Crashing Your Way to Medium-IL: Exploiting the PDB Parser for Privilege Escalation

Gal De Leon (@galdeleon)
Palo Alto Networks

Who am I?

- Gal De Leon (@galdeleon)
- Principal security researcher at Palo Alto Networks
- Interested in fuzzing, vulnerabilities, exploits and mitigations
- Microsoft MSRC MVSR 2018, 2019, 2020
  - ~40 vulnerabilities

Agenda

- What are PDBs?
- Finding vulnerabilities in PDB parser
- Attack surfaces
- Exploit & Demo

What are PDB Files?

- Store debugging info (symbols) about an executable
- Function names, globals, type info ...
- Created from source files during build
- Used by debuggers

```
0:007> .reload /f notepad.exe
0:007> x notepad!*
00007ff7`9b4c4520 notepad!__scrt_uninitialize_thread_safe_statics (void)
00007ff7`9b4a86b0 notepad!ShowOpenSaveDialog (void)
00007ff7`9b4c09e8 notepad!StringLengthWorkerW (void)
00007ff7`9b4c38e0 notepad!initialize_printf_standard_rounding (void)
00007ff7`9b4a1640 notepad!`dynamic initializer for 'szFileName'' (void)
...
```

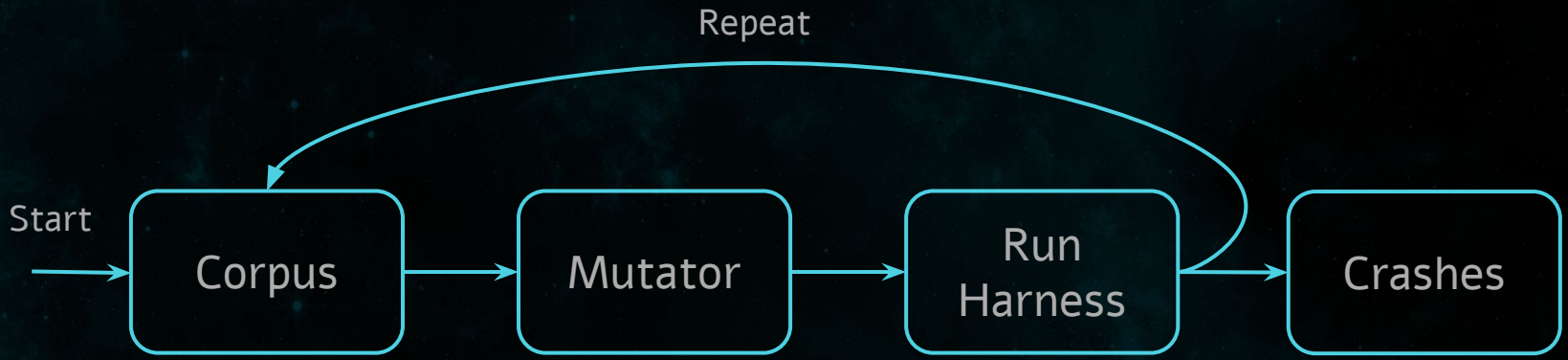
The PDB File Format

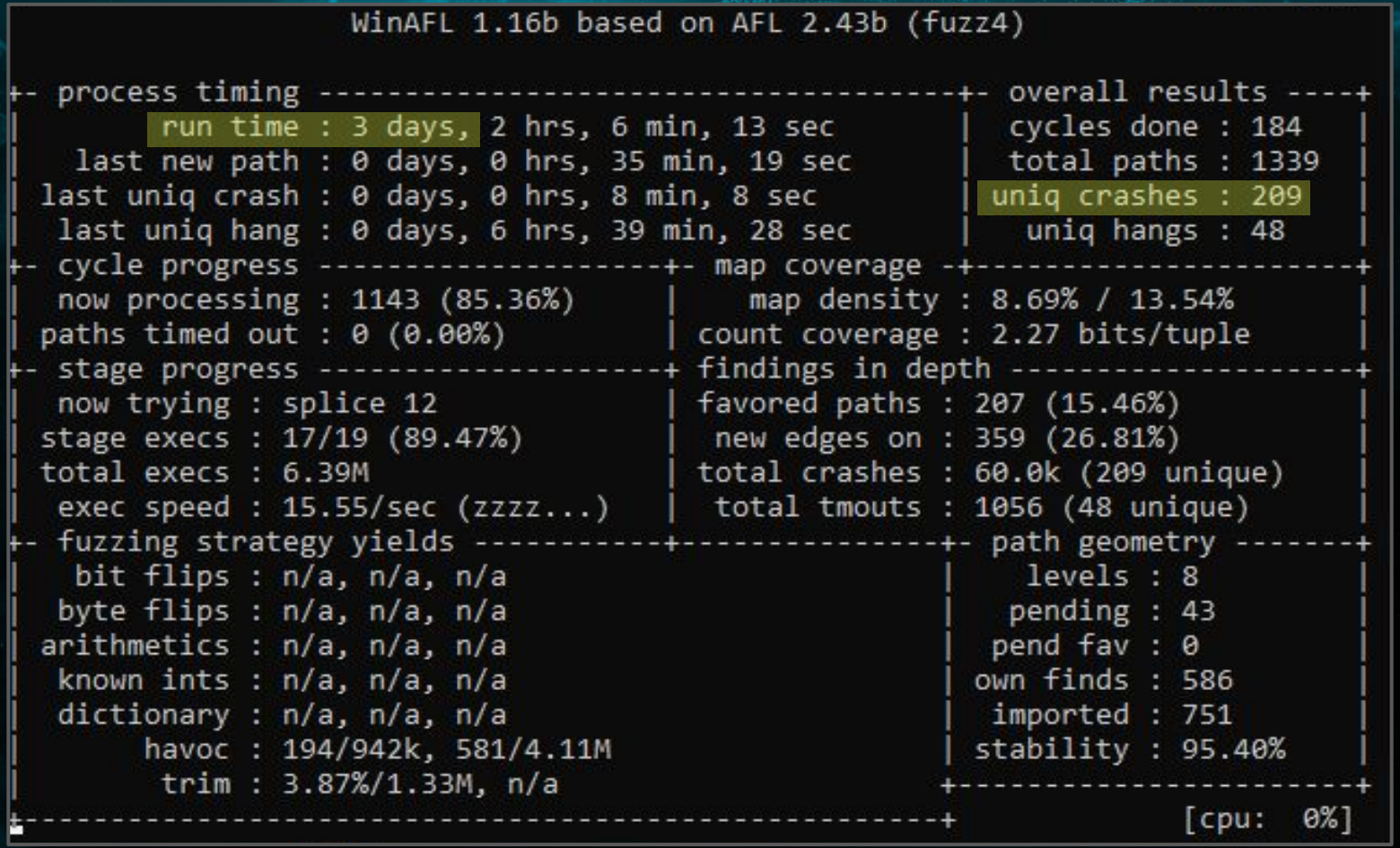
- Proprietary file format by Microsoft
  - Binary
  - Multi Stream Format (MSF)
- Open sourced for non-MS compilers to produce PDBs
- Parser implemented in Dbghelp.dll
  - Shipped by default
  - API to debug a process, load PDBs, extract symbols data …

