Detecting and mitigating elevation-of-privilege exploit for CVE-2017-0005

microsoft.com/security/blog/2017/03/27/detecting-and-mitigating-elevation-of-privilege-exploit-for-cve-2017-0005/

March 27, 2017

On March 14, 2017, Microsoft released security bulletin MS17-013 to address CVE-2017-0005, a vulnerability in the Windows *Win32k* component that could potentially allow elevation of privileges. A report from a trusted partner identified a zero-day exploit for this vulnerability. The exploit targeted older versions of Windows and allowed attackers to elevate process privileges on these platforms.

In this article, we walk through the technical details of the exploit and assess the performance of tactical mitigations in Windows 10 Anniversary Update—released in August, 2016—as well as strategic mitigations like Supervisor Mode Execution Prevention (SMEP) and virtualization-based security (VBS). We also show how upcoming Creators Update enhancements to Windows Defender Advanced Threat Protection (Windows Defender ATP) can detect attacker elevation-of-privilege (EoP) activity, including EoP activities associated with the exploit.

To test how Windows Defender ATP can help your organization detect, investigate, and respond to advanced attacks, **sign up for a free trial**.

Zero-day elevation-of-privilege exploit

Upon review of its code, we found that this zero-day EoP exploit targets computers running Windows 7 and Windows 8. The exploit has been created so that it avoids executing on newer platforms.

The exploit package unfolds in four stages:

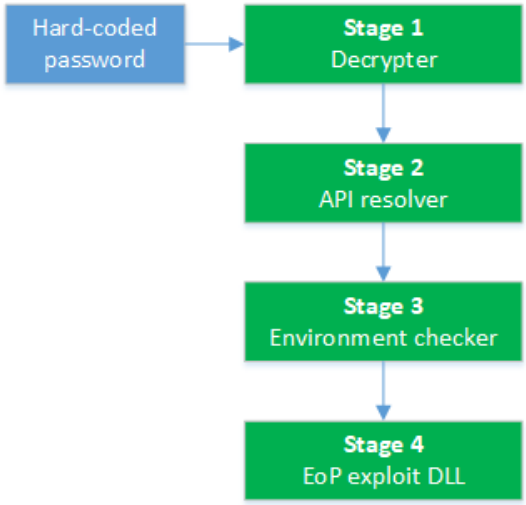

Figure 1. Execution stages of the exploit package and corresponding functionality

Stages 1 and 2 – Decryptor and API resolver

To protect the main exploit code, attackers have encrypted the initial stage PE file using AES-256 algorithm. To load code for the next stage, a password must be passed as a parameter to the main entry function. Using the *CryptHashData* API, the password is used as a key to decrypt the loader for the next stage.

Stage 2 acts as an intermediate stage where API resolution is performed. API resolution routines in this stage resemble how shellcode or position-independent code works.

The following code shows part of the *GetProcAddress* API resolution routine. This code appears to obfuscate the succeeding payload and stifle analysis.

Figure 2. Locating kernel32!GetProcAddress location using EAT traverse

Stage 3 – Avoiding newer platforms

In stage 3, the exploit package performs environmental checks, specifically to identify the operating system platform and version number. The attacker ensures that the exploit code runs on vulnerable systems that have fewer built-in mitigations, particularly Windows 7 and Windows 8 devices.

Figure 3. Code that performs environmental checks

Analysis of the exploit code reveals targeting of systems running specific versions of Windows:

- Major release version 5
- Major release version 6 and minor version 0, 1, or 2

These versions map to Windows operating systems between Windows 2000 and Windows 8, notably excluding Windows 8.1 and Windows 10. Also, upon examination of its architecture-checking routine, we find that the exploit code targets 64-bit systems.

The next stage payload is loaded through DLL reflection.

Stage 4 – Exploit routine

After the environmental checks, the attacker code begins actual exploit of the Windows kernel vulnerability CVE-2017-0005, resulting in arbitrary memory corruption and privileged code execution.

PALETTE.pfnGetNearestFromPalentry corruption

Code execution in the kernel space is made possible by a corrupted pointer in the *PALETTE.pfnGetNearestFromPalentry* function. Microsoft security researchers have been closely tracking this exploitation technique, which is designed to execute code in the kernel courtesy of a malformed *PALETTE* object. Observed in an unrelated sample used during the *Duqu* incident, we have described this relatively old exploit technique in a Virus Bulletin 2015 presentation.

