An Introduction to Bypassing User Mode EDR Hooks

malwaretech.com/2023/12/an-introduction-to-bypassing-user-mode-edr-hooks.html

Marcus Hutchins December 25, 2023

A Trip Back In Time

Recently I got back into malware research and was going through some of my old notes for an article I'm writing. While cross-referencing notes against old blog posts, I realized that I never actually published the majority of my work on system calls and user mode hooking. Since my next article will require that readers be familiar with both concepts, I decided to take the time to polish up and publish the rest of my research. And hey, who's to turn down a free extra blog post?

Whilst this article is designed to stand on its own, if you're interested, you can find my previous articles on these topics here, here, here and here. Surprisingly, despite all this research being over a decade old, it's still completely relevant today. The more things change, the more they stay the same, I guess?

What Is a Syscall

System calls are the standard way to transition from user mode to kernel mode. They are the modern, faster, version of software interrupts.

The system call interface is extremely complex, but since most of it isn't relevant to what we're doing I'm just going to give a higher level summary. For the most part, you won't really need an in-depth understanding of how it works to utilize these techniques, but it can be helpful to know.

On Windows, the kernel has a table of functions that are allowed to be called from user mode. These functions are sometimes referred to as System Services, Native function, or Nt Functions. They are the functions that begin with Nt or Zw and are located in ntoskrnl.exe. The table of system services is known as the System Service Descriptor Table, or SSDT for short.

To call a system service from user mode a system call must be performed, which is done via the `syscall` instruction. The application tells the kernel which system service it wants to call by storing its ID into the eax register. The System Service ID (often called a System Service Number, System Call Number, or simply SSN), is the index of the function's entry within the SSDT. So, setting eax to 0 will call the first function in the SSDT, 1 will call the second, 2 will call the third, and so on…

the lookup looks something like this `entry = nt!KiServiceTable+(SSN * 4)`.

The syscall instruction causes the CPU to switch into kernel mode and invoke the system call handler, which takes the SSN from the eax register and calls the corresponding SSDT function.

Let's say an application calls the `OpenProcess()` function from kernel32.dll to open a handle to a process.

```
HANDLE __stdcall OpenProcess(DWORD dwDesiredAccess, BOOL bInheritHandle, DWORD dwProcessId)
{
  CLIENT_ID client_id; // [rsp+20h] [rbp-48h] BYREF
  struct _OBJECT_ATTRIBUTES ObjectAttributes; // [rsp+30h] [rbp-38h] BYREF
  void *process_handle; // [rsp+88h] [rbp+20h] BYREF

  client_id.UniqueThread = 0i64;
  ObjectAttributes.Length = 48;
  ObjectAttributes.RootDirectory = 0i64;
  client_id.UniqueProcess = dwProcessId;
  ObjectAttributes.Attributes = bInheritHandle ? 2 : 0;
  ObjectAttributes.ObjectName = 0i64;
  *&ObjectAttributes.SecurityDescriptor = 0i64;
  if ( NtOpenProcess(&process_handle, dwDesiredAccess, &ObjectAttributes, &client_id) >= 0 )
    return process_handle;
  BaseSetLastNTError();
  return 0i64;
}
```

A disassembly of kernel32!OpenProcess().

As you can see, all the function really does is set up a call to `NtOpenProcess()`, which is located in ntdll.dll.

Now, let's take a look at the `NtOpenProcess()` logic.

