Hooking Heaven’s Gate — a WOW64 hooking technique

medium.com/@fsx30/hooking-heavens-gate-a-wow64-hooking-technique-5235e1aeed73

Hoang Bui February 3, 2020

Hoang Bui

This is not new, this is not novel, and definitely not my research — but I used it recently so here is my attempt at explaining some cool WOW64 concept. I also want to take a break from reading AMD/Intel manual to write this hypervisor. I also think the term “Heaven’s Gate” is quite appropriate and is the coolest thing ever, so here we have it.

Introduction

I usually add some pictures here to show how I started my journey but because it was 2 months ago on a free slack (shoutout to GuidedHacking), I don’t have the log anymore. Either way, it went something like this…

```
Me: Yoooooooo any good technique to catch a manual syscall?!?!?: That is going to be
tough.: Wait, is it Wow64?Me: Yes: You can’t manual syscall on Wow64, you coconut.Me:
????
```

So there you have it, no such thing as a manual syscall on WOW64. Well, there is one way but I will covert that topic at a later time. (Hint: Heaven’s Gate)

First, we need to understand a bit about WOW64.

WoW64 (Windows 32-bit on Windows 64-bit)

I will covert a very brief part simply due to the fact of how complicated the subsystem is and prone for possible mistakes that I might make.

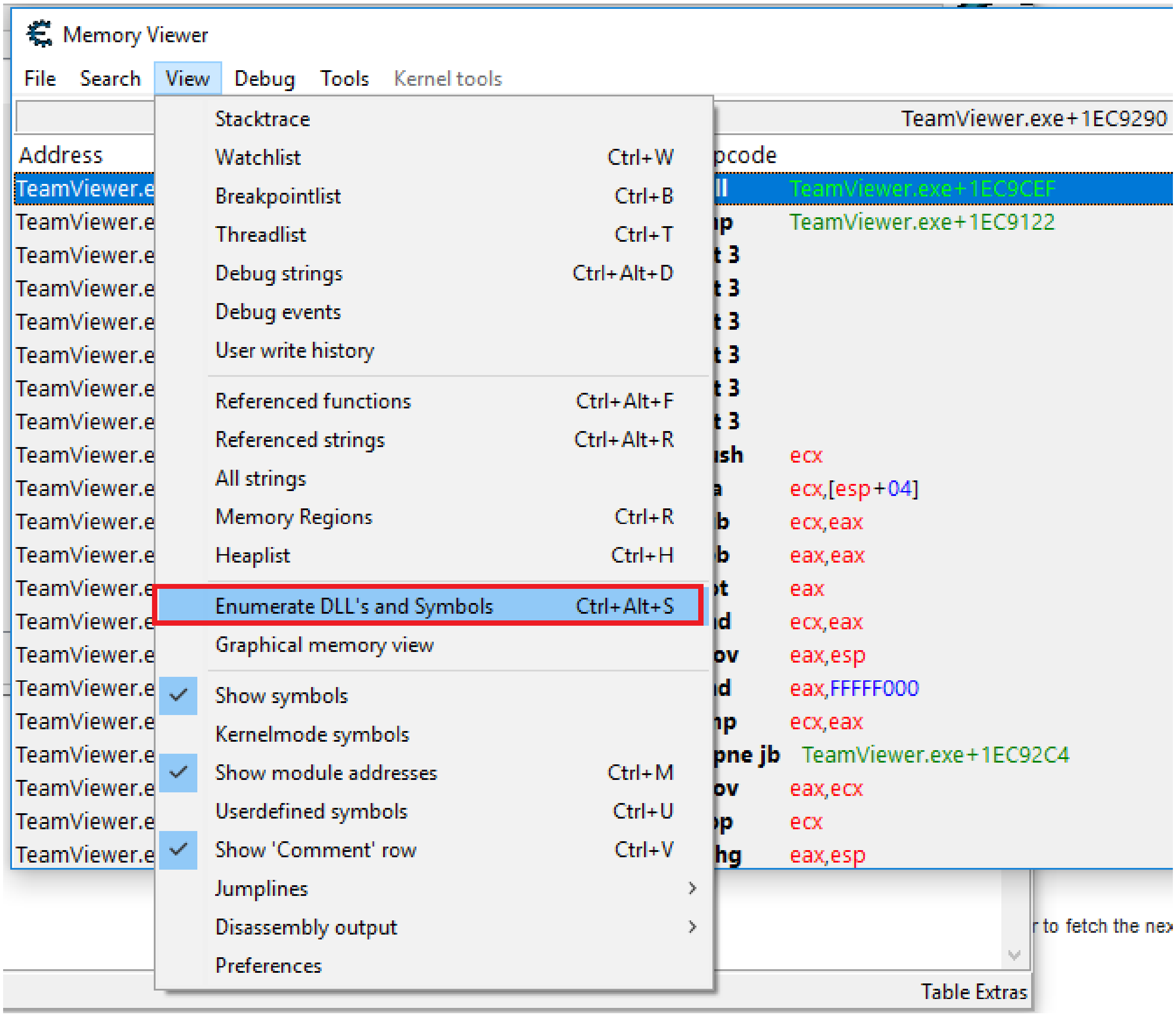
WOW64 applies to 32 bit applications running on a 64 bit machine. This mean that while there is very small different in how the 32 bit and the 64 bit kernel work, there is no doubt incompatibilities. This subsystem tries to mitigate those incompatibilities through various interfaces such as `wow64.dll`, `wow64win.dll`, and `wow64cpu.dll`. There is also a different registry environment for wow64 applications vs native 64-bit applications but let’s not get into that mess.

