 7d1fac12 RPCRT4!NdrClientCall2+0x22e  
[...]
```

## User Dumps

**Waiting Thread Time** kernel pattern (page 1137) shows how to calculate thread waiting time in kernel and complete memory dumps. Unfortunately, kernel information is not available in user space and, therefore, this can be done only approximately by calculating an average or maximum waiting time.

The key is to take advantage of **!runaway** WinDbg command. Modern versions of postmortem debuggers and process dumpers usually save thread time information additionally. For example, **/ma** switch for **.dump** command does it. The *procdump.exe* and the later versions of *userdump.exe* also do it. When **!runaway** is invoked with **Oy111** (7) flag it shows each thread user and kernel times and also the time elapsed since each thread was created. Therefore for threads that didn't do a much CPU-intensive computation their average waiting time will almost correspond to their elapsed time. Unfortunately, we cannot say when that computation took place. It could be the case that the thread in question was waiting 98.9% of the time, and then did some computation for 0.01% and was waiting again 1% of the time, or it could be the case that it was doing computation 0.01% of the time initially and then waiting the rest of the time (99.9%) until the dump was saved.

Consider this example of a sleeping thread:

```
0:000> ~32kL 100
ChildEBP RetAddr
0a18f03c 7c826f4b ntdll!KiFastSystemCallRet
0a18f040 77e41ed1 ntdll!NtDelayExecution+0xc
0a18f0a8 77e424ed kernel32!SleepEx+0x68
0a18f0b8 1b0ac343 kernel32!Sleep+0xf
0a18f100 1b00ba49 msjet40!System::AllocatePages+0x15e
0a18f11c 1b00ba08 msjet40!PageDesc::WireCurrentPage+0x2f
0a18f13c 1b00b8dd msjet40!PageDesc::ReadPage+0x119
0a18f164 1b01b567 msjet40!Database::ReadPage+0x7a
0a18f190 1b00e309 msjet40!TableMover::GetNextRow+0xc9
0a18f1a8 1b015de9 msjet40!TableMover::Move+0xc4
0a18f1d8 1b015d9c msjet40!ErrIsamMove+0x6c
0a18f1f0 1b038aa4 msjet40!ErrDispMove+0x43
0a18f43c 1b015d9c msjet40!ErrJPMoveRange+0x350
0a18f454 1b0200b3 msjet40!ErrDispMove+0x43
0a18f6a0 1b021e4d msjet40!ErrMaterializeRows+0x1fd
0a18f6f0 1b021c0d msjet40!ErrJPSetColumnSort+0x191
0a18f718 1b0210a4 msjet40!ErrJPOpenSort+0x105
0a18f750 1b020de5 msjet40!ErrJPOpenRvt+0x171
0a18f9f0 1b039b82 msjet40!ErrExecuteQuery+0x548
0a18fa3c 1b05548b msjet40!ErrExecuteTempQuery+0x13d
0a18fa6c 4c23b01e msjet40!JetExecuteTempQuery+0xc9
0a18fa94 4c23a8d1 odbcjt32!DoJetCloseTable+0x64
0a18fd10 4c23a6b6 odbcjt32!SQLInternalExecute+0x217
0a18fd20 4c23a694 odbcjt32!SQLExecuteCover+0x1f
0a18fd28 488b3fc1 odbcjt32!SQLExecute+0x9
0a18fd44 0c4898b5 odbc32!SQLExecute+0xd3
...
...
```

The process uptime can be seen from **vertarget** WinDbg command:

```
0:000> vertarget
...
Process Uptime: 0 days 1:17:22.000
...
```

Then we dump thread times (the output is long, and the only information for the 32nd thread is shown here):

```
0:000> !runaway 0y111
User Mode Time
 Thread      Time
 32:cfc      0 days 0:00:00.109
...
Kernel Mode Time
 Thread      Time
 32:cfc      0 days 0:00:00.062
...
Elapsed Time
 Thread      Time
...
 32:cfc      0 days 1:17:20.703
...
```

We see that 1 hour, 17 minutes and almost 21 seconds passed since the thread was created. By subtracting this time from process uptime we see that it was created in the first 2 seconds, and it was consuming less than one second of CPU time. Therefore, most of the time this thread was waiting. Unfortunately, we cannot say when it was waiting most of the time, in the beginning before it started sleeping or after. Fortunately, we might continue guessing by looking at *Sleep* function argument:

```
0:032> kv
ChildEBP RetAddr  Args to Child
0a18f03c 7c826f4b 77e41ed1 00000000 0a18f080 ntdll!KiFastSystemCallRet
0a18f040 77e41ed1 00000000 0a18f080 00000000 ntdll!NtDelayExecution+0xc
0a18f0a8 77e424ed 00000032 00000000 04390000 kernel32!SleepEx+0x68
0a18f0b8 1b0ac343 00000032 042a34b4 0a18f1c8 kernel32!Sleep+0xf
...
```

We see that it had been sleeping for at most 32 milliseconds and, perhaps, this is a retry / sleep loop. We might guess that the thread was recently spinning in that loop and therefore waiting all the time before that. Alternatively, we might guess that the retry portion is very small and fast and 32 milliseconds are spread over all elapsed time, and the thread was in this loop the significant proportion of time. What we can surely say that the last waiting time was no more than 32 milliseconds. In the case of waiting for an event, for example, at present, it seems there is no any reliable way of calculating this time.

## Comments

Note that “*sleeping for at most 32 milliseconds*” means 32 HEX milliseconds.

## Well-Tested Function

When looking at stack traces software engineers immediately spot the right function to dig into its source code:

```
STACK_TEXT:  
05b3f514 66ed52d3 08d72fee 08da6fc4 05b3f540 msvcrt!wcscpy+0xe  
05b3f53c 77c50193 07516fc8 06637fc4 00000006 ModuleA!Add+0xd8  
05b3f568 77cb33e1 66ed51fb 05b3f750 00000006 rpcrt4!Invoke+0x30  
05b3f968 77cb1968 08d6afe0 0774cf4c 0660af28 rpcrt4!NdrStubCall12+0x299  
[...]
```

Most will start with *ModuleA!Add* and examine parameters to *wcscpy* function. This is because *wcscpy* (UNICODE version of *strcpy*) is considered **Well-Tested**. For the purposes of default analysis via **!analyse -v** command it is possible to configure WinDbg to ignore our own functions and modules as well if we are sure they were **Well-Tested** or **Pass Through** (page 787). For details, please see minidump analysis case study<sup>235</sup>.

---

<sup>235</sup> Component Identification, Memory Dump Analysis Anthology, Volume 1, page 46

## Well-Tested Module

We introduce this pattern by analogy with **Well-Tested Function** (page 1146). **Well-Tested Module** is a module we usually skip when analyzing **Stack Trace** (page 926) because we suspect it the least. WinDbg can also be customized to skip such modules for the default analysis command as shown in component identification minidump analysis example<sup>236</sup>.

---

<sup>236</sup> Component Identification, Memory Dump Analysis Anthology, Volume 1, page 46

## Wild Code

The case when a function pointer or a return address becomes **Wild Pointer** (page 1151) and EIP or RIP value lies in a valid region of memory the execution path may continue through a region called **Wild Code**. This might loop on itself or eventually reach non-executable or invalid pages and produce an exception. **Local Buffer Overflow** (page 611) might lead to this behavior and also data corruption that overwrites function pointers with valid memory addresses.

Our favorite example is when a function pointer points to zeroed pages with EXECUTE page attribute. What will happen next when we dereference it? All zeroes are the perfect x86/x64 code:

```
0:001> dd 0000000`771afdf0
0000000`771afdf0 00000000 00000000 00000000 00000000
0000000`771afe00 00000000 00000000 00000000 00000000
0000000`771afe10 00000000 00000000 00000000 00000000
0000000`771afe20 00000000 00000000 00000000 00000000
0000000`771afe30 00000000 00000000 00000000 00000000
0000000`771afe40 00000000 00000000 00000000 00000000
0000000`771afe50 00000000 00000000 00000000 00000000
0000000`771afe60 00000000 00000000 00000000 00000000

0:001> u
ntdll!DbgUserBreakPoint:
0000000`771afe00 0000 add byte ptr [rax],al
0000000`771afe02 0000 add byte ptr [rax],al
0000000`771afe04 0000 add byte ptr [rax],al
0000000`771afe06 0000 add byte ptr [rax],al
0000000`771afe08 0000 add byte ptr [rax],al
0000000`771afe0a 0000 add byte ptr [rax],al
0000000`771afe0c 0000 add byte ptr [rax],al
0000000`771afe0e 0000 add byte ptr [rax],al
```

Now if RAX points to a valid memory page with WRITE attribute the code will modify the first byte at that address:

```
0:001> dq @rax
000007ff`ffffdc000 0000000`00000000 0000000`035a0000
000007ff`ffffdc010 0000000`0359c000 0000000`00000000
000007ff`ffffdc020 0000000`00001e00 0000000`00000000
000007ff`ffffdc030 000007ff`ffffdc000 0000000`00000000
000007ff`ffffdc040 0000000`0000142c 0000000`00001504
000007ff`ffffdc050 0000000`00000000 0000000`00000000
000007ff`ffffdc060 000007ff`ffffd8000 0000000`00000000
000007ff`ffffdc070 0000000`00000000 0000000`00000000
```

Therefore the code will be perfectly executed:

```
0:001> t
ntdll!DbgBreakPoint+0x2:
0000000`771afdf2 0000 add byte ptr [rax],al ds:000007ff`ffffdc000=00
```

```

0:001> t
ntdll!DbgBreakPoint+0x4:
0000000`771afdf4 0000    add     byte ptr [rax],al ds:000007ff`ffffdc000=00

0:001> t
ntdll!DbgBreakPoint+0x6:
0000000`771afdf6 0000    add     byte ptr [rax],al ds:000007ff`ffffdc000=00

0:001> t
ntdll!DbgBreakPoint+0x8:
0000000`771afdf8 0000    add     byte ptr [rax],al ds:000007ff`ffffdc000=00

0:001> t
ntdll!DbgBreakPoint+0xa:
0000000`771afdfa 0000    add     byte ptr [rax],al ds:000007ff`ffffdc000=00

```

## Comments

Sometimes we see **Incorrect Stack Trace** (page 499) pattern since it doesn't make sense that waiting functions call GDI and other modules:

```

0:000> k
ChildEBP RetAddr
03ced1b0 771d6aec ntdll!KiFastSystemCallRet
03ced1b4 75406a8e ntdll!NtWaitForMultipleObjects+0xc
03ced250 7734be76 KERNELBASE!WaitForMultipleObjectsEx+0x100
03ced298 7734bee4 kernel32!WaitForMultipleObjectsExImplementation+0xe0
03ced2b4 7736072f kernel32!WaitForMultipleObjects+0x18
03ced320 773609ca kernel32!WerFaultInternal+0x186
03ced334 77360978 kernel32!WerFault+0x70
03ced344 773608f3 kernel32!BaseReportFault+0x20
03ced3d0 7720820a kernel32!UnhandledExceptionFilter+0x1af
03ced3d8 771ae364 ntdll!_RtlUserThreadStart+0x62
03ced3ec 771ae1fc ntdll!_EH4_CallFilterFunc+0x12
03ced414 771d72b9 ntdll!_except_handler4+0x8e
03ced438 771d728b ntdll!ExecuteHandler2+0x26
03ced45c 771af9d7 ntdll!ExecuteHandler+0x24
03ced4e8 771d7117 ntdll!RtlDispatchException+0x127
03ced4e8 63050001 ntdll!KiUserExceptionDispatcher+0xf
03cedaa4 771e73e2 ModuleA!FunctionA+0xc1
03ceda84 0141b848 ntdll!_SEH_epilog4_GS+0xa
03cedcb4 767cbbf4 ModuleB!FunctionB+0x188
03cedcc8 767cbc5 gdi32!NtGdiOpenDCW+0xc
03cedf70 013bc7df gdi32!hdcCreateDCW+0x517
03cee0c8 7734c413 ModuleB!FunctionC+0xff
03cee0e0 7734c3c2 kernel32!WaitForSingleObjectExImplementation+0x75
03cee0f4 65e66c9c kernel32!WaitForSingleObject+0x12
WARNING: Frame IP not in any known module. Following frames may be wrong.
00000000 00000000 0x65e66c9c

```

Also the return address for *ModuleA!FunctionA* looks coincidental: 63050001. Although we are able to see code we are not able to disassemble it backwards:

```
0:000> u 63050001
ModuleA!FunctionA+0xc1:
63050001 a1056383c4 mov eax,dword ptr ds:[C4836305h]
63050006 108d4c241051 adc byte ptr [ebp+5110244Ch],cl
6305000c ff15b4a00563 call dword ptr [ModuleA!_imp__OutputDebugStringA (6305a0b4)]
63050012 8b4e14 mov ecx,dword ptr [esi+14h]
63050015 85c9 test ecx,ecx
63050017 740e je ModuleA!FunctionA+0xe7 (63050027)
63050019 8b11 mov edx,dword ptr [ecx]
6305001b 8b4204 mov eax,dword ptr [edx+4]

0:000> ub 63050001
^ Unable to find valid previous instruction for 'ub 63050001'

0:000> ub 63050001-1
^ Unable to find valid previous instruction for 'ub 63050001-1'

0:000> ub 63050001-2
^ Unable to find valid previous instruction for 'ub 63050001-2'

0:000> ub 63050001-3
ModuleA!FunctionA+0xa4:
6304ffe4 ffd2      call edx
6304ffe6 84c0      test al,al
6304ffe8 7541      jne ModuleA!FunctionA+0xeb (6305002b)
6304ffa 6818c20563 push offset ModuleA!`string' (6305c218)
6304ffef 68bcc10563 push offset ModuleA!`string' (6305c1bc)
6304ffff 8d442418 lea eax,[esp+18h]
6304ffff 6800010000 push 100h
6304ffffd 50       push eax
```

## Wild Pointer

This pattern was briefly mentioned in **Local Buffer Overflow** (page 611) which shows examples of pointers having UNICODE structure in their values. We can also observe pointers with ASCII structure as **.formats** WinDbg command indicates:

```
FAILED_INSTRUCTION_ADDRESS:
65747379`735c5357 ??          ???

