WOW64!Hooks: WOW64 Subsystem Internals and Hooking Techniques

mandiant.com/resources/wow64-subsystem-internals-and-hooking-techniques

Microsoft is known for their backwards compatibility. When they rolled out the 64-bit variant of Windows years ago they needed to provide compatibility with existing 32-bit applications. In order to provide seamless execution regardless of application bitness, the WoW (Windows on Windows) system was coined. This layer, which will be referred to as 'WOW64' from here on out, is responsible for translating all Windows API calls from 32-bit userspace to the 64-bit operating system kernel. This blog post is broken up into two sections. First we start by diving deep into the WOW64 system. To do this, we trace a call from 32-bit userspace and follow the steps it takes to finally transition to the kernel. The second part of the post assesses two hooking techniques and their effectiveness. I will cover how this system works, the ways malware abuses it, and detail a mechanism by which all WoW syscalls can be hooked from userspace. Note that all information here is true as of Windows 10, version 2004 and in some cases *has* changed from how previous Windows versions were implemented.

Recognition

First and foremost, this is a topic which has existing research by multiple authors. This work was critical in efficient exploration of the internals and research would have taken much longer had these authors not publicly posted their awesome work. I would like to callout the following references:

- (wbenny): An extremely detailed view of WOW64 internals on ARM
- (ReWolf): A PoC heaven's gate implementation
- (JustasMasiulis): A very clean C++ heaven's gate implementation
- (MalwareTech): A WOW64 segmentation explanation

WOW64 Internals

To understand how the WOW64 system works internally we will explore the call sequence starting in 32-bit usermode before transitioning into the kernel from within a system DLL. Within these system DLLs the operating system will check arguments and eventually transition to a stub known as a syscall stub. This syscall stub is responsible for servicing the API call in the kernel. On a 64-bit system, the syscall stub is straightforward as it directly executes the syscall instruction as shown in Figure 1.

```
0:000> u ntdll!NtResumeThread
ntdll!NtResumeThread:
775c6970 b852000700        mov      eax,70052h
775c6975 ba10585e77        mov      edx,offset ntdll!Wow64SystemServiceCall (775e5810)
775c697a ffd2              call     edx
775c697c c20800            ret      8
775c697f 90                nop
```

Figure 1: Native x64 Syscall Stub

Figure 2 shows a syscall stub for a 32-bit process running on WOW64

```
0:013> u ntdll!NtResumeThread
ntdll!NtResumeThread:
00007ff9`1d72c7f0 4c8bd1              mov      r10,rcx
00007ff9`1d72c7f3 b852000000          mov      eax,52h
00007ff9`1d72c7f8 f604250803fe7f01    test     byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ff9`1d72c800 7503                jne      ntdll!NtResumeThread+0x15 (00007ff9`1d72c805)
00007ff9`1d72c802 0f05                syscall
00007ff9`1d72c804 c3                  ret
00007ff9`1d72c805 cd2e                int      2Eh
00007ff9`1d72c807 c3                  ret
```

Figure 2: WOW64 Syscall Stub

Notice that instead of a syscall instruction in the WOW64 version, Wow64SystemServiceCall is called. In the WOW64 system what would normally be an entry into the kernel is instead replaced by a call to a usermode routine. Following this Wow64SystemServiceCall, we can see in Figure 3 that it immediately performs an indirect jmp through a pointer named Wow64Transition.

```
ntdll_77550000!Wow64SystemServiceCall:
775e5810 ff2528c26777    jmp     dword ptr [ntdll_77550000!Wow64Transition (7767c228)]
```

Figure 3: Wow64SystemService transitions through a pointer 'Wow64Transition'

Note that the Wow64SystemServiceCall function is found within ntdll labeled as ntdll_77550000; a WOW64 process has two ntdll modules loaded, a 32-bit one and a 64-bit one. WinDbg differentiates between these two by placing the address of the module after the 32-bit variant. The 64-bit ntdll can be found in %WINDIR%\System32 and the 32-bit in %WINDIR%\SysWOW64. In the PDBs, the 64bit and 32bit ntdlls are referred to as ntdll.pdb and wntdll.pdb respectively, try loading them in a disassembler! Continuing with the call trace, if we look at what the Wow64Transition pointer holds we can see its destination is wow64cpu!KiFastSystemCall. As an aside, note that the address of wow64cpu!KiFastSystemCall is held in the 32-bit TEB (Thread Environment Block) via member WOW32Reserved, this isn't relevant for this trace but is useful to know. In Figure 4 we see the body of KiFastSystemCall.

```
wow64cpu!KiFastSystemCall:
77546000 ea0960547733 00    jmp    0033:77546009
```

Figure 4: KiFastSystemCall transitions to x64 mode via segment selector 0x33

The KiFastSystemCall performs a jmp using the 0x33 segment selector to a memory location just after the instruction. This 0x33 segment transitions the CPU into 64-bit mode via a GDT entry as described by (MalwareTech).

