在笔者上一篇文章《内核实现SSDT挂钩与摘钩》中介绍了如何对SSDT函数进行Hook挂钩与摘钩的，本章将继续实现一个新功能，如何检测SSDT函数是否挂钩，要实现检测挂钩状态有两种方式，第一种方式则是类似于《摘除InlineHook内核钩子》文章中所演示的通过读取函数的前16个字节与原始字节做对比来判断挂钩状态，另一种方式则是通过对比函数的当前地址与起源地址进行判断，为了提高检测准确性本章将采用两种方式混合检测。

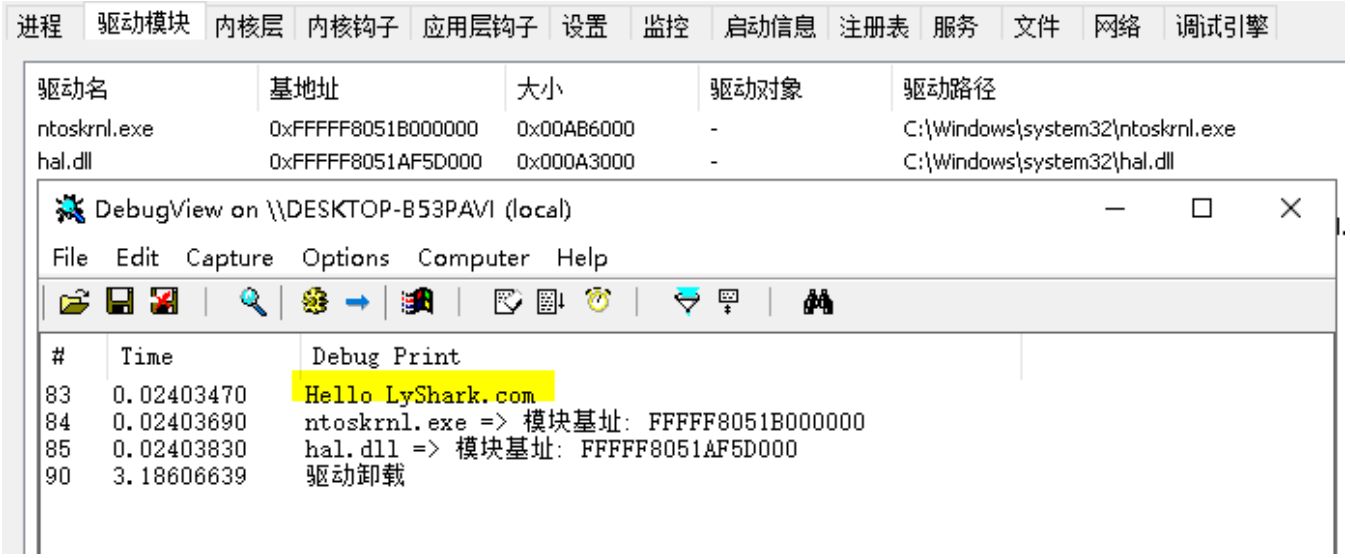
具体原理，通过解析内核文件PE结构找到导出表，依次计算出每一个内核函数的RVA相对偏移，通过与内核模块基址相加此相对偏移得到函数的原始地址，然后再动态获取函数当前地址，两者作比较即可得知指定内核函数是否被挂钩。