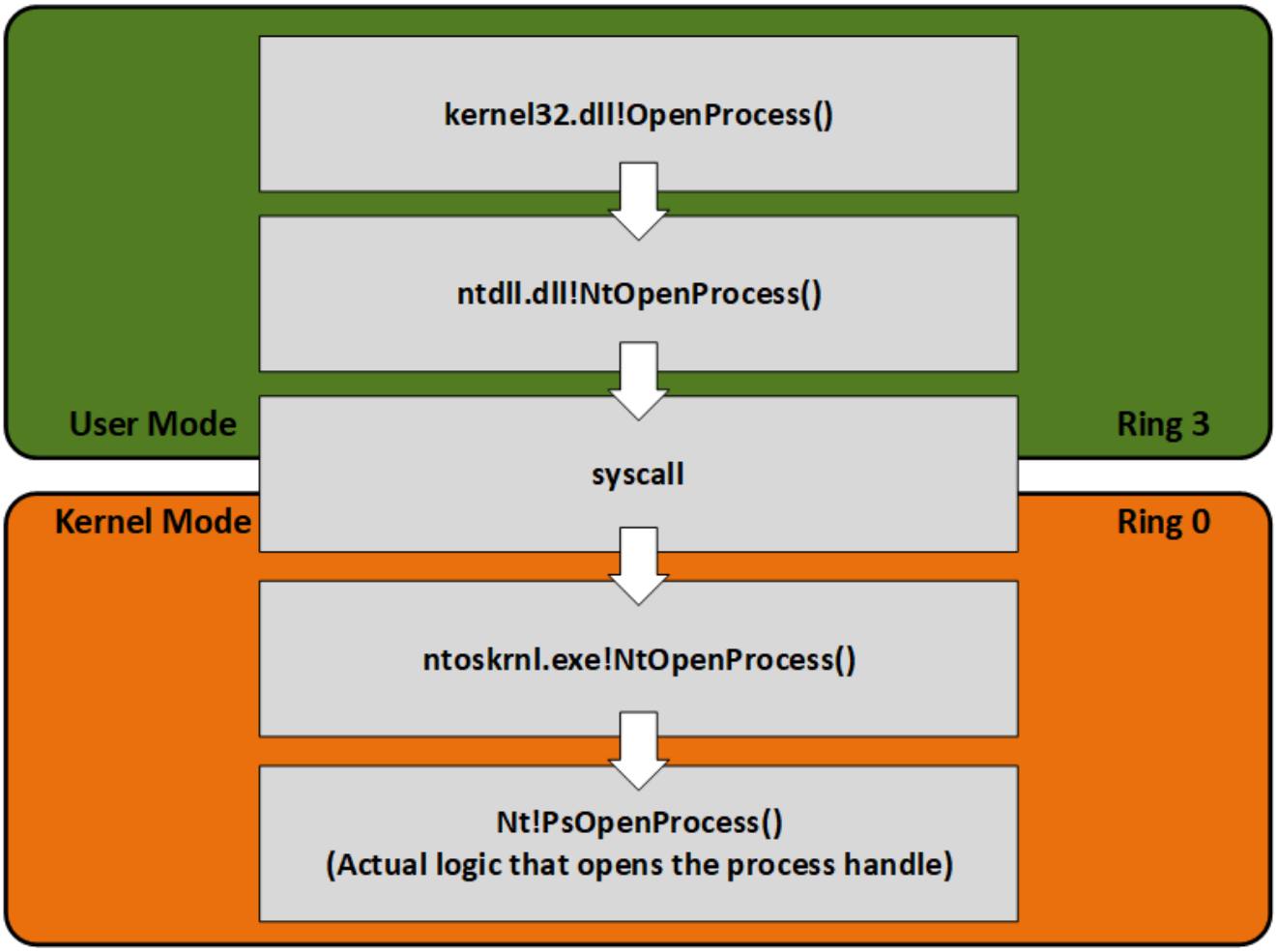
```
; __int64 NtOpenProcess()
                public NtOpenProcess
NtOpenProcess   proc near

                mov     r10, rcx
                mov     eax, 26h        ; NtOpenProcess SSDT ID
                test    byte ptr ds:7FFE0308h, 1
                jnz     short loc_18009D4A5
                syscall                 ; Low latency system call
                retn
; ---------------------------------------------------------------------------
```

A disassembly of ntdll!OpenProcess().

Inside `NtOpenProcess()`, there's hardly any code at all. This is because like all functions beginning with Nt or Zw, `NtOpenProcess()` is actually located in the kernel. The ntdll (user mode) versions of these functions simply perform syscalls to call their kernel mode counterparts, which is why they're often referred to as system call stubs.