Let’s Fuzz PDB

Fuzzing Setup

- Corpus – ~5000 PDBs from several sources
- Test Harness – A program that loads a PDB file and parse it
  - Dbghelp!SymLoadModule
- WinAFL fuzzer



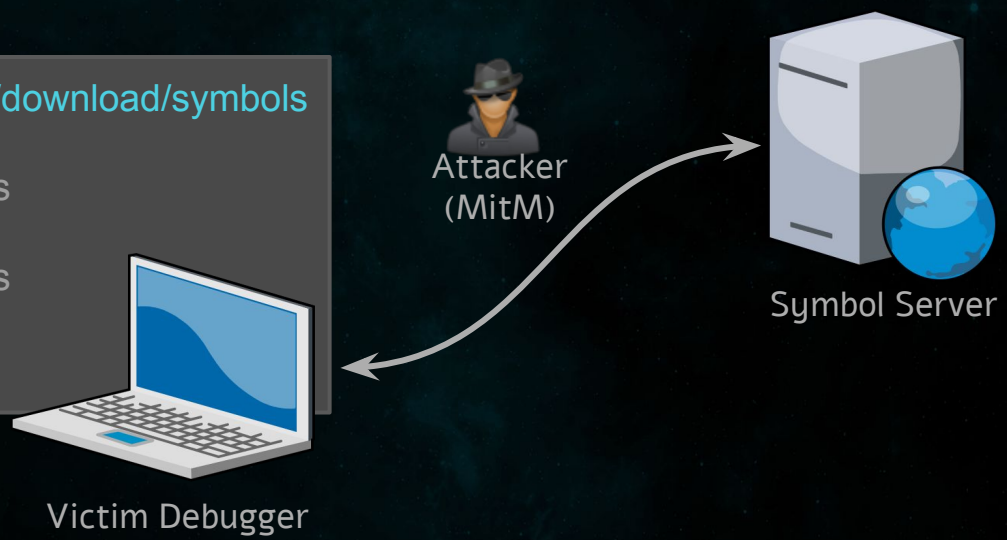

WinAFL 1.16b based on AFL 2.43b (fuzz4)
process timing
run time : 3 days, 2 hrs, 6 min, 13 sec
last new path : 0 days, 0 hrs, 35 min, 19 sec
last uniq crash : 0 days, 0 hrs, 8 min, 8 sec
last uniq hang : 0 days, 6 hrs, 39 min, 28 sec
overall results
cycles done : 184
total paths : 1339
uniq crashes : 209
uniq hangs : 48
cycle progress
now processing : 1143 (85.36%)
paths timed out : 0 (0.00%)
map coverage
map density : 8.69% / 13.54%
count coverage : 2.27 bits/tuple
stage progress
now trying : splice 12
stage execs : 17/19 (89.47%)
total execs : 6.39M
exec speed : 15.55/sec (zzzz...)
findings in depth
favored paths : 207 (15.46%)
new edges on : 359 (26.81%)
total crashes : 60.0k (209 unique)
total tmouts : 1056 (48 unique)
fuzzing strategy yields
bit flips : n/a, n/a, n/a
byte flips : n/a, n/a, n/a
arithmetics : n/a, n/a, n/a
known ints : n/a, n/a, n/a
dictionary : n/a, n/a, n/a
havoc : 194/942k, 581/4.11M
trim : 3.87%/1.33M, n/a
path geometry
levels : 8
pending : 43
pend fav : 0
own finds : 586
imported : 751
stability : 95.40%
[cpu: 0%]

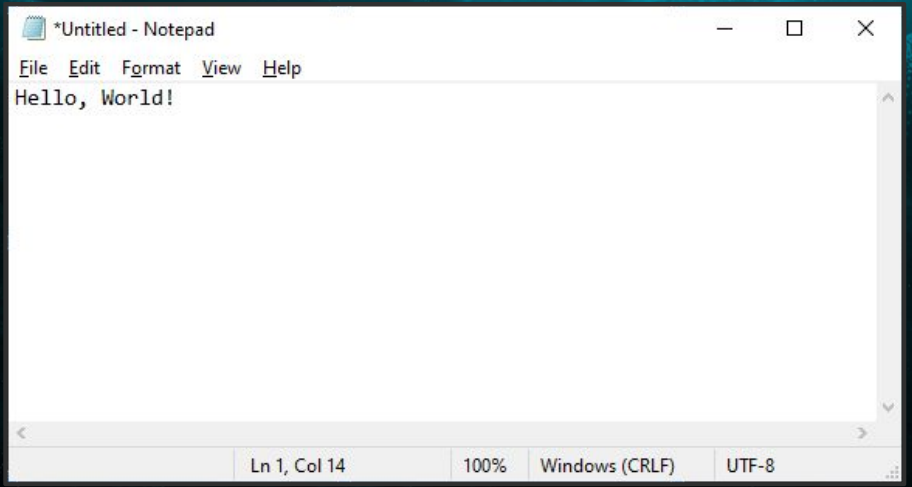
What is the Attack Surface?

