前置知识：VC
关键词：编程、通告例程、驱动

使用通告例程监控驱动及 DLL 加载

文/ NewbieCoder[0GiNr]

随着隐藏进程检测技术的进步，木马也转而向无进程方向发展。通常，它们会将代码编译成一个 DLL（在本文中，DLL 特指用户模式下的动态链接库）或者一个驱动而随其他进程加载(驱动被加载到 System 进程),所以如何监控驱动以及 DLL 加载成为了防马的重要一环。本文将介绍一种通过使用 Windows NT 系统所提供的注册通告例程来监控驱动以及 DLL 加载的方法。

在驱动中存在两种相似的可以被注册的例程：回调（Callback）以及通告（Notification)。其中，回调函数拥有返回值，可以通过返回值来表明允许或是拒绝正在执行的操作，而通告函数只是告知驱动一个操作正在发生。通常而言，通告函数没有返回值，因此驱动无法阻绝执行正在执行的操作（Vista Service Pack 1 新引入的 PsSetCreateProcessNotifyRoutineEx 是一个特例，通过该函数所注册的通告函数 CreateProcessNotifyEx，可以通过修改参数 CreateInfo 的成员 CreationStatus 来控制进程创建，这有可能是为了在函数名上向前兼容而导致的)。但是甚至于在 WDK 的说明文档中，这两个术语也经常被混用。广义上而言，通告也是一种回调，但这种回调不会对函数执行产生影响，因此，本文将会把注册的加载模块通告函数称之为回调。

注册回调

注册模块加载回调通过 PsSetLoadImageNotifyRoutine(LoadImageNotifyRoutine)即可完成，之后每当系统中有新的驱动或者 DLL 被加载，LoadImageNotifyRoutine 就会被调用，其参数 FullImageName 指明了正在加载的模块文件名（但是信息不一定完全，具体后文会谈到），ProcessId 指明了加载该模块的进程 PID，ImageInfo 的成员 SystemModeInfo 指明了是否为内核模块（通常为驱动)。

模块文件名

1）内核模块的文件名

对于内核模块而言，LoadImageNotifyRoutine 的参数 FullImageName 即指明了该内核模块的文件名,但并不是我们常见的 DOS 风格的文件名(DOS 风格的文件名是指“C:\Xxx.Xxx”这样的文件名)，通过一些转换即可获取该模块的 DOS 风格文件名：

对于“\??\X:\xxx\xxx”，去掉前缀“\??\”即可（前面的“\??”实际上是“DosDevices”这个符号链接的目标)；对于“\SystemRoot\xxx”，将“\SystemRoot\”替换为 Windows 目录（SystemRoot 实质上是一个符号链接，目标是 Windows 目录)；对于“\WINDOWS\system32\xxx”，在字符串前端加上 Windows 所在分区的盘符即可。

这是一项虽然繁琐但并不困难的工作，本文不多讨论。对于 1、2 两种情况，均可使用后文所提到的转换 NT 风格路径的方法来转换为 DOS 风格路径。

2）DLL 的文件名

尽管本文主要讨论的是模块加载，但由于创建进程时，EXE 自身也会被系统作为模块的加载来处理，换句话说，也会引发回调，因此，EXE 的加载将在此处一起被讨论。

从 Vista 起，IMAGE_INFO 结构中多了成员 ExtendedInfoPresent，当

ExtendedInfoPresent=1 时，可以通过 CONTAINING_RECORD(ImageInfo,IMAGE_INFO_EX,ImageInfo)得到 IMAGE_INFO_EX 结构的数据，其中多出了 FileObject，我们可以直接通过 FileObject 方便的获取信息。以下只讨论 Vista 之前系统的情况。

对于 DLL（包括 EXE）的加载，可以分为三种情况。

①对于 EXE 文件自身

由于 EXE 以及 ntdll.dll 是由系统在进程开始执行之前就映射入进程空间的，因此这两个文件的加载回调被系统特殊处理（普通的 DLL 加载是通过 ZwMapViewOfSection（如未特殊指明，本文中的 ZwMapViewOfSection 均特指 ntdll!ZwMapViewOfSection）加载的，而 EXE 是在创建进程空间之后通过 MmInitializeProcessAddressSpace 映射入进程空间的，ntdll.dll 是在进程创建了进程空间之后由 PsMapSystemDll 映射入进程空间的（均在 PspCreateProcess（以及该函数所调用的函数）中完成））。