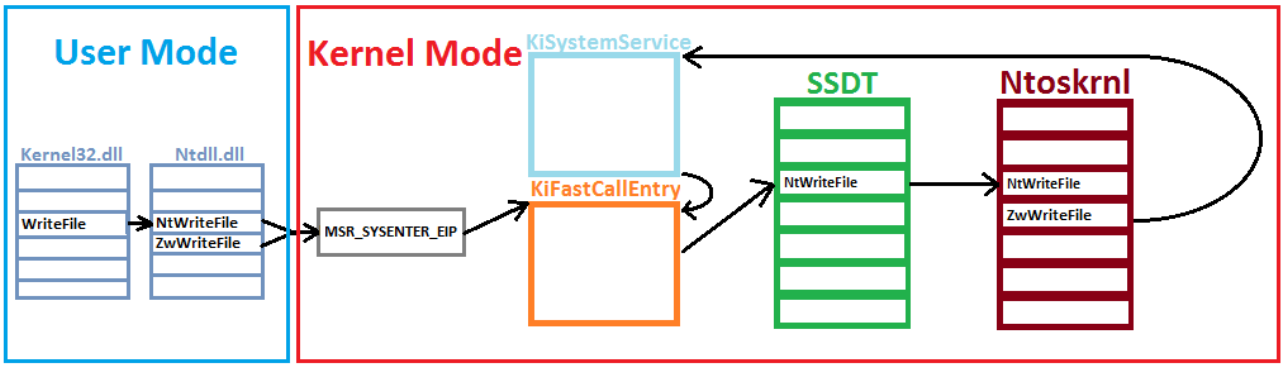
In our case, the SSN for NtOpenProcess is 0x26, but this number changes across Windows version so don't expect it to be the same for you.

From a simplified high-level view, the call flow looks somewhat like this:



A simplified x64 system call flow.

Here is a more detailed overview of an x86 system call flow from a previous article.

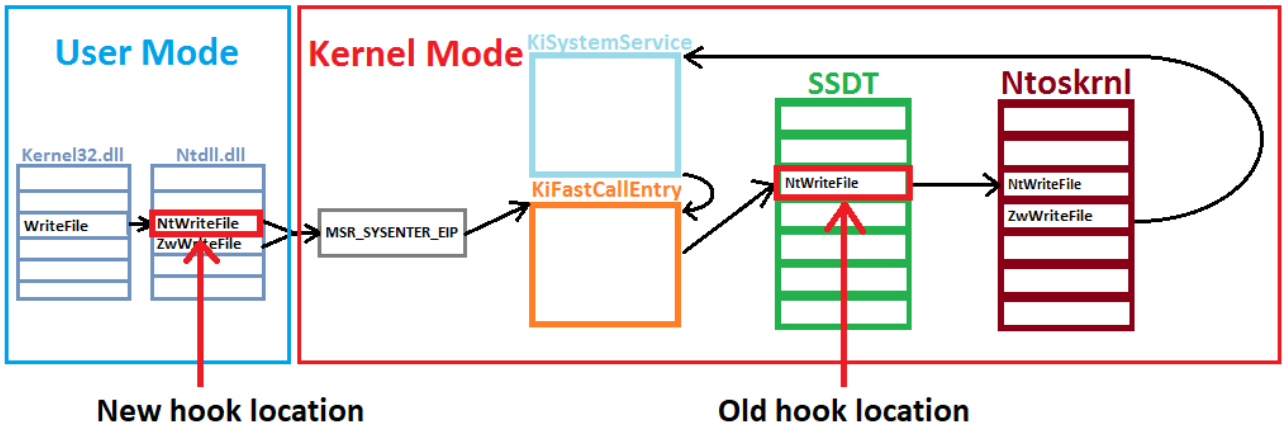


A more detailed x86 system call flow.

Note: from user mode, both the Nt and Zw version of a function are identical. From kernel mode the Zw function takes a slightly different path. This is due to the fact that Nt functions are designed to be called from user mode, therefore do more extensive validation of function parameters.

EDRs and User Mode Hooking

Since Microsoft introduced Kernel Patch Protection (aka PatchGuard) in 2005, many modifications to the kernel are now prevented. Previously, security products monitored user mode calls from inside the kernel by hooking the SSDT. Since all Nt/Zw functions are implemented in the kernel, all user mode calls must go through the SSDT, and are therefore subject to SSDT hooks. Patch guard makes SSDT hooking off-limits, so many EDRs resorted to hooking ntdll.



