Signed kernel drivers – Unguarded gateway to Windows’ core

welivesecurity.com/2022/01/11/signed-kernel-drivers-unguarded-gateway-windows-core

January 11, 2022

There are various types of kernel drivers; the first that come to mind are device drivers that provide a software interface to hardware devices like *plug and play* interfaces or *filter drivers*. These low-level system components have a strict development process including scrutiny regarding security. However, there are additional “software” drivers that are designed to run in Ring 0 and provide specific, non-hardware related features like software debugging and diagnostics, system analysis, etc. As you can see below, these are prone to extend the attack surface significantly.

While directly loading a malicious, unsigned driver is no longer possible in the newer versions of Windows (unless driver signature enforcement is explicitly disabled during boot) and kernel rootkits are considered to be a thing of the past, there are still ways to load malicious code into the kernel. While actual vulnerabilities and exploits that achieve that get a lot of attention, there is a much easier way: abusing legitimate, signed drivers. There are many drivers from various hardware and software vendors lying around that offer functionality to fully access the kernel with minimal effort.

Vulnerabilities in signed drivers are mostly utilized by game cheat developers to circumvent anti-cheat mechanisms, but they have also been observed being used by several APT groups and in commodity malware alike.

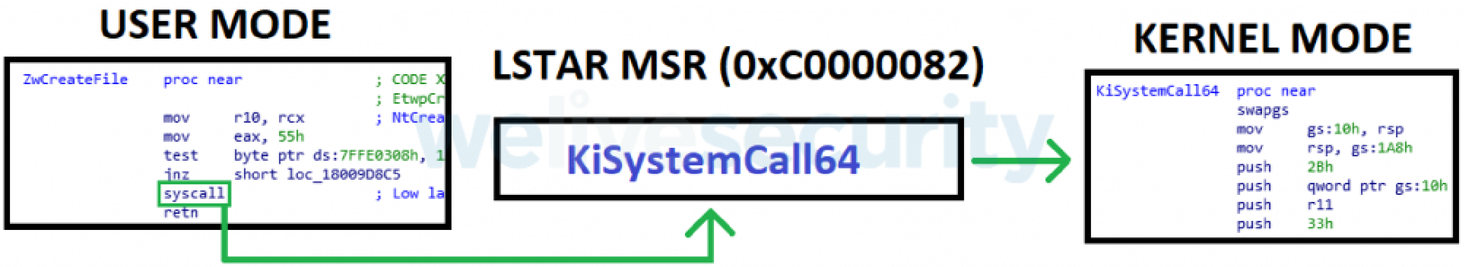
This paper discusses the types of vulnerabilities that commonly occur in kernel drivers, provides several case studies of malware utilizing such vulnerable drivers, analyzes examples of vulnerable drivers that we discovered during our research, and outlines effective mitigation techniques against this type of exploitation. While this problem is not new and relevant research about the topic has been presented in the past, mainly during 2018 and 2019 ([1], [2], [3]), it is still a problem as of this writing.

Common types of driver vulnerabilities

While every vulnerability is different, similar types of vulnerabilities seem to be recurrent in unrelated kernel drivers. This may be partially caused by *(ancient) driver code samples* that were created back when access to kernel mode was not restricted to signed drivers and developers did not take security into consideration (malware could simply load unsigned rootkit drivers instead). The following sections describe the vulnerabilities most frequently observed in drivers from a large variety of, and even high-profile, hardware and software vendors.

MSR read/write

Model-specific registers (MSRs) were introduced in Pentium 80586 CPUs in 1993. MSRs can be thought of as “global variables” of a CPU (or of a specific core). Some contain various information about the processor or specific CPU core – such as temperature, power, …. Additionally, there are also many MSRs that contain data critical for the working of a system, such as IA32_LSTAR (0xC0000082) for SYSCALL or IA32_SYSENTER_EIP (0x00000176) for SYSENTER, both of which contain pointers to an address in the kernel where the CPU jumps when a SYSCALL or SYSENTER instruction is executed. On newer Windows x64 platforms such as Windows 10 or 11, SYSCALL is used for both AMD and Intel CPUs where IA32_LSTAR should point to the KiSystemCall64 function found in ntoskrnl.exe. The mechanism of the transition to Windows kernel when executing SYSCALL is displayed in Figure 1.

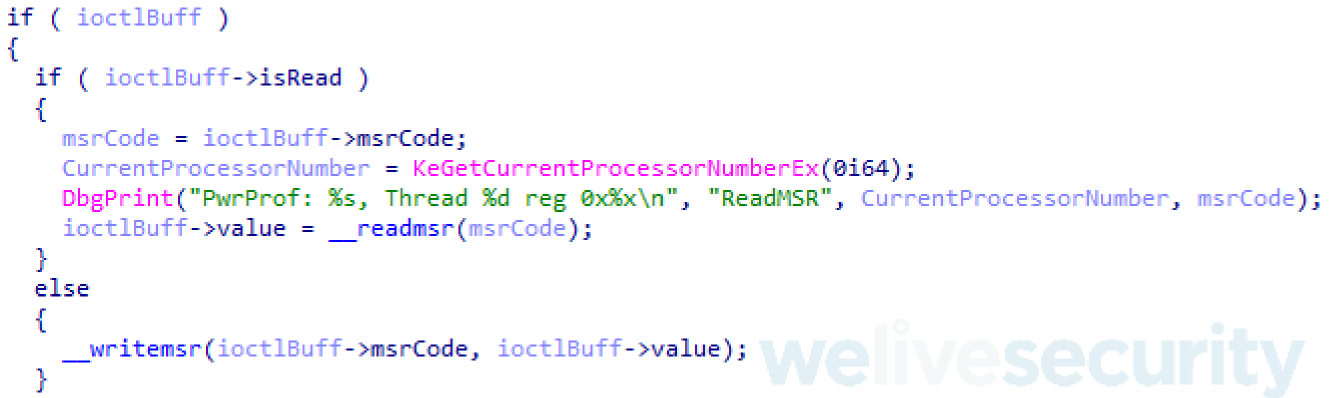


```
ZwCreateFile    proc near                       ; CODE X
                                                ; EtwpCr
                mov     r10, rcx                ; NtCrea
                mov     eax, 55h
                test    byte ptr ds:7FFE0308h, 1
                jnz     short loc_18009D8C5
                syscall                         ; Low la
                retn
```

```
KiSystemCall64  proc near
                swapgs
                mov     gs:10h, rsp
                mov     rsp, gs:1A8h
                push    2Bh
                push    qword ptr gs:10h
                push    r11
                push    33h
```

Figure 1. How SYSCALL is handled in x64 Windows

MSRs are indexed by a number and accessed by the privileged *RDMSR* and *WRMSR* instructions, which can only be executed in kernel mode. Many commercial drivers implement functionality for user-mode applications to access these instructions through an IOCTL mechanism. This is usually intended to be able to read or write a few specific innocent MSRs (like CPU voltage, temperature, …), but developers sometimes do not add any additional checks to restrict access to critical MSRs, such as the example seen in Figure 2. This gives potential attackers an opportunity to, for example, patch the SYSCALL/SYSENTER entry point MSRs, which are pointers to a function that handles any system call from user mode.



```
if ( ioctlBuff )
{
  if ( ioctlBuff->isRead )
  {
    msrCode = ioctlBuff->msrCode;
    CurrentProcessorNumber = KeGetCurrentProcessorNumberEx(0i64);
    DbgPrint("PwrProf: %s, Thread %d reg 0x%x\n", "ReadMSR", CurrentProcessorNumber, msrCode);
    ioctlBuff->value = __readmsr(msrCode);
  }
  else
  {
    __writemsr(ioctlBuff->msrCode, ioctlBuff->value);
  }
```

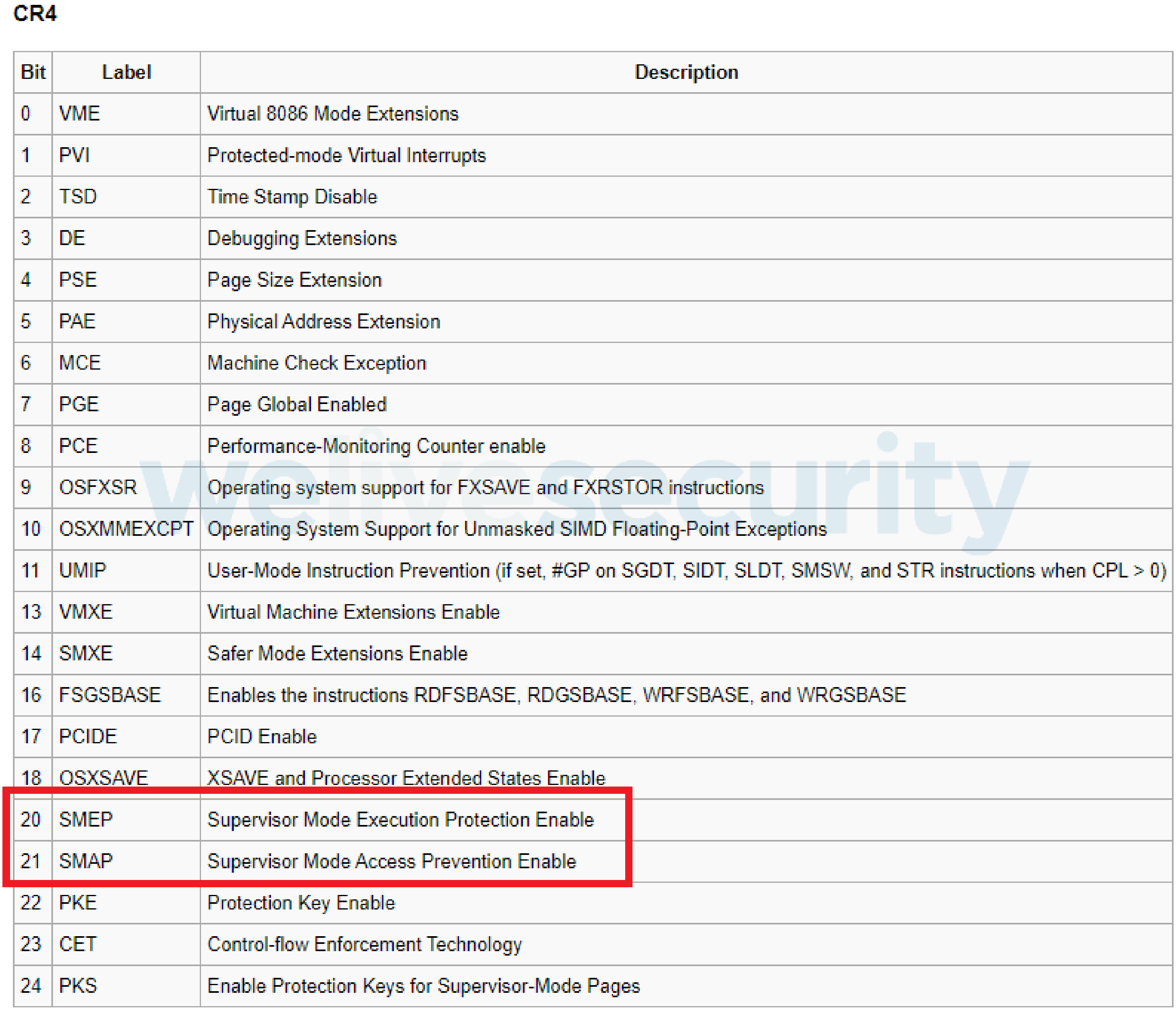
Figure 2. Vulnerable MSR IOCTL handler in the AMDPowerProfiler.sys driver

Overwriting the system call pointer was trivial and very powerful on older CPUs and systems before mitigations like Supervisor Mode Execution Prevention (SMEP) were introduced. On such systems, simply changing the pointer to the address of an arbitrary user-mode executable buffer containing malicious code, and then immediately executing a system call instruction on a same CPU core, was enough to gain kernel-level code execution. This is no longer the case with newer systems due to modern exploitation mitigations. That being said, with clever use of various techniques, it is still possible to bypass most of these mitigations and achieve kernel-level code execution on Windows 10 or even brand-new Windows 11 systems (as of December 2021).

All the mitigations in the following sections are in place on most modern machines and need to be bypassed to achieve successful kernel-mode exploitation.

SMEP

SMEP is a protection mechanism introduced in 2011 in Intel processors based on the Ivy Bridge architecture and enabled by default since Windows 8.0. It prevents execution of code in user-mode pages from Ring 0, and is implemented by assigning a user-mode or kernel-mode value to a flag bit on every virtual memory page in the page table. If a system attempts to execute code in a user-mode page from kernel space, a 0x000000FC error (ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY) will be triggered and cause a BSOD. SMEP can be dynamically toggled on and off during execution with its status saved in the CR4 register for each CPU core individually (see Figure 3).