IP_ON_HEAP: 65747379735c5357

0:012> .formats rip
Evaluate expression:
Hex:      65747379`735c5357
Decimal: 7310595060592825175
Octal:   0625643467456327051527
Binary:  01100101 01110100 01110011 01111001 01110011 01011100 01010011 01010111
Chars:   etsys\SW
Time:    Wed May 10 22:00:59.282 24767 (GMT+1)
Float:   low 1.7456e+031 high 7.21492e+022
Double:  5.30388e+180
```

Here is another example of a pointer having UNICODE structure:

```
FAILED_INSTRUCTION_ADDRESS:
0045004c`00490046 ??          ???

0:014> .formats rip
Evaluate expression:
Hex:      0045004c`00490046
Decimal: 19422099815333958
Octal:   0001050004600022200106
Binary:  00000000 01000101 00000000 01001100 00000000 01001001 00000000 01000110
Chars:   .E.L.I.F
Time:    Wed Jul 19 07:46:21.533 1662 (GMT+1)
Float:   low 6.70409e-039 high 6.33676e-039
Double:  2.33646e-307
```

When we have EIP or RIP pointers, we have another pattern to name when the value coincidentally lies inside some valid region of memory: **Wild Code** (page 1148). Here is one example of the latter pattern:

```
IP_ON_STACK:
+e1ffa8

STACK_TEXT:
0e11ff7c 098eeef2 0xe1ffa8
0e11ff84 77b6b530 dll!StartWorking+0xcab
0e11ffb8 7c826063 msvcrt!_endthreadex+0xa3
0e11ffec 00000000 kernel32!BaseThreadStart+0x34
```

```
0:020> u
0e11ffa8 dcff fdiv st(7),st
...
```

We see that EIP is very close to EBP/ESP, and this explains why `!analyze -v` reports IP\_ON\_STACK. Clearly floating-point code is not what we should expect. This example shows that wild pointers sometimes are valid but either through code chain or pointer chain the execution reaches **Invalid Pointer** (page 589) and a process or system crashes.

## Comments

The following example shows UNICODE data in RCX register:

```
0:001> k
# Child-SP          RetAddr           Call Site
00 00000005`8d71deb8 00007ffb`322b918f ntdll!NtWaitForMultipleObjects+0xa
01 00000005`8d71dec0 00007ffb`322b908e KERNELBASE!WaitForMultipleObjectsEx+0xef
02 00000005`8d71e1c0 00007ffb`32c2155c KERNELBASE!WaitForMultipleObjects+0xe
03 00000005`8d71e200 00007ffb`32c21088 kernel32!WerReportFaultInternal+0x494
04 00000005`8d71e770 00007ffb`322e03cd kernel32!WerReportFault+0x48
05 00000005`8d71e7a0 00007ffb`34e48cd2 KERNELBASE!UnhandledExceptionFilter+0x1fd
06 00000005`8d71e8a0 00007ffb`34e34296 ntdll!RtlUserThreadStart$filt$0+0x3e
07 00000005`8d71e8e0 00007ffb`34e4666d ntdll!_C_specific_handler+0x96
08 00000005`8d71e950 00007ffb`34dc3c00 ntdll!RtlpExecuteHandlerForException+0xd
09 00000005`8d71e980 00007ffb`34e4577a ntdll!RtlDispatchException+0x370
0a 00000005`8d71f080 00007ffb`2fb57749 ntdll!KiUserExceptionDispatch+0x3a
0b 00000005`8d71f790 00007ffb`2fbafba5 uDWM!CBaseObject::Release+0x15
0c 00000005`8d71f7c0 00007ffb`2fb9f6dc uDWM!CWindowData::SetIconicBitmap+0x21
0d 00000005`8d71f7f0 00007ffb`2fbaddde uDWM!ScalingCompatLogging::Instance'::`2'::`dynamic atexit
destructor for 'wrapper''+0x1497c
0e 00000005`8d71f820 00007ffb`2fbae13b uDWM!CIconicBitmapRegistry::AcceptBitmap+0x56
0f 00000005`8d71f860 00007ffb`2fbb9d09 uDWM!CIconicBitmapRegistry::BitmapReceived+0x267
10 00000005`8d71fa50 00007ffb`2fb9d9b0 uDWM!CWindowList::SetIconicThumbnail+0x105
11 00000005`8d71fac0 00007ffb`2fb12c0f uDWM!ScalingCompatLogging::Instance'::`2'::`dynamic atexit
destructor for 'wrapper''+0x12c50
12 00000005`8d71fc40 00007ffb`2fb1d171 dwmredir!CSessionPort::ProcessCommand+0x34f
13 00000005`8d71fcf0 00007ffb`2fb1c889 dwmredir!CPortBase::PortThreadInternal+0x241
14 00000005`8d71fda0 00007ffb`32c12d92 dwmredir!CPortBase::PortThread+0x9
15 00000005`8d71fdd0 00007ffb`34db9f64 kernel32!BaseThreadInitThunk+0x22
16 00000005`8d71fe00 00000000`00000000 ntdll!RtlUserThreadStart+0x34

0:001> .cxr 00000005`8d71f080
rax=0000000000000001 rbx=00390035003a0039 rcx=00390035003a0039
rdx=0000000000000000 rsi=00000000ffffffffff rdi=00000005919e6f20
rip=00007ffb2fb57749 rsp=000000058d71f790 rbp=000000058d71f960
 r8=00000000000002e3 r9=00000000000002e3 r10=000000058bcc7630
r11=000000059400fb0 r12=000000058bcc7620 r13=00000005920e0000
r14=000000000000047 r15=0000000000000006c
iopl=0          nv up ei ng nz na po nc
cs=0033 ss=002b ds=002b es=002b fs=0053 gs=002b                 efl=00010286
uDWM!CBaseObject::Release+0x15:
00007ffb`2fb57749 f00fc17108      lock xadd dword ptr [rcx+8],esi ds:00390035`003a0041=????????
```

## Window Hint

Windows processes may contain **Execution Residue** (page 365) such as ASCII window class names in mapped memory regions pointing to other running processes (perhaps as a result of **Hookware**, page 1175). For example, *calc.exe* process memory dump saved on my Windows 10 notebook “knows” about Visio and WinDbg windows that were opened at that time:

```
0:000> s-a 0 L?XXXXXXXXXXXXXXX "VISIOA"
00000015`42c6bdd0 56 49 53 49 4f 41 00 00-00 00 00 00 00 00 00 VISIOA.....
0:000> s-a 0 L?XXXXXXXXXXXXXXX "WinDbg"
00000015`42d19720 57 69 6e 44 62 67 46 72-61 6d 65 43 6c 61 73 73 WinDbgFrameClass
```

This may be useful for some troubleshooting scenarios, for example, pointing to processes which are known for their problematic behavior or **Special Processes** (page 877). Of course, we assume that those windows or classes were genuine, not faked. We call this analysis pattern **Window Hint** similar to **Environment Hint** (page 324) and **Module Hint** (page 696) analysis patterns.

Going deeper, we can dump strings from the whole region limiting the output to the strings with length more than 5:

```
0:000> !address 00000015`42d19720

Usage: <unknown>
Base Address: 00000015`42b20000
End Address: 00000015`42d3a000
Region Size: 00000000`0021a000 ( 2.102 MB)
State: 00001000 MEM_COMMIT
Protect: 00000002 PAGE_READONLY
Type: 00040000 MEM_MAPPED
Allocation Base: 00000015`42b20000
Allocation Protect: 00000002 PAGE_READONLY

Content source: 1 (target), length: 208e0

0:000> s-[15]sa 00000015`42b20000 00000015`42d3a000
00000015`42b20a60 "#32769"
00000015`42b20cc0 "Message"
00000015`42b20f40 "#32774"
00000015`42b21060 "#32772"
00000015`42b21510 "Ghost"
00000015`42b215e0 "LivePreview"
00000015`42b216f0 "UserAdapterWindowClass"
00000015`42b21ce0 "MSCTFIME Composition"
00000015`42b222a0 "#32772"
00000015`42b22390 "#32772"
00000015`42b22460 "RichEdit20W"
00000015`42b22530 "RichEdit20A"
00000015`42b22600 "ToolbarWindow32"
00000015`42b226e0 "tooltips_class32"
```

```

00000015`42b227c0 "msctls_statusbar32"
00000015`42b228a0 "SysListView32"
00000015`42b22980 "SysHeader32"
00000015`42b22a50 "SysTabControl32"
00000015`42b22b30 "SysTreeView32"
00000015`42b22c10 "msctls_trackbar32"
00000015`42b22cf0 "msctls_updown32"
00000015`42b22dd0 "msctls_progress32"
00000015`42b22eb0 "msctls_hotkey32"
00000015`42b22f8f "SysAnimate32"
00000015`42b230f0 "SysIPAddress32"
00000015`42b231d0 "ReBarWindow32"
00000015`42b232b0 "ComboBoxEx32"
00000015`42b23390 "SysMonthCal32"
00000015`42b23470 "SysDateTimePick32"
00000015`42b23550 "DropDown"
00000015`42b23620 "SysLink"
00000015`42b236f0 "SysPager"
00000015`42b23960 "msctls_netaddress"

[...]

00000015`42d175e0 "OutlookFbThreadWnd"
00000015`42d19720 "WinDbgFrameClass"
00000015`42d19750 "DockClass"
00000015`42d19770 "GhostClass"
00000015`42d19a30 "ATL:00007FF60D792730"
00000015`42d1a0f0 "MSCTIME Composition"
00000015`42d1a4af "%OleMainThreadWndClass"
00000015`42d1be10 "CicMarshalWndClass"
00000015`42d1c0e0 "VSyncHelper-00000040EC4CA5F0-1f8"
00000015`42d1c100 "8855daf"
00000015`42d1c190 "URL Moniker Notification Window"
00000015`42d1c390 "UserAdapterWindowClass"
00000015`42d1d080 "@>zG#"
00000015`42d1dcfa "!VSyncHelper-00000040D60C5850-1e"
00000015`42d1dccf "ef0477df"
00000015`42d20d50 "VSyncHelper-00000040F39C5650-1f0"
00000015`42d20d70 "313c5a0"
00000015`42d250d0 "#32770"
00000015`42d250f0 "URL Moniker Notification Window"
00000015`42d29270 "VSyncHelper-00000079321C32E0-1f2"
00000015`42d29290 "fb11f8c"
00000015`42d2a1d0 "MSCTIME Composition"
00000015`42d2a480 "CicMarshalWndClass"
00000015`42d2ac80 "MSCTIME Composition"
00000015`42d2b8d0 "ShockwaveFlashFullScreen"
00000015`42d2bbb8 "P?U!\""
00000015`42d2c690 "Xaml_WindowedPopupClass"
00000015`42d30a10 "ShockwaveFlashFullScreen"
00000015`42d30b50 "MSCTIME UI"
00000015`42d30b90 "WinBaseClass"
00000015`42d3441f "IAlternate Owner"
00000015`42d34460 "ShockwaveFlashFullScreen"
00000015`42d344a0 "ATL:00007FF60D792530"
00000015`42d34a50 "SysAnimate32"

```

```
00000015`42d34a7f  "'ComboBoxEx32"
00000015`42d34ed0  "tooltips_class32"
00000015`42d34f00  "msctls_statusbar32"
00000015`42d35e70  "RawInputClass"
00000015`42d36a10  "SysTabControl32"
00000015`42d38650  "CicMarshalWndClass"
00000015`42d38eb0  "#32772"
00000015`42d3951f  "!VSyncHelper-000000C9DA06CD10-1f"
00000015`42d3953f  "110e8d16"
```

## Y

## Young System

Opposite to **Oversized System** (page 774), sometimes we can see **Young System** pattern. It means that the system didn't have time to initialize and subsequently mature or reach the state when the problem could surface. Usual signs are less than a minute system uptime (or larger, depends on a problem context) and the low number of processes and services running. Also, sometimes, the problem description mentions a terminal services session, but there is only one console session in the dump or two as in Vista and Windows Server 2008:

```
System Uptime: 0 days 0:00:18.562
```

```
3: kd> !vm
```

```
...
```

0248 lsass.exe	1503 (	6012 Kb)
020c winlogon.exe	1468 (	5872 Kb)
03b8 svchost.exe	655 (	2620 Kb)
023c services.exe	416 (	1664 Kb)
01f0 csrss.exe	356 (	1424 Kb)
0338 svchost.exe	298 (	1192 Kb)
02dc svchost.exe	259 (	1036 Kb)
0374 svchost.exe	240 (	960 Kb)
039c svchost.exe	224 (	896 Kb)
01bc smss.exe	37 (	148 Kb)
0004 System	8 (	32 Kb)

```
3: kd> !session
```

```
Sessions on machine: 1
```

```
Valid Sessions: 0
```

In the case of the fully initialized system **Manual Dump** (page 625) might have been taken after reboot when the bugcheck already happened or any other reason stemming from the usual confusion between crashes and hangs<sup>237</sup>.

Similar considerations apply to a young process as well, where *Process Uptime* value from user dumps or *Elapsed Time* value from kernel or complete memory dumps is too small unless we have the obvious crash or hang signs inside, for example, exceptions, **Deadlock** (page 182), **Wait Chain** (page 1082) or **Blocked Thread** (page 82) waiting for another **Coupled Process** (page 148):

---

<sup>237</sup> Crashes and Hangs Differentiated, Memory Dump Analysis Anthology, Volume 1, page 36

```

Process Uptime: 0 days 0:00:10.000

3: kd> !process 8a389d88
PROCESS 8a389d88 SessionId: 0 Cid: 020c Peb: 7ffdf000 ParentCid: 01bc
  DirBase: 7fbe6080 ObjectTable: e1721008 HandleCount: 455.
  Image: winlogon.exe
  VadRoot 8a65d070 Vads 194 Clone 0 Private 1166. Modified 45. Locked 0.
  DeviceMap e10030f8
  Token e139bde0
  ElapsedTime 00:00:01.062
  UserTime 00:00:00.046
  KernelTime 00:00:00.015
  QuotaPoolUsage[PagedPool] 71228
  QuotaPoolUsage[NonPagedPool] 72232
  Working Set Sizes (now,min,max) (2265, 50, 345) (9060KB, 200KB, 1380KB)
  PeakWorkingSetSize 2267
  VirtualSize 41 Mb
  PeakVirtualSize 42 Mb
  PageFaultCount 2605
  MemoryPriority BACKGROUND
  BasePriority 13
  CommitCharge 1468

```

## Comments

---

Another example of no sessions:

```

0: kd> !session
There are ZERO session on machine.

