在笔者上一篇文章`《内核RIP劫持实现DLL注入》`介绍了通过劫持RIP指针控制程序执行流实现插入DLL的目的，本章将继续探索全新的注入方式，通过`NtCreateThreadEx`这个内核函数实现注入DLL的目的，需要注意的是该函数在微软系统中未被导出使用时需要首先得到该函数的入口地址，`NtCreateThreadEx`函数最终会调用`ZwCreateThread`，本章在寻找函数的方式上有所不同，前一章通过内存定位的方法得到所需地址，本章则是通过解析导出表实现。

内核导出表远程线程是一种实现DLL注入的常见技术之一。通过使用该技术，注入代码可以利用目标进程的导出表中已有的函数来加载DLL，并在远程线程中执行DLL代码，从而实现DLL注入。

具体而言，内核导出表远程线程实现DLL注入的过程包括以下步骤：

- 打开目标进程，获取其进程句柄。
- 在目标进程的内存空间中分配一段可执行代码的内存空间，将注入代码写入其中。
- 通过GetProcAddress函数获取目标进程中已有的一个导出函数的地址，如LoadLibraryA等函数。
- 在目标进程中创建一个远程线程，将获取到的导出函数地址作为线程的入口点，并将DLL路径等参数传递给导出函数。
- 远程线程在目标进程中运行，并调用导出函数。导出函数会将DLL加载到目标进程的内存中，并返回DLL的句柄。
- 远程线程继续执行注入代码，利用DLL的句柄和GetProcAddress函数获取目标函数的地址，从而实现DLL注入。

需要注意的是，内核导出表远程线程作为一种内核级别的注入技术，可能会被安全软件或操作系统检测到，并对其进行防御。因此，在使用这种技术进行DLL注入时，需要谨慎使用，并采取必要的安全措施，以防止被检测和防御。

在内核模式中实现这一过程的具体方法可分为如下步骤；

- 1.通过`GetKeServiceDescriptorTable64`获取到SSDT表基址
- 2.通过`KeStackAttachProcess`附加到远程进程内
- 3.通过`GetUserModuleAddress`获取到`Ntdll.dll`模块内存基址
- 4.通过`GetModuleExportAddress`获取到`LdrLoadDll`函数的内存地址
- 5.调用`GetNative32Code`生成拉起特定DLL的`ShellCode`片段
- 6.通过`NtCreateThreadEx`将`ShellCode`执行起来，并自动加载DLL
- 7.通过`KeUnstackDetachProcess`取消附加远程进程，并做好最后的清理工作

首先需要定义一个标准头文件，并将其命名为`lyshark.h`其定义部分如下所示，此部分内容包含了微软官方结构定义，以及一些通用函数的规整，已做较为详细的分析和备注，由于前面课程中都有介绍，此处不再介绍具体原理，如果需要了解结构体内的含义，请去自行查阅微软官方文档。

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

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

#define THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER 0x00000004
```

```
// ------------------------------------------------------------------------------
-----
// 声明未导出函数
// ------------------------------------------------------------------------------
-----

NTKERNELAPI PPEB NTAPI PsGetProcessPeb(IN PEPROCESS Process);
NTKERNELAPI UCHAR* PsGetProcessImageFileName(IN PEPROCESS Process);
NTKERNELAPI PVOID NTAPI PsGetProcessWow64Process(IN PEPROCESS Process);
NTKERNELAPI HANDLE PsGetProcessInheritedFromUniqueProcessId(IN PEPROCESS
Process);

NTSYSAPI NTSTATUS NTAPI ZwQueryInformationThread(
    IN HANDLE ThreadHandle,
    IN THREADINFOCLASS ThreadInformationClass,
    OUT PVOID ThreadInformation,
    IN ULONG ThreadInformationLength,
    OUT PULONG ReturnLength OPTIONAL
    );

typedef NTSTATUS(NTAPI* LPFN_NTCREATETHREADEX)(
    OUT PHANDLE ThreadHandle,
    IN ACCESS_MASK DesiredAccess,
    IN PVOID ObjectAttributes,
    IN HANDLE ProcessHandle,
    IN PVOID StartAddress,
    IN PVOID Parameter,
    IN ULONG Flags,
    IN SIZE_T StackZeroBits,
    IN SIZE_T SizeOfStackCommit,
    IN SIZE_T SizeOfStackReserve,
    OUT PVOID ByteBuffer
    );

// ------------------------------------------------------------------------------
-----
// 结构体声明
// ------------------------------------------------------------------------------
-----

// SSDT表结构
typedef struct _SYSTEM_SERVICE_TABLE
{
    PVOID       ServiceTableBase;
    PVOID       ServiceCounterTableBase;
    ULONGLONG   NumberOfServices;
    PVOID       ParamTableBase;
} SYSTEM_SERVICE_TABLE, *PSYSTEM_SERVICE_TABLE;

PSYSTEM_SERVICE_TABLE KeServiceDescriptorTable;

typedef struct _PEB_LDR_DATA32
{
    ULONG Length;
    UCHAR Initialized;
    ULONG SsHandle;
    LIST_ENTRY32 InLoadOrderModuleList;
    LIST_ENTRY32 InMemoryOrderModuleList;
    LIST_ENTRY32 InInitializationOrderModuleList;
} PEB_LDR_DATA32, *PPEB_LDR_DATA32;

typedef struct _PEB_LDR_DATA
{
    ULONG Length;
    UCHAR Initialized;
    PVOID SsHandle;
    LIST_ENTRY InLoadOrderModuleList;
    LIST_ENTRY InMemoryOrderModuleList;
    LIST_ENTRY InInitializationOrderModuleList;
} PEB_LDR_DATA, *PPEB_LDR_DATA;

// PEB32/PEB64
typedef struct _PEB32
{
    UCHAR InheritedAddressSpace;
    UCHAR ReadImageFileExecOptions;
    UCHAR BeingDebugged;
    UCHAR BitField;
    ULONG Mutant;
    ULONG ImageBaseAddress;
    ULONG Ldr;
    ULONG ProcessParameters;
    ULONG SubSystemData;
    ULONG ProcessHeap;
    ULONG FastPebLock;
    ULONG AtlThunkSListPtr;
    ULONG IFEOKey;
    ULONG CrossProcessFlags;
    ULONG UserSharedInfoPtr;
    ULONG SystemReserved;
    ULONG AtlThunkSListPtr32;
    ULONG ApiSetMap;
} PEB32, *PPEB32;

typedef struct _PEB
{
    UCHAR InheritedAddressSpace;
    UCHAR ReadImageFileExecOptions;
    UCHAR BeingDebugged;
    UCHAR BitField;
    PVOID Mutant;
    PVOID ImageBaseAddress;
    PPEB_LDR_DATA Ldr;
    PVOID ProcessParameters;
    PVOID SubSystemData;
    PVOID ProcessHeap;
    PVOID FastPebLock;
    PVOID AtlThunkSListPtr;
    PVOID IFEOKey;
    PVOID CrossProcessFlags;
    PVOID KernelCallbackTable;
    ULONG SystemReserved;
    ULONG AtlThunkSListPtr32;
    PVOID ApiSetMap;
} PEB, *PPEB;

typedef struct _LDR_DATA_TABLE_ENTRY32
{
    LIST_ENTRY32 InLoadOrderLinks;
    LIST_ENTRY32 InMemoryOrderLinks;
    LIST_ENTRY32 InInitializationOrderLinks;
    ULONG DllBase;
    ULONG EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING32 FullDllName;
    UNICODE_STRING32 BaseDllName;
    ULONG Flags;
    USHORT LoadCount;
    USHORT TlsIndex;
    LIST_ENTRY32 HashLinks;
    ULONG TimeDateStamp;
} LDR_DATA_TABLE_ENTRY32, *PLDR_DATA_TABLE_ENTRY32;

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;

typedef struct _THREAD_BASIC_INFORMATION
{
    NTSTATUS ExitStatus;
    PVOID TebBaseAddress;
    CLIENT_ID ClientId;
    ULONG_PTR AffinityMask;
    LONG Priority;
    LONG BasePriority;
} THREAD_BASIC_INFORMATION, *PTHREAD_BASIC_INFORMATION;

typedef struct _NT_PROC_THREAD_ATTRIBUTE_ENTRY
{
    ULONG Attribute;    // PROC_THREAD_ATTRIBUTE_XXX
    SIZE_T Size;
    ULONG_PTR Value;
    ULONG Unknown;
} NT_PROC_THREAD_ATTRIBUTE_ENTRY, *NT_PPROC_THREAD_ATTRIBUTE_ENTRY;
```

```
typedef struct _NT_PROC_THREAD_ATTRIBUTE_LIST
{
    ULONG Length;
    NT_PROC_THREAD_ATTRIBUTE_ENTRY Entry[1];
} NT_PROC_THREAD_ATTRIBUTE_LIST, *PNT_PROC_THREAD_ATTRIBUTE_LIST;


