Windows 10 Hooking Nirvana explained

11.04.2016

Preface

If you accidentally missed a very interesting RECON 2015 presentation from Alex Ionescu, then… I will not repeat. Watch, read, build some code and jmp back here after. Hooking Nirvana is a stealthy instrumentation technique used to control and monitor user-mode execution of running process. Hooking Nirvana is integrated into Windows kernel. Okay, now you know something. But what exactly do you know about how Hooking Nirvana works? If you're curios as I do, then follow me. I'll show how exactly Hooking Nirvana is implemented in kernel (well… ehm…), how it works on a different platforms and how it is important to use Google's images search (no, it's not) and do not use the very first picture (yes, it is).

Historical note

With full respect to Alex's work I should mention that there are some guys who were found (and used) a strange InstrumentationCallback field of the nt!_KPROCESS structure in 2013. I found an article of Everdox on Code Project named «Windows x64 system service hooks and advanced debugging» and a blog post named «Instrumentationcallback and advanced debugging» (read comments).

Let's have fun

I must say that InstrumentationCallback is a field of the nt!_KPROCESS structure from the Vista x64 RTM. It exists on 64-bit Windows only until Windows 10240 from where it appears as a field of the x86 nt!_EPROCESS structure also.

Windows 10586 x64:

```
struct _KPROCESS {
  /*0x0*/ /*|0x18|*/ struct _DISPATCHER_HEADER Header;
  /*0x18*/ /*|0x10|*/ struct _LIST_ENTRY ProfileListHead;
  /*0x28*/ /*|0x8|*/ unsigned __int64 DirectoryTableBase;
  /*0x30*/ /*|0x10|*/ struct _LIST_ENTRY ThreadListHead;
  /*0x268*/ /*|0x4|*/ unsigned long FreezeCount;
  ...
  /*0x26c*/ /*|0x4|*/ unsigned long KernelTime;
  /*0x270*/ /*|0x4|*/ unsigned long UserTime;
  /*0x274*/ /*|0x2|*/ unsigned short LdtFreeSelectorHint;
  /*0x276*/ /*|0x2|*/ unsigned short LdtTableLength;
  /*0x278*/ /*|0x10|*/ union _KGDTENTRY64 LdtSystemDescriptor;
  /*0x288*/ /*|0x8|*/ void* LdtBaseAddress;
  /*0x290*/ /*|0x38|*/ struct _FAST_MUTEX LdtProcessLock;
  /*0x2c8*/ /*|0x8|*/ void* InstrumentationCallback;            // I'm here
  /*0x2d0*/ /*|0x8|*/ unsigned __int64 SecurePid;
};
// size 0x2d8
```

Windows 10586 x86:

```
struct _EPROCESS {
  /*0x0*/ /*|0xa8|*/ struct _KPROCESS Pcb;
  /*0xa8*/ /*|0x4|*/ struct _EX_PUSH_LOCK ProcessLock;
  /*0xac*/ /*|0x4|*/ struct _EX_RUNDOWN_REF RundownProtect;
  /*0xb0*/ /*|0x4|*/ void* VdmObjects;
  /*0xb4*/ /*|0x4|*/ void* UniqueProcessId;
  /*0xb8*/ /*|0x8|*/ struct _LIST_ENTRY ActiveProcessLinks;
  /*0xc0*/ /*|0x4|*/ unsigned long Flags2;
  ...
  /*0x310*/ /*|0x4|*/ unsigned long KeepAliveCounter;
  /*0x314*/ /*|0x4|*/ unsigned long NoWakeKeepAliveCounter;
  /*0x318*/ /*|0x4|*/ unsigned long HighPriorityFaultsAllowed;
  /*0x31c*/ /*|0x4|*/ void* InstrumentationCallback;            // I'm here
  /*0x320*/ /*|0x4|*/ struct _PROCESS_ENERGY_VALUES* EnergyValues;
  /*0x324*/ /*|0x4|*/ void* VmContext;
  ...
};
// size 0x370
```