通过在 LoadImageNotifyRoutine 上下断，可以看到如下调用堆栈：

```
LoadImageNotifyDriver!LoadImageNotifyRoutine
nt!PsCallImageNotifyRoutines+0x36
nt!DbgkCreateThread+0xa2
nt!PspUserThreadStartup+0x9d
nt!KiThreadStartup+0x16
```

一层一层向上回溯：

```
nt!PsCallImageNotifyRoutines+0x2b:
push    dword ptr [ebp+10h]
push    dword ptr [ebp+0Ch]
push    dword ptr [ebp+8]
//回调函数的参数 FullImageName，也是 PsCallImageNotifyRoutine 的第一个参数
call    eax //调用回调函数
```

再向上回溯一层：

```
nt!DbgkCreateThread+0x7d:
lea     eax,[ebp-38h]
push    eax //参数 FileNameInfo
push    dword ptr [esi+138h] //参数 SectionObject
call    nt!MmGetFileNameForSection (8059d646)
cmp     eax,ebx // ebx = 0
lea     eax,[ebp-2Ch]
push    eax
push    dword ptr [esi+84h]
jl      nt!DbgkCreateThread+0xad (8063909d)
//若 MmGetFileNameForSection 失败则跳走

nt!DbgkCreateThread+0x9a:
push    dword ptr [ebp-38h]
```

```
//前面由 MmGetFileNameForSection 填充
call    nt!PsCallImageNotifyRoutines (805c5b40) //调用回调
```

可以看到，对于 EXE，FullImageName 传入的是 MmGetFileNameForSection 传出的 FileNameInfo，其内容是 EXE 的完整路径（NT 风格路径，类似于“\Device\HarddiskVolume1\xxx”，NT 风格路径通过转换可以得到 DOS 风格路径（转换方法将在后文提到））。

②对于 ntdll.dll

对于 ntdll 而言，回调同样是从 DbgkCreateThread 发起的，查看调用堆栈：

```
LoadImageNotifyDriver!LoadImageNotifyRoutine
nt!PsCallImageNotifyRoutines+0x36
nt!DbgkCreateThread+0x125
nt!PspUserThreadStartup+0x9d
nt!KiThreadStartup+0x16
```

回溯到 DbgkCreateThread：

```
nt!DbgkCreateThread+0xfe:
push    offset nt!DbgkpSectionToFileHandle+0x70 (80638f96)
//要初始化的 UNICODE 字符串
lea     eax,[ebp-0E4h]
push    eax
call    nt!RtlInitUnicodeString (8052af2c)

nt!DbgkCreateThread+0x10f:
lea     eax,[ebp-2Ch]
push    eax
push    dword ptr [esi+84h]
lea     eax,[ebp-0E4h]
//前面使用 RtlInitUnicodeString 初始化的字符串
push    eax
call    nt!PsCallImageNotifyRoutines (805c5b40) //调用回调
kd> du nt!DbgkpSectionToFileHandle+0x70
"\SystemRoot\System32\ntdll.dll" //这就是被初始化的字符串
```

可以看出，对于 ntdll.dll，FullImageName 被硬编码为“\SystemRoot\System32\ntdll.dll”，按照前文处理内核模块文件名的方法，可以转换为 DOS 风格路径。

为了方便编码，我没有对 EXE 以及 ntdll 做特殊处理，而是将 EXE、ntdll 以及其他模块使用相同的方法获取文件名（详见后文）。

③对于普通 DLL

无论是程序启动时加载的 DLL（ntdll.dll 除外，其余的启动时加载的 dll 均是由 ntdll.dll 加载的)还是后来加载以及注入的 DLL,均是通过 ZwMapViewOfSection 来加载的。

同样，我们先查看调用堆栈：

```
LoadImageNotifyDriver!LoadImageNotifyRoutine
nt!PsCallImageNotifyRoutines+0x36
nt!MiMapViewOfImageSection+0x4c1
nt!MmMapViewOfSection+0x13c
nt!NtMapViewOfSection+0x2bd
nt!KiFastCallEntry+0xf8
ntdll!KiFastSystemCallRet
ntdll!NtMapViewOfSection+0xc
//ntdll 中 ZwMapViewOfSection 与 NtMapViewOfSection 是同一个函数
```

