Blog

[MS15-010 / CVE-2015-0057] Exploitation

MS15-010/CVE-2015-0057 win32k Local Privilege Escalation
Date: 2015-12-17
Exploit author: Jean-Jamil Khalife
Tested on Windows 8.1 x64 EN
Home: http://www.hdwsec.fr (http://www.hdwsec.fr)
Blog: http://www.hdwsec.fr/blog/ (http://www.hdwsec.fr/blog/)

Introduction

At the beginning of 2015 Udi Yavo [1] found a Windows kernel vulnerability that can be exploited from Windows XP up to Windows 10 (preview).
A couple articles about CVE-2015-0057 were published last summer:

1. FireEye described some technical details about the Dyre malware that exploited this vulnerability [2].
2. NCC Group explained technical solutions to achieve reliable exploitation on Windows 8.1; however they did not release any code [3].

Overview

The vulnerability is a kernel use-after-free, which allows getting a non-arbitrary write primitive and then corrupts an adjacent object.
This yields a write-what-where primitive wherever you want in kernel space (and user space).

More precisely, the exploit triggers the use-after-free of a scrollbar object and replaces the freed object by a new proplist object
of the same size. Because of the use-after-free, the header of the new object will be modified by an OR primitive.

```
1 | win32k !xxxEnableWndSBArrows+0xaa:
```

```
2 | fffff960`0036cdfe 0903          or      dword ptr [rbx],eax
```

This vulnerability leads to increasing the maximum number of properties that you can push into the heap.
So one can overwrite the next object by adding new properties with SetProp(HWND hWnd, LPCTSTR lpString, HANDLE hData).

For the remainder of this article we will use the notation "SetProp(hWnd, V1, V2)" in place of "SetProp(HWND hWnd, LPCTSTR lpString, HANDLE hData)"

Exploitation on Windows 8.1 (64-bit)

According to the FireEye paper, the malware uses the technique explained above to corrupt an adjacent menu object.
Indeed, one of the screenshots shows that the value of tagMENU.rgItems and tagMENU.cItems are overwritten.

```
typedef struct _MENU
{
…
  DWORD         cItems  // number of items contained by the items array
…
  PITEM         rgItems // pointer to the items array
…
}
```

The Dyre malware sets rgItems to 0x38, and cItems to -1. This provides a write primitive on Windows XP to overwrite nt!HalDispatchTable+0x4.
We then attempted to reproduce the technique detailed by FireEye and NCC Group on Windows 8.1 64-bit.

A few obstacles were encountered:

1. The contents of each property are not fully controlled by the attacker: while the 8 bytes of V2 are completely controlled, only two bytes of V1 will be used.
2. The _MENU structure detailed above changed, and due to the lack of control over V1's content, cItems and rgItems could not be fully controlled.
3. Both ASLR and SMEP have to be bypassed as well.
4. The heap entry encoding mechanism must be bypassed.
5. Finally, overwritten objects must be restored to avoid a crash when freed by the kernel.

Getting an arbitrary write

Instead of directly overwriting rgItems and cItems, NCC Group found and provided a workaround involving the manipulation of heap headers and two overlays so as to ultimately control a window structure, providing a read-write primitive. We recommend reading their excellent article [3], in which the technique is fully detailed.

I chose instead to control a menu object, as this provides a common write primitive across a variety of operating systems (from Windows XP to 8.1); the lack of a read primitive is not a showstopper in this case.

While the Dyre malware corrupted cItems and rgItems, we corrupt only the rgItems field:

a. A menu with one item is created, using CreateMenu() and InsertMenuItem()
b. The rgItems (address to the item array) is then corrupted with NCC Group's overlay control method.
c. The first item of the array pointed to by rgItems contains a wID field, which we modify using SetMenuItemInfo(), leading to an arbitrary write primitive.

The next steps of the exploitation is fairly classic: a function pointer at nt!HalDispatchTable+8 is overwritten to run a payload containing two stages:

a. The first one restores the desktop heap and the original function pointer.
b. The second one is a shellcode which changes the token of the current process with the one of the system process.