// 注入ShellCode结构
typedef struct _INJECT_BUFFER
{
    UCHAR Code[0x200];
    union
    {
        UNICODE_STRING Path64;
        UNICODE_STRING32 Path32;
    };
    wchar_t Buffer[488];
    PVOID ModuleHandle;
    ULONG Complete;
    NTSTATUS Status;
} INJECT_BUFFER, *PINJECT_BUFFER;

// ----------------------------------------------------------------------------------
// 一些开发中的通用函数封装，可任意拷贝使用
// ----------------------------------------------------------------------------------

// 传入函数名获取SSDT导出表RVA
// 参数1：传入函数名称
ULONG GetSSDTRVA(UCHAR *function_name)
{
    NTSTATUS Status;
    HANDLE FileHandle;
    IO_STATUS_BLOCK ioStatus;
    FILE_STANDARD_INFORMATION FileInformation;

    // 设置NTDLL路径
    UNICODE_STRING uniFileName;
    RtlInitUnicodeString(&uniFileName, L"\\SystemRoot\\system32\\ntoskrnl.exe");

    // 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
    ULONG uFileSize = FileInformation.EndOfFile.LowPart;

    // 分配内存
    PVOID pBuffer = ExAllocatePoolWithTag(PagedPool, uFileSize + 0x100,
(ULONG)"PGu");
    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;
    }

    // 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)pBuffer;

    // 取出PE头结构
    pNtHeaders = (PIMAGE_NT_HEADERS)((ULONGLONG)pBuffer + 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)pBuffer + 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;
        }
    }
    AddressOfFunctions = (PULONG)((ULONGLONG)pBuffer + 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;
        }
    }
    AddressOfNameOrdinals = (PUSHORT)((ULONGLONG)pBuffer + 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;

    // 循环所有节
    for (UINT16 Index = 0; Index < pNtHeaders->FileHeader.NumberOfSections;
Index++, pSectionHeader++)
    {
        // 寻找符合条件的节
        if (pSectionHeader->VirtualAddress <= FileOffset && FileOffset <=
pSectionHeader->VirtualAddress + pSectionHeader->SizeOfRawData)
        {
            // 得到文件偏移
            FileOffset = FileOffset - pSectionHeader->VirtualAddress +
pSectionHeader->PointerToRawData;
        }
    }
    AddressOfNames = (PULONG)((ULONGLONG)pBuffer + FileOffset);

    //DbgPrint("\n AddressOfFunctions %llX AddressOfNameOrdinals %llX
AddressOfNames %llX  \n", (ULONGLONG)AddressOfFunctions- (ULONGLONG)pBuffer,
(ULONGLONG)AddressOfNameOrdinals- (ULONGLONG)pBuffer, (ULONGLONG)AddressOfNames-
(ULONGLONG)pBuffer);
    //DbgPrint("\n AddressOfFunctions %llX AddressOfNameOrdinals %llX
AddressOfNames %llX  \n", pExportDirectory->AddressOfFunctions, pExportDirectory-
>AddressOfNameOrdinals, pExportDirectory->AddressOfNames);

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

    // 循环导出表
    for (ULONG uIndex = 0; uIndex < pExportDirectory->NumberOfNames; uIndex++,
AddressOfNames++, AddressOfNameOrdinals++)
    {
        uAddressOfNames = *AddressOfNames;
        pSectionHeader = pOldSectionHeader;
        for (UINT16 Index = 0; Index < pNtHeaders->FileHeader.NumberOfSections;
Index++, pSectionHeader++)
        {
            // 函数地址在某个范围内
            if (pSectionHeader->VirtualAddress <= uAddressOfNames && uAddressOfNames <= pSectionHeader->VirtualAddress + pSectionHeader->SizeOfRawData)
            {
                uOffset = uAddressOfNames - pSectionHeader->VirtualAddress + pSectionHeader->PointerToRawData;
            }
        }

        // 得到函数名
        FunName = (LPSTR)((ULONGLONG)pBuffer + uOffset);

        // 判断是否符合要求
        if (!_stricmp((const char *)function_name, FunName))
        {
            // 返回函数地址
            TargetOff = (ULONG)AddressOfFunctions[*AddressOfNameOrdinals];
            DbgPrint("索引 [ %p ] 函数名 [ %s ] 相对RVA [ %p ] \n", *AddressOfNameOrdinals, FunName, TargetOff);
        }

    }

    ExFreePoolWithTag(pBuffer, (ULONG)"PGu");
    ZwClose(FileHandle);
    return TargetOff;
}

// 传入函数名 获取该函数所在模块下标
ULONG GetIndexByName(UCHAR *function_name)
{
    NTSTATUS Status;
    HANDLE FileHandle;
    IO_STATUS_BLOCK ioStatus;
    FILE_STANDARD_INFORMATION FileInformation;

    // 设置NTDLL路径
    UNICODE_STRING uniFileName;
    RtlInitUnicodeString(&uniFileName, L"\\SystemRoot\\system32\\ntdll.dll");

    // 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
    ULONG uFileSize = FileInformation.EndOfFile.LowPart;

    // 分配内存
    PVOID pBuffer = ExAllocatePoolWithTag(PagedPool, uFileSize + 0x100, (ULONG)"Ntdl");
    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;
    }

    // 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)pBuffer;

    // 取出PE头结构
    pNtHeaders = (PIMAGE_NT_HEADERS)((ULONGLONG)pBuffer + 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)pBuffer + 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;
        }
    }

    // 此处需要注意foa和rva转换过程
    AddressOfFunctions = (PULONG)((ULONGLONG)pBuffer + 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;
        }
    }

    // 此处需要注意foa和rva转换过程
    AddressOfNameOrdinals = (PUSHORT)((ULONGLONG)pBuffer + 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;
        }
    }

    // 此处需要注意foa和rva转换过程
    AddressOfNames = (PULONG)((ULONGLONG)pBuffer + FileOffset);

    // 分析导出表
    ULONG uNameOffset;
    ULONG uOffset;
    LPSTR FunName;
    PVOID pFuncAddr;
    ULONG uServerIndex;
    ULONG uAddressOfNames;

    for (ULONG uIndex = 0; uIndex < pExportDirectory->NumberOfNames; uIndex++,
AddressOfNames++, AddressOfNameOrdinals++)
    {
        uAddressOfNames = *AddressOfNames;
        pSectionHeader = pOldSectionHeader;
        for (UINT32 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)pBuffer + uOffset);

        // 判断开头是否是Zw
        if (FunName[0] == 'Z' && FunName[1] == 'w')
        {
            pSectionHeader = pOldSectionHeader;

            // 如果是则根据AddressOfNameOrdinals得到文件偏移
            uOffset = (ULONG)AddressOfFunctions[*AddressOfNameOrdinals];

            for (UINT32 Index = 0; Index < pNtHeaders-
>FileHeader.NumberOfSections; Index++, pSectionHeader++)
            {
                if (pSectionHeader->VirtualAddress <= uOffset && uOffset <=
pSectionHeader->VirtualAddress + pSectionHeader->SizeOfRawData)
                {
                    uNameOffset = uOffset - pSectionHeader->VirtualAddress +
pSectionHeader->PointerToRawData;
                }
            }

            pFuncAddr = (PVOID)((ULONGLONG)pBuffer + uNameOffset);
            uServerIndex = *(PULONG)((ULONGLONG)pFuncAddr + 4);
            FunName[0] = 'N';
            FunName[1] = 't';

            // 获得指定的编号
            if (!_stricmp(FunName, (const char *)function_name))
            {
                ExFreePoolWithTag(pBuffer, (ULONG)"Ntdl");
                ZwClose(FileHandle);
                return uServerIndex;
            }
        }
    }

    ExFreePoolWithTag(pBuffer, (ULONG)"Ntdl");
    ZwClose(FileHandle);
    return 0;
}

// 获取模块导出函数
PVOID GetModuleExportAddress(IN PVOID ModuleBase, IN PCCHAR FunctionName, IN
PEPROCESS EProcess)
{
    PIMAGE_DOS_HEADER ImageDosHeader = (PIMAGE_DOS_HEADER)ModuleBase;
    PIMAGE_NT_HEADERS32 ImageNtHeaders32 = NULL;
    PIMAGE_NT_HEADERS64 ImageNtHeaders64 = NULL;
    PIMAGE_EXPORT_DIRECTORY ImageExportDirectory = NULL;
    ULONG ExportDirectorySize = 0;
    ULONG_PTR FunctionAddress = 0;

    if (ModuleBase == NULL)
    {
        return NULL;
```

```c
    }

    if (ImageDosHeader->e_magic != IMAGE_DOS_SIGNATURE)
    {
        return NULL;
    }

    ImageNtHeaders32 = (PIMAGE_NT_HEADERS32)((PUCHAR)ModuleBase + ImageDosHeader->e_lfanew);
    ImageNtHeaders64 = (PIMAGE_NT_HEADERS64)((PUCHAR)ModuleBase + ImageDosHeader->e_lfanew);