0: kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS ffffffa8003c77b10
SessionId: none Cid: 0004 Peb: 00000000 ParentCid: 0000
DirBase: 00187000 ObjectTable: fffff8a0000012f0 HandleCount: 231.
Image: System

PROCESS ffffffa80050afb10
SessionId: none Cid: 0198 Peb: 7fffffdf000 ParentCid: 0004
DirBase: a9cf0000 ObjectTable: fffff8a000525970 HandleCount: 22.
Image: smss.exe

```

## Zombie Processes

Sometimes, when listing processes, we see the so-called **Zombie Processes**. They are better visible in the output of **!vm** command as processes with zero private memory size values:

```
0: kd> !vm

*** Virtual Memory Usage ***
Physical Memory:    999294 ( 3997176 Kb)
Page File: \??\C:\pagefile.sys
Current: 5995520 Kb  Free Space: 5324040 Kb
Minimum: 5995520 Kb  Maximum: 5995520 Kb
Available Pages: 626415 ( 2505660 Kb)
ResAvail Pages: 902639 ( 3610556 Kb)
Locked IO Pages: 121 ( 484 Kb)
Free System PTEs: 201508 ( 806032 Kb)
Free NP PTEs: 32766 ( 131064 Kb)
Free Special NP: 0 ( 0 Kb)
Modified Pages: 256 ( 1024 Kb)
Modified PF Pages: 256 ( 1024 Kb)
NonPagedPool Usage: 12304 ( 49216 Kb)
NonPagedPool Max: 65359 ( 261436 Kb)
PagedPool 0 Usage: 18737 ( 74948 Kb)
PagedPool 1 Usage: 2131 ( 8524 Kb)
PagedPool 2 Usage: 2104 ( 8416 Kb)
PagedPool 3 Usage: 2140 ( 8560 Kb)
PagedPool 4 Usage: 2134 ( 8536 Kb)
PagedPool Usage: 27246 ( 108984 Kb)
PagedPool Maximum: 67072 ( 268288 Kb)
Shared Commit: 60867 ( 243468 Kb)
Special Pool: 0 ( 0 Kb)
Shared Process: 14359 ( 57436 Kb)
PagedPool Commit: 27300 ( 109200 Kb)
Driver Commit: 1662 ( 6648 Kb)
Committed pages: 501592 ( 2006368 Kb)
Commit limit: 2456879 ( 9827516 Kb)

Total Private: 368810 ( 1475240 Kb)
...
 3654 explorer.exe      2083 ( 8332 Kb)
 037c MyService.exe     2082 ( 8328 Kb)
 315c explorer.exe      2045 ( 8180 Kb)
...
 0588 svchost.exe       360 ( 1440 Kb)
 3f94 csrss.exe         288 ( 1152 Kb)
 0acc svchost.exe       245 ( 980 Kb)
 0380 smss.exe          38 ( 152 Kb)
 0004 System             7 ( 28 Kb)
 6ee8 cmd.exe            0 ( 0 Kb)
 6d7c cmd.exe            0 ( 0 Kb)
 6ca8 cmd.exe            0 ( 0 Kb)
```

6b48 IEXPLORE.EXE	0 (	0 Kb)
6ac4 cmd.exe	0 (	0 Kb)
69e8 cmd.exe	0 (	0 Kb)
69dc cmd.exe	0 (	0 Kb)
68dc AcroRd32.exe	0 (	0 Kb)
6860 cmd.exe	0 (	0 Kb)
6858 cmd.exe	0 (	0 Kb)
67d8 cmd.exe	0 (	0 Kb)
6684 AcroRd32.exe	0 (	0 Kb)
6484 cmd.exe	0 (	0 Kb)
6464 cmd.exe	0 (	0 Kb)
6288 cmd.exe	0 (	0 Kb)
626c cmd.exe	0 (	0 Kb)
6260 cmd.exe	0 (	0 Kb)
6258 cmd.exe	0 (	0 Kb)
620c IEXPLORE.EXE	0 (	0 Kb)
60f0 cmd.exe	0 (	0 Kb)
5fa4 cmd.exe	0 (	0 Kb)
5f60 cmd.exe	0 (	0 Kb)
5eec cmd.exe	0 (	0 Kb)
5d24 IEXPLORE.EXE	0 (	0 Kb)
5bd4 cmd.exe	0 (	0 Kb)
5b9c cmd.exe	0 (	0 Kb)
5b10 cmd.exe	0 (	0 Kb)
5b08 cmd.exe	0 (	0 Kb)
5a4c cmd.exe	0 (	0 Kb)
5a08 cmd.exe	0 (	0 Kb)
5934 cmd.exe	0 (	0 Kb)
58b8 cmd.exe	0 (	0 Kb)
56dc cmd.exe	0 (	0 Kb)
558c cmd.exe	0 (	0 Kb)
5588 cmd.exe	0 (	0 Kb)
5574 cmd.exe	0 (	0 Kb)
5430 cmd.exe	0 (	0 Kb)
5424 cmd.exe	0 (	0 Kb)
53b0 cmd.exe	0 (	0 Kb)
5174 explorer.exe	0 (	0 Kb)
5068 cmd.exe	0 (	0 Kb)
5028 IEXPLORE.EXE	0 (	0 Kb)
5004 cmd.exe	0 (	0 Kb)
4f3c javaw.exe	0 (	0 Kb)
4de4 cmd.exe	0 (	0 Kb)
4dd8 cmd.exe	0 (	0 Kb)
4c50 cmd.exe	0 (	0 Kb)
4c48 cmd.exe	0 (	0 Kb)
4c08 cmd.exe	0 (	0 Kb)
4a8c cmd.exe	0 (	0 Kb)
49ac cmd.exe	0 (	0 Kb)
4938 cmd.exe	0 (	0 Kb)
4928 cmd.exe	0 (	0 Kb)
491c cmd.exe	0 (	0 Kb)
4868 POWERPNT.EXE	0 (	0 Kb)
4724 cmd.exe	0 (	0 Kb)
46cc cmd.exe	0 (	0 Kb)
44a8 cmd.exe	0 (	0 Kb)
43cc cmd.exe	0 (	0 Kb)

4350 cmd.exe	0 (	0 Kb)
4208 cmd.exe	0 (	0 Kb)
41f4 cmd.exe	0 (	0 Kb)
41ec cmd.exe	0 (	0 Kb)
4170 cmd.exe	0 (	0 Kb)
40bc cmd.exe	0 (	0 Kb)
3ddc cmd.exe	0 (	0 Kb)
3dcc cmd.exe	0 (	0 Kb)
3db8 cmd.exe	0 (	0 Kb)
3d88 cmd.exe	0 (	0 Kb)
3d10 cmd.exe	0 (	0 Kb)
3cac cmd.exe	0 (	0 Kb)
3ca4 cmd.exe	0 (	0 Kb)
3c88 cmd.exe	0 (	0 Kb)
337c cmd.exe	0 (	0 Kb)
3310 cmd.exe	0 (	0 Kb)
3308 cmd.exe	0 (	0 Kb)
32f0 cmd.exe	0 (	0 Kb)
32b8 cmd.exe	0 (	0 Kb)
2ed0 cmd.exe	0 (	0 Kb)
2eb8 cmd.exe	0 (	0 Kb)
2e28 cmd.exe	0 (	0 Kb)
2d44 AcroRd32.exe	0 (	0 Kb)
2d24 cmd.exe	0 (	0 Kb)
2c94 cmd.exe	0 (	0 Kb)
2c54 IEXPLORE.EXE	0 (	0 Kb)
2a28 cmd.exe	0 (	0 Kb)
29e4 cmd.exe	0 (	0 Kb)
2990 cmd.exe	0 (	0 Kb)
28c0 cmd.exe	0 (	0 Kb)
25a0 cmd.exe	0 (	0 Kb)
2558 cmd.exe	0 (	0 Kb)
2478 cmd.exe	0 (	0 Kb)
244c cmd.exe	0 (	0 Kb)
23dc cmd.exe	0 (	0 Kb)
2320 cmd.exe	0 (	0 Kb)
2280 cmd.exe	0 (	0 Kb)
2130 cmd.exe	0 (	0 Kb)
205c cmd.exe	0 (	0 Kb)
2014 cmd.exe	0 (	0 Kb)
1fd8 cmd.exe	0 (	0 Kb)
1fa0 cmd.exe	0 (	0 Kb)
1eb8 cmd.exe	0 (	0 Kb)
1d68 IEXPLORE.EXE	0 (	0 Kb)
1cb8 cmd.exe	0 (	0 Kb)
1c9c cmd.exe	0 (	0 Kb)
1c50 cmd.exe	0 (	0 Kb)
1a74 cmd.exe	0 (	0 Kb)
1954 cmd.exe	0 (	0 Kb)
1948 cmd.exe	0 (	0 Kb)
06e4 cmd.exe	0 (	0 Kb)
0650 cmd.exe	0 (	0 Kb)

We see lots of cmd.exe processes. Let's examine a few of them:

```
0: kd> !process 0650
Searching for Process with Cid == 650
PROCESS 89237d88 SessionId: 0 Cid: 0650 Peb: 7ffdde000 ParentCid: 037c
  DirBase: f3b31940 ObjectTable: 00000000 HandleCount: 0.
  Image: cmd.exe
  VadRoot 00000000 Vads 0 Clone 0 Private 0. Modified 2. Locked 0.
  DeviceMap e10038a8
  Token e4eb5b98
  ElapsedTime 1 Day 00:16:11.706
  UserTime 00:00:00.015
  KernelTime 00:00:00.015
  QuotaPoolUsage[PagedPool] 0
  QuotaPoolUsage[NonPagedPool] 0
  Working Set Sizes (now,min,max) (7, 50, 345) (28KB, 200KB, 1380KB)
  PeakWorkingSetSize 588
  VirtualSize 11 Mb
  PeakVirtualSize 14 Mb
  PageFaultCount 663
  MemoryPriority BACKGROUND
  BasePriority 8
  CommitCharge 0
```

No active threads

```
0: kd> !process 2130
Searching for Process with Cid == 2130
PROCESS 89648020 SessionId: 0 Cid: 2130 Peb: 7fffdc000 ParentCid: 037c
  DirBase: f3b31060 ObjectTable: 00000000 HandleCount: 0.
  Image: cmd.exe
  VadRoot 00000000 Vads 0 Clone 0 Private 0. Modified 2. Locked 0.
  DeviceMap e10038a8
  Token e5167bb8
  ElapsedTime 15:40:17.643
  UserTime 00:00:00.015
  KernelTime 00:00:00.000
  QuotaPoolUsage[PagedPool] 0
  QuotaPoolUsage[NonPagedPool] 0
  Working Set Sizes (now,min,max) (7, 50, 345) (28KB, 200KB, 1380KB)
  PeakWorkingSetSize 545
  VirtualSize 11 Mb
  PeakVirtualSize 14 Mb
  PageFaultCount 621
  MemoryPriority BACKGROUND
  BasePriority 8
  CommitCharge 0
```

No active threads

Most of them have Parent PID as 037c, which is *MyService.exe*. Let's peek inside its handle table:

```
0: kd> !kdexts.handle 0 3 037c
processor number 0, process 0000037c
Searching for Process with Cid == 37c
PROCESS 8a8fa8c0 SessionId: 0 Cid: 037c Peb: 7ffd8000 ParentCid: 04ac
DirBase: f3b10360 ObjectTable: e1c276b8 HandleCount: 500.
Image: MyService.exe

Handle table at e272d000 with 500 Entries in use
0004: Object: e1000638 GrantedAccess: 00000003 Entry: e1caf008
Object: e1000638 Type: (8ad79ad0) KeyedEvent
    ObjectHeader: e1000620 (old version)
        HandleCount: 151 PointerCount: 152
        Directory Object: e1001898 Name: CritSecOutOfMemoryEvent