A look at where security products place hooks before and after patch guard.

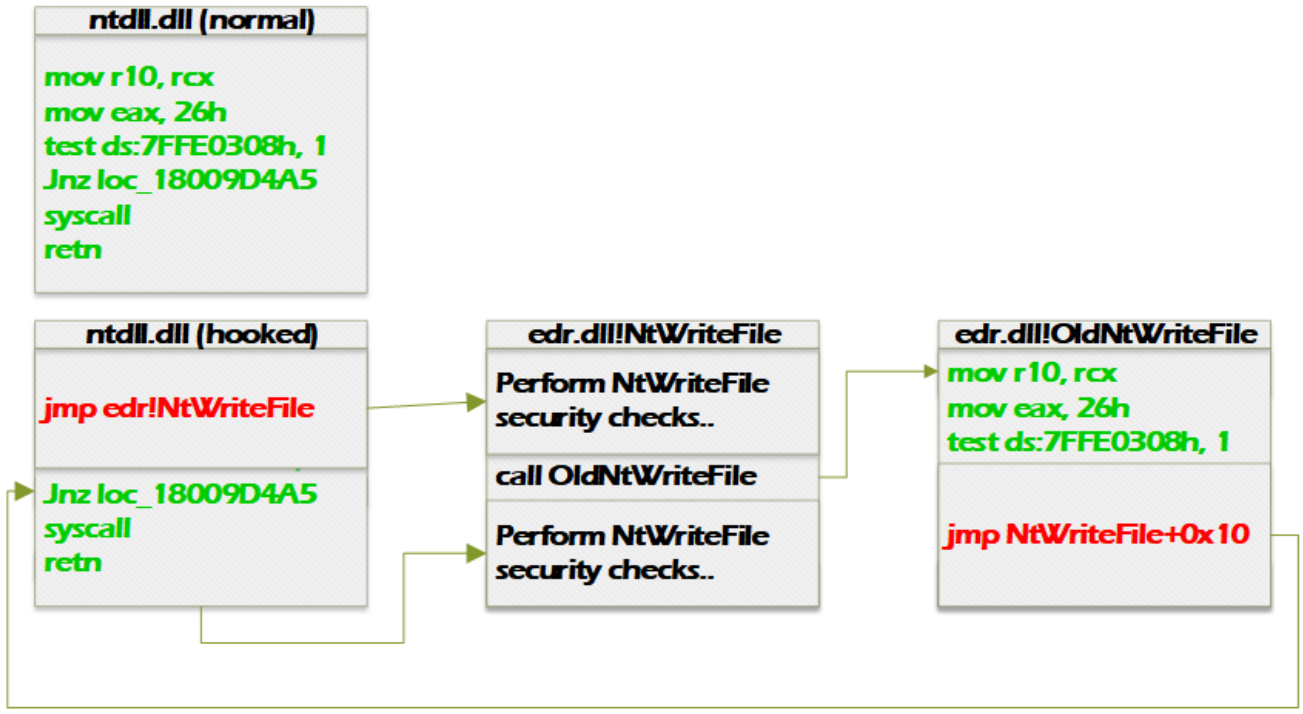
Since the SSDT exists in the kernel, user mode applications were not able to interfere with these hooks without loading a kernel driver. Now, the hooks are placed in user mode, alongside the application.

So, what does a user mode hook look like?

An example of a ntdll function before and after hooking.

To hook a function in ntdll.dll, most EDRs just overwrite the first 5 bytes of the function's code with a jmp instruction. The jmp instruction will redirect code execution to some code within the EDR's own DLL (which is automatically loaded into every process). After the CPU has

been redirected to the EDR's DLL, the EDR can perform security checks by inspecting the function parameters and return address. Once the EDR is done, it can resume the ntdll call by executing the overwritten instructions, then jumping to the location in ntdll right after the hook (jmp instruction).



Control flow example for hooked ntdll function.

In the above example, NtWriteFile is hooked. The green instructions are the original instructions from NtWriteFile. The first 3 instructions of NtWriteFile have been overwritten by the EDR's hook (a jmp that redirects execution to a function named NtWriteFile in edr.dll). Whenever the EDR wants to call the real NtWriteFile, it executes the 3 overwritten instructions, then jumps to the 4th instruction of the hooked function to complete the syscall.

