Native Debugging Desktop Framework

Overview

While Visual Studio 2008 and lower are useful for debugging development issues, when there are problems on a machine where Visual Studio is not installed, the best way to investigate is often to use one of the native debugging tools. Usually this amounts to getting one or several dumps at or near the time of the error, and then analyzing that dump to identify the problem.

Debugging Tools

Native debugging tools and their downloaded sites are given below. These need to be installed and run, generally with admin rights on the machine where the issue is occurring to capture the dump. Once a dump has been created, it can generally be analyzed on a different machine.

DebugDialog

http://www.microsoft.com/download/en/details.aspx?id=26798 (for 32 bit use the x86 version)

This is a dialog based tool providing much of the functionality of other command line tools. It allows the definition of crash rules so that a dump will automatically be created under specific crash scenarios. You can also attach to an existing process and create a dump while it is running.

Debugging tools for windows:

http://msdn.microsoft.com/en-us/windows/hardware/gg463009.aspx

There are several tools with this download which can be used both to create dumps and also to analyze an existing dump. Among the most useful are windbg and adplus.

Application Verifier

http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=20028

In certain situations, such as when investigating heap corruption, application verifier may be used together with the other tools.

System Internals Tools

http://technet.microsoft.com/en-us/sysinternals

System Internal Tools contains other tool which can be useful in analyzing a dump.

Process Monitor:

http://technet.microsoft.com/en-us/sysinternals/bb896645

Process Monitor shows useful information about the processes.

Visual Studio 2010

There are several useful tools in VS 2010 that can be used to analyze dumps without the more cryptic techniques required in the other mentioned debugging tools.

This post by Tess Ferrandez is a good starting point for learning about these tools:

http://blogs.msdn.com/b/tess/archive/2010/04/28/new-visual-studio-2010-and-net-4-debugging-content.aspx

CLR Profiler

http://www.microsoft.com/download/en/confirmation.aspx?id=14727

The CLR Profiler can be useful while debugging.

NP Profiler

\\wmphew1bbs001\swat$\BRAD\_MS\Tools\NP\4.1

Useful Links

The best place to start learning about using the Native Debugger for creating and analyzing dumps is the blog by Tess Ferrandez:

http://blogs.msdn.com/b/tess

How to get a dump

This is a good summary by Tess Ferrandez:

http://blogs.msdn.com/b/tess/archive/2006/01/11/back-to-basics-how-do-i-get-the-memory-dumps-in-the-first-place-and-what-is-sos-dll.aspx

This blog on system internals is also good:

http://technet.microsoft.com/en-us/sysinternals/bb963890

Where to put a dump

Usually you will be passing the dump onto one of our Microsoft support people, so you will probably put it somewhere on the swat share. The path for Brad Linscott, for example is:

\\wmphew1bbs001\swat$\BRAD\_MS\DF\_Issues\EEE\_11-07-18

Creating a Dump

As mentioned above a good overview on creating a dump is given in the blog

http://blogs.msdn.com/b/tess/archive/2006/01/11/back-to-basics-how-do-i-get-the-memory-dumps-in-the-first-place-and-what-is-sos-dll.aspx

Besides reading that, here are some specific steps I have used when working with Microsoft to debug problems.

Creating a Crash Rule with DebugDiag

Open DebugDiag. This will probably need to be done from the command line in admin mode, since problems are usually investigated on a machine where the application is running under a user account. Here is a screen shot of running the command line. You can probably skip the dbgsvc /Service step, but if you get an error setting the crash rule that the service is not running, you may need to execute it:

The DebugDiag screen will appear as follows. This one shows an existing rule, but the first time it will be empty and you will have to create the crash rule.

Clicking Add Rule shows the following where you choose Crash:

Choose a specific process:

Type in “DFContainer.exe”

Click Next.

Click Next again.

Now you start adding exceptions. Usually you have an idea of which exceptions to add from the problem which has been occurring. A dump will be created when the application crashes on the exceptions chosen.

You can choose an exception from the list, or you can type the code in manually. It is important to choose “Full User Dump” as the action type.

Some of the common exceptions chosen are displayed here:

When choosing a CLR (.Net) exception such as E0434F4D, you then specify which specific exception, and can have more than one. In this case you will have to type in the exact Exception name with correct case (e.g. System.OutOfMemoryException). The following screen shot shows a System.ExecutionEngineException:

After setting the exceptions click “Save & Close”. The output paths will be shown:

Finally you indicate whether the rule should be activated right away. You deactivate and activate the rule any time by right clicking on it.

When the rule is active, a dump will be created whenever the chosen executable exits because of one of the selected exceptions.

It is also possible to choose a running process through the processes tab, and by right clicking on it create a dump or do other analyses:

Command Line Debugging

An alternative to using DebugDiag is to debug from the command line. This offers the advantage of watching as the dump is shown on the screen, breaking at various exceptions.

This can be invoked by typing the following command while the executable is running as seen in the following screen shot where the bottom command prompt is the output screen.

ntsd –pn DFContainer.exe

When a break occurs, as here you can continue by entering ‘g’.

To avoid repeated breaks at a n error you can run the following command which would skip all access violations:

sxd av

then enter ‘g’ to continue.

You can create a dump as follows:

.dump /ma output\_file

If you want to create a dump while the application is running, interrupt it first as with CTL-Break.

If you wanted to investigate heap corruption, you would run the application verifier before debugging:

1. Run application verifier

2. Add the DFContainer.exe app

3. Deselect all on the right hand side except for heaps

4. Right click on heaps and make sure full is checked

5. Start the application

cscript adplus.vbs -quiet -hang -pn dfcontainer.exe -pn thomsononefull2008.exe -pn dfglobalhost.exe

cscript adplus\_old.vbs -quiet -hang -pn dfcontainer.exe

TO TELL ADPLUS TO GET A DUMP ON A CRASH AND RUN IN THE BACKGROUND:

Adplus.exe –crash –pmn DFContainer.exe -0 .

PERFORMANCE MONITOR

.Net Memory – Bytes in all heaps – Managed memory

Process -> Private Bytes – all memory – as seen in task manager commit size

Using DebugDiag to get a dump when a memory threshold is exceeded

Choose Performance

Add Perf Trigger

Choose Process

Choose Private Bytes

Add Process

Double Click Trigger

Enter Action Rules

Add Dump Target = process

Configure Dump Series

Name Rule

Analyzing the Dump

It is very difficult to glean anything useful from a dump without the aid of an experienced native debugger, so for the most part once we have dump we will pass it along to one of the Microsoft support personnel.

The best place to start learning about analyzing a dump is at the Tess Ferrandez blog :

http://blogs.msdn.com/b/tess

As mentioned above Visual Studio 2010 has some tools which make debugging less cryptic.

Use either ntsd -z <dumpname> or windbg

Before using Windbg, you can download some useful extensions:

http://www.microsoft.com/en-us/download/details.aspx?id=10640

unzup the following into the Windbg folder:

Sos.zip

Adplus\_scripts.zip

SIEExtPub.zip

To check bitness of a binary: corflags.exe <binary>

To start debugging with windbg open it either from Program Files, or the command line under debugging tools. Under File you can then open a dump:

At the command line you would then start entering the commands.

.symfix – gets MS symbols

.sympath+ srv\*C:\Program Files\Merrill Lynch\DF\bin – path to DF symbols

.sympath+ srv\*c:\symcache\*\\wmphew1bbs001\swat$\aaron\_ms\symcache – symbols on server if above is not available.

Or \\phccwdd9sj1pm1\New Folder\Symcache

.sympath – shows symbol path

X <function name> – tries to resolve symbols for a function: X kernel32!CreateFileW – shows address

ln <address> – shows symbols for an address

from command line: symstore.exe add will create a symbol store.

!chksym – checks validity of symbols

LOAD EXTENSIONS:

.reload /f – load symbols

.load SIEExtPub

.load SOSEX

.load psscor4.dll

.load C:\Program Files\DebugDiag\Exts\psscor2.dll –

.loadby sos mscorwks - for clr debugging – 32 bit

. loadby sos clr - for clr debugging – 32 bit proc 64 bit

loadby sos - for clr debugging – 64 bit proc 64 bit

.lastevent – shows likely cause of crash

Which thread is synchronizationcontext send calling:

http://blogs.msdn.com/b/dsui\_team/archive/2012/10/31/debugging-windows-forms-application-hangs-during-systemevents.userpreferencechanged.aspx

!peb – environment information including COMPUTERNAME

!uniqstack – finds unique stacks

dv – shows local variables in current frame

.frame <number> - sets current frame number

dt – dumps data structures

u – disassembles code at current instruction pointer

ub – disassembles code going backwards

uf – disassembles entire function

You would then typically start by examining the native stack with:

~ summarizes threads

!DumpDomain – dump1 Application Domains – Domain 1 is the default domain.

