A New CVE-2015-0057 Exploit Technology

yu.wang@fireeye.com

September 28, 2015

February 10, 2015, Patch Tuesday - Microsoft pushed many system-level patches including CVE-2015-0057/MS15-010[1]. On the same day, Udi Yavo, the CTO of enSilo and the discoverer of the vulnerability, released a technical blog[5] on that topic. Udi described the CVE-2015-0057 vulnerability in detail and demonstrated the process of exploiting the vulnerability on the 64-bit Windows 10 Technical Preview Operating System. Four months later, on the 17th of June, a new variant of the Dyre Banking Trojan[3] was detected by FireEye. This variant of Dyre attempts to exploit CVE-2013-3660[12] and CVE-2015-0057 in order to obtain system privileges. This is the first time CVE-2015-0057 was found to be exploited in the wild. Then, on July 8th, NCC Group published their technical blog[6], describing their exploit technique in detail, which allows the exploit to work reliably on all 32 and 64-bit Windows - from Windows XP to Windows 8.1.

It is worth noting that in this year, we have repeatedly caught APT class zero-day attacks[2] [4] - all of which target the Win32K subsystem's User Mode Callback mechanism. This leads us to re-visit this old-school kernel attack surface[7] [9] [11]. This paper will focus on CVE-2015-0057 and the User Mode Callback mechanism. We will examine the User Mode Callback mechanism from two aspects: exploit methodology and vulnerability detection. Additionally, from an attacker's perspective, this paper will also reveal some new exploit techniques.

Background of the Vulnerability

CVE-2015-0057 is a traditional Use-After-Free vulnerability, which is introduced by the User Mode Callback mechanism. There are actually two options available to trigger a USE operation after the call to DestroyWindow, namely SET and UNSET. Let us follow Udi Yavo's ideas, and follow the SET code branches, use tagPROPLIST objects as memory placeholders[10] and set off on our exploitation journey.

Opting to use the tagPROPLIST structure as the Use-After-Free memory placeholder is primarily based on the following considerations:

1. tagPROPLIST object and vulnerability related object tagSBINFO, are all allocated from the Win32K Desktop Heap.
2. Inside the tagPROPLIST object, there is an array named tagPROP. This array's size can be adjusted to meet the needs of the exploit.
3. The contents of the tagPROPLIST object can be manipulated through User32 APIs, such as SetProp routine.
4. Crucially, the behavior of SET after the Use-After-Free vulnerability will overwrite tagPROPLIST's cEntries field, which means that any subsequent operations on this tagPROPLIST object can result in another Out-Of-Bounds access.

Further analysis of the vulnerability and the tagPROPLIST object's properties reveals that, in order to achieve arbitrary kernel memory read and write, we have to overcome two obstacles, namely:

1. The write ability of the tagPROPLIST object's SetProp method is restricted, due to the implementation of InternalSetProp.
2. Since the write, and hence repair, capability is limited, continuous memory corruption is unacceptable. Our next step in this journey would be to solve the problems listed above to achieve precise memory writes using SetProp.

```
1    typedef struct tagPROP {
2        KHANDLE hData;
3        ATOM atomKey;
4        WORD fs;
5    } PROP, *PPROP;
```

tagPROP

```
1   BOOL InternalSetProp(LPWSTR pszKey, HANDLE hData, DWORD dwFlags)
2   {
3       PPROP pprop = FindorCreateProp();
4
5       ......
6
7       pprop->atomKey = PTR_TO_ID(pszKey);
8       pprop->fs = dwFlags; /* cannot be controlled (0/2) */
9       pprop->hData = hData;
10
11      ......
12  }
```

InternalSetProp Pseudocode

Obstacle 1 as shown in pseudocode InternalSetProp. Due to the characteristics of this routine, we cannot control the fs field of the tagPROP structure. In other words, from the perspective of exploitation, for every eight bits there are two bits out of control on 32-bit platforms. On 64-bit platforms, out of every sixteen bits there are six bits out of our control.

```
0:  kd> dd bc65b468 lc
bc65b468  00060006 00080100 00000004 00000004
bc65b478  00bc1040 0000a918 00000000 00002141
bc65b488  00000000 00003141 deadbeef 0002c01a /* 2 bytes are out of control */
```

32-Bit SetProp Restrictions

```
0:  kd> dq fffff900‘c0841898 l6
fffff900‘c0841898  08000003‘00010003 00000002‘00000002
fffff900‘c08418a8  00000000‘02f47580 00000000‘0000a918
fffff900‘c08418b8  deadbeef‘deadbeef 00000000‘0002c04d /* 6 bytes are out of control */
```

64-Bit SetProp Restrictions

Let us take a look at how NCC Group's Aaron Adams worked to overcome this obstacle.

NCC Group's 32-bit Exploit Method

Aaron Adams's 32-bit exploit method uses tagPROPLIST and the tagWND objects as heap Feng Shui[13] layouts. Aaron's exploit sprays tagPROPLIST twice. The first tagPROPLIST spray is used to create a placeholder object when the Use-After-Free happens. In order to facilitate the distinction, let's call it U-A-F tagPROPLIST object. The second tagPROPLIST spray is used to create a stepping stone object used to achieve relative memory reading and writing. We can call it the Zombie

tagPROPLIST object. Lastly, tagWND is the host object ultimately used to achieve arbitrary kernel memory reading and writing. Aaron chose it because the tagWND's strName.Buffer field has ability to read and write bytes in kernel memory.