An interesting behavior to notice while executing a WOW64 application is that all kernel-mode components on a 64-bit machine will always execute in 64-bit mode, regardless whether the application’s instructions are 64-bit or not.

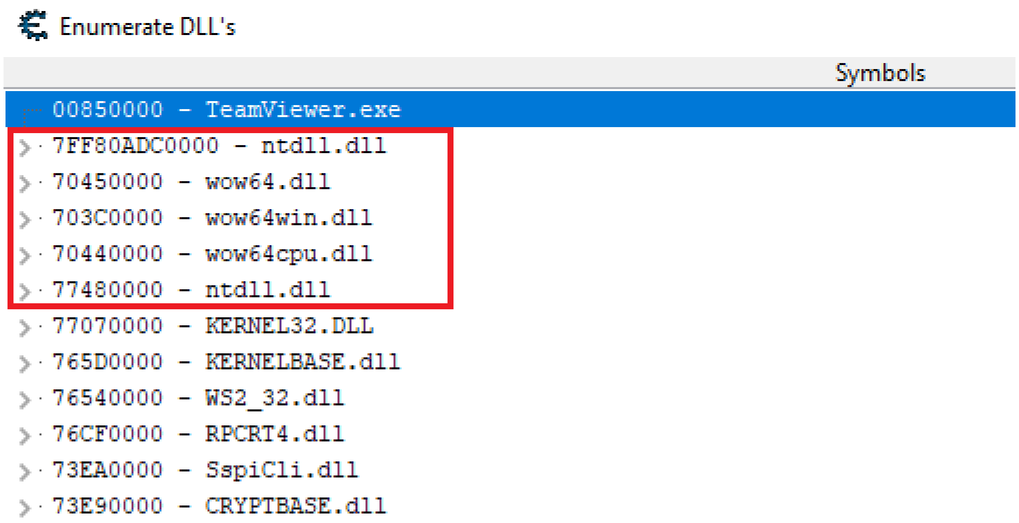
This in conclusion means that WOW64 applications run a bit differently than a native 64 bit application. We are going to take advantage of that. Let's look at the difference when it comes to calling a `WINAPI`.

NTDLL.dll vs NTDLL.dll

Ntdll.dll on a Windows machine is widely covered and I won't go too deep into that. We are only interested in the feature of ntdll.dll when performing a WINAPI call that requires a syscall. Let's pick Cheat Engine as our debugger (because it can see both DLLs) and Teamviewer as our WOW64 application.



If you can't find the functionality



Enumerate DLL's

Symbols

```
00850000 - TeamViewer.exe
7FF80ADC0000 - ntdll.dll
70450000 - wow64.dll
703C0000 - wow64win.dll
70440000 - wow64cpu.dll
77480000 - ntdll.dll
77070000 - KERNEL32.DLL
765D0000 - KERNELBASE.dll
76540000 - WS2_32.dll
76CF0000 - RPCRT4.dll
73EA0000 - SspiCli.dll
73E90000 - CRYPTBASE.dll
```

Ara ara? What is so strange about this

If this was a live conversation, I would torment you with this question but this is not a live session. Noticed, there are those 3 wow64 interface dlls that I mentioned earlier, but the particular thing you want to notice is the **two**`ntdll.dll`. What even more bizarre is that one of the `ntdll.dll` is currently residing in a 64 bit address space. Wtf? How? This is a 32 bit application!

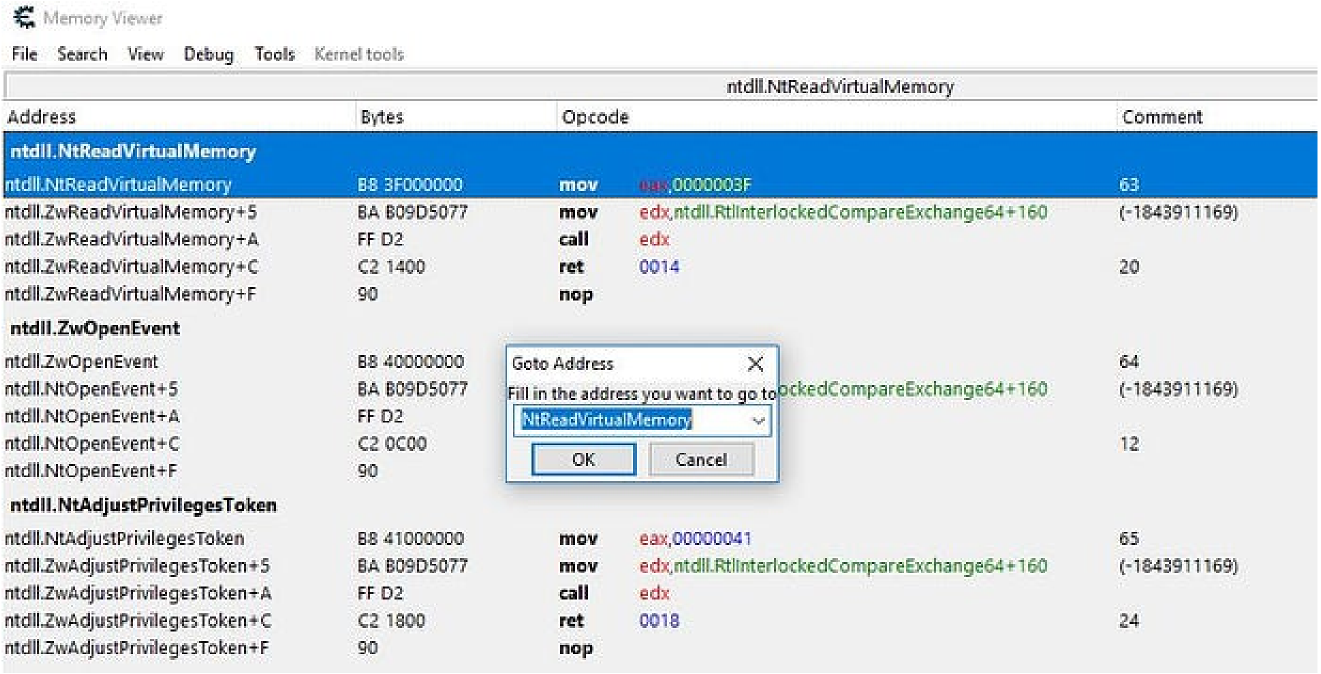
The answer: **WOW64**.