!DumpAssembly <assemblyaddress>

!DumpObject (!do) – dump object

!DumpArray (!da) – dump array of objects

!PrintException (!pe) – Prints Exception. Use !do for custom exceptions for extra object information.

sxe <exception code> - breaks on exception

!mdv – local variables

!strings – strings

!u – disassembles code at address

kb – shows frame call with args

~\*kb – shows what all threads were doing when dump was taken

!handle <address> 7 – details of handle at address

dc <address> - shows text in arg

And the clr stack with

!clrstack

!clrstack –l – loacal variables

!clrstack –a (or-p) - shows arguments

~\*e!ClrStack – displays all call stacks (~\*e = apply command to every thread)

dq <variable address> - (dump quad) shows contents of variable for value type for 64 bit

!threads – see threads

~0s – switch to thread 0

!analyze –v – exception analysis

!eeheap –gc – managed heap size

!gcgen <address> - (sosex) generation of an object

Garbage Collection

bp <address of Waitformultipleobjects on ­­­ thread> - will break when finalizer runs

Allocation Bursts cause gen 2 to grow. If this happens repeatedly fragmentation can result

Large Object Heap – greater than 85,000 bytes. Does not get compacted as Small Object Heap does, resulting in more fragmentation.

!GCHandles will show pinned objects

!DumpHeap -stat showing a lot of free space instances indicates fragmentation

Since !DumpHeap shows method handles you can match the objects to !GCHandles output

CLR 4.0 workstation Background GC allows ephemeral allocation to continue during GC

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Diagnostics

!VerifyHeap – Validates managed heap integrity

!GCHandles – handles in process

!GCHandleleaks – tries to find leaked handles (unreferenced). Only Strong and Pinned handles

!VMStat – Virtual memory stats

!gch (sosex) – more details on handles (type of handle)

!DumpIL – IL for address

Managed Debug Assistants (MDA)

Add to registry:

HKEY\_LOCAL\_MACHINE\Software\Microsoft\.NETFramework\MDA=”1”

Add config file: <appname>..exe.dma.config in same directory as exe.

MDA Categories:

P/Invoke

COM Interop

Loader

Threading

Miscellaneous

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Analyzing Loader memory:

!eeheap –loader – loader heap

!DumpDomain <domain address> (from loader heap) – shows details of domain

!Dumpmodule <module address> (from !DumpDomain, module name)

Dc <module starter address + length> - (from dumpmodule) shows contents

Search for a string indicating a possible dynamic assembly generation (e.g. xml serializer)

!gcroot ouput:

RSP = stack pointer register

Use !gcroot –nostacks to skip stacks

HANDLE(Pinned) indicates a static root

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Investigate Method Table (type handle)

!DumpHeap –t <object type>

!DumpHeap –mt <method table address> (from dumped object type)

!do <address of one instance of method table>

!gcroot <address> - finds root of an object

!gcroot -all <address> - finds all references to an object. In output handletable indicates static references.

!heap –s shows native heaps.

!heap –stat –h <heapaddress> gives breakdown of heap.

!heap –flt s <hexsize> - all heap blocks of this size

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Method Descriptors

!DumpMT –md – dumps method descriptors in method table.

!DumpMT <method descriptor pointer> - more details of method

CodeAddr – machine code

U <CodeAddr> - displays machine code

!P2MD – Returns method descriptor

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Synchronization

!syncblk

Index of lock

SyncBlock – pointer to syncblk

MonitorHeld – lock on syncblk

Recursion – How many times owning thread entered lock

Owning Thread Infor – thread owning lock

Syncblk Onwer – who owns syncblk

Lock information can be stored in the object header

SYNCBLK

An eight in the object header as below indicates a syncblk table. In this case the table is 2.

in a 0x08000002

THINLOCK

Information is directly in the object header (thread ID)

!do on local variables to see what they are (e.g. System.Object)

Instead of using !Syncblk it can be done as follows:

dd <address>-0x4 l1 (win86)

Something like 08000001 indicates syncblk index

If there is no 8, then the thread ID (clr not debugger) is shown indicating thinlock. Use !threads to translate clr id to debugger id.

!do on thinlock will show owner.

!DumpHeap – thinlock shows all thinlocks and owner

DEADLOCK

!syncblk can show deadlocks:

See if 2 locks are held by different threads

Check thread stacks. WaitForMultipleObjects indicates waiting – check forMonitorEnterWorker, and AwareLock

First argument in EnterEpilog frame indicates syncblk thread is waiting for

Third argument in EnterEpilog frame is lock object

Sosex extension !dlk does the above

DIFFERENCE BETWEEN MUTEX AND SEMAPHORE

Mutex is single handle, semaphore can be multiple. i.e. a semaphore can be set with account of 4, so four threads can enter before it is non signaled.

Mutex has thread ownership – owning thread must release it

Semaphore has no ownership – any thread can release and same thread can acquire it more than once posing a deadlock release

MANUAL RESET can release any number of threads, AUTO only 1 thread.

READERWRITERLOCKS

allow reading but not writing. Deprecated and replaced by System.Threading.ReaderWriterLoackSlim

!do with show \_dwWriterID shows that a lock is held. Last column shows owning thread for ReaderWriterLock

!do on ReaderWriterLockSlim show write waiters and write waiters. Too many writers waiting indicates Monitor should be used for performance.

writeLockOwnerID shows owning thread.

WaitHandle.Waitone in stack indicates wait for lock

Dumping

WIN32 LOCK CONSTRUCTS

.Net wrappers:

Semaphore

Events

Mutants

Etc.

Search for these types: e.g. !DumpHeap –type System.Threading.Mutex

!do <address> of object will show waithandle

!handle <value> 8 shows details of lock (owner = process and thread id (operating system))

~~[<os threadid>]s gives debugging thread id.

XXXXXXXXXXXXXXXXXXXXXXXXX

P/INVOKE

You can set a breakpoint in native code, e.g.bp KERNEL32!Beep

In Native Stack:

Mscorwks!DoNDirectCall\_\_PatchGetThreadCall+0x7b – calls from CLR into native

In Managed Stack:

DomainBoundILStubClass.IL\_STUB(UInt32, UInt32) – call from CLR to native

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

COMInterop

RCW – From .Net to COM

CCW – Exposing .Net to COM (e.g. delegate to COM)

TlbImp.exe <COM object DLL> /out:<interop assembly name> generateds Interop Assembly from COM object.

Generated code creates RCW

IN RCW: COM objects are reference counted

Marshal.ReleaseObject explicitly releases interface (not usually used. Instead GC collects.)

!syncblk shows total active RCW’s in the process

!COMState shows apartment model

!DumpHeap –type Interop – shows interop types.

Bp <dll name>!<class name>::<method name> Set breakpoint to method in Interop DLL (e.g. 07comobj!CBasicMath::Add)

If not loaded, a deferred breakpoint will be set.

GC CONSIDERATIONS

Avoid moving of objects by Pinning

Synchronous calls are automatically pinned

Asynchronous calls must be manually pinned for as short a period as possible.

INTEROP MDA can be used for debugging Interop

ENABLE DELEGATE MDA IN MDA CONFIG:

<mdaConfig>

<assistants>

<callbackOnCollectedDelegate listSize=”1500” />

</assistants>

</mdaConfig>

INTEROP LEAKS

If Managed heap stays flat but memory grows this indicates a native leak.

Need to use live UMDH to get logs to investigate

Use !address –summary to see total memory

!eeheap –gc shows managed heap.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

HEAP CORRUPTION

Pageheap annotates heap blocks to trigger fault at time of write

Light pageheap uses fill patterns to see if fill area is overwritten, but only on ‘new’ or ‘free’

Full pageheap uses fill patterns and guard pages – appending an inaccessible page of memory to every heap block. An AV happens as soon as the overwrite occurs. Full page heap is very memory intensive.