在实现这个功能之前我们需要解决两个问题，第一个问题是如何得到特定内核模块的内存模块基址此处我们需要封装一个GetOsBaseAddress()用户只需要传入指定的内核模块即可得到该模块基址，如此简单的代码没有任何解释的必要；

```
#include <ntifs.h>
#include <ntimage.h>
#include <ntstrsafe.h>

typedef struct _LDR_DATA_TABLE_ENTRY
{
    LIST_ENTRY InLoadOrderLinks;
    LIST_ENTRY InMemoryOrderLinks;
    LIST_ENTRY InInitializationOrderLinks;
    PVOID DllBase;
    PVOID EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    ULONG Flags;
    USHORT LoadCount;
    USHORT TlsIndex;
    LIST_ENTRY HashLinks;
    ULONG TimeDateStamp;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;

// 得到内核模块基址
ULONGLONG GetOsBaseAddress(PDRIVER_OBJECT pDriverObject, WCHAR *wzData)
{
    UNICODE_STRING osName = { 0 };
    // WCHAR wzData[0x100] = L"ntoskrnl.exe";
    RtlInitUnicodeString(&osName, wzData);

    LDR_DATA_TABLE_ENTRY *pDataTableEntry, *pTempDataTableEntry;
    //双循环链表定义
    PLIST_ENTRY    pList;
    //指向驱动对象的DriverSection
    pDataTableEntry = (LDR_DATA_TABLE_ENTRY*)pDriverObject->DriverSection;
    //判断是否为空
    if (!pDataTableEntry)
    {
        return 0;
    }
```

```
    //得到链表地址
    pList = pDataTableEntry->InLoadOrderLinks.Flink;

    // 判断是否等于头部
    while (pList != &pDataTableEntry->InLoadOrderLinks)
    {
        pTempDataTableEntry = (LDR_DATA_TABLE_ENTRY *)pList;
        if (RtlEqualUnicodeString(&pTempDataTableEntry->BaseDllName, &osName, TRUE))
        {
            return (ULONGLONG)pTempDataTableEntry->DllBase;
        }
        pList = pList->Flink;
    }
    return 0;
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("驱动卸载 \n");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("Hello LyShark.com \n");

    ULONGLONG kernel_base = GetOsBaseAddress(Driver, L"ntoskrnl.exe");
    DbgPrint("ntoskrnl.exe => 模块基址: %p \n", kernel_base);

    ULONGLONG hal_base = GetOsBaseAddress(Driver, L"hal.dll");
    DbgPrint("hal.dll => 模块基址: %p \n", hal_base);

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

如上直接编译并运行，即可输出`ntoskrnl.exe`以及`hal.dll`两个内核模块的基址；

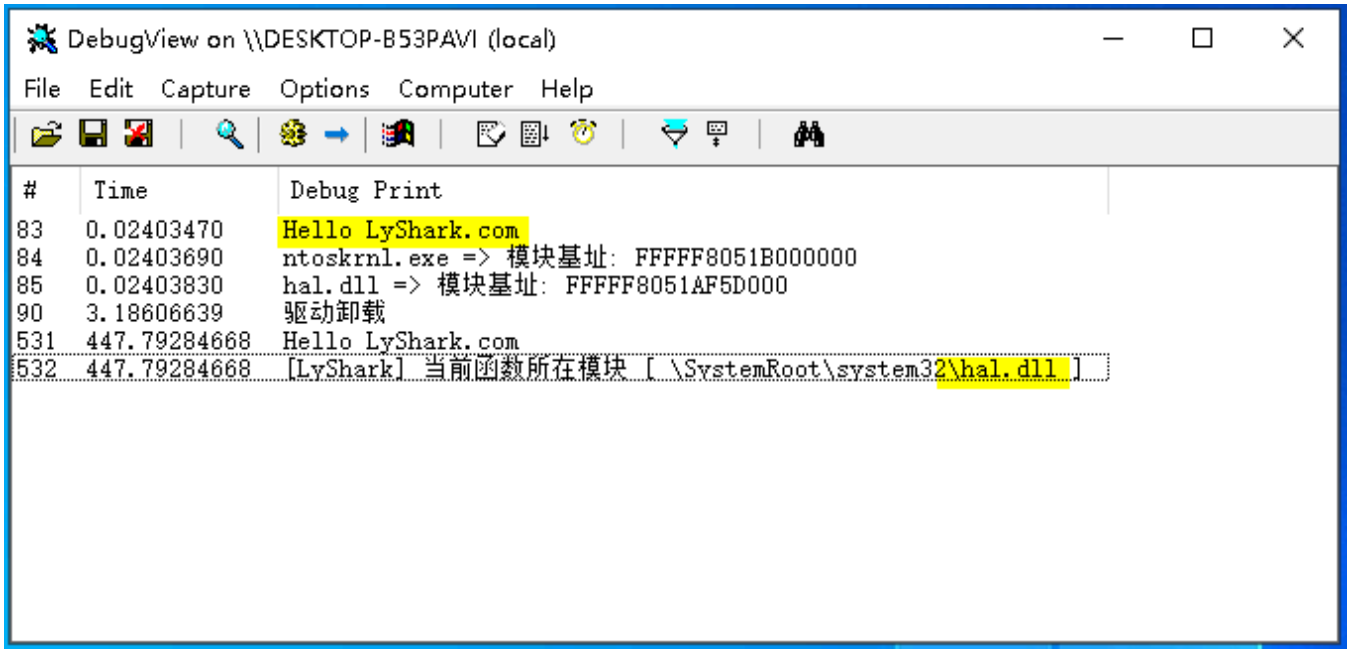


其次我们还需要实现另一个功能，此时想像一下当我告诉你一个内存地址，我想要查该内存地址属于哪个模块该如何实现，其实很简单只需要拿到这个地址依次去判断其是否大于等于该模块的基地址，并小于等于该模块的结束地址，那么我们就认为该地址落在了此模块上，在这个思路下`LyShark`实现了以下代码片段。