由于 MiMapViewOfImageSection 的反汇编太长，因此不再贴出，其内部通过 ControlArea（由 MmMapViewOfSection 产生）->FilePointer 得到被映射的文件的 FileObject，然后将 FileObject->FileName 作为 FullImageName 传给 PsCallImageNotifyRoutine，因此得到的 FullImageName 没有盘符，只有路径以及文件名（类似于“\WINDOWS\system32\kernel32.dll”），显然这样的信息不够找到一个文件，因此我们需要通过其他方法来获取文件名。

我们知道，ZwQueryVirtualMemory 可以根据 VA（Virtual Address，虚拟地址）来获取映射到该地址的文件的文件名（其内部会遍历 VAD 树找到该 VA 对应的树枝，然后得到 ControlArea，进而得到 FileObject），因此，从理论上讲，我们可以通过 ZwQueryVirtualMemory(ZwCurrentProcess(),MemoryMappedFilenameInformation,ImageFileName,BufferSize,&BufferSize)来获得文件名。但是实际操作后会发现，我们得不到任何输出，加载模块的线程也不会继续执行。查看该线程的调用堆栈：

```
nt!KiSwapContext+0x2e
nt!KiSwapThread+0x46
nt!KeWaitForSingleObject+0x1c2
hal!ExAcquireFastMutex+0x2a
nt!NtQueryVirtualMemory+0x1a9
nt!KiFastCallEntry+0xf8
LoadImageNotifyDriver!CallSystemService+0x11
LoadImageNotifyDriver!NewZwQueryVirtualMemory+0x55
LoadImageNotifyDriver!LoadImageNotifyRoutine+0xdf
nt!PsCallImageNotifyRoutines+0x36
nt!MiMapViewOfImageSection+0x4c1
```

可以看到，线程调用了 KeWaitForSingleObject 在等待某个对象变为信号态，而 KeWaitForSingleObject 的调用是由 NtQueryVirtualMemory 想要获取某个快速互斥体而引起的。我们回溯到 NtQueryVirtualMemory：

```
nt!NtQueryVirtualMemory+0x189:
push    eax
push    esi //要附加的进程（以便操作内存空间）
```

```
call    nt!KeStackAttachProcess (804f8638)
mov     dword ptr [ebp-28h],1
jmp     nt!NtQueryVirtualMemory+0x19d (805adec7)
and     dword ptr [ebp-28h],0
lea     ecx,[esi+0F0h] //esi 为附加上的进程（ExAcquireFastMutex 是 fastcall，参数 FastMutex 储存在 ecx）
call    dword ptr [nt!_imp_ExAcquireFastMutex (804d870c)]
```

查看 EPROCESS 结构：

```
ntdll!_EPROCESS
……
+0x0f0 AddressCreationLock : _FAST_MUTEX
……
```

可以看出，NtQueryVirtualMemory 想要获取 AddressCreationLock。同时，从调用堆栈可知，我们的回调是由 MmMapViewOfSection 一层一层引发的，反汇编 MmMapViewOfSection 可得如下结果：

```
nt!MmMapViewOfSection+0xa8:
push    eax
push    edi //要附加的进程
call    nt!KeStackAttachProcess (804f8638)
mov     dword ptr [ebp-4],1
lea     ecx,[edi+0F0h] // FastMutex
mov     dword ptr [ebp+0Ch],ecx
call    dword ptr [nt!_imp_ExAcquireFastMutex (804d870c)]
```

显然，在 MmMapViewOfSection 中，AddressCreationLock 已经被获取了，而快速互斥体不能被递归的获取（请注意快速互斥体与互斥体的区别），因此会导致死锁，所以，在调用 ZwQueryVirtualMemory 之前，我们需要通过某种方法来避免再次试图获取这个互斥体。

拦截模块加载

无论是内核模块还是 DLL，它们的入口点都会返回一个值来表明初始化成功或是失败，因此我们可以修改入口点的代码，使其直接返回失败，系统便会撤销之前所做的操作，使此次加载模块失败。

其它问题

由于 ZwQueryVirtualMemory 不一定被导出（对于这些未文档化的函数，不同系统导出情况不同，比如对于我所使用的 Windows Server 2008 R2，ZwQueryVirtualMemory 就被导出了，但是对于 Windows XP 而言，该函数就没有被导出），所以我们需要自己定位或者自己实现这个函数。

由于每次查询模块文件名都需要分配缓冲区，而一个进程启动免不了大量的加载模块，如果每次都分配/释放，难免导致内存碎片，降低系统性能。

具体实现

1)ZwQueryVirtualMemory 实现

这个函数很短，我们可以自行实现这个函数，代码如下：

```
nt!ZwQueryVirtualMemory:
mov     eax,0B2h
lea     edx,[esp+4]
pushfd
push    8
call    nt!KiSystemService (8053d651)
ret     18h
```