Let's recap the trace we've performed to this point. We started from a call in ntdll, NtResumeThread. This function calls the Wow64SystemServiceCall function which then executes the Wow64Transition. The KiFastSystemCall performs the transition from 32-bit to 64-bit execution. The flow is shown in Figure 5.
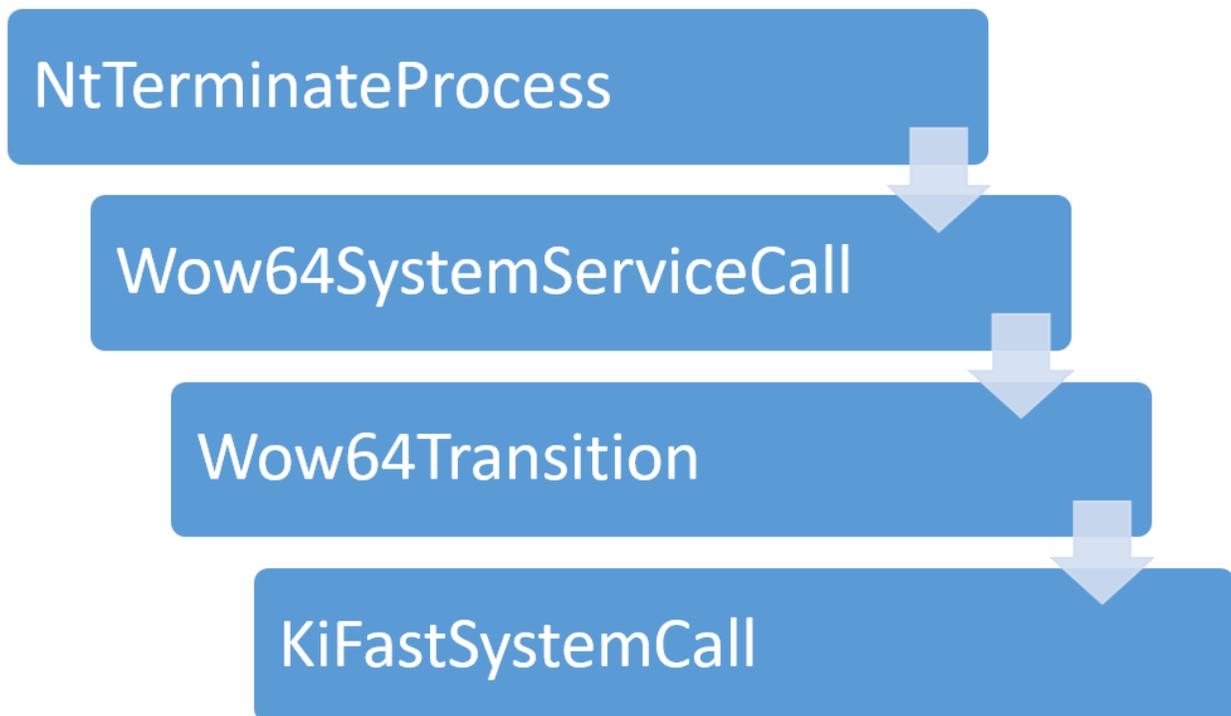

Figure 5: 32-bit to 64-bit transition

The destination of the CPU transition jump is the 64-bit code show in Figure 6.


```
jmp    qword ptr [r15+0F8h]
```

Figure 6: Destination of KiFastSystemCall

Figure 6 shows the first 64-bit instruction we've seen executed in this call trace so far. In order to understand it, we need to look at how the WOW64 system initializes itself. For a detailed explanation of this refer to (wbenny). For now, we can look at the important parts in wow64cpu!RunSimulatedCode.

```
push    r15
push    r14
push    r13
push    r12
push    rbx
push    rsi
push    rdi
push    rbp
sub     rsp, 68h
mov     r12, gs:30h        ; r12 = TEB
lea     r15, TurboThunkDispatch
mov     r13, [r12+(_TEB64.TlsSlots+8)] ; WOW64_CPURESERVED
add     r13, 80h ; '€'
```

Figure 7: 64bit registers are saved in RunSimulatedCode

Figure 7 depicts the retrieval of the 64-bit TEB which is used to access Thread Local Storage at slot index 1. Then the moving of a function pointer table into register r15. The TLS data retrieved is an undocumented data structure WOW64_CPURESERVED that contains register data and CPU state information used by the WOW64 layer to set and restore registers across the 32-bit and 64-bit boundaries. Within this structure is the WOW64_CONTEXT structure, partially documented on the Microsoft website. I have listed both structures at the end of this post. We'll look at how this context structure is used later, but for our understanding of the jmp earlier all we need to know is that r15 is a function pointer table.

It's interesting to note at this point the architecture of the WOW64 layer. From the perspective of the 64-bit kernel the execution of 32-bit (Wow64) usermode applications is essentially a big while loop. The loop executes x86 instructions in the processor's 32-bit execution mode and occasionally exits the loop to service a system call. Because the kernel is 64-bit, the processor mode is temporarily switched to 64-bit, the system call serviced, then the mode switched back and the loop continued where it was paused. One could say the WOW64 layer acts like an emulator where the instructions are instead executed on the physical CPU.

Going back to the jmp instruction we saw in Figure 6, we now know what is occurring. The instruction jmp [r15 + 0xF8] is equivalent to the C code jmp TurboThunkDispatch[0xF8 / sizeof(uint64_t)]. Looking at the function pointer at this index we can see we're at the function wow64cpu!CpupReturnFromSimulatedCode (Figure 8).

```
CpupReturnFromSimulatedCode: ;
xchg    rsp, r14              ; r14 = pStack
mov     r8d, [r14]            ; r8d = retAddr
add     r14, 4               ; r14d = pCallerRetAddr
mov     [r13+(WOW64_CPURESERVED.Context._Eip-80h)], r8d
mov     [r13+(WOW64_CPURESERVED.Context._Esp-80h)], r14d
lea     r11, [r14+4]     ; r11 = pArgs
mov     [r13+(WOW64_CPURESERVED.Context._Edi-80h)], edi
mov     [r13+(WOW64_CPURESERVED.Context._Esi-80h)], esi
mov     [r13+(WOW64_CPURESERVED.Context._Ebx-80h)], ebx
mov     [r13+(WOW64_CPURESERVED.Context._Ebp-80h)], ebp
pushfq
pop     r8
mov     [r13+(WOW64_CPURESERVED.Context.EFlags-80h)], r8d
```

Figure 8: TurboThunk table's last function pointer entry is an exit routine

This routine is responsible for saving the state of the 32-bit registers into the WOW64_CONTEXT structure we mentioned before as well as retrieving the arguments for the syscall. There is some trickiness going on here, so let's examine this in detail. First a pointer to the stack is moved into r14 via xchg, the value at this location will be the return address from the syscall stub where Wow64SystemServiceCall was called. The stack pointer r14 is then incremented by 4 to get a pointer to where the stack should be reset when it's time to restore all these context values. These two values are then stored in the context's EIP and ESP variables respectively. The r14 stack pointer is then incremented one more time to get the location where the __stdcall arguments are (remember stdcall passes all arguments on the stack). This argument array is important for later, remember it. The arguments pointer is moved into r11, so in C this means that r11 is equivalent to an array of stack slots where each slot is an argument uint32_t r11[argCount]. The rest of the registers and EFlags are then saved.