CR4

Bit	Label	Description
0	VME	Virtual 8086 Mode Extensions
1	PVI	Protected-mode Virtual Interrupts
2	TSD	Time Stamp Disable
3	DE	Debugging Extensions
4	PSE	Page Size Extension
5	PAE	Physical Address Extension
6	MCE	Machine Check Exception
7	PGE	Page Global Enabled
8	PCE	Performance-Monitoring Counter enable
9	OSFXSR	Operating system support for FXSAVE and FXRSTOR instructions
10	OSXMMEXCPT	Operating System Support for Unmasked SIMD Floating-Point Exceptions
11	UMIP	User-Mode Instruction Prevention (if set, #GP on SGDT, SIDT, SLDT, SMSW, and STR instructions when CPL > 0)
13	VMXE	Virtual Machine Extensions Enable
14	SMXE	Safer Mode Extensions Enable
16	FSGSBASE	Enables the instructions RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE
17	PCIDE	PCID Enable
18	OSXSAVE	XSAVE and Processor Extended States Enable
20	SMEP	Supervisor Mode Execution Protection Enable
21	SMAP	Supervisor Mode Access Prevention Enable
22	PKE	Protection Key Enable
23	CET	Control-flow Enforcement Technology
24	PKS	Enable Protection Keys for Supervisor-Mode Pages

Figure 3. CR4 register flags of a CPU (image credit: Wikipedia)

SMEP mitigates the naïve exploitation technique of abusing an MSR R/W IOCTL in order to change the LSTAR MSR to point directly to malicious user-mode code. That being said, since the attacker is in control of the stack that is passed to kernel mode on system calls, they may utilize a technique called a *ROP chain* to manipulate the stack. By placing a chain of return addresses on the stack, the attacker can arrange to execute a carefully picked set of instructions in kernel mode by changing the LSTAR MSR via a vulnerable IOCTL. With the stack suitably prepared, executing the system call instruction results in the first “gadget” in the ROP chain executing and when complete its code will “return” to the next gadget in the chain, which was supplied on the stack with the rest of the chain.

The functionality of such a ROP chain is limited by the availability of suitable code chunks, called gadgets, available in the kernel’s modules. Since an attacker can read kernel modules from the file system and knows at which addresses the modules are loaded, the gadgets can be easily looked up and if those gadgets exist, a working ROP chain can then be constructed.

To properly initialize the transition to kernel, the ROP chain needs to swap the GS register using the SWAPGS instruction. On 64-bit Windows, the GS register holds the address of the Thread Environment Block (TEB) in user mode, and the address of the Kernel Processor Control Region (KPCR) in kernel mode. Therefore, it is crucial that those two addresses change before any operations happen in the kernel. It is no surprise that the SWAPGS instruction is also the first instruction in the Windows kernel KiSystemCall64 function.

The next step is restoring the original value of the LSTAR MSR using the WRMSR instruction, in order to avoid executing the ROP chain on the next execution of the SYSCALL instruction.

At this point, the malicious code properly executes in the kernel and the attacker can execute whatever payload is desired. This can be additional ROP gadgets to, for example, disable SMEP or SMAP protections by overwriting CR4, or even be direct calls to an exported ntoskrnl API function.

The attacker can overwrite CR4 utilizing a MOV CR4 gadget to disable SMEP and also SMAP, which is covered in the next section. They then can proceed to execute a user-mode payload directly. The only difficulty with this approach is precalculating a valid CR4 value. Although most of the CR4 values can be guessed from user mode by running the CPUID instruction, there may be some inconsistencies between different versions of Windows.

To transition back to user mode safely, once the attacker’s ROP chain has run they need to execute the SWAPGS instruction again and then execute the SYSRET instruction.

Older exploits bypassing SMEP are described in [4], (2015).

SMAP

Supervisor Mode Access Prevention (SMAP) is a newer mitigation that has been introduced to complement SMEP and further restrict access from the kernel to user-mode pages – it disallows both reads and writes. Just as SMEP, its status is stored as a bit in the CR4 register (see Figure 3).

SMAP should render the previously described ROP chain technique useless, since the stack containing the ROP chain is in fact a user-mode page. A system with SMAP active will bluescreen the moment it tries to access the stack after transitioning to the kernel via the system call.

SMAP can also be temporarily disabled by setting the AC flag in the EFLAGS CPU register. This feature is also the downfall of this mitigation in regard to MSR exploitation – it turns out the AC flag can be set from user mode, right before transitioning to kernel mode, by utilizing the POPF and PUSHF instructions. This is caused by the SFMASK MSR that controls which EFLAGS register bits are cleared when the

SYSCALL instruction is executed. Even on the newest Windows 11 machines, the mask does not have the AC flag bit set, which means it is not cleared upon transitioning to the kernel so SMAP can be disabled by the user.

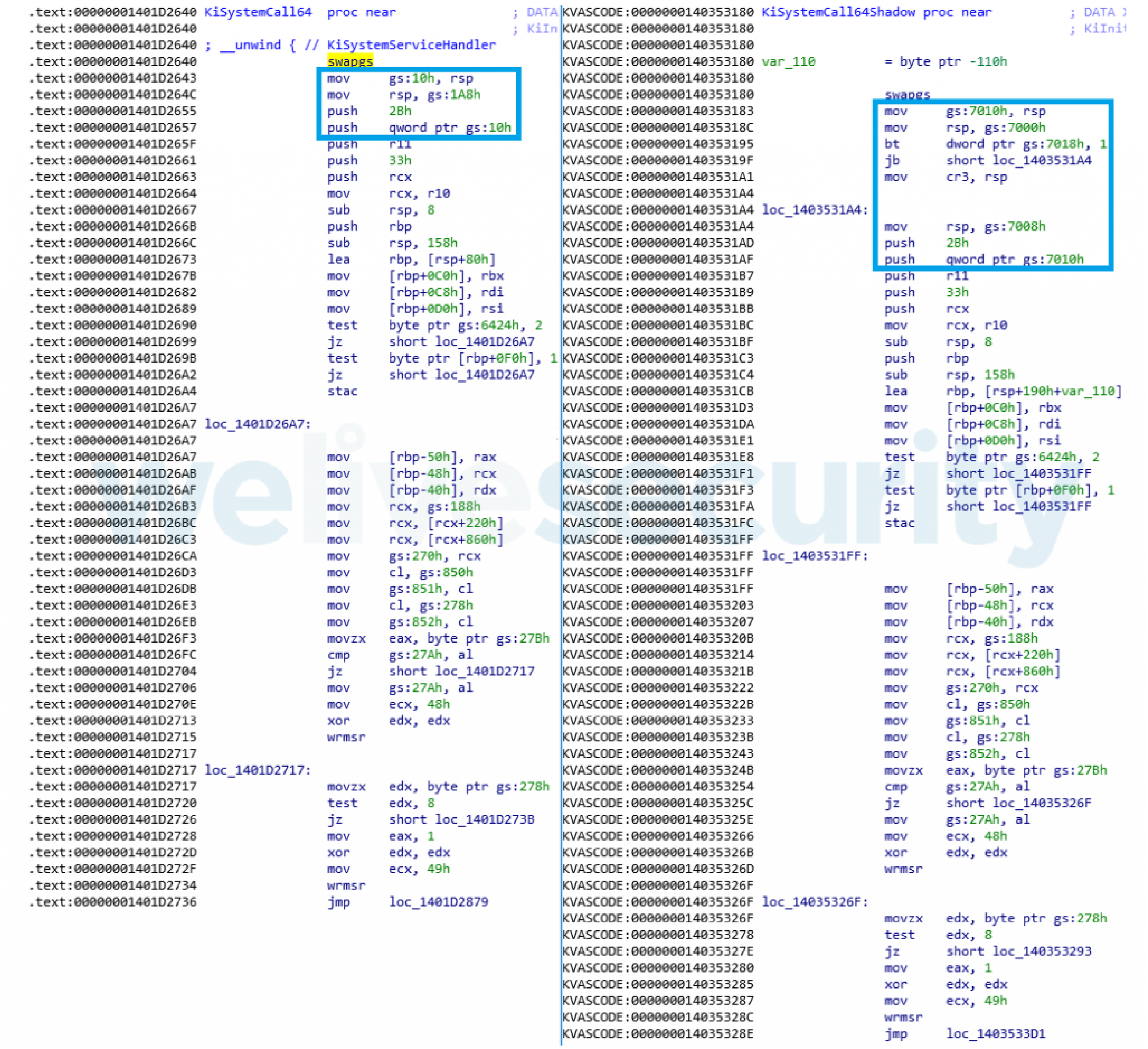
As the SFMASK is controlled by another MSR (0xC0000084), even if Microsoft changed SFMASK to implicitly clear the AC flag, theoretically the attacker could patch this MSR prior to the exploitation anyway.

It is worth noting that SMAP has only recently been enabled by default in Windows 10 x64 with newer hardware.

KVA shadowing

KVA shadowing was introduced as a software mitigation for the *Meltdown* CPU vulnerability discovered at the end of 2017.

The basic idea of this mitigation is that the virtual address space is split into two – user mode and kernel mode. The user-mode address space has access only to very restricted parts of the ntoskrnl module, specifically a single code section called .KVASCODE that is responsible for low-level operations like entering and leaving the kernel when handling a system call. This is handled by implementing “Shadow” equivalents of the responsible functions, like KiSystemCall64Shadow that works as the original KiSystemCall64, but contains differences responsible for handling KVA shadowing and switching the address space context properly (see Figure 4). The rest of the kernel is completely separated and mapped to its own address space and cannot be accessed even directly by the CPU from user-mode address space until the context is appropriately switched.



```
.text:00000001401D2640 KiSystemCall64  proc near               ; DATA
.text:00000001401D2640                                         ; KiIn
.text:00000001401D2640 ; __unwind { // KiSystemServiceHandler
.text:00000001401D2640                 swapgs
.text:00000001401D2643                 mov     gs:10h, rsp
.text:00000001401D264C                 mov     rsp, gs:1A8h
.text:00000001401D2655                 push    2Bh
.text:00000001401D2657                 push    qword ptr gs:10h
.text:00000001401D265F                 push    r11
.text:00000001401D2661                 push    33h
.text:00000001401D2663                 push    rcx
.text:00000001401D2664                 mov     rcx, r10
.text:00000001401D2667                 sub     rsp, 8
.text:00000001401D266B                 push    rbp
.text:00000001401D266C                 sub     rsp, 158h
.text:00000001401D2673                 lea     rbp, [rsp+80h]
.text:00000001401D267B                 mov     [rbp+0C0h], rbx
.text:00000001401D2682                 mov     [rbp+0C8h], rdi
.text:00000001401D2689                 mov     [rbp+0D0h], rsi
.text:00000001401D2690                 test    byte ptr gs:6424h, 2
.text:00000001401D2699                 jz      short loc_1401D26A7
.text:00000001401D269B                 test    byte ptr [rbp+0F0h], 1
.text:00000001401D26A2                 jz      short loc_1401D26A7
.text:00000001401D26A4                 stac
.text:00000001401D26A7
.text:00000001401D26A7 loc_1401D26A7:
.text:00000001401D26A7
.text:00000001401D26A7                 mov     [rbp-50h], rax
.text:00000001401D26AB                 mov     [rbp-48h], rcx
.text:00000001401D26AF                 mov     [rbp-40h], rdx
.text:00000001401D26B3                 mov     rcx, gs:188h
.text:00000001401D26BC                 mov     rcx, [rcx+220h]
.text:00000001401D26C3                 mov     rcx, [rcx+860h]
.text:00000001401D26CA                 mov     gs:270h, rcx
.text:00000001401D26D3                 mov     cl, gs:850h
.text:00000001401D26DB                 mov     gs:851h, cl
.text:00000001401D26E3                 mov     cl, gs:278h
.text:00000001401D26EB                 mov     gs:852h, cl
.text:00000001401D26F3                 movzx   eax, byte ptr gs:27Bh
.text:00000001401D26FC                 cmp     gs:27Ah, al
.text:00000001401D2704                 jz      short loc_1401D2717
.text:00000001401D2706                 mov     gs:27Ah, al
.text:00000001401D270E                 mov     ecx, 48h
.text:00000001401D2713                 xor     edx, edx
.text:00000001401D2715                 wrmsr
.text:00000001401D2717
.text:00000001401D2717 loc_1401D2717:
.text:00000001401D2717                 movzx   edx, byte ptr gs:278h
.text:00000001401D2720                 test    edx, 8
.text:00000001401D2726                 jz      short loc_1401D273B
.text:00000001401D2728                 mov     eax, 1
.text:00000001401D272D                 xor     edx, edx
.text:00000001401D272F                 mov     ecx, 49h
.text:00000001401D2734                 wrmsr
.text:00000001401D2736                 jmp     loc_1401D2879
```

```
KVASCODE:0000000140353180 KiSystemCall64Shadow proc near          ; DATA
KVASCODE:0000000140353180                                         ; KiInit
KVASCODE:0000000140353180
KVASCODE:0000000140353180 var_110         = byte ptr -110h
KVASCODE:0000000140353180
KVASCODE:0000000140353180                 swapgs
KVASCODE:0000000140353183                 mov     gs:7010h, rsp
KVASCODE:000000014035318C                 mov     rsp, gs:7000h
KVASCODE:0000000140353195                 bt      dword ptr gs:7018h, 1
KVASCODE:000000014035319F                 jb      short loc_1403531A4
KVASCODE:00000001403531A1                 mov     cr3, rsp
KVASCODE:00000001403531A4
KVASCODE:00000001403531A4 loc_1403531A4:
KVASCODE:00000001403531A4                 mov     rsp, gs:7008h
KVASCODE:00000001403531AD                 push    2Bh
KVASCODE:00000001403531AF                 push    qword ptr gs:7010h
KVASCODE:00000001403531B7                 push    r11
KVASCODE:00000001403531B9                 push    33h
KVASCODE:00000001403531BB                 push    rcx
KVASCODE:00000001403531BC                 mov     rcx, r10
KVASCODE:00000001403531BF                 sub     rsp, 8
KVASCODE:00000001403531C3                 push    rbp
KVASCODE:00000001403531C4                 sub     rsp, 158h
KVASCODE:00000001403531CB                 lea     rbp, [rsp+190h+var_110]
KVASCODE:00000001403531D3                 mov     [rbp+0C0h], rbx
KVASCODE:00000001403531DA                 mov     [rbp+0C8h], rdi
KVASCODE:00000001403531E1                 mov     [rbp+0D0h], rsi
KVASCODE:00000001403531E8                 test    byte ptr gs:6424h, 2
KVASCODE:00000001403531F1                 jz      short loc_1403531FF
KVASCODE:00000001403531F3                 test    byte ptr [rbp+0F0h], 1
KVASCODE:00000001403531FA                 jz      short loc_1403531FF
KVASCODE:00000001403531FC                 stac
KVASCODE:00000001403531FF
KVASCODE:00000001403531FF loc_1403531FF:
KVASCODE:00000001403531FF
KVASCODE:00000001403531FF                 mov     [rbp-50h], rax
KVASCODE:0000000140353203                 mov     [rbp-48h], rcx
KVASCODE:0000000140353207                 mov     [rbp-40h], rdx
KVASCODE:000000014035320B                 mov     rcx, gs:188h
KVASCODE:0000000140353214                 mov     rcx, [rcx+220h]
KVASCODE:000000014035321B                 mov     rcx, [rcx+860h]
KVASCODE:0000000140353222                 mov     gs:270h, rcx
KVASCODE:000000014035322B                 mov     cl, gs:850h
KVASCODE:0000000140353233                 mov     gs:851h, cl
KVASCODE:000000014035323B                 mov     cl, gs:278h
KVASCODE:0000000140353243                 mov     gs:852h, cl
KVASCODE:000000014035324B                 movzx   eax, byte ptr gs:27Bh
KVASCODE:0000000140353254                 cmp     gs:27Ah, al
KVASCODE:000000014035325C                 jz      short loc_14035326F
KVASCODE:000000014035325E                 mov     gs:27Ah, al
KVASCODE:0000000140353266                 mov     ecx, 48h
KVASCODE:000000014035326B                 xor     edx, edx
KVASCODE:000000014035326D                 wrmsr
KVASCODE:000000014035326F
KVASCODE:000000014035326F loc_14035326F:
KVASCODE:000000014035326F                 movzx   edx, byte ptr gs:278h
KVASCODE:0000000140353278                 test    edx, 8
KVASCODE:000000014035327E                 jz      short loc_140353293
KVASCODE:0000000140353280                 mov     eax, 1
KVASCODE:0000000140353285                 xor     edx, edx
KVASCODE:0000000140353287                 mov     ecx, 49h
KVASCODE:000000014035328C                 wrmsr
KVASCODE:000000014035328E                 jmp     loc_1403533D1
```