从 IA32 开发手册可以得知，相同特权级发生中断时，CPU 所执行的操作为（这个中断不涉及错误码）：依次将 Eflags、CS、EIP 入栈，然后转跳到中断处理例程。入栈 EIP，然后转跳即是一个 call，因此，以上代码模拟了一个中断现场，然后通过 KiSystemService 转而调用 NtQueryVirtualMemory。我们也可以模拟一个：

```
#define RING0_CS 8
__declspec(naked)
NTSTATUS
__cdecl
CallSystemService(
IN ULONG ServiceId,
……
)
{
__asm {
mov eax, [esp + 4]; //eax = service id
lea edx, [esp + 8]; //edx = arguments
pushfd;         //eflags
push RING0_CS;  //code segement selector
call KiSystemService;//eip and enter handler
ret;
}
}
```

然而，由于 KiSystemService 未导出，因此我们还需要定位 KiSystemService，显然，对于任何一个 ZwXxx 函数，其中只会出现一次 call，而这个 call 正是对 KiSystemService 的调用。因此，我们可以通过在某个已文档化的 ZwXxx 函数中搜索来定位 KiSystemService：

```
PVOID
LocateKiSystemService(
VOID
```

```
)
{
UCHAR * CodePtr, * OpCode;
ULONG LengthOfCode;
PVOID LocalKiSystemService;
for (CodePtr = reinterpret_cast<UCHAR *>(ZwTerminateProcess); CodePtr != reinterpret_cast<UCHAR *>(ZwTerminateProcess) + SizeOfProc(reinterpret_cast<UCHAR *>(ZwTerminateProcess)); CodePtr += LengthOfCode) {
LengthOfCode = SizeOfCode(CodePtr, &OpCode);
if (!LengthOfCode) {
return NULL;
}
if (*OpCode == 0xe8) {
// call imm32
LocalKiSystemService = reinterpret_cast<PVOID>(reinterpret_cast<CHAR *>(OpCode) + 5 + *reinterpret_cast<ULONG *>(OpCode + 1));
return LocalKiSystemService;
}
}
return NULL;
}
```

同时，为了得到 ZwQueryVirtualMemory 的 ServiceId，我们需要加载 ntdll.dll，并从中找到 ntdll!ZwQueryVirtualMemory。由于 ntdll!ZwQueryVirtualMemory 的第一条指令是“mov eax,ServiceId_ZwQueryVirtualMemory”，因此我们从*reinterpret_cast<ULONG *>(ZwQueryVirtualMemory + 1)即可得到 ServiceId_ZwQueryVirtualMemory。

```
ULONG
GetServiceIdByName(
IN PSTR ProcedureName
)
{
PIMAGE_NT_HEADERS NtHeaders;
PIMAGE_EXPORT_DIRECTORY ImageExportDirectory;
PVOID BinaryBase;
ULONG BinarySize;
NTSTATUS Status;
UNICODE_STRING NtdllPath = RTL_CONSTANT_STRING(NTDLL_PATH);
PSTR * ProcedureNames;
ULONG * ProcedureRvas;
ULONG ProcedureIndex;
CHAR * NtProcedurePtr(0);
ULONG LocalServiceId;
```

```
Status = LoadFile(&NtdllPath, &BinaryBase, &BinarySize);
if (!NT_SUCCESS(Status)) {
DbgPrint("[LoadImageNotifyDriver] GetServiceIdByName: Unable to load file into memory.\n", Status);
return -1; // indicates failure
}
NtHeaders = Add2Ptr(BinaryBase, reinterpret_cast<PIMAGE_DOS_HEADER>(BinaryBase)->e_lfanew, IMAGE_NT_HEADERS);
ImageExportDirectory = Add2Ptr(BinaryBase, RvaToRawOffset(BinaryBase, NtHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress), IMAGE_EXPORT_DIRECTORY);
ProcedureNames = Add2Ptr(BinaryBase, RvaToRawOffset(BinaryBase, ImageExportDirectory->AddressOfNames), PSTR);
ProcedureRvas = Add2Ptr(BinaryBase, RvaToRawOffset(BinaryBase, ImageExportDirectory->AddressOfFunctions), ULONG);
for (ProcedureIndex = 0; ProcedureIndex != ImageExportDirectory->NumberOfFunctions; ++ProcedureIndex) {
if (ProcedureNames[ProcedureIndex]) {
if (!_stricmp(Add2Ptr(BinaryBase, RvaToRawOffset(BinaryBase, reinterpret_cast<ULONG>(ProcedureNames[ProcedureIndex])), CHAR), ProcedureName)) {
NtProcedurePtr = Add2Ptr(BinaryBase, RvaToRawOffset(BinaryBase, ProcedureRvas[ProcedureIndex]), CHAR);
break;
}
}
}
if (!NtProcedurePtr) {
DbgPrint("[LoadImageNotifyDriver] GetServiceIdByName: Unable to find function %s.\n", ProcedureName);
return -1;
}
LocalServiceId = *reinterpret_cast<ULONG *>(NtProcedurePtr + 1);
ExFreePoolWithTag(BinaryBase, DRIVER_POOL_TAG);
return LocalServiceId;
}
```