Once the 32-bit context is saved, the WOW64 layer then calculates the appropriate TurboThunk to invoke by grabbing the upper 16 bits of the syscall number and dispatches to that thunk. Note that at the beginning of this array is the function TurboDispatchJumpAddressEnd, shown in Figure 9, which is invoked for functions that do not support TurboThunks.

```
TurboDispatchJumpAddressStart:
mov     ecx, eax
shr     ecx, 10h         ; rcx = TurboThunkIdx
jmp     qword ptr [r15+rcx*8] ; r15 = pTurboThunkTable OR -> TurboDispatchJumpAddressEnd
```

Figure 9: TurboThunk table's first function pointer entry is an entry routine

TurboThunks are described by (wbenny)—read his blog post at this point if you have not. To summarize the post, for functions that have simple arguments with widths <= sizeof(uint32_t) the WOW64 layer will directly widen these arguments to 64 bits via zero or sign-extension and then perform a direct syscall into the kernel. This all occurs within wow64cpu, rather than executing a more complex path detailed as follows. This acts as an optimization. For more complex functions that do not support TurboThunks the TurboDispatchJumpAddressEnd stub is used which dispatches to wow64!SystemServiceEx to perform the system call as shown in Figure 10.

```
TurboDispatchJumpAddressEnd: ; syscallNum
mov     ecx, eax
mov     rdx, r11          ; ArgsArray
call    cs:__imp_Wow64SystemServiceEx
mov     [r13+(WOW64_CPURESERVED.Context._Eax-80h)], eax
jmp     RestoreState
```

Figure 10: Complex system calls go through Wow64SystemServiceEx

We'll look at this routine in a moment as it's the meat of this blog post, but for now let's finish this call trace. Once Wow64SystemServiceEx returns from doing the system call the return value in eax is moved into the WOW64_CONTEXT structure and then the 32-bit register states are restored. There's two paths for this, a common case and a case that appears to exist only to be used by NtContinue and other WOW64 internals. A flag at the start of the WOW64_CPURESERVED structure retrieved from the TLS slot is checked, and controls which restore path to follow as shown in Figure 11.

```
RestoreState:
btr     dword ptr [r13+(WOW64_CPURESERVED.Flags-80h)], 0
jb      short DoReservedCPUReturn ; WOW64_CPURESERVED_FLAG_RESET_STATE
```

```
mov     edi, [r13+(WOW64_CPURESERVED.Context._Edi-80h)]
mov     esi, [r13+(WOW64_CPURESERVED.Context._Esi-80h)]
mov     ebx, [r13+(WOW64_CPURESERVED.Context._Ebx-80h)]
mov     ebp, [r13+(WOW64_CPURESERVED.Context._Ebp-80h)]
mov     eax, [r13+(WOW64_CPURESERVED.Context._Eax-80h)]
mov     r14, rsp
mov     dword ptr [rsp+0A8h+var_A8+4], 23h ; '#'
mov     r8d, 2Bh ; '+'
mov     ss, r8d
mov     r9d, [r13+(WOW64_CPURESERVED.Context._Eip-80h)]
mov     dword ptr [rsp+0A8h+var_A8], r9d
mov     esp, [r13+(WOW64_CPURESERVED.Context._Esp-80h)]
jmp     fword ptr [r14]
```

```
DoReservedCPUReturn:
mov     ecx, 2Bh ; '+'
mov     ds, ecx
assume ds:nothing
mov     es, ecx
assume es:nothing
mov     gs, ecx
assume gs:nothing
mov     r8, cs:CpupUserSharedData
test    byte ptr [r8+28Ah], 0FFh ; is64BitMode
jnz     short setFS64bit
```

```
mov     r8d, 53h ; 'S'  ; setFS32Bit
mov     fs, r8d
jmp     short loc_776C1743
```

```
setFS64bit:
mov     r8, gs:0
wrfsbase r8
```

```
loc_776C1743:
movaps  xmm0, xmmword ptr [r13+(WOW64_CPURESERVED.Context._Xmm0.Low-80h)]
movaps  xmm1, xmmword ptr [r13+(WOW64_CPURESERVED.Context._Xmm1.Low-80h)]
movaps  xmm2, xmmword ptr [r13+(WOW64_CPURESERVED.Context._Xmm2.Low-80h)]
movaps  xmm3, xmmword ptr [r13+(WOW64_CPURESERVED.Context._Xmm3.Low-80h)]
movaps  xmm4, xmmword ptr [r13+(WOW64_CPURESERVED.Context._Xmm4.Low-80h)]
movaps  xmm5, xmmword ptr [r13+(WOW64_CPURESERVED.Context._Xmm5.Low-80h)]
mov     ecx, [r13+(WOW64_CPURESERVED.Context._Ecx-80h)]
mov     edx, [r13+(WOW64_CPURESERVED.Context._Edx-80h)]
and     dword ptr [r13+(WOW64_CPURESERVED.Flags-80h)], 0FFFFFFFFh
mov     edi, [r13+(WOW64_CPURESERVED.Context._Edi-80h)]
mov     esi, [r13+(WOW64_CPURESERVED.Context._Esi-80h)]
mov     ebx, [r13+(WOW64_CPURESERVED.Context._Ebx-80h)]
mov     ebp, [r13+(WOW64_CPURESERVED.Context._Ebp-80h)]
mov     eax, [r13+(WOW64_CPURESERVED.Context._Eax-80h)]
mov     r14, rsp
mov     [rsp+0A8h+var_A0], 23h ; '#'
mov     [rsp+0A8h+var_88], 2Bh ; '+'
mov     r8d, [r13+(WOW64_CPURESERVED.Context.EFlags-80h)]
and     [r13+(WOW64_CPURESERVED.Context.EFlags-80h)], 0FFFFFEFFh
mov     [rsp+0A8h+var_98], r8d
mov     r8d, [r13+(WOW64_CPURESERVED.Context._Esp-80h)]
mov     [rsp+0A8h+var_90], r8
mov     r8d, [r13+(WOW64_CPURESERVED.Context._Eip-80h)]
mov     [rsp+0A8h+var_A8], r8
iretq
```
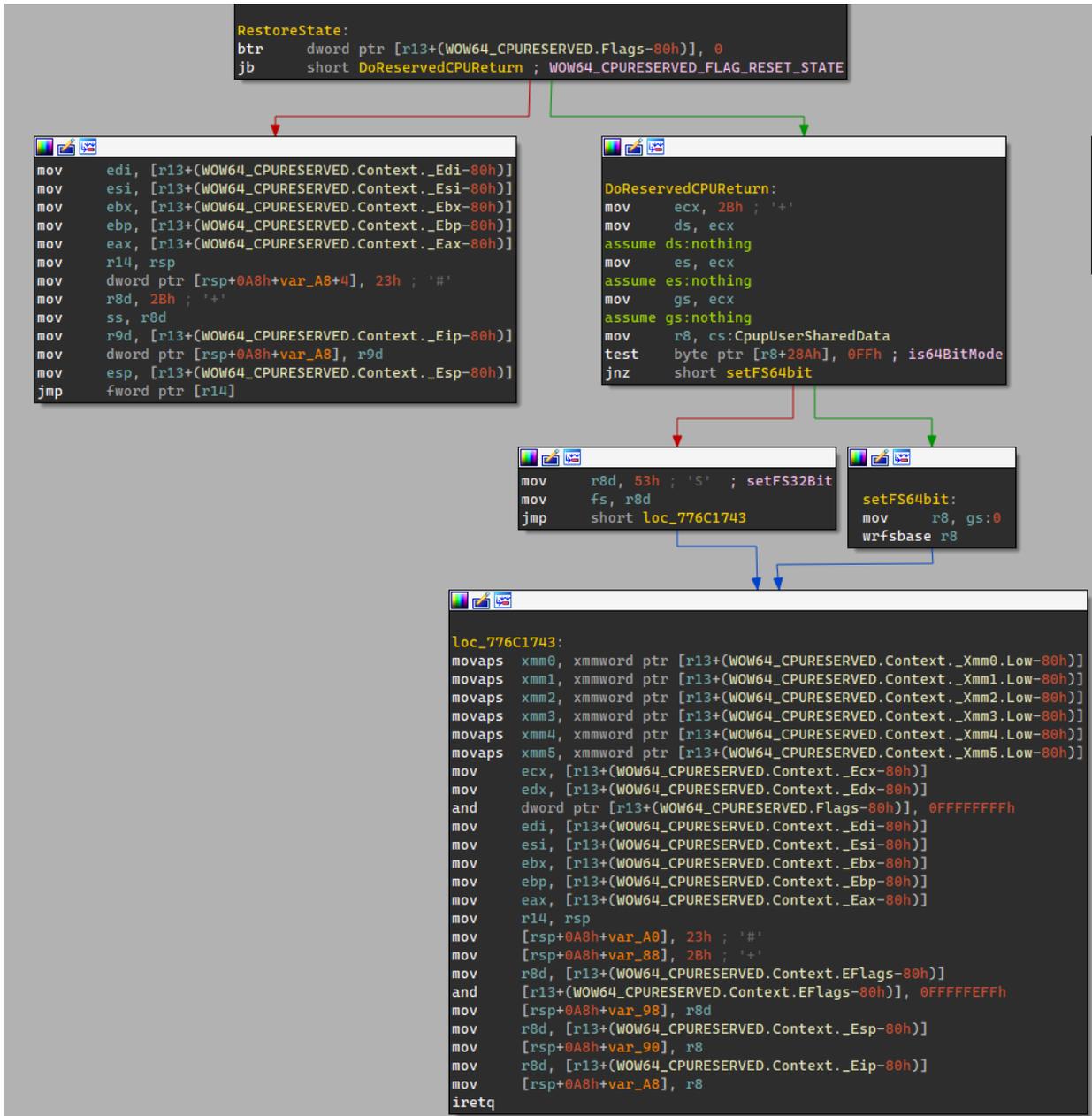
Figure 11: CPU state is restored once the system call is done; there's a simple path and a complex one handling XMM registers