Starting from Windows 10 Hooking Nirvana engine is able to instrument not only native process, but a WOW64 process too. You can set up instrumentation callback manually from driver or with the help of undocumented NtSetInformationProcess routine passing process information class equal to 40 (0x28) and a valid filled structure. To instrument process's execution you should inject DLL into that process.

```
typedef struct
_PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION
{
    ULONG Version;
    ULONG Reserved;
    PVOID Callback;
} PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION;
```

On Windows 10586 x64 you still need to use an old method of callback set up (used in previous Windows versions) to instrument a WOW64 process. So, you'll need to pass a 64-bit callback's pointer instead of a pointer to the structure itself. Btw, I cannot confirm information about setting Version field to 1 for WOW64 instrumentation (see below).

Common stuff

Previously I told you to read comments of the «Instrumentationcallback and advanced debugging» blogpost. Waliedassar comment was very helpful, but this time I'll repeat. Callback set up step by step.

1. Checks ProcessInformationLength.
    1. If length is 8, then use a supplied 64-bit pointer as a callback pointer and fill a local structure on stack.
    2. Else, use provided structure.
2. Checks that Reserved field is 0.
3. Checks that Version field is 0 too.
4. Checks that callback address is in canonical form (48 bit).
5. References target process object by handle.
6. Gets caller's (current) process object and checks that it has a debug privilege enabled.
7. Acquires a rundown protection of target process.

Now we have only two options: native process and a WOW64 process. Let's start with native one.

Native process

1. Attaches to target process.
2. Calls MmValidateUserCallTarget passing callback address. MmValidateUserCallTarget (probably) checks that if target process has CFG, then it checks that a passed callback address belongs to the process's CFG map.
3. Detaches from target process.
4. Locks target process using PspLockProcessExclusive.
5. Fills callback address at pTargetProcess->Pcb.InstrumentationCallback.

6. Walks through the list of target process's threads and set «Instrumented» bit of nt!_DISPATCHER_HEADER structure on if callback address is not null. Unset if null.
7. Unlocks target process using PspUnlockProcessExclusive.

WOW64 process

Well, a little bit of architecture's nightmare begins.

1. Checks that both a calling process and a target process are WOW64. Why? I don't know.
2. Attaches to target process.
3. Checks callback address against maximum user-mode address. Calls MmValidateUserCallTarget passing callback address.
4. Now feel the moment.
    1. Gets a pointer to the WoW64Process field of nt!_EPROCESS. It's a pointer to a small structure (nt!_EWOW64PROCESS) that contains a pointer to PEB, to nt!_PEB32 to be precise.
    2. Gets that pointer to PEB.
    3. Fills callback address inside PEB structure? No, fills something beyond PEB: mov [rcx+**46Ch**], eax. Because sizeof(nt!_PEB32) is exactly 0x460 bytes on Windows 10586 x64.
5. Detaches from target process.

```
struct _EPROCESS {
  /*0x0*/ /*|0x2d8|*/ struct _KPROCESS Pcb;
  /*0x2d8*/ /*|0x8|*/ struct _EX_PUSH_LOCK ProcessLock;
  /*0x2e0*/ /*|0x8|*/ struct _EX_RUNDOWN_REF RundownProtect;
  ...
  /*0x420*/ /*|0x8|*/ void* DebugPort;
  /*0x428*/ /*|0x8|*/ struct _EWOW64PROCESS* WoW64Process;           // Remember this
  /*0x430*/ /*|0x8|*/ void* DeviceMap;
  ..
  /*0x778*/ /*|0x8|*/ unsigned __int64* AllowedCpuSetsIndirect;
  /*0x780*/ /*|0x8|*/ unsigned __int64* DefaultCpuSetsIndirect;
};
// size 0x788
```

If you didn't catch this. For WOW64 process callback address resides (somewhere) in user-mode memory. Oh my…
Common stuff continues with releasing rundown protection, dereferencing target process object, returning status of the operation and so on.