2）AddressCreationLock 实现

我们可以通过每次调用 ZwQueryVirtualMemory 之前先 Release AddressCreationLock，之后在 LoadImageNotifyRoutine 返回之前再次 Acquire 来执行 ZwQueryVirtualMemory（使用这种做法时请注意，调用 EXE 以及 ntdll 的加载回调时并不会 Acquire AddressCreationLock，因此，在系统调用 EXE 以及 dll 的回调时，请不要直接 Release AddressCreationLock),但这就存在我们Release之后再次被其他线程Acquire到的可能性，

而且由于这个快速互斥体是用来避免多个线程同时操作同一进程的 VAD（虚拟空间描述符）的，我们在操作 VAD 完成之前就将之 Release，而使得其他线程开始操作该进程的 VAD 会导致不可预料的结果。

因此，此处我们通过 Hook ExAcquireFastMutex 以及 ExReleaseFastMutex，使得 AddressCreationLock 这个快速互斥体成为普通的互斥体来避免死锁，考虑到 ZwQueryVirtualMemory 不会操作 VAD，我们这样做是安全的。

另一方面，为了得到 AddressCreationLock 在 EPROCESS 结构中的 Offset，我们通过一个全局变量 OffsetGotten 来指明是否已经获得了该 Offset，然后调用 ZwQueryVirtualMemory，在 NewExAcquireFastMutex 中进行判断，如果尚未获得并且 Caller 为我们自己的线程，我们就将 reinterpret_cast<CHAR *>(FastMutex)-reinterpret_cast<CHAR *>(PsGetCurrentProcess()) 保存在 AddressCreationLockOffset 中。

同时，我们应当注意到，ExAcquireFastMutex 会将 IRQL 提升至 APC_LEVEL，来避免被 APC 中断。同时 ExAcquireFastMutex 又是一个不可能失败的函数（就我们当前的目的而言，我们也不能让 ExAcquireFastMutex 失败），因此我们的 NewExAcquireFastMutex 必须提升 IRQL，之后在 NewExReleaseFastMutex 降低 IRQL。

```
VOID
FASTCALL
NewExAcquireFastMutex(
……
)
{
if (PsGetCurrentThread() == CurrentProcessingThread) {
if (!OffsetGotten) {
AddressCreationLockOffset = reinterpret_cast<CHAR *>(FastMutex) -
reinterpret_cast<CHAR *>(PsGetCurrentProcess());
} else {
if (FastMutex == Add2Ptr(PsGetCurrentProcess(), AddressCreationLockOffset,
FAST_MUTEX)) {
KeRaiseIrql(APC_LEVEL, &FastMutex->OldIrql); // 提升 IRQL
return;
}
}
}
return (ProxyExAcquireFastMutex(FastMutex));
}
VOID
FASTCALL
NewExReleaseFastMutex(
……
)
{
if (PsGetCurrentThread() == CurrentProcessingThread) {
```

```
if (OffsetGotten) {
if (FastMutex == Add2Ptr(PsGetCurrentProcess(), AddressCreationLockOffset,
FAST_MUTEX)) {
KeLowerIrql(FastMutex->OldIrql); // 还原 IRQL
return;
}
}
}
return (ProxyExReleaseFastMutex(FastMutex));
}
```