The Differences

I am sure there are a ton more differences between the two dlls but let's cover the very first obvious difference, the syscalls.

We all know (if not, now you do) that ntdll.dll in a normal native application is the one responsible for performing the syscall/sysenter, handing the execution over to the kernel. But I also mentioned earlier that you cannot perform a syscall on a WOW64 application. So how does WOW64 application do… anything?

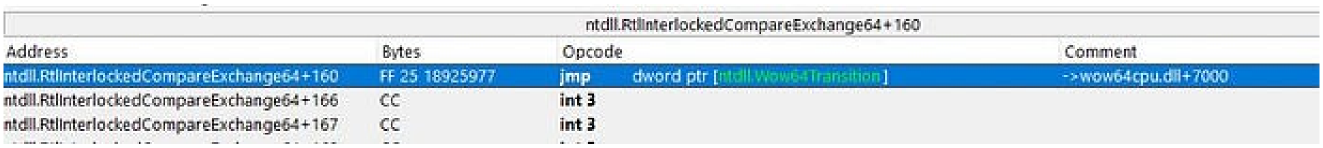
By going into an example function such as `NtReadVirtualMemory`, we should be expecting a `service id` to be placed on the `eax` register and follow by a `syscall/sysenter` instruction.



```
                                    ntdll.NtReadVirtualMemory
Address                          Bytes         Opcode                                                Comment
ntdll.NtReadVirtualMemory
ntdll.NtReadVirtualMemory        B8 3F000000   mov    eax,0000003F                                   63
ntdll.ZwReadVirtualMemory+5      BA B09D5077   mov    edx,ntdll.RtlInterlockedCompareExchange64+160  (-1843911169)
ntdll.ZwReadVirtualMemory+A      FF D2         call   edx
ntdll.ZwReadVirtualMemory+C      C2 1400       ret    0014                                           20
ntdll.ZwReadVirtualMemory+F      90            nop
ntdll.ZwOpenEvent
ntdll.ZwOpenEvent                B8 40000000                                                         64
ntdll.NtOpenEvent+5              BA B09D5077          ockedCompareExchange64+160                     (-1843911169)
ntdll.NtOpenEvent+A              FF D2
ntdll.NtOpenEvent+C              C2 0C00                                                             12
ntdll.NtOpenEvent+F              90
ntdll.NtAdjustPrivilegesToken
ntdll.NtAdjustPrivilegesToken    B8 41000000   mov    eax,00000041                                   65
ntdll.ZwAdjustPrivilegesToken+5  BA B09D5077   mov    edx,ntdll.RtlInterlockedCompareExchange64+160  (-1843911169)
ntdll.ZwAdjustPrivilegesToken+A  FF D2         call   edx
ntdll.ZwAdjustPrivilegesToken+C  C2 1800       ret    0018                                           24
ntdll.ZwAdjustPrivilegesToken+F  90            nop
```

No syscall, at all

Okay, now that's weird. There is no `syscall`. Instead, there is a `call` and I know for sure you can't just enter kernel land with just a `call.` Let's follow the `call`!



```
                          ntdll.RtlInterlockedCompareExchange64+160
Address                                Bytes          Opcode                                   Comment
ntdll.RtlInterlockedCompareExchange64+160  FF 25 18925977  jmp    dword ptr [ntdll.Wow64Transition]  ->wow64cpu.dll+7000
ntdll.RtlInterlockedCompareExchange64+166  CC              int 3
ntdll.RtlInterlockedCompareExchange64+167  CC              int 3
```