Figure 4. Comparison of KiSystemCall64 and KiSystemCall64Shadow versions of the system call handler – minor differences can be spotted at the beginning of the function

While KVA shadowing was designed as a fix for the Meltdown vulnerability, it also potentially causes trouble for other kinds of vulnerabilities, including the MSR one.

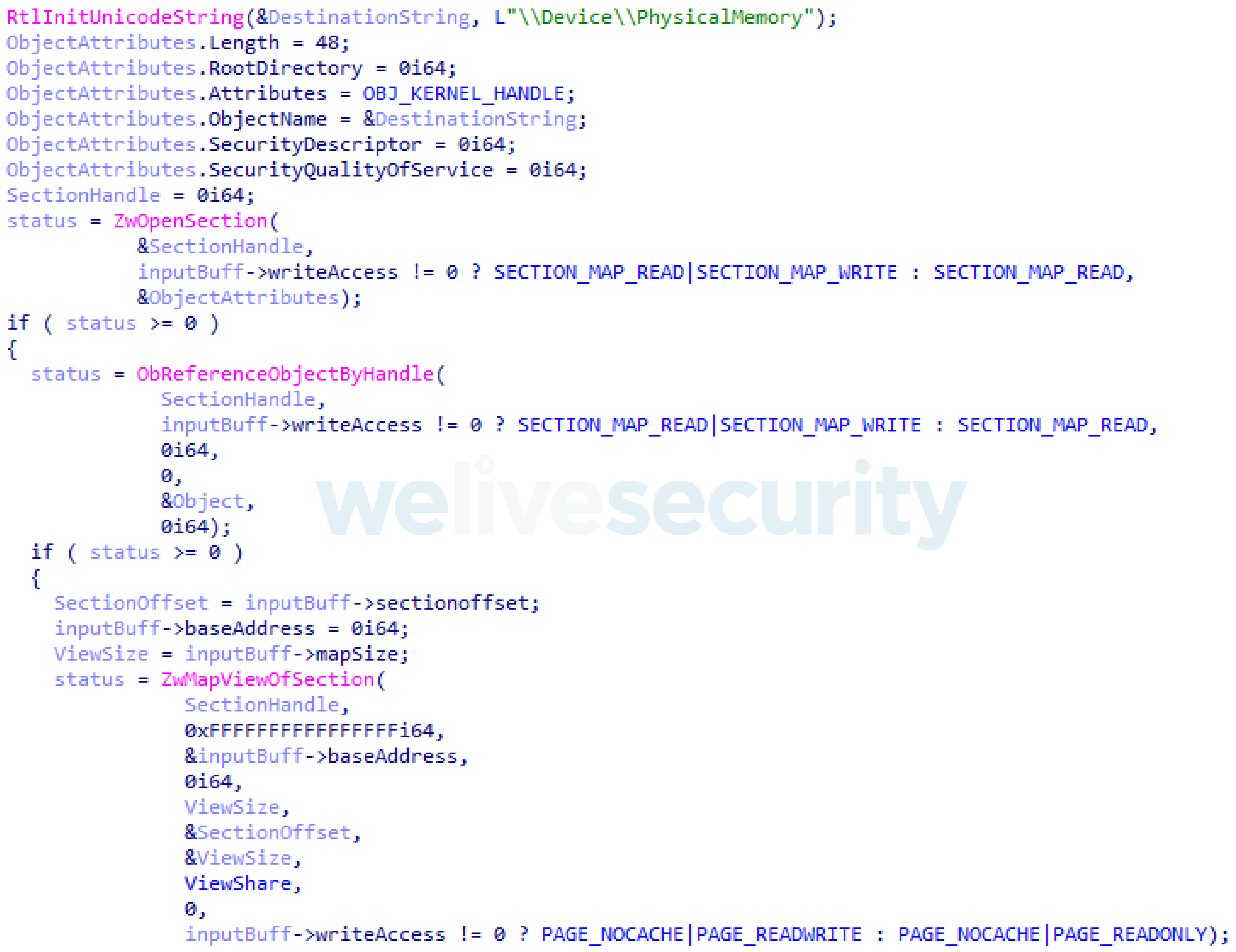
There are generally two approaches to disable the mitigation – one is to disable it as a setting in the registry. This requires admin access and a reboot afterwards for the changes to take effect.

Alternatively, when building a ROP chain for MSR exploitation, an attacker tries to find gadgets exclusively in the .KVASCODE section of the ntoskrnl module – since that section handles the system call transition, it is possible to build a working ROP chain.

Similar mitigation was also introduced in Linux systems, where it is called Kernel Page Table Isolation (KPTI).

Physical memory read/write

Being able to directly read and write physical memory seems to be a common feature in many low-level kernel drivers. This is achieved by mapping a specific range of physical memory to a virtual memory buffer that can be read or written and even passed to a user-mode application. There are several ways to achieve this, the most common one being an ability to map the \Device\PhysicalMemory section to virtual memory, as shown in Figure 5.



```
RtlInitUnicodeString(&DestinationString, L"\\Device\\PhysicalMemory");
ObjectAttributes.Length = 48;
ObjectAttributes.RootDirectory = 0i64;
ObjectAttributes.Attributes = OBJ_KERNEL_HANDLE;
ObjectAttributes.ObjectName = &DestinationString;
ObjectAttributes.SecurityDescriptor = 0i64;
ObjectAttributes.SecurityQualityOfService = 0i64;
SectionHandle = 0i64;
status = ZwOpenSection(
           &SectionHandle,
           inputBuff->writeAccess != 0 ? SECTION_MAP_READ|SECTION_MAP_WRITE : SECTION_MAP_READ,
           &ObjectAttributes);
if ( status >= 0 )
{
  status = ObReferenceObjectByHandle(
             SectionHandle,
             inputBuff->writeAccess != 0 ? SECTION_MAP_READ|SECTION_MAP_WRITE : SECTION_MAP_READ,
             0i64,
             0,
             &Object,
             0i64);
  if ( status >= 0 )
  {
    SectionOffset = inputBuff->sectionoffset;
    inputBuff->baseAddress = 0i64;
    ViewSize = inputBuff->mapSize;
    status = ZwMapViewOfSection(
               SectionHandle,
               0xFFFFFFFFFFFFFFFFi64,
               &inputBuff->baseAddress,
               0i64,
               ViewSize,
               &SectionOffset,
               &ViewSize,
               ViewShare,
               0,
               inputBuff->writeAccess != 0 ? PAGE_NOCACHE|PAGE_READWRITE : PAGE_NOCACHE|PAGE_READONLY);
```

Figure 5. Physical memory map vulnerability in Passmark DirectIO64.sys driver

A potential drawback for the attackers is that they first need to translate the virtual address to a physical one. Drivers that implement physical memory I/O sometimes also offer an IOCTL for physical to virtual address translation, but even if the driver does not have any such address conversion, there are still many ways to utilize this feature.

The most straightforward use case is simply to walk through all the physical memory looking for specific artifacts that represent critical data structures that the attacker wants to find. For example, the attacker might try to look for the EPROCESS structure of the malicious process and elevate it to SYSTEM privileges by stealing a token from a more privileged process or modifying its rights. Some of these strategies are demonstrated here and here.

Since physical memory mappings disregard any virtual memory protection features, it is also possible to write to executable memory pages. This gives the attacker an opportunity to look up specific kernel modules and chunks of code, carefully modify them and, if patched code can be executed via a system API or an IOCTL of a driver, to achieve malicious kernel-level code execution.

Virtual memory read/write

Virtual memory access IOCTLs are not as commonly found in these drivers as physical memory ones, but they have very similar repercussions. Utilizing these is even easier, as there is no address translation needed and all the virtual kernel addresses found from user mode can be accessed directly. A potential downside is that the access is limited by the memory protection of the target address, so it is not possible to write read-only memory pages without changing the protection first.

Therefore, this vulnerability is commonly used to manipulate various kernel data structures to achieve things like elevating a malicious process to SYSTEM rights by stealing tokens from such kernel structures.

The most common way this vulnerability arises is via an IOCTL with a simple pointer dereference in kernel mode, so it can be hard to detect this vulnerability using heuristic methods.

Case studies

When malware actors need to run malicious code in the Windows kernel on x64 systems with driver signature enforcement (DSE) in place, carrying a vulnerable signed kernel driver seems to be a viable option for doing so. This technique is known as Bring Your Own Vulnerable Driver (BYOVD) and has been observed being used in the wild by both high-profile APT actors and in commodity malware.

In the following sections we present some examples.

Slingshot APT

Slingshot is a cyberespionage platform that was uncovered by Kaspersky in 2018 [5] and is believed to have been active since at least 2012. The actors behind this malware decided to implement their main module, called Cahnadr, as a kernel-mode driver. On older x86 systems, the driver would be loaded directly by the user-mode module. On newer systems with active DSE, they decided to implement a custom driver loader that leverages the following signed kernel drivers with MSR vulnerabilities: *Goad*, SpeedFan (*CVE-2007-5633*), Sandra (*CVE-2010-1592*), and ElbyCDIO (*CVE-2009-0824*). The exploitation targeted pre-Windows 8 systems, so the exploitation was a simple modification of the LSTAR MSR to point to a malicious payload in a user-mode buffer.

Note that the Kaspersky researchers estimated these threat actors to have been active from 2012 to 2018, which means that these exploits were quite old and well known, but that was no reason for the attackers to stop using them as the certificates of those vulnerable drivers were never revoked.