The exploit process begins from the U-A-F tagPROPLIST object. By calling the SetProp routine on the U-A-F tagPROPLIST object, we can effectively control the adjacent cEntries and iFirstFree fields of the Zombie tagPROPLIST object. With reference to obstacle 1, although we are unable to fully control the value of iFirstFree, it is still enough to jumpstart the whole exploit process. Thereafter, the subsequent invocations of SetProp on the Zombie tagPROPLIST object actually let the exploit code gain the ability to write kernel memory accurately.

Even if SetProp's limited write capability did not prevent us from directly controlling the Zombie tagPROPLIST.iFirstFree field, SetProp's limitations will be evident when the exploit attempts to control the tagWND.strName.Buffer field. In order to solve this problem Aaron came up with a clever set of complex schemes. First, he manipulates the tagWND.pSBInfo field, which can be completely controlled, by pointing it to the tagWND.strName field. Aaron then rewrites the tagWND's strName.Buffer field indirectly through the SetScrollInfo routine - this roundabout scheme finally allows him to bypass the write restrictions we faced earlier and achieve arbitrary read and write capabilities.

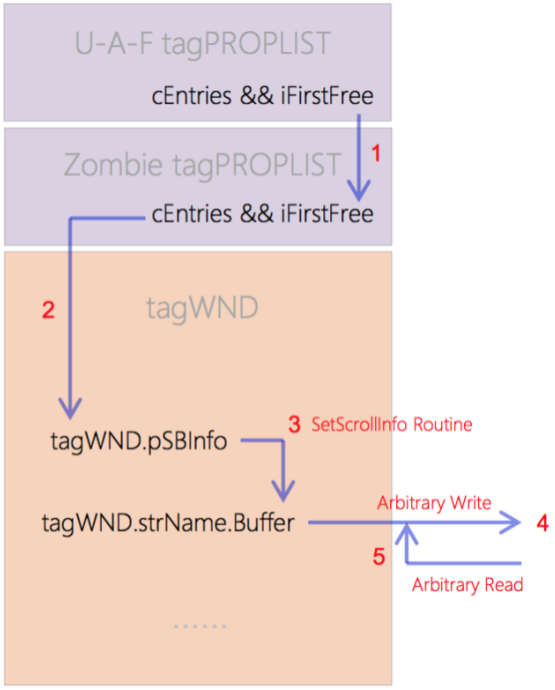
32-bit exploitation process can be summarized as follows:

1. Use the U-A-F tagPROPLIST object's Out-Of-Bounds write capabilities to manipulate Zombie tagPROPLIST's properties.
2. Use the corrupted Zombie tagPROPLIST object to rewrite the adjacent tagWND object's pSBInfo field.
3. Rewrite tagWND object's strName.Buffer field indirectly through the SetScrollInfo routine.
4. tagWND's strName.Buffer field fully under control means that the exploit code achieves arbitrary kernel memory read and write.

Obstacles solutions also can be summarized as follows:

1. Manipulating the tagWND.pSBInfo field by pointing it to the tagWND.strName, and then rewriting tagWND's strName.Buffer field indirectly through SetScrollInfo means obstacle 1 is solved.
2. The full control of the Zombie tagPROPLIST object means obstacle 2 is solved.

32-bit exploit process as shown below:



NCC Group's 32-Bit Exploit Logic

NCC Group's 64-bit Exploit Method

Note that the heap header (_HEAP_ENTRY) will be completely overwritten when calling SetProp on a 64-bit Operating System. Aaron Adams's 64-bit exploit method changes to use U-A-F tagPROPLIST, Window Text 1, tagWND and plus Window Text 2 objects for heap Feng Shui.

The exploit process still begins from the U-A-F tagPROPLIST object. SetProp's Out-Of-Bounds write capability is used to overwrite Window Text 1's _HEAP_ENTRY with a specially crafted replacement _HEAP_ENTRY. In order to maintain a valid heap layout[8], a fake _HEAP_ENTRY is also stored in Window Text 2.

This solution not only solved the problem of a system crash caused by heap header corruption, it also has the side effect that the tagWND object will be completely merged into the Window Text 1 heap block. This is a very important point because freeing the Window Text 1 also means that the tagWND object will be freed at the same time. Additionally, due to the User32!gSharedInfo information disclosure problem, an exploit can easily forge a new tagWND object. The difference between the new (forged) tagWND object and the old one is tagWND's strName.Buffer field is fully under an attacker's control. From another point of view, this exploit process can also be seen as a user-created or forced Use-After-Free. The semantics of the tagWND's strName.Buffer field changes when memory free operation is triggered.