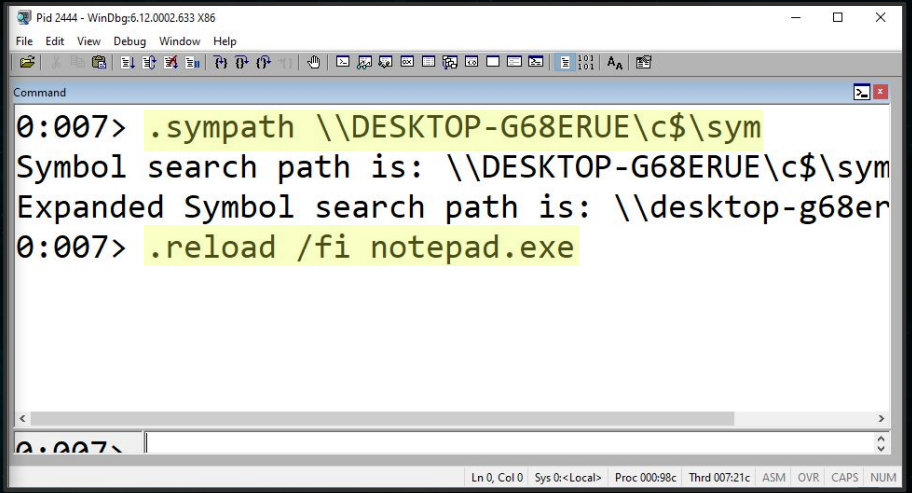
- Remote symbols servers
- Attacker controls / MitM symbols server can serve arbitrary PDBs

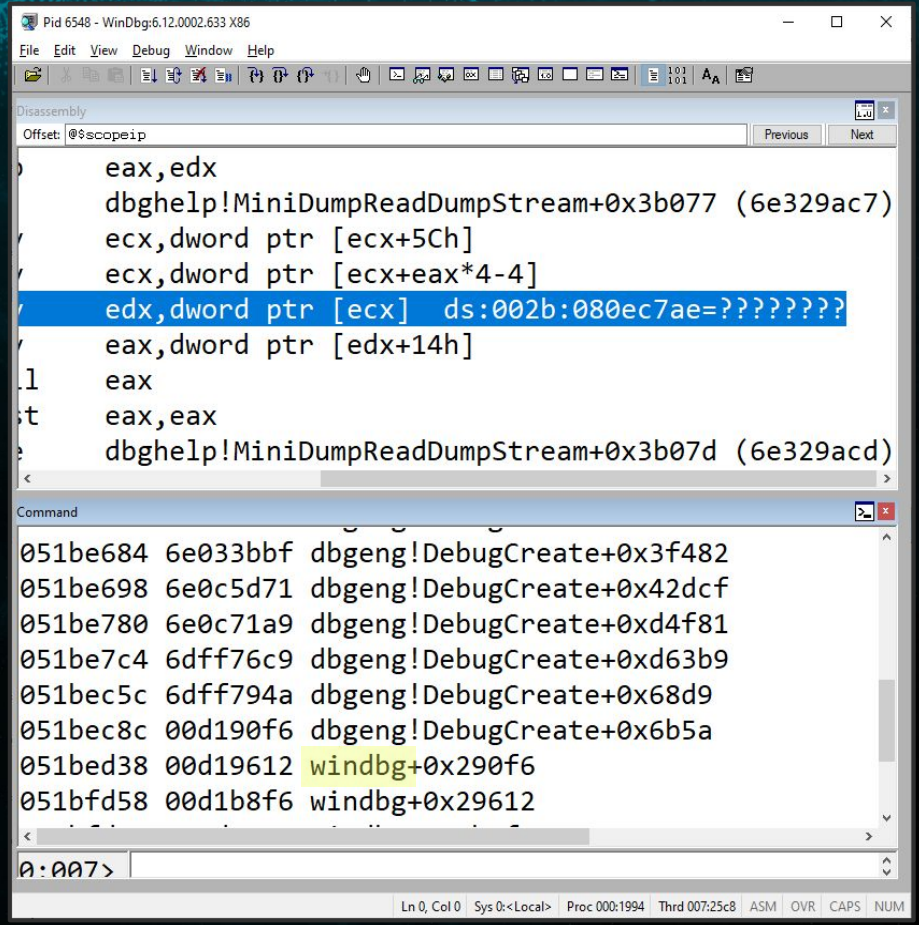
```
0:007> .sympath srv*http://msdl.microsoft.com/download/symbols
Symbol search path is:
srv*http://msdl.microsoft.com/download/symbols
Expanded Symbol search path is:
srv*http://msdl.microsoft.com/download/symbols

0:007> .reload /f notepad.exe
```




*Untitled - Notepad
File Edit Format View Help
Hello, World!
Ln 1, Col 14
100%
Windows (CRLF)
UTF-8


Pid 2444 - WinDbg:6.12.0002.633 X86
File Edit View Debug Window Help
Command
0:007> .sympath \\DESKTOP-G68ERUE\c$\sym
Symbol search path is: \\DESKTOP-G68ERUE\c$\sym
Expanded Symbol search path is: \\desktop-g68er
0:007> .reload /fi notepad.exe
Ln 0, Col 0
Sys 0:<Local>
Proc 000:98c
Thrd 007:21c
ASM
OVR
CAPS
NUM