Bypassing heap header encoding

In this exploitation case, the heap header of the first overlay has to be modified and we have to build a fake header in the content of the second overlay. The problem is that Windows 8.1 heap headers are encoded with a cookie. For further details, please consult Chris Valasek and Tarjei Mandt excellent papers ([9] and [10]).

NCC Group provided a workaround that relies on the read-only user-land mapping of the desktop heap, which discloses the cookie value.

Firstly, we have to find the cookie used to encode the heap header by determining the heap base of the desktop heap. Then we can get the cookie stored at offset 0x80 as seen in the structure below:

```
nt!_HEAP
   +0x000 Entry            : _HEAP_ENTRY
   +0x010 SegmentSignature : Uint4B
   […]
   +0x07c EncodeFlagMask   : Uint4B
   +0x080 Encoding         : _HEAP_ENTRY
   […]
```

Now one can decode the heap header by xor'ing it with the Encoding field.
Thirdly, once the entry is decoded and modified we only need to calculate the new SmallTagIndex checksum:

```
Entry.SmallTagIndex = Entry.SizeLow ^ Entry.SizeHigh ^ Entry.Flags;
```

Bypassing ASLR in Medium Integrity Level

ASLR is not a problem in medium integrity privilege because one can:

a. Get ntoskrnl and hal.dll base addresses using NtQuerySystemInformation()
b. Dynamically obtain the offsets to ROP gadgets and hal!HaliQuerySystemInformation() (to be

restored) reading ntoskrnl.exe and hal.dll respectively.

Bypassing ASLR in Low Integrity Level

NtQuerySystemInformation() can not be called from a low integrity level. Although not implemented in the exploit source code, Alex Ionescu [5] presented the use of SIDT as a kernel information leak that works regardless of the integrity level.