Whilst EDR hooks may vary slightly from vendor to vendor, the principal is still the same, and all share the same weakness: they're located in user mode. Since both the hooks and the EDR's DLL have to be placed inside every process's address space, a malicious process can tamper with them.

Bypassing EDR Hooks

There are a multitude of ways to bypass EDR hooks, so I'll cover just the main ones.

EDR Unhooking

Since the hooked ntdll is located in our own process's memory, we can use `VirtualProtect()` to make the memory writable, then overwrite the EDR's jmp instruction with the original function code. In order to replace the hooks, we'll of course need to know what the original assembly instructions were. The most common way to do this is by reading the ntdll.dll file from disk, then comparing the in-memory version against the disk version. This assumes the EDR doesn't detect or prevent manually reading ntdll.dll from disk.

The main drawback of this method is that the EDR could just periodically check the memory of ntdll to see if its hooks have been removed. If the EDR detects its hooks have been removed, it may write them back, or worse, terminate the process and trigger a detection event. While the hooks may need to be placed in user mode, checking them can be done from kernel mode, so there's not much we could do to prevent it.

Manually Mapping DLLs

Instead of reading a clean copy of ntdll from disk to enable us to unhook the original ntdll, we could just load the clean copy into our proccess's memory and use that instead of the original. Since functions like `LoadLibrary()` and `LdrLoadDll()` don't allow the system to load the same DLL twice, we have to load it manually. The code for manually mapping DLLs can be extensive and also prone to errors or detection.

DLLs often also perform calls to other DLLs, so we'll either be restricted to only using functions from our manually loaded ntdll, or loading a second copy of every DLL we need and patching them to only make use of other manually loaded DLLs, which can get pretty messy. There's also a good chance of detection if an antivirus does a memory scan and sees multiple copies of every DLL loaded into memory.

Direct Syscalls

As discussed earlier, user mode Nt/Zw functions don't actually do anything other than execute syscalls. So we don't really need to map an entire new copy of ntdll just to do some syscalls. Instead, we can implement the syscall logic directly into our own code. All we need to do is move the SSN for the function we want to call into the eax register, then execute the syscall instruction.

It's as simple as

```
__asm {
  mov r10, rcx
  mov eax, 0x123
  syscall
  ret
}
```

Unfortunately, because the EDR's hook typically overwrites the instruction that sets the eax register, we can't simply just extract it from the hooked function. But…there's a few ways we can find out what it is.

Reading a clean copy of ntdll

You're probably bored with this idea by now, but we could just read a clean copy of ntdll from disk and extract the SSNs from there. Since the SSN is always put into the eax register, all we need to do is scan the function we want to call for the `mov eax, imm32` instruction. But, what if we want a method that isn't just some variation of reading ntdll from disk? Well, do not fear!

Calculating the system call number based on function order

System call ids are indexes, and therefore sequential. If the SSN for the function we want to call is 0x18, then the one directly before it will likely be 0x17 and the one directly after, 0x19. Since the EDR doesn't hook every Nt function, we can simply grab the SSN from the nearest non-hooked function, then calculate the one we want by adding or subtracting how many functions are between it and our target function.

```
ntdll!NtQueryValueKey:
00007ffb`5f6ad2b0 4c8bd1          mov     r10,rcx
00007ffb`5f6ad2b3 b817000000      mov     eax,17h
00007ffb`5f6ad2b8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffb`5f6ad2c0 7503            jne     ntdll!NtQueryValueKey+0x15 (00007ffb`5f6ad2c5)
00007ffb`5f6ad2c2 0f05            syscall
00007ffb`5f6ad2c4 c3              ret
ntdll!NtAllocateVirtualMemory:
00007ffb`5f6ad2d0 e9c13c2cc4      jmp     00007ffb`23970f96  I wonder what this could possibly be?
00007ffb`5f6ad2e0 7503            jne     ntdll!NtAllocateVirtualMemory+0x15 (00007ffb`5f6ad2e5)
00007ffb`5f6ad2e2 0f05            syscall
00007ffb`5f6ad2e4 c3              ret
ntdll!NtQueryInformationProcess:
00007ffb`5f6ad2f0 4c8bd1          mov     r10,rcx
00007ffb`5f6ad2f3 b819000000      mov     eax,19h
00007ffb`5f6ad2f8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffb`5f6ad300 7503            jne     ntdll!NtQueryInformationProcess+0x15 (00007ffb`5f6ad305)
00007ffb`5f6ad302 0f05            syscall
00007ffb`5f6ad304 c3              ret
```