    // 判断PE结构位数
    if (ImageNtHeaders64->OptionalHeader.Magic == IMAGE_NT_OPTIONAL_HDR64_MAGIC)
    {
        ImageExportDirectory = (PIMAGE_EXPORT_DIRECTORY)(ImageNtHeaders64->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress + (ULONG_PTR)ModuleBase);
        ExportDirectorySize = ImageNtHeaders64->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].Size;
    }
    else
    {
        ImageExportDirectory = (PIMAGE_EXPORT_DIRECTORY)(ImageNtHeaders32->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress + (ULONG_PTR)ModuleBase);
        ExportDirectorySize = ImageNtHeaders32->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].Size;
    }

    // 解析内存导出表
    PUSHORT pAddressOfOrds = (PUSHORT)(ImageExportDirectory->AddressOfNameOrdinals + (ULONG_PTR)ModuleBase);
    PULONG  pAddressOfNames = (PULONG)(ImageExportDirectory->AddressOfNames + (ULONG_PTR)ModuleBase);
    PULONG  pAddressOfFuncs = (PULONG)(ImageExportDirectory->AddressOfFunctions + (ULONG_PTR)ModuleBase);

    for (ULONG i = 0; i < ImageExportDirectory->NumberOfFunctions; ++i)
    {
        USHORT OrdIndex = 0xFFFF;
        PCHAR  pName = NULL;

        // 如果函数名小于等于0xFFFF 则说明是序号导出
        if ((ULONG_PTR)FunctionName <= 0xFFFF)
        {
            OrdIndex = (USHORT)i;
        }

        // 否则则说明是名字导出
        else if ((ULONG_PTR)FunctionName > 0xFFFF && i < ImageExportDirectory->NumberOfNames)
        {
            pName = (PCHAR)(pAddressOfNames[i] + (ULONG_PTR)ModuleBase);
            OrdIndex = pAddressOfOrds[i];
        }
        // 未知导出函数
        else
        {
            return NULL;
        }

        // 对比模块名是否是我们所需要的
        if (((ULONG_PTR)FunctionName <= 0xFFFF && (USHORT)
((ULONG_PTR)FunctionName) == OrdIndex + ImageExportDirectory->Base) ||
((ULONG_PTR)FunctionName > 0xFFFF && strcmp(pName, FunctionName) == 0))
        {
            // 是则保存下来
            FunctionAddress = pAddressOfFuncs[OrdIndex] + (ULONG_PTR)ModuleBase;
            break;
        }
    }
    return (PVOID)FunctionAddress;
}

// 获取指定用户模块基址
PVOID GetUserModuleAddress(IN PEPROCESS EProcess, IN PUNICODE_STRING ModuleName,
IN BOOLEAN IsWow64)
{
    if (EProcess == NULL)
    {
        return NULL;
    }

    __try
    {
        // 定时250ms毫秒
        LARGE_INTEGER Time = { 0 };
        Time.QuadPart = -250ll * 10 * 1000;

        // 32位执行
        if (IsWow64)
        {
            // 得到进程PEB进程环境块
            PPEB32 Peb32 = (PPEB32)PsGetProcessWow64Process(EProcess);
            if (Peb32 == NULL)
            {
                return NULL;
            }

            // 等待 250ms * 10
            for (INT i = 0; !Peb32->Ldr && i < 10; i++)
            {
                // 等待一会在执行
                KeDelayExecutionThread(KernelMode, TRUE, &Time);
            }

            // 没有找到返回空
            if (!Peb32->Ldr)
            {
                return NULL;
            }

            // 搜索 InLoadOrderModuleList
            for (PLIST_ENTRY32 ListEntry = (PLIST_ENTRY32)((PPEB_LDR_DATA32)Peb32->Ldr)->InLoadOrderModuleList.Flink; ListEntry != &((PPEB_LDR_DATA32)Peb32->Ldr)->InLoadOrderModuleList; ListEntry = (PLIST_ENTRY32)ListEntry->Flink)
            {
                UNICODE_STRING UnicodeString;
                PLDR_DATA_TABLE_ENTRY32 LdrDataTableEntry32 = CONTAINING_RECORD(ListEntry, LDR_DATA_TABLE_ENTRY32, InLoadOrderLinks);
                RtlUnicodeStringInit(&UnicodeString, (PWCH)LdrDataTableEntry32->BaseDllName.Buffer);

                // 判断模块名是否符合要求
                if (RtlCompareUnicodeString(&UnicodeString, ModuleName, TRUE) == 0)
                {
                    // 符合则返回模块基址
                    return (PVOID)LdrDataTableEntry32->DllBase;
                }
            }
        }

        // 64位执行
        else
        {
            // 得到进程PEB进程环境块
            PPEB Peb = PsGetProcessPeb(EProcess);
            if (!Peb)
            {
                return NULL;
            }

            // 等待
            for (INT i = 0; !Peb->Ldr && i < 10; i++)
            {
                // 将当前线程置于指定间隔的可警报或不可操作的等待状态
                KeDelayExecutionThread(KernelMode, TRUE, &Time);
            }
            if (!Peb->Ldr)
            {
                return NULL;
            }

            // 遍历链表
            for (PLIST_ENTRY ListEntry = Peb->Ldr->InLoadOrderModuleList.Flink; ListEntry != &Peb->Ldr->InLoadOrderModuleList; ListEntry = ListEntry->Flink)
            {
                PLDR_DATA_TABLE_ENTRY LdrDataTableEntry = CONTAINING_RECORD(ListEntry, LDR_DATA_TABLE_ENTRY, InLoadOrderLinks);

                // 判断模块名是否符合要求
                if (RtlCompareUnicodeString(&LdrDataTableEntry->BaseDllName, ModuleName, TRUE) == 0)
                {
                    // 返回模块基址
                    return LdrDataTableEntry->DllBase;
                }
            }
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
        return NULL;
    }

    return NULL;
}

//得到ntos的基址
ULONGLONG GetOsBaseAddress(PDRIVER_OBJECT pDriverObject)
{
    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;

        // 如果是ntoskrnl.exe则返回该模块基址
        if (RtlEqualUnicodeString(&pTempDataTableEntry->BaseDllName, &osName,
TRUE))
        {
            return (ULONGLONG)pTempDataTableEntry->DllBase;
        }
        pList = pList->Flink;
    }
    return 0;
}

// 得到SSDT表的基地址
```

```
ULONGLONG GetKeServiceDescriptorTable64(PDRIVER_OBJECT DriverObject)
{

    /*
    nt!KiSystemServiceUser+0xdc:
    fffff806`42c79987 8bf8            mov     edi,eax
    fffff806`42c79989 c1ef07          shr     edi,7
    fffff806`42c7998c 83e720          and     edi,20h
    fffff806`42c7998f 25ff0f0000      and     eax,0FFFh

    nt!KiSystemServiceRepeat:
    fffff806`42c79994 4c8d15e59e3b00  lea     r10,[nt!KeServiceDescriptorTable
(fffff806`43033880)]
    fffff806`42c7999b 4c8d1dde203a00  lea     r11,
[nt!KeServiceDescriptorTableShadow (fffff806`4301ba80)]
    fffff806`42c799a2 f7437880000000  test    dword ptr [rbx+78h],80h
    fffff806`42c799a9 7413            je      nt!KiSystemServiceRepeat+0x2a
(fffff806`42c799be)
    */
    char KiSystemServiceStart_pattern[14] =
"\x8B\xF8\xC1\xEF\x07\x83\xE7\x20\x25\xFF\x0F\x00\x00";

    /*
    ULONG rva = GetRvaFromModule(L"\\SystemRoot\\system32\\ntoskrnl.exe",
"_stricmp");
    DbgPrint("NtReadFile VA = %p \n", rva);
    ULONG _stricmp_offset = 0x19d710;
    */

    ULONGLONG CodeScanStart = GetSSDTRVA((UCHAR *)"_stricmp") +
GetOsBaseAddress(DriverObject);

    ULONGLONG i, tbl_address, b;
    for (i = 0; i < 0x50000; i++)
    {
        // 比较特征
        if (!memcmp((char*)(ULONGLONG)CodeScanStart + i,
(char*)KiSystemServiceStart_pattern, 13))
        {
            for (b = 0; b < 50; b++)
            {
                tbl_address = ((ULONGLONG)CodeScanStart + i + b);