```
#include <ntifs.h>
#include <ntimage.h>
#include <ntstrsafe.h>

typedef struct _LDR_DATA_TABLE_ENTRY
{
    LIST_ENTRY InLoadOrderLinks;
    LIST_ENTRY InMemoryOrderLinks;
    LIST_ENTRY InInitializationOrderLinks;
    PVOID DllBase;
    PVOID EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    ULONG Flags;
    USHORT LoadCount;
    USHORT TlsIndex;
    LIST_ENTRY HashLinks;
    ULONG TimeDateStamp;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;

// 扫描指定地址是否在某个模块内
VOID ScanKernelModuleBase(PDRIVER_OBJECT pDriverObject, ULONGLONG address)
{
    LDR_DATA_TABLE_ENTRY *pDataTableEntry, *pTempDataTableEntry;
    PLIST_ENTRY pList;
    pDataTableEntry = (LDR_DATA_TABLE_ENTRY*)pDriverObject->DriverSection;
    if (!pDataTableEntry)
    {
        return;
    }

    // 得到链表地址
    pList = pDataTableEntry->InLoadOrderLinks.Flink;

    // 判断是否等于头部
    while (pList != &pDataTableEntry->InLoadOrderLinks)
    {
        pTempDataTableEntry = (LDR_DATA_TABLE_ENTRY *)pList;

        ULONGLONG start_address = (ULONGLONG)pTempDataTableEntry->DllBase;
        ULONGLONG end_address = start_address + (ULONG)pTempDataTableEntry->SizeOfImage;

        // 判断区间
        // DbgPrint("起始地址 [ %p ] 结束地址 [ %p ] \n",start_address,end_address);
        if (address >= start_address && address <= end_address)
        {
```

```
            DbgPrint("[LyShark] 当前函数所在模块 [ %ws ] \n", (CHAR *)pTempDataTableEntry->FullDllName.Buffer);
        }
        pList = pList->Flink;
    }
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("驱动卸载 \n");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("Hello LyShark.com \n");

    ScanKernelModuleBase(Driver, 0xFFFFF8051AF5D030);