0008: Object: 8a8cfdf8 GrantedAccess: 001f0003 Entry: e1caf010
Object: 8a8cfdf8 Type: (8ad7a990) Event
    ObjectHeader: 8a8cfde0 (old version)
        HandleCount: 1 PointerCount: 1

000c: Object: e186d690 GrantedAccess: 00000003 Entry: e1caf018
Object: e186d690 Type: (8ad84e70) Directory
    ObjectHeader: e186d678 (old version)
        HandleCount: 150 PointerCount: 181
        Directory Object: e1003b28 Name: KnownDlls

0010: Object: 8a8d1328 GrantedAccess: 00100020 (Inherit) Entry: e1caf020
Object: 8a8d1328 Type: (8ad74900) File
    ObjectHeader: 8a8d1310 (old version)
        HandleCount: 1 PointerCount: 1
    Directory Object: 00000000 Name: \WINDOWS\system32 {HarddiskVolume1}
...
...
...
0484: Object: 89648020 GrantedAccess: 001f0fff Entry: e1caf908
Object: 89648020 Type: (8ad84900) Process
    ObjectHeader: 89648008 (old version)
        HandleCount: 1 PointerCount: 2
...
...
...
0510: Object: 89237d88 GrantedAccess: 001f0fff Entry: e1cafa20
Object: 89237d88 Type: (8ad84900) Process
    ObjectHeader: 89237d70 (old version)
        HandleCount: 1 PointerCount: 2
...
...
...
```

We may guess that *MyService.exe* probably forgot to close process handles either after launching *cmd.exe* or after waiting for their exit when process objects become signaled:

```
0510: Object: 89237d88 GrantedAccess: 001f0fff Entry: e1cafa20
Object: 89237d88 Type: (8ad84900) Process
    ObjectHeader: 89237d70 (old version)
        HandleCount: 1 PointerCount: 2

0: kd> dt _DISPATCHER_HEADER 89237d88
ntdll!_DISPATCHER_HEADER
+0x000 Type          : 0x3 '' ; PROCESS OBJECT
+0x001 Absolute      : 0 ''
+0x001 NpxIrql       : 0 ''
+0x002 Size          : 0x1e ''
+0x002 Hand          : 0x1e ''
+0x003 Inserted      : 0 ''
+0x003 DebugActive   : 0 ''
+0x000 Lock          : 1966083
+0x004 SignalState   : 1
+0x008 WaitListHead  : _LIST_ENTRY [ 0x89237d90 - 0x89237d90 ]
```

This pattern can also be seen a specialization of a more general **Handle Leak** pattern (page 526).

## Comments

---

We may want to enable object reference tracing for *Proc* pool tag if we have very large number of such processes especially if they are not reflected as some parent process handle leak. Please see **!obtrace** example in WinDbg help. There is also **Reference Leak** pattern (page 830).

If PPID doesn't exist anymore we can dump all handle tables from all processes and search for our process object:

```
!handle 0 3 0 Process
```

There was a question:

**Q.** The following output is from LiveKD; in VM we observed 8000+ cmd.exe instances running in System context. How can system uptime be 22 days and KernelTime is 700+ Days for that processor?

```
1kd> .time
Debug session time: Mon Apr 12 06:07:15.797 2010 (GMT-4)
System Uptime: 22 days 10:41:32.579

1kd> !process 00a4
Searching for Process with Cid == a4
Cid Handle table at e29a6000 with 10925 Entries in use
PROCESS fb32cd88 SessionId: 0 Cid: 00a4 Peb: 7ffff000 ParentCid: 2ce0
DirBase: bff54960 ObjectTable: e3c046a0 HandleCount: 0.
Image: cmd.exe
VadRoot fcc99398 Vads 6 Clone 0 Private 4. Modified 0. Locked 0.
DeviceMap e16008e8
Token e3ce6030
Elapsed Time 21 Days 14:20:04.071
User Time 00:00:00.000
Kernel Time 762 Days 03:42:46.250
QuotaPoolUsage[PagedPool] 6780
QuotaPoolUsage[NonPagedPool] 160
Working Set Sizes (now,min,max) (16, 50, 345) (64KB, 200KB, 1380KB)
PeakWorkingSetSize 16
VirtualSize 1 Mb
PeakVirtualSize 1 Mb
PageFaultCount 9
MemoryPriority BACKGROUND
BasePriority 10
CommitCharge 39

No active threads
```

**A.** We find this in many dumps and have a suspicion it is related to "orphaned" processes:

```
HandleCount: 0
No active threads
```

## Bibliography

- Accelerated .NET Memory Dump Analysis, Second Edition: Training Course Transcript and WinDbg Practice Exercises (ISBN: 978-1-908043597)
- Accelerated Disassembly, Reconstruction and Reversing: Training Course Transcript and WinDbg Practice Exercises with Memory Cell Diagrams (ISBN: 978-1-908043672)
- Accelerated Linux Core Dump Analysis: Training Course Transcript with GDB Practice Exercises (ISBN: 978-1-908043979)
- Accelerated Mac OS X Core Dump Analysis, Second Edition: Training Course Transcript with GDB and LLDB Practice Exercises (ISBN: 978-1-908043719)
- Accelerated Windows Debugging 3: Training Course Transcript and WinDbg Practice Exercises (ISBN: 978-1-908043566)
- Accelerated Windows Malware Analysis with Memory Dumps: Training Course Transcript and WinDbg Practice Exercises (ISBN: 978-1-908043443)
- Accelerated Windows Memory Dump Analysis: Training Course Transcript and WinDbg Practice Exercises with Notes, Fourth Edition (ISBN: 978-1-908043467)
- Advanced Windows Memory Dump Analysis with Data Structures: Training Course Transcript and WinDbg Practice Exercises with Notes, Second Edition (ISBN: 978-0-955832888)
- Advanced Windows RT Memory Dump Analysis, ARM Edition: Training Course Transcript and WinDbg Practice Exercises (ISBN: 978-1-908043733)
- Fundamentals of Physical Memory Analysis (ISBN: 978-1-906717155)
- Introduction to Pattern-Driven Software Problem Solving (ISBN: 978-1-908043177)
- Memory Dump Analysis Anthology, Volume 1 (ISBN: 978-0-955832802)
- Memory Dump Analysis Anthology, Volume 2 (ISBN: 978-0-955832871)
- Memory Dump Analysis Anthology, Volume 3 (ISBN: 978-1-906717438)
- Memory Dump Analysis Anthology, Volume 4 (ISBN: 978-1-906717865)
- Memory Dump Analysis Anthology, Volume 5 (ISBN: 978-1-906717964)
- Memory Dump Analysis Anthology, Volume 6 (ISBN: 978-1-908043191)
- Memory Dump Analysis Anthology, Volume 7 (ISBN: 978-1-908043511)
- Memory Dump Analysis Anthology, Volume 8a (ISBN: 978-1-908043535)
- Memory Dump Analysis Anthology, Volume 8b (ISBN: 978-1-908043542)
- Memory Dump Analysis Anthology, Volume 9a (ISBN: 978-1-908043351)
- Memory Dump Analysis Anthology, Volume 9b (ISBN: 978-1-908043368)
- Pattern-Based Software Diagnostics: An Introduction (ISBN: 978-1-908043498)
- Pattern-Driven Software Diagnostics: An Introduction (ISBN: 978-1-908043382)
- Pattern-Oriented Memory Forensics: A Pattern Language Approach (ISBN: 978-1-908043764)
- Practical Foundations of Windows Debugging, Disassembling, Reversing: Training Course (ISBN: 978-1-908043948)
- Software Trace and Memory Dump Analysis: Patterns, Tools, Processes and Best Practices (ISBN: 978-1-908043238)
- Systemic Software Diagnostics: An Introduction (ISBN: 978-1908043399)
- The Old New Crash: Cloud Memory Dump Analysis (ISBN: 978-1-908043283)
- Theoretical Software Diagnostics: Collected Articles (ISBN: 978-1-908043986)
- Victimware: The Missing Part of the Equation (ISBN: 978-1-908043634)

## Appendix A

### Reference Stack Traces

The following volumes contain normal thread stacks and other information from Windows complete memory dumps:

[http://www.dumpanalysis.org/reference/ReferenceStackTraces\\_Vista\\_x86.pdf](http://www.dumpanalysis.org/reference/ReferenceStackTraces_Vista_x86.pdf)

[http://www.dumpanalysis.org/reference/ReferenceStackTraces\\_Vista\\_x64.pdf](http://www.dumpanalysis.org/reference/ReferenceStackTraces_Vista_x64.pdf)

[http://www.dumpanalysis.org/reference/ReferenceStackTraces\\_Windows2003\\_x86.pdf](http://www.dumpanalysis.org/reference/ReferenceStackTraces_Windows2003_x86.pdf)

Such references are useful when trying to spot anomalies in complete and kernel memory dumps coming from problem servers and workstations. For updates and future volumes, please check DumpAnalysis.org.

## Appendix B

### .NET / CLR / Managed Space Patterns

Windows:

- **Annotated Disassembly** (JIT .NET code) (page 75)
- **Caller-n-Callee** (page 111)
- **CLR Thread** (page 124)
- **Deadlock** (managed space, page 202)
- **Distributed Exception** (managed code, page 243)
- **Duplicate Extension** (page 283)
- **Dynamic Memory Corruption** (managed heap, page 301)
- **Execution Residue** (managed space, page 369)
- **Handled Exception** (.NET CLR, page 428)
- **Inline Function Optimization** (managed code, page 512)
- **JIT Code** (.NET) (page 591)
- **Managed Code Exception** (page 617)
- **Managed Stack Trace** (page 624)
- **Memory Leak** (.NET heap) (page 636)
- **Mixed Exception** (page 688)
- **Multiple Exceptions** (managed space, page 713)
- **Nested Exceptions** (managed code, page 723)
- **Object Distribution Anomaly** (.NET heap, page 756)
- **Special Thread** (.NET CLR, page 883)
- **Stack Trace Collection** (managed space, page 940)
- **Technology-Specific Subtrace** (JIT .NET code, page 994)
- **Version-Specific Extension** (page 1058)
- **Wait Chain** (CLR monitors, page 1082)

## Contention Patterns

Windows:

- **High Contention** (.NET CLR monitors, page 472)
- **High Contention** (critical sections, page 475)
- **High Contention** (executive resources, page 477)
- **High Contention** (processors, page 480)

Linux:

- **Critical Region** (page 157)

## Deadlock and Livelock Patterns

Windows:

- **Deadlock** (critical sections, page 182)
- **Deadlock** (executive resources, page 194)
- **Deadlock** (LPC/ALPC, page 197)
- **Deadlock** (managed space, page 202)
- **Deadlock** (mixed objects, kernel space, page 205)
- **Deadlock** (mixed objects, user space, page 210)
- **Deadlock** (self, page 218)
- **Livelock** (page 606)

## DLL Link Patterns

Windows:

- **Duplicated Module** (page 287)
- **Missing Component** (general, page 668)
- **Missing Component** (static linking, user mode, page 672)
- **Unloaded Module** (page 1041)

## Dynamic Memory Corruption Patterns

Windows:

- **Double Free** (kernel pool, page 260)
- **Double Free** (process heap, page 267)
- **Dynamic Memory Corruption** (kernel pool, page 292)
- **Dynamic Memory Corruption** (managed heap, page 301)
- **Dynamic Memory Corruption** (process heap, page 307)

Mac OS X:

- **Double Free** (process heap, page 278)
- **Dynamic Memory Corruption** (process heap, page 305)

Linux:

- **Dynamic Memory Corruption** (process heap, page 304)

## Executive Resource Patterns

Executive resource (ERESOURCE) patterns may help with analysis of the output of **!locks** WinDbg command from Windows kernel and complete memory dumps:

- **Accidental Lock** (page 53)
- **Deadlock** (executive resources, page 194)
- **Deadlock** (mixed objects, kernel space, page 205)
- **Deadlock** (self, page 218)
- **High Contention** (executive resources, page 477)
- **Livelock** (page 606)
- **Semantic Split** (page 853)
- **Swarm of Shared Locks** (page 966)
- **Wait Chain** (executive resources, page 1089)

## Exception Patterns

Windows:

- **C++ Exception** (page 110)
- **Custom Exception Handler** (kernel space, page 169)
- **Custom Exception Handler** (user space, page 171)
- **Distributed Exception** (managed code, page 243)
- **Exception Module** (page 328)
- **Exception Stack Trace** (page 363)
- **FPU Exception** (page 398)
- **Handled Exception** (.NET CLR, page 428)
- **Handled Exception** (kernel space, page 433)
- **Handled Exception** (user space, page 434)
- **Hidden Exception** (kernel space, page 455)
- **Hidden Exception** (user space, page 457)
- **Invalid Exception Information** (page 568)
- **Managed Code Exception** (page 617)
- **Mixed Exception** (page 688)
- **Multiple Exceptions** (kernel mode, page 708)
- **Multiple Exceptions** (managed space, page 713)
- **Multiple Exceptions** (user mode, page 714)
- **Nested Exceptions** (managed code, page 723)
- **Nested Exceptions** (unmanaged code, page 726)
- **Problem Exception Handler** (page 810)
- **Software Exception** (page 875)
- **Stored Exception** (page 959)
- **Translated Exception** (page 1013)

Mac OS X:

- **C++ Exception** (page 109)
- **Multiple Exceptions** (page 706)

Linux:

- **C++ Exception** (page 108)

## Falsity and Coincidence Patterns

Windows:

- **Coincidental Error Code** (page 128)
- **Coincidental Frames** (page 130)
- **Coincidental Symbolic Information** (page 137)
- **False Effective Address** (page 389)
- **False Function Parameters** (page 390)
- **False Positive Dump** (page 393)

Mac OS X:

- **Coincidental Symbolic Information** (page 135)

Linux:

- **Coincidental Symbolic Information** (page 134)

## Hooksware Patterns

“Hooksware” word describes applications heavily dependent on various hooks that are either injected by normal Windows hooking mechanism, registry, or via more elaborate tricks like remote threads or code patching. There are various patterns in memory dump analysis that help in detection, troubleshooting, and debugging hooksware:

- **Changed Environment** (page 114)

Loaded hooks shift other modules by changing their load address and may expose dormant bugs.

- **Coincidental Symbolic Information** (page 137)

Sometimes hooks are loaded at round addresses like 0x10000000, and these values are very frequently used as flags or constants too.

- **Execution Residue** (page 371)

Here we can find various hooks that use normal Windows hooking mechanism. Sometimes the search for “hook” word in symbolic raw stack output of **dds** command reveals them but beware of coincidental symbolic information. See also how to dump raw stack from x64/x86 process dump files<sup>238</sup>, x86 processes saved as x64 process dump file<sup>239</sup>, process spaces from x86 complete memory dumps<sup>240</sup>, and x86/x64 process spaces from x64 complete memory dumps<sup>241</sup>.

- **Hidden Module** (page 463)

Some hooks may hide themselves.

- **Hooked Functions** (kernel space, Volume 1, page 468)
- **Hooked Functions** (user space, Volume 1, page 468)
- **Hooking Level** (page 489)

This is the primary detection mechanism for hooks that patch code (**Patched Code**, 802).

See also **Raw Pointer** (page 828) and **Out-of-Module Pointer** (page 773) patterns.

<sup>238</sup> Raw Stack Dump of All Threads (Process Dump), Memory Dump Analysis Anthology, Volume 1, page 231

<sup>239</sup> Raw Stack Dump of All Thread Stacks, Memory Dump Analysis Anthology, Volume 5, page 39

<sup>240</sup> Raw Stack Dump of All Threads (Complete Dump), Memory Dump Analysis Anthology, Volume 1, page 236

<sup>241</sup> Complete Stack Traces from x64 System, Memory Dump Analysis Anthology, Volume 5, page 20

- **Hooked Modules** (page 489)

The WinDbg script to run when we don't know which module was patched.

- **Insufficient Memory** (module fragmentation, page 544)

Hooks loaded in the middle of address space limit the maximum amount of memory that can be allocated at once. For example, various virtual machines reserve the big chunk of memory at startup.

- **Message Hooks** (page 663)

Windows message hooking pattern example. See also modeling example<sup>242</sup>.

- **No Component Symbols** (page 734)

We can get an approximate picture of what a 3rd-party hook module does by looking at its import table or in the case of patching by looking at the list of deviations returned by `.chkimg` command.

- **Unknown Component** (page 1035)

This pattern might give an idea about the author of the hook.

- **Wild Code** (page 1148)

When hooking goes wrong, the execution path goes into the wild territory.

---

<sup>242</sup> Models of Software Behaviour, Message Hooks Pattern, Memory Dump Analysis Anthology, Volume 5, page 326

## Memory Consumption Patterns

Windows:

- **Handle Leak** (page 416)
- **Handle Limit** (GDI, kernel space, page 417)
- **Handle Limit** (GDI, user space, page 423)
- **Insufficient Memory** (committed memory, page 523)
- **Insufficient Memory** (control blocks, page 525)
- **Insufficient Memory** (handle leak, page 526)
- **Insufficient Memory** (kernel pool, page 535)
- **Insufficient Memory** (module fragmentation, page 544)
- **Insufficient Memory** (physical memory, page 552)
- **Insufficient Memory** (PTE, page 555)
- **Insufficient Memory** (region, page 556)
- **Insufficient Memory** (reserved virtual memory, page 559)
- **Insufficient Memory** (session pool, page 562)
- **Insufficient Memory** (stack trace database, page 563)
- **Memory Fluctuation** (process heap, page 634)
- **Memory Leak** (.NET heap, page 636)
- **Memory Leak** (I/O completion packets, page 643)