InvisiMole

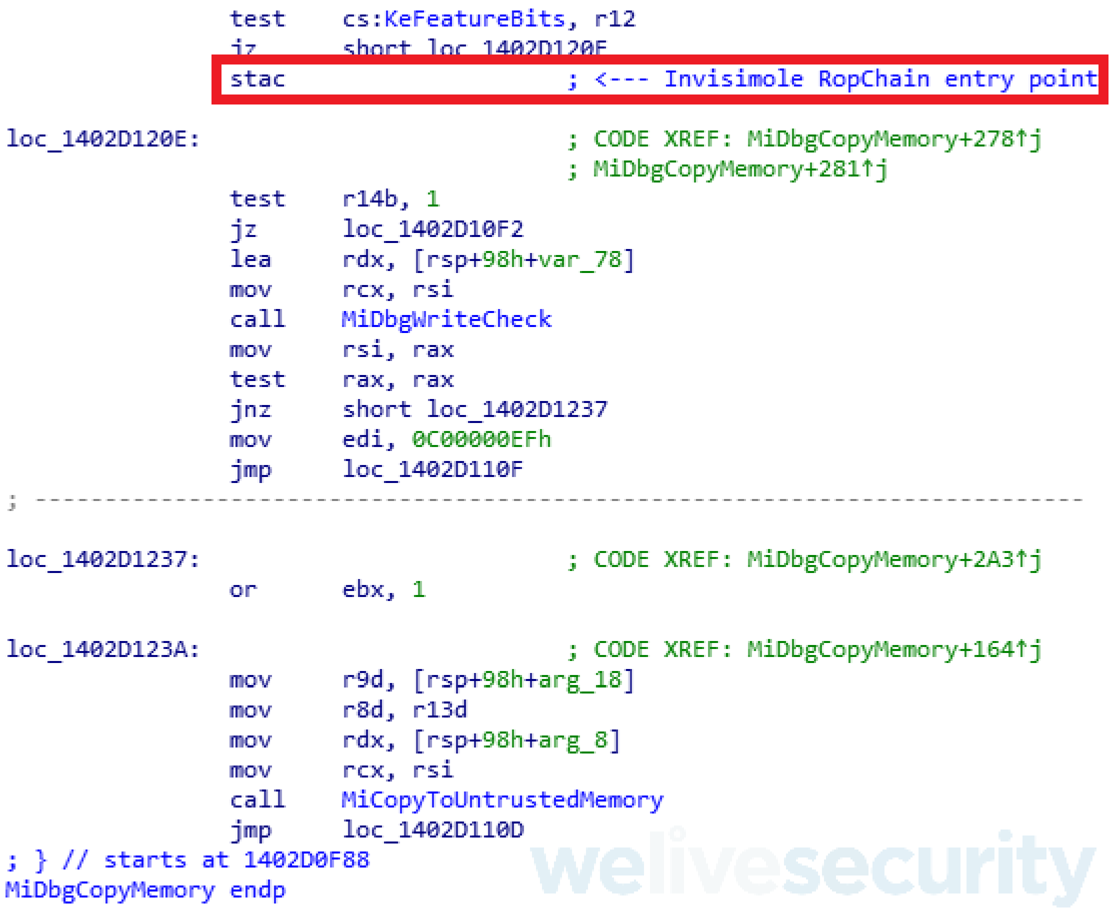
In 2018, fellow ESET researchers uncovered a sophisticated APT actor that they named InvisiMole [6]. The group has been tracked by ESET ever since and in 2020 an extensive *white paper* about the group and its toolset was published. In that white paper, our colleagues reported that InvisiMole used the BYOVD technique, exploiting the MSR vulnerability in the speedfan.sys driver (CVE-2007-5633) to load a malicious unsigned driver. While this campaign was targeting older x86 systems and the exploitation using a malicious driver was trivial from the modern standpoint, due to the fact there were no mitigations like SMEP in place, it was still an interesting case showing that the group behind this malware is very technically capable.

Later during that investigation, however, a newer variant of the InvisiMole malware using the BYOVD technique was discovered. This variant is the only case to date that we have observed of MSR exploitation on Windows 10 x64 systems being used in the wild by a malicious actor. It employs advanced techniques to bypass mitigations like SMEP and even SMAP. That being said, the exploitation is mitigated by KVA shadowing, which the authors failed to take care of. Coincidentally, MSR exploitation was used to deploy a malicious driver that attempted to disable our security products. Although the whole compromise chain is more extensive, we will focus on a specific part of the malware that leverages the BYOVD technique and MSR exploitation that happens in the main user-mode module.

User-mode module

InvisiMole's authors seem to have developed a sophisticated ROP chain exploitation framework that they use for MSR exploitation – although the sample contains many debug messages in the code, we were not able to identify and link them to any known projects. This leads us to believe the framework is an original work of these malware authors and that non-negligible resources have been spent on developing it. The framework is an extensive C++ codebase with various classes.

It appears that InvisiMole's authors did not know about the possibility of setting the AC flag with the PUSHF and POPF instructions as described in the *SMAP* section, and instead chose a very complex ROP gadget found in the MiDbgCopyMemory kernel function that starts with the privileged STAC instruction, which is dedicated to setting the AC flag (see Figure 6). On top of that, InvisiMole utilizes the IRETQ instruction after every gadget that explicitly sets the RFLAGS register with the AC flag set, which stabilizes the exploit even further.



```
                test    cs:KeFeatureBits, r12
                jz      short loc_1402D120E
                stac                            ; <--- Invisimole RopChain entry point

loc_1402D120E:                                  ; CODE XREF: MiDbgCopyMemory+278↑j
                                                ; MiDbgCopyMemory+281↑j
                test    r14b, 1
                jz      loc_1402D10F2
                lea     rdx, [rsp+98h+var_78]
                mov     rcx, rsi
                call    MiDbgWriteCheck
                mov     rsi, rax
                test    rax, rax
                jnz     short loc_1402D1237
                mov     edi, 0C00000EFh
                jmp     loc_1402D110F
; ---------------------------------------------------------------------------

loc_1402D1237:                                  ; CODE XREF: MiDbgCopyMemory+2A3↑j
                or      ebx, 1

loc_1402D123A:                                  ; CODE XREF: MiDbgCopyMemory+164↑j
                mov     r9d, [rsp+98h+arg_18]
                mov     r8d, r13d
                mov     rdx, [rsp+98h+arg_8]
                mov     rcx, rsi
                call    MiCopyToUntrustedMemory
                jmp     loc_1402D110D
; } // starts at 1402D0F88
MiDbgCopyMemory endp
```

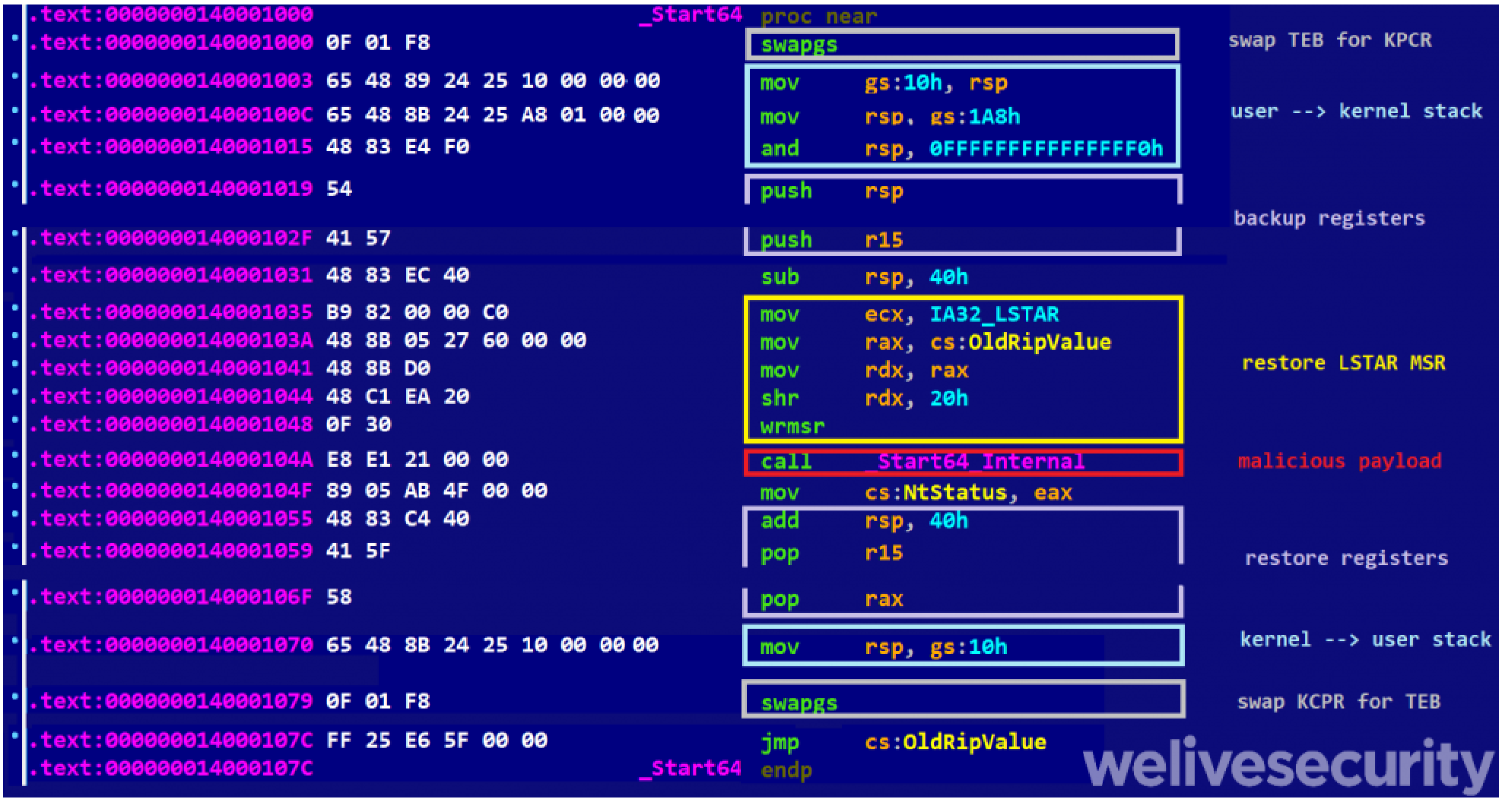
Figure 6. ROP gadget used by InvisiMole to disable SMAP mitigation

The initial gadget jumps directly to the STAC instruction, which immediately disables SMAP by setting the AC flag. Since this gadget appears in the middle of the MiDbgCopyMemory function, the malware carefully prepares the stack and registers to safely leave the function. Once the MiDbgCopyMemory function returns, the ROP chain proceeds to the SWAPGS gadget [7] in order to properly switch to kernel mode, followed by the WRMSR gadget to set the LSTAR MSR back to its original value. At this point, InvisiMole will proceed with executing the payload, which may be an exported kernel function or the entry point of a loaded malicious driver.

Driver loader

The driver loading technique is quite complex – InvisiMole will first install a "driver loader" – another kernel driver module that is used to load the malicious payload (yet another driver) passed as an argument. To initialize the driver loader, InvisiMole executes several separate MSR exploitations, where every instance carries a dedicated ROP chain with a single API call payload. The malware will start by executing ExAllocatePoolWithTag to allocate an executable kernel memory buffer for the loader, followed by preparing the image in user mode to reflect its future address in the kernel – sections are moved to their virtual offsets; imports are resolved, and relocations fixed. Once the image is ready, it is copied over from user mode to the allocated kernel buffer using memcpy from the ntoskrnl module.

To transfer code execution to the loader once it is copied to the kernel, InvisiMole's authors also leverage the MSR vulnerability and designed several dedicated PE exports in the loader (see Figure 7 for an example) that are intended to handle transitions from user-mode system calls. It works very similarly to the ROP chain gadgets – swap the GS register, swap the user-mode and kernel-mode stacks, save all registers on the stack, restore the original LSTAR MSR value and then call the actual function. Once finished, this process is reversed.