The simpler case will build a jmp that uses the segment selector 0x23 to transition back to 32-bit mode after restoring all the saved registers in the WOW64_CONTEXT. The more complex case will additionally restore some segments, xmm values, and the saved registers in the WOW64_CONTEXT structure and then will do an iret to transition back. The common case jmp once built is shown in Figure 12.


```
00000000`7754170b 41ff2e        jmp     fword ptr [r14] ds:00000000`00a6ef40=0023775c697c
```

Figure 12: Dynamically built jmp to transition back to 32bit mode

At this point our call trace is complete. The WOW64 layer has transitioned back to 32-bit mode and will continue execution at the ret after Wow64SystemServiceCall in the syscall stub we started with. Now that an understanding of the flow of the WOW64 layer itself is understood, let's examine the Wow64SystemServiceEx call we glossed over before.

A little bit into the Wow64SystemServiceEx routine, Figure 13 shows some interesting logic that we will use later.

```
whService = *(__int64 (__fastcall **)(unsigned int *))&ServiceTables[ServiceTable].Descriptor.ServiceTable[2 * syscall];
logInst.ServiceTable = (*(unsigned int *)&syscallNum >> 12) & 3;
logInst.ServiceNumber = *(_WORD *)&syscallNum & 0xFFF;
Teb->LastErrorValue = HIDWORD(Teb_1->NtTib.Self);
if ( Wow64LogSystemService )
{
  logInst.Arguments = ArgsArray;
  logInst.PostCall = 0;
  LogService(&logInst);
  status = whService(ArgsArray);
  logInst.PostCall = 1;
  logInst.Status = status;
  LogService(&logInst);
}
```

Figure 13: Logging routines invoked before and after dispatching the syscalls

The routine starts by indexing into service tables which hold pointers to routines that convert the passed argument array into the wider 64-bit types expected by the regular 64-bit system modules. This argument array is exactly the stack slot that was stored earlier in r14.

Two calls to the LogService function exist, however these are only called if the DLL %WINDIR%\system32\wow64log.dll is loaded and has the exports Wow64LogInitialize, Wow64LogSystemService, Wow64LogMessageArgList, and Wow64LogTerminate. This DLL is not present on Windows by default, but it can be placed there with administrator privileges.

The next section will detail how this logging DLL can be used to hook syscalls that transition through this wow64layer. Because the logging routine LogService is invoked before and after the syscall is serviced we can achieve a standard looking inline hook style callback function capable of inspecting arguments and return values.