之后每次调用 ZwQueryVirtualMemory 之前，我们先 Hook ExAcquireFastMutex 以及 ExReleaseFastMutex，在调用完之后再恢复。

```
NTSTATUS
NewZwQueryVirtualMemory(
IN HANDLE ProcessHandle,
IN PVOID BaseAddress,
IN MEMORY_INFORMATION_CLASS MemoryInformationClass,
OUT PVOID MemoryInformation,
IN SIZE_T MemoryInformationLength,
OUT OPTIONAL PSIZE_T ReturnLength
)
{
NTSTATUS Status;
CurrentProcessingThread = PsGetCurrentThread();
HookSystemRoutine(ExAcquireFastMutex, NewExAcquireFastMutex);
HookSystemRoutine(ExReleaseFastMutex, NewExReleaseFastMutex);
Status = CallSystemService(ServiceId_ZwQueryVirtualMemory, ProcessHandle,
BaseAddress, MemoryInformationClass, MemoryInformation, MemoryInformationLength,
ReturnLength);
RestoreSystemRoutine(ExAcquireFastMutex, ProxyExAcquireFastMutex);
RestoreSystemRoutine(ExReleaseFastMutex, ProxyExReleaseFastMutex);
return Status;
}
```

3）减少内存碎片

显然，每次获取模块文件名都调用 ExAllocatePoolWithTag 会造成大量的内存分配/释放，为了避免此种情况的发生，我们可以使用 LookasideList。在内部，每次我们释放内存的时候不会引发真正的内存释放，而是将内存再存入 LookasideList，尽管这样会更多的消耗内存，但这样避免了多次内存分配与释放所带来的内存碎片以及性能开销。

```
// use look-aside-list for better system performance
ExInitializeNPagedLookasideList(&ImageFileNamePoolLookasideList, NULL, NULL, 0,
```

```
PAGE_SIZE, DRIVER_POOL_TAG, 0);
```

但是由于文件名长度我们无法确定，因此，尽管我们将分配的内存大小设置为 PAGE_SIZE，但仍然存在缓冲区不够大的可能性。对于这种情况，我们使用 ExAllocatePoolWithTag 来为其分配内存。

```
ImageFileName =
reinterpret_cast<POBJECT_NAME_INFORMATION>(ExAllocateFromNPagedLookasideList(&I
mageFileNamePoolLookasideList)); //首先从 LookasideList 中获取一个缓冲区
if (!ImageFileName) {
DbgPrint("[LoadImageNotifyDriver] LoadImageNotifyRoutine: Unable to allocate pool
for ImageFileName");
return;
}
Status = NewZwQueryVirtualMemory(ZwCurrentProcess(), ImageInfo->ImageBase,
MemoryMappedFilenameInformation, ImageFileName, PAGE_SIZE, &BufferSize);
if (Status == STATUS_INFO_LENGTH_MISMATCH) {
//PAGE_SIZE 仍然不够大
NewZwQueryVirtualMemory(ZwCurrentProcess(),ImageInfo->ImageBase,MemoryMappedFil
enameInformation, NULL, 0, &BufferSize); //获取所需的大小
ExFreeToNPagedLookasideList(&ImageFileNamePoolLookasideList, ImageFileName);
ImageFileName =
reinterpret_cast<POBJECT_NAME_INFORMATION>(ExAllocatePoolWithTag(NonPagedPool,
BufferSize, DRIVER_POOL_TAG)); //使用 ExAllocatePoolWithTag 分配缓冲区
if (!ImageFileName) {
DbgPrint("[LoadImageNotifyDriver] LoadImageNotifyRoutine: Unable to allocate pool
for ImageFileName");
KeRaiseIrql(OriginalIrql, &OriginalIrql);
return;
}
ImageFileNameFromPool = TRUE;
Status = NewZwQueryVirtualMemory(ZwCurrentProcess(), ImageInfo->ImageBase,
MemoryMappedFilenameInformation, ImageFileName, BufferSize, &BufferSize);
}
```

另一方面，由于我们每次调用 ZwQueryVirtualMemory 之前都会对 ExAcquireFastMutex 以及 ExReleaseFastMutex 进行 HOOK，调用完毕之后又会恢复 HOOK，因此如果我们在 HOOK 的时候分配代理函数，在恢复的时候释放代理函数，必然导致大量的内存分配释放。因此，我们在驱动加载时分配代理函数，在驱动卸载时释放代理函数。在 HOOK 时只修改要 HOOK 的函数头为转跳代码，在恢复的时候也只恢复函数头的代码。这个很容易实现，代码就不再贴出了。