A jump to wow64transition inside wow64cpu.dll

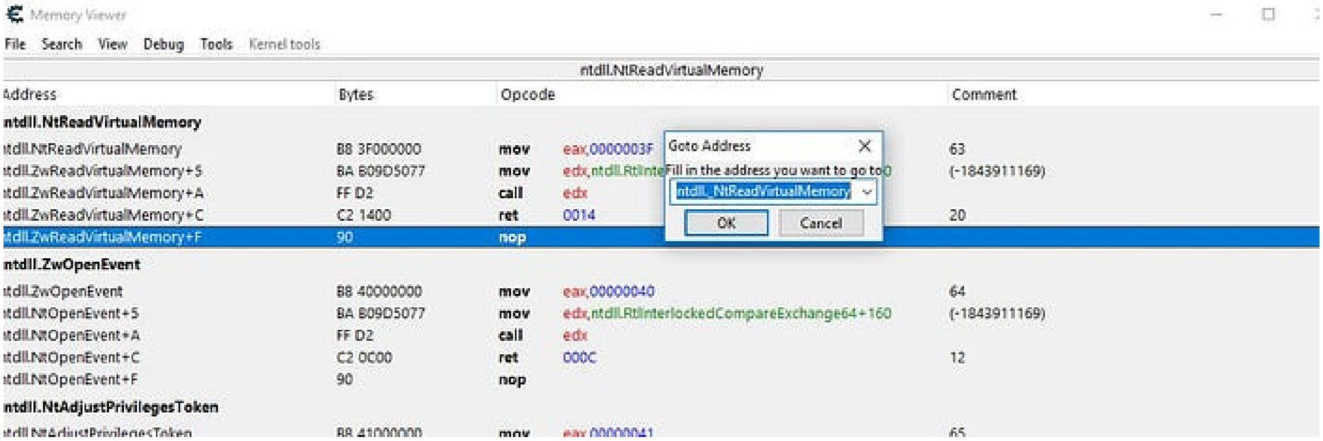
Another jump, into another jump…hold up, is that "RAX" I see?.. isn't RAX a 64-bit register ?

We are now at some place inside `wow64cpu.dll` called `Wow64Transition` that is now executing with 64 bits instruction set. We also see that it is referencing `CS:0x33` segment. What is going on?

In Alex Lonescu' blog, he said:

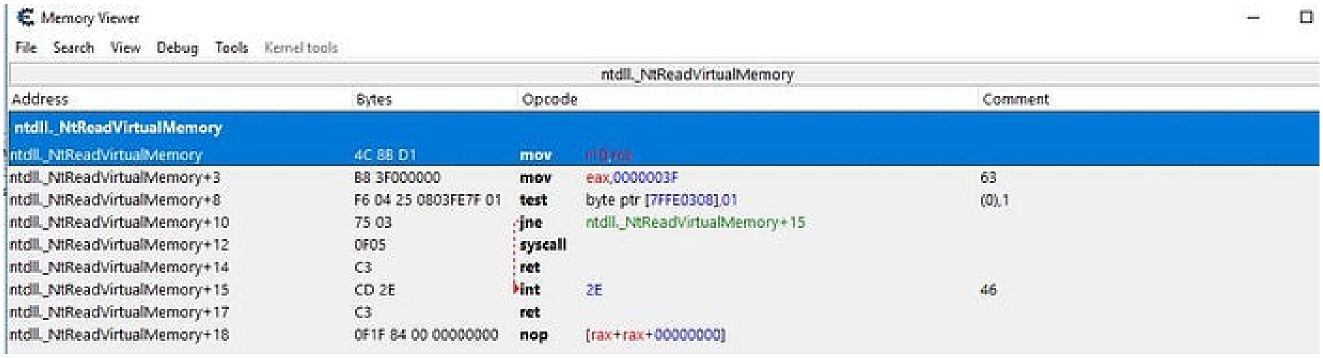
> In fact, on 64-bit Windows, the first piece of code to execute in *any* process, is always the 64-bit NTDLL, which takes care of initializing the process in user-mode (as a 64-bit process!). It's only later that the Windows-on-Windows (WoW64) interface takes over, loads a 32-bit NTDLL, and execution begins in 32-bit mode through a far jump to a compatibility code segment. The 64-bit world is never entered again, **except whenever the 32-bit code attempts to issue a system call. The 32-bit NTDLL that was loaded, instead of containing the expected SYSENTER instruction, actually contains a series of instructions to jump back into 64-bit mode, so that the system call can be issued with the SYSCALL instruction, and so that parameters can be sent using the x64 ABI, sign-extending as needed**.

So what this mean is that when the 32-bit code is trying to perform a `syscall`, it would go through the 32-bit `ntdll.dll`, and then to this particular transition gate (Heaven’s Gate) and performs a `far jump` instruction which **switches into long-mode (64-bit)** enabled code segment. That is the `0033:wow64cpu.dll+0x7009` you see in the latest screenshot. Now that we are in 64-bit context, we can finally go to the 64-bit ntdll.dll which is where the real syscall is performed.