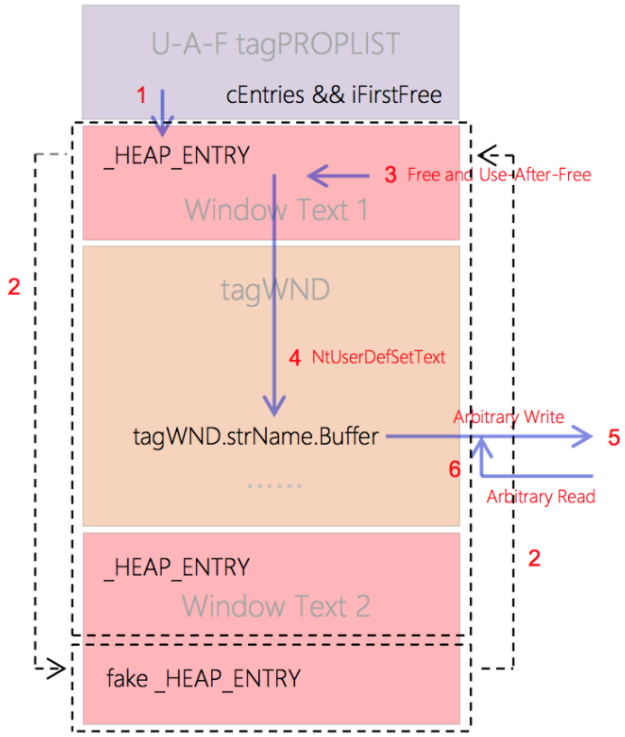
64-bit exploitation process can be summarized as follows:

1. Overwrite Window Text 1 block's _HEAP_ENTRY based on the U-A-F tagPROPLIST object's Out-Of-Bounds write.
2. Reconstruct an ideal heap layout by using a specially crafted _HEAP_ENTRY for Window Text 1 and another faked _HEAP_ENTRY which is stored in the Window Text 2.
3. Due to the fake _HEAP_ENTRY for Window Text 1, a free operation on Window Text 1 causes following tagWND object to be freed as well.
4. Rebuild a new tagWND object based on information leaked from User32!gSharedInfo. The difference between the new tagWND object and the old object is that the new tagWND's strName.Buffer field is fully under control. This process can also be seen as a user forced Use-After-Free process.
5. Finally, having the tagWND's strName.Buffer field under control means that the exploit code is able to read and write arbitrary kernel memory.

Obstacles solutions also can be summarized as follows:

1. The faking of heap headers leading to the control of tagWND's strName.Buffer means that obstacle 1 is resolved.
2. The forced Use-After-Free technique and the re-creation of a new tagWND based on kernel information disclosed by User32!gSharedInfo cleverly bypasses obstacle 2.

64-bit exploit process as shown below:



NCC Group's 64-Bit Exploit Logic

As can be seen, NCC Group's 32-bit exploit technique converts a Use-After-Free bug into a relative heap memory read and write primitive. They then extend this exploit primitive to achieve full arbitrary kernel memory read and write. Their 64-bit exploitation method starts off in a similar manner, also converting a Use-After-Free bug into a relative heap memory read and write primitive. However, for 64-bit exploits, they converted this read and write primitive into a forced Use-After-Free. This allows them to achieve object hijacking which as we saw, finally led to having full kernel read and write capabilities.

Both the 32 bit and 64 bit exploits ultimate aim was to corrupt the tagWND's strName.Buffer field and they both relied on information leaked through User32!gSharedInfo to a some extent.

A New CVE-2015-0057 Exploit Technology

As a response to Aaron Adams's inquiry, I would like to share with you a few different exploit techniques and as an expression of gratitude to Aaron Adams. The following technique can work stably on both 32 and 64-bit platforms from Windows XP to Windows 10 Technical Preview.

32-bit Exploit Method

Similar as the Aaron's method, my 32-bit exploit method uses tagPROPLIST and tagMENU objects to perform heap Feng Shui. The exploit also sprays tagPROPLIST twice at least. Let's call them U-A-F and Zombie tagPROPLIST object respectively. tagMENU is the host object ultimately used to achieve arbitrary kernel memory read and write. Similarly, I chose the tagMENU object because the tagMENU's rgItems field has ability to read and write bytes in kernel memory.

The exploit process begins from the U-A-F tagPROPLIST object. By calling the SetProp routine on the U-A-F tagPROPLIST object, we can effectively control the adjacent Zombie tagPROPLIST object. Thereafter, subsequent invocations of SetProp on the Zombie tagPROPLIST object actually let the exploit code gain the ability to write kernel memory accurately. Compared with Aaron Adams's scheme, the Zombie tagPROPLIST object gives us better control over the tagMENU.rgItems and tagMENU.cItems fields. Thus the exploit process is relatively simple and works without the help of User32!gSharedInfo's kernel information disclosure.

32-bit exploitation process can be summarized as follows:

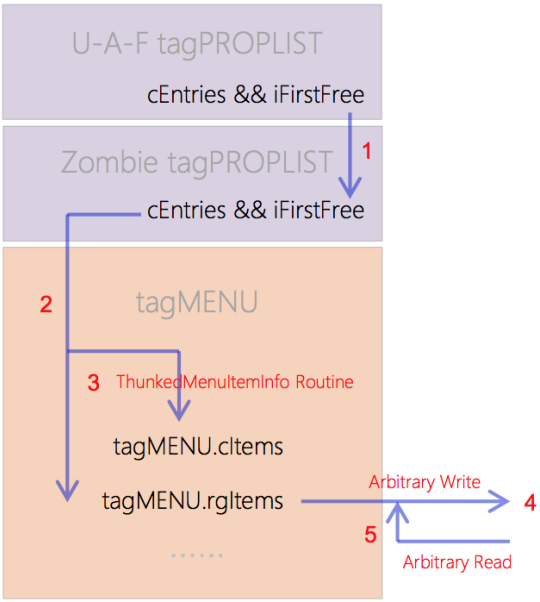
1. Use the U-A-F tagPROPLIST object's Out-Of-Bounds write capabilities to manipulate Zombie tagPROPLIST's properties.
2. Use the corrupted Zombie tagPROPLIST object to rewrite the adjacent tagMENU object's rgItems and cItems fields.
3. tagWND's rgItems field fully under control means that the exploit code achieves arbitrary kernel memory read and write.