Pseudo code

Thereby, on 64-bit Windows 10586 callback set up looks like this.

But… how callback actually works?

Short answer

Trap frame.

Long answer

Trap frame.

Theory

Let’s set up a breakpoint at nt!IoCreateFileEx and run a stack back trace after.

```
0: kd> bp nt!IoCreateFileEx

0: kd> kv
 # Child-SP          RetAddr           : Call Site
00 ffffd000`32f3ba38 fffff800`4c894932 : nt!IoCreateFileEx
01 ffffd000`32f3ba40 fffff800`4c898466 : FLTMGR!FltpExpandFilePathWorker+0x2a2
...
16 ffffd000`32f3c410 fffff801`0fa6e328 : FLTMGR!FltpCreate+0x333
17 ffffd000`32f3c4c0 fffff801`0fa64c96 : nt!IopParseDevice+0x7c8
18 ffffd000`32f3c690 fffff801`0fa6369c : nt!ObpLookupObjectName+0x776
19 ffffd000`32f3c830 fffff801`0fa8c8c8 : nt!ObOpenObjectByNameEx+0x1ec
1a ffffd000`32f3c950 fffff801`0fa8c429 : nt!IopCreateFile+0x3d8
1b ffffd000`32f3ca00 fffff801`0f7c6ca3 : nt!NtCreateFile+0x79
1c ffffd000`32f3ca90 00007ffd`c96b57f4 : nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @
ffffd000`32f3cb00)  <-- Trap frame created on syscall
1d 000000e4`d2ffdeb8 00007ffd`c5f5a484 : ntdll!NtCreateFile+0x14
1e 000000e4`d2ffdec0 00007ffd`c5f5a156 : KERNELBASE!CreateFileInternal+0x314
1f 000000e4`d2ffe040 00007ffd`c34b460a : KERNELBASE!CreateFileW+0x66
...

0: kd> dt nt!_KTRAP_FRAME ffffd000`32f3cb00
   +0x000 P1Home           : 0xffffe001`52a70840
   +0x008 P2Home           : 0x00000206`8d488b00
   +0x010 P3Home           : 0xffffe001`52a70840
   +0x018 P4Home           : 0xffffd000`32f3cb80
   +0x020 P5               : 0
   +0x028 PreviousMode     : 0 ''
   +0x029 PreviousIrql     : 0 ''
   +0x02a FaultIndicator   : 0x1 ''
   +0x02b ExceptionActive  : 0x2 ''
   +0x02c MxCsr            : 0x1fa0
   +0x030 Rax              : 0x6583
   +0x038 Rcx              : 0x880
   +0x040 Rdx              : 0
   +0x048 R8               : 0x00000206`8d45703e
   +0x050 R9               : 0x00000206`8b0a0000
   +0x058 R10              : 0
   +0x060 R11              : 0x246
   +0x068 GsBase           : 0x000000e4`cf77a000
   +0x068 GsSwap           : 0x000000e4`cf77a000
   +0x070 Xmm0             : _M128A
   +0x080 Xmm1             : _M128A
   +0x090 Xmm2             : _M128A
   +0x0a0 Xmm3             : _M128A
   +0x0b0 Xmm4             : _M128A
   +0x0c0 Xmm5             : _M128A
   +0x0d0 FaultAddress     : 0x00000206`8d45bccf
   +0x0d0 ContextRecord    : 0x00000206`8d45bccf
   +0x0d0 TimeStampCKCL    : 0x00000206`8d45bccf
   +0x0d8 Dr0              : 0
   +0x0e0 Dr1              : 0
   +0x0e8 Dr2              : 0
```