Bypassing Inline Hooks

As described in this blog post, Windows provides a way for 32-bit applications to execute 64-bit syscalls on a 64-bit system using the WOW64 layer. However, the segmentation switch we noted earlier can be manually performed, and 64-bit shellcode can be written to setup a syscall. This technique is popularly called "Heaven's Gate". JustasMasiulis' work call_function64 can be used as a reference to see how this may be done in practice (JustasMasiulis). When system calls are performed this way the 32-bit syscall stub that the WOW64 layer uses is completely skipped in the execution chain. This is unfortunate for security products or tracing tools because any inline hooks in-place on these stubs are also

bypassed. Malware authors know this and utilize "Heaven's Gate" as a bypass technique in some cases. Figure 14 and Figure 15 shows the execution flow of a regular syscall stub through the WOW64 layer, and hooked syscall stub where malware utilizes "Heaven's Gate".
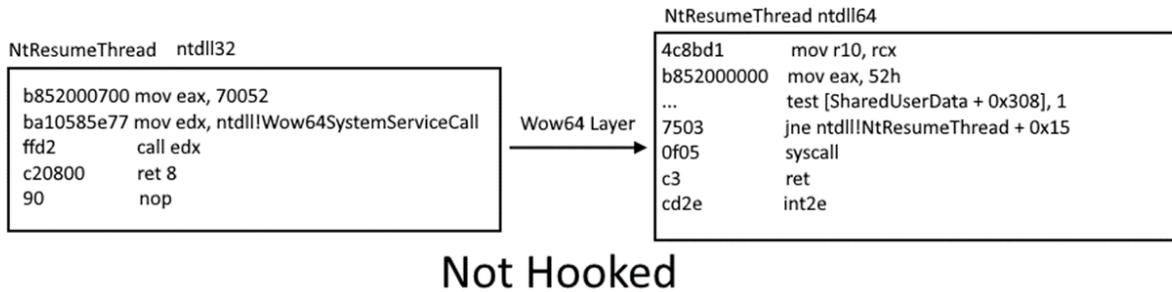

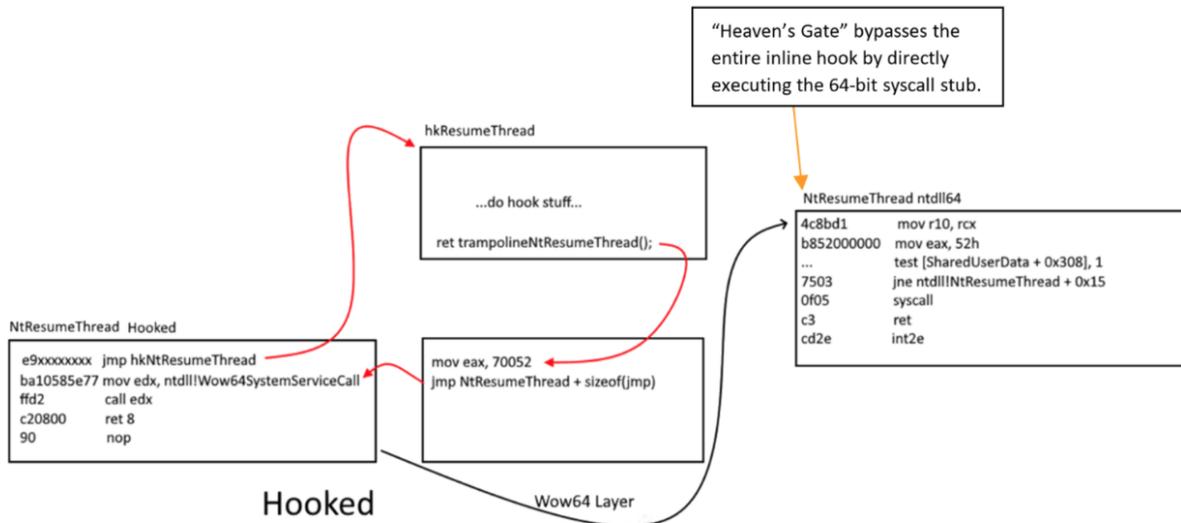Figure 14: NtResumeThread transitioning through the WOW64 layer

Figure 15: NtResumeThread inline hook before transitioning through the WOW64 layer

As seen in Figure 15, when using the Heaven's Gate technique, execution begins *after* the inline hook and WOW64 layer is done. This is an effective bypass technique, but one that is easy to detect from a lower level such as a driver or hypervisor. The easiest bypass to inline hooks is simply to restore the original function bytes, usually from bytes on disk. Malware such as AgentTesla and Conti has been known to utilize this last evasion technique.

Hooking WOW64 via Inline Hooks

As a malware analyst being able to detect when samples attempt to bypass the WOW64 layer can be very useful. The obvious technique to detect this is to place inline hooks on the 64-bit syscall stubs as well as the 32-bit syscall stubs. If the 64-bit hook detects an invocation that didn't also pass through the 32-bit hook, then it's known that a sample is utilizing Heaven's Gate. This technique can detect both evasion techniques previously detailed. However, in practice this is very difficult to implement. Looking at the requirements that must be satisfied to hook the 64-bit syscall stub we come up with this list:

1. Install 64-bit hook from a 32-bit module
    How do you read/write 64-bit address space from a 32-bit module?
2. Implement a 64-bit callback from a 32-bit module
    Typically, inline hooking uses C functions as callback stubs, but we're compiling a
    32-bit module so we'll have a 32-bit callback instead of the required 64-bit one.

To solve the first challenge ntdll kindly provides the exports
NtWow64ReadVirtualMemory64, NtWow64WriteVirtualMemory64, and
NtWow64QueryInformationProcess64. Using these it is possible to read memory, write
memory, and retrieve the PEB of a 64-bit module from a 32-bit process. However, the second
challenge is much harder as either shellcode or a JIT will be required to craft a callback stub
of the right bitness. In practice ASMJIT may be utilized for this. This is however a very
tedious technique to trace a large number of APIs. There are other challenges to this
technique as well. For example, in modern Windows 10 the base address of ntdll64 is set to a
high 64-bit address rather than a lower 32-bit address as in Windows 7. Due to this,
supporting returns from callbacks back up to the original hooked stub and allocating a
trampoline within the required memory range is difficult since the standard ret instruction
doesn't have enough bits on the stack to represent the 64-bit return address.