The following snippet shows the corrupted state of the PALETTE function pointer:

```
1: kd> dt win32k!PALETTE @rax
        …
   +0x060 pfnGetNearestFromPalentry : 0x00000000`00610000     unsigned long  +610000 <--
Corrupt function pointer to shellcode
```

Figure 4. PALETTE.pfnGetNearestFromPalentry corruption

The exploit code calls the native API *NtGdiEngBitBlt* to trigger an *win32k!XLATEOBJ_iXlate* function call that uses the corrupted handler. This passes the control flow to a previously allocated shellcode. As a comparison, the exploit code in the *Duqu* 2.0 case used a *GetNearestPaletteIndex* call from *Gdi32.dll* to pass execution to the corrupt callback handler. This difference clearly indicates that these two exploits are unrelated, despite similarities in their code—similarities that can be attributed to the fact that these exploitation techniques are well-documented.

The exploit uses dynamically constructed *syscall* code snippets to call native Windows APIs.

```
00000000`02221660 4c8bd1          mov     r10,rcx
00000000`02221663 b8da110000      mov     eax,11DAh
00000000`02221668 0f05            syscall
00000000`0222166a c3              ret
```

Figure 5. Dynamically constructed calls to kernel functions

During the execution of the shellcode, the call stack looks like following:

```
fffff880`0595b840 fffff960`002dfa1a win32k!XLATEOBJ_iXlate+0x27b
fffff880`0595b880 fffff960`000a62f0 win32k!vLoadAndConvert32BitfieldsToBGRA+0x2e
fffff880`0595b8b0 fffff960`000a5f1c win32k!AlphaScanLineBlend+0x2c0
fffff880`0595b970 fffff960`0027e449 win32k!EngAlphaBlend+0x4dc
fffff880`0595bc30 fffff960`0027ea22 win32k!RenderNineGrid+0x275
fffff880`0595bcf0 fffff960`0027ec91 win32k!xxEngNineGrid+0x3c2
fffff880`0595be10 fffff960`0027ee50 win32k!EngNineGrid+0xb1
fffff880`0595beb0 fffff960`000c615d win32k!EngDrawStream+0x1a0
fffff880`0595bf60 fffff960`002b0e60 win32k!EngBitBlt+0x4cd
fffff880`0595c060 fffff800`0267b853 win32k!NtGdiEngBitBlt+0x708
fffff880`0595c1b0 00000000`0222166a nt!KiSystemServiceCopyEnd+0x13
00000000`0210f158 fffff800`02673c10 0x222166a
fffff880`0595c400 00005463`00000060 nt!KiCallUserMode
fffff880`0595c408 fffffa83`03456870 0x00005463`00000060
fffff880`0595c410 00000000`00000001 0xfffffa83`03456870
fffff880`0595c418 00000000`00000000 0x1
```

Figure 6. Example of the call stack when passing control flow using the corrupted function handler