Pid 6548 - WinDbg:6.12.0002.633 X86
File Edit View Debug Window Help
Disassembly
Offset: @$scopeip
Previous
Next
eax,edx
dbghelp!MiniDumpReadDumpStream+0x3b077 (6e329ac7)
ecx,dword ptr [ecx+5Ch]
ecx,dword ptr [ecx+eax*4-4]
edx,dword ptr [ecx] ds:002b:080ec7ae=????????
eax,dword ptr [edx+14h]
eax
eax,eax
dbghelp!MiniDumpReadDumpStream+0x3b07d (6e329acd)
Command
051be684 6e033bbf dbgeng!DebugCreate+0x3f482
051be698 6e0c5d71 dbgeng!DebugCreate+0x42dcf
051be780 6e0c71a9 dbgeng!DebugCreate+0xd4f81
051be7c4 6dff76c9 dbgeng!DebugCreate+0xd63b9
051bec5c 6dff794a dbgeng!DebugCreate+0x68d9
051bec8c 00d190f6 dbgeng!DebugCreate+0x6b5a
051bed38 00d19612 windbg+0x290f6
051bfd58 00d1b8f6 windbg+0x29612
0:007>
Ln 0, Col 0
Sys 0:<Local>
Proc 000:1994
Thrd 007:25c8
ASM
OVR
CAPS
NUM

Report to Microsoft MSRC

- 18-08-2020 - Initial report to Microsoft MSRC
- 15-09-2020 - Doesn't meet the bar for security servicing
  - Attack surface is too complex

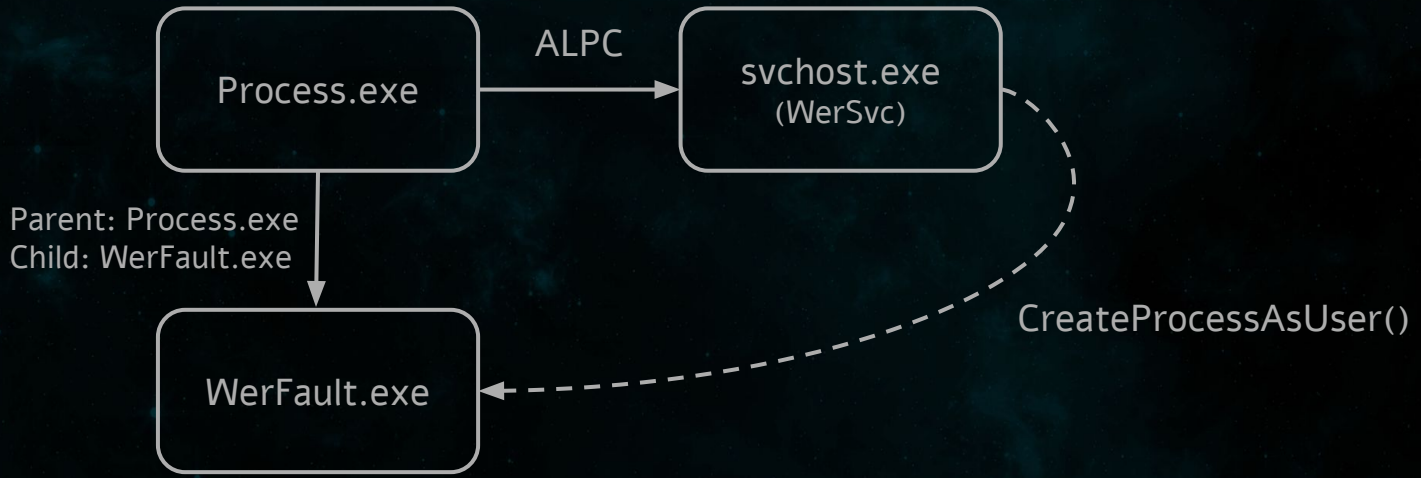
"... In this case, Microsoft has decided that it will not be fixing this vulnerability in the current version and we are closing this case. **In order to exploit this an attacker would need to control the symbol server or MitM the connection. Then the victim would have to load a PDB from the server.** At this time, you are able to blog about/discuss this case and/or present your findings publicly about the current version. ..."

Other Attack Surfaces

- Other components that use Dbghelp.dll to parse PDBs
  - How about elevation of privileges?
- Text-search 'Dbghelp.dll' in all binaries under C:\Windows
  - Appverif.exe, appverifUI.dll, comsvcs.dll, devinv.dll, taskkill.exe … **faultrep.dll, wer.dll**
- WER uses Dbghelp.dll!
  - I already discovered ~15 vulnerabilities in WER… Check out my BlueHatIL talk

Windows Error Reporting Recap

- WER collects info regarding crashes / hangs and reports to Microsoft
- Process crash -> WerFault.exe worker is launched



How Does WerFault.exe Use Dbghelp.dll?

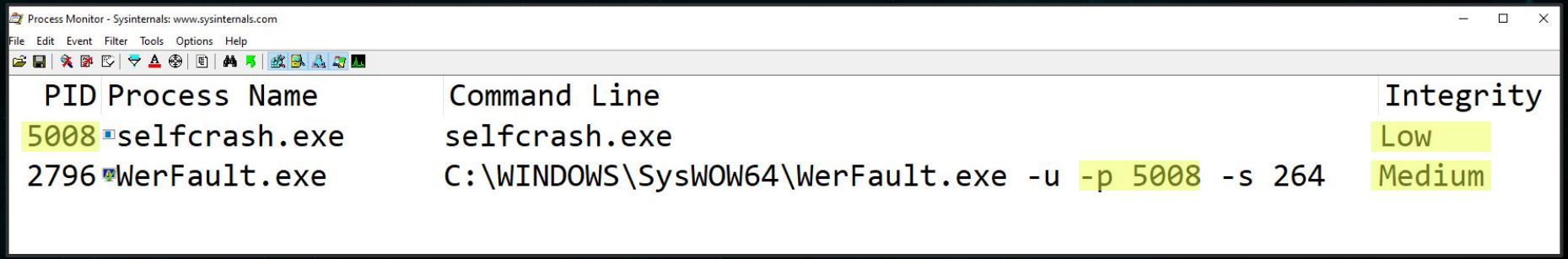
- Parse the stacktrace of the crashing thread
- Add stacktrace hash to error report
  - Allows Microsoft to group crashes by stacktrace

```
long long UtilGetStackTrace(long ProcessId, long ThreadId) {
    /* ... */
    HANDLE CrashingProc = OpenProcess(
        PROCESS_ALL_ACCESS, 0, ProcessId);
    SymInitialize(CrashingProc, NULL, fInvadeProcess=TRUE);
    /* ... */
    for ( ... ) { StackWalk(); }
```