As an aside, it should be noted that the WOW64 layer contains what is likely a bug when
dealing with the NtWow64* functions. These APIs all take a HANDLE as first argument,
which *should* be sign extended to 64-bits. However, this does not occur for these APIs,
therefore when using the pseudo handle -1 the call fails with STATUS_INVALID_HANDLE.
This bug was introduced in an unknown Windows 10 version. To successfully use these APIs
OpenProcess must be used to retrieve a real, positive valued handle.

I will not be covering the internals of how to inline hook the 64-bit syscall stub since this post
is already very long. Instead I will show how my hooking library PolyHook2 can be extended
to support cross-architecture hooking using these Windows APIs, and leave the rest as an
exercise to the reader. This works because PolyHook's trampolines are not limited to +-2GB
and do not spoil registers. The internals of how *that* is achieved is a topic for another post.
Figure 16 depicts how to overload the C++ API of polyhook to read/write memory using the
aforementioned WinAPIs.

```
3    #include "polyhook2/ZydisDisassembler.hpp"
4    #include "polyhook2/Detour/x64Detour.hpp"
5
6    #include "Wow64.hpp"
7
8    class Wow64Detour : public PLH::x64Detour
9    {
10   public:
11       Wow64Detour(const uint64_t fnAddress, const uint64_t fnCallback, const uint8_t argCount, uint64_t* userTrampVar, PLH::ADisassembler& dis) :
12           x64Detour(fnAddress, fnCallback, userTrampVar, dis) // note callback is rebound later
13       {
14           m_hSelf = OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION | PROCESS_VM_READ | PROCESS_VM_WRITE | PROCESS_VM_OPERATION, FALSE, GetCurrentProcessId());
15           m_hNtDll64 = GetModuleHandle64(L"ntdll.dll");
16           NtProtectVirtualMemory64 = GetProcAddress64(m_hNtDll64, "NtProtectVirtualMemory");
17
18           // generate shellcode which transitions from 64bit land in wow64 layer to 32bit user given callback
19           m_x64x86_callback = Genx64x86Callback(fnCallback, argCount);
20           this->m_fnCallback = m_x64x86_callback;
21       }
22
23       /**
24        * Helper that uses a template to calculate the argCount of the callback automatically
25        **/
26       template<typename Ret, typename... Args>
27       Wow64Detour(const uint64_t fnAddress, Ret(*fnCallback)(Args...), uint64_t* userTrampVar, PLH::ADisassembler& dis) : Wow64Detour(fnAddress, (uint64_t)fnCallback, sizeof...(Args), userTrampVar, dis){}
28
29       virtual ~Wow64Detour()
30       {
31           Freex64x86Callback(m_x64x86_callback);
32       }
33
34       virtual bool mem_copy(uint64_t dest, uint64_t src, uint64_t size) const override {
35           std::byte* tmp = new std::byte[size];
36           ULONGLONG read_written = 0;
37
38           // dest always limited to 32bits, so do read then write seperate
39           NtWow64ReadVirtualMemory64(m_hSelf, (PVOID64)src, (PVOID)tmp, size, &read_written);
40
41           read_written = 0;
42
43           NtWow64WriteVirtualMemory64(m_hSelf, (PVOID64)dest, (PVOID)tmp, size, &read_written);
44           delete[] tmp;
45           return true;
46       }
47
48       virtual bool safe_mem_write(uint64_t dest, uint64_t src, uint64_t size, size_t& written) const noexcept override {
49           written = 0;
50           ULONGLONG written64 = 0;
51           NTSTATUS status = NtWow64WriteVirtualMemory64(m_hSelf, (PVOID64)dest, (PVOID)src, size, (PULONGLONG)&written64);
52           written = (size_t)written64;
53           return status == STATUS_SUCCESS;
54       }
55
56       virtual bool safe_mem_read(uint64_t src, uint64_t dest, uint64_t size, size_t& read) const noexcept override {
57           read = 0;
58           ULONGLONG read64 = 0;
59           NTSTATUS status = NtWow64ReadVirtualMemory64(m_hSelf, (PVOID64)src, (PVOID)dest, size, (PULONGLONG)&read64);
60           read = (size_t)read64;
61           return status == STATUS_SUCCESS || status == STATUS_PARTIAL_COPY;
62       }
63
64       virtual PLH::ProtFlag mem_protect(uint64_t dest, uint64_t size, PLH::ProtFlag prot, bool& status) const override {
65           uint64_t prot_size = size;
66           uint64_t old_prot = 0;
67           uint64_t base = dest;
68           NTSTATUS nt_status = call_function64(NtProtectVirtualMemory64, m_hSelf, &base, &prot_size, TranslateProtection(prot), (uint64_t)&old_prot);
69           status = nt_status == STATUS_SUCCESS;
70           return PLH::TranslateProtection(old_prot);
71       }
72   private:
73       uint64_t m_hNtDll64;
74       uint64_t NtProtectVirtualMemory64;
75       uint64_t m_x64x86_callback;
76
77       // Wow64 layer fails to sign extend pseudo handle -1 to 64-bit. Must use real, postive valued handle
78       HANDLE m_hSelf;
79   };
```

Figure 16: Overloading the memory operations to read/write/protect 64-bit memory

Once these inline hooks are in-place on the 64-bit syscall stubs, any application utilizing Heaven's Gate will be properly intercepted. This hooking technique is very invasive and complicated and can still be bypassed if a sample was to directly execute a syscall instruction rather than using the 64-bit module's syscalls stub. Therefore, a driver or hypervisor is more suitable to detect this evasion technique. Instead we can focus on the more common byte restoration evasion techniques and look for a way to hook the WOW64 layer itself. This doesn't involve assembly modifications at all.

Hooking WOW64 via LogService

Thinking back to the WOW64 layer's execution flow we know that all calls which are sent through the Wow64SystemServiceEx routine may invoke the routine Wow64LogSystemService if the logging DLL is loaded. We can utilize this logging DLL and routine to implement hooks which can be written the exact same way as inline hooks, without modifying any assembly.