Obstacles solutions also can be summarized as follows:

1. The full control of the tagMENU.rgItems and tagMENU.cItems fields means obstacle 1 is solved.

2. The full control of the Zombie tagPROPLIST object means obstacle 2 is solved.

32-bit exploit process which based on the tagMENU object as shown below:



Another Way to Exploit CVE-2015-0057 on 32-bit Windows

64-bit Exploit Method - The write control capability of the misaligned tagPROPLIST object

With reference to obstacle 1, if we want to reuse the above scheme on 64-bit platforms we will find that the U-A-F tagPROPLIST object actually can't control the Zombie tagPROPLIST's iFirstFree field. This seems to be a fatal blow to the exploitation of the vulnerability. Losing control of the iFirstFree field means that we are unable to achieve exact offset writing based on SetProp and continuous memory corruption seems to be inevitable. This is also the problem which was described in the obstacle 2. In order to solve this problem, my 64-bit exploit method uses U-A-F tagPROPLIST, Zombie tagPROPLIST, tagWND and tagMENU objects to perform heap Feng Shui. It is understood that tagPROPLIST is pointed by the ppropList field of the tagWND object. On 64-bit platforms, we happen to have full control over tagWND.ppropList.

If it were possible to point the tagWND.ppropList field to our pre-constructed, fake tagPROPLIST object, it will mean that we can achieve relative heap memory reads and writes.

First, the areas controlled by the Zombie tagPROPLIST object is shown in blue:

```
0:  kd> dq fffff900‘c085dfc8 l24
fffff900‘c085dfc8  08000004‘00010003 00000002‘0000000e  /* U-A-F tagPROPLIST */
fffff900‘c085dfd8  00000000‘025300b0 00000000‘0000a918
fffff900‘c085dfe8  00000008‘00000008 00000000‘00004190

fffff900‘c085dff8  08000003‘00010003 00000007‘00000007 /* Zombie tagPROPLIST */
fffff900‘c085e008  00000000‘02510600 00000000‘0000a918
fffff900‘c085e018  00000000‘00000000 00000000‘00004131

fffff900‘c085e028  10000003‘00010014 00000000‘00020536       /* tagWND */
fffff900‘c085e038  00000000‘00000003 fffff900‘c0723490
fffff900‘c085e048  fffffa80‘0c3a91d0 fffff900‘c085e030
fffff900‘c085e058  80000700‘40000018 04cf0000‘00000100
fffff900‘c085e068  00000000‘00000000 00000000‘02a80000
fffff900‘c085e078  fffff900‘c085a3d0 fffff900‘c08070c0
fffff900‘c085e088  fffff900‘c0800b90 00000000‘00000000
fffff900‘c085e098  00000000‘00000000 00000000‘00000000
fffff900‘c085e0a8  00000026‘00000084 0000001e‘00000008
fffff900‘c085e0b8  0000001e‘0000007c 00000000‘7761946c
fffff900‘c085e0c8  fffff900‘c083ff70 00000000‘00000000
fffff900‘c085e0d8  fffff900‘c085dfe8 00000000‘000000aa   /* tagWND.ppropList */
```

Control Area of the Zombie tagPROPLIST

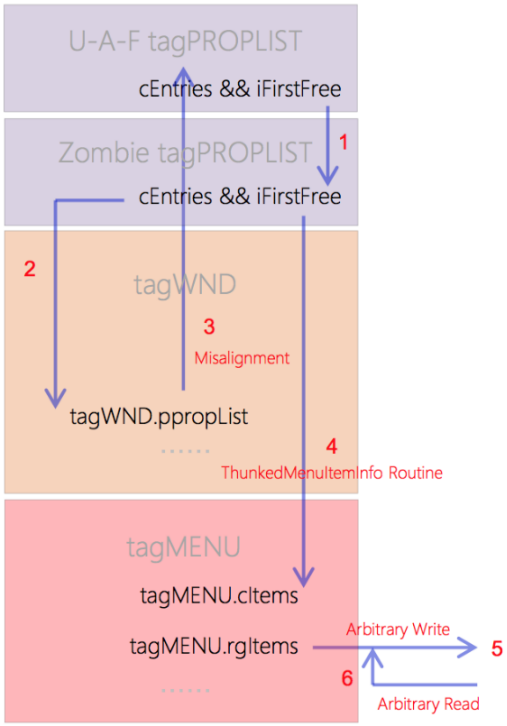
Next, we see that the control capability of the fake tagPROPLIST and Zombie tagPROPLIST complement each other well:

```
     0:  kd> dq fffff900‘c085dfc8 l24
     fffff900‘c085dfc8  08000004‘00010003 00000002‘0000000e  /* U-A-F tagPROPLIST */
     fffff900‘c085dfd8  00000000‘025300b0 00000000‘0000a918
+--> fffff900‘c085dfe8  00000008‘00000008 00000000‘00004190
|
|    fffff900‘c085dff8  08000003‘00010003 00000007‘00000007 /* Zombie tagPROPLIST */
|    fffff900‘c085e008  00000000‘02510600 00000000‘0000a918
|    fffff900‘c085e018  00000000‘00000000 00000000‘00004131
|
|    fffff900‘c085e028  10000003‘00010014 00000000‘00020536       /* tagWND */
|    fffff900‘c085e038  00000000‘00000003 fffff900‘c0723490
|    fffff900‘c085e048  fffffa80‘0c3a91d0 fffff900‘c085e030
|    fffff900‘c085e058  80000700‘40000018 04cf0000‘00000100
|    fffff900‘c085e068  00000000‘00000000 00000000‘02a80000
|    fffff900‘c085e078  fffff900‘c085a3d0 fffff900‘c08070c0
|    fffff900‘c085e088  fffff900‘c0800b90 00000000‘00000000
|    fffff900‘c085e098  00000000‘00000000 00000000‘00000000
|    fffff900‘c085e0a8  00000026‘00000084 0000001e‘00000008
|    fffff900‘c085e0b8  0000001e‘0000007c 00000000‘7761946c
|    fffff900‘c085e0c8  fffff900‘c083ff70 00000000‘00000000
+--o fffff900‘c085e0d8  fffff900‘c085dfe8 00000000‘000000aa   /* tagWND.ppropList */
```