                // 4c 8d 15 e5 9e 3b 00  lea r10,[nt!KeServiceDescriptorTable
(fffff802`64da4880)]
                // if (*(USHORT*)((ULONGLONG)tbl_address) == (USHORT)0x158d4c)
                if (*(USHORT*)((ULONGLONG)tbl_address) == (USHORT)0x8d4c)
                {
                    return ((LONGLONG)tbl_address + 7) + *(LONG*)(tbl_address +
3);
                }
            }
        }
    }
    return 0;
}
```

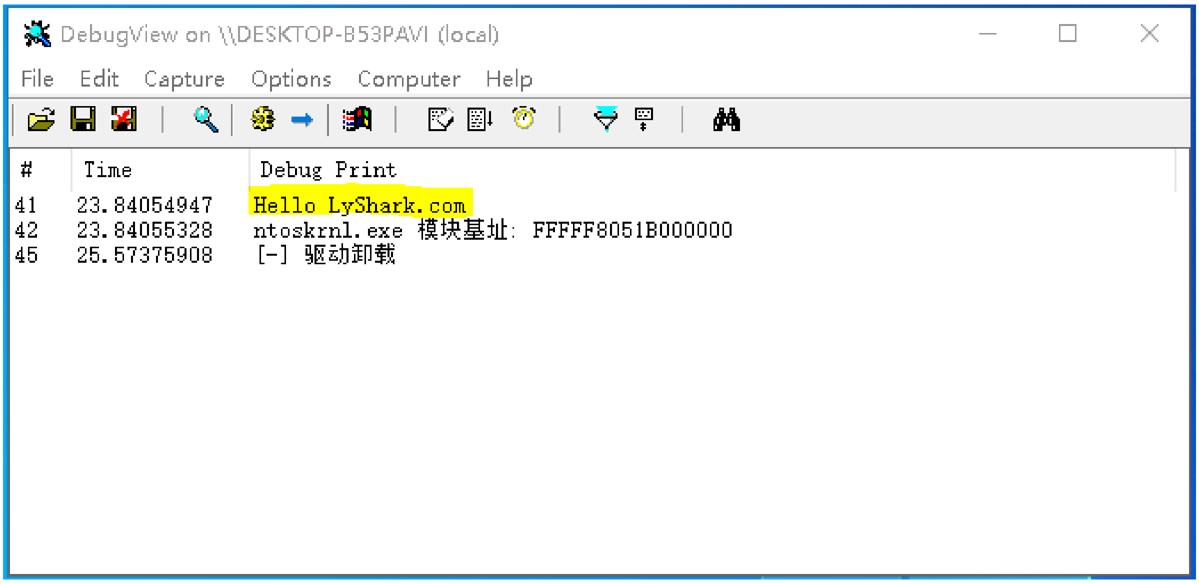
```
// 根据SSDT序号得到函数基址
ULONGLONG GetSSDTFuncCurAddr(ULONG index)
{
    /*
    mov rax, rcx                   ; rcx=Native API 的 index
    lea r10,[rdx]                  ; rdx=ssdt 基址
    mov edi,eax                    ; index
    shr edi,7
    and edi,20h
    mov r10, qword ptr [r10+rdi]   ; ServiceTableBase
    movsxd r11,dword ptr [r10+rax] ; 没有右移的假ssdt的地址
    mov rax,r11
    sar r11,4
    add r10,r11
    mov rax,r10
    ret
    */
    LONG dwtmp = 0;
    PULONG ServiceTableBase = NULL;
    ServiceTableBase = (PULONG)KeServiceDescriptorTable->ServiceTableBase;
    dwtmp = ServiceTableBase[index];

    // 先右移4位之后加上基地址 就可以得到ssdt的地址
    dwtmp = dwtmp >> 4;

    return (LONGLONG)dwtmp + (ULONGLONG)ServiceTableBase;
}

// 根据进程ID返回进程EPROCESS
PEPROCESS LookupProcess(HANDLE Pid)
{
    PEPROCESS eprocess = NULL;
    if (NT_SUCCESS(PsLookupProcessByProcessId(Pid, &eprocess)))
    {
        return eprocess;
    }
    else
    {
        return NULL;
    }
}

// 根据用户传入进程名得到该进程PID
HANDLE GetProcessID(PCHAR ProcessName)
{
    ULONG i = 0;
    PEPROCESS eproc = NULL;
    for (i = 4; i<100000000; i = i + 4)
    {
        eproc = LookupProcess((HANDLE)i);
        if (eproc != NULL)
        {
            ObDereferenceObject(eproc);

            // 根据进程名得到进程EPEPROCESS
            if (strstr(PsGetProcessImageFileName(eproc), ProcessName) != NULL)
            {
                return PsGetProcessId(eproc);
            }
        }
    }
    return NULL;
}
```

为了能更好的完成驱动注入实现原理的讲解，也可以让用户理解如上方所封装的API函数的使用流程，接下来将依次讲解上方这些通用API函数的作用以及使用方法，其目的是让用户可以更好的学会功能运用，以此在后期项目开发中可以更好的使用这些功能。

GetOsBaseAddress: 该函数可实现输出特定内核模块的基地址，本例中写死在了变量`wzData`中，如果需要改进只需要替换参数传递即可实现自定义取值，调用该函数你只需要传入`PDRIVER_OBJECT`自身驱动对象即可，代码如下所示；

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#include "lyshark.h"

VOID Unload(PDRIVER_OBJECT pDriverObj)
{
    DbgPrint("[-] 驱动卸载 \n");
}

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegPath)
{
    DbgPrint("Hello LyShark.com \n");

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

    DriverObject->DriverUnload = Unload;
    return STATUS_SUCCESS;
}
```

编译并运行如上代码片段，即可输出`ntoskrnl.exe`内核模块的基址，效果图如下所示；

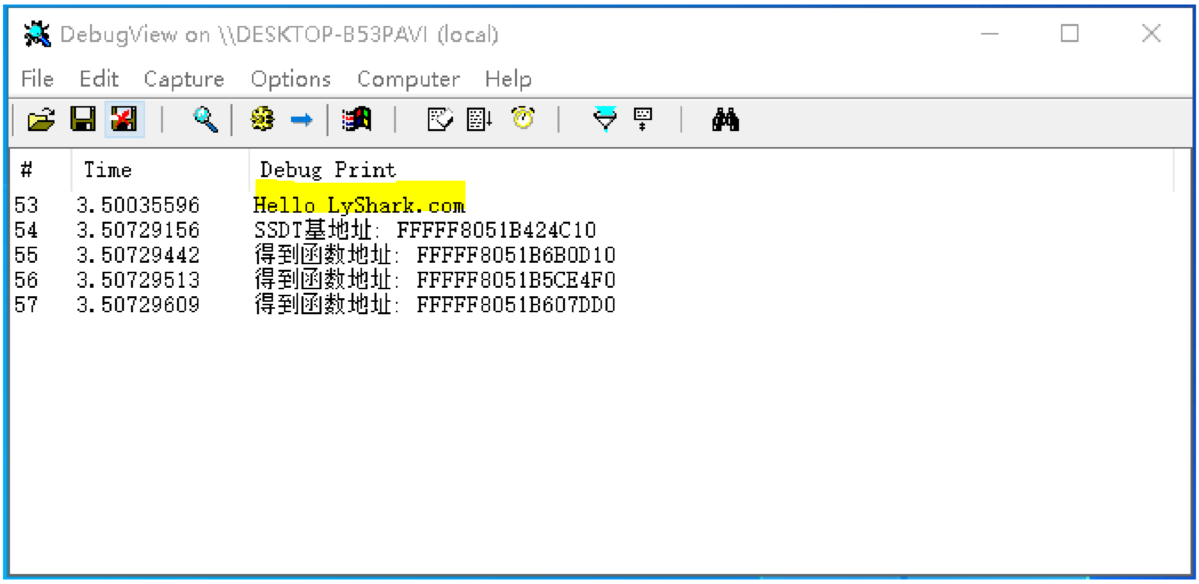

DebugView on \\DESKTOP-B53PAVI (local)
File Edit Capture Options Computer Help

```
#    Time          Debug Print
41   23.84054947   Hello LyShark.com
42   23.84055328   ntoskrnl.exe 模块基址: FFFFF8051B000000
45   25.57375908   [-] 驱动卸载
```

GetSSDTFuncCurAddr: 该函数可实现根据用户传入的SSDT表下标，获取到该函数的基址，代码如下所示；

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#include "lyshark.h"

VOID Unload(PDRIVER_OBJECT pDriverObj)
{
    DbgPrint("[-] 驱动卸载 \n");
}

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegPath)
{
    DbgPrint("Hello LyShark.com \n");

    // 得到SSDT基地址
    KeServiceDescriptorTable =
(PSYSTEM_SERVICE_TABLE)GetKeServiceDescriptorTable64(DriverObject);
    DbgPrint("SSDT基地址: %p \n", KeServiceDescriptorTable->ServiceTableBase);

    // 根据序号得到指定函数地址
    ULONGLONG address = NULL;

    address = GetSSDTFuncCurAddr(10);
    DbgPrint("得到函数地址: %p \n", address);

    address = GetSSDTFuncCurAddr(11);
    DbgPrint("得到函数地址: %p \n", address);

    address = GetSSDTFuncCurAddr(12);
    DbgPrint("得到函数地址: %p \n", address);