NtAllocateVirtualMemory is hooked by the EDR, but the function before and after it are not. The function before it is system call number 0x17, and the function after it is 0x19. We can easily assume that the SSN we want is 0x18.

This method does have one flaw though: we can't 100% guarantee system call numbers will remain sequential forever, or the DLL won't skip a few.

Hardcoding

The simplest method of all, is to just hard code the system call numbers. Whilst they do change from version to version, they haven't changed a huge amount in the past. It's not too much work to detect the OS version and load the correct SSN set. In fact, j00ru has kindly

published a list of every system call number for every Windows version. The only drawback of this method is the code many not automatically work on new Windows version if the system call numbers change.

The problem with direct syscalls

Direct syscalls have been the go to for bypassing user mode hooks for over a decade. I actually first experimented with this method myself way back in 2012. Unfortunately, much work has been done to try and prevent this kind of bypass. The most common detection is by having the EDR's kernel mode driver inspect the callstack.

Although the EDR can no longer hook a lot of places in the kernel, it can use monitoring functionality provided by the operating system, such as:

- ETW events
- Kernel Callbacks
- Filter Drivers

If we perform a manual syscall, and somewhere along the way the kernel function we call hits any of the above, the EDR could take the opportunity to inspect the callstack of our thread. By unwinding the call stack and inspecting return addresses, the EDR can see the entire chain of function calls that led to this syscall.

If we were to perform a normal call to, say, `kernel32!VirtualAlloc()`, the callstack may look like so:

```
1: kd> k
 # Child-SP          RetAddr               Call Site
00 ffff870f`33e6fa88 fffff803`1b211235     nt!NtAllocateVirtualMemory
01 ffff870f`33e6fa90 00007ffc`b788d2e4     nt!KiSystemServiceCopyEnd+0x25
02 000000a0`f9affa78 00007ffc`b4c92316     ntdll!NtAllocateVirtualMemory+0x14
03 000000a0`f9affe20 00007ff6`554f10c3     KERNELBASE!VirtualAlloc+0x48
04 000000a0`f9affe60 00007ff6`554f13a0     ManualSyscall!main+0x53
05 (Inline Function) --------`--------     ManualSyscall!invoke_main+0x22
06 000000a0`f9affec0 00007ffc`b5d27344     ManualSyscall!__scrt_common_main_seh+0x10c
07 000000a0`f9afff00 00007ffc`b78426b1     KERNEL32!BaseThreadInitThunk+0x14
08 000000a0`f9afff30 00000000`00000000     ntdll!RtlUserThreadStart+0x21
```

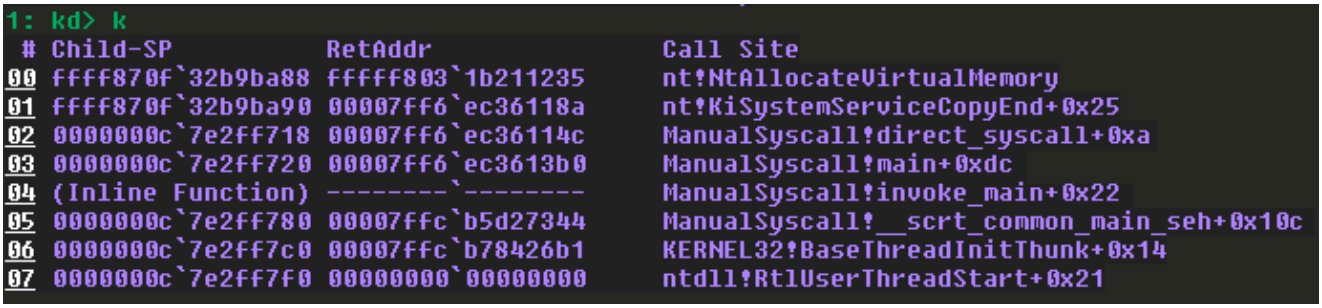
The callstack of a call to VirtualAlloc().