Process Name	Operation	Path
WerFault.exe	CreateFile	C:\Windows\System32\notepad.pdb
WerFault.exe	CreateFile	C:\Windows\System32\notepad.pdb
WerFault.exe	CreateFile	C:\Windows\System32\notepad.pdb
WerFault.exe	CreateFile	C:\Windows\System32\ntdll.pdb
WerFault.exe	CreateFile	C:\Windows\System32\ntdll.pdb
WerFault.exe	CreateFile	C:\Windows\System32\ntdll.pdb
WerFault.exe	CreateFile	C:\Windows\System32\kernel32.pdb
WerFault.exe	CreateFile	C:\Windows\System32\kernel32.pdb
WerFault.exe	CreateFile	C:\Windows\System32\kernel32.pdb
WerFault.exe	CreateFile	C:\Windows\System32\kernelbase.pdb
WerFault.exe	CreateFile	C:\Windows\System32\kernelbase.pdb
WerFault.exe	CreateFile	C:\Windows\System32\kernelbase.pdb
WerFault.exe	CreateFile	C:\Windows\System32\gdi32.pdb
WerFault.exe	CreateFile	C:\Windows\System32\gdi32.pdb
WerFault.exe	CreateFile	C:\Windows\System32\gdi32.pdb

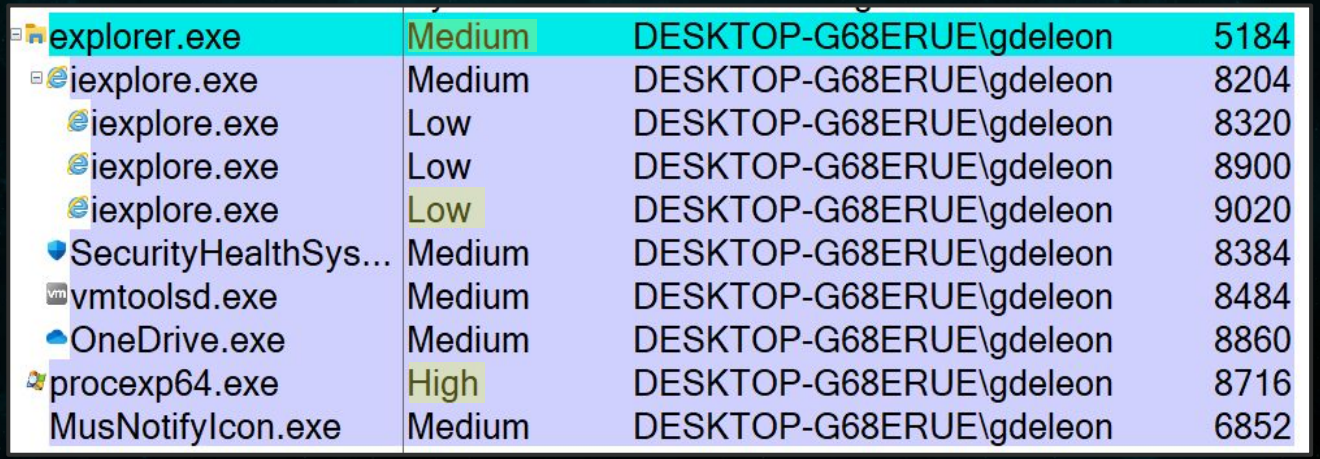
WerFault.exe Permissions

- Usually WerFault.exe runs with the same privileges of the crashing process
- One exception: Low-IL crash -> Medium-IL WerFault.exe



Integrity Levels (IL)

- ILs restrict processes running under the same user account

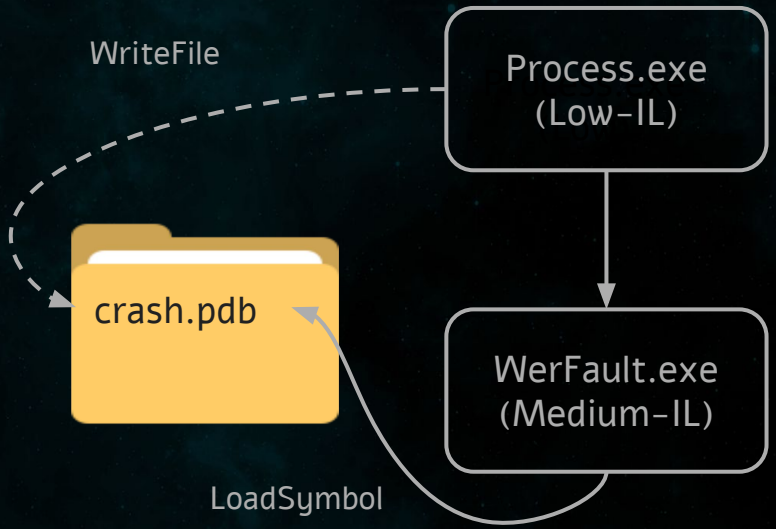


explorer.exe	Medium	DESKTOP-G68ERUE\gdeleon	5184
iexplore.exe	Medium	DESKTOP-G68ERUE\gdeleon	8204
iexplore.exe	Low	DESKTOP-G68ERUE\gdeleon	8320
iexplore.exe	Low	DESKTOP-G68ERUE\gdeleon	8900
iexplore.exe	Low	DESKTOP-G68ERUE\gdeleon	9020
SecurityHealthSys...	Medium	DESKTOP-G68ERUE\gdeleon	8384
vmtoolsd.exe	Medium	DESKTOP-G68ERUE\gdeleon	8484
OneDrive.exe	Medium	DESKTOP-G68ERUE\gdeleon	8860
procexp64.exe	High	DESKTOP-G68ERUE\gdeleon	8716
MusNotifyIcon.exe	Medium	DESKTOP-G68ERUE\gdeleon	6852