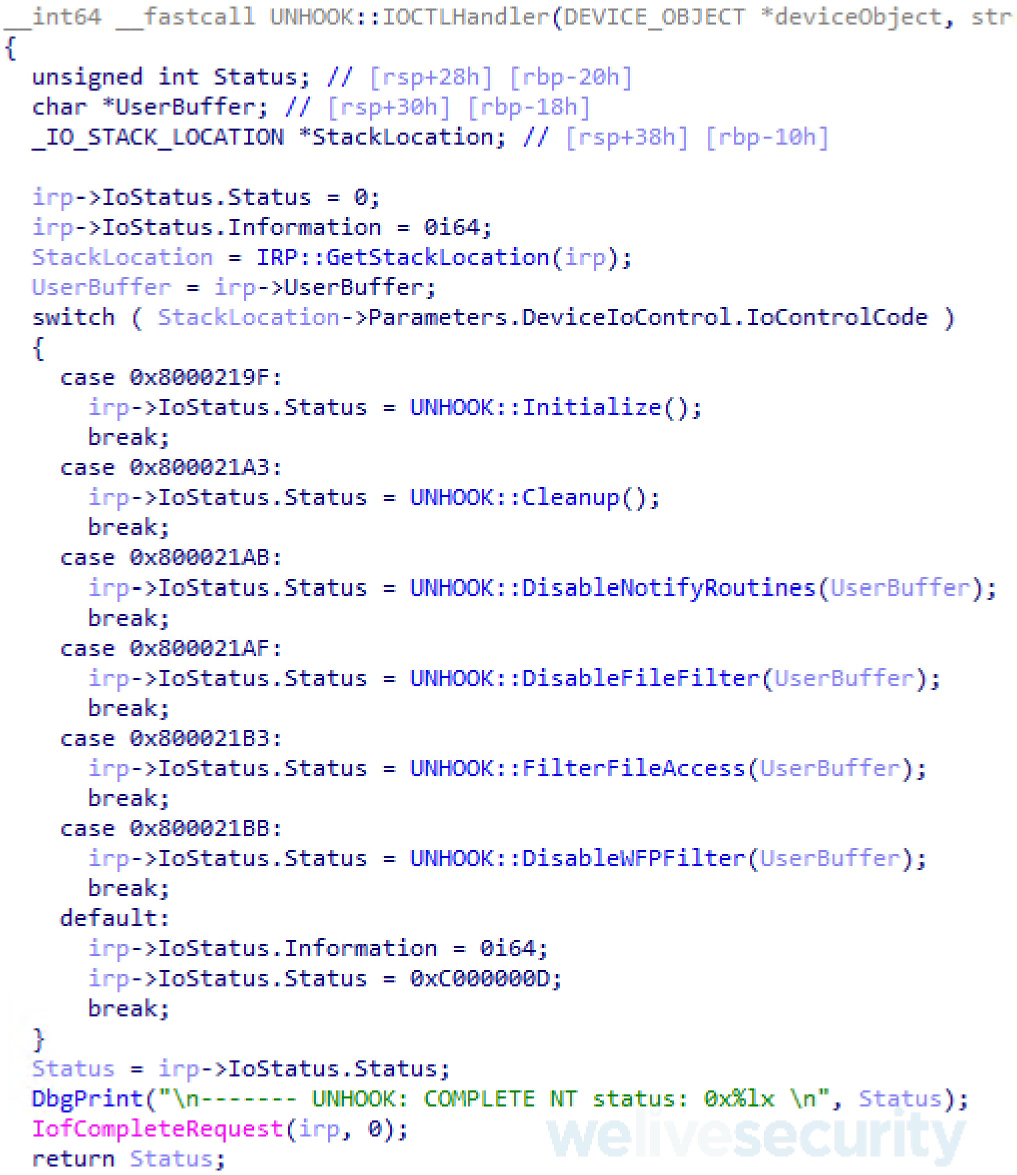
```
.text:0000000140001000                               _Start64 proc near
.text:0000000140001000 0F 01 F8                       swapgs
.text:0000000140001003 65 48 89 24 25 10 00 00 00     mov     gs:10h, rsp
.text:000000014000100C 65 48 8B 24 25 A8 01 00 00     mov     rsp, gs:1A8h
.text:0000000140001015 48 83 E4 F0                    and     rsp, 0FFFFFFFFFFFFFFF0h
.text:0000000140001019 54                             push    rsp
.text:000000014000102F 41 57                          push    r15
.text:0000000140001031 48 83 EC 40                    sub     rsp, 40h
.text:0000000140001035 B9 82 00 00 C0                 mov     ecx, IA32_LSTAR
.text:000000014000103A 48 8B 05 27 60 00 00           mov     rax, cs:OldRipValue
.text:0000000140001041 48 8B D0                       mov     rdx, rax
.text:0000000140001044 48 C1 EA 20                    shr     rdx, 20h
.text:0000000140001048 0F 30                          wrmsr
.text:000000014000104A E8 E1 21 00 00                 call    _Start64_Internal
.text:000000014000104F 89 05 AB 4F 00 00              mov     cs:NtStatus, eax
.text:0000000140001055 48 83 C4 40                    add     rsp, 40h
.text:0000000140001059 41 5F                          pop     r15
.text:000000014000106F 58                             pop     rax
.text:0000000140001070 65 48 8B 24 25 10 00 00 00     mov     rsp, gs:10h
.text:0000000140001079 0F 01 F8                       swapgs
.text:000000014000107C FF 25 E6 5F 00 00              jmp     cs:OldRipValue
.text:000000014000107C                               _Start64 endp
```

Figure 7. Exported function in the driver loader module that is called by changing LSTAR MSR

When the loader is properly initialized in the kernel, an export named _Start64 is executed by changing LSTAR MSR to the export address in the kernel. After handling the transition to the kernel, _Start64 registers a deferred routine that is responsible for loading the payload driver, and returns to user mode. The deferred loader routine will attempt to initialize the payload driver in a “proper” fashion – creating registry keys and kernel driver objects, performing all the necessary steps to register the driver in the system as if the operating system were loading the driver itself, and eventually calling IoCreateDriver. The proper initialization approach was chosen so the loaded payload driver can process I/O request packets and communicate with a user-mode module using IOCTLs.

Payload driver

The payload driver offers IOCTL functionality for disabling various notification callbacks (see Figure 8), mostly aimed at disarming third-party security solutions, and the possibility of protecting a file in the file system. The specific commands are passed from the user-mode module. Interestingly, the user-mode module will attempt to disable various parts of ESET protection in the kernel.



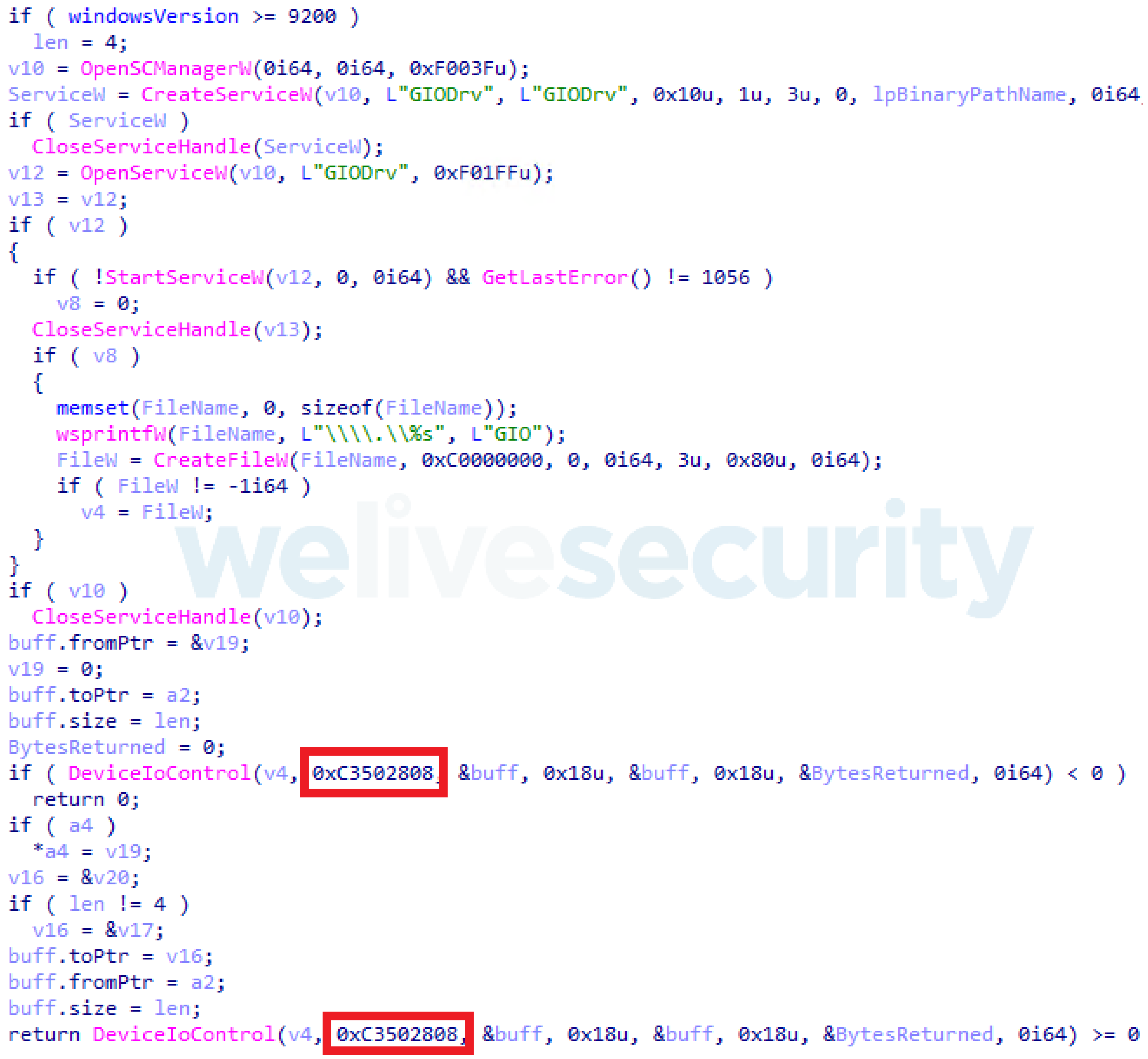
```
__int64 __fastcall UNHOOK::IOCTLHandler(DEVICE_OBJECT *deviceObject, str
{
  unsigned int Status; // [rsp+28h] [rbp-20h]
  char *UserBuffer; // [rsp+30h] [rbp-18h]
  _IO_STACK_LOCATION *StackLocation; // [rsp+38h] [rbp-10h]

  irp->IoStatus.Status = 0;
  irp->IoStatus.Information = 0i64;
  StackLocation = IRP::GetStackLocation(irp);
  UserBuffer = irp->UserBuffer;
  switch ( StackLocation->Parameters.DeviceIoControl.IoControlCode )
  {
    case 0x8000219F:
      irp->IoStatus.Status = UNHOOK::Initialize();
      break;
    case 0x800021A3:
      irp->IoStatus.Status = UNHOOK::Cleanup();
      break;
    case 0x800021AB:
      irp->IoStatus.Status = UNHOOK::DisableNotifyRoutines(UserBuffer);
      break;
    case 0x800021AF:
      irp->IoStatus.Status = UNHOOK::DisableFileFilter(UserBuffer);
      break;
    case 0x800021B3:
      irp->IoStatus.Status = UNHOOK::FilterFileAccess(UserBuffer);
      break;
    case 0x800021BB:
      irp->IoStatus.Status = UNHOOK::DisableWFPFilter(UserBuffer);
      break;
    default:
      irp->IoStatus.Information = 0i64;
      irp->IoStatus.Status = 0xC000000D;
      break;
  }
  Status = irp->IoStatus.Status;
  DbgPrint("\n------- UNHOOK: COMPLETE NT status: 0x%lx \n", Status);
  IofCompleteRequest(irp, 0);
  return Status;
```

Figure 8. IOCTL handler in the InvisiMole payload driver

RobbinHood

Seeing a BYOVD technique in commodity malware that aims to reach as many people as possible is rare, but the RobbinHood ransomware family shows that it may still prove useful [8].

This ransomware leverages a vulnerable GIGABYTE motherboard driver GDRV.SYS (*CVE-2018-19320*; see Figure 9 and Figure 10) to disable DSE in order to install its own malicious driver.

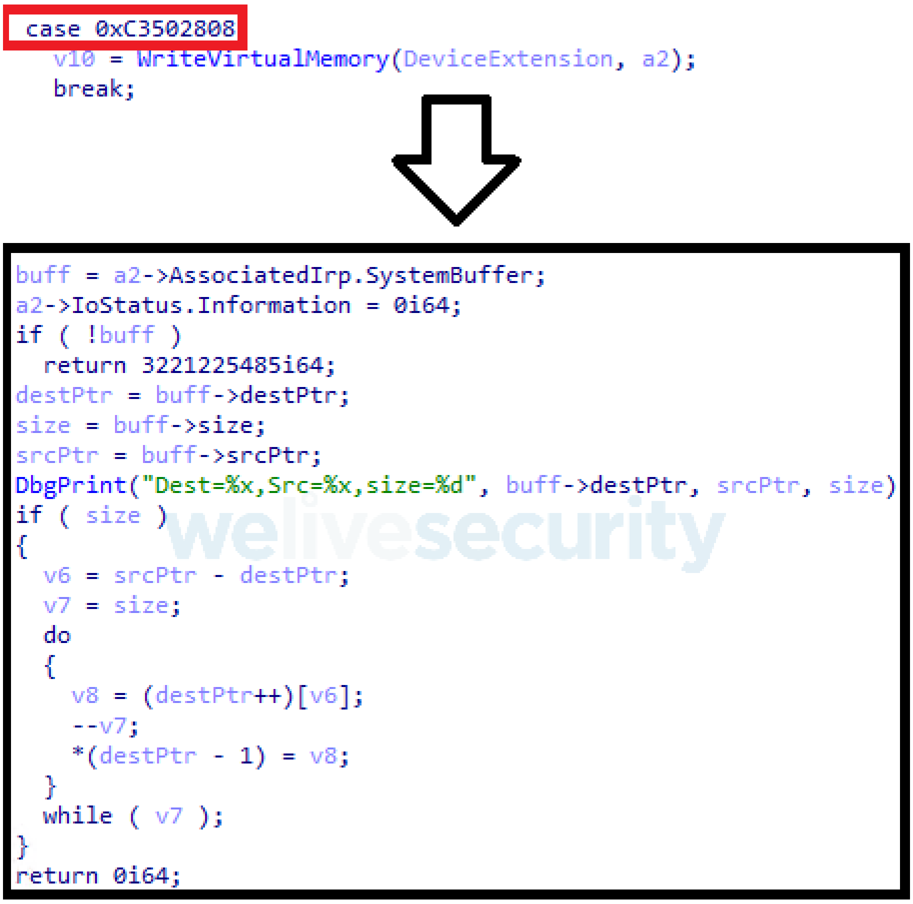


```
if ( windowsVersion >= 9200 )
  len = 4;
v10 = OpenSCManagerW(0i64, 0i64, 0xF003Fu);
ServiceW = CreateServiceW(v10, L"GIODrv", L"GIODrv", 0x10u, 1u, 3u, 0, lpBinaryPathName, 0i64
if ( ServiceW )
  CloseServiceHandle(ServiceW);
v12 = OpenServiceW(v10, L"GIODrv", 0xF01FFu);
v13 = v12;
if ( v12 )
{
  if ( !StartServiceW(v12, 0, 0i64) && GetLastError() != 1056 )
    v8 = 0;
  CloseServiceHandle(v13);
  if ( v8 )
  {
    memset(FileName, 0, sizeof(FileName));
    wsprintfW(FileName, L"\\\\.\\%s", L"GIO");
    FileW = CreateFileW(FileName, 0xC0000000, 0, 0i64, 3u, 0x80u, 0i64);
    if ( FileW != -1i64 )
      v4 = FileW;
  }
}
if ( v10 )
  CloseServiceHandle(v10);
buff.fromPtr = &v19;
v19 = 0;
buff.toPtr = a2;
buff.size = len;
BytesReturned = 0;
if ( DeviceIoControl(v4, 0xC3502808, &buff, 0x18u, &buff, 0x18u, &BytesReturned, 0i64) < 0 )
  return 0;
if ( a4 )
  *a4 = v19;
v16 = &v20;
if ( len != 4 )
  v16 = &v17;
buff.toPtr = v16;
buff.fromPtr = a2;
buff.size = len;
return DeviceIoControl(v4, 0xC3502808, &buff, 0x18u, &buff, 0x18u, &BytesReturned, 0i64) >= 0
```