    DriverObject->DriverUnload = Unload;
    return STATUS_SUCCESS;
}
```

编译并运行如上代码片段，即可输出下标为`10，11，12`的SSDT函数基址，效果图如下所示；

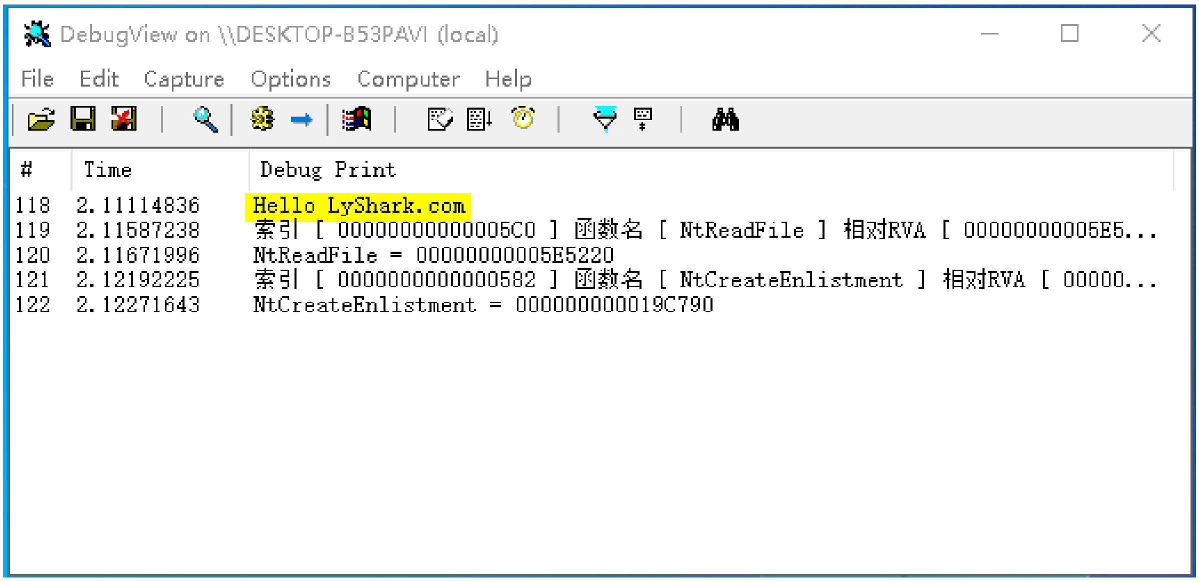

DebugView on \\DESKTOP-B53PAVI (local)
File Edit Capture Options Computer Help

```
#    Time          Debug Print
53   3.50035596    Hello LyShark.com
54   3.50729156    SSDT基地址: FFFFF8051B424C10
55   3.50729442    得到函数地址: FFFFF8051B6B0D10
56   3.50729513    得到函数地址: FFFFF8051B5CE4F0
57   3.50729609    得到函数地址: FFFFF8051B607DD0
```

GetSSDTRVA: 根据传入的函数名获取该函数的RVA地址，用户传入一个特定模块下导出函数的函数名，动态得到该函数的相对偏移地址。

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#include "lyshark.h"

VOID Unload(PDRIVER_OBJECT pDriverObj)
{
    DbgPrint("[-] 驱动卸载 \n");
}

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegPath)
{
    DbgPrint("Hello LyShark.com \n");

    ULONG64 ReadFile_RVA = GetSSDTRVA("NtReadFile");
    DbgPrint("NtReadFile = %p \n", ReadFile_RVA);

    ULONG64 NtCreateEnlistment_RVA = GetSSDTRVA("NtCreateEnlistment");
    DbgPrint("NtCreateEnlistment = %p \n", NtCreateEnlistment_RVA);

    DriverObject->DriverUnload = Unload;
    return STATUS_SUCCESS;
}
```

编译并运行如上代码片段，即可输出`NtReadFile,NtCreateEnlistment`两个内核函数的RVA地址，效果图如下所示；

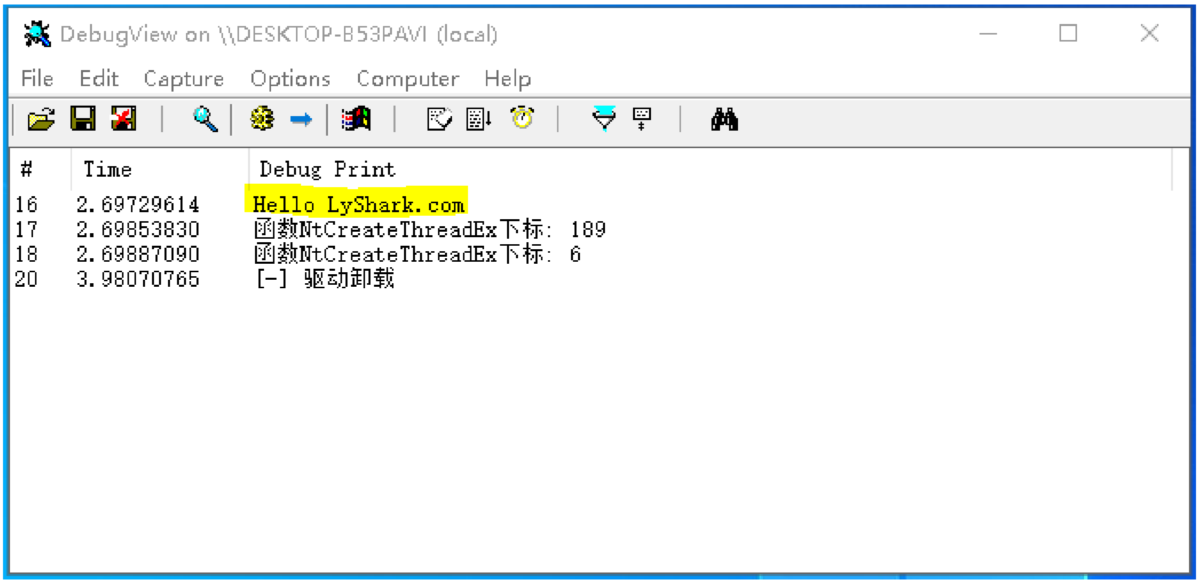

DebugView on \\DESKTOP-B53PAVI (local)
File Edit Capture Options Computer Help

```
#    Time          Debug Print
118  2.11114836    Hello LyShark.com
119  2.11587238    索引 [ 00000000000005C0 ] 函数名 [ NtReadFile ] 相对RVA [ 00000000005E5...
120  2.11671996    NtReadFile = 00000000005E5220
121  2.12192225    索引 [ 0000000000000582 ] 函数名 [ NtCreateEnlistment ] 相对RVA [ 00000...
122  2.12271643    NtCreateEnlistment = 000000000019C790
```

GetIndexByName: 该函数接收用户传入的一个`SSDT`函数名，并返回该函数所对应的下标，调用代码如下；

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#include "lyshark.h"

VOID Unload(PDRIVER_OBJECT pDriverObj)
{
    DbgPrint("[-] 驱动卸载 \n");
}

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegPath)
{
    DbgPrint("Hello LyShark.com \n");

    ULONG index1 = GetIndexByName((UCHAR *)"NtCreateThreadEx");
    DbgPrint("函数NtCreateThreadEx下标: %d \n", index1);

    ULONG index2 = GetIndexByName((UCHAR *)"NtReadFile");
    DbgPrint("函数NtReadFile下标: %d \n", index2);

    DriverObject->DriverUnload = Unload;
    return STATUS_SUCCESS;
}
```

编译并运行如上代码片段，即可输出`NtCreateThreadEx,NtReadFile`两个内核函数的下标，效果图如下所示；

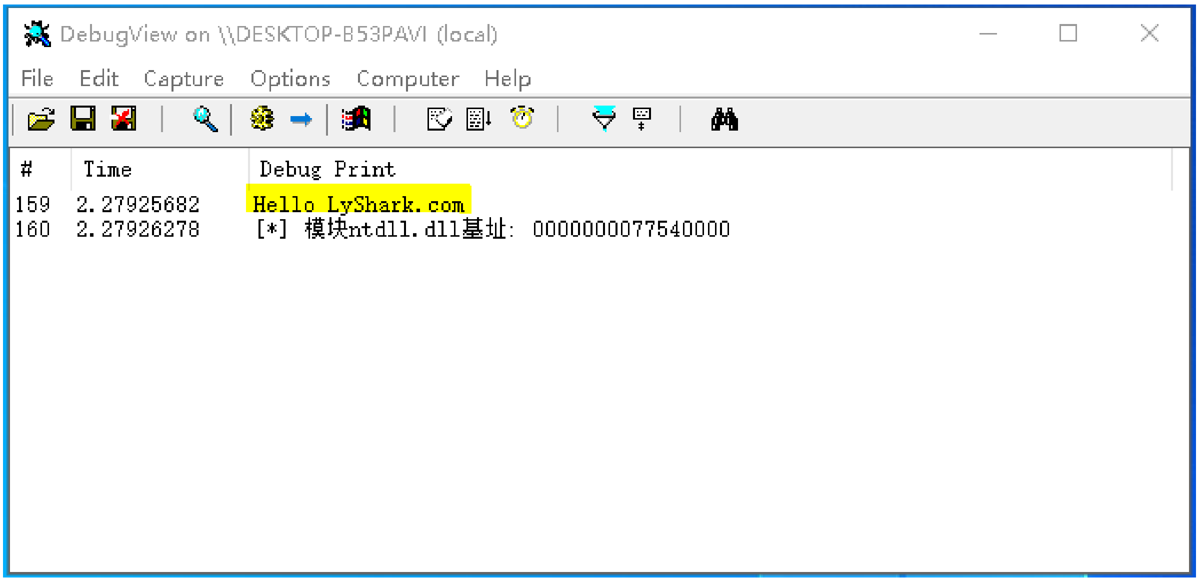

DebugView on \\DESKTOP-B53PAVI (local)
File Edit Capture Options Computer Help