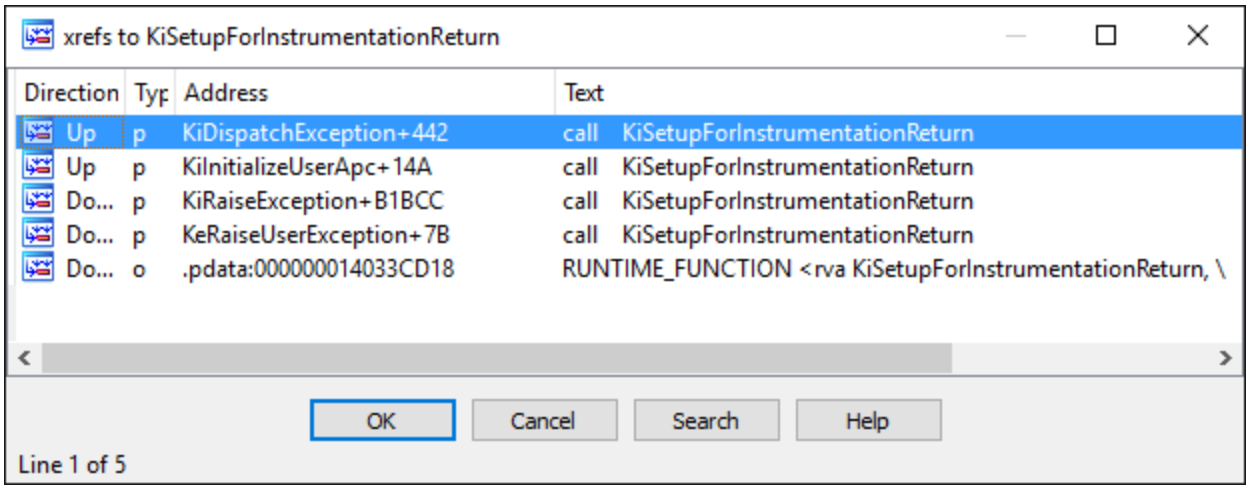
```
   +0x0f0 Dr3             : 0
   +0x0f8 Dr6             : 0
   +0x100 Dr7             : 0
   +0x108 DebugControl    : 0
   +0x110 LastBranchToRip  : 0
   +0x118 LastBranchFromRip : 0
   +0x120 LastExceptionToRip : 0
   +0x128 LastExceptionFromRip : 0
   +0x130 SegDs           : 0
   +0x132 SegEs           : 0
   +0x134 SegFs           : 0
   +0x136 SegGs           : 0
   +0x138 TrapFrame       : 0
   +0x140 Rbx             : 0x00007ffd`c5fb2750
   +0x148 Rdi             : 0
   +0x150 Rsi             : 0x80
   +0x158 Rbp             : 0x000000e4`d2ffdfc0
   +0x160 ErrorCode       : 6
   +0x160 ExceptionFrame  : 6
   +0x160 TimeStampKlog   : 6
   +0x168 Rip             : 0x00007ffd`c96b57f4              <-- Return address
   +0x170 SegCs           : 0x33                             <-- SegCs
   +0x172 Fill0           : 0 ''
   +0x173 Logging         : 0 ''
   +0x174 Fill1           : [2] 0
   +0x178 EFlags          : 0x246
   +0x17c Fill2           : 0
   +0x180 Rsp             : 0x000000e4`d2ffdeb8
   +0x188 SegSs           : 0x2b
   +0x18a Fill3           : 0
   +0x18c Fill4           : 0

0: kd> ln 0x00007ffd`c96b57f4
(00007ffd`c96b57e0)   ntdll!NtCreateFile+0x14   |  (00007ffd`c96b5800)
ntdll!NtQueryEvent

0: kd> u ntdll!NtCreateFile
ntdll!NtCreateFile:
00007ffd`c96b57e0 4c8bd1          mov     r10,rcx
00007ffd`c96b57e3 b855000000      mov     eax,55h
00007ffd`c96b57e8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308
(00000000`7ffe0308)],1
00007ffd`c96b57f0 7503            jne     ntdll!NtCreateFile+0x15
(00007ffd`c96b57f5)
00007ffd`c96b57f2 0f05            syscall
00007ffd`c96b57f4 c3              ret                        <-- After syscall we
should return here
```

I think this self-explained (or just read Intel's manual, google, whatever).