Figure 9. GIODrv driver exploitation in RobbinHood sample

The way DSE is disabled depends on the Windows version – on Windows 7 and older, an nt!g_CiEnabled variable is modified directly in the ntoskrnl module. On newer Windows 8 through Windows 11 systems, the variable ci!g_CiOptions in the ci.dll module is modified instead. Finding this variable is slightly more complicated and it looks like the authors adopted a method found in an open-source project called *DSEFix* that is available on GitHub. Moreover, since Windows 8.1, the variables in ci.dll are protected by *PathcGuard* and tampering with the module will eventually cause the system to BSOD, even if the variable is changed back to an original value.

The malicious driver is then used to kill a long list of processes and delete their files, mainly focusing on endpoint protection software and other utilities. Since the termination is done from kernel mode, most self-protection mechanisms employed by security software are circumvented and the technique is more likely to succeed than attempting to disarm the protections from user mode.



```
case 0xC3502808
  v10 = WriteVirtualMemory(DeviceExtension, a2);
  break;
```

```
buff = a2->AssociatedIrp.SystemBuffer;
a2->IoStatus.Information = 0i64;
if ( !buff )
  return 3221225485i64;
destPtr = buff->destPtr;
size = buff->size;
srcPtr = buff->srcPtr;
DbgPrint("Dest=%x,Src=%x,size=%d", buff->destPtr, srcPtr, size)
if ( size )
{
  v6 = srcPtr - destPtr;
  v7 = size;
  do
  {
    v8 = (destPtr++)[v6];
    --v7;
    *(destPtr - 1) = v8;
  }
  while ( v7 );
}
return 0i64;
```

Figure 10. WriteVirtualMemory IOCTL handler in GIODrv

LoJax

In 2018, ESET researchers discovered the first-ever UEFI rootkit used in the wild. In order to have access to victims' UEFI modules, the malware utilizes a powerful utility called RWEverything.

The RWEverything driver was recently disabled by Microsoft directly in Windows 10 and 11 using the HVCI memory integrity feature described in the *Virtualization-based security* section.

Discovered vulnerabilities

During our research we decided not only to catalog existing vulnerabilities, but also to look for new ones. We set up YARA rules to hunt for kernel drivers with specific functionality and indicators of being potentially vulnerable. We also created a proof-of-concept exploitation framework for testing the newly found drivers and confirming that they are exploitable.

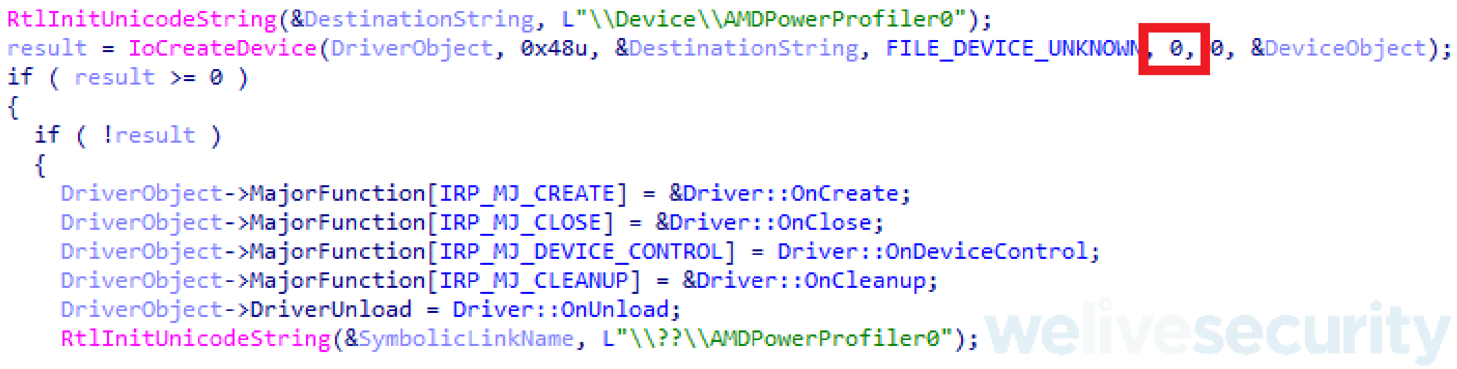
We went through hundreds of different kernel drivers that matched our criteria and aside from finding the already discovered drivers, we also found three drivers previously not known to be vulnerable, some containing several unrelated bugs. The fact that even after several independent research groups tackled this area we were still able to find new vulnerabilities, even from reputable vendors, shows that Windows driver security is still an issue.

While we were looking for all kinds of vulnerabilities described in previous sections, finding ones with MSR access turned out to be the easiest, appearing most commonly due to the use of special privileged instructions (RDMSR/WRMSR) that give away this functionality. Interestingly enough, in many cases it would turn out that this class of drivers also contained other kinds of vulnerabilities like arbitrary physical or virtual memory read and write functions.

AMD µProf (CVE-2021-26334)

We have identified an MSR vulnerability in the AMDPowerProfiler.sys kernel driver, which is a part of *AMD µProf* profiling software.

What makes this driver stand out is that once the underlying software package is installed, the driver runs on every system boot. The unfiltered MSR IOCTL access combined with the lack of FILE_DEVICE_SECURE_OPEN flags (see Figure 11) and on-boot presence gives the attacker a good opportunity to exploit the driver even as an unprivileged user – this is an advantage compared to the BYOVD approach when the attacker needs to load the driver themselves.



```
RtlInitUnicodeString(&DestinationString, L"\\Device\\AMDPowerProfiler0");
result = IoCreateDevice(DriverObject, 0x48u, &DestinationString, FILE_DEVICE_UNKNOWN, 0, 0, &DeviceObject);
if ( result >= 0 )
{
  if ( !result )
  {
    DriverObject->MajorFunction[IRP_MJ_CREATE] = &Driver::OnCreate;
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = &Driver::OnClose;
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = Driver::OnDeviceControl;
    DriverObject->MajorFunction[IRP_MJ_CLEANUP] = &Driver::OnCleanup;
    DriverObject->DriverUnload = Driver::OnUnload;
    RtlInitUnicodeString(&SymbolicLinkName, L"\\??\\AMDPowerProfiler0");
```

Figure 11. AMD uProf kernel driver device creation without the FILE_DEVICE_SECURE_OPEN flag allowing non-admin access

On the other hand, the software is a niche utility for developers and not a package distributed to a large number of systems. We have not identified any other vulnerability in the driver.

The vulnerable IOCTL is IOCTL_ACCESS_MSR (0x222030).

AMD acknowledged the vulnerability (*CVE-2021-26334*) and released a fix in the November 2021 *Patch Tuesday* release.

Passmark software

Passmark is a company offering various computer benchmark and diagnostic tools. To achieve such functionality in user mode, a lot of low-level system features need to be accessed by leveraging a kernel-mode driver.

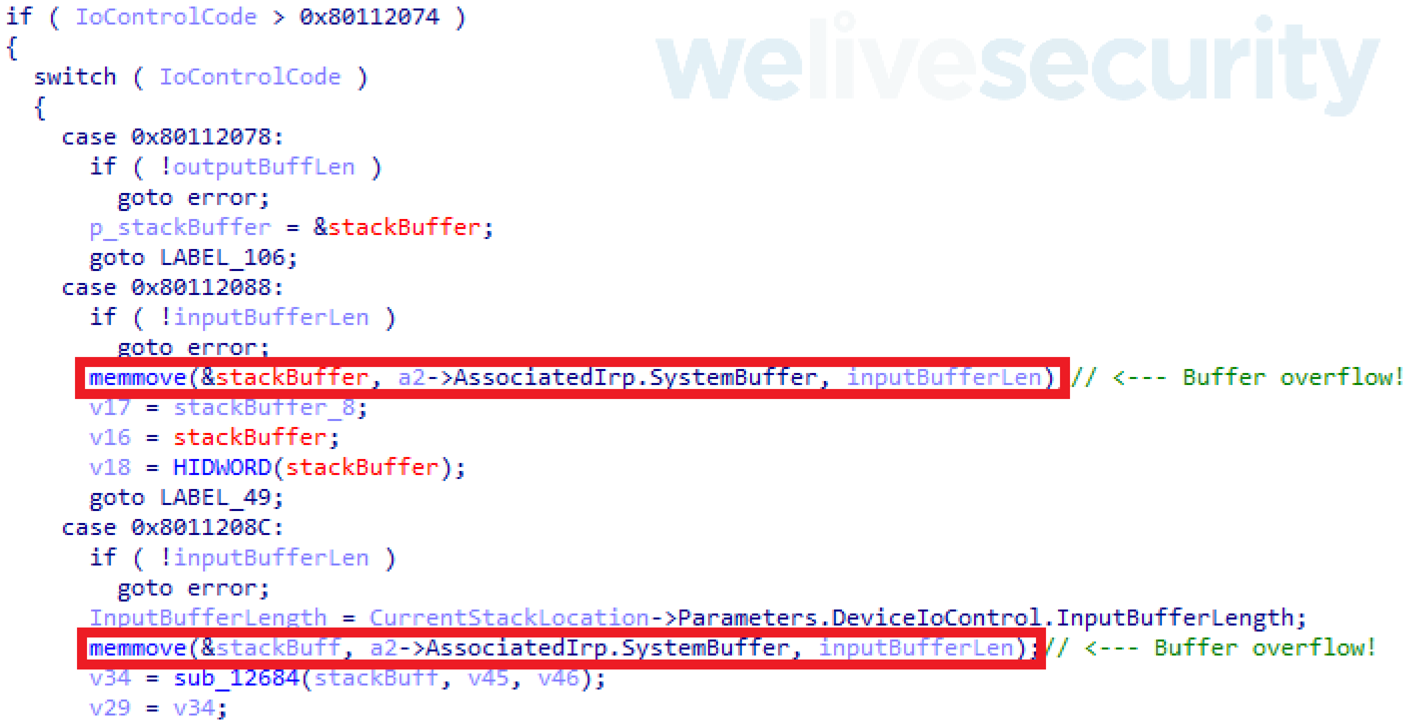
Passmark's DirectIo32.sys and DirectIo64.sys kernel drivers are a common framework shared and distributed among several of the vendor's applications – namely BurnInTest, PerformanceTest and OSForensics.

The driver contains direct, unfiltered MSR R/W access (*CVE-2020-15480*), an ability to map physical memory (*CVE-2020-15481*) for both reading and writing, and the IOCTL handler also contains a buffer overflow (*CVE-2020-15479*) due to blindly copying, without any size check, an IOCTL input buffer of arbitrary size to a local variable on the stack.

Passmark acknowledged these vulnerabilities and released a fixed version soon thereafter.