```
#   Time         Debug Print
16  2.69729614   Hello LyShark.com
17  2.69853830   函数NtCreateThreadEx下标: 189
18  2.69887090   函数NtCreateThreadEx下标: 6
20  3.98070765   [-] 驱动卸载
```

GetUserModuleAddress: 该函数用于获取进程模块基址，在内核模式下附加到应用层指定进程上，并动态获取到该进程所加载的指定模块的基址，调用代码如下；

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#include "lyshark.h"

VOID Unload(PDRIVER_OBJECT pDriverObj)
{
    DbgPrint("[-] 驱动卸载 \n");
}

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegPath)
{
    DbgPrint("Hello LyShark.com \n");

    HANDLE ProcessID = (HANDLE)6932;

    PEPROCESS EProcess = NULL;
    NTSTATUS Status = STATUS_SUCCESS;
    KAPC_STATE ApcState;

    // 根据PID得到进程EProcess结构
    Status = PsLookupProcessByProcessId(ProcessID, &EProcess);
    if (Status != STATUS_SUCCESS)
    {
        DbgPrint("[-] 获取EProcessID失败 \n");
        return Status;
    }

    // 判断目标进程是32位还是64位
    BOOLEAN IsWow64 = (PsGetProcessWow64Process(EProcess) != NULL) ? TRUE :
FALSE;

    // 验证地址是否可读
    if (!MmIsAddressValid(EProcess))
    {
        DbgPrint("[-] 地址不可读 \n");
        DriverObject->DriverUnload = Unload;
        return STATUS_SUCCESS;
    }

    // 将当前线程连接到目标进程的地址空间(附加进程)
    KeStackAttachProcess((PRKPROCESS)EProcess, &ApcState);

    __try
    {
        UNICODE_STRING NtdllUnicodeString = { 0 };
        PVOID NtdllAddress = NULL;

        // 得到进程内ntdll.dll模块基地址
        RtlInitUnicodeString(&NtdllUnicodeString, L"Ntdll.dll");

        NtdllAddress = GetUserModuleAddress(EProcess, &NtdllUnicodeString,
IsWow64);

        if (!NtdllAddress)
        {
            DbgPrint("[-] 没有找到基址 \n");
            DriverObject->DriverUnload = Unload;
            return STATUS_SUCCESS;
        }

        DbgPrint("[*] 模块ntdll.dll基址: %p \n", NtdllAddress);
    }
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
    }

    // 取消附加
    KeUnstackDetachProcess(&ApcState);
    DriverObject->DriverUnload = Unload;
    return STATUS_SUCCESS;
}
```

编译并运行如上代码片段，则获取进程`PID=6932`里面的`ntdll.dll`模块的基址，输出效果图如下所示;



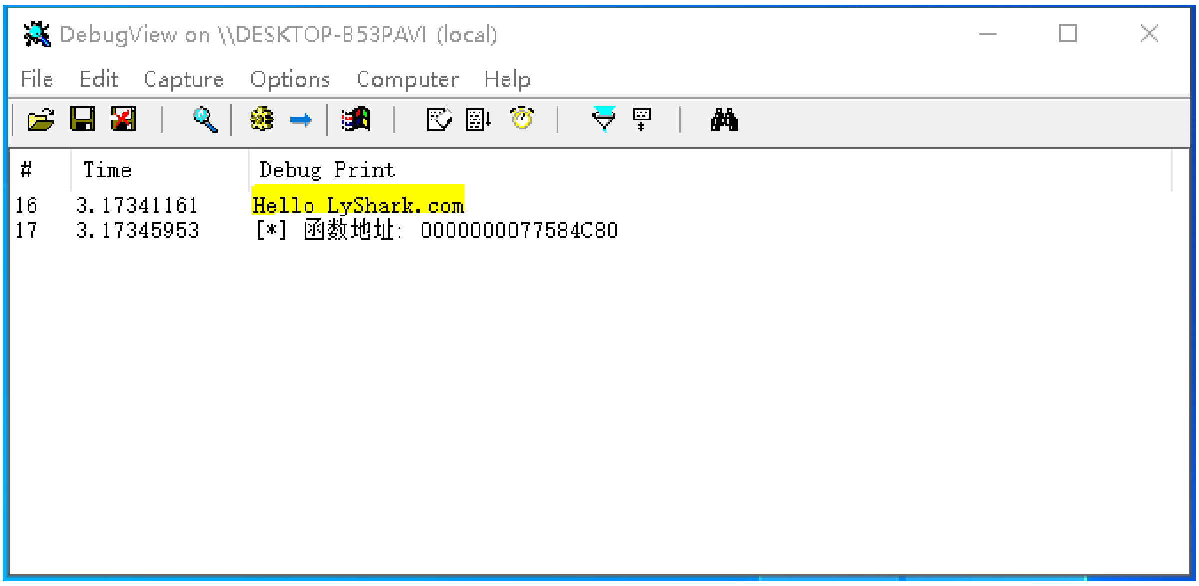
DebugView on \\DESKTOP-B53PAVI (local)
File Edit Capture Options Computer Help

```
#    Time          Debug Print
159  2.27925682    Hello LyShark.com
160  2.27926278    [*] 模块ntdll.dll基址: 0000000077540000
```

GetModuleExportAddress: 该函数可用于获取特定模块中特定函数的基址，此功能需要配合获取模块基址一起使用，调用代码如下；

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#include "lyshark.h"

VOID Unload(PDRIVER_OBJECT pDriverObj)
{
    DbgPrint("[-] 驱动卸载 \n");
}

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegPath)
{
    DbgPrint("Hello LyShark.com \n");

    HANDLE ProcessID = (HANDLE)6932;
    PEPROCESS EProcess = NULL;
    NTSTATUS Status = STATUS_SUCCESS;

    // 根据PID得到进程EProcess结构
    Status = PsLookupProcessByProcessId(ProcessID, &EProcess);
    if (Status != STATUS_SUCCESS)
    {
        DbgPrint("[-] 获取EProcessID失败 \n");
        return Status;
    }

    PVOID BaseAddress = (PVOID)0x77540000;
    PVOID RefAddress = 0;

    // 传入Ntdll.dll基址 + 函数名 得到该函数地址
    RefAddress = GetModuleExportAddress(BaseAddress, "LdrLoadDll", EProcess);
    DbgPrint("[*] 函数地址: %p \n", RefAddress);

    DriverObject->DriverUnload = Unload;
    return STATUS_SUCCESS;
}
```

编译并运行如上代码片段，则获取进程`PID=6932`里面的`ntdll.dll`模块里的`LdrLoadDll`函数基址，输出效果图如下所示；


DebugView on \\DESKTOP-B53PAVI (local)
File Edit Capture Options Computer Help