Once the shellcode is executed, the exploit uses a common token-swapping technique to obtain elevated, SYSTEM privileges for the current process. This technique is often observed in similar EoP exploits.

```
1: kd> u @rip
00000000`00180000 55              push    rbp
00000000`00180001 4883ec50        sub     rsp,50h
00000000`00180005 488d4c2420      lea     rcx,[rsp+20h]
00000000`0018000a 4889cd          mov     rbp,rcx
00000000`0018000d 48b820e7940200f8ffff mov rax,offset nt!PsLookupProcessByProcessId
(fffff800`0294e720)
00000000`00180017 ba04000000      mov     edx,4
00000000`0018001c 488d4d08        lea     rcx,[rbp+8]
00000000`00180020 48894d18        mov     qword ptr [rbp+18h],rcx
00000000`00180024 4889d1          mov     rcx,rdx
00000000`00180027 488b5518        mov     rdx,qword ptr [rbp+18h]
00000000`0018002b ffd0            call    rax
00000000`0018002d 894500          mov     dword ptr [rbp],eax
00000000`00180030 48b820e7940200f8ffff mov rax,offset nt!PsLookupProcessByProcessId
(fffff800`0294e720)
00000000`0018003a badc0a0000      mov     edx,0ADCh
00000000`0018003f 488d4d10        lea     rcx,[rbp+10h]
00000000`00180043 48894d20        mov     qword ptr [rbp+20h],rcx
00000000`00180047 4889d1          mov     rcx,rdx
00000000`0018004a 488b5520        mov     rdx,qword ptr [rbp+20h]
00000000`0018004e ffd0            call    rax
00000000`00180050 894504          mov     dword ptr [rbp+4],eax
00000000`00180053 b808020000      mov     eax,208h
00000000`00180058 48034508        add     rax,qword ptr [rbp+8]
00000000`0018005c ba08020000      mov     edx,208h
00000000`00180061 48035510        add     rdx,qword ptr [rbp+10h]
00000000`00180065 488b00          mov     rax,qword ptr [rax]
00000000`00180068 488902          mov     qword ptr [rdx],rax ← token replacement of target
EPROCESS.Token with SYSTEM's EPROCESS.Token
00000000`0018006b b800000000      mov     eax,0
00000000`00180070 488d6530        lea     rsp,[rbp+30h]
00000000`00180074 5d              pop     rbp
00000000`00180075 c3              ret
```

Figure 7. Token-swapping shellcode

Mitigation and detection

As previously mentioned, this zero-day exploit does not target modern systems like Windows 10. If environmental checks in the exploit code are bypassed and it is forced to execute on such systems, our tests indicate that the exploit would be unable to completely execute, mitigated by additional layers of defenses. Let's look at both the tactical mitigations—medium-term mitigations designed to break exploitation techniques—as well as the strategic mitigations—durable, long-term mitigations designed to eliminate entire classes of vulnerabilities—that stop the exploit.

Tactical mitigation – prevention of *pfnGetNearestFromPalentry* abuse

The use of *PALETTE.pfnGetNearestFromPalentry* as a control transfer point has been tracked by Microsoft security researchers for quite some time. In fact, this method is on the list tactical mitigations we have been pursuing. In August 2016, with the Windows 10 Anniversary Update, Microsoft released tactical mitigation

designed to prevent the abuse of *pfnGetNearestFromPalentry*. The mitigation checks the validity of *PALETTE* function pointers when they are called, ensuring that only a predefined set of functions are called and preventing any abuse of the structure.

Strategic mitigations

Other than the described tactical mitigation, this exploit could also be stopped in Windows 10 by SMEP, ASLR improvements in Windows kernel 64-bit, and virtualization-based security (VBS).

Supervisor Mode Execution Prevention (SMEP)

SMEP is a strategic mitigation feature supported by newer Intel CPUs and adopted since Windows 8.

With SMEP, bits in the page table entry (PTE) serve as *User/Supervisor (U/S)* flags that designate the page to be either in user mode or kernel mode. If a user-mode page is called from kernel-mode code, SMEP generates an access violation and the system triggers a bug check that halts code execution and reports a security violation. This mechanism broadly stops attempts at using user-mode allocated executable pages to run shellcode in kernel mode, a common method used by EoP exploits.

```
A fatal system error has occurred.
Debugger entered on first try; Bugcheck callbacks have not been invoked.

A fatal system error has occurred.

The debuggee is ready to run
win32k!vSolidFillRect1+0x10f:
fffff960`0002794e 48894820        mov     qword ptr [rax+20h],rcx

1: kd> kp
 # Child-SP          RetAddr           Call Site
00 fffff880`070286f8 fffff803`26fff0ea nt!DbgBreakPointWithStatus
01 fffff880`07028700 fffff803`26ffe742 nt!KiBugCheckDebugBreak+0x12
02 fffff880`07028760 fffff803`26f04144 nt!KeBugCheck2+0x79f
03 fffff880`07028e80 fffff803`270726ee nt!KeBugCheckEx+0x104
04 fffff880`07028ec0 fffff803`27072674 nt! ?? ::FNODOBFM::`string'+0x33594
05 fffff880`07028f00 fffff803`26f3ea09 nt! ?? ::FNODOBFM::`string'+0x3351d
06 fffff880`07028f50 fffff803`26f01aee nt!MmAccessFault+0x3e9
07 fffff880`07029090 00000026`113c0000 nt!KiPageFault+0x16e ← page fault from SMEP
08 fffff880`07029228 fffff960`00052287 0x00000026`113c0000
09 fffff880`07029230 fffff960`002ac5e4 win32k!XLATEOBJ_iXlate+0x117
0a fffff880`07029260 fffff960`0033df76 win32k!vLoadAndConvert32BitfieldsToBGRA+0x34
```