- Low-IL is used for sandboxing
  - E.g. iexplore renderers run under Low-IL
- Browser exploit chains: renderer RCE -> sandbox escape/EoP

What's the Game Plan?

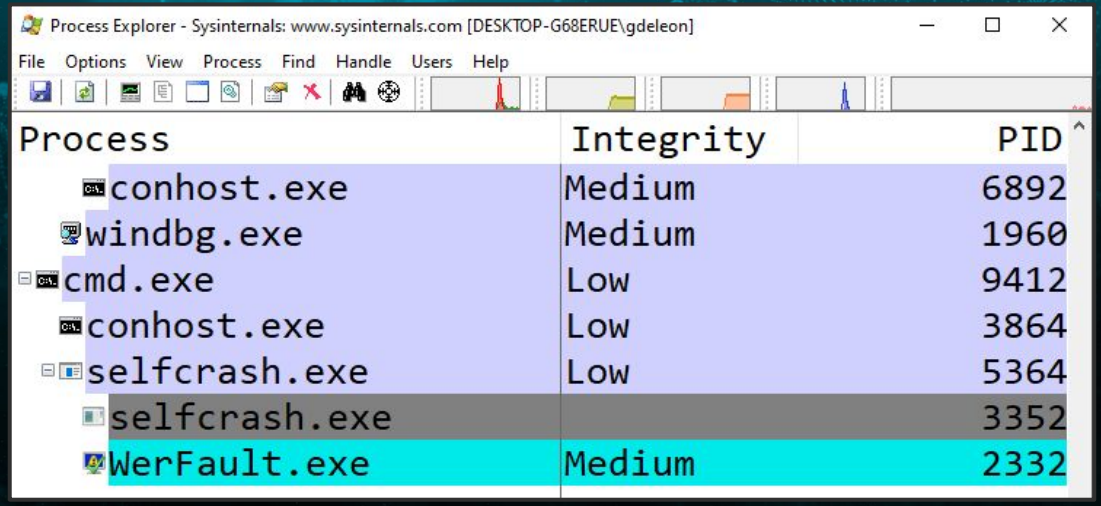
- Elevate privileges from Low-IL to Medium-IL
- From Low-integrity process:
  - Write malformed PDB file to disk
  - Crash my own process (Low-IL)
  - WerFault.exe (Medium-IL) launches
  - WerFault.exe loads malformed PDB file
    - Exploit PDB parsing bug for EoP



Load PDBs From Arbitrary Paths

- Cannot write to most paths as Low-IL
  - C:\users\gdeleon\AppData\LocalLow directory (%AppData%\LocalLow)
- How to get WerFault.exe to load PDB from %AppData%\LocalLow?
- PdbFilePath in PE / Executable header
  - Run my own EXE, point PdbFilePath to %AppData%\LocalLow

```
struct CV_INFO_PDB20
{
  CV_HEADER CvHeader;
  DWORD Signature;
  DWORD Age;
  BYTE PdbFilePath[];
};
```




```
(91c.1024): Access violation - code c0000005 (first chance)
6dd52abe mov     eax,dword ptr [ebx+164h]
ds:002b:08007e6e=????????
0:000> k
ChildEBP RetAddr
02b8a4ec 6dd4ac82 dbghelp!ModCache::pbSyms+0xe
...
02b8f7f0 0072ceaf faultrep!WerpInitiateCrashReporting+0x40f
02b8f838 007039cd WerFault!UserCrashMain+0x2b1
02b8f878 007419a0 WerFault!wmain+0x13e
```

Dbghelp.dll Bug

DbgHelp!SymCache::PsymForImodOff

```
int Index = …; /* Read from PDB file */;
/* Certain checks on Index */;
Object* Obj = ObjectsArray[Index - 1];
Obj->VirtualFunctionCall();
```

DbgHelp!SymCache::PsymForImodOff

- The bug – Index is allowed to be equal to 0
  - Type confusion

```
int Index = 0; /* Read from PDB file */;
/* Certain checks on Index */;
Object* Obj = ObjectsArray[Index - 1];
Obj->VirtualFunctionCall();
```

-1	0	1	2
??	Obj0	Obj1	Obj2

What's on Index -1?