Native process. Practice

Let’s prove a theory for any native process. Choose any native process you want and edit a callback pointer manually in debugger. Don’t forget to set up a hardware breakpoint on read. It may break a few times on thread(s) insertion, but finally it will break on nt!KiSetupForInstrumentationReturn.

```
0: kd> dt nt!_KPROCESS ffffe00156cf9080
   +0x000 Header           : _DISPATCHER_HEADER
   +0x018 ProfileListHead  : _LIST_ENTRY [ 0xffffe001`56cf9098 - 0xffffe001`56cf9098
]
   +0x028 DirectoryTableBase : 0x00000001`01086000
   +0x030 ThreadListHead   : _LIST_ENTRY [ 0xffffe001`565a2b38 - 0xffffe001`51902378
]
   ...
   +0x278 LdtSystemDescriptor : _KGDTENTRY64
   +0x288 LdtBaseAddress   : (null)
   +0x290 LdtProcessLock   : _FAST_MUTEX
   +0x2c8 InstrumentationCallback : (null)                     <-- Patch here
   +0x2d0 SecurePid        : 0

0: kd> ep ffffe00156cf9080+2c8 0x0000000000001000

0: kd> dt nt!_KPROCESS ffffe00156cf9080
   +0x000 Header           : _DISPATCHER_HEADER
   +0x018 ProfileListHead  : _LIST_ENTRY [ 0xffffe001`56cf9098 - 0xffffe001`56cf9098
]
   +0x028 DirectoryTableBase : 0x00000001`01086000
   +0x030 ThreadListHead   : _LIST_ENTRY [ 0xffffe001`565a2b38 - 0xffffe001`51902378
]
   ...
   +0x278 LdtSystemDescriptor : _KGDTENTRY64
   +0x288 LdtBaseAddress   : (null)
   +0x290 LdtProcessLock   : _FAST_MUTEX
   +0x2c8 InstrumentationCallback : 0x00000000`00001000 Void    <-- Patched address
   +0x2d0 SecurePid        : 0

1: kd> ba r8 ffffe00156cf9080+2c8                               <-- Set hardware
breakpoint

0: kd> g
Breakpoint 0 hit
nt!KiSetupForInstrumentationReturn+0x17:
fffff801`0f7599b3 4d85c0          test    r8,r8                 <-- Gotcha!
```



Sorry guys, assembly code this time.

Obviously this routine does one simple thing: replaces current RIP with the callback address and saves the old one in R10. That's it. That's how it works for native process.