Figure 8. SMEP capturing exploit attempt

Strategic mitigation like SMEP can effectively raise the bar for a large pool of attackers by instantly rendering hundreds of EoP exploits ineffective, including old-school exploitation methods that call user-mode shellcode directly from the kernel, such as the zero-day exploit for CVE-2017-0005.

To check whether a computer supports SMEP, one can use the Coreinfo tool. The tool uses CPUID instructions to show the sets of CPUs and platforms that should support the feature. The following screen shows that the tested CPU supports SMEP. SMEP is supported on Windows 8 and later.

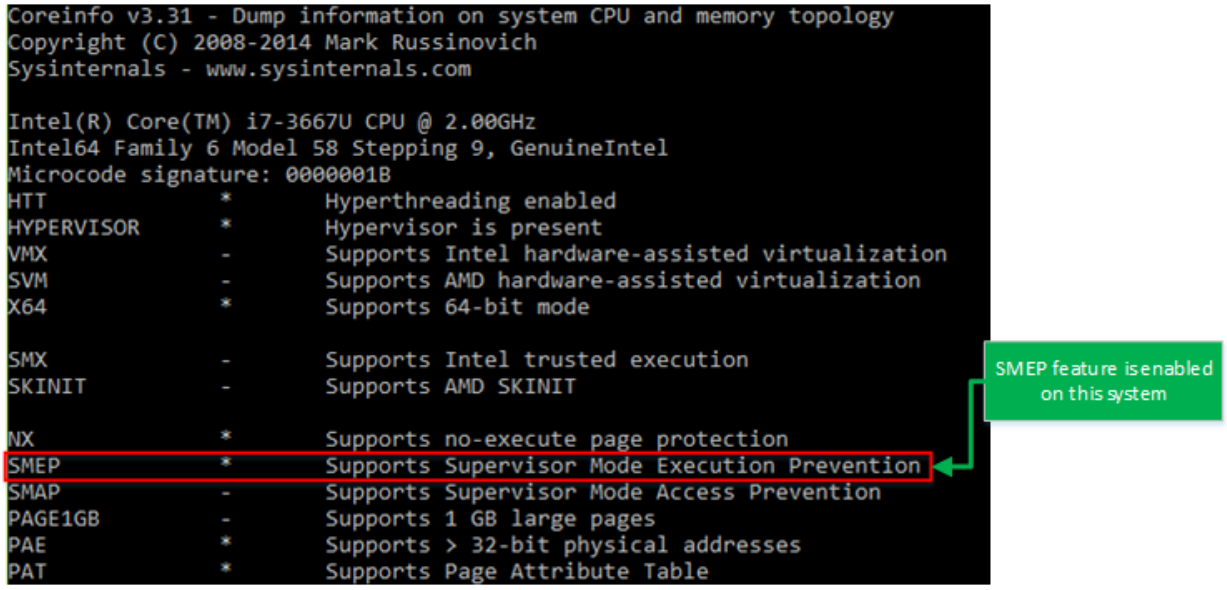

Figure 9. Coreinfo shows whether SMEP is enabled

Windows kernel 64-bit ASLR improvements

Although attackers are forced to work harder to create more sophisticated exploits with SMEP, we do know from studies shared in security conferences and documented incidents that there are ways to potentially bypass SMEP mitigation. These bypass mechanisms include the use of kernel ROP gadgets or direct PTE modifications through read-write (RW) primitives. To respond to these foreseeable developments in exploitation techniques, Microsoft has provided Windows kernel 64-bit ASLR improvements with the Windows 10 Anniversary Update and has made SMEP stronger with randomized kernel addresses, mitigating a bypass vector resulting from direct PTE corruption.