Manual Heap Debugging:

!heap –s : statistical dump of heaps

!heap –a <heaphandle> : gives more information

First entry is the address to the heap block. Add 8 bytes for user area

In each line there is a previous size and current size which should match from one line to the next. If not, this indicates the preallocation metadata has been overwritten.

dd <heapblock + 0x8> shows data

du <heapblock + 0x8> shows Unicode of data

Light PageHeap Debugging

Enabled through Application Verifier:

 Add Application

 Basics

 Heaps

 Right Click

 Properties

 Unclick Full

 OK

 Save

Run in debugger – Application Verifier will interrupt with issues it finds

Full PageHeap Debugging

Same as above but leave ‘Full’ clicked

Access Violation will occur closer to point of corruption.

XXXXXXXXXXXXXXXXXX

RESOURCE LEAKS

Handles, Synchronization Primitives, Heap Memory, Virtual Memory, etc.

Handles

!htrace gives call stack of handle creation.

!htrace –enable : enables handle tracing, taking snapshot

Enter !htrace; run; break; enter !htrace again. : Will show when handles open, close and call stacks

!htrace –diff : will only list open handles

Memory Tracking

UMDH : Tracks heap based memory;

Requires OS instrumentation to be enabled through gflags (image file tab -> name of app without path -> create user mode stack trace data base)

Run UMDH twice and compare output

Set symbols:

Set \_NT\_SYMBOL\_PATH=http://msdl.microsoft.com/symbols/download;<own symbol path>

1) Start app

2) Umdh.exe –p:<pid> > <logfile>

3) Continue app

4) Repeat 2 to a different log file.

5) Umdg.exe –d <log1> <log2> > <result log> (outputs diffs to a text file)

Results will show a ‘+’ sign for net gains per call stack

Preemptive Strategies

Prefast : Static source code analysis tool

XXXXXXXXXXXXXXXXXX

SYNCHRONIZATION

CONSTRUCTS:

Critical Section – User Mode, single process, cheaper than mutex which is kernel. No timeout besides infinite or zero. Zero – TryEnterCriticalSection. In .Net is a lock.

Event – Kernel Mode Handle passed to User Mode

Mutex – Kernel Mode Handle. Exclusive Access

Semaphore – Kernel Mode HandleMultiple Access through count

Interlocks – Atomic

Critical Section

Per Process. May or may not use a Kernel mode event

Allows 1 thread access to shared data

Represented by RTL\_CRITICAL\_SECTION

dt RTL\_CRITICAL\_SECTION <address>

Fields:

LockCount – threads waiting

RecursionCount – Number of times a thread has acquired the critical section (reentrancy). Entering and leaving must be equal before the thread is released

OwningThread (Os ID) : ~~[<OS ID>]s : gives debug ID>

SpinCount – How many times thread would spin before entering wait state. Avoids waiting to acquire kernel event.

DebugInfo – not always provided

ProcessLocksLists – Node in linked list of critical sections (per process)

CriticalSection – Back pointer to the critical section. Checking this field has address of critical section assures no corruption

CreatorBackTraceIndex – used to collect extended instrumentation

!cs <address> : displays critical sections

Mutex (mutant)

Can be used across processes by name

In user mode is a handle

!handle <handle value> to view

Parrallel.For : synchronized for loop

Interlocked.Increment : synchronized ++ operator.

Semaphore

Named across processes

Specify number of threads which can access it.

!handle to view

ANALYZING ISSUE

RtlEnterCriticalSection : Waiting to enter CS

WaitForSingleObject/WaitForMultipleObjects : waiting for lock

Spinning – use !runaway

FINDING OWNER OF CRITICAL SECTION

1st arg in RtlEnterCriticalSection is address of CS

Use !cs <address> to see details

DEADLOCK

Detect deadlock by checking owning threads of critical sections

OPHANED THREAD

If !cs <handle> gives “Illegal thread error” : indicates orphaned thread (CS entered but not left. Terminate thread could also cause this.)

Application Verifier can be used to identify orphan:

Enable Locks button to cause break in debugger when an orphan occurs. Right click and Check Verifier Stop options to see when Application Verifier will stop.

FIND THREAD HANDLE IN WAITFORSINGLEOBJECT

.frame <frame number counted from top of stack starting with 0>

dv will show handle of thread owning lock

!handle <handle> <number of lines of data (e.g. 8)>

~~[OS ID]s to get debugging thread ID

Switch to thread and investigate

LOADERLOCK

Whenever a dll is loaded a loader lock is acquired which stops other threads.

Be careful using DLLMain because a loaderlock is always taken.

XXXXXXXXXXXXXXXXXX

du <address> - Unicode representation

?<hexnumber> -> converts to decimal.

!dae – dump all .Net exceptions

or

!dumpheap -type Exception

!pe <exception addr>- exception details

!runaway shows thread cpu usage

!pae – all exceptions

!address <addr> - address details

lm – list modules beginning and end load addresses of modules. Load assemblies at beginning (01) or end (76)

!sam - assemblies

!critlist – find critical sections in use

!locks – show locks

!cs lists all critical sections

!dumpheap -type Exception –stat – get exceptions

!dso – dump objects

lmvm – module information (e.g. corporation)

From there you can examine specific threads, objects, cpu usage, etc, as described in Tess Ferrandez’s blog.

FIND PROCESS AND THREAD FROM AN RPC:

00f2f3bc 7752c982 0666f3f8 00f2f4b4 00f2f4a4 ole32!CRpcChannelBuffer::SendReceive2+0xc8

0:002> dd poi(0666f3f8 +18)+8 l2

00110730 000017c0 0000178c

PROCESS:

0:002> ?000017c0

Evaluate expression: 6080 = 000017c0

THREAD:

0:002> ?0000178c

Evaluate expression: 6028 = 0000178c

FIND METHOD NAME FROM HANDLER:

Dump the method name

The key point is the EventHandler object.

Collapse | Copy Code

0:000> !do 07b3ebd4

Name: System.EventHandler

MethodTable: 79b92a30

EEClass: 79882d28

Size: 32(0x20) bytes

File: C:\WINDOWS\Microsoft.Net\assembly\GAC\_32\mscorlib\v4.0\_4.0.0.0\_\_b77a5c561934e089\mscorlib.dll

Fields:

MT Field Offset Type VT Attr Value Name

79b9f744 4000076 4 System.Object 0 instance 0375891c \_target

79b9f744 4000077 8 System.Object 0 instance 00000000 \_methodBase

79b9ab88 4000078 c System.IntPtr 1 instance 7454518 \_methodPtr

79b9ab88 4000079 10 System.IntPtr 1 instance 0 \_methodPtrAux

79b9f744 400007a 14 System.Object 0 instance 00000000 \_invocationList

79b9ab88 400007b 18 System.IntPtr 1 instance 0 \_invocationCount

In fact, the first bold target address is 0375891c; if you scroll up, you'll find this is the MainWindow object.

The second \_methodPtr will tell us something. This is an IntPtr, a pointer to something. The address is 7454518. Usually, this points to a Method Descriptor. So let's dump the information by a SOS command.

Collapse | Copy Code

0:000> !ip2md 7454518

Failed to request MethodData, not in JIT code range

What's this? This means the method may not be JITted yet. Or, this is the entry point to a real method. Let's check the disassemble code.

Collapse | Copy Code

0:000> !u 7454518

Unmanaged code

07454518 e93336a703 jmp 0aec7b50

0745451d 5f pop edi

0745451e 0000 add byte ptr [eax],al

07454520 c45dc6 les ebx,fword ptr [ebp-3Ah]

07454523 0500000000 add eax,0

07454528 e8bbddce71 call clr!PrecodeFixupThunk (791422e8)

0745452d 5e pop esi

0745452e 0000 add byte ptr [eax],al

07454530 58 pop eax

07454531 5d pop ebp

The first line is a jmp, it jumps to address 0aec7b50. In fact, this address is pointed to a real managed method.

We can use !ip2md to dump the descriptor information.

Collapse | Copy Code

0:000> !ip2md 0aec7b50

MethodDesc: 05c65dc4

Method Name: SomeWPFApp.MainWindow.dispatcherTimer\_Tick(System.Object, System.EventArgs)

Class: 06117abc

MethodTable: 05c6689c

mdToken: 06000054

Module: 03572ea4

IsJitted: yes

CodeAddr: 0aec7b50

Transparency: Critical

\*\*\* WARNING: Unable to verify checksum for C:\SomeFolder\SomeWPFApp.