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

我们以 `0xFFFFF8051AF5D030` 地址为例对其进行判断可看到输出了如下结果，此地址被落在了 `hal.dll` 模块上；



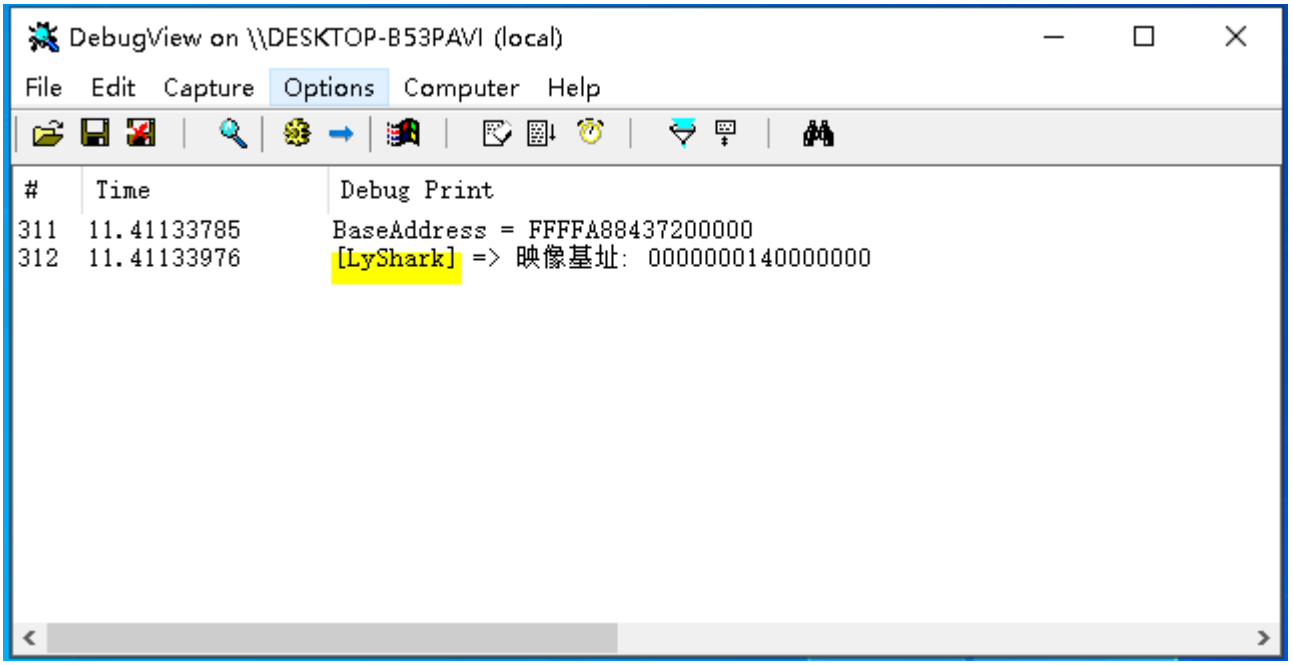
为了能读入磁盘PE文件到内存此时我们还需要封装一个 `LoadKernelFile()` 函数，该函数的作用是读入一个内核文件到内存空间中，此处如果您使用前一篇 `《内核解析PE结构导出表》` 文章中的内存映射函数来读写则会蓝屏，原因很简单 `KernelMapFile()` 是映射而映射一定无法一次性完整装载其次此方法本质上还在占用原文件，而 `LoadKernelFile()` 则是读取磁盘文件并将其完整拷贝一份，这是两者的本质区别，如下代码则是实现完整拷贝的实现；

```
#include <ntifs.h>
#include <ntimage.h>
#include <ntstrsafe.h>
```

```
// 将内核文件装载入内存(磁盘)
PVOID LoadKernelFile(WCHAR *wzFileName)
{
    NTSTATUS Status;
    HANDLE FileHandle;
    IO_STATUS_BLOCK ioStatus;
    FILE_STANDARD_INFORMATION FileInformation;

    // 设置路径
    UNICODE_STRING uniFileName;
    RtlInitUnicodeString(&uniFileName, wzFileName);

    // 初始化打开文件的属性
    OBJECT_ATTRIBUTES objectAttributes;
    InitializeObjectAttributes(&objectAttributes, &uniFileName, OBJ_KERNEL_HANDLE |
OBJ_CASE_INSENSITIVE, NULL, NULL);

    // 打开文件
    Status = IoCreateFile(&FileHandle, FILE_READ_ATTRIBUTES | SYNCHRONIZE,
&objectAttributes, &ioStatus, 0, FILE_READ_ATTRIBUTES, FILE_SHARE_READ, FILE_OPEN,
FILE_SYNCHRONOUS_IO_NONALERT, NULL, 0, CreateFileTypeNone, NULL, IO_NO_PARAMETER_CHECKING);
    if (!NT_SUCCESS(Status))
    {
        return 0;
    }

    // 获取文件信息
    Status = ZwQueryInformationFile(FileHandle, &ioStatus, &FileInformation,
sizeof(FILE_STANDARD_INFORMATION), FileStandardInformation);
    if (!NT_SUCCESS(Status))
    {
        ZwClose(FileHandle);
        return 0;
    }

    // 判断文件大小是否过大
    if (FileInformation.EndOfFile.HighPart != 0)
    {
        ZwClose(FileHandle);
        return 0;
    }

    // 取文件大小
    ULONG64 uFileSize = FileInformation.EndOfFile.LowPart;

    // 分配内存
    PVOID pBuffer = ExAllocatePoolWithTag(NonPagedPool, uFileSize + 0x100,
(ULONG)"LyShark");
    if (pBuffer == NULL)
    {
        ZwClose(FileHandle);
        return 0;
    }
```

```
    // 从头开始读取文件
    LARGE_INTEGER byteOffset;
    byteOffset.LowPart = 0;
    byteOffset.HighPart = 0;
    Status = ZwReadFile(FileHandle, NULL, NULL, NULL, &ioStatus, pBuffer, uFileSize,
&byteOffset, NULL);
    if (!NT_SUCCESS(Status))
    {
        ZwClose(FileHandle);
        return 0;
    }

    // ExFreePoolWithTag(pBuffer, (ULONG)"LyShark");
    ZwClose(FileHandle);
    return pBuffer;
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("驱动卸载 \n");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    // 加载内核模块
    PVOID BaseAddress = LoadKernelFile(L"\\SystemRoot\\system32\\ntoskrnl.exe");
    DbgPrint("BaseAddress = %p\n", BaseAddress);

    // 解析PE头
    PIMAGE_DOS_HEADER pDosHeader;
    PIMAGE_NT_HEADERS pNtHeaders;

    // DLL内存数据转成DOS头结构
    pDosHeader = (PIMAGE_DOS_HEADER)BaseAddress;

    // 取出PE头结构
    pNtHeaders = (PIMAGE_NT_HEADERS)((ULONGLONG)BaseAddress + pDosHeader->e_lfanew);

    DbgPrint("[LyShark] => 映像基址: %p \n", pNtHeaders->OptionalHeader.ImageBase);

    // 结束后释放内存
    ExFreePoolWithTag(BaseAddress, (ULONG)"LyShark");

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

运行如上这段程序，则会将`ntoskrnl.exe`文件载入到内存，并读取出其中的`OptionalHeader.ImageBase`映像基址，如下图所示；



有了上述方法，最后一步就是组合并实现判断即可，如下代码通过对导出表的解析，并过滤出所有的`Nt`开头的系列函数，然后依次对比起源地址与原地址是否一致，得出是否被挂钩，完整代码如下所示；