Memory Viewer
File
Search
View
Debug
Tools
Kernel tools
ntdll.NtReadVirtualMemory
Goto Address
Fill in the address you want to go to0
ntdll._NtReadVirtualMemory
OK
Cancel

```
Address                          Bytes        Opcode                                                  Comment
ntdll.NtReadVirtualMemory
tdll.NtReadVirtualMemory         B8 3F000000  mov   eax,0000003F                                      63
tdll.ZwReadVirtualMemory+5       BA B09D5077  mov   edx,ntdll.RtlInte                                 (-1843911169)
tdll.ZwReadVirtualMemory+A       FF D2        call  edx
tdll.ZwReadVirtualMemory+C       C2 1400      ret   0014                                              20
tdll.ZwReadVirtualMemory+F       90           nop
ntdll.ZwOpenEvent
tdll.ZwOpenEvent                 B8 40000000  mov   eax,00000040                                      64
tdll.NtOpenEvent+5               BA B09D5077  mov   edx,ntdll.RtlInterlockedCompareExchange64+160     (-1843911169)
tdll.NtOpenEvent+A               FF D2        call  edx
tdll.NtOpenEvent+C               C2 0C00      ret   000C                                              12
tdll.NtOpenEvent+F               90           nop
ntdll.NtAdjustPrivilegesToken
tdll.NtAdjustPrivilegesToken     B8 41000000  mov   eax,00000041                                      65
```

You can specify in Cheat Engine 64bit WINAPI version with _ before the API’s name



```
Address                          Bytes                  Opcode                                          Comment
ntdll._NtReadVirtualMemory
ntdll._NtReadVirtualMemory       4C 8B D1               mov     r10,rcx
ntdll._NtReadVirtualMemory+3     B8 3F000000            mov     eax,0000003F                            63
ntdll._NtReadVirtualMemory+8     F6 04 25 0803FE7F 01   test    byte ptr [7FFE0308],01                  (0),1
ntdll._NtReadVirtualMemory+10    75 03                  jne     ntdll._NtReadVirtualMemory+15
ntdll._NtReadVirtualMemory+12    0F05                   syscall
ntdll._NtReadVirtualMemory+14    C3                     ret
ntdll._NtReadVirtualMemory+15    CD 2E                  int     2E                                      46
ntdll._NtReadVirtualMemory+17    C3                     ret
ntdll._NtReadVirtualMemory+18    0F1F 84 00 00000000    nop     [rax+rax+00000000]
```

Finally the expected syscall

There you have it, the full WOW64 syscall chain. Let’s summarize.

```
32-bit ntdll.dll -> wow64cpu.dll’s Heaven’s Gate -> 64-bit ntdll.dll -> syscall into
the kernel
```

Now that we understand the full execution chain, let’s get hooking!

Hooking Heaven’s Gate

So as hackers, we are always looking for a stealthy way to hook stuff. While hooking heaven’s gate is in no way stealthy, it is a lot stealthier (and more useful) than hooking the single Winapi functions. That is because **ALL** syscall go through **ONE** gate, meaning by hooking this ONE gate — you are hooking **ALL** syscalls.

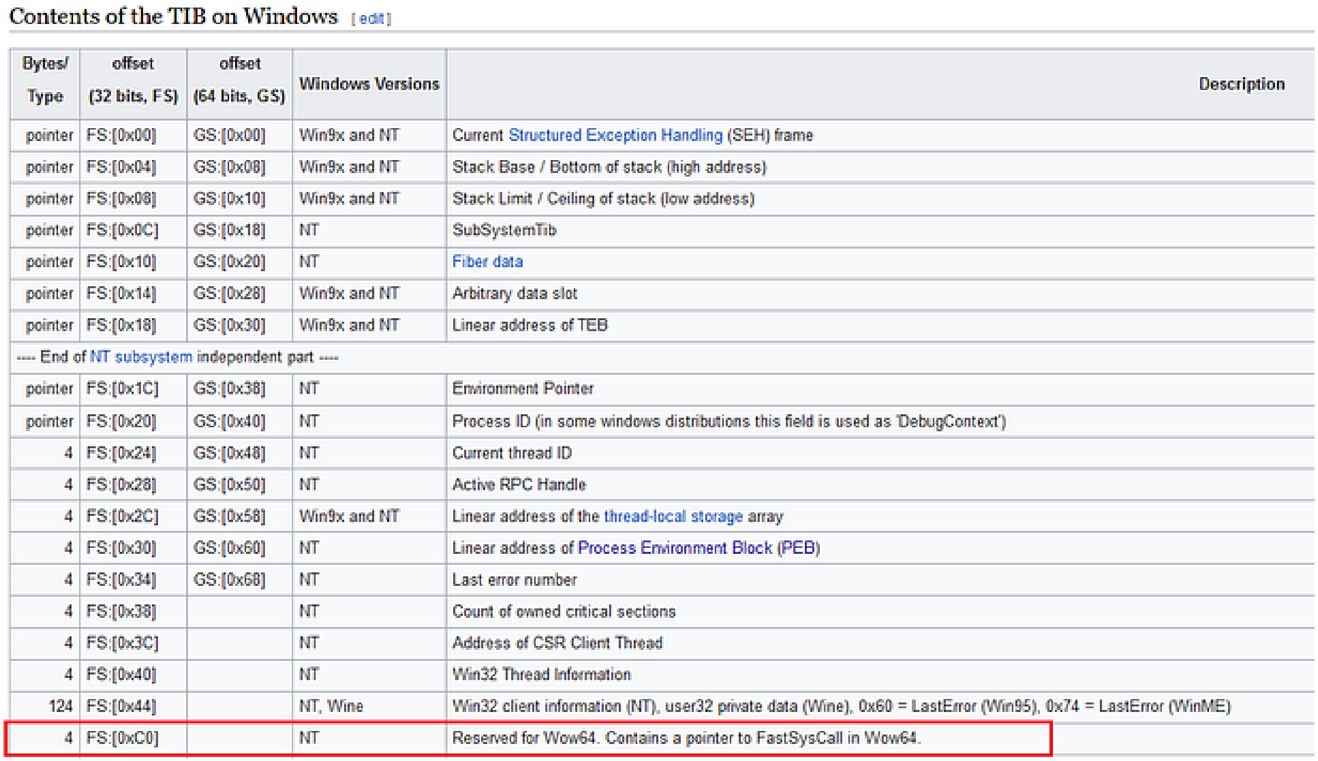
The Plan

Our plan is quite simple. We will do what we usually do with a normal detour hook.