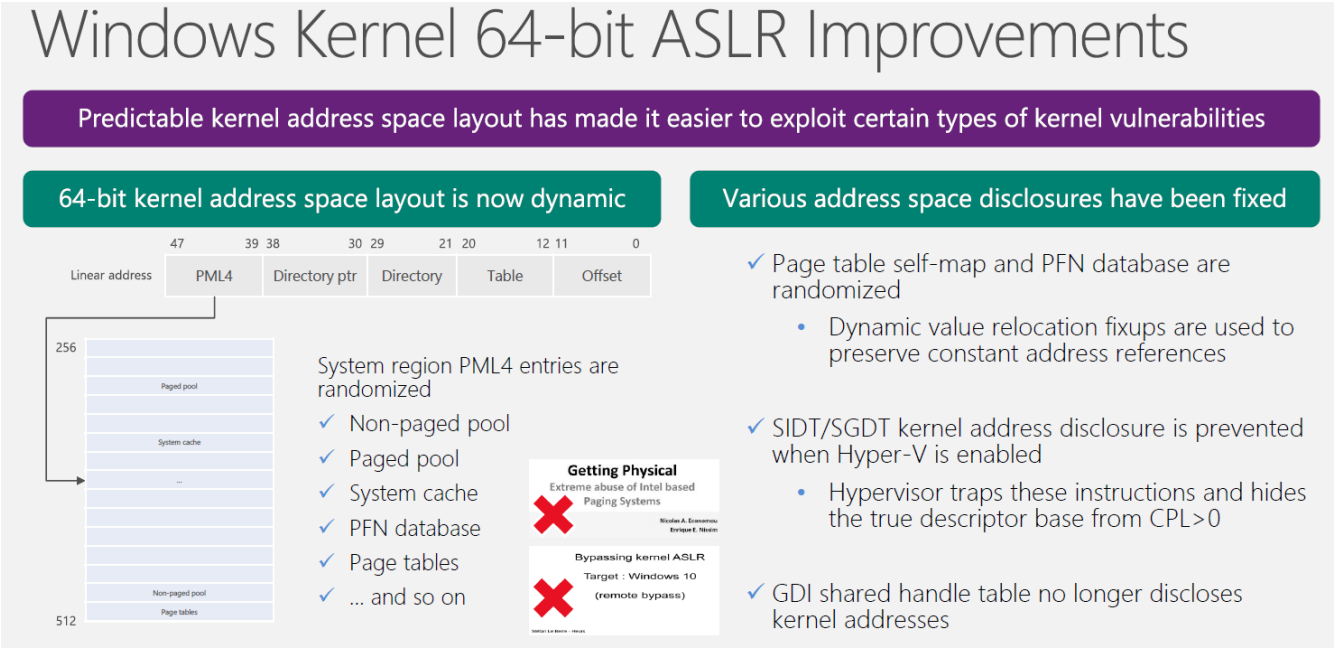

Figure 10. Windows Kernel 64-bit ASLR improvements

Virtualization-based security (VBS)

Virtualization-based security (VBS) enhancements provide another layer of protection against attempts to execute malicious code in the kernel. For example, Device Guard blocks code execution in a non-signed area in kernel memory, including kernel EoP code. Enhancements in Device Guard also protect key MSRs, control registers, and descriptor table registers. Unauthorized modifications of the CR4 control register bitfields, including the SMEP field, are blocked instantly.

Windows Defender ATP detections

With the upcoming Creators Update release, Windows Defender ATP will be able to detect attempts at a SMEP bypass through CR4 register modifications. Windows Defender ATP will monitor the status of the CR4.SMEP bit and will report inconsistencies. In addition to this, Windows Defender ATP will detect token-swapping attempts by monitoring the state of the token field of a process structure.

The following screenshot shows Windows Defender ATP catching exploit code performing the token-swapping technique to elevate privileges.