CVE-2020-15479

When the driver receives an IOCTL request from a user-mode program, it will first copy the requested input buffer into a local buffer on the stack. The size of the memmove is based only on the size of the input buffer (see Figure 12) and does not consider the capacity of the stack buffer. This may lead to a buffer overflow, if a large enough IOCTL buffer is provided. There are multiple unchecked memmove calls in the IOCTL handler function and several IOCTLs can be used to leverage the buffer overflow.



```
if ( IoControlCode > 0x80112074 )
{
  switch ( IoControlCode )
  {
    case 0x80112078:
      if ( !outputBuffLen )
        goto error;
      p_stackBuffer = &stackBuffer;
      goto LABEL_106;
    case 0x80112088:
      if ( !inputBufferLen )
        goto error;
      memmove(&stackBuffer, a2->AssociatedIrp.SystemBuffer, inputBufferLen) // <--- Buffer overflow!
      v17 = stackBuffer_8;
      v16 = stackBuffer;
      v18 = HIDWORD(stackBuffer);
      goto LABEL_49;
    case 0x8011208C:
      if ( !inputBufferLen )
        goto error;
      InputBufferLength = CurrentStackLocation->Parameters.DeviceIoControl.InputBufferLength;
      memmove(&stackBuff, a2->AssociatedIrp.SystemBuffer, inputBufferLen);// <--- Buffer overflow!
      v34 = sub_12684(stackBuff, v45, v46);
      v29 = v34;
```

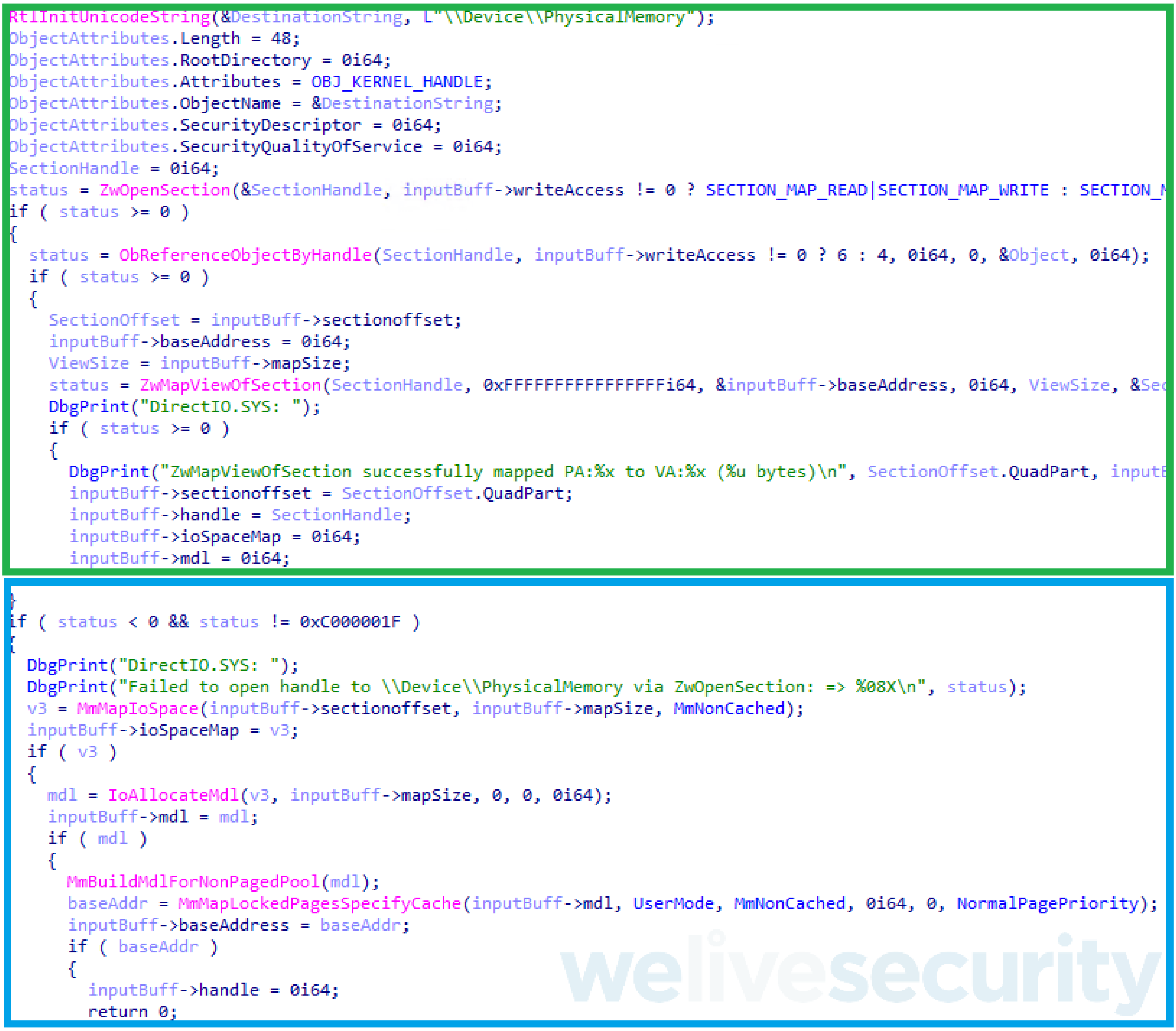
Figure 12. Buffer overflows in the vulnerable Passmark drivers

CVE-2020-15480

This driver offers RDMSR and WRMSR functionality exposed via an IOCTL that allows an unprivileged user-mode program to read and write arbitrary CPU MSRs without any additional checks. The vulnerable IOCTLs are IOCTL_READ_MSR (0x80112060) and IOCTL_WRITE_MSR (0x80112088).

CVE-2020-15481

Physical memory mapping functionality is exposed via a single control code – IOCTL_MAP_PHYSICAL_MEMORY (0x80112044). The implementation is split into two parts: the primary version is done through the ZwMapViewOfSection API; if for some reason this method fails, the function also implements a secondary approach as a backup, through the MmMapIoSpace and MmMapLockedPages kernel APIs. Both are illustrated in Figure 13.



```
RtlInitUnicodeString(&DestinationString, L"\\Device\\PhysicalMemory");
ObjectAttributes.Length = 48;
ObjectAttributes.RootDirectory = 0i64;
ObjectAttributes.Attributes = OBJ_KERNEL_HANDLE;
ObjectAttributes.ObjectName = &DestinationString;
ObjectAttributes.SecurityDescriptor = 0i64;
ObjectAttributes.SecurityQualityOfService = 0i64;
SectionHandle = 0i64;
status = ZwOpenSection(&SectionHandle, inputBuff->writeAccess != 0 ? SECTION_MAP_READ|SECTION_MAP_WRITE : SECTION_M
if ( status >= 0 )
{
  status = ObReferenceObjectByHandle(SectionHandle, inputBuff->writeAccess != 0 ? 6 : 4, 0i64, 0, &Object, 0i64);
  if ( status >= 0 )
  {
    SectionOffset = inputBuff->sectionoffset;
    inputBuff->baseAddress = 0i64;
    ViewSize = inputBuff->mapSize;
    status = ZwMapViewOfSection(SectionHandle, 0xFFFFFFFFFFFFFFFFi64, &inputBuff->baseAddress, 0i64, ViewSize, &Sec
    DbgPrint("DirectIO.SYS: ");
    if ( status >= 0 )
    {
      DbgPrint("ZwMapViewOfSection successfully mapped PA:%x to VA:%x (%u bytes)\n", SectionOffset.QuadPart, inputB
      inputBuff->sectionoffset = SectionOffset.QuadPart;
      inputBuff->handle = SectionHandle;
      inputBuff->ioSpaceMap = 0i64;
      inputBuff->mdl = 0i64;
}
if ( status < 0 && status != 0xC000001F )
{
  DbgPrint("DirectIO.SYS: ");
  DbgPrint("Failed to open handle to \\Device\\PhysicalMemory via ZwOpenSection: => %08X\n", status);
  v3 = MmMapIoSpace(inputBuff->sectionoffset, inputBuff->mapSize, MmNonCached);
  inputBuff->ioSpaceMap = v3;
  if ( v3 )
  {
    mdl = IoAllocateMdl(v3, inputBuff->mapSize, 0, 0, 0i64);
    inputBuff->mdl = mdl;
    if ( mdl )
    {
      MmBuildMdlForNonPagedPool(mdl);
      baseAddr = MmMapLockedPagesSpecifyCache(inputBuff->mdl, UserMode, MmNonCached, 0i64, 0, NormalPagePriority);
      inputBuff->baseAddress = baseAddr;
      if ( baseAddr )
      {
        inputBuff->handle = 0i64;
        return 0;
```

Figure 13. Physical memory IOCTL implementation in Passmark's drivers

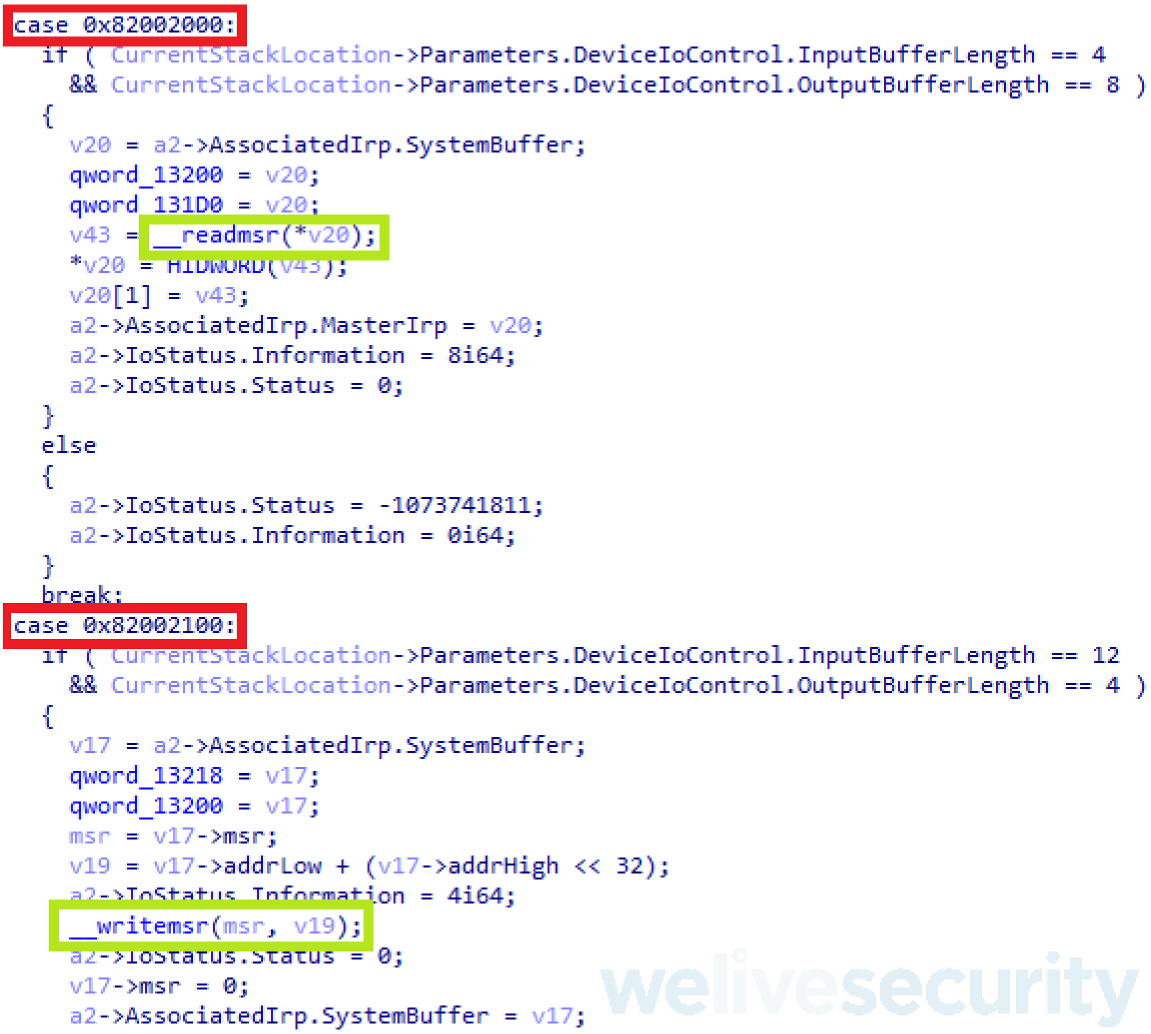
Devid Espenschied PC Analyser

PC Analyser is another utility for inspecting various details about the machine. The PCADRVX64.sys kernel driver distributed with the application contains two separate vulnerabilities – unfiltered MSR access (*CVE-2020-28921*) and ability to read from and write to arbitrary physical memory addresses (*CVE-2020-28922*). When creating the driver device, the FILE_DEVICE_SECURE_OPEN flag is unspecified, allowing unprivileged users to retrieve a handle to the driver.

Devid Espenschied acknowledged the vulnerabilities and released an updated version.