1. We will place a jmp of some sort on the transition gate/Heaven's Gate, which will then jump to our shellcode
2. Our shellcode will select what service id to hook and jump to the appropriate hook.
3. Our hook once finished execution, will jump to the transition gate/Heaven's Gate.
4. Transition gate/Heaven's Gate will continue on with the context switch into 64-bit and execute as normal

But first, how does the application knows where is heaven's gate located?

Answer: **FS:0xC0 aka TIB + 0xC0**



Contents of the TIB on Windows [edit]

Bytes/ Type	offset (32 bits, FS)	offset (64 bits, GS)	Windows Versions	Description
pointer	FS:[0x00]	GS:[0x00]	Win9x and NT	Current Structured Exception Handling (SEH) frame
pointer	FS:[0x04]	GS:[0x08]	Win9x and NT	Stack Base / Bottom of stack (high address)
pointer	FS:[0x08]	GS:[0x10]	Win9x and NT	Stack Limit / Ceiling of stack (low address)
pointer	FS:[0x0C]	GS:[0x18]	NT	SubSystemTib
pointer	FS:[0x10]	GS:[0x20]	NT	Fiber data
pointer	FS:[0x14]	GS:[0x28]	Win9x and NT	Arbitrary data slot
pointer	FS:[0x18]	GS:[0x30]	Win9x and NT	Linear address of TEB
---- End of NT subsystem independent part ----				
pointer	FS:[0x1C]	GS:[0x38]	NT	Environment Pointer
pointer	FS:[0x20]	GS:[0x40]	NT	Process ID (in some windows distributions this field is used as 'DebugContext')
4	FS:[0x24]	GS:[0x48]	NT	Current thread ID
4	FS:[0x28]	GS:[0x50]	NT	Active RPC Handle
4	FS:[0x2C]	GS:[0x58]	Win9x and NT	Linear address of the thread-local storage array
4	FS:[0x30]	GS:[0x60]	NT	Linear address of Process Environment Block (PEB)
4	FS:[0x34]	GS:[0x68]	NT	Last error number
4	FS:[0x38]		NT	Count of owned critical sections
4	FS:[0x3C]		NT	Address of CSR Client Thread
4	FS:[0x40]		NT	Win32 Thread Information
124	FS:[0x44]		NT, Wine	Win32 client information (NT), user32 private data (Wine), 0x60 = LastError (Win95), 0x74 = LastError (WinME)
4	FS:[0xC0]		NT	Reserved for Wow64. Contains a pointer to FastSysCall in Wow64.

FastSysCall is the another name for the Transition Gate aka Heaven's Gate

So, in theory — we could determine where Heaven's Gate is by using this code snippet.

```
const DWORD_PTR __declspec(naked) GetGateAddress(){    __asm    {        mov eax,
dword ptr fs : [0xC0]        ret    }}
```

Now that we know where the current Heaven's Gate is at, and we are going to hook it — let's create a "backup" of the code we are about to modify.

```
const LPVOID CreateNewJump()
{
    lpJmpRealloc = VirtualAlloc(nullptr, 4096, MEM_RESERVE | MEM_COMMIT,
        PAGE_EXECUTE_READWRITE);
    memcpy(lpJmpRealloc, (void *)GetGateAddress(), 9);

    return lpJmpRealloc;}
```

This will effectively allocate a new page and copy 9 bytes **far jmp** from heaven’s gate over. Why we do this will not be covered but if you want to know the specific term, we are creating a **trampoline** for our **detour hook**. This will allow us to preserve the **far jmp instructions** that we are about to overwrite in the next step.

> The 9 bytes is the instruction we are backing up: jmp 0033:wow64cpu.dll + 7009

Next, we are going to replace that far jmp with a `PUSH Addr, RET`effectively acting as an absolute address jump. (Push the address you want to jump onto the stack, Ret will pop it from the stack and jmp there)

```
void __declspec(naked) hk_Wow64Trampoline()
{
    __asm
    {
        cmp eax, 0x3f //64bit Syscall id of NtRVM
        je hk_NtReadVirtualMemory
        cmp eax, 0x50 //64bit Syscall id of NtPVM
        je hk_NtProtectVirtualMemory
        jmp lpJmpRealloc
    }
}

const LPVOID CreateNewJump()
{
    DWORD_PTR Gate = GetGateAddress();
    lpJmpRealloc = VirtualAlloc(nullptr, 0x1000, MEM_RESERVE | MEM_COMMIT,
        PAGE_EXECUTE_READWRITE);
    memcpy(lpJmpRealloc, (void *)Gate, 9);

    return lpJmpRealloc;
}

const void WriteJump(const DWORD_PTR dwWow64Address, const void *pBuffer, size_t
ulSize)
{
    DWORD dwOldProtect = 0;
    VirtualProtect((LPVOID)dwWow64Address, 0x1000, PAGE_EXECUTE_READWRITE,
&dwOldProtect);
    (void)memcpy((void *)dwWow64Address, pBuffer, ulSize);
    VirtualProtect((LPVOID)dwWow64Address, 0x1000, dwOldProtect, &dwOldProtect);
}

const void EnableWow64Redirect()
{
    LPVOID Hook_Gate = &hk_Wow64Trampoline;

    char trampolineBytes[] =    {        0x68, 0xDD, 0xCC, 0xBB, 0xAA,       /*push
0xAABBCCDD*/        0xC3,                              /*ret*/        0xCC, 0xCC,
0xCC                    /*padding*/    };    memcpy(&trampolineBytes[1], &Hook_Gate,
4);    WriteJump(GetGateAddress(), trampolineBytes, sizeof(trampolineBytes));}
```