In this case the call to `VirtualAlloc()` is initiated by ManualSyscall!main+0x53. The relevant parts of the callstack in order of call are:

1. ManualSyscall!main+0x53
2. KERNELBASE!VirtualAlloc+0x48
3. ntdll!NtAllocateVirtualMemory+0x14
4. nt!KiSystemServiceCopyEnd+0x25

This tells us (or the EDR) that the executable (ManualSyscall.exe) called `VirtualAlloc()`, which called `NtAllocateVirtualMemory()`, which then performed a system call to transition into kernel mode.

Now let's look at the call stack when we do a direct syscall:



the callstack for a direct syscall to NtAllocateVirtualMemory().

The relevant parts of this callstack in order are:

1. ManualSyscall!direct_syscall+0xa
2. nt!KiSystemServiceCopyEnd+0x25

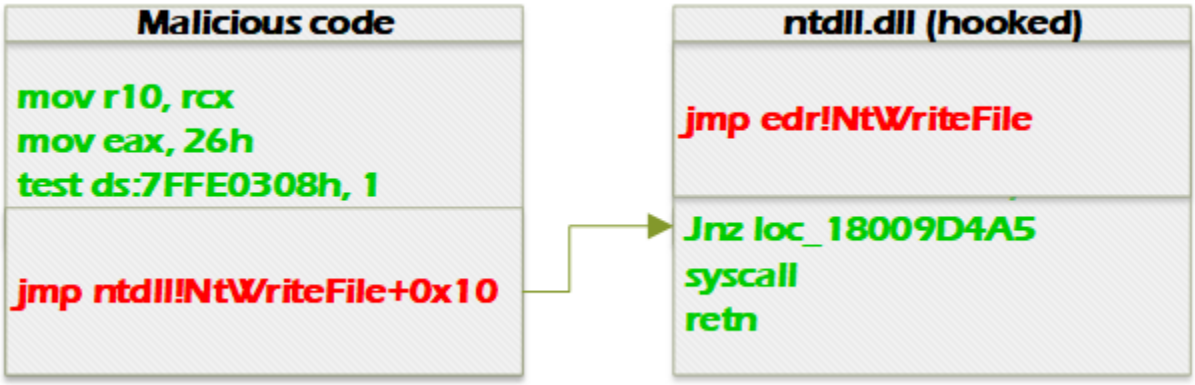
Here it's clear the kernel transition was triggered by code inside ManualSyscall.exe and not ntdll. But, what's the problem with this?

Well, on systems like linux it's completely normal for application to initiate system calls directly. But remember that I mentioned system call ids change between Windows versions? As a result it's highly impractical to write Windows software that relies on direct system calls. Due to the fact ntdll already implements every system call for you, there's almost no reason to do a manual syscall. Unless, of course, you're writing malware to bypass EDR hooks. Are you writing malware to bypass EDR hooks?

Because direct system calls are such a strong indicator of malicious activity, more sophisticated EDRs will log detections for system call that originated outside ntdll. Truth be told, you can still get away with it a lot of the time, but where's the fun in that?

Indirect syscalls

Most EDRs write their hooks at the start of the Nt function, overwriting the SSN but leaving the syscall instruction intact. This allows us to utilize the syscall instructions already provided by ntdll instead of bringing our own. We can just set up the r10 and eax registers ourselves, then jump to the syscall instruction inside the hooked ntdll function (which comes after the EDR's hook).



Code flow for an indirect syscall.

Note: we don't strictly need the test or jnz instruction, these are just there for backwards compatibility. Some ancient CPUs don't support the syscall instruction and use `int 0x2e` instead. The test instruction checks if syscalls are enabled, and if not, falls back to software interrupts. If we wish to support these systems, we could just perform the check ourselves, then jump to the int 0x2e instruction (which is also located within Nt function) if needed.