Control Area of the Misaligned tagPROPLIST

After rewriting the tagWND's ppropList field to point to our crafted fake tagPROPLIST object, the only thing left to do is to use the fake tagPROPLIST and the Zombie tagPROPLIST alternately. We can then achieve the relative heap memory read and write primitive.

64-bit exploit process as shown below:



Another Way to Exploit CVE-2015-0057 on 64-bit Windows

64-bit exploitation process can be summarized as follows:

1. Use the U-A-F tagPROPLIST object's Out-Of-Bounds write capabilities to manipulate Zombie tagPROPLIST's properties.
2. Use the corrupted Zombie tagPROPLIST object to rewrite the adjacent tagWND object's ppropList field.
3. Point the tagWND.ppropList field to a fake tagPROPLIST object containing user-controlled 'misaligned' values.
4. Use the Zombie tagPROPLIST and the misaligned tagPROPLIST object alternately, this will allow the exploit code to have the ability to modify the rgItems and cItems fields of the tagMENU object.

5. With tagMENU's rgItems field fully under control, the exploit code achieves arbitrary kernel memory read and write.

Obstacles solutions also can be summarized as follows:

1. Using the Zombie tagPROPLIST and the crafted fake tagPROPLIST object alternately to modify the rgItems and cItems fields of tagMENU means that obstacle 1 is solved.
2. Having full control over the cEntries and iFirstFree fields of the Zombie tagPROPLIST object means that obstacle 2 is solved.

The 32 and 64-bit exploit process described above both try to convert the Use-After-Free into a relative heap memory read and write vulnerability. Next, exploit process will convert the relative read and write vulnerability to achieve full arbitrary kernel memory read and write. In both techniques, having control over tagMENU.rgItems and tagMENU.cItems is the core of the exploit. That is because the structure array rgItems and its corresponding methods are required in order to achieve our read and write primitives. This is the same reason why the tagWND.strName.Buffer was selected before.

The 32-bit exploit process is relatively simple and does not require using User32!gSharedInfo's kernel information disclosure. The 64-bit exploit process is somewhat more complicated. By constructing tagWND.ppropList and pointing it to the fake tagPROPLIST object, the exploit code eventually overcomes the restricted write capability of SetProp.

The Others

In order to achieve high reliability of the exploit, we also need to solve some small but important problems, such as: heap data repair, interference from 64-bit platform's memory alignment, interference from the CThemeWnd::_AttachInstance and so on.

Let's take the interference from memory alignment as an example. When trying to perform heap Feng Shui, we will have a certain probability of facing the memory alignment issue. This behavior will lead to a chain reaction, such as the changes in _HEAP_ENTRY, the overwrite offsets, etc. Fortunately, based on kernel information disclosed by the User32!gSharedInfo, we can predict the details of the current heap memory layout and modify the exploit code's logic dynamically.

```
0:  kd> dq fffff900‘c0841f58 fffff900‘c0841ff0
fffff900‘c0841f58  08000003‘00010003 00000002‘00000002 /* chunk #01 */
fffff900‘c0841f68  00000000‘02f7ad30 00000000‘0000a918
fffff900‘c0841f78  00000000‘00000000 00000000‘00004136

fffff900‘c0841f88  18000003‘00010004 00000002‘00000002 /* chunk #02 */
fffff900‘c0841f98  00000000‘02f7b2c0 00000000‘0000a918
fffff900‘c0841fa8  00000000‘00000000 00000000‘00004137
fffff900‘c0841fb8  00000000‘00000000 00000000‘00000000 /* alignment */

fffff900‘c0841fc8  08000004‘00010003 00000002‘00000002 /* chunk #03 */
fffff900‘c0841fd8  00000000‘02f9b850 00000000‘0000a918
fffff900‘c0841fe8  00000000‘00000000 00000000‘00004190
```

0x10 Bytes Memory Alignment

In addition, there is a tiny flaw in the Udi Yavo's CVE-2015-0057 technical blog[5], which I will point out here. At the end of the blog, Udi mentioned that there is a piece of dead-code residing in Win32k!xxxEnableWndSBArrows over 15-years:

"Looking at the code, there are two conditional calls to the function, xxxWindowEvent. These calls are executed only if the old flags of the scrollbar information differ from those of the new flags. However, by the time these conditions appear, the values of the old flags and the new flags are always equal. Hence, the condition for calling xxxWindowEvent is never met. This practically means that this dead-code was there for about 15-years doing absolutely nothing."