This code will overwrite the 9 bytes **FAR JMP** along with all the VirtualProtect you need.

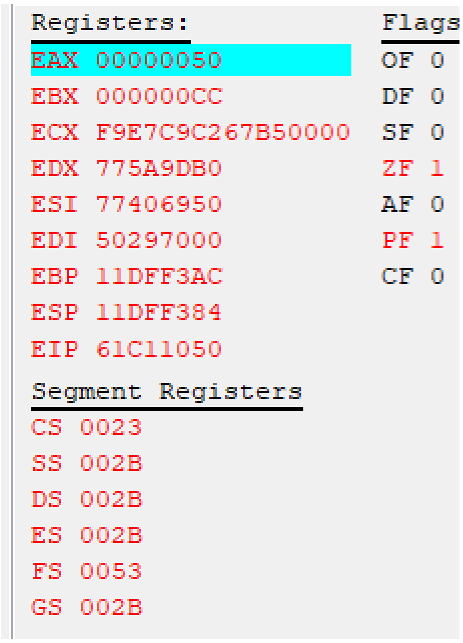
Let’s dissect hk_Wow64Trampoline.

So we know that before any syscall happen, the service id is ALWAYS in the EAX register. Therefore, we can use a `cmp`instruction to determine what is being called and jmp to the appropriate hook function. In our case we are doing 2 cmp (but you can do as many as you want), one with 0x3f and one with 0x50 — NtRVM and NtPVM. If the EAX register holds the correct syscall, je or jump-equal will execute, effectively jumping to our hook function. If it is not the syscall we want, it will take a jmp to lpJmpRealloc (which we created in our CreateNewJump function. This is the 9 original bytes that we copied over before overwriting it).

```
void __declspec(naked) hk_NtProtectVirtualMemory()
{
    __asm {
        mov Backup_Eax, eax
        mov eax, [esp + 0x8]
        mov Handle, eax
        mov eax, [esp + 0xC]
        mov Address_1, eax
        mov eax, [esp + 0x10]
        mov DwSizee, eax
        mov eax, [esp + 0x14]
        mov New, eax
        mov eax, [esp + 0x18]
        mov Old, eax
        mov eax, Backup_Eax
        pushad
    }

    printf("NtPVM Handle: [%x] Address: [0x%x]  Size: [%d]  NewProtect: [0x%x]\n",
Handle, Address_1, *DwSizee, New);

    __asm popad
    __asm jmp lpJmpRealloc
}

void __declspec(naked) hk_NtReadVirtualMemory()
{
    __asm pushad

    printf("Calling NtReadVirtualMemory.\n");

    __asm popad    __asm jmp lpJmpRealloc}
```

Note that before you are doing any sort of stuff within the hook function, you must pushad/pushfd and then later popfd/popad to preserve the registers and the flags. If you do not do this, expect the program to crash in no time.

Similarly, I've tried very hard to get the values from the declspec(naked) function through arguments but it just can't do because you will end up usign ECX as a register and ECX just happens to hold a 64bit value in my experience.



```
Registers:              Flags
EAX 00000050            OF 0
EBX 000000CC            DF 0
ECX F9E7C9C267B50000    SF 0
EDX 775A9DB0            ZF 1
ESI 77406950            AF 0
EDI 50297000            PF 1
EBP 11DFF3AC            CF 0
ESP 11DFF384
EIP 61C11050

Segment Registers
CS 0023
SS 002B
DS 002B
ES 002B
FS 0053
GS 002B
```

PUSHAD will lose the first 4 bytes of ECX

Please let's me know if you know of a way to get something like this to work.

```
DWORD __declspec(naked) hk_NtProtectVirtualMemory(    IN HANDLE
ProcessHandle,     IN OUT PVOID            *BaseAddress,     IN OUT PULONG
NumberOfBytesToProtect,     IN ULONG                NewAccessProtection,     OUT PULONG
OldAccessProtection)
```