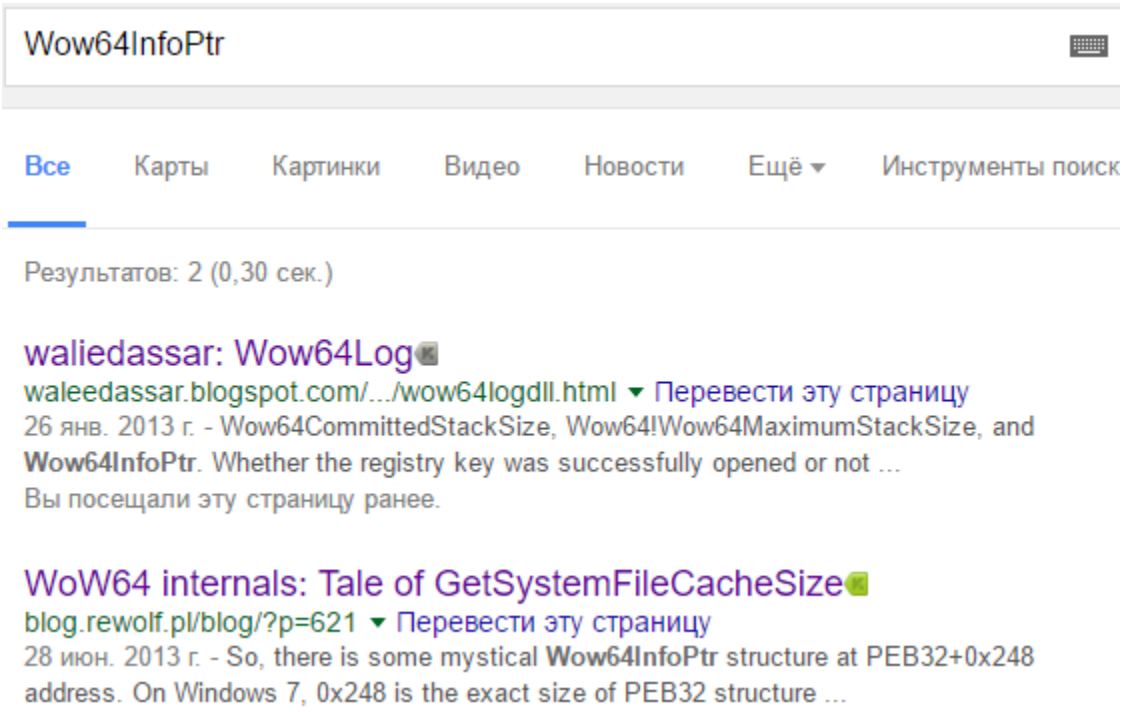
WOW64 process. Practice

Let's do the same for 32-bit (WOW64) process now. Don't forget to change process context.

```
0: kd> dt nt!_EPROCESS ffffe001519ba840
   +0x000 Pcb              : _KPROCESS
   +0x2d8 ProcessLock      : _EX_PUSH_LOCK
   +0x2e0 RundownProtect   : _EX_RUNDOWN_REF
   ...
   +0x428 WoW64Process     : 0xffffe001`53610f10 _EWOW64PROCESS     <-- Did you
remember this?
   +0x430 DeviceMap        : 0xffffc001`96b8cb10 Void
   +0x438 EtwDataSource    : 0xffffe001`52ae2dd1 Void
   ...
   +0x778 AllowedCpuSets   : 0
   +0x780 DefaultCpuSets   : 0
   +0x778 AllowedCpuSetsIndirect : (null)
   +0x780 DefaultCpuSetsIndirect : (null)

0: kd> dt nt!_EWOW64PROCESS 0xffffe001`53610f10
   +0x000 Peb              : 0x00000000`002b3000 Void              <-- Pointer to
nt!_PEB32
   +0x008 Machine          : 0x14c

0: kd> .process /r/p ffffe001519ba840                              <-- Set process
context
Implicit process is now ffffe001`519ba840
.cache forcedecodeuser done
Loading User Symbols
.....

0: kd> dd 0x00000000`002b3000+46c L1                               <-- mov
[rcx+46Ch], eax
00000000`002b346c  00000000

0: kd> ed 0x00000000`002b3000+46c 0x1000                           <-- Patch

0: kd> dd 0x00000000`002b3000+46c L1                               <-- Check
00000000`002b346c  00001000

0: kd> ba r4 /p 0xffffe001519ba840 0x00000000`002b3000+46c         <-- Set
breakpoint on read/execute

0: kd> g
Breakpoint 0 hit
wow64!Wow64SystemServiceEx+0x228:
0033:00000000`71236318 85c9            test    ecx,ecx             <-- Gotcha!

0: kd> u @eip-10
wow64!Wow64SystemServiceEx+0x218:
00000000`71236308 8981e0140000    mov     dword ptr [rcx+14E0h],eax
00000000`7123630e 488b059b0b0400  mov     rax,qword ptr [wow64!Wow64InfoPtr
(00000000`71276eb0)]           <-- WTF is this?
00000000`71236315 8b480c          mov     ecx,dword ptr [rax+0Ch]
<-- 0x460 (size of nt!_PEB32) + 0xC. Hmm!
00000000`71236318 85c9            test    ecx,ecx
```

```
<-- We're here
00000000`7123631a 0f85464c0100    jne     wow64!Wow64KiUserCallbackDispatcher+0x2756
(00000000`7124af66)    <-- Let's see...
```

As you can see, callback pointer resides inside a structure pointed by wow64!Wow64InfoPtr. If you'll google for it, you'll find... almost nothing.