- ObjectsArray is allocated on the heap
- There's a heap header prior to every heap allocation
  - Metadata about the allocation

-1	0	1	2
Heap Header	Obj0	Obj1	Obj2

Heap Header Structure (32 bit)

- 8 bytes header prior to every heap allocation
  - Low DWORD part is confused with Object*

-2	-1	0
Header High	**Header Low**	Obj1

```
0:000> dt -t _HEAP_ENTRY
   +0x000 UnpackedEntry    : _HEAP_UNPACKED_ENTRY
   +0x000 Size             : Uint2B
   +0x002 Flags            : UChar
   +0x003 SmallTagIndex    : UChar
   +0x004 PreviousSize     : Uint2B
   +0x006 SegmentOffset    : UChar
   +0x007 UnusedBytes      : UChar
```

Heap Encoding

- ‘Security-cookie’ to prevent heap overrun exploits
  - Header XOR random key (_HEAP->Encoding)
- 8-bytes key generated per heap at runtime (ntdll!RtlpCreateHeapEncoding)
- Part of the key is always set zero!
    - 2 high bytes of the second dword
  - Remains cleartext (X ^ 0 = X)

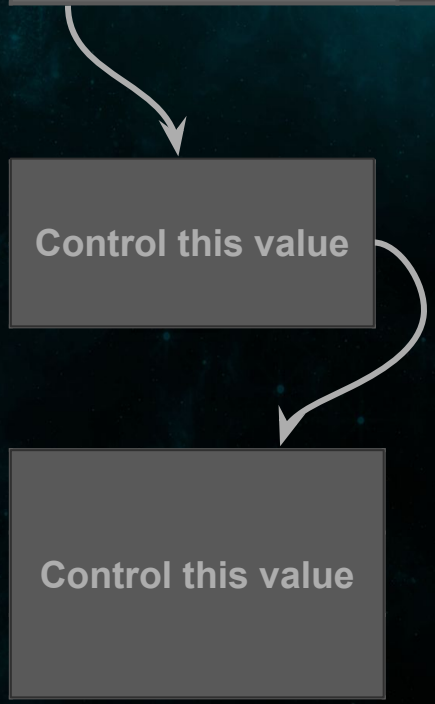
```
0:000> dt -t _HEAP
ntdll!_HEAP
   +0x050 Encoding
0:000> dd 01360000+50 L2
    01360050  18be3a5a 00006ab6
```

Is the Heap Header Value Predictable?

- Encoded header example: AAAAAAAA **XX**YYBBBB
  - XXYYBBBB => Fake Object*
- MSB (XX) meaning is UnusedBytes
  - Diff between malloc(size) and actual chunk size
  - Remains cleartext (XX ^ 0)
- ObjectArray is a small allocation
  - 12 bytes
- UnsuedBytes is predictable
  - 0x18
- Fake Object* is a low usermode address
  - 0x18XXXXXX

```
0:000> dt -t _HEAP_ENTRY
   +0x000 UnpackedEntry
   +0x000 Size             : Uint2B
   +0x002 Flags            : UChar
   +0x003 SmallTagIndex    : UChar
   +0x004 PreviousSize     : Uint2B
   +0x006 SegmentOffset    : UChar
   +0x007 UnusedBytes      : UChar
```

Heap Header	0	1	2
0x18XXXXXX	Obj0	Obj1	Obj2




```
int Index = 0;
Object* Obj = ObjectsArray[0 - 1 = -1];
Obj->Func2();
```

```
MOV eax, [ecx] ; Obtain vptr
CALL [eax + 4] ; Call vfunc
```

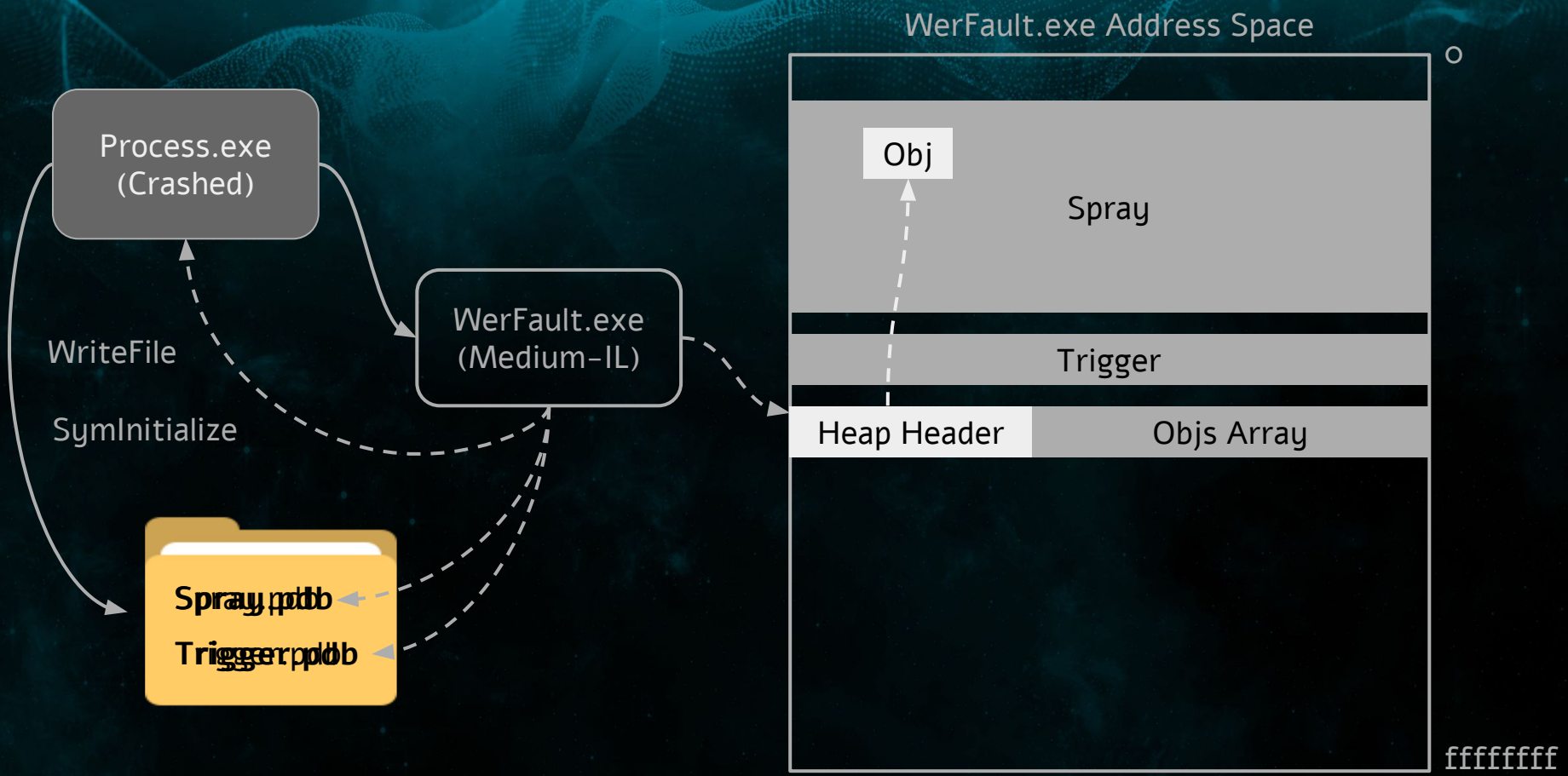
32Bit Crash? 32Bit WerFault!