Summary

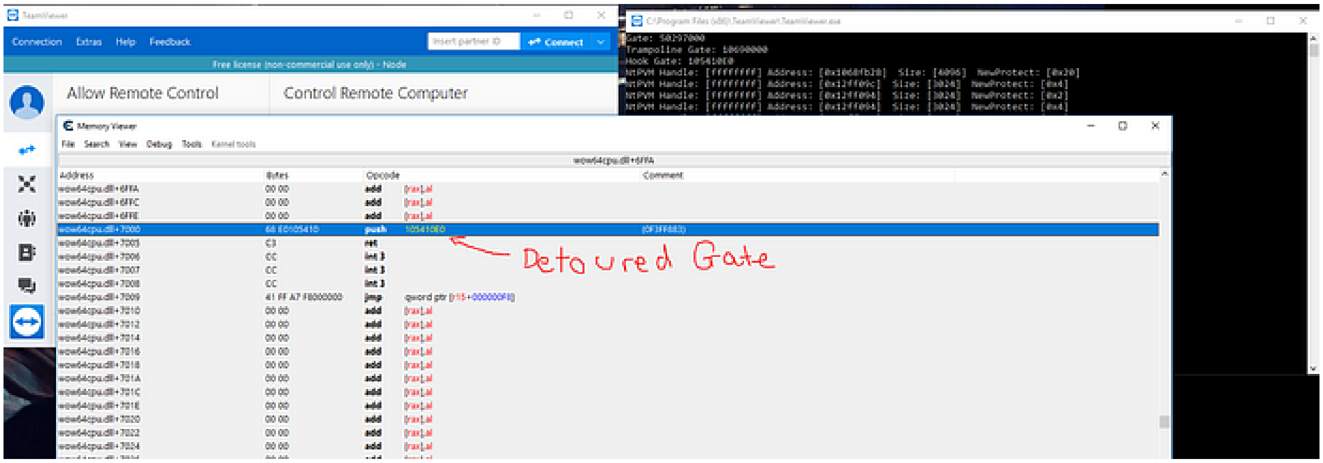
In summary, when you are running as a Wow64 process — you cannot access the kernel directly. You have to go through a transition gate aka Heaven's Gate to transition into 64bit mode before entering Kernel Land. This transition can be hook with a traditional detour which this post covers.

The technique detour the transition gate into a fake gate that does conditional jump based on the service number to the correct hook function. Once the hook function finished execution, it is then jump to a transition gate that we backed up. This will change our 32bit mode into 64bit mode, in which we will then continue with the execution by going into the 64bit Ntdll. 64bit Ntdll will then perform a syscall/sysenter and enter Kernel land.

```
32bit Ntdll-> Heaven's Gate (hooked) -> Fake Gate -> hook_function -> Heaven's
Gate Trampoline -> 64bit Ntdll -> Kernel land
```

Result

Take a look at the example code here.


TeamViewer
Connection Extras Help Feedback
Insert partner ID
Connect
Free license (non-commercial use only) - Node
Allow Remote Control
Control Remote Computer
Memory Viewer
File Search View Debug Tools Kernel tools
Detoured Gate

```
C:\Program Files (x86)\TeamViewer\TeamViewer.exe
Gate: 50297000
Trampoline Gate: 10000000
Hook Gate: 105410E0
NtPVM Handle: [ffffffff] Address: [0x1068fb28]  Size: [4096]  NewProtect: [0x20]
NtPVM Handle: [ffffffff] Address: [0x12ff09c]  Size: [3026]  NewProtect: [0x4]
NtPVM Handle: [ffffffff] Address: [0x12ff094]  Size: [3026]  NewProtect: [0x2]
NtPVM Handle: [ffffffff] Address: [0x12ff094]  Size: [3026]  NewProtect: [0x4]
```

```
wow64cpu.dll+6FFA
Address              Bytes              Opcode                       Comment
wow64cpu.dll+6FFA    00 00              add   [rax],al
wow64cpu.dll+6FFC    00 00              add   [rax],al
wow64cpu.dll+6FFE    00 00              add   [rax],al
wow64cpu.dll+7000    68 E0105410        push  105410E0               (0F3FF883)
wow64cpu.dll+7005    C3                 ret
wow64cpu.dll+7006    CC                 int 3
wow64cpu.dll+7007    CC                 int 3
wow64cpu.dll+7008    CC                 int 3
wow64cpu.dll+7009    41 FF A7 F8000000  jmp   qword ptr [r15+000000F8]
wow64cpu.dll+7010    00 00              add   [rax],al
wow64cpu.dll+7012    00 00              add   [rax],al
wow64cpu.dll+7014    00 00              add   [rax],al
wow64cpu.dll+7016    00 00              add   [rax],al
wow64cpu.dll+7018    00 00              add   [rax],al
wow64cpu.dll+701A    00 00              add   [rax],al
wow64cpu.dll+701C    00 00              add   [rax],al
wow64cpu.dll+701E    00 00              add   [rax],al
wow64cpu.dll+7020    00 00              add   [rax],al
wow64cpu.dll+7022    00 00              add   [rax],al
wow64cpu.dll+7024    00 00              add   [rax],al
```

10/10 paint job

Another thing to notice is that you cannot just printf the syscall Id within the Wow64 hook, and that is because printf requires a syscall (I believe so) and if you hook the printf syscall while calling printf inside the hook, you are going to have a bad time (Infinite loop).

Conclusion

Hooking is a technique consists of multiple methods. How you hook depends on your creativitiy and your understanding of the system. So far, we have prove that we can hook any function at almost all stages. Maybe next we will go into SSDT hook or some sort. However, my OSCE exam is tomorrow so wish me the best of luck. It took me over a month to finish this because I got so side-tracked. Please forgive me if there are more mistakes toward the 2nd half!

-Fs0x30