Source file: E:\work\SomeFolder\SomeWPFApp\MainWindow.xaml. @ 1300

FIND CPU USAGE:

0:000> !ThreadPool

CPU utilization: 8%

Worker Thread: Total: 11 Running: 0 Idle: 11 MaxLimit: 8191 MinLimit: 8

Work Request in Queue: 0

--------------------------------------

Number of Timers: 22

--------------------------------------

Completion Port Thread:Total: 4 Free: 4 MaxFree: 16 CurrentLimit: 4 MaxLimit: 1000 MinLimit: 8

Link for debugging exceptions:

http://blogs.msdn.com/b/johan/archive/2008/01/31/using-windbg-hunting-exceptions.aspx

HANG ANALYSIS:

!analyze -v -hang

Signature of Execution Engine Exception in log:

Begin~~~13932~~~TraceActivityStart~~~2012-05-31 14:47:09.7334~~~DFContainer~~~AMF\_OnContextChange1 - Exception caught but unhandled: System.Exception: OnContextChange

at MerrillLynch.Framework.Desktop.WMWAdapter.WebAppHelper.RaiseEvent(String attrName, Object attrValue) in e:\Builds\MLDF\WMWAdapter\_Main\Sources\WMWAdapter\Source\MerrillLynch.Framework.Desktop.WMWAdapter\WebAppHelper.cs:line 350

at MerrillLynch.Framework.Desktop.WMWAdapter.WebAppHelper.df\_OnContextChange(IContextItemCollection contextItems) in e:\Builds\MLDF\WMWAdapter\_Main\Sources\WMWAdapter\Source\MerrillLynch.Framework.Desktop.WMWAdapter\WebAppHelper.cs:line 327

at MerrillLynch.Framework.Desktop.WMWAdapter.WebAppHelper.FireOnSetContext() in e:\Builds\MLDF\WMWAdapter\_Main\Sources\WMWAdapter\Source\MerrillLynch.Framework.Desktop.WMWAdapter\WebAppHelper.cs:line 660

at MerrillLynch.Framework.Desktop.DFContainer.ThomsonExternal.<>c\_\_DisplayClass7.<AMF\_OnContextChange>b\_\_5()~~~~~~No Arguments Logged~~~End

wiagentie – in stack indicates qtp

TO FIND STA THREAD BEING SWITCHED TO:

We can grab the 1st parameter to GetToSTA to figure out what thread it is trying to get to

0a25ead0 77789905 049d75c0 0ca91600 00000000 ole32!GetToSTA+0x6f

If we dump it out with dc, the 2nd and 3rd dword will tell us what process it is trying to get to as well as the thread id (the reason we have to go in a little roundabout fashion to find these is because public symbols don't have type information, otherwise we could have dumped out the parameter with dt or looked at it in windbg, but now you will just have to rely on the fact that it is dword 2 and 3 (at least in this version of the dll)

0:041> dc 0x049d75c0

049d75c0 049d7640 049d7540 000091d0 00009a04 @v..@u..........

049d75d0 db94d897 0b8e692b e9cda17b 88696fd7 ....+i..{....oi.

049d75e0 e9cda17b 88696fd7 00001822 9a0491d0 {....oi.".......

049d75f0 80605d98 1cdb921b 00000103 03b80260 .]`.........`...

049d7600 049e86c0 00000000 00000000 00000000 ................

049d7610 00000001 ffffffff 00097ec0 04be013c .........~..<...

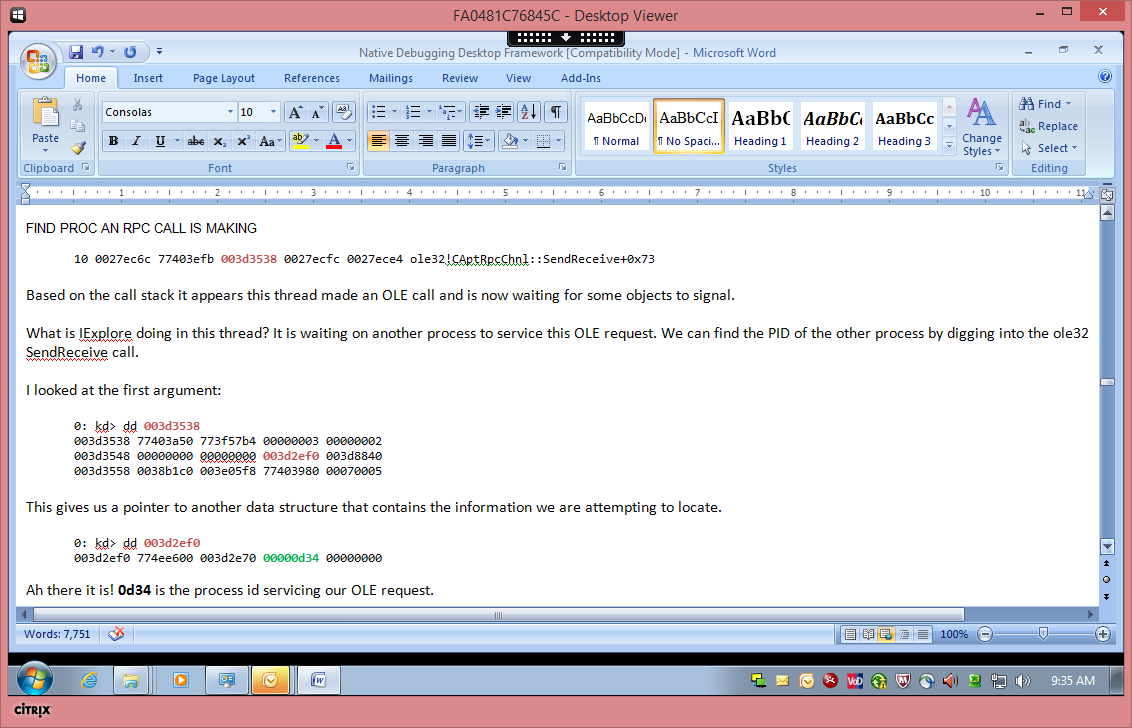
049d7620 00000003 00000000 00000000 00000000 ................

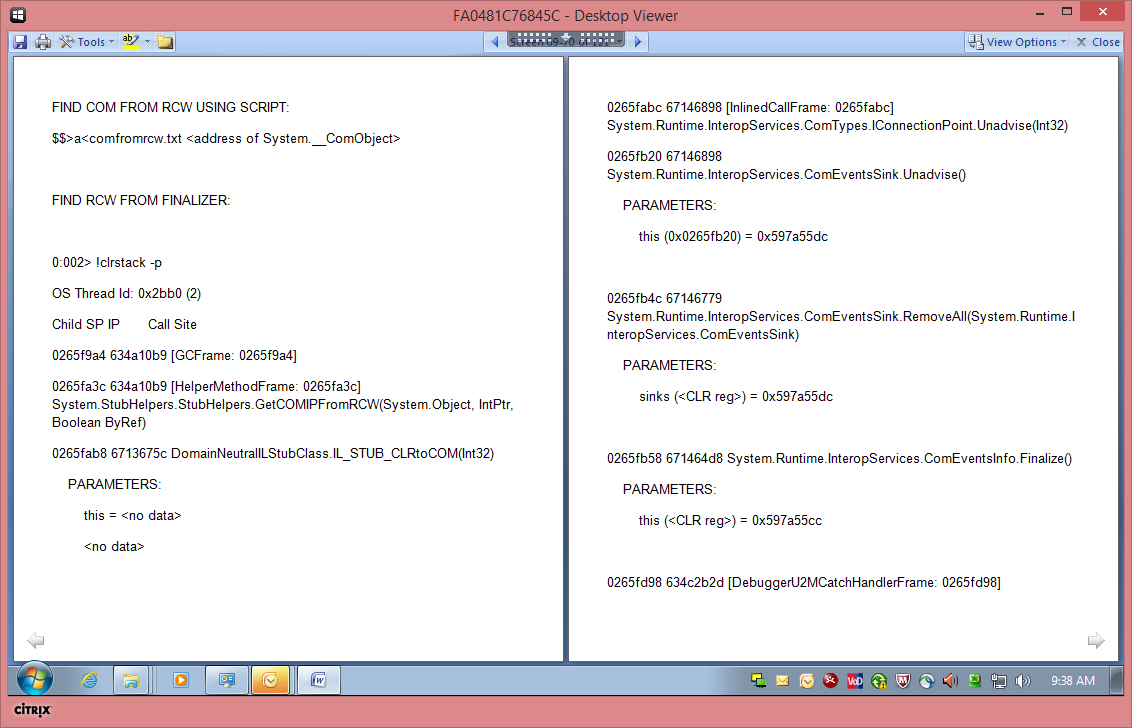
049d7630 00000000 00070005 00000000 00000000 ................