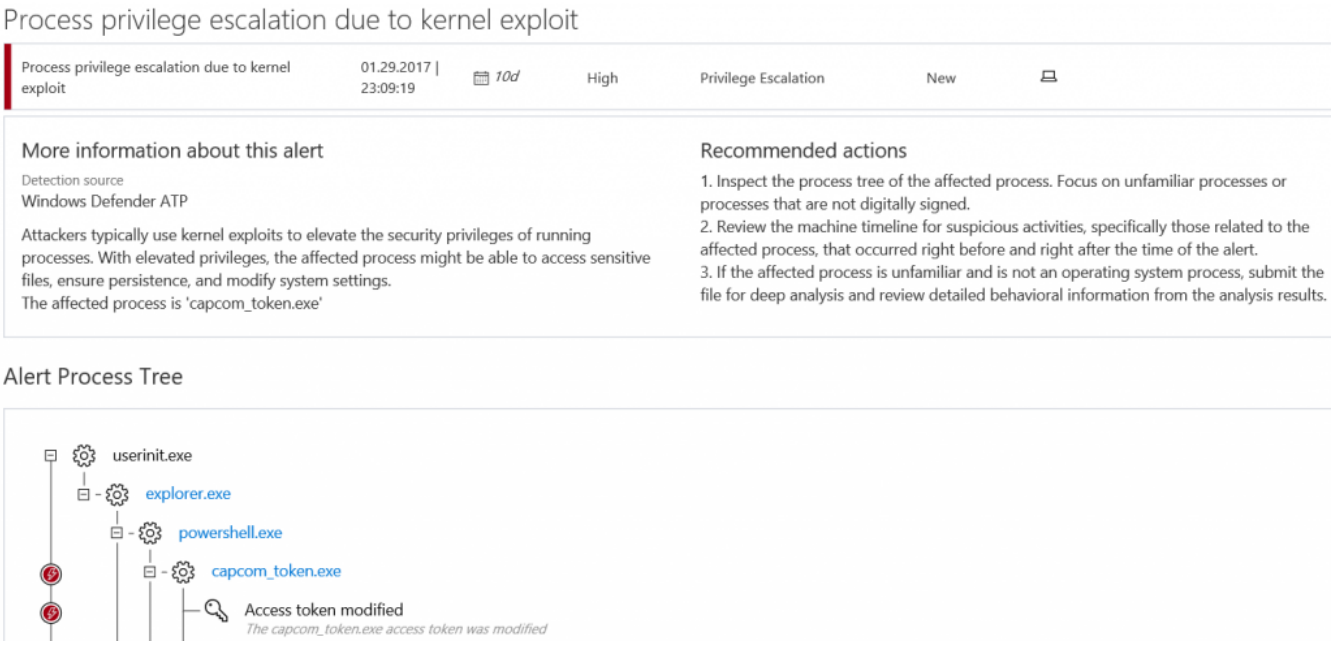

Figure 11. Detection of token-swapping technique on Windows Defender ATP

Conclusion: Resiliency with mitigation and behavioral detection

The zero-day exploit for CVE-2017-0005 shied away from newer systems because it would have simply been stopped and would have only managed to get unnecessary exposure. Attackers are not so much focusing on legacy systems but avoiding security enhancements present in modern hardware and current platforms like Windows 10 Anniversary Update. While patches continue to provide single-point fixes for specific vulnerabilities, this attacker behavior highlights how built-in exploit mitigations like SMEP, the ASLR improvements, and virtualization-based security (VBS) are providing resiliency.

Windows Defender ATP with Creators Update—now available for public preview—extends defenses further by detecting exploit behavior on endpoints. With the upcoming enhancements, Windows Defender ATP could raise alerts so that SecOps personnel are immediately made aware of EoP activity and can respond accordingly. Read our previous post about uncovering cross-process injection to learn more about how Windows Defender ATP detects sophisticated breach activity.

In addition to strengthening generic detection of EoP exploits, Microsoft security researchers are actively gathering threat intelligence and indicators attributable to ZIRCONIUM, the activity group using the CVE-2017-0005 exploit. Comprehensive threat intelligence about activity groups and their attack methods are available to Windows Defender ATP customers.

Windows Defender ATP is built into the core of Windows 10 Enterprise and can be evaluated free of charge.

Matt Oh
Windows Defender ATP Research Team

Talk to us

Questions, concerns, or insights on this story? Join discussions at the Microsoft community and Windows Defender Security Intelligence.