4）NT 风格文件名转换

网上对于 NT 风格文件名转换有很多成熟的代码，比如使用 QueryDosDevice 或是获取所

有 DOS 风格的分区根目录所对应的 NT 路径，然后反查等等。本文将介绍另一种方法。

对于 ZwOpenFile、ZwCreateFile、IoCreateFile 这些函数而言，它们能够处理 NT 风格路径，因此，我们将 NT 风格路径传给 ZwOpenFile 得到 FileHandle，然后将 FileHandle 传给 ObReferenceObjectByHandle 得到 FileObject，之后我们便可以使用以前的方法来从 FileObject 得到 DOS 风格的路径了。

关于从 FileObject 得到 DOS 风格路径，本文也不再使用惯用的 IoVolumeDeviceToDosName 方法，而使用 Windows XP 新引入的函数 IoQueryFileDosDeviceName，该函数接收 FileObject 作为参数，返回该 FileObject 所对应的 DOS 风格的路径。

```
InitializeObjectAttributes(&ObjectAttributes, &ImageFileName->Name,
OBJ_CASE_INSENSITIVE | ((ExGetPreviousMode() == KernelMode) ? OBJ_KERNEL_HANDLE :
0), NULL, NULL);
Status = ZwOpenFile(&FileHandle, FILE_READ_ATTRIBUTES, &ObjectAttributes,
&IoStatusBlock, FILE_SHARE_READ, 0);
if (ImageFileNameFromPool) {
ExFreePoolWithTag(ImageFileName, DRIVER_POOL_TAG);
} else {
ExFreeToNPagedLookasideList(&ImageFileNamePoolLookasideList, ImageFileName);
}
if (!NT_SUCCESS(Status)) {
DbgPrint("[LoadImageNotifyDriver] LoadImageNotifyRoutine: Unable to open
file %wZ.\n", &ImageFileName->Name);
KeRaiseIrql(OriginalIrql, &OriginalIrql);
return;
}
Status = ObReferenceObjectByHandle(FileHandle, FILE_READ_ATTRIBUTES,
*IoFileObjectType, KernelMode, reinterpret_cast<PVOID *>(&FileObject), NULL);
ZwClose(FileHandle);
if (!NT_SUCCESS(Status)) {
DbgPrint("[LoadImageNotifyDriver] LoadImageNotifyRoutine: Unable to get Dos-style
file name for %wZ.\n", &ImageFileName->Name);
KeRaiseIrql(OriginalIrql, &OriginalIrql);
return;
}
Status = IoQueryFileDosDeviceName(FileObject, &FileName);
ObDereferenceObject(FileObject);
KeRaiseIrql(OriginalIrql, &OriginalIrql);
```

事实上，若只是得到 ZwQueryVirtualMemory 返回的 NT 风格路径对应的 DOS 风格路径的话，我们还有另外一种做法。

在 NtQueryVirtualMemory 内部，该函数调用了（并且也仅调用了一次）ObQueryNameString（请自行分析 NtQueryVirtualMemory 流程），因此，通过 Hook ObQueryNameString，我们即可得到所查询的地址处映射的文件的 FileObject，之后再使用

IoQueryFileDosDeviceName 即可得到 DOS 风格路径。

```
PFILE_OBJECT ImageFileObject;
//使用全局变量保存 FileObject
NTSTATUS
NTAPI
NewObQueryNameString(
……
)
{
if (PsGetCurrentThread() == CurrentProcessingThread) {
ImageFileObject = reinterpret_cast<FILE_OBJECT *>(Object); //记录 FileObject
return STATUS_SUCCESS;
}
return ProxyObQueryNameString(Object, ObjectNameInfo, Length, ReturnLength);
}
//回调函数中
POBJECT_NAME_INFORMATION ImageFileName;
SIZE_T BufferSize;
KIRQL OriginalIrql(KeGetCurrentIrql());
KeLowerIrql(PASSIVE_LEVEL); //降低 IRQL（回调调用时处在 APC_LEVEL）
CurrentProcessingThread = PsGetCurrentThread();
HookSystemRoutine(ObQueryNameString, NewObQueryNameString);
Status = NewZwQueryVirtualMemory(ZwCurrentProcess(), ImageInfo->ImageBase,
MemoryMappedFilenameInformation, NULL, 0, &BufferSize);
RestoreSystemRoutine(ObQueryNameString, ProxyObQueryNameString);
if (NT_SUCCESS(Status)) {
Status = IoQueryFileDosDeviceName(ImageFileObject, &FileName);
}
KeRaiseIrql(OriginalIrql, &OriginalIrql);
```