- **Memory Leak** (page tables, page 644)
- **Memory Leak** (process heap, page 650)
- **Memory Leak** (regions, page 657)
- **Object Distribution Anomaly** (.NET heap, page 756)
- **Punctuated Memory Leak** (page 819)
- **Reference Leak** (page 830)
- **Relative Memory Leak** (page 834)

## Meta-Memory Dump Patterns

Windows:

- **Abridged Dump** (page 49)
- **Clone Dump** (page 118)
- **Corrupt Dump** (page 137)
- **Early Crash Dump** (page 312)
- **Evental Dumps** (page 328)
- **False Positive Dump** (page 393)
- **Fat Process Dump** (page 395)
- **Inconsistent Dump** (page 498)
- **Late Crash Dump** (page 601)
- **Lateral Damage** (page 602)
- **Manual Dump** (kernel, page 625)
- **Manual Dump** (process, page 630)
- **Mirror Dump Set** (page 666)
- **No Process Dumps** (page 740)
- **No System Dumps** (page 741)
- **Quiet Dump** (page 823)
- **Self-Dump** (page 850)
- **Step Dumps** (page 955)
- **Tampered Dump** (page 974)
- **Truncated Dump** (page 1015)
- **Unsynchronized Dumps** (page 1050)

Mac OS X:

- **Truncated Dump** (page 1014)

## Module Patterns

Windows:

- **Blocking Module** (page 96)
- **Coupled Modules** (page 147)
- **Deviant Module** (page 222)
- **Diachronic Module** (page 230)
- **Directing Module** (page 235)
- **Dry Weight** (page 281)
- **Duplicated Module** (page 287)
- **Effect Component** (page 315)
- **Exception Module** (page 328)
- **Foreign Module Frame** (page 398)
- **Hidden Module** (page 463)
- **Hooked Modules** (page 489)
- **Insufficient Memory** (module fragmentation, page 544)
- **Missing Component** (general, page 668)
- **Missing Component** (static linking, user mode, page 672)
- **Module Collection** (general, page 693)
- **Module Collection** (predicate, page 695)
- **Module Hint** (page 696)
- **Module Product Process** (page 697)
- **Module Variety** (page 703)
- **Nested Offender** (page 730)
- **No Component Symbols** (page 734)
- **Origin Module** (page 771)
- **Problem Module** (page 812)
- **Template Module** (page 997)
- **Top Module** (page 1012)
- **Ubiquitous Component** (kernel space, page 1020)
- **Ubiquitous Component** (user space, page 1023)
- **Unknown Component** (page 1035)
- **Unloaded Module** (page 1041)
- **Well-Tested Module** (page 1147)

## Optimization Patterns

Windows:

- **False Function Parameters** (page 390)
- **Hidden Parameter** (page 465)
- **Inline Function Optimization** (managed code, page 512)
- **Inline Function Optimization** (unmanaged code, page 514)
- **OMAP Code Optimization** (page 756)
- **Optimized Code** (page 767)
- **Optimized VM Layout** (page 769)

## Process Patterns

Windows:

- **Coupled Processes** (semantics, page 148)
- **Coupled Processes** (strong, page 149)
- **Coupled Processes** (weak, page 151)
- **Crashed Process** (page 156)
- **Fat Process Dump** (page 395)
- **Frozen Process** (page 407)
- **Hidden Process** (page 467)
- **Manual Dump** (process, page 630)
- **Missing Process** (page 682)
- **No Process Dumps** (page 740)
- **One-Thread Process** (page 765)
- **Process Factory** (page 814)
- **Special Process** (page 877)
- **Value Adding Process** (page 1052)
- **Virtualized Process** (WOW64, page 1068)
- **Wait Chain** (process objects, page 1108)
- **Zombie Processes** (page 1158)

## RPC, LPC and ALPC Patterns

The following patterns may help in the analysis of the output of **!lpc** and **!alpc** WinDbg commands from Windows kernel and complete memory dumps.

- **Blocked Queue** (LPC/ALPC, page 77)
- **Deadlock** (LPC, page 197)
- **Paged Out Data** (page 778)
- **Screwbolt Wait Chain** (page 843)
- **Semantic Structures** (PID.TID, page 860)
- **Wait Chain** (LPC/ALPC, page 1097)
- **Wait Chain** (process objects, page 1111)
- **Wait Chain** (RPC, page 1118)

## Stack Overflow Patterns

Windows:

- **Stack Overflow** (kernel mode, page 900)
- **Stack Overflow** (software implementation, page 910)
- **Stack Overflow** (user mode, page 912)

Mac OS X:

- **Stack Overflow** (page 897)

Linux:

- **Stack Overflow** (page 895)

## Stack Trace Patterns

Windows:

- **Coincidental Frames** (page 130)
- **Constant Subtrace** (page 141)
- **Critical Stack Trace** (page 168)
- **Dual Stack Trace** (page 282)
- **Empty Stack Trace** (page 321)
- **Exception Stack Trace** (page 363)
- **First Fault Stack Trace** (page 397)
- **Foreign Module Frame** (page 398)
- **Glued Stack Trace** (page 413)
- **Hidden Stack Trace** (page 469)
- **Incorrect Stack Trace** (page 499)
- **Internal Stack Trace** (page 568)
- **Least Common Frame** (page 606)
- **Managed Stack Trace** (page 624)
- **Module Stack Trace** (page 700)
- **Past Stack Trace** (page 800)
- **Quotient Stack Trace** (page 824)
- **RIP Stack Trace** (page 834)
- **Rough Stack Trace** (page 839)
- **Special Stack Trace** (page 882)
- **Stack Trace** (database, page 919)
- **Stack Trace** (file system filters, page 924)
- **Stack Trace** (general, page 926)
- **Stack Trace** (I/O request, page 930)
- **Stack Trace Change** (page 919)
- **Stack Trace Collection** (CPUs, page 933)
- **Stack Trace Collection** (I/O requests, page 933)
- **Stack Trace Collection** (managed space, page 940)
- **Stack Trace Collection** (predicate, page 943)
- **Stack Trace Collection** (unmanaged space, page 943)
- **Stack Trace Set** (page 952)
- **Stack Trace Signature** (page 955)
- **Stack Trace Surface** (page 957)
- **Technology-Specific Subtrace** (COM client call, page 987)
- **Technology-Specific Subtrace** (COM interface invocation, page 988)
- **Technology-Specific Subtrace** (dynamic memory, page 992)
- **Technology-Specific Subtrace** (JIT .NET code, page 994)

- **Truncated Stack Trace** (page 1015)
- **Unified Stack Trace** (page 1035)
- **Variable Subtrace** (page 1058)

Mac OS X:

- **Incomplete Stack Trace** (page 495)
- **Stack Trace** (page 918)

Linux:

- **Incomplete Stack Trace** (page 495)
- **Module Stack Trace** (page 699)
- **Stack Trace** (page 917)

## Symbol Patterns

Windows:

- **Coincidental Symbolic Information** (page 137)
- **Incorrect Symbolic Information** (page 505)
- **Injected Symbols** (page 510)
- **No Component Symbols** (page 734)
- **Reduced Symbolic Information** (page 829)
- **Unrecognizable Symbolic Information** (1045)

Mac OS X:

- **Coincidental Symbolic Information** (page 135)

Linux:

- **Coincidental Symbolic Information** (page 134)

## Thread Patterns

Windows:

- **Active Thread** (page 66)
- **Affine Thread** (page 72)
- **Blocked Thread** (hardware, page 80)
- **Blocked Thread** (software, page 82)
- **Blocked Thread** (timeout, page 92)
- **CLR Thread** (page 124)
- **Ghost Thread** (page 411)
- **Main Thread** (page 614)
- **Missing Thread** (page 683)
- **No Current Thread** (page 737)
- **Not My Thread** (page 742)
- **One-Thread Process** (page 765)
- **Passive System Thread** (kernel space, page 789)
- **Passive Thread** (user space, page 793)
- **Special Thread** (.NET CLR, page 883)
- **Spiking Thread** (page 888)
- **Suspended Thread** (page 964)
- **Thread Age** (page 1001)
- **Thread Cluster** (page 1003)
- **Thread Poset** (page )
- **Thread Starvation** (normal priority, page 1006)
- **Thread Starvation** (realtime priority, page 1008)
- **Wait Chain** (thread objects, page 1128)
- **Waiting Thread Time** (kernel dumps, page 1137)
- **Waiting Thread Time** (user dumps, page 1144)

Mac OS X:

- **Active Thread** (page 64)
- **Spiking Thread** (page 886)

Linux:

- **Active Thread** (page 63)
- **Spiking Thread** (page 885)

## Wait Chain Patterns

Windows:

- [Distributed Wait Chain \(page 253\)](#)
- [Screwbolt Wait Chain \(page 843\)](#)
- [Wait Chain \(C++11, condition variable, page 1082\)](#)
- [Wait Chain \(CLR monitors, page 1082\)](#)
- [Wait Chain \(critical sections, page 1086\)](#)
- [Wait Chain \(executive resources, page 1089\)](#)
- [Wait Chain \(general, page 1092\)](#)
- [Wait Chain \(LPC/ALPC, page 1097\)](#)
- [Wait Chain \(modules, page 1103\)](#)
- [Wait Chain \(mutex objects, page 1104\)](#)
- [Wait Chain \(named pipes, page 1106\)](#)
- [Wait Chain \(nonstandard synchronization, page 1108\)](#)
- [Wait Chain \(process objects, page 1108\)](#)
- [Wait Chain \(pushlocks, page 1116\)](#)
- [Wait Chain \(RPC, page 1118\)](#)
- [Wait Chain \(RTL\\_RESOURCE, page 1122\)](#)
- [Wait Chain \(thread objects, page 1128\)](#)
- [Wait Chain \(window messaging, page 1132\)](#)

## Appendix C

### Crash Dump Analysis Checklist

#### General:

- Symbol servers (**.symfix**)
- Internal database(s) search
- Google or Microsoft search for suspected components as this could be a known issue. Sometimes a simple search immediately points to the fix on a vendor's site
- The tool used to save a dump (to flag false positive, incomplete or inconsistent dumps)
- OS/SP version (**version**)
- Language
- Debug time
- System uptime
- Computer name (**ds srv/srvcomputername** or **!envvar COMPUTERNAME**)
- List of loaded and unloaded modules (**!lmv** or **!dlls**)
- Hardware configuration (**!sysinfo**)
- **.kframes 1000**

#### Application or service:

- Default analysis (**!analyze -v** or **!analyze -v -hang** for hangs)
- Critical sections (**!cs -s -l -o, !locks**) for both crashes and hangs
- Component timestamps, duplication, and paths. DLL Hell? (**!lmv** and **!dlls**)
- Do any newer components exist?
- Process threads (**~\*kv** or **!uniqstack**) for multiple exceptions and blocking functions
- Process uptime
- Your components on the full raw stack of the problem thread
- Your components on the full raw stack of the main application thread
- Process size
- Number of threads
- Gflags value (**!gflag**)
- Time consumed by threads (**!runaway**)
- Environment (**!peb**)
- Import table (**!dh**)
- Hooked functions (**!chkimg**)
- Exception handlers (**!exchain**)
- Computer name (**!envvar COMPUTERNAME**)
- Process heap stats and validation (**!heap -s, !heap -s -v**)
- CLR threads? (**mscorwks** or **clr** modules on stack traces) Yes: use .NET checklist below
- Hidden (unhandled and handled) exceptions on thread raw stacks

**System hang:**

- Default analysis (**!analyze -v -hang**)
- ERESOURCE contention (**!locks**)
- Processes and virtual memory including session space (**!vm 4**)
- Important services are present and not hanging (for example, terminal or IMA services for Citrix environments)
- Pools (**!poolused**)
- Waiting threads (**!stacks**)
- Critical system queues (**!exqueue f**)
- I/O (**!irpfind**)
- The list of all thread stack traces (**!process 0 3f**)
- LPC/ALPC chain for suspected threads (**!lpc message** or **!alpc /m** after search for "Waiting for reply to LPC" or "Waiting for reply to ALPC" in **!process 0 3f** output)
- RPC threads (search for "RPCRT4!OSF" in **!process 0 3f** output)
- Mutants (search for "Mutants - owning thread" in **!process 0 3f** output)
- Critical sections for suspected processes (**!cs -l -o -s**)
- Sessions, session processes (**!session**, **!sprocess**)
- Processes (size, handle table size) (**!process 0 0**)
- Running threads (**!running**)
- Ready threads (**!ready**)
- DPC queues (**!dpcs**)
- The list of APCs (**!apc**)
- Internal queued spinlocks (**!qlocks**)
- Computer name (**dS srv!srvcomputername**)
- File cache, VACB (**!filecache**)
- File objects for blocked thread IRPs (**!irp -> !fileobj**)
- Network (**!ndiskd.miniports** and **!ndiskd.pkt pools**)
- Disk (**!scsikd.classext -> !scsikd.classext class\_device 2**)
- Modules rdbss, mrxdav, mup, mrxsmb in stack traces
- Functions Ntfs!Ntfs\*, nt!Fs\* and fltmgr!Flt\* in stack traces

**BSOD:**

- Default analysis (**!analyze -v**)
- Pool address (**!pool**)
- Component timestamps (**!mv**)
- Processes and virtual memory (**!vm 4**)
- Current threads on other processors
- Raw stack
- Bugcheck description (including In exception address for corrupt or truncated dumps)
- Bugcheck callback data (**!bugdump** for systems prior to Windows XP SP1)
- Bugcheck secondary callback data (**.enumtag**)
- Computer name (**dS srv!srvcomputername**)
- Hardware configuration (**!sysinfo**)

**.NET application or service:**

- CLR module and SOS extension versions (**!lmv** and **.chain**)
- Managed exceptions (**~\*e !pe**)
- Nested managed exceptions (**!pe -nested**)
- Managed threads (**!Threads -special**)
- Managed stack traces (**~\*e !CLRStack**)
- Managed execution residue (**~\*e !DumpStackObjects** and **!DumpRuntimeTypes**)
- Managed heap (**!VerifyHeap**, **!DumpHeap -stat** and **!eeheap -gc**)