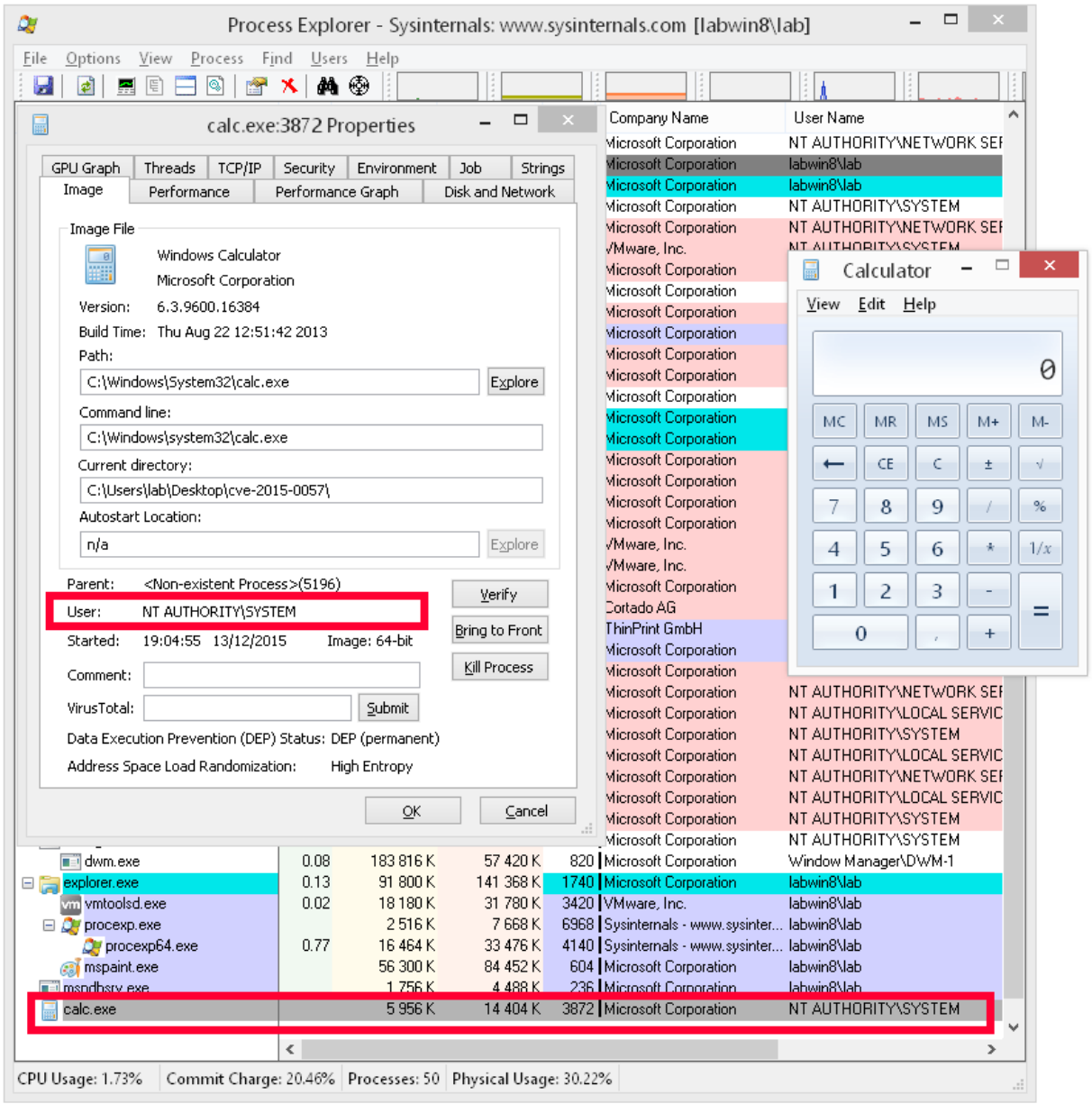
Bypassing SMEP

In May 2011, Dan Rosenberg presented a novel technique to bypass SMEP on Linux in order to make user-mode pages executable from kernel [6]. j00ru and Gynvael Coldwind further developed this technique on Windows [7]. In the context of this exploit, I followed siberas' method as explained in their 2014 pwn2own paper [8].

Preventing post-exploitation crashes

To prevent any post-exploitation crashes, the desktop heap has to be cleaned. Indeed the corrupted objects and the original pointer to hal!HaliQuerySystemInformation() have to be restored.

The following screenshot show the result of successful exploitation:



You can download the exploit here (CVE-2015-0057.zip).

References:

[1] http://breakingmalware.com/vulnerabilities/one-bit-rule-bypassing-windows-10-protections-using-single-bit/ (http://breakingmalware.com/vulnerabilities/one-bit-rule-bypassing-windows-10-protections-using-single-bit/)

[2] https://www.fireeye.com/blog/threat-research/2015/07/dyre_banking_trojan.html (https://www.fireeye.com/blog/threat-research/2015/07/dyre_banking_trojan.html)

[3] https://www.nccgroup.trust/globalassets/newsroom/uk/blog/documents/2015/07/exploiting-cve-2015.pdf (https://www.nccgroup.trust/globalassets/newsroom/uk/blog/documents/2015/07/exploiting-cve-2015.pdf)

[4] https://media.blackhat.com/bh-us-11/Mandt/BH_US_11_Mandt_win32k_WP.pdf (https://media.blackhat.com/bh-us-11/Mandt/BH_US_11_Mandt_win32k_WP.pdf)

[5] http://recon.cx/2013/slides/Recon2013-Alex%20Ionescu-I%20got%2099%20problems%20but%20a%20kernel%20pointer%20ain't%20one.pdf (http://recon.cx/2013/slides/Recon2013-Alex%20Ionescu-I%20got%2099%20problems%20but%20a%20kernel%20pointer%20ain't%20one.pdf)

[6] http://vulnfactory.org/blog/2011/06/05/smep-what-is-it-and-how-to-beat-it-on-linux/ (http://vulnfactory.org/blog/2011/06/05/smep-what-is-it-and-how-to-beat-it-on-linux/)

[7] http://j00ru.vexillium.org/?p=783 (http://j00ru.vexillium.org/?p=783)
[8] http://www.siberas.de/papers/Pwn2Own_2014_AFD.sys_privilege_escalation.pdf (http://www.siberas.de/papers/Pwn2Own_2014_AFD.sys_privilege_escalation.pdf)
[9] http://illmatics.com/Understanding_the_LFH.pdf (http://illmatics.com/Understanding_the_LFH.pdf)
[10] http://illmatics.com/Windows%208%20Heap%20Internals.pdf (http://illmatics.com/Windows%208%20Heap%20Internals.pdf)