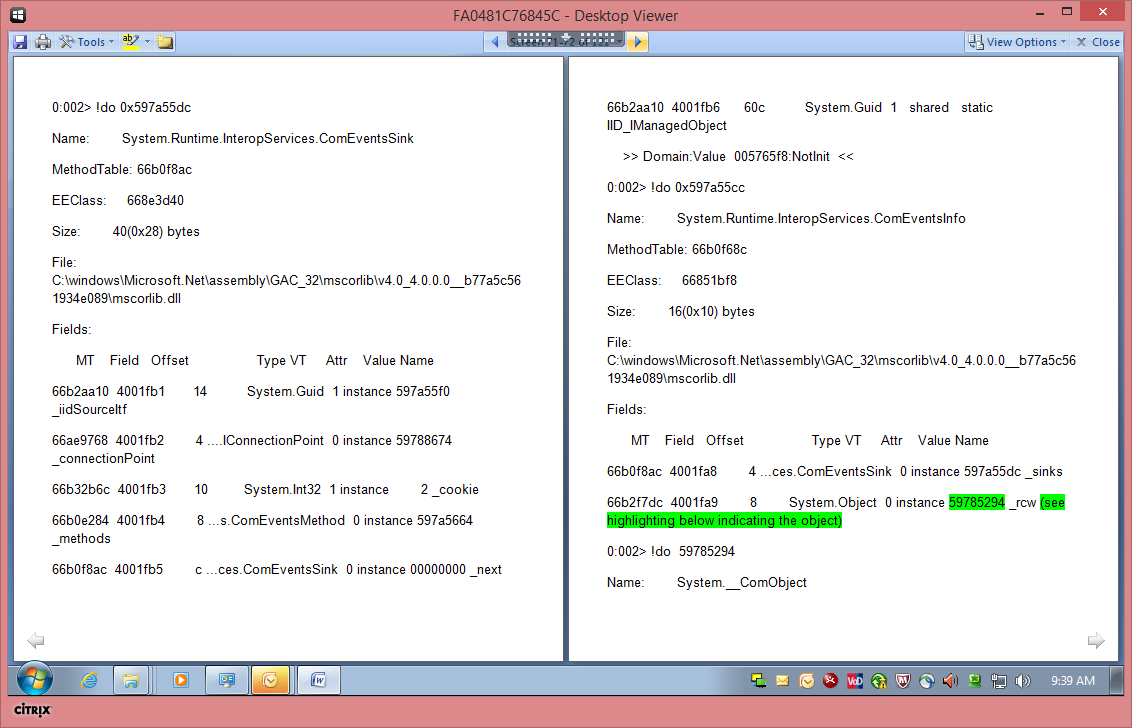
The PID (process ID) here is 91d0 and the thread id is 9a04

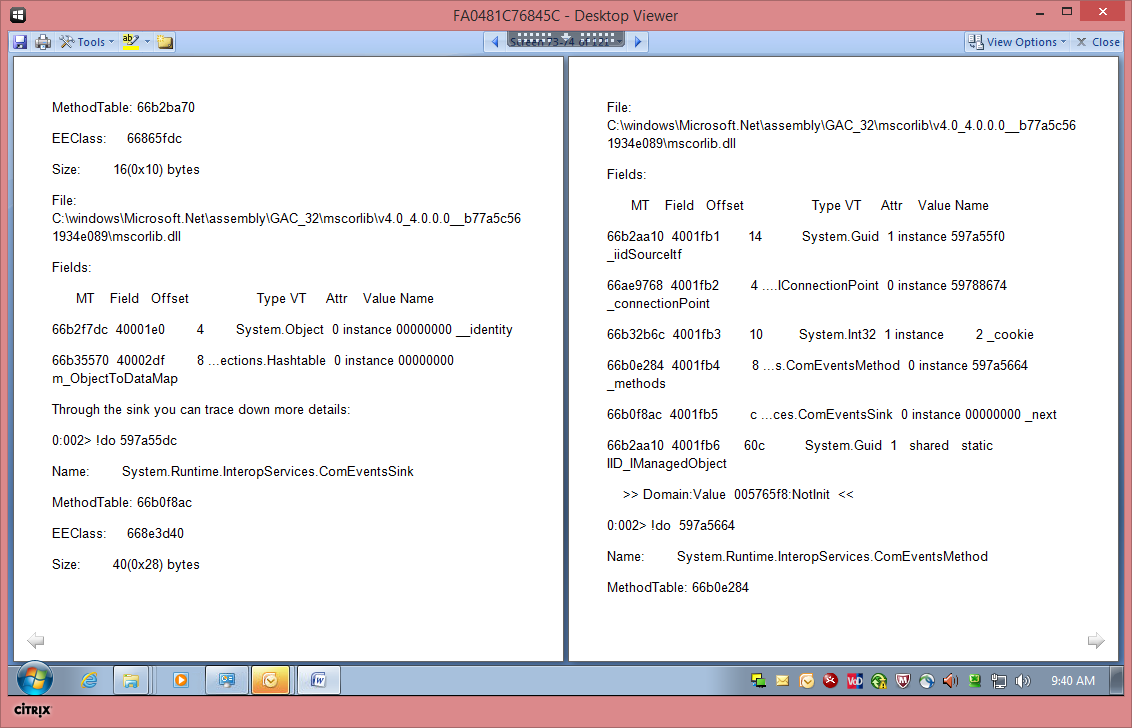
For SendMessage, the first arg is the handle: 000700ec

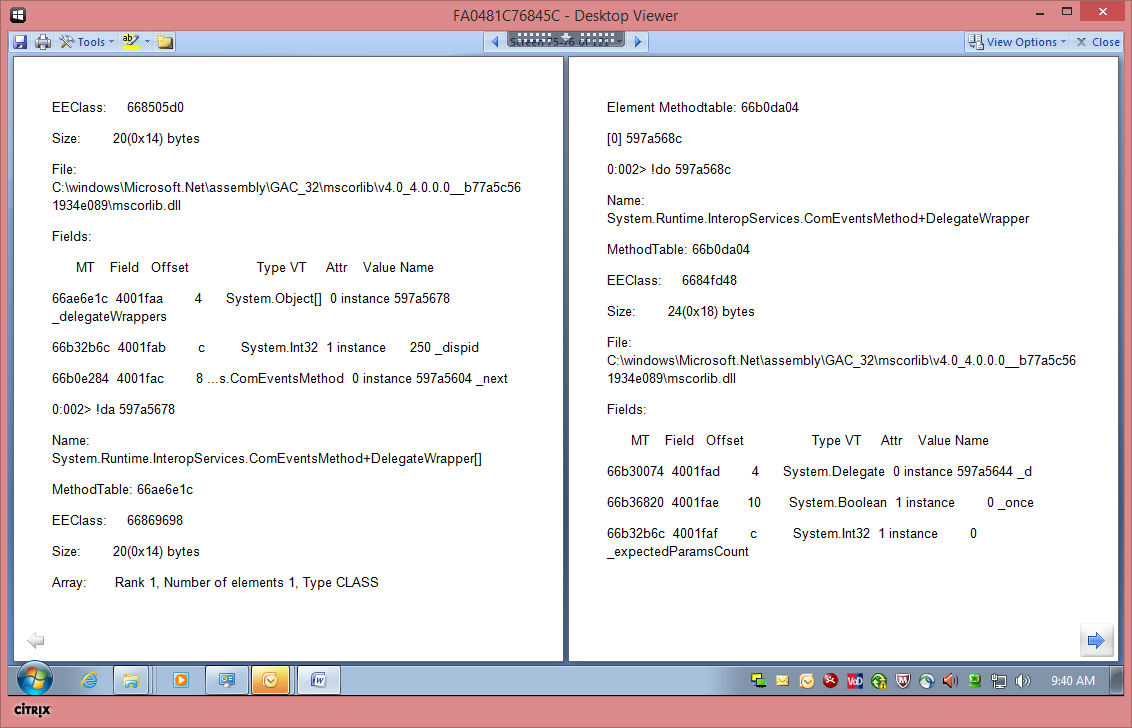
0006ed60 7cac404e 000700ec 0000004a 0006052e user32!SendMessageW+0x7f