These conclusions could be drawn based on the following logic:

```
1   ......
2
3   /*
4    * update the display of the horizontal scroll bar
5    */
6
7   if (pw->WSBflags != wOldFlags) {
8       bRetValue = TRUE;
9       wOldFlags = pw->WSBflags;
10      if (TestWF(pwnd, WFHPRESENT) && !TestWF(pwnd, WFMINIMIZED) &&
11          IsVisible(pwnd)) {
12          xxxDrawScrollBar(pwnd, hdc, FALSE);
13      }
14  }
15
16  if (FWINABLE()) {
17      /* left button */
18      if ((wOldFlags & ESB_DISABLE_LEFT) != (pw->WSBflags & ESB_DISABLE_LEFT)) {
19          xxxWindowEvent(EVENT_OBJECT_STATECHANGE, pwnd, OBJID_HSCROLL,
20                         INDEX_SCROLLBAR_UP, WEF_USEPWNDTHREAD); /* dead-code? */
21      }
22
23      /* right button */
24      if ((wOldFlags & ESB_DISABLE_RIGHT) != (pw->WSBflags & ESB_DISABLE_RIGHT)) {
25          xxxWindowEvent(EVENT_OBJECT_STATECHANGE, pwnd, OBJID_HSCROLL,
26                         INDEX_SCROLLBAR_DOWN, WEF_USEPWNDTHREAD); /* dead-code? */
27      }
28  }
29
30  ......
```

Win32k!xxxEnableWndSBArrows Pseudocode

The seventh line of the code (the conditional statement) and the ninth line of the code (the assignment statement) are deceptive. It appears to assign WSBflags to wOldFlags and so it seems that the conditions on the eighteenth line and the twenty-fourth line will never be true. But, is this truly the case?

Please do not forget, User Mode Callback is the root cause of many state inconsistency type vulnerabilities in Win32K. In this example, to activate the code branch of xxxWindowEvent, attackers only need to enable the scroll bar within the xxxDrawScrollBar callback. Not only is the xxxWindowEvent branch NOT dead code, it is also another entry point of the vulnerability. If we can invoke SetWinEventHook in the exploit and then call DestroyWindow during event callback, we can trigger another Use-After-Free. The call stack as shown below:

```
0:  kd> kb
ChildEBP RetAddr Args to Child
f52838e8 bf8faf21 bc675e20 0012fbcc f5283904 win32k!xxxDestroyWindow
f52838f8 8054160c 00030124 0012fc98 7c92eb94 win32k!NtUserDestroyWindow+0x21
f52838f8 7c92eb94 00030124 0012fc98 7c92eb94 nt!KiFastCallEntry+0xfc
0012fbbc 77d1e672 0042c7ad 00030124 0012fd58 ntdll!KiFastSystemCallRet
0012fc98 77d5906d 00130103 0000800a 00030124 USER32!NtUserDestroyWindow+0xc
```

```
0012fcc8 7c92eae3 0012fcd8 00000020 0042a762 USER32!__ClientCallWinEventProc+0x2a
0012fcc8 80501a60 0012fcd8 00000020 0042a762 ntdll!KiUserCallbackDispatcher+0x13
f5283bbc 805a1779 f5283c68 f5283c64 bf9a2ccc nt!KiCallUserMode+0x4
f5283c18 bf92c55a 00000050 f5283c44 00000020 nt!KeUserModeCallback+0x87
f5283c88 bf91ccb4 0042a762 e10740a0 bf9a2ccc win32k!xxxClientCallWinEventProc+0x68
f5283cb8 bf8081e8 bf9a2ccc bc675f18 bc675e20 win32k!xxxProcessNotifyWinEvent+0xb9
f5283cfc bf8d136b 0000800a 00000000 fffffffa win32k!xxxWindowEvent+0x182
f5283d2c bf91140e 00000003 00000003 00000003 win32k!xxxEnableWndSBArrows+0xaf
f5283d50 8054160c 00030124 00000003 00000003 win32k!NtUserEnableScrollBar+0x69
f5283d50 7c92eb94 00030124 00000003 00000003 nt!KiFastCallEntry+0xfc
0012fcc8 7c92eae3 0012fcd8 00000020 0042a762 ntdll!KiFastSystemCallRet
0012fcf4 77d6c6ee 5adeb71f 00030124 00000003 ntdll!KiUserCallbackDispatcher+0x13
0012fd14 77d67c01 00030124 00000003 00000003 USER32!NtUserEnableScrollBar+0xc
0012fd54 0042c663 00030124 00000003 00000003 USER32!EnableScrollBar+0x54
0012fe88 0042c807 00400000 80000001 0007ddb4 cve_2015_0057+0x2c663
0012ff60 0042ca4b 00400000 00000000 0015234b cve_2015_0057+0x2c807
0012ffb8 0042c91d 0012fff0 7c816d4f 80000001 cve_2015_0057+0x2ca4b
0012ffc0 7c816d4f 80000001 0007ddb4 7ffdf000 cve_2015_0057+0x2c91d
0012fff0 00000000 0042ad7a 00000000 78746341 kernel32!BaseProcessStart+0x23
```