Btw, I suggest you to read ReWolf's «WoW64 internals: Tale of GetSystemFileCacheSize» blogpost.

I've RE Wow64InfoPtr structure, here how it looks like:

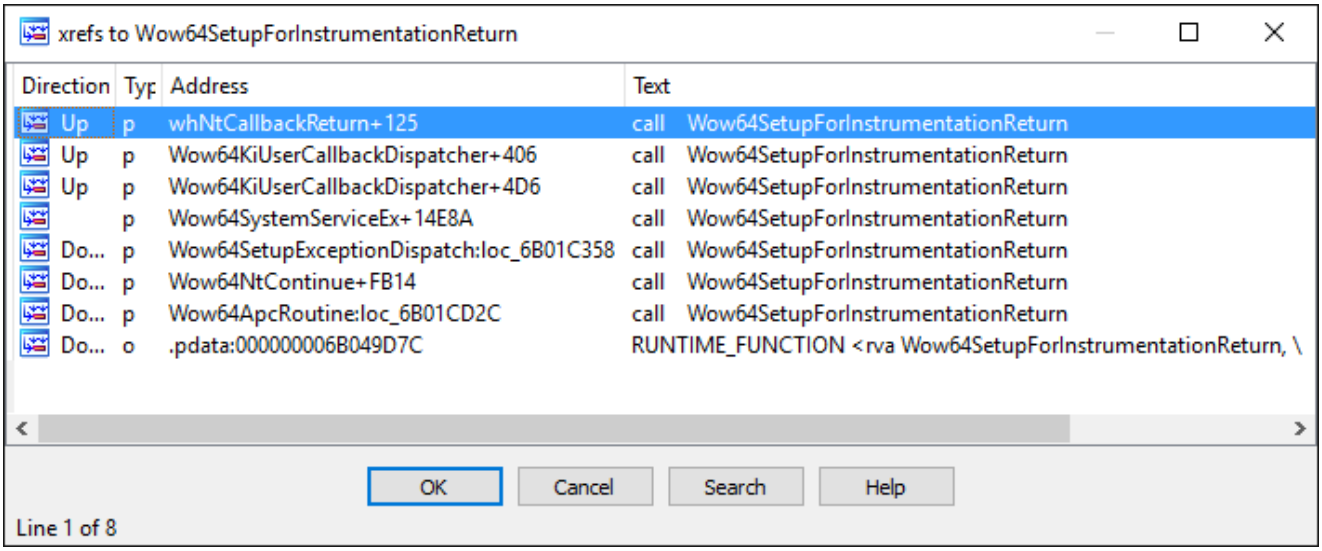
```
struct _WOW64_INFO {
    ULONG32 PageSize;
    ULONG32 Wow64ExecuteFlags;
    ULONG32 Unknown;
    ULONG32
InstrumentationCallback;
} WOW64_INFO, *PWOW64_INFO;
```

Let's return to our muttons.

```
0: kd> u 00000000`7124af66
wow64!Wow64KiUserCallbackDispatcher+0x2756:
00000000`7124af66 8b442440        mov     eax,dword ptr [rsp+40h]
00000000`7124af6a a803            test    al,3
00000000`7124af6c 0f85aeb3feff    jne     wow64!Wow64SystemServiceEx+0x230
(00000000`71236320)
00000000`7124af72 a804            test    al,4
00000000`7124af74 0f85a6b3feff    jne     wow64!Wow64SystemServiceEx+0x230
(00000000`71236320)
00000000`7124af7a e821430000      call    wow64!Wow64SetupForInstrumentationReturn
(00000000`7124f2a0)      <-- Instrumentation!
00000000`7124af7f 90              nop
00000000`7124af80 e99bb3feff      jmp     wow64!Wow64SystemServiceEx+0x230
(00000000`71236320)
```