```
#   Time          Debug Print
16  3.17341161    Hello LyShark.com
17  3.17345953    [*] 函数地址: 0000000077584C80
```

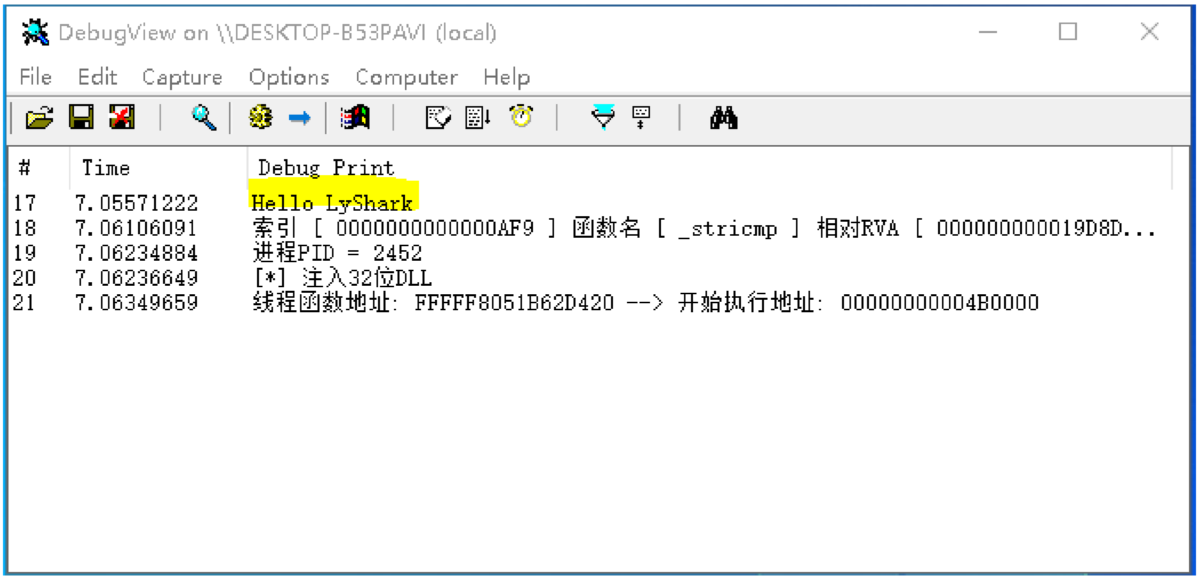
SeCreateThreadEx: 该函数则是实际执行注入的函数，此段代码中需要注意的是`pPrevMode`中的偏移值，每个系统中都不相同，用户需要自行在`WinDBG`中输入`!_KTHREAD`得到线程信息，并找到`PreviousMode`字段，该字段中的偏移值需要`(PUCHAR)PsGetCurrentThread() + 0x232`才可得到正确位置。

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#include "lyshark.h"

// --------------------------------------------------------------------------------
-----
// 注入代码生成函数
// --------------------------------------------------------------------------------
-----

// 创建64位注入代码
PINJECT_BUFFER GetNative64Code(IN PVOID LdrLoadDll, IN PUNICODE_STRING
DllFullPath)
{
    NTSTATUS Status = STATUS_SUCCESS;
    PINJECT_BUFFER InjectBuffer = NULL;
    SIZE_T Size = PAGE_SIZE;

    UCHAR Code[] = {
        0x48, 0x83, 0xEC, 0x28,                 // sub rsp, 0x28
        0x48, 0x31, 0xC9,                       // xor rcx, rcx
        0x48, 0x31, 0xD2,                       // xor rdx, rdx
        0x49, 0xB8, 0, 0, 0, 0, 0, 0, 0, 0,     // mov r8, ModuleFileName
offset +12
```

```
        0x49, 0xB9, 0, 0, 0, 0, 0, 0, 0, 0,     // mov r9, ModuleHandle
offset +28
        0x48, 0xB8, 0, 0, 0, 0, 0, 0, 0, 0,     // mov rax, LdrLoadDll
offset +32
        0xFF, 0xD0,                             // call rax
        0x48, 0xBA, 0, 0, 0, 0, 0, 0, 0, 0,     // mov rdx, COMPLETE_OFFSET
offset +44
        0xC7, 0x02, 0x7E, 0x1E, 0x37, 0xC0,     // mov [rdx], CALL_COMPLETE
        0x48, 0xBA, 0, 0, 0, 0, 0, 0, 0, 0,     // mov rdx, STATUS_OFFSET
offset +60
        0x89, 0x02,                             // mov [rdx], eax
        0x48, 0x83, 0xC4, 0x28,                 // add rsp, 0x28
        0xC3                                    // ret
    };

    Status = ZwAllocateVirtualMemory(ZwCurrentProcess(), &InjectBuffer, 0, &Size,
MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    if (NT_SUCCESS(Status))
    {
        PUNICODE_STRING UserPath = &InjectBuffer->Path64;
        UserPath->Length = 0;
        UserPath->MaximumLength = sizeof(InjectBuffer->Buffer);
        UserPath->Buffer = InjectBuffer->Buffer;

        RtlUnicodeStringCopy(UserPath, DllFullPath);

        // Copy code
        memcpy(InjectBuffer, Code, sizeof(Code));

        // Fill stubs
        *(ULONGLONG*)((PUCHAR)InjectBuffer + 12) = (ULONGLONG)UserPath;
        *(ULONGLONG*)((PUCHAR)InjectBuffer + 22) = (ULONGLONG)&InjectBuffer-
>ModuleHandle;
        *(ULONGLONG*)((PUCHAR)InjectBuffer + 32) = (ULONGLONG)LdrLoadDll;
        *(ULONGLONG*)((PUCHAR)InjectBuffer + 44) = (ULONGLONG)&InjectBuffer-
>Complete;
        *(ULONGLONG*)((PUCHAR)InjectBuffer + 60) = (ULONGLONG)&InjectBuffer-
>Status;

        return InjectBuffer;
    }

    UNREFERENCED_PARAMETER(DllFullPath);
    return NULL;
}

// 创建32位注入代码
PINJECT_BUFFER GetNative32Code(IN PVOID LdrLoadDll, IN PUNICODE_STRING
DllFullPath)
{
    NTSTATUS Status = STATUS_SUCCESS;
    PINJECT_BUFFER InjectBuffer = NULL;
    SIZE_T Size = PAGE_SIZE;

    // Code
    UCHAR Code[] = {
        0x68, 0, 0, 0, 0,                       // push ModuleHandle
offset +1
        0x68, 0, 0, 0, 0,                       // push ModuleFileName
offset +6
        0x6A, 0,                                // push Flags
        0x6A, 0,                                // push PathToFile
        0xE8, 0, 0, 0, 0,                       // call LdrLoadDll
offset +15
        0xBA, 0, 0, 0, 0,                       // mov edx, COMPLETE_OFFSET
offset +20
        0xC7, 0x02, 0x7E, 0x1E, 0x37, 0xC0,     // mov [edx], CALL_COMPLETE
        0xBA, 0, 0, 0, 0,                       // mov edx, STATUS_OFFSET
offset +31
        0x89, 0x02,                             // mov [edx], eax
        0xC2, 0x04, 0x00                        // ret 4
    };

    Status = ZwAllocateVirtualMemory(ZwCurrentProcess(), &InjectBuffer, 0, &Size,
MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    if (NT_SUCCESS(Status))
    {
        // Copy path
        PUNICODE_STRING32 pUserPath = &InjectBuffer->Path32;
        pUserPath->Length = DllFullPath->Length;
        pUserPath->MaximumLength = DllFullPath->MaximumLength;
        pUserPath->Buffer = (ULONG)(ULONG_PTR)InjectBuffer->Buffer;

        // Copy path
        memcpy((PVOID)pUserPath->Buffer, DllFullPath->Buffer, DllFullPath-
>Length);

        // Copy code
        memcpy(InjectBuffer, Code, sizeof(Code));

        // Fill stubs
        *(ULONG*)((PUCHAR)InjectBuffer + 1) = (ULONG)(ULONG_PTR)&InjectBuffer-
>ModuleHandle;
        *(ULONG*)((PUCHAR)InjectBuffer + 6) = (ULONG)(ULONG_PTR)pUserPath;
        *(ULONG*)((PUCHAR)InjectBuffer + 15) = (ULONG)((ULONG_PTR)LdrLoadDll -
((ULONG_PTR)InjectBuffer + 15) - 5 + 1);
        *(ULONG*)((PUCHAR)InjectBuffer + 20) = (ULONG)(ULONG_PTR)&InjectBuffer-
>Complete;
        *(ULONG*)((PUCHAR)InjectBuffer + 31) = (ULONG)(ULONG_PTR)&InjectBuffer-
>Status;

        return InjectBuffer;
    }

    UNREFERENCED_PARAMETER(DllFullPath);
    return NULL;
}

// ------------------------------------------------------------------------------
-----
// 启动子线程函数(注入函数)
// --------------------------------------------------------------------------------
-----

// 启动线程
NTSTATUS NTAPI SeCreateThreadEx(OUT PHANDLE ThreadHandle, IN ACCESS_MASK
DesiredAccess, IN PVOID ObjectAttributes, IN HANDLE ProcessHandle, IN PVOID
StartAddress, IN PVOID Parameter, IN ULONG Flags, IN SIZE_T StackZeroBits, IN
SIZE_T SizeOfStackCommit, IN SIZE_T SizeOfStackReserve, IN
PNT_PROC_THREAD_ATTRIBUTE_LIST AttributeList)
{
    NTSTATUS Status = STATUS_SUCCESS;

    // 根据字符串NtCreateThreadEx得到下标,并通过下标查询SSDT函数地址
    LPFN_NTCREATETHREADEX NtCreateThreadEx = (LPFN_NTCREATETHREADEX)
(GetSSDTFuncCurAddr(GetIndexByName((UCHAR *)"NtCreateThreadEx")));
    DbgPrint("线程函数地址: %p --> 开始执行地址: %p \n", NtCreateThreadEx,
StartAddress);

    if (NtCreateThreadEx)
    {
        // 如果之前的模式是用户模式，地址传递到ZwCreateThreadEx必须在用户模式空间
        // 切换到内核模式允许使用内核模式地址
        /*
        dt !_KTHREAD
        +0x1c8 Win32Thread      : Ptr64 Void
        + 0x140 WaitBlockFill11 : [176] UChar
        + 0x1f0 Ucb : Ptr64 _UMS_CONTROL_BLOCK
        + 0x232 PreviousMode : Char
        */

        // Windows10 PreviousMode = 0x232
        PUCHAR pPrevMode = (PUCHAR)PsGetCurrentThread() + 0x232;

        // 64位 pPrevMode = 01
        UCHAR prevMode = *pPrevMode;

        // 内核模式
        *pPrevMode = KernelMode;

        // 创建线程
        Status = NtCreateThreadEx(ThreadHandle, DesiredAccess, ObjectAttributes,
ProcessHandle, StartAddress, Parameter, Flags, StackZeroBits, SizeOfStackCommit,
SizeOfStackReserve, AttributeList);

        // 恢复之前的线程模式
        *pPrevMode = prevMode;
    }
    else
    {
        Status = STATUS_NOT_FOUND;
    }
    return Status;
}

// 执行线程
```