xxxWindowEvent Use-After-Free Call Stack

Fortunately, Microsoft's developers realized the problem. MS15-010 blocks five possible entrances of the vulnerability in total, completely covering these seemingly dead code branches.

```
35    if ( pw->WSBflags != wOldFlags )
36    {
37      v7 = 1;
38      wOldFlags = pw->WSBflags;
39      if ( pwnd->state & 4 )
40      {
41        if ( !(BYTE3(pwnd->style) & 0x20) )
42        {
43          if ( IsVisible(pwnd) )
44          {
45            xxxDrawScrollBar(pwnd, hdc, 0);
46            if ( pw != pwnd->pSBInfo )
47              goto LABEL_42;
48          }
49        }
50      }
51    }
52    if ( (wOldFlags ^ LOBYTE(pw->WSBflags)) & ESB_DISABLE_LEFT )
53    {
54      xxxWindowEvent(EVENT_OBJECT_STATECHANGE, pwnd, OBJID_HSCROLL, 1, 1);
55      if ( pw != pwnd->pSBInfo )
56        goto LABEL_42;
57    }
58    if ( (wOldFlags ^ LOBYTE(pw->WSBflags)) & ESB_DISABLE_RIGHT )
59    {
60      xxxWindowEvent(EVENT_OBJECT_STATECHANGE, pwnd, OBJID_HSCROLL, 5, 1);
61      if ( pw != pwnd->pSBInfo )
62        goto LABEL_42;
63    }
```

Win32k!xxxEnableWndSBArrows, MS15-010

Conclusion

From CVE-2015-0057, CVE-2015-1701 to CVE-2015-2360[15], the disclosure of Win32K subsystem's zero-day vulnerabilities reminds us that User Mode Callback, the old-school Windows kernel attack surface, cannot be neglected. On the 8th of September 2015, the CVE-2015-2545 and the CVE-2015-2546 vulnerabilities were exposed[4]. These APT zero-day exploits leveraged the PostScript embedded Microsoft Office document as an attack vector, and Use-After-Free vulnerability on the User Mode Callback mechanism as the way of privilege escalation. Looking at the operation mechanism of these vulnerabilities, I think of the following aspects worth pondering:

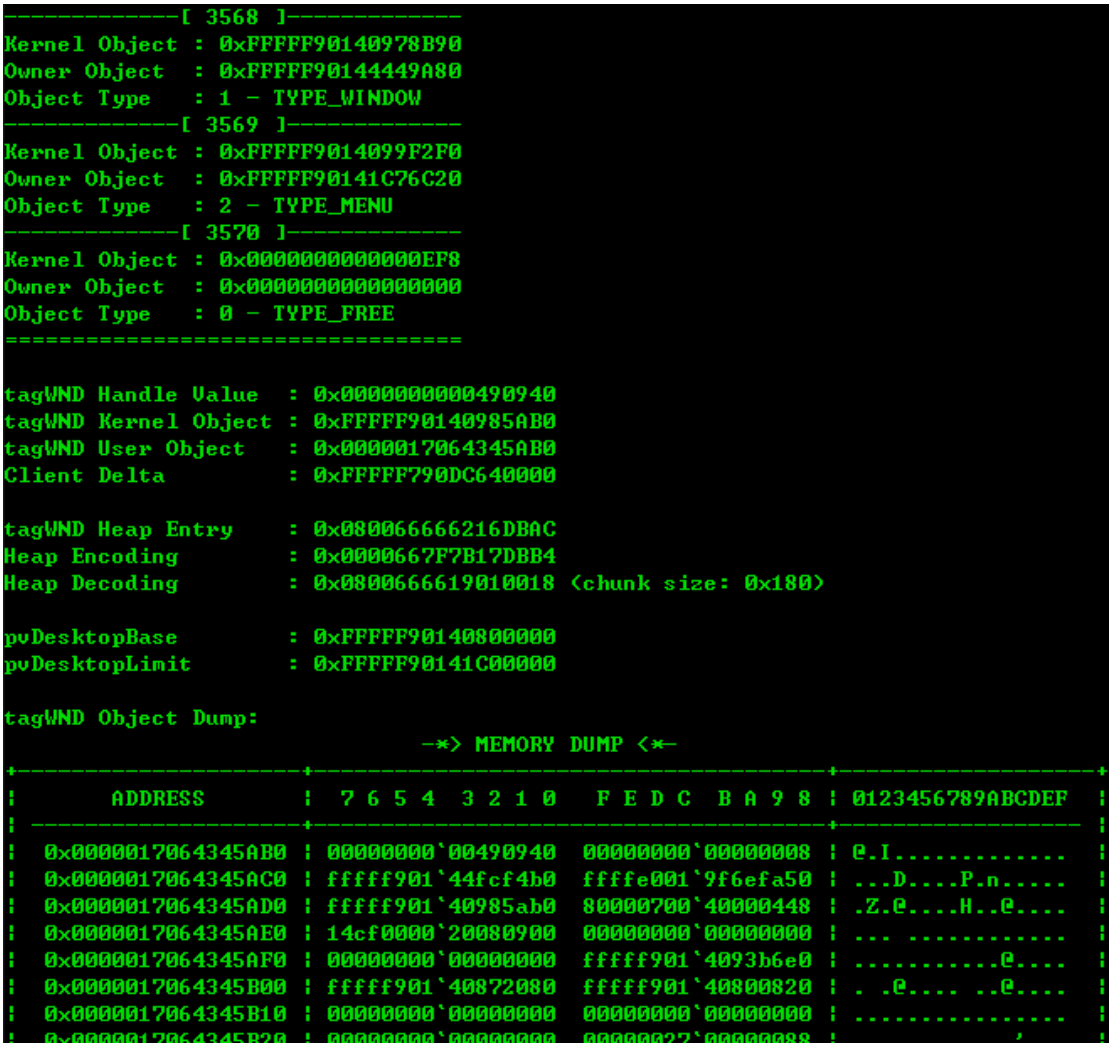
The legacy kernel code need to be re-audited carefully. Many of the above Use-After-Free vulnerability history can be traced back to Windows 2000 or even Windows NT era. Although, Microsoft corporation implemented a lot of improvements and mitigations for Windows kernel security in recent years, However, we still see that there is still a room for improvement. For instance, the design of Win32K subsystem's lock mechanism, the life cycle management of window related objects and the gSharedInfo sharing mechanism and so on.