```
ULONGLONG ntoskrnl_base = 0;

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("Hello LyShark.com \n");

    // 加载内核模块
    PVOID BaseAddress = LoadKernelFile(L"\\SystemRoot\\system32\\ntoskrnl.exe");
    DbgPrint("BaseAddress = %p\n", BaseAddress);

    // 获取内核模块地址
    ntoskrnl_base = GetOsBaseAddress(Driver, L"ntoskrnl.exe");

    // 取出导出表
    PIMAGE_DOS_HEADER pDosHeader;
    PIMAGE_NT_HEADERS pNtHeaders;
    PIMAGE_SECTION_HEADER pSectionHeader;
    ULONGLONG FileOffset;
    PIMAGE_EXPORT_DIRECTORY pExportDirectory;

    // DLL内存数据转成DOS头结构
    pDosHeader = (PIMAGE_DOS_HEADER)BaseAddress;
    // 取出PE头结构
    pNtHeaders = (PIMAGE_NT_HEADERS)((ULONGLONG)BaseAddress + pDosHeader->e_lfanew);
    // 判断PE头导出表表是否为空
    if (pNtHeaders-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress == 0)
    {
        return 0;
    }
```

```
    // 取出导出表偏移
    FileOffset = pNtHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;

    // 取出节头结构
    pSectionHeader = (PIMAGE_SECTION_HEADER)((ULONGLONG)pNtHeaders + sizeof(IMAGE_NT_HEADERS));
    PIMAGE_SECTION_HEADER pOldSectionHeader = pSectionHeader;

    // 遍历节结构进行地址运算
    for (UINT16 Index = 0; Index < pNtHeaders->FileHeader.NumberOfSections; Index++, pSectionHeader++)
    {
        if (pSectionHeader->VirtualAddress <= FileOffset && FileOffset <= pSectionHeader->VirtualAddress + pSectionHeader->SizeOfRawData)
        {
            FileOffset = FileOffset - pSectionHeader->VirtualAddress + pSectionHeader->PointerToRawData;
        }
    }

    // 导出表地址
    pExportDirectory = (PIMAGE_EXPORT_DIRECTORY)((ULONGLONG)BaseAddress + FileOffset);

    // 取出导出表函数地址
    PULONG AddressOfFunctions;
    FileOffset = pExportDirectory->AddressOfFunctions;

    // 遍历节结构进行地址运算
    pSectionHeader = pOldSectionHeader;
    for (UINT16 Index = 0; Index < pNtHeaders->FileHeader.NumberOfSections; Index++, pSectionHeader++)
    {
        if (pSectionHeader->VirtualAddress <= FileOffset && FileOffset <= pSectionHeader->VirtualAddress + pSectionHeader->SizeOfRawData)
        {
            FileOffset = FileOffset - pSectionHeader->VirtualAddress + pSectionHeader->PointerToRawData;
        }
    }

    // 这里注意一下foa和rva
    AddressOfFunctions = (PULONG)((ULONGLONG)BaseAddress + FileOffset);

    // 取出导出表函数名字
    PUSHORT AddressOfNameOrdinals;
    FileOffset = pExportDirectory->AddressOfNameOrdinals;

    // 遍历节结构进行地址运算
    pSectionHeader = pOldSectionHeader;
    for (UINT16 Index = 0; Index < pNtHeaders->FileHeader.NumberOfSections; Index++, pSectionHeader++)
    {
```