- GC handles (**!GCHandle**, **!GCHandleLeaks**)
- Finalizer queue (**!FinalizeQueue**)
- Sync blocks (**!syncblk**)

## Index

!

!address, 119, 128, 154, 161, 162, 166, 167, 222, 226, 281, 476, 545, 547, 551, 557, 558, 560, 564, 657, 658, 737, 819, 820, 821, 822, 826, 920, 1153  
!alpc, 78, 221, 483, 862, 1111, 1113, 1182, 1189  
!analyze, 53, 143, 220, 243, 260, 292, 293, 403, 457, 470, 499, 523, 575, 617, 623, 673, 720, 723, 736, 904, 926, 933, 1068, 1072, 1086, 1152, 1188, 1189  
!analyze -v, 53, 60, 153, 169, 205, 255, 256, 260, 265, 279, 284, 309, 310, 363, 394, 396, 457, 523, 570, 587, 617, 623, 687, 688, 708, 720, 806, 807, 868, 870, 904, 908, 910, 926, 974, 1064, 1068, 1072, 1076, 1086, 1188  
!analyze -v -hang, 1086  
!avrf, 517  
!bugdump, 1189  
!chkimg, 177, 178, 385, 484, 487, 488, 489, 490, 491, 492, 493, 802, 1188  
!CLRStack, 75, 284, 286, 452, 582, 584, 585, 620, 621, 624, 940, 995, 1066, 1190  
!cppexpr, 110  
!cs, 184, 191, 192, 193, 254, 412, 1019, 1110, 1188, 1189  
!dd, 644  
!ddstack, 383  
!devobj, 462  
!devstack, 462  
!dh, 220, 222, 226, 227, 387, 463, 549, 734, 775, 1038, 1188  
!dlk, 202, 204  
!dlls, 289, 290, 678, 1188  
!do, 583, 585  
!dpcs, 76, 1189  
!dso, 369, 582, 584  
!DumpHeap, 303, 637, 639, 640, 642, 756, 1190  
!dumpobj, 243, 303, 369, 807  
!DumpRuntimeTypes, 370, 1190  
!DumpStack, 111, 112, 113, 369, 382, 432, 617, 618, 624, 806, 1064  
!DumpStackObjects, 369, 1190  
!eeheap, 637, 639, 640, 1190  
!EEStack, 382, 618, 624  
!envvar, 1188  
!error, 95, 142, 402, 672, 714, 716, 717, 718, 876  
!errpkt, 448

!errrec, 448  
!exchain, 170, 171, 172, 691, 732, 810, 1188  
!exqueue, 790, 1189  
!filecache, 525, 1189  
!fileobj, 94, 1107, 1189  
!FinalizeQueue, 1190  
!findhandle, 1057  
!findstack, 944, 945  
!fltkd, 925  
!for\_each\_module, 385, 387, 487, 490, 491, 1057  
!for\_each\_process, 324, 812  
!for\_each\_thread, 97, 597, 694, 943, 951  
!GCHandleLeaks, 1190  
!GCHandles, 1190  
!gcroot, 642  
!gflag, 273, 275, 517, 518, 520, 583, 586, 652, 920, 1188  
!gle, 596, 597, 674  
!handle, 95, 211, 527, 532, 533, 577, 873, 874, 1057, 1082, 1125, 1129, 1130, 1164  
!heap, 120, 128, 267, 268, 269, 270, 273, 275, 277, 307, 308, 309, 312, 460, 476, 499, 523, 557, 558, 559, 560, 563, 565, 636, 637, 638, 639, 640, 650, 651, 652, 653, 654, 655, 656, 735, 774, 1039, 1057, 1188  
!help, 621, 623  
!htrace, 483, 530, 532, 577, 585  
!IP2MD, 75, 513, 593, 994, 995, 1019  
!irp, 93, 279, 462, 483, 662, 910, 925, 930, 931, 1095, 1106, 1189  
!irpfind, 462, 483, 759, 936, 1189  
!kdexts.handle, 1162  
!list, 99, 182, 194, 195, 483, 498, 527, 542, 603, 621, 653, 703, 735, 789, 944, 948, 949, 1138  
!lmi, 128, 129, 388, 698, 1046, 1047  
!locks, 53, 54, 56, 59, 162, 163, 164, 167, 182, 184, 189, 190, 191, 194, 195, 199, 205, 206, 210, 211, 297, 475, 477, 479, 498, 542, 553, 606, 687, 853, 966, 1087, 1089, 1172, 1188, 1189  
!ipc, 198, 200, 483, 778, 1098, 1100, 1142, 1182, 1189  
!ndiskd.miniport, 236  
!ndiskd.miniports, 236, 1189  
!ndiskd.pkt pools, 733, 1189  
!ntsdexts.locks, 184, 199  
!object, 84, 280, 534, 831, 971, 972  
!obtrace, 483, 832, 1057, 1164

!ipc, 900, 903  
!pe, 579, 584, 624, 688, 713, 807, 1067, 1190  
!peb, 289, 324, 387, 677, 1188  
!pool, 175, 176, 260, 262, 263, 295, 299, 308, 523, 524, 526, 527, 535, 536, 537, 538, 539, 542, 696, 793, 826, 869, 871, 993, 1189  
!poolfind, 97, 264, 467, 643  
!poolused, 147, 416, 417, 522, 527, 535, 538, 539, 541, 543, 562, 643, 831, 832, 1189  
!poolval, 262, 295, 299, 697, 993  
!PrintException, 620, 713, 723, 724, 807, 1067  
!process, 83, 97, 98, 156, 325, 326, 327, 408, 416, 467, 478, 505, 507, 527, 532, 533, 554, 643, 644, 660, 682, 765, 789, 812, 815, 817, 830, 853, 877, 878, 879, 880, 944, 948, 949, 964, 973, 1046, 1097, 1115, 1135, 1157, 1161, 1164, 1189  
!pte, 174, 649, 710, 905  
!ptov, 645, 646, 647  
!qlocks, 242, 1189  
!ready, 83, 91, 99, 102, 103, 480, 970, 1006, 1137, 1189  
!reg, 845  
!runaway, 49, 65, 68, 171, 245, 249, 250, 472, 521, 566, 884, 888, 890, 893, 1001, 1054, 1144, 1145, 1188  
!running, 70, 74, 82, 90, 99, 102, 103, 114, 480, 606, 640, 709, 720, 892, 894, 933, 944, 970, 1006, 1008, 1078, 1137, 1139, 1189  
!scsikd.classext, 238, 239, 1189  
!search, 1057  
!session, 171, 313, 314, 496, 536, 562, 626, 630, 631, 966, 1068, 1156, 1189  
!sprocess, 407, 496, 497, 543, 951, 1052, 1189  
!stacks, 96, 184, 483, 498, 618, 621, 720, 789, 825, 882, 892, 944, 945, 951, 1004, 1020, 1095, 1141, 1189  
!syncblk, 202, 474, 1190  
!sysinfo, 743, 1188, 1189  
!sysptes, 556  
!teb, 321, 373, 383, 404, 434, 457, 500, 597, 598, 612, 663, 668, 674, 681, 689, 728, 731, 839, 913, 915, 1013, 1055  
!thread, 49, 56, 58, 59, 70, 71, 72, 74, 76, 81, 82, 83, 84, 85, 90, 99, 100, 103, 104, 105, 106, 131, 176, 193, 198, 199, 200, 207, 208, 241, 282, 315, 412, 437, 455, 456, 462, 477, 480, 481, 529, 552, 661, 709, 778, 779, 825, 826, 854, 855, 856, 857, 858, 860, 890, 894, 930, 968, 969, 970, 1006, 1008, 1009, 1010, 1011, 1078, 1091, 1098, 1099, 1101, 1126, 1128, 1129, 1138  
!Threads, 713, 883, 1190  
!token, 229  
!U, 75, 513  
!uniqstack, 793, 944, 952, 953, 1188  
!validatealist, 97  
!verifier, 263, 516  
!VerifyHeap, 302, 303, 1190  
!vm, 97, 219, 524, 526, 535, 536, 537, 538, 539, 541, 553, 555, 562, 682, 701, 702, 812, 814, 972, 1015, 1156, 1158, 1189  
!w2kfre\kdex2x86.xpool, 540  
!whattime, 616, 1141  
!whea, 448  
!wow64exts.sw, 1075  
.  
.asm, 100, 249, 296, 423, 513, 592, 691, 730, 902, 1077  
.bugcheck, 263, 552  
.catch, 312, 589, 683, 686, 900  
.chain, 283, 284, 285, 618, 1065, 1190  
.cordll, 1064, 1066  
.cxr, 60, 86, 95, 153, 180, 256, 326, 385, 389, 401, 417, 442, 455, 456, 459, 460, 461, 465, 470, 570, 573, 605, 690, 729, 731, 732, 780, 868, 870, 905, 906, 907, 928, 975, 978, 986, 1043, 1048, 1049, 1108, 1109, 1124  
.dump, 49, 290, 1144  
.ecxr, 313, 314, 471, 572, 603, 631, 673, 728, 729, 959, 975, 1068, 1069  
.effmach, 326, 694, 943, 1072  
.enumtag, 1189  
.exprtr, 364, 592  
.exr, 60, 133, 169, 259, 401, 442, 470, 570, 575, 576, 603, 617, 632, 668, 673, 676, 688, 714, 732, 738, 806, 849, 868, 870, 906, 912, 959, 974, 1064, 1069, 1072  
.for, 54, 56, 99, 102, 105, 116, 138, 150, 172, 182, 184, 194, 195, 196, 197, 198, 199, 200, 201, 210, 211, 217, 261, 272, 307, 308, 312, 313, 314, 393, 459, 460, 477, 483, 488, 489, 498, 502, 523, 535, 537, 539, 540, 542, 589, 614, 615, 618, 620, 622, 626, 650, 652, 653, 683, 686, 703, 705, 763, 764, 767, 768, 789, 790, 791, 792, 793, 795, 888, 900, 904, 905, 907, 926, 927, 928, 929, 944, 951, 1015, 1037, 1086, 1088, 1092, 1093, 1094, 1137, 1138, 1139, 1140, 1141  
.foreach, 212  
.formats, 181, 508, 509, 715, 871, 1151  
.frame, 390, 581, 1124  
.if, 597, 943  
.imgscan, 220, 222, 228, 463, 464  
.kframes, 1017, 1188

.lastevent, 674  
 .lines, 716  
 .load, 238, 284, 326, 579, 623, 693, 694, 1065, 1072  
 .loadby, 618, 622, 623, 637, 641  
 .logopen, 653  
 .NET runtime, 591  
 .printf, 212  
 .process, 86, 94, 199, 254, 326, 411, 505, 507, 527, 530, 543, 649, 661, 812, 817, 877, 879, 948, 949, 1045  
 .reload, 95, 506, 511, 527, 559, 693, 694, 943, 944, 945, 948, 949, 1049  
 .symfix, 559, 1188  
 .sympath+, 510, 1049  
 .thread, 54, 82, 85, 86, 94, 95, 180, 254, 326, 385, 412, 465, 482, 573, 597, 605, 690, 693, 694, 709, 732, 818, 905, 907, 928, 943, 975, 986, 1008, 1009, 1043, 1047, 1048, 1049, 1123, 1124  
 .time, 1002, 1050, 1164  
 .trap, 86, 169, 174, 175, 241, 255, 307, 389, 523, 629, 708, 710, 900, 903, 908, 909, 928, 992  
 .tss, 255, 900, 901, 903, 904, 908, 909  
 .ttime, 1001  
 .unload, 285, 1065  
 .while, 426, 427

**?**

?, 180, 426, 427  
 ??, 1138, 1139

**~**

~, 425, 472, 601, 737, 873, 918, 1041, 1188, 1190  
 ~\*e, 713, 940  
 ~\*k, 118, 469, 472, 601, 742, 1041  
 ~\*kbL, 1132, 1133  
 ~\*kc, 329, 1023  
 ~\*kv, 183, 189, 191, 460, 612, 888, 944, 1188  
 ~~, 475  
 ~e, 99, 182, 211, 312, 589, 616, 685, 720, 736, 888, 1037, 1039

**A**

Abridged Dump, 49, 699, 1012, 1178  
 Accidental Lock, 53, 1172  
 Activation Context, 60, 875  
 Active Thread, 63, 66, 68, 230, 328, 1186

Activity Resonance, 70  
 add-symbol-file (GDB), 918  
 Adjoint Space, 328  
 Affine Thread, 72, 1186  
 Alien Component, 603, 744  
 Annotated Disassembly, 75, 1167

**B**

Bifurcation Point, 141  
 Blocked DPC, 76  
 Blocked Queue, 77, 221, 765, 1182  
 Blocked Thread, 80, 82, 89, 92, 93, 96, 209, 614, 660, 853, 855, 856, 881, 924, 944, 997, 1082, 1097, 1104, 1106, 1114, 1132, 1156, 1186  
 Blocking File, 93, 931  
 Blocking Module, 93, 96, 168, 230, 607, 813, 843, 925, 1012, 1020, 1103, 1179  
 Blocking Thread, 1050  
 Broken Link, 97  
 bt (GDB), 63, 64, 65, 108, 109, 258, 278, 304, 305, 365, 367, 608, 609, 745, 746, 747, 749, 885, 886, 895, 897, 899, 917, 918  
 Busy System, 66, 99, 102, 778, 1007

**C**

C++ Exception, 108, 109, 180, 731, 875, 1173  
 Caller-n-Callee, 111, 369, 432, 453, 839, 1167  
 Changed Environment, 114, 740, 769, 1175  
 Clone Dump, 118, 120, 633  
 Cloud Environment, 122  
 CLR Thread, 124, 204, 450, 579, 994, 1085, 1167, 1186  
 Code Path Locality, 175  
 Coincidental Error Code, 128, 1174  
 Coincidental Frames, 130, 1174  
 Coincidental Symbolic Information, 112, 130, 134, 135, 137, 221, 361, 366, 373, 440, 504, 569, 697, 839, 1054, 1083, 1174, 1175, 1185  
 Collapsed Stack Trace, 700, 824  
 Compact Stack Trace, 824  
 Component Variety, 703  
 Constant Subtrace, 141, 1036, 1082, 1083, 1108, 1110  
 Corrupt Dump, 142, 603, 741, 1178  
 Corrupt Structure, 144, 161, 844  
 Coupled Activities, 328  
 Coupled Machines, 146  
 Coupled Modules, 147, 1003, 1179

Coupled Processes, 146, 148, 149, 151, 245, 250, 328, 660, 682, 778, 803, 944, 1001, 1050, 1051, 1156, 1181  
 Crash Signature, 153, 155, 604, 955  
 Crash Signature Invariant, 155  
 Crashed Process, 156, 1181  
 Critical Region, 157  
 Critical Section Corruption, 161, 191  
 Critical Stack Trace, 168  
 Custom Exception Handler, 169, 171, 172, 690, 810, 850, 1013, 1173

## D

da, 189, 847, 962, 963  
 dA, 661  
 Data Alignment, 174, 708  
 Data Contents Locality, 175  
 Data Correlation, 180, 875  
 Data Flow, 782  
 db, 490, 548, 669, 869, 871, 1037, 1039  
 dc, 97, 232, 233, 234, 464, 697, 781, 829  
 dd, 52, 62, 75, 128, 176, 469, 644, 826, 904, 1016, 1044, 1129, 1138, 1148  
 dds, 131, 212, 263, 373, 377, 404, 437, 458, 485, 500, 550, 613, 669, 675, 685, 710, 731, 905, 909, 913, 915, 951, 1015, 1055, 1078, 1175  
 Deadlock, 148, 182, 184, 196, 197, 201, 202, 205, 210, 218, 1051, 1085, 1089, 1093, 1104, 1156, 1167, 1169, 1172, 1182  
 Debugger Bug, 219, 220  
 Debugger Omission, 220, 228, 464  
 Design Value, 221  
 Deviant Module, 222, 1179  
 Deviant Token, 229  
 Diachronic Module, 230  
 Dialog Box, 232, 423, 997  
 Directing Module, 147, 235, 1003, 1179  
 disass (GDB), 135, 136, 747, 886, 1014  
 disassemble (GDB), 134, 159, 745, 749, 885  
 Disconnected Network Adapter, 236  
 Discontinuity, 230  
 Disk Packet Buildup, 238  
 Dispatch Level Spin, 72, 241  
 Distributed Exception, 244  
 Distributed Spike, 245, 472, 563  
 Distributed Wait Chain, 253, 1020  
 Divide by Zero, 255, 257, 258  
 dl, 677

DLL Variety, 703  
 Double Free, 161, 260, 264, 267, 276, 278, 279, 517, 586, 1171  
 Double IRP Completion, 279  
 dp, 129, 203, 204, 424, 701, 714, 750, 781, 825, 873, 1044, 1077, 1125  
 dpa, 670, 962  
 dpp, 861  
 dps, 50, 111, 227, 263, 277, 296, 299, 309, 310, 315, 321, 401, 435, 455, 456, 474, 566, 576, 663, 670, 689, 716, 717, 719, 735, 781, 920, 978, 1013, 1018  
 dpS, 121, 382, 383, 839  
 dpu, 466, 670, 962, 963  
 dq, 424, 426, 1148  
 dqs, 728, 729  
 Driver Device Collection, 280  
 Dry Weight, 281, 1179  
 dS, 702, 1188, 1189  
 dt, 61, 62, 73, 74, 121, 144, 145, 161, 162, 166, 176, 194, 207, 238, 407, 408, 446, 482, 511, 530, 715, 739, 826, 827, 829, 845, 901, 903, 964, 965, 971, 1018, 1069, 1123, 1130, 1137, 1163  
 du, 167, 464, 549, 677, 818, 962  
 Dual Stack Trace, 282, 1051  
 Duplicate Extension, 283, 1167  
 Duplicate Module, 283  
 Duplicated Module, 287, 744  
 dv, 390, 391  
 dw, 426  
 dyd, 242  
 Dynamic Memory Corruption, 161, 267, 292, 301, 304, 305, 586, 632, 696, 868, 1167, 1171

## E

Early Crash Dump, 171, 312, 574, 601, 729, 740, 1178  
 Effect Component, 315, 1179  
 Embedded Comments, 320  
 Empty Stack Trace, 321  
 Environment Hint, 324, 1153  
 Error Reporting Fault, 325, 397  
 Evental Dumps, 66, 328, 742  
 Exception Handling Residue, 371, 376  