All the magic occurs in wow64!Wow64SetupForInstrumentationReturn. Function's prototype:

```
__int64 __fastcall Wow64SetupForInstrumentationReturn(unsigned int
Address);
```



To be honest, I have no idea what the heck is going inside Wow64SetupForInstrumentationReturn. I'm OK with assumed step-by-step explanation:

1. Gets current CPU area.
2. Gets CPU context.
3. Sets CPU context.
4. Calls CpuSetInstructionPointer passing callback address and some (bValidate?) bool variable.
    1. If bValidate is true, then calls Wow64ValidateUserCallTarget on passed address.
    2. Gets current CPU area again.

3. Sets callback address inside CPU area and returns.

Interesting value 0x1C4 == IMAGE_FILE_MACHINE_ARMNT (just near of 0x14C == IMAGE_FILE_MACHINE_I386) has been found inside CpuSetInstructionPointer. ARM instrumenting?

I know, a little bit confusing, sorry. Anyway, that's how it works for WOW64 process.

Bonus

Windows 10 Redstone 1

Yeah, Hooking Nirvana is present on Windows 10 RS1. BinDiff shows no differences, but I'm not dig too deep.

WinDBG Anti-RootKit extension

I've added a Hooking Nirvana detector to the WDBGARK extension for the Windows Debugger. Detector should work for any supported Windows version, even for the Windows 10 RS1 x86/x64. It's still a little bit buggy, so, please, build it yourself.

Check this out!

```
1: kd> .load wdbgark
1: kd> !wa_colorize /enable
Connected to Windows 10 10586 x64 target at (Sun Apr 10 11:24:48.219 2016 (UTC + 3:00)), ptr64 TRUE
Loading Kernel Symbols
...............................................................
...

Press ctrl-c (cdb, kd, ntsd) or ctrl-break (windbg) to abort symbol loads that take too long.
Run !sym noisy before .reload to track down problems loading symbols.

.............................................................
......................................
Loading User Symbols

Loading unloaded module list
.......Unable to enumerate user-mode unloaded modules, Win32 error 0n30
*** WARNING: Unable to verify timestamp for dummypdb_10736
[+] Colorizing enabled
1: kd> !wa_chknirvana
[+] Searching for Hooking Nirvana instrumentation
+------------------------------------------------------------------------------------------------------------------------+
|           Address|                    Type|                                        Symbol|      Module|Suspicious|              Info|
+------------------------------------------------------------------------------------------------------------------------+
|0x00007ffdb8f71450|                 Nirvana|                          instrument!Dummy+0x70|  instrument|          |  dx .process u50|
+------------------------------------------------------------------------------------------------------------------------+
|0x00000000742913e0|                 Nirvana|                                    *UNKNOWN*|   *UNKNOWN*|         Y|  dx .process u50|
+------------------------------------------------------------------------------------------------------------------------+
+------------------------------------------------------------------------------------------------------------------------+
```

Side effect

What a beautiful undocumented feature, yeah? A sweet side effect has been found – process termination. You just need to open any process with PROCESS_SET_INFORMATION and set up a Nirvana Hooking callback to some nonexistent or simply invalid address. Of course, you still need to have a debug privilege enabled in your process. By the way, you can't kill PPL process with this method. And some AV services too :)

Conclusion

Conclusion? What kind of conclusion do you want? Okay, Microsoft is doing a lot of undocumented stuff for themselves only. They adhere to the old rule – security through obscurity, it's an outdated trend I think. That's all I wanna say. Oh, btw, Microsoft, don't use user-mode pointers in a such things. This smells like a teen spirit.