CVE-2020-28921

As with previous drivers, MSR access is unrestricted (see Figure 14) and the IOCTL code handler contains FILE_ANY_ACCESS flags allowing even an unprivileged user to leverage the functionality.



```
case 0x82002000:
  if ( CurrentStackLocation->Parameters.DeviceIoControl.InputBufferLength == 4
    && CurrentStackLocation->Parameters.DeviceIoControl.OutputBufferLength == 8 )
  {
    v20 = a2->AssociatedIrp.SystemBuffer;
    qword_13200 = v20;
    qword_131D0 = v20;
    v43 = __readmsr(*v20);
    *v20 = HIDWORD(v43);
    v20[1] = v43;
    a2->AssociatedIrp.MasterIrp = v20;
    a2->IoStatus.Information = 8i64;
    a2->IoStatus.Status = 0;
  }
  else
  {
    a2->IoStatus.Status = -1073741811;
    a2->IoStatus.Information = 0i64;
  }
  break;
case 0x82002100:
  if ( CurrentStackLocation->Parameters.DeviceIoControl.InputBufferLength == 12
    && CurrentStackLocation->Parameters.DeviceIoControl.OutputBufferLength == 4 )
  {
    v17 = a2->AssociatedIrp.SystemBuffer;
    qword_13218 = v17;
    qword_13200 = v17;
    msr = v17->msr;
    v19 = v17->addrLow + (v17->addrHigh << 32);
    a2->IoStatus.Information = 4i64;
    __writemsr(msr, v19);
    a2->IoStatus.Status = 0;
    v17->msr = 0;
    a2->AssociatedIrp.SystemBuffer = v17;
```

Figure 14. MSR IOCTL implementation in PC Analyser driver

CVE-2020-28922

The driver’s physical memory read and write functionality is implemented with separate IOCTLs based on the size of the read or write request. It offers the following control codes, none of which make any checks on the memory addresses targeted by the request:

IOCTL_READ_PHYSICAL_MEMORY_BYTE (0x82002400)
IOCTL_READ_PHYSICAL_MEMORY_WORD (0x82002500)
IOCTL_READ_PHYSICAL_MEMORY_DWORD (0x82002600)
IOCTL_WRITE_PHYSICAL_MEMORY_BYTE (0x82002700)
IOCTL_WRITE_PHYSICAL_MEMORY_WORD (0x82002800)
IOCTL_WRITE_PHYSICAL_MEMORY_DWORD (0x82002900)

Mitigations

While we have already mentioned several mechanisms employed by the CPU and/or the operating system, most of them can be bypassed with some clever techniques and are not very effective if the attacker prepares for them ahead of time. In this section we would like to mention some mitigation ideas that are actually effective at completely stopping the abuse of vulnerable drivers.

Virtualization-based security

Virtualization-based security or VBS is a feature introduced in Windows 10 that leverages hardware virtualization and makes the kernel sandboxed by a hypervisor in order to secure the operating system with various protections.

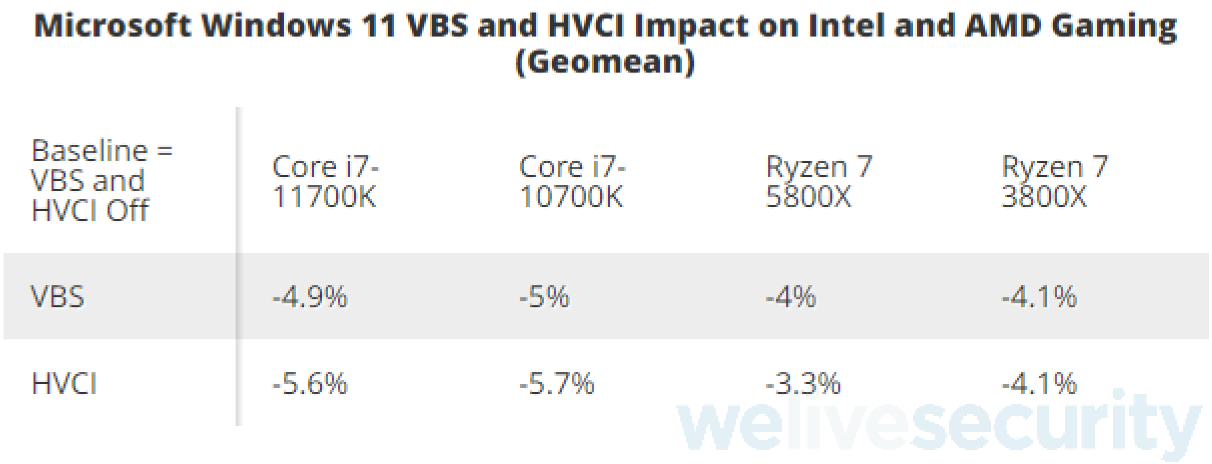
VBS offers several protection features with the most prominent one being Hypervisor-Protected Code Integrity (HVCI), which also comes as a standalone feature. HVCI enforces code integrity in the kernel and allows only signed code to be executed. It also employs blocklisting functionality, where a known piece of code signed by a specific, valid signature can be blocklisted and not allowed to be run or to be loaded. One of the drivers *that has been blocklisted already* via this method is the RWEverything utility.

HVCI effectively prevents vulnerable drivers from being abused to execute unsigned kernel code or load malicious drivers (regardless of the exploitation method used) and it seems that malware abusing vulnerable drivers to load malicious code was one of the *main motivations behind Microsoft implementing this feature*:

VBS provides significant security gains against practical attacks including several we saw last year, including human-operated *ransomware attacks like RobbinHood and sophisticated malware attacks like Trickbot, which employ kernel drivers and techniques that can be mitigated by HVCI*. Our research shows that there were 60% fewer active malware reports from machines reporting detections to Microsoft 365 Defender with HVCI enabled compared to systems without HVCI. The Surface Book 3 shipped in May 2020 and the Surface Laptop Go shipped in October 2020, and users may not have noticed they are running VBS and are therefore better protected based on the work done under the hood. *[emphasis added]*

Aside from enforcing kernel code integrity, VBS also secures important MSRs and disallows any changes to them. Unsurprisingly, this protection also affects the LSTAR MSR and mitigates all of the exploitation possibilities described above.

While VBS is an effective protection against MSR exploitation and running malicious code in the kernel in general, the adoption of this new feature is quite limited, as it has *several hardware requirements* that only newer machines can fulfill. There are also some drawbacks with the most notable being *a performance hit*, which may be quite noticeable depending on the workload. While *some benchmarks estimate the performance hit being as high as 25%* in specific video games, the *more detailed benchmarking by Tom’s Hardware* estimates the performance hit being around 5% depending on the specific benchmarks and hardware configuration (see Figure 15), which is still not a negligible amount and may lead some users to consider turning this feature off. There might also be some compatibility issues with legacy drivers and software. With the release of Windows 11, Microsoft has decided to enable HVCI by default for all compatible devices.



Microsoft Windows 11 VBS and HVCI Impact on Intel and AMD Gaming (Geomean)

Baseline = VBS and HVCI Off	Core i7-11700K	Core i7-10700K	Ryzen 7 5800X	Ryzen 7 3800X
VBS	-4.9%	-5%	-4%	-4.1%
HVCI	-5.6%	-5.7%	-3.3%	-4.1%

Figure 15. VBS benchmark results by Tom's Hardware

Third-party hypervisor

Similar to Microsoft's VBS, with new enough hardware, a third-party security solution may deploy its own custom hypervisor. Running the operating system under a hypervisor gives a detailed oversight of the state of the machine and provides the possibility of inspecting and intercepting any event including execution of a specific instruction. As with VBS, this comes at a cost, namely performance and compatibility.

Certificate revocation

On modern Windows systems, drivers need to have a valid signature based on an "acceptable" certificate. Hence, revoking the certificate of a vulnerable driver would be an easy way to "disarm" it and render it useless in most cases.

Sadly, revocation rarely ever happens and of the vulnerable drivers documented in our research above, not a single one of them has had its signature revoked. There are probably a multitude of reasons why such revocations are not happening, but the primary ones are likely to be time and cost. As nobody requires the revocation, it does not make much sense from the vendor's standpoint to ask for revocation, as this will be a costly and time-consuming process. Moreover, a signing certificate is usually shared among other projects, so the potential revocation because of a single driver could hinder the development of every project.

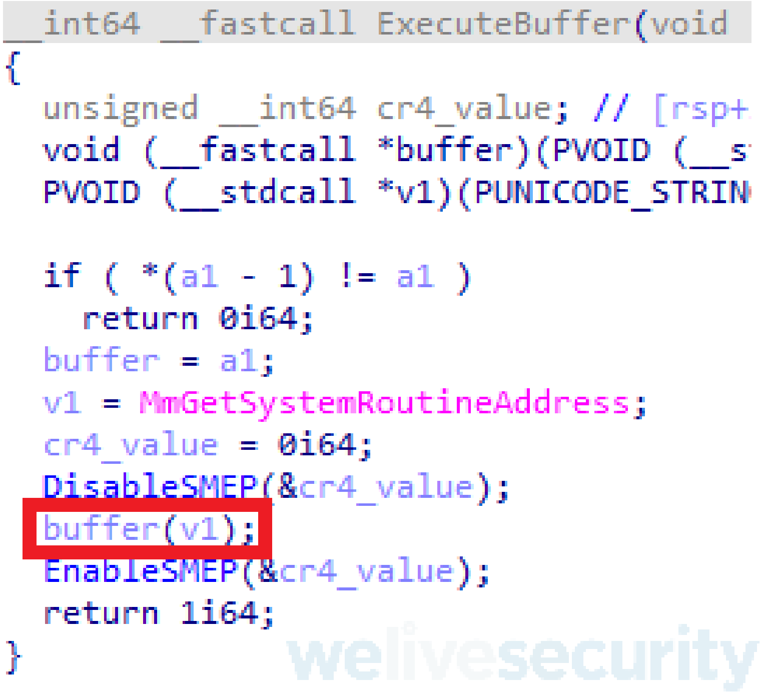
Further, during our research we *learned* that drivers with revoked certificates are not always blocked and that this problem is more complicated than it seemed at first. Revocation may not be the easiest solution after all.

Driver blocklisting

Driver blocklisting is a practice adopted by both Microsoft and various third-party security product vendors. Several of the most notorious vulnerable drivers are detected by ESET security products and deleted when found on a system. Microsoft also opted to blocklist drivers not only with its security solution, but also directly by the operating system utilizing their HVCI mitigation, which is part of virtualization-based security. While blocklisting is effective, it is not a proactive solution – only previously discovered vulnerable drivers can be blocklisted, and it must be done manually by each vendor. This means that this mitigation will not be effective against previously unknown, zero-day driver vulnerabilities that may be used in a sophisticated APT attack.

Probably the most prominent blocklisted driver is the Capcom “anti-cheat” driver Capcom.sys, which explicitly implements an IOCTL that simply executes the contents of the provided buffer in kernel mode (see Figure 16). To be able to execute a buffer provided from user mode, it even temporarily disables SMEP!

When discovered, the driver made several headlines and many unsigned driver loader tools were created based on abusing this function of the driver. Consequently, the driver was eventually *blocklisted by many security product vendors* including Microsoft and ESET.



```
__int64 __fastcall ExecuteBuffer(void
{
  unsigned __int64 cr4_value; // [rsp+
  void (__fastcall *buffer)(PVOID (__s
  PVOID (__stdcall *v1)(PUNICODE_STRIN

  if ( *(a1 - 1) != a1 )
    return 0i64;
  buffer = a1;
  v1 = MmGetSystemRoutineAddress;
  cr4_value = 0i64;
  DisableSMEP(&cr4_value);
  buffer(v1);
  EnableSMEP(&cr4_value);
  return 1i64;
}
```

Figure 16. Code snippet from Capcom anti-cheat driver

Conclusion

Vulnerable drivers have been a known problem for a long time and have been abused by the game-cheating community and malware authors alike, and while some effort has been made to mitigate the effects, it is still an ongoing battle. It seems that all the responsible parties involved want to solve this problem – the vendors we contacted were incredibly proactive during the disclosure process, eager to fix the vulnerabilities we uncovered. Microsoft is trying to strengthen the operating system from the inside and last but not least, third-party security vendors are trying to come up with clever ways to detect and mitigate such drivers themselves.

However, it seems that there is still a piece missing – a common, unified way of handling these issues including more thorough “disarming” of the drivers, whether by revoking or blocklisting their certificates, or some public, shared blocklists adopted by the security companies.

Bibliography

[1] J. Desimone and G. Landau, “BlackHat 2018: Kernel-mode Threats and Practical Defences,” 9 August 2018. [Online].
[2] R. Warns and T. Harrison, “INFILTRATE 2019: Device Driver Debauchery and MSR Madness,” 2 May 2019. [Online].
[3] J. Michael and M. Skhatov, “Defcon 27: Get off the Kernel if you can’t Drive,” 13 August 2019. [Online].
[4] N. Economou and E. Nissim, “ekoparty 2015: Windows SMEP bypass: U=S,” 2015. [Online].
[5] A. Shulmin, S. Yunakovsky, V. Berdnikov and A. Dolgushev, “The Slingshot APT,” 6 March 2018. [Online].
[6] Z. Hromcová and A. Cherepanov, “InvisiMole: The Hidden Part of the Story – Unearthing InvisiMole’s Espionage Toolset and Strategic Cooperations,” 18 June 2020. [Online].
[7] A. Ionescu, “UMPOwn: Ring 3 to Ring 0 in 3 acts,” in PoC||GTFO, vol. 2, No Starch Press, 2018, p. 768.
[8] A. Brandt and M. Loman, “Living off another land: Ransomware borrows vulnerable driver to remove security software,” Sophos, 7 February 2020. [Online].