5）模块拦截

通过修改正在加载的模块的 EntryPoint 的代码（注意在调用我们的回调之时，系统已经将模块映射入内存，但是尚未调用模块的 EP 函数），我们可以让模块返回一个失败的信息，从而使得系统撤销加载该模块的操作。

但是需要注意，在 PE 格式说明文档中明确指出，DLL 可以没有入口点函数（NtHeaders->OptionaHeader.AddressOfEntryPoint=0），对于这种情况，我们可以将该 DLL 的入口点函数设置为该 DLL 的代码基部（NtHeaders->OptionalHeader.BaseOfCode），然后再进行修改。

若 BaseOfCode 为 0，我们可以修改 BaseOfData 的页面属性，将入口点函数放置于 BaseOfData 处：

```
VOID
PreventLibraryLoading(
```

```
PVOID Base,
BOOLEAN KernelModeLibrary
)
{
PIMAGE_NT_HEADERS                                    NtHeaders(Add2Ptr(Base,
reinterpret_cast<PIMAGE_DOS_HEADER>(Base)->e_lfanew, IMAGE_NT_HEADERS));
PVOID  EntryPoint(Add2Ptr(Base,  NtHeaders->OptionalHeader.AddressOfEntryPoint,
VOID));
CHAR NewEntryPointCode[] = "\xB8\x22\x00\x00\xC0\xC2\x08\x00";
//9 bytes long ( mov eax, imm32 ; ret imm16 )
if (!KernelModeLibrary) {
NTSTATUS Status;
PVOID BaseAddress;
ULONG OldProtect;
SIZE_T RegionSize(sizeof(NewEntryPointCode));
*reinterpret_cast<USHORT *>(NewEntryPointCode + 1) = 0x0; // false
*reinterpret_cast<USHORT *>(NewEntryPointCode + 6) = 0x000C; // the number of
parameter of user mode dll's entrypoint is three.
if (EntryPoint == Base) { //没有 EP
NtHeaders->OptionalHeader.AddressOfEntryPoint                                =
NtHeaders->OptionalHeader.BaseOfCode;
EntryPoint = Add2Ptr(Base, NtHeaders->OptionalHeader.AddressOfEntryPoint, VOID);
}
if (EntryPoint == Base) { //BaseOfCode 也为 0
NtHeaders->OptionalHeader.AddressOfEntryPoint                                =
NtHeaders->OptionalHeader.BaseOfData;
EntryPoint = Add2Ptr(Base, NtHeaders->OptionalHeader.AddressOfEntryPoint, VOID);
BaseAddress = EntryPoint;
NewZwProtectVirtualMemory(ZwCurrentProcess(),     &BaseAddress,     &RegionSize,
PAGE_EXECUTE_READ, &OldProtect);
```

```
}
BaseAddress = EntryPoint;
Status = NewZwProtectVirtualMemory(ZwCurrentProcess(), &BaseAddress, &RegionSize,
PAGE_READWRITE, &OldProtect);
if (!NT_SUCCESS(Status)) {
return;
}
SafeCopyMemory(EntryPoint, NewEntryPointCode, sizeof(NewEntryPointCode));
NewZwProtectVirtualMemory(ZwCurrentProcess(), &BaseAddress, &RegionSize,
OldProtect, &OldProtect);
} else {
SafeCopyMemory(EntryPoint, NewEntryPointCode, sizeof(NewEntryPointCode));
}
return;
}
```

此处不考虑 DLL 既不包含代码也不包含数据的情况。同样，ZwProtectVirtualMemory 前后也需要 HOOK ExAcquireFastMutex 以及 ExReleaseFastMutex，因为 ZwProtectVirtualMemory 也需要获取 AddressCreationLock，同时，考虑到 ZwProtectVirtualMemory 不会增减 VAD 树，这种做法是安全的。

结束语

本文通过使用加载模块回调，实现了拦截模块的加载。但是应当意识到的是，从前文我们可以看出,这样的做法是建立在调用方无法在系统调用入口点函数之前进行操作的前提之上的。因此，对于用户态 DLL 而言，这种做法不能确保一定能够拦截成功，这是这种做法的缺陷之所在。但是，对于内核模块而言，这种做法能够拦截多种加载驱动的方法（如 ZwLoadDriver、ZwSetSystemInformation、直接调用 MmLoadSystemImage 等），要比传统的做法更加可靠。