The first step to implementing this is to force all API call paths through the Wow64SystemServiceEx routine so that the log routine may be called. Remember earlier that those that support TurboThunks will not take this path. Lucky for us we know that any TurboThunk entry that points to TurboDispatchJumpAddressEnd will take this path. Therefore, by pointing every entry in the TurboThunk table to point at that address, the desired behavior is achieved. Windows kindly implements this patching via wow64cpu!BTCpuTurboThunkControl as shown in Figure 17.

```
ThunkTable = (uint64_t *)&TurboThunkTable;
for ( ThunkIndex = 0; ThunkIndex < 0x20; ++ThunkIndex )
{
  if ( ThunkIndex != 0x1F )
    *ThunkTable = (uint64_t)&TurboDispatchJumpAddressEnd;
  ++ThunkTable;
}
```

Figure 17: Patching the TurboThunk table is implemented for us

Note that in previous Windows versions the module which exported this and how it did is different to Windows 10, version 2004. After invoking this patch routine all syscall paths through WOW64 go through Wow64SystemServiceEx and we can focus on crafting a logging DLL that man-in-the-middles (MITMs) all calls. There are a couple of challenges to be considered here:

1. How do we determine which system call is currently occurring from the logging DLL?
2. How are callbacks written? Wow64log is 64-bit DLL, we'd like a 32-bit callback. Is shellcode required, or can we make nice C style function callbacks?
3. What APIs may we call? All that's loaded is 64-bit ntdll.

The first concern is rather easy, from within the wow64log DLL we can read the syscall number from the syscall stubs to create a map of number to name. This is possible because syscall stubs always start with the same assembly and the syscall number is at a static offset of 0x4. Figure 18 shows how we can then compare the values in this map against the syscall number passed to Wow64LogSystemService's parameter structure WOW64_LOG_SERVICE.

```c
typedef uint32_t* WOW64_ARGUMENTS;
struct WOW64_LOG_SERVICE
{
    uint64_t BtLdrEntry;
    WOW64_ARGUMENTS Arguments;
    ULONG ServiceTable;
    ULONG ServiceNumber;
    NTSTATUS Status;
    BOOLEAN PostCall;
};

EXTERN_C
__declspec(dllexport)
NTSTATUS
NTAPI
Wow64LogSystemService(WOW64_LOG_SERVICE* service)
{
    for (uint32_t i = 0; i < LAST_SYSCALL_ID; i++) {
        const char* sysname = SysCallMap[i].name;
        uint32_t syscallNum = SysCallMap[i].SystemCallNumber;
        if (ServiceParameters->ServiceNumber != syscallNum)
            continue;
        //LOG sysname
    }
}
```

Figure 18: Minimal example of determining which syscall is occurring—in practice the service table must be checked too

Writing callbacks is a bit more challenging. The wow64log DLL is executing in 64-bit mode and we'd like to be able to write callbacks in 32-bit mode since it's very easy to load additional 32-bit modules into a WOW64 process. The best way to handle this is to write shellcode which is capable of transitioning back to 32-bit mode, execute the callback, then go back to 64-bit mode to continue execution in the wow64log DLL. The segment transitions themselves are rather easy at this point, we know we just need to use 0x23 or 0x33 segment selectors when jumping. But we also need to deal with the calling convention differences between 64-bit and 32-bit. Our shellcode will therefore be responsible for moving 64-bit arguments' register/stack slots to the 32-bit arguments register/stack slots. Enforcing that 32-bit callbacks may only be __cdecl makes this easier as all arguments are on the stack and the shellcode has full control of stack layout and cleanup. Figure 19 shows the locations of the arguments for each calling convention. Once the first 4 arguments are relocated all further arguments can be moved in a loop since it's simply moving stack values into lower slots. This is relatively easy to implement using external masm files in MSVC. Raw bytes will need to be emitted at points rather than using the assembler due to the mix of architectures. Alternatively, GCC or Clang inline assembly could be used. ReWolf's work achieves the opposite direction of 32-bit -> 64-bit and implements the shellcode via msvc inline asm. X64

MSVC doesn't support this and there are complications with REX prefixes when using that method. It's nicer to use external masm files and rely on the linker to implement this shellcode.

Arg Number	Cdecl Location	Fastcall Location	Special Case?
0	[ebp + 8]	rcx	Yes
1	[ebp + 12]	rdx	Yes
2	[ebp + 16]	r8d	Yes
3	[ebp + 20]	r9d	Yes
4	[ebp + 24]	[rbp + 32 + 8]	No
5	[ebp + 28]	[rbp + 32 + 16]	No
6	[ebp + 32]	[rbp + 32 + 24]	No

Figure 19: Cdecl vs Fastcall argument positions

Once this shellcode is written and wrapped into a nice C++ function, it's possible for the wow64log DLL to invoke the callback via a simple C style function pointer call shown in Figure 20.

```
switch (SysCallMap[i].id) {
case NtResumeThread_ID: {
    if (!g_Settings.p_hNtResumeThread)
        break;

    call_function32(g_Settings.p_hNtResumeThread, ServiceParameters->Status, ServiceParameters->PostCall, pArgs[0], pArgs[1]);
    break;
}
```

Figure 20: call_function32 invokes shellcode to call a 32-bit callback from the 64-bit logging DLL

From within the 32-bit callback any desired MITM operations can be performed, but restrictions exist on which APIs are callable. Due to the context saving that the WOW64 layer performs, 32-bit APIs that would re-enter the WOW64 layer may not be called as the context values would be corrupted. We are therefore limited to only APIs that won't re-enter WOW64, which are those that are exported from the 64-bit ntdll. The NtWriteFile export may be used to easily write to stdout or a file, but we must re-enter the 64-bit execution mode and do the inverse argument mapping as before. This logging routine can be called from within the 32-bit callbacks and is shown in Figure 21 and Figure 22.

```cpp
template<typename... Args>
NTSTATUS Log(const char* fmt, Args... args) {
    char buf[256] = { 0 };
    sprintf_s(buf, fmt, args...);

    NtDefs::IO_STATUS_BLOCK IoStatusBlock = { 0 };
    NTSTATUS status = call_function64(NtWriteFile, (uint64_t)GetStdHandle(STD_OUTPUT_HANDLE), 0, 0, 0, &IoStatusBlock, (uint64_t)buf, strlen(buf), 0, 0);
    return status;
}
```