Let's take gSharedInfo as an example, this mechanism should be reconsidered. From the disclosure of zero-day vulnerabilities used by APT actors to the latest Open-Type zero-day exploit code leaked by Hacking Team[16], we continuously saw the use of User32!gSharedInfo mechanism. For the Desktop Heap and the Shared Heap, all efforts of kernel randomization will come to naught due to the gSharedInfo sharing mechanism. Even the kernel _HEAP_ENTRY random encoding[14] which was introduced in Windows 8 will also be leaked to User Mode. If the mechanism cannot be disabled due to compatibility issues, Microsoft should minimize the use of gSharedInfo.

In order to demonstrate the risk of the User32!gSharedInfo mechanism, I wrote a demo tool. On 64-bit Windows 10 system, this tool will demonstrate that leaked kernel data is not only limited to the following information. From an attacker's perspective, these leaked data will greatly facilitate additional kernel exploits:

- Kernel Routine Address Information
- Kernel Data Structure Information

- Kernel Object Memory Layout Information
- Desktop Heap Information
- Kernel _HEAP_ENTRY Random Cookie Information



User32!gSharedInfo Information Disclosure

Finally, the activation of the Win32k!xxxWindowEvent code branch is a vivid example of the complexity of User Mode Callback mechanism. Another wonderful example comes from the CVE-2015-1701. It reminds us that unlike the traditional Use-After-Free, NULL Pointer Dereference vulnerability, the inconsistency of User Mode Callback is rather more mysterious side of the vulnerability. As a defender, we must always maintain a clear understanding.

References

[1] Microsoft Security Bulletin MS15-010
https://technet.microsoft.com/en-us/library/security/ms15-010.aspx

[2] Operation RussianDoll: Adobe & Windows Zero-Day Exploits Likely Leveraged by Russia's APT28 in Highly-Targeted Attack
https://www.fireeye.com/blog/threat-research/2015/04/probable_apt28_useo.html

[3] Dyre Banking Trojan Exploits CVE-2015-0057
https://www.fireeye.com/blog/threat-research/2015/07/dyre_banking_trojan.html

[4] Two for One: Microsoft Office Encapsulated PostScript and Windows Privilege Escalation Zero-Days
https://www.fireeye.com/blog/threat-research/2015/09/attack_exploitingmi.html

[5] One-Bit To Rule Them All: Bypassing Windows'10 Protections using a Single Bit
http://breakingmalware.com/vulnerabilities/one-bit-rule-bypassing-windows-10-protections-using-single-bit

[6] Exploiting the Win32k!xxxEnableWndSBArrows Use-After-Free (CVE-2015-0057) Bug on Both 32-Bit and 64-Bit
https://www.nccgroup.trust/globalassets/newsroom/uk/blog/documents/2015/07/exploiting-cve-2015.pdf

[7] Analyzing Local Privilege Escalations in Win32K
http://www.uninformed.org/?a=2&t=sumry&v=10

[8] Kernel Pool Exploitation on Windows 7
https://media.blackhat.com/bh-dc-11/Mandt/BlackHat_DC_2011_Mandt_kernelpool-wp.pdf

https://media.blackhat.com/bh-dc-11/Mandt/BlackHat_DC_2011_Mandt_kernelpool-Slides.pdf

[9] Kernel Attacks through User-Mode Callbacks
http://blogs.norman.com/2011/for-consumption/kernel-attacks-through-user-mode-callbacks

[10] Afd.sys Dangling Pointer Vulnerability
http://www.siberas.de/papers/Pwn2Own_2014_AFD.sys_privilege_escalation.pdf

[11] Kernel Exploitation - R0 to R3 Transitions via KeUserModeCallback
http://j00ru.vexillium.org/?p=614

[12] EPATHOBJ Local Ring - Exploit
https://www.exploit-db.com/exploits/25912/

[13] Heap Feng Shui in JavaScript
https://www.blackhat.com/presentations/bh-usa-07/Sotirov/Whitepaper/bh-usa-07-sotirov-WP.pdf
https://www.blackhat.com/presentations/bh-usa-07/Sotirov/Presentation/bh-usa-07-sotirov.pdf

[14] Exploit Mitigation Improvements in Windows 8
https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf

[15] The Mystery of Duqu 2.0: A Sophisticated Cyberespionage Actor Returns
https://securelist.com/blog/research/70504/the-mystery-of-duqu-2-0-a-sophisticated-cyberespionage-actor-returns/

[16] CVE-2015-2426, OpenType Font Driver Vulnerability
http://web.archive.org/web/20150707012022/https://ht.transparencytoolkit.org/gitlab/Windows-Multi-Browser/demo_calc.exe/source_shellcode/