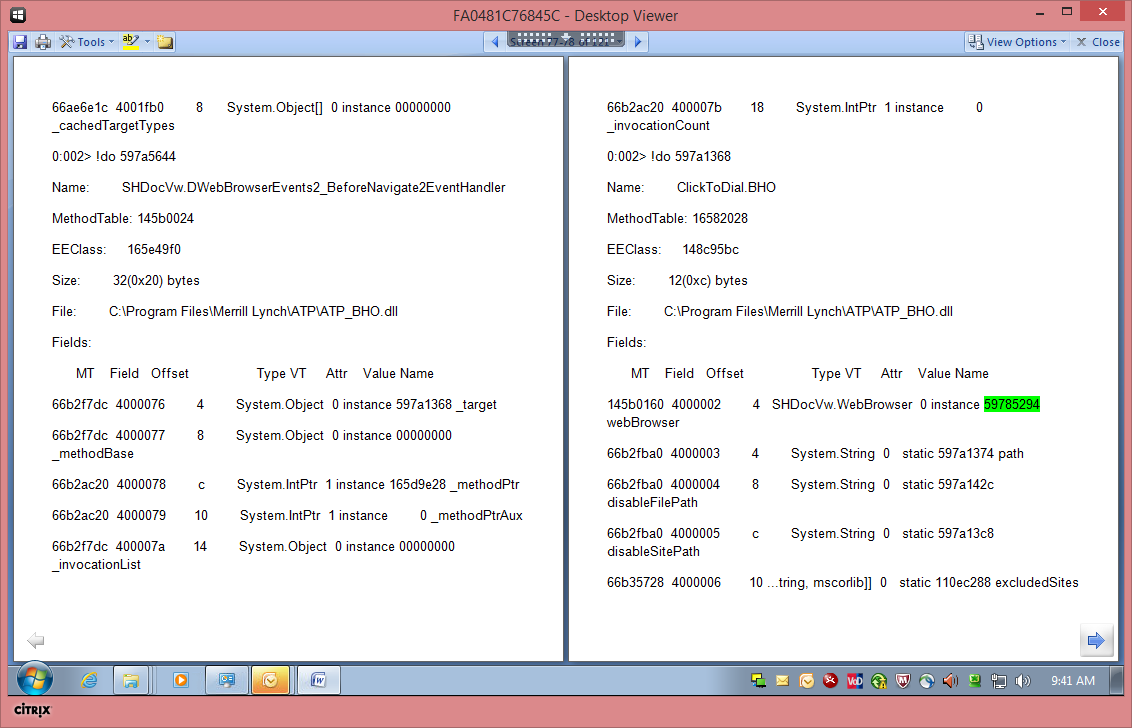


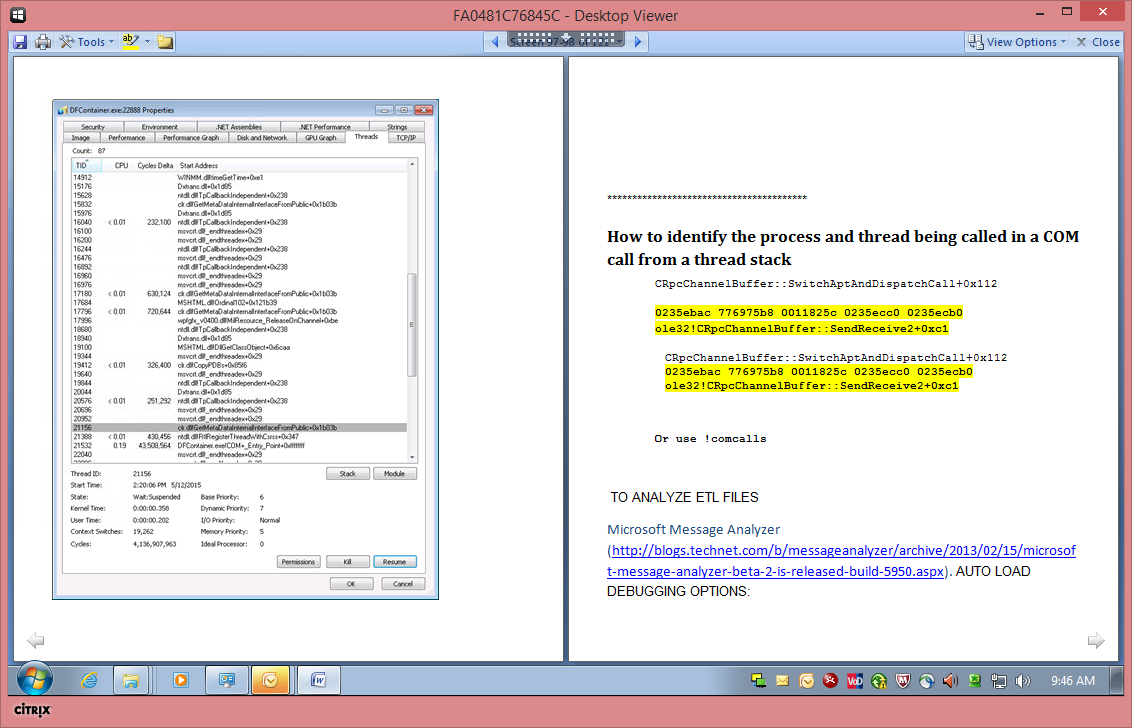


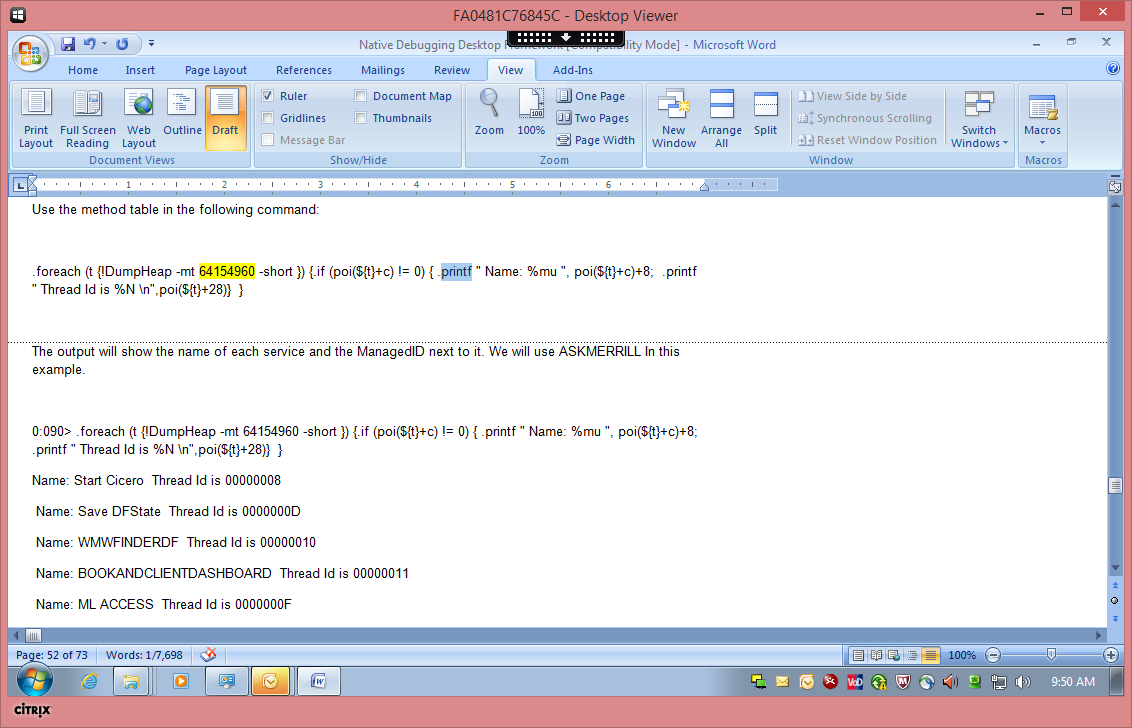












Investigating hangs:

http://blogs.msdn.com/b/ntdebugging/archive/2007/06/15/hung-window-no-source-no-problem-part-2.aspx

http://blogs.msdn.com/b/tess/archive/2006/10/16/net-hang-debugging-walkthrough.aspx

http://www.dumpanalysis.org/blog/index.php/crash-dump-analysis-patterns/

USING TTT (Time Travel Trace)

Here are instructions to get the TTT.