Figure 21: call_function64 invokes shellcode to call the 64bit WriteFile from with the 32bit callback

```cpp
uint32_t __cdecl hNtResumeThread(NTSTATUS status, uint32_t isPostCall, HANDLE ThreadHandle, PULONG SuspendCount)
{
    if (isPostCall) {
        Log("wow64log!whNtQueueApcThread Status: %d %d", status, ThreadHandle);
    } else {
        Log("wow64log!whNtQueueApcThread %d", ThreadHandle);
    }
    return 0;
}
```

Figure 22: 32bit callbacks must log via routines that only call non-reentrant WOW64 APIs

The result is clean looking callback stubs that function exactly how inline hooks might, but with zero assembly modifications required. Arguments can easily be manipulated as well, but the return status may not be modified unless a little stack walk hackery is implemented. The only other consideration is that the wow64log DLL itself needs to be carefully crafted to not build with any CRT mechanisms. The flags required are:

- Disable CRT with /NODEFAULT LIB (all C APIs now unavailable), set a new entry point name to not init CRT NtDllMain
- Disable all CRT security routines /GS-
- Disable C++ exceptions
- Remove default linker libraries, only link ntdll.lib
- Use extern "C" __declspec(dllimport) <typedef> to link against the correct NtApis

An example of a program hooking its own system calls via wow64log inline hooks is shown in Figure 23.
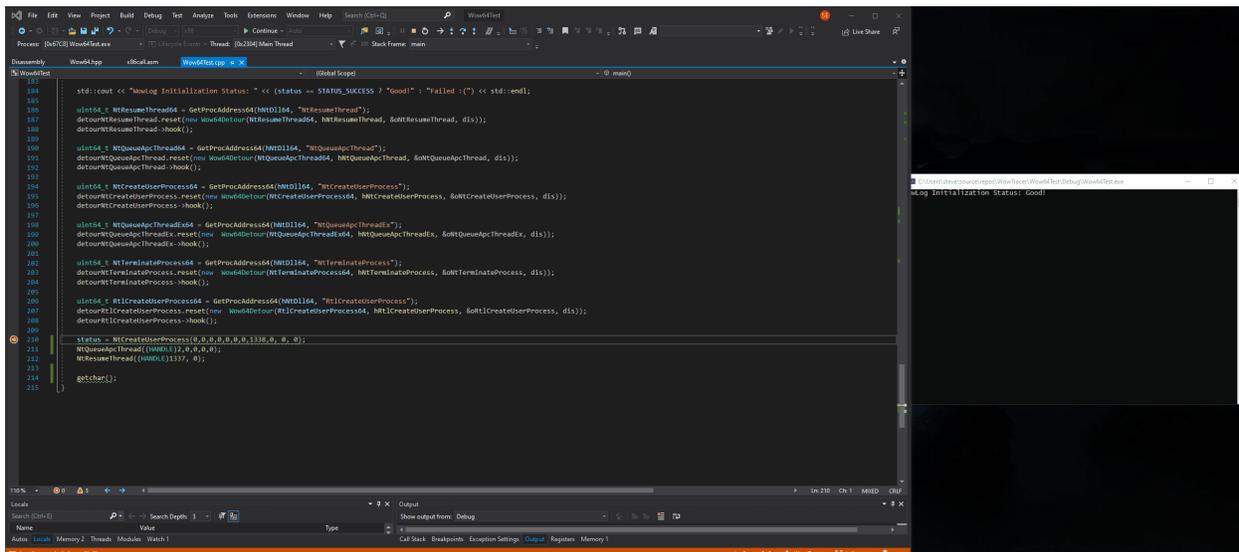

Figure 23: Demonstration of inline hooks in action

Conclusion

Using inline WOW64 hooks, wow64log hooks, and kernel/hypervisor hooks, all techniques of usermode hook evasion can be identified easily and automatically. Detecting which layers of hooks are skipped or bypassed will give insight into which evasion technique is employed. The identifying table is:

Evasion Mode	32bit Inline	wow64Log	64bit Inline	Kernel/Hypervisor
Prologue Restore	✘	✔	✔	✔
Heavens Gate sys-stub	✘	✘	✔	✔
Heavens Gate direct syscall	✘	✘	✘	✔

Structure Appendix

```
struct _WOW64_CPURESERVED
{
  USHORT Flags;
  USHORT MachineType;
  WOW64_CONTEXT Context;
  char ContextEx[1024];
};

typedef ULONG *WOW64_LOG_ARGUMENTS;
struct _WOW64_SYSTEM_SERVICE
{
  unsigned __int32 SystemCallNumber : 12;
  unsigned __int32 ServiceTableIndex : 4;
  unsigned __int32 TurboThunkNumber : 5;
  unsigned __int32 AlwaysZero : 11;
};
#pragma pack(push, 1)
struct _WOW64_FLOATING_SAVE_AREA
{
  DWORD ControlWord;
  DWORD StatusWord;
  DWORD TagWord;
  DWORD ErrorOffset;
```

```c
  DWORD ErrorSelector;
  DWORD DataOffset;
  DWORD DataSelector;
  BYTE RegisterArea[80];
  DWORD Cr0NpxState;
};
#pragma pack(pop)

#pragma pack(push, 1)
struct _WOW64_CONTEXT
{
  DWORD ContextFlags;
  DWORD Dr0;
  DWORD Dr1;
  DWORD Dr2;
  DWORD Dr3;
  DWORD Dr6;
  DWORD Dr7;
  WOW64_FLOATING_SAVE_AREA FloatSave;
  DWORD SegGs;
  DWORD SegFs;
  DWORD SegEs;
  DWORD SegDs;
  DWORD Edi;
  DWORD Esi;
  DWORD Ebx;
  DWORD Edx;
  DWORD Ecx;
  DWORD Eax;
  DWORD Ebp;
  DWORD Eip;
  DWORD SegCs;
  DWORD EFlags;
  DWORD Esp;
  DWORD SegSs;
  BYTE ExtendedRegistersUnk[160];
  M128A Xmm0;
  M128A Xmm1;
  M128A Xmm2;
  M128A Xmm3;
  M128A Xmm4;
  M128A Xmm5;
  M128A Xmm6;
```

```c
    M128A Xmm7;
    M128A Xmm8;
    M128A Xmm9;
    M128A Xmm10;
    M128A Xmm11;
    M128A Xmm12;
    M128A Xmm13;
    M128A Xmm14;
    M128A Xmm15;
};
#pragma pack(pop)
```