 Exception Module, 361, 875, 1173, 1179  
 Exception Stack Trace, 133, 326, 361, 363, 470, 579, 974  
 Exception Thread, 363  
 Execution Residue, 49, 111, 120, 121, 134, 135, 175, 244, 315, 365, 367, 369, 371, 397, 462, 465, 569, 582, 601,

659, 696, 782, 804, 839, 852, 860, 920, 974, 1013, 1054, 1078, 1082, 1153, 1167, 1175

## F

Fake Module, 385  
 False Effective Address, 389, 1174  
 False Function Parameters, 221, 389, 390, 782, 1174, 1180  
 False Positive Dump, 172, 320, 393, 1174, 1178  
 Fat Process Dump, 395, 1178, 1181  
 Fault Context, 396  
 First Stack Trace, 397  
 Foreign Module Frame, 398  
 FPU Exception, 401, 1173  
 frame (GDB), 278, 895, 898  
 Frame Pointer Omission, 180, 403

## G

g, 254, 278, 560, 574, 686  
 Ghost Thread, 411, 1186  
 Glued Activity, 412  
 Glued Stack Trace, 413, 1018

## H

Handle Leak, 416, 526, 535, 818, 830, 843, 919, 1051, 1104, 1163, 1177  
 Handle Limit, 417, 423  
 Handled Exception, 370, 428, 433, 434, 435, 1167, 1173  
 Hardware Activity, 437  
 Hardware Error, 441  
 Hidden Call, 450, 470  
 Hidden Exception, 121, 363, 371, 397, 433, 455, 461, 469, 470, 601, 633, 676, 689, 731, 738, 875, 978, 1013, 1064, 1173  
 Hidden IRP, 462  
 Hidden Module, 463, 803, 1175  
 Hidden Parameter, 465, 782, 1180  
 Hidden Process, 467, 1181  
 Hidden Stack Trace, 469, 471  
 High Contention, 53, 472, 475, 477, 479, 480, 542, 606, 987, 1006, 1168, 1172  
 Historical Information, 371, 483, 1051  
 Hooked Functions, 177, 442, 484, 488, 490, 492, 663, 803, 1175  
 Hooked Modules, 490, 1176, 1179  
 Hooking Level, 492, 1175

Hooksware, 492, 742, 802, 812, 1012, 1153, 1175

## I

Implementation Discourse, 990  
 Incomplete Session, 496, 497, 600  
 Inconsistent Dump, 498, 666, 1104, 1178  
 Incorrect Stack Trace, 154, 499, 503, 700, 850, 1012, 1017, 1149  
 Incorrect Symbolic Information, 505, 1185  
 info r (GDB), 257, 258, 610, 745, 746, 747, 1014  
 info threads (GDB), 63, 64, 602, 885, 886, 1014  
 Injected Symbols, 510, 1185  
 Inline Function Optimization, 133, 389, 512, 1167, 1180  
 Instrumentation Information, 279, 516, 586  
 Instrumentation Side Effect, 520, 563, 832  
 Insufficient Memory, 147, 416, 417, 523, 557, 562, 563, 643, 650, 657, 778, 832, 919, 1053, 1104, 1176, 1177, 1179  
 Inter-Correlation, 141  
 Internal Stack Trace, 568  
 Intra-Correlation, 141  
 Invalid Exception Information, 154, 570, 1173  
 Invalid Handle, 574, 578, 588, 1013  
 Invalid Parameter, 586, 588  
 Invalid Pointer, 417, 589, 750, 752, 780, 1152  
 IRP Distribution Anomaly, 760

## J

JIT Code, 591, 594, 838, 1019, 1167

## K

k, 52, 242, 261, 264, 323, 326, 385, 414, 448, 474, 524, 559, 580, 583, 594, 626, 737, 762, 800, 837, 904, 919, 928, 944, 974, 975, 978, 986, 1081  
 kbnL, 390  
 kc, 155, 361, 423, 520, 565, 824, 848, 849, 875, 952, 956, 1001, 1108, 1109  
 kn, 581, 1124  
 knf, 905, 907  
 kv, 95, 146, 153, 177, 180, 183, 232, 233, 255, 390, 403, 457, 469, 482, 499, 511, 572, 573, 588, 592, 750, 782, 810, 818, 829, 847, 873, 888, 900, 944, 952, 962, 1015, 1042, 1043, 1077, 1082, 1133, 1145  
 kvl, 92, 780, 782

**L**

Last Error Collection, 596  
 Last Object, 156, 599  
 Late Crash Dump, 601, 1178  
 Lateral Damage, 49, 97, 154, 602, 603, 975, 1178  
 Least Common Frame, 604  
 Livelock, 606, 1169, 1172  
 lm, 114, 122, 138, 139, 179, 287, 288, 290, 463, 510, 511, 544, 546, 769, 773, 808, 812  
 lmft, 744  
 lmn, 118  
 Imp, 385  
 lmt, 281, 507, 703, 998  
 lmtD, 705  
 imu, 506, 693, 812  
 lmv, 122, 123, 127, 128, 222, 228, 261, 283, 289, 291, 386, 400, 542, 548, 579, 693, 698, 703, 705, 734, 803, 842, 907, 929, 969, 998, 999, 1012, 1016, 1037, 1038, 1042, 1046, 1047, 1048, 1064, 1188, 1189, 1190  
 ln, 128, 129, 735, 736, 901, 1044, 1189  
 Local Buffer Overflow, 162, 403, 608, 609, 611, 868, 975, 1148, 1151  
 Lost Opportunity, 612

**M**

Main Thread, 89, 92, 410, 423, 614, 1128, 1134, 1186  
 maintenance info sections (GDB), 896, 897, 918  
 Managed Code Exception, 110, 124, 579, 617, 668, 688, 714, 806, 1065, 1167, 1173  
 Managed Stack Trace, 124, 452, 579, 618, 624, 995, 1035  
 Manual Dump, 205, 241, 320, 625, 630, 708, 712, 1016, 1064, 1156, 1178, 1181  
 Memory Fluctuation, 563, 634, 756, 1177  
 Memory Leak, 235, 563, 634, 636, 650, 657, 819, 834, 919, 1017, 1053, 1167, 1177  
 Message Box, 89, 232, 660, 847, 1134  
 Message Hooks, 400, 569, 663, 803, 1176  
 Mirror Dump Set, 498, 666  
 Missing Component, 668, 672, 682, 875, 1170, 1179  
 Missing Process, 682, 1181  
 Missing Thread, 411, 682, 683, 1051, 1104, 1114, 1186  
 Mixed Exception, 688, 714, 1167, 1173  
 Module Collection, 693, 695, 698, 1179  
 Module Hint, 559, 601, 659, 696, 974, 1153, 1179  
 Module Product Process, 569, 698, 843, 1051, 1179  
 Module Stack Trace, 699, 700, 1035

**Module Variable**, 701

Module Variety, 281, 287, 703, 744, 803, 1179  
 Multiple Exceptions, 174, 259, 307, 457, 471, 633, 706, 708, 713, 714, 720, 872, 875, 1167, 1173

**N**

Namespace, 722  
 Nested Exceptions, 243, 397, 688, 708, 723, 726, 732, 807, 1013, 1167, 1173  
 Nested Offender, 397, 730, 732, 1179  
 Network Packet Buildup, 238, 733  
 No Component Symbols, 549, 700, 722, 734, 736, 739, 1176, 1179, 1185  
 No Current Thread, 737, 1186  
 No Data Types, 739  
 No Process Dumps, 740, 741  
 No System Dumps, 741, 1178  
 Not My Thread, 742  
 Not My Version, 410, 743, 744, 1012  
 NULL Code Pointer, 371, 750, 752  
 NULL Data Pointer, 389, 750, 978  
 NULL Pointer, 169, 389, 590, 706, 745, 746, 747, 749, 750

**O**

Object Distribution Anomaly, 238, 635, 642, 756, 836  
 OMAP Code Optimization, 735, 761, 1180  
 One-Thread Process, 765, 1051, 1181, 1186  
 Optimized Code, 391, 588, 767, 872, 1180  
 Optimized VM Layout, 769, 1180  
 Origin Module, 771, 832, 1179  
 Out-of-Module Pointer, 773, 1175  
 Overaged System, 774, 778, 1104, 1156

**P**

p (GDB), 304, 706, 707, 747, 749  
 Packed Code, 775, 809  
 Paged Out Data, 327, 397, 778, 789, 1182  
 Parameter Flow, 780, 782  
 Paratext, 63, 64, 783, 785  
 Pass Through Function, 787, 1103, 1146  
 Passive System Thread, 787, 789, 1186  
 Passive Thread, 89, 151, 614, 787, 793, 1186  
 Past Stack Trace, 470, 568, 782, 800, 931, 1083  
 Patched Code, 802, 1175  
 Pervasive System, 803

Place Trace, 804, 805, 1082  
 Platformorphic Fault, 389  
 Platform-Specific Debugger, 806, 807, 1064  
 Pleiades, 808  
 poi, 781  
 Pre-Obfuscation Residue, 809  
 print (GDB), 747  
 Private Modification, 739  
 Problem Exception Handler, 810, 1173  
 Problem Module, 771, 812, 1003, 1179  
 Problem Vocabulary, 813  
 Procedure Call Chain, 149, 944  
 Process Factory, 814, 818, 1051, 1104, 1181  
 ptype (GDB), 747  
 Punctuated Memory Leak, 819

## Q

Quiet Dump, 823, 1178  
 Quotient Stack Trace, 824, 1035, 1103  
 Quotient Trace, 824

## R

r, 149, 155, 183, 187, 199, 220, 254, 269, 274, 326, 401, 426, 427, 499, 511, 513, 604, 611, 618, 632, 649, 694, 704, 709, 752, 793, 794, 798, 812, 817, 829, 865, 866, 867, 888, 898, 948, 949, 1015, 1037, 1039, 1044, 1045, 1047, 1048, 1073, 1074  
 Random Object, 825, 826, 871  
 Rare Stack Trace, 568  
 Raw Pointer, 828, 1175  
 Reduced Symbolic Information, 510, 699, 700, 829, 1185  
 Reference Leak, 771, 830, 1164, 1177  
 Regular Data, 144, 300, 833, 873  
 Relative Memory Leak, 834  
 RIP Stack Trace, 837  
 rMF, 402  
 Rough Stack Trace, 568, 800, 839

## S

s, 220, 254, 278, 305, 327, 368, 401, 465, 559, 560, 562, 748, 777, 809, 845, 864, 865, 866, 867, 872, 873, 884, 886, 960, 1020, 1041, 1057, 1188, 1189  
 s-a, 1153  
 Same Vendor, 521, 842, 1105  
 Screwbolt Wait Chain, 843

Self-Diagnosis, 279, 844, 845, 847, 875, 992  
 Self-Dump, 847, 850, 875, 1178  
 Semantic Split, 853, 1104, 1172  
 Semantic Structure, 860, 873, 987, 1182  
 set backtrace (GDB), 495  
 Shared Buffer Overwrite, 833, 864  
 Shared Structure, 872  
 Singleton Event, 156  
 Small Value, 221, 580, 873  
 Snapshot Collection, 1058  
 Software Exception, 60, 361, 574, 672, 714, 850, 875, 905, 910, 1013, 1068, 1173  
 Special Process, 156, 410, 682, 877, 965, 1153, 1181  
 Special Stack Trace, 568, 601, 672, 742, 850, 877, 882  
 Special Thread, 742, 883, 1167, 1186  
 Spike Interval, 230, 884  
 Spiking Thread, 63, 64, 66, 68, 70, 141, 152, 171, 230, 241, 245, 477, 521, 563, 565, 568, 606, 708, 800, 823, 885, 886, 935, 1007, 1008, 1051, 1053, 1058, 1091, 1110, 1186  
 s-sa, 1153  
 Stack Overflow, 397, 708, 811, 875, 895, 897, 899, 900, 910, 912, 1183  
 Stack Trace, 93, 131, 154, 168, 196, 235, 253, 269, 272, 274, 282, 321, 327, 363, 371, 389, 404, 450, 495, 499, 514, 564, 606, 618, 625, 655, 665, 716, 730, 771, 798, 802, 824, 832, 847, 850, 917, 918, 919, 920, 924, 926, 944, 988, 1003, 1007, 1053, 1080, 1103, 1104, 1147, 1166, 1167, 1173, 1175, 1184, 1185  
 Stack Trace Change, 932  
 Stack Trace Collection, 93, 141, 168, 253, 282, 327, 328, 363, 410, 469, 471, 563, 568, 599, 606, 693, 695, 787, 793, 812, 813, 823, 825, 861, 872, 931, 933, 940, 944, 952, 957, 1001, 1004, 1005, 1017, 1023, 1035, 1046, 1053, 1083, 1085, 1104, 1109, 1111, 1167, 1184  
 Stack Trace Database, 235, 483, 563, 564, 920  
 Stack Trace Set, 952, 1036  
 Stack Trace Signature, 824, 955  
 Stack Trace Surface, 957  
 Step Dumps, 958, 1178  
 Stored Exception, 470, 714, 959, 975, 1173  
 String Hint, 300, 722, 960  
 String Parameter, 232, 465, 696, 829, 962  
 Suspended Thread, 410, 682, 879, 964  
 Swarm of Shared Locks, 966, 1053, 1172  
 symbol-file (GDB), 917  
 System Objects, 382, 971

**T**

t, 1148, 1149  
 Tampered Dump, 974, 1178  
 Technology-Specific Subtrace, 141, 987, 1167  
 Template Module, 842, 997, 1179  
 thread (GDB), 65, 602, 706, 707, 885, 886  
 Thread Age, 843, 893, 1001, 1050, 1186  
 thread apply all (GDB), 158, 602  
 Thread Cluster, 695, 1003, 1186  
 Thread of Activity, 230  
 Thread Poset, 1004, 1005  
 Thread Starvation, 72, 1006, 1008, 1186  
 Time Delta, 230

Top Module, 168, 230, 235, 453, 521, 771, 924, 1003, 1012, 1020, 1103, 1179  
 Translated Exception, 875, 1013, 1173  
 Truncated Dump, 97, 142, 154, 205, 741, 1014, 1015, 1178  
 Truncated Stack Trace, 413, 700, 771, 832, 852, 919, 1017

**U**

u, 85, 128, 133, 140, 296, 327, 393, 474, 486, 487, 488, 489, 493, 494, 592, 593, 629, 701, 761, 762, 763, 764, 802, 1038, 1040, 1148, 1152  
 ub, 68, 81, 85, 112, 130, 131, 139, 140, 203, 204, 249, 261, 297, 390, 404, 405, 433, 435, 440, 454, 465, 504, 576, 582, 588, 593, 594, 595, 664, 692, 718, 719, 729, 731, 732, 750, 752, 763, 764, 780, 781, 869, 872, 911, 919, 1012, 1018, 1019, 1044, 1054, 1056, 1150  
 Ubiquitous Component, 231, 803, 1020, 1036, 1179  
 uf, 100, 216, 242, 391, 405, 423, 691, 763, 767, 768, 902, 1077, 1110, 1123  
 Unified Stack Trace, 1035  
 Unknown Component, 261, 463, 548, 744, 1037, 1176, 1179  
 Unknown Module, 722  
 Unloaded Module, 811, 1041, 1170, 1179  
 Unrecognizable Symbolic Information, 700, 1045, 1185  
 Unsynchronized Dumps, 1050, 1178  
 User Space Evidence, 1051

**V**

Value Adding Process, 1052, 1181  
 Value Deviation, 1053  
 Value References, 535, 804, 1057

Variable Subtrace, 141, 1058, 1110

Vendor Collection, 693  
 version, 95, 122, 123, 283, 285, 739, 752, 1050, 1064, 1065, 1066, 1129, 1130  
 Version Specific Extension, 940  
 Version-Specific Extension, 283, 286, 994, 1064, 1167  
 vertarget, 171, 556, 1145  
 Virtualized Process, 94, 191, 325, 425, 597, 1068, 1072, 1104, 1181  
 Virtualized System, 94, 191, 325, 462, 626, 743, 1076, 1181  
 Vocabulary Index, 813

**W**

W2K, 553  
 Wait Chain, 72, 93, 96, 141, 209, 221, 230, 253, 328, 411, 568, 599, 667, 682, 695, 778, 813, 824, 843, 853, 859, 873, 881, 957, 987, 1001, 1020, 1051, 1082, 1085, 1086, 1088, 1089, 1091, 1092, 1093, 1096, 1097, 1102, 1103, 1104, 1106, 1109, 1110, 1111, 1118, 1122, 1128, 1131, 1132, 1156, 1167, 1172, 1181, 1182, 1186, 1187  
 Waiting Thread Time, 483, 813, 843, 1003, 1104, 1137, 1144, 1186  
 Well-Tested Function, 1146, 1147  
 Well-Tested Module, 361, 771, 1103, 1146, 1147, 1179  
 where (GDB), 495  
 Wild Code, 176, 443, 591, 1148, 1151, 1176  
 Wild Pointer, 162, 588, 611, 750, 868, 1148, 1151  
 Window Hint, 1153

**X**

x, 514, 515, 701, 734  
 x (GDB), 135, 136, 258, 278, 304, 367, 610, 746, 747, 865, 866, 867, 895, 897, 1014  
 x/a (GDB), 365, 608  
 x/i, 134  
 x/i (GDB), 63, 257, 304, 746  
 x/w (GDB), 257

**Y**

Young System, 462, 1156

**Z**

Zombie Processes, 156, 644, 818, 830, 1051, 1158, 1181