1. dfs.ml.com\amrs\groups\CAIISSFRAMEWORK\Users\Patrick\_McKeon\Debugging Tools for Windows (x86)\ TTTSetup\_x86\_external.msi

2. Download the TTT msi there, and run it on the box where you’re going to repro this.

3. Open a command prompt and CD to your TTT folder ( ex. cd "C:\Program Files\Debugging Tools for Windows\ttt" ). Note the Debugging Tools don’t have to be on this box.

4. Type this command for each process, using a separate cmd prompt for each process we are attaching to:

TTTracer.exe -attach <pid> -dumpFull

pid is the process id of the app that you are debugging. You should see a small dialog box pops up that has the title "dfcontainer01.run."

If you ever have to launch a process under the tracer (which we don’t want, in this case) just do:

TTTracer.exe -dumpFull -launch <path to exe> <launching process command line options>

5. Reproduce the problem.

6. Uncheck “Tracing on” in the dialog box and dismiss them. At this point you should see an .out file and a .run file under your ttt folder. Upload the .out and .run files as well as the dump file to your workspace that you downloaded the TTT installer from. Also, be sure to put the zipped-up dump there as well.

HANG CAUSED BY DICTIONARY ITERATION

The following in a thread during a hang indicates the possibility that an unsynchronized dictionary is being iterated and updated at the same time causing the hang:

0:000> !clrstack

OS Thread Id: 0x1328 (0)

ESP EIP

0006e504 792f518d System.Collections.Generic.Dictionary`2[[System.\_\_Canon, mscorlib],[System.\_\_Canon, mscorlib]].FindEntry(System.\_\_Canon)

A description of the problem: http://blogs.msdn.com/b/tess/archive/2009/12/21/high-cpu-in-net-app-using-a-static-generic-dictionary.aspx

EXAMPLE OF THREAD HOLDING A LOCK AND OTHERS WAITING:

NOTE: syncblk with psscor loaded will show waiting threads.

0:037> !syncblk

Index SyncBlock MonitorHeld Recursion Owning Thread Info SyncBlock Owner

256 0b4ff89c 97 1 0b63f418 127c 37 0177ff00 System.Object

-----------------------------

Total 1753

CCW 251

RCW 215

ComClassFactory 3

Free 55

0:037> ~24s

eax=00000bf0 ebx=0c9bee50 ecx=00000000 edx=fbffffff esi=00000000 edi=7ffdd000

eip=7c90e514 esp=0c9bee28 ebp=0c9beec4 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:

7c90e514 c3 ret

0:024> kb

ChildEBP RetAddr Args to Child

0c9bee24 7c90df4a 7c809590 00000001 0c9bee50 ntdll!KiFastSystemCallRet

0c9bee28 7c809590 00000001 0c9bee50 00000001 ntdll!ZwWaitForMultipleObjects+0xc

0c9beec4 79fccf6a 00000001 0b4ff8b0 00000000 kernel32!WaitForMultipleObjectsEx+0x12c

0c9bef2c 79fccb9a 00000001 0b4ff8b0 00000000 mscorwks!WaitForMultipleObjectsEx\_SO\_TOLERANT+0x6f

0c9bef4c 79fccca3 00000001 0b4ff8b0 00000000 mscorwks!Thread::DoAppropriateAptStateWait+0x3c

0c9befd0 79fccd38 00000001 0b4ff8b0 00000000 mscorwks!Thread::DoAppropriateWaitWorker+0x13c

0c9bf020 79fcceb9 00000001 0b4ff8b0 00000000 mscorwks!Thread::DoAppropriateWait+0x40

0c9bf07c 79e7547a ffffffff 00000001 00000000 mscorwks!CLREvent::WaitEx+0xf7

0c9bf090 79fd771d ffffffff 00000001 00000000 mscorwks!CLREvent::Wait+0x17

0c9bf11c 79f016d0 05fbfd00 ffffffff 0b4ff89c mscorwks!AwareLock::EnterEpilog+0x8c

0c9bf138 79fd7ca8 cbd43966 05fbfd00 05fbfd00 mscorwks!AwareLock::Enter+0x61

0c9bf1a0 79fd7a87 ffffffff cbd43a86 0c9bf28c mscorwks!AwareLock::Contention+0x199

0c9bf240 00ea25c1 0177ff00 00000000 00000000 mscorwks!JITutil\_MonContention+0xa3

TO FIND OBJECTS WAIT FOR MULTIPLE IS WAITING FOR:

06a4ebc8 769c15e9 00000002 06a4ec18 00000001 ntdll!ZwWaitForMultipleObjects+0x15

0:011> dd 06a4ec18

06a4ec18 0000055c 00000570 00000000 00000000

06a4ec28 000005d0 001f0003 06a4ec4c 70ca8bb9

06a4ec38 00000000 00000001 00000001 00000002

06a4ec48 00000000 06a4ebe4 70b01af9 06a4ed00

06a4ec58 769e6fd0 6737a60d 00000000 06a4ecac

06a4ec68 76151a2c 06a4ec18 06a4ec8c 00000000

06a4ec78 ffffffff 00000001 05f0f9c0 00000000

06a4ec88 00000000 0000055c 00000570 000fc000

0:011> !handle 0000055c f

Handle 0000055c

Type Timer

Attributes 0

GrantedAccess 0x1f0003:

Delete,ReadControl,WriteDac,WriteOwner,Synch

QueryState,ModifyState

HandleCount 2

PointerCount 4

Name <none>

No object specific information available

0:011> !handle 00000570 f

Handle 00000570

Type Timer

Attributes 0

GrantedAccess 0x1f0003:

Delete,ReadControl,WriteDac,WriteOwner,Synch

QueryState,ModifyState

HandleCount 2

PointerCount 4

Name <none>

No object specific information available

SAVEMODULE FOR DUMPBIN ANALYSIS:

!savemodule

INVESTIGATE PEEKMESSAGE ON STACK:

http://www.dumpanalysis.org/blog/index.php/2008/04/21/what-was-this-process-doing/

Exception Details:

!PrintException 60500188 (address of exception object)

FIND OBJECT SIZE:

0:000> .foreach(s {!dumpheap -mt 0x0993675c -short} ) { !objsize s; }

sizeof(0225d580) = 83,802,412 ( 0x4feb92c) bytes (MerrillLynch.Framework.Desktop.DFContainer.AppPreview)

<<TRUNCATED>>…

0:000> !do 0225d580

Name: MerrillLynch.Framework.Desktop.DFContainer.AppPreview

MethodTable: 0993675c

EEClass: 0cc6d268

Size: 24(0x18) bytes

GC Generation: 2

(C:\program files\merrill lynch\df\bin\DFContainer.exe)

Fields:

MT Field Offset Type VT Attr Value Name

0100c87c 400082a 4 ....ApplicationClass 0 instance 3c8aebb4 \_app

54829e7c 400082b 8 ...Media.ImageSource 0 instance 3c8b1d5c \_img

7b21d968 400082c c ...indows.Forms.Form 0 instance 116fc2b4 \_hostWindow

7a5edd78 400082d 10 ...angedEventHandler 0 instance 0254c53c PropertyChanged

Looking at each property we can see majority of the size is due to the image source.

0:000> !objsize 3c8aebb4

sizeof(3c8aebb4) = 1,816 ( 0x718) bytes (MerrillLynch.Framework.Desktop.DFContainer.extApplicationClass)

0:000> !objsize 3c8b1d5c

sizeof(3c8b1d5c) = 83,802,412 ( 0x4feb92c) bytes (System.Windows.Interop.InteropBitmap)

TO ANALYZE PROD ISSUES:

Team,

Pls use DFSupportDashboard Tool for analyzing any PROD issues. The “how to use” document is also in the same share(though it is self explanatory).