- 32bit process crash -> 32bit WerFault.exe
  - Much easier to spray in 32bit
  - Allocators more predictable

selfcrash.exe	Low	32-bit	5364
selfcrash.exe			3352
WerFault.exe	Medium	32-bit	2332

Spray Primitive

- Goal: Spray WerFault.exe address space from crashing process (Low-IL)
- Dbghelp!SymInitialize loads PDBs for all loaded module
  - PDB #1 – Spray
  - PDB #2 – Trigger vulnerability
- How to spray from PDB #1?
  - Very large PDB file
  - Entire PDB is mapped using kernel32!MapViewOfFile
    - kernel32!MapViewOfFile is predictable (64k alignment)
  - Dbghelp doesn't unmap invalid PDBs
    - When bug is triggered (PDB #2) sprayed memory (PDB #1) is in place


Process.exe
(Crashed)
WerFault.exe
(Medium-IL)
WriteFile
SymInitialize
Spray.pdb
Trigger.pdb
WerFault.exe Address Space
0
Obj
Spray
Trigger
Heap Header
Objs Array
ffffffff

```
0:000:x86> ub
dbghelp!ModCache::pbSyms+0xe9:
701e2f39 8b8354010000    mov     eax,dword ptr [ebx+154h]
701e2f3f 8bb88c000000    mov     edi,dword ptr [eax+8Ch]
701e2f45 8b07            mov     eax,dword ptr [edi]
701e2f47 8b7078          mov     esi,dword ptr [eax+78h]
701e2f4a 8bce            mov     ecx,esi
701e2f4c ff1568243370    call    dword ptr [dbghelp!__guard_check_icall_fptr]
701e2f52 8bcf            mov     ecx,edi
701e2f54 ffd6            call    esi
0:000:x86> .frame 0
00 006fe2e0 701e2f56 0x0e0e0e0e
```

Where to Call to?

- Problem: CFG is enabled on WerFault.exe
  - CFI mitigation to prevent ROP/code-reuse attacks
  - Can only call CFG valid call targets
- kernel32!LoadLibrary is a valid CFG target!
  - Load DLL from '%AppData%\LocalLow' and run payload for entrypoint
- ASLR is not an issue
  - DLLs loaded at same address regardless of IL
  - Fetch kernel32!LoadLibrary address at runtime (Low-IL)
    - Write it to spray.pdb
- How to control kernel32!LoadLibrary argument?
  - Different calling conventions

Dbghelp Gadget (Arguments Reorder)

```
virtual long __thiscall DbhStackServices::GetSegmentDescriptor(..)

mov     edi, edi
push    ebp
mov     ebp, esp
…
mov     edi, ecx    ; Put ‘this’ in edi (‘this’=0x0c0c0c0c)
push    0
push    2Ch ; ','
mov     esi, [edi+0Ch]  ; Get next virtual func address, from ‘edi+0Ch’
mov     ecx, esi
push    dword ptr [edi+4] ; Push an argument on stack, from ‘edi+4’
call    ds:___guard_check_icall_fptr
call    esi
```

Escape Internet Explorer EPM Sandbox

- Enhanced Protected Mode – Low IL+AppContainer
  - iexplore.exe(Low+AC) -> WerFault.exe(Medium)
- PDB bug behaves differently
  - Fake Object*/_HEAP_ENTRY points to a kernel-mode address
    - LFH - _HEAP_ENTRY struct is different (ExtendedBlockSignature vs UnusedBytes)

```
(b00.990): Access violation - code c0000005 (first chance)
dbghelp!ModCache::pbSyms+0xe:
6db12abe 8b8364010000    mov     eax,dword ptr [ebx+164h] ds:002b:880002d8=????????

0:000> dd @edx-4 L4
007306ec  88000174 007076c0 007316f8 00731878

0:000> !heap -x @edx
Entry             User         Size    PrevSize   Unused    Flags
-----------------------------------------------------------------
007306e8     007306f0        0        -          8     LFH;busy
```

Escape Internet Explorer EPM Sandbox

- Process creation is allowed from IE's sandbox
- Create a child process and exploit the bug from there
  - iexplore.exe(Low+AC) -> exploit.exe(Low+AC) -> WerFault.exe(Medium)

Microsoft Fix (CVE-2021-24090 / KB5000802)

- WerFault.exe no longer parses PDB files
  - dbghelp!SymSetExtendedOption
  - (IMAGEHLP_EXTENDED_OPTIONS)3 = LOAD_SYMBOLS_DISABLED

WerFault.exe:

```
SymSetExtendedOption(3, TRUE);
v33_Ret = SymInitialize(v11_CrashingProc, NULL, TRUE);
```

Dbghelp.dll:

```
if (SymGetExtendedOption(3)) {
    _pwprint(L"load symbols is disabled!\n");
    return 4; }
```

Takeaways

- Fuzzing is very efficient for the right targets
- Exploit works despite all mitigations
  - 32bit compatibility layer isn't as strongly mitigated
- One bug, multiple attack surfaces

@galdeleon