Just like with direct syscalls, we still need the system call number to put into eax, but we can use all the same techniques previously detailed in the direct syscalls section.

Setting up a system call this way will give us a call stack that look like the following:

```
1: kd> k
 # Child-SP          RetAddr               Call Site
00 ffff870f`346c7a88 fffff803`1b211235     nt!NtAllocateVirtualMemory
01 ffff870f`346c7a90 00007ffc`b788d2e4     nt!KiSystemServiceCopyEnd+0x25
02 00000078`9b2ffb48 00007ff7`2491114c     ntdll!NtAllocateVirtualMemory+0x14
03 00000078`9b2ffb50 00007ff7`249113c0     ManualSyscall!main+0xdc
04 (Inline Function) --------`--------     ManualSyscall!invoke_main+0x22
05 00000078`9b2ffbb0 00007ffc`b5d27344     ManualSyscall!__scrt_common_main_seh+0x10c
06 00000078`9b2ffbf0 00007ffc`b78426b1     KERNEL32!BaseThreadInitThunk+0x14
07 00000078`9b2ffc20 00000000`00000000     ntdll!RtlUserThreadStart+0x21
```

The call stack for an indirect system call.

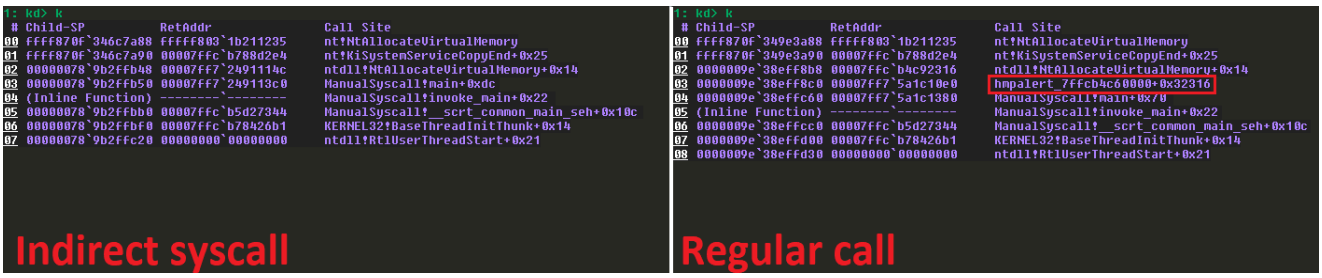
As you can see, the call stack now look as if the call came from the `ntdll!NtAllocateVirtualMemory()` insted of our executable, because technically it did.

One issue we could run into is if the EDR hooks or overwrites the syscall instruction part of the Nt call. I've never seen this happen, but it could in theory. In this case, we could jump to a syscall instruction inside a different, non-hooked, Nt function. This would still bypass EDRs which only validate that the call name from ntdll, but would fail any checks verifying that the kernel function called matches the ntdll function the syscall came from.

The larger problem is, what if the EDR checks more than just the first return address. Not just where the syscall came from, but who called the function that executed the syscall. If we're doing indirect syscalls from some shellcode located in dynamically allocated memory,

then the EDR is going to see that. Calls coming from outside a valid PE section (exe or DLL memory) are fairly suspicious.

Furthermore, since the function is hooked by the EDR, the EDR's hook would be expected to appear in the call stack. I'm actually not sure which EDRs, if any, check this. But, as you can see here it's clear from the call stack that we bypassed the EDR hook.



The EDR's hook is visible in the call stack from a regular call but not from an indirect syscall.

Ideally, we want to fake more than just the system call return address. An interesting solution to this is call stack spoofing, which I'm probably going to cover in a separate article. With call stack spoofing it's possible to fake the entire call stack, but it can be challenging to implement without crashing the calling thread.

You can read more about call stack spoofing here:
Spoofing Call Stacks To Confuse EDRs - WithSecure
Long Live Custom Call Stacks - DarkVortex

That's All, For Now

so there you have it, you now understand the basics of user mode EDR hooks and some common techniques to bypass them. This knowledge will be important for my next article, which will go deeper into how EDR hooks work and detail some alternative methods of bypassing them.

Part 2: https://malwaretech.com/2023/12/silly-edr-bypasses-and-where-to-find-them.html