```
        if (pSectionHeader->VirtualAddress <= FileOffset && FileOffset <= pSectionHeader->VirtualAddress + pSectionHeader->SizeOfRawData)
        {
            FileOffset = FileOffset - pSectionHeader->VirtualAddress + pSectionHeader->PointerToRawData;
        }
    }

    // 注意一下foa和rva
    AddressOfNameOrdinals = (PUSHORT)((ULONGLONG)BaseAddress + FileOffset);

    // 取出导出表函数序号
    PULONG AddressOfNames;
    FileOffset = pExportDirectory->AddressOfNames;

    // 遍历节结构进行地址运算
    pSectionHeader = pOldSectionHeader;
    for (UINT16 Index = 0; Index < pNtHeaders->FileHeader.NumberOfSections; Index++, pSectionHeader++)
    {
        if (pSectionHeader->VirtualAddress <= FileOffset && FileOffset <= pSectionHeader->VirtualAddress + pSectionHeader->SizeOfRawData)
        {
            FileOffset = FileOffset - pSectionHeader->VirtualAddress + pSectionHeader->PointerToRawData;
        }
    }

    // 注意一下foa和rva
    AddressOfNames = (PULONG)((ULONGLONG)BaseAddress + FileOffset);

    // 分析导出表
    ULONG uOffset;
    LPSTR FunName;
    ULONG uAddressOfNames;
    ULONG TargetOff = 0;

    for (ULONG uIndex = 0; uIndex < pExportDirectory->NumberOfNames; uIndex++, AddressOfNames++, AddressOfNameOrdinals++)
    {
        uAddressOfNames = *AddressOfNames;
        pSectionHeader = pOldSectionHeader;
        for (UINT16 Index = 0; Index < pNtHeaders->FileHeader.NumberOfSections; Index++, pSectionHeader++)
        {
            if (pSectionHeader->VirtualAddress <= uAddressOfNames && uAddressOfNames <= pSectionHeader->VirtualAddress + pSectionHeader->SizeOfRawData)
            {
                uOffset = uAddressOfNames - pSectionHeader->VirtualAddress + pSectionHeader->PointerToRawData;
            }
        }
        FunName = (LPSTR)((ULONGLONG)BaseAddress + uOffset);
```

```
        if (FunName[0] == 'N' && FunName[1] == 't')
        {
            // 得到相对RVA
            TargetOff = (ULONG)AddressOfFunctions[*AddressOfNameOrdinals];

            // LPSTR -> UNCODE
            // 先转成ANSI 然后在转成 UNCODE
            ANSI_STRING ansi = { 0 };
            UNICODE_STRING uncode = { 0 };

            RtlInitAnsiString(&ansi, FunName);
            RtlAnsiStringToUnicodeString(&uncode, &ansi, TRUE);

            // 得到当前地址
            PULONGLONG local_address = MmGetSystemRoutineAddress(&uncode);

            /*
            // 读入内核函数前6个字节
            unsigned char local_opcode[6] = { 0 };
            unsigned char this_opcode[6] = { 0 };

            RtlCopyMemory(local_opcode, (void *)local_address, 6);
            RtlCopyMemory(this_opcode, (void *)(ntoskrnl_base + TargetOff), 6);

            // 当前机器码
            for (int x = 0; x < 6; x++)
            {
            DbgPrint("当前 [ %d ] 机器码 [ %x ] ", x, local_opcode[x]);
            }

            // 起源机器码
            for (int y = 0; y < 6; y++)
            {
            DbgPrint("起源 [ %d ] 机器码 [ %x ] ", y, this_opcode[y]);
            }

            */

            // 检测是否被挂钩 [不相等则说明被挂钩了]
            if (local_address != (ntoskrnl_base + TargetOff))
            {
                DbgPrint("索引 [ %d ] RVA [ %p ] \n --> 起源地址 [ %p ] | 当前地址 [ %p ] | 函数
名 [ %s ] \n\n",
                    *AddressOfNameOrdinals, TargetOff, ntoskrnl_base + TargetOff,
local_address, FunName);
            }

            // 检查当前地址所在模块
            // ScanKernelModuleBase(Driver, (PULONGLONG)local_address);
        }
    }
```

```
    // 结束后释放内存
    ExFreePoolWithTag(BaseAddress, (ULONG)"LyShark");

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

使用ARK工具手动改写几个Nt开头的函数，并运行这段代码，观察是否可以输出被挂钩的函数详情；