1. Unzip the DFSupportDashBoard.zip to your local folder.

2. You need to enable prompt for userid and launch the DFSupportDashboard.exe.

3. When prompted give wmtp\dftest3 | Welcome2

\\dfs.ml.com\amrs\groups\CAIISSFRAMEWORK\Projects\Desktop Framework\DesktopEvolution\Tools\DFSupportDashboard

TO ANALYZE ETL FILES

Microsoft Message Analyzer (http://blogs.technet.com/b/messageanalyzer/archive/2013/02/15/microsoft-message-analyzer-beta-2-is-released-build-5950.aspx). AUTO LOAD DEBUGGING OPTIONS:

http://blogs.msdn.com/b/andrew\_richards/archive/2011/05/05/windbg-context-menu.aspx

BREAK ON DF START:

BREAK BEFORE OOM

Gcbreakonoom

DebugDiag

Can be used to create and analyze dumps, and track memory leaks.

Custom Scripts can be created and placed in the DebugDiag Scripts folder

ProcDump

Used to generate crash dumps from many triggers

Has minimal footprint without setup – one small executable

Supports process reflection using –m switch. Clones parent process and operates on clone. Good for generating dumps from huge processes.

Start c:\tools\procedump.exe –c 90 –s 5 –n 3 –o <process name > /ma –w <dump output path> :

if cpu is over ninety for 5 seconds create up to 3 full dumps of process and wait until process starts to start mnitoring

ILDASM

Allows viewing of assemblies (methods, disassembly)

SOURCE LEVEL DEBUGGING

MDBG – Managed Debugger

Source can be downloaded

Path:

C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\Bin (32 bit)

C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\Bin\NETFX 4.0 Tools (64 bit))

Mdbg.exe – starts mdbg

Mdbg.exe <processname> - starts and attaches to process

a – lists processes

a <processname> – attaches to process

b [Classname.Method] – sets breakpoint

b [FileName:Line number] – sets breakpoint

f – resumes execution

ex – exit MDBG

k – kills currently active process

n – moves to next line

o – steps out of a function

s – steps into a function

pa – sets current source path

t – list or set active thread

u – moves active stack frame (up)

d – moves active stack frame (down)

l – displays loaded modules

p – displays local variables

w – displays the callstack for currently active thread

x – displays functions in a module

sh – shows lines of code around current breakpoint

EXAMPLE:

x – get module names

x <module name> - module details

b (breakpoint)

sh – see code

p – examine local variables

r – shows registers

64 BIT DEBUGGING

1st 4 arguments to a function are passed through the registry,

Mov instructions before call instead of push as in x86

Arguments can also be in Home Space on stack

Stack:

1. Home Space

2. Additinoal Arguments

3. Return Value

Bp x64!main – breakpoint at start of app (x64! + method name)

Uf . – unassembles function

Mov dword (or other type)– put on stack

Mov r9d.dword (or other type ) or r8d or edx or ecx – show placement in registers

THE PROLOGUE – instructions at beginning of function

Places arguments from registers into home space.

THE EPILOGUE – Destroys the stack frame

Restores Registers

Sets up return vale (RAX)

Returns to calling functon

EX:

Mov dword ptr [rsp+20h].eax

.

.

.

Mov eax.dword ptr [rsp+20h] – moves value from stack to return vale registry

OPTIMIZATIONS

Debug puts args in home space, Release may not

VISUAL STUDIO DEBUGGING

TO DEBUG INTO CLR CODE:

Debug->Options and Settings->Debugging->General->

Check Enable .NET Framework source debugging and Enable source server support

Under Debug->Options and Settings->Debugging->Symbols

Add the following server

http://referencesource.microsoft.com/symbols

TO ONLY SHOW MY CODE IN CALLK STACK:

Debug->Options and Settings->Debugging->General->

Check Enable Just My Code (Managed only)

KERNEL DEBUGGING

C:\Windows\system32\livekd –w : To kernel debug with windbg

!process 0 0 : Shows all processes.

!process <address> : specific process :

Cid = ProcessID

.process <process address> : switches active process

dt nt!\_EPROCESS : structures in a process

Pcb : KERNEL Process

dt nt!\_KPROCESS : structures in a kernel process

dt nt!\_PEB : Process Environment Block

!peb : details of block

ManualReset Event is called Notification in Kernel

Auto reset event is called Synchronization in Kernel

XXXXXXXXXXXXXXXXXXXXXXXXX

Performance Profiling

Dfs.ml.com\amrs\groups\CAIISSFRAMEWORK\Users\Patrick\_McKeon\ NP\_4800\_x64.zip

Dfs.ml.com\amrs\groups\CAIISSFRAMEWORK\Users\Patrick\_McKeon\ NP\_4800\_x86.zip

1) Run np.exe:

2) Choose ‘New Profiler Run’

3) Choose ‘Start a New Project’ and click ‘Next’|

4) Choose ‘.net Windows Application’ and click ‘Next’

5) Browse to Executable and click ‘Next’

6) Choose ‘Troubleshoot Performance Issues’ and click ‘Next’

7) Choose ‘Namespace based filter’ and click ‘Next’

8) Choose ‘Ignore System and Microsoft namespace’ and click ‘Next’

9) Set ‘Enable Pausing’ to True and click ‘Next’

10) Click ‘Start Profiling’ to start the app

11) To capture profiling, click ‘Start Capturing’

12) When done press ‘Stop Profiling’

13) Close application, Output will be under logs where NP is installed:

Network Monitor Trace

Dfs.ml.com\amrs\groups\CAIISSFRAMEWORK\Users\Patrick\_McKeon\ NM34\_x86.exe

Dfs.ml.com\amrs\groups\CAIISSFRAMEWORK\Users\Patrick\_McKeon\ NM34\_x64.exe

Perform steps 1-11 of Performance Profiling

1) Open Network Monitor:

2) Click ‘New Capture’, then ‘Start’

3) Click ‘Stop’ and ‘Save As’ to save report:

PerfView:

Pat, get DF in the state of the issue (since we can't have PerfView running all the time since we never know when we're going to have a repro). Then before you open the modal dialog for IVR, open Perfview. Using the GUI you can select Collect from the Collect menu. Then in the resulting window click Start Collection. Then click the button that repros the problem.

you should only do this on Win7.

Misc:

T1 logs on B7:

C:\Users\nbkvbyp\AppData\Local\Temp\Thomson ONE 5.0\Logs

64 BIT:

<http://blogs.microsoft.co.il/sasha/2013/05/15/obtaining-reliable-thread-call-stacks-of-64-bit-processes/>

CMKD extension:

<http://www.codemachine.com/tool_cmkd.html>

WPF LEAKS:

https://www.codeproject.com/articles/217670/the-third-way-of-change-notification-the-valuechan

public static void RemoveSourceFromValueChangedEventManager( object source)

{

*// Remove the source from the ValueChangedEventManager.*

Assembly assembly = Assembly.GetAssembly( typeof( FrameworkElement));

Type type = assembly.GetType( "MS.Internal.Data.ValueChangedEventManager");

PropertyInfo propertyInfo = type.GetProperty( "CurrentManager",

BindingFlags.NonPublic | BindingFlags.Static);

MethodInfo currentManagerGetter = propertyInfo.GetGetMethod( true);

object manager = currentManagerGetter.Invoke( null, null);

MethodInfo remove = type.GetMethod( "Remove", BindingFlags.NonPublic |

BindingFlags.Instance);

remove.Invoke(manager, new object[] { source });

*// The code above removes the instances of ValueChangedRecord from the*

*// WeakEventTable, but they are still rooted by the property descriptors of*

*// the source object. We need to clean them out of the property descriptors*

*// as well, to allow them to be garbage collected. (Which is necessary*

*// because they contain a hard reference to the source, which is what we*

*// really want garbage collected.)*

FieldInfo valueChangedHandlersInfo = typeof( PropertyDescriptor).GetField

( "valueChangedHandlers", BindingFlags.Instance | BindingFlags.NonPublic);

PropertyDescriptorCollection pdc = TypeDescriptor.GetProperties(source);

foreach ( PropertyDescriptor pd in pdc)

{

Hashtable changeHandlers =

( Hashtable)valueChangedHandlersInfo.GetValue(pd);

if (changeHandlers != null)

{

changeHandlers.Remove(source);

}

}

}