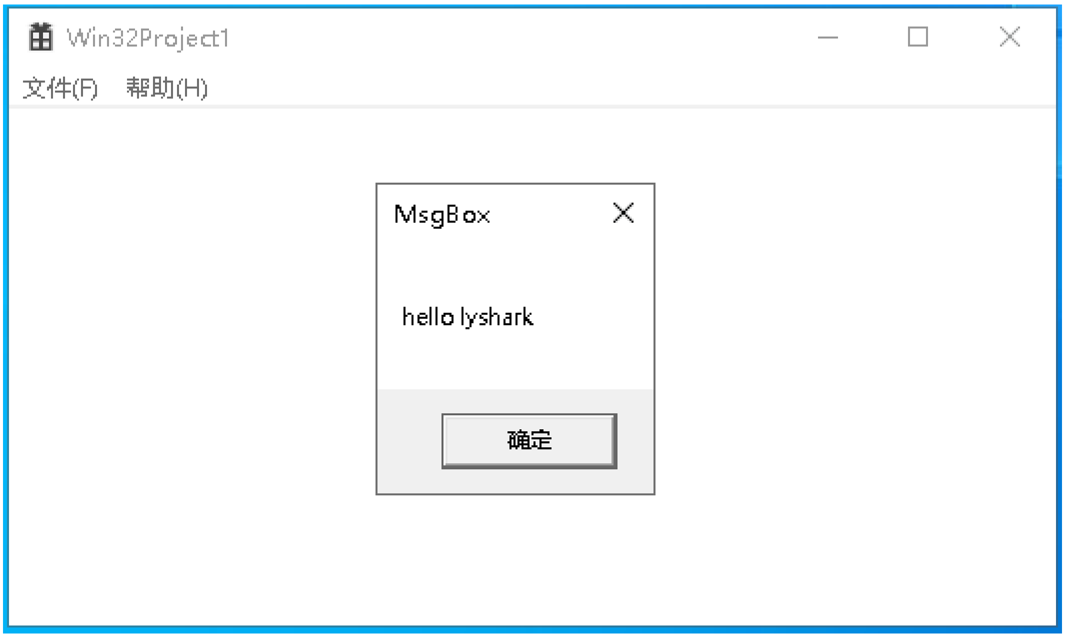
```
NTSTATUS ExecuteInNewThread(IN PVOID BaseAddress, IN PVOID Parameter, IN ULONG
Flags, IN BOOLEAN Wait, OUT PNTSTATUS ExitStatus)
{
    HANDLE ThreadHandle = NULL;
    OBJECT_ATTRIBUTES ObjectAttributes = { 0 };

    // 初始化对象属性
    InitializeObjectAttributes(&ObjectAttributes, NULL, OBJ_KERNEL_HANDLE, NULL,
NULL);

    // 创建线程
    NTSTATUS Status = SeCreateThreadEx(&ThreadHandle,
THREAD_QUERY_LIMITED_INFORMATION, &ObjectAttributes, ZwCurrentProcess(),
BaseAddress, Parameter, Flags, 0, 0x1000, 0x100000, NULL);
    // 等待线程完成
    if (NT_SUCCESS(Status) && Wait != FALSE)
    {
        // 延迟 60s
        LARGE_INTEGER Timeout = { 0 };
        Timeout.QuadPart = -(60ll * 10 * 1000 * 1000);

        Status = ZwWaitForSingleObject(ThreadHandle, TRUE, &Timeout);
        if (NT_SUCCESS(Status))
        {
            // 查询线程退出码
            THREAD_BASIC_INFORMATION ThreadBasicInfo = { 0 };
            ULONG ReturnLength = 0;

            Status = ZwQueryInformationThread(ThreadHandle,
ThreadBasicInformation, &ThreadBasicInfo, sizeof(ThreadBasicInfo),
&ReturnLength);

            if (NT_SUCCESS(Status) && ExitStatus)
            {
                // 这里是查询当前的dll是否注入成功
                *ExitStatus = ThreadBasicInfo.ExitStatus;
            }
            else if (!NT_SUCCESS(Status))
            {
                DbgPrint("%s: ZwQueryInformationThread failed with status
0x%X\n", __FUNCTION__, Status);
            }
        }
        else
        {
            DbgPrint("%s: ZwWaitForSingleObject failed with status 0x%X\n",
__FUNCTION__, Status);
        }
    }
    else
    {
        DbgPrint("%s: ZwCreateThreadEx failed with status 0x%X\n", __FUNCTION__,
Status);
    }

    if (ThreadHandle)
    {
        ZwClose(ThreadHandle);
    }
    return Status;
}

// 切换到目标进程创建内核线程进行注入 (cr3切换)
NTSTATUS AttachAndInjectProcess(IN HANDLE ProcessID, PWCHAR DllPath)
{
    PEPROCESS EProcess = NULL;
    KAPC_STATE ApcState;
    NTSTATUS Status = STATUS_SUCCESS;

    if (ProcessID == NULL)
    {
        Status = STATUS_UNSUCCESSFUL;
        return Status;
    }

    // 获取EProcess
    Status = PsLookupProcessByProcessId(ProcessID, &EProcess);
    if (Status != STATUS_SUCCESS)
    {
        return Status;
    }

    // 判断目标进程x86 or x64
    BOOLEAN IsWow64 = (PsGetProcessWow64Process(EProcess) != NULL) ? TRUE :
FALSE;

    // 将当前线程连接到目标进程的地址空间
    KeStackAttachProcess((PRKPROCESS)EProcess, &ApcState);
    __try
    {
        PVOID NtdllAddress = NULL;
        PVOID LdrLoadDll = NULL;
        UNICODE_STRING NtdllUnicodeString = { 0 };
        UNICODE_STRING DllFullPath = { 0 };

        // 获取ntdll模块基地址
        RtlInitUnicodeString(&NtdllUnicodeString, L"Ntdll.dll");
        NtdllAddress = GetUserModuleAddress(EProcess, &NtdllUnicodeString,
IsWow64);
        if (!NtdllAddress)
        {
            Status = STATUS_NOT_FOUND;
        }

        // 获取LdrLoadDll
        if (NT_SUCCESS(Status))
        {
            LdrLoadDll = GetModuleExportAddress(NtdllAddress, "LdrLoadDll",
EProcess);
            if (!LdrLoadDll)
            {
                Status = STATUS_NOT_FOUND;
            }
        }

        PINJECT_BUFFER InjectBuffer = NULL;
        if (IsWow64)
        {
            // 注入32位DLL
            RtlInitUnicodeString(&DllFullPath, DllPath);
            InjectBuffer = GetNative32Code(LdrLoadDll, &DllFullPath);
            DbgPrint("[*] 注入32位DLL \n");
        }
        else
        {
            // 注入64位DLL
            RtlInitUnicodeString(&DllFullPath, DllPath);
            InjectBuffer = GetNative64Code(LdrLoadDll, &DllFullPath);
            DbgPrint("[*] 注入64位DLL \n");
        }

        //创建线程,执行构造的 shellcode
        ExecuteInNewThread(InjectBuffer, NULL,
THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER, TRUE, &Status);
        if (!NT_SUCCESS(Status))
        {
            DbgPrint("ExecuteInNewThread Failed\n");
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
        Status = STATUS_UNSUCCESSFUL;
    }
    // 释放EProcess
    KeUnstackDetachProcess(&ApcState);
    ObDereferenceObject(EProcess);
    return Status;
}

VOID Unload(PDRIVER_OBJECT pDriverObj)
{
    DbgPrint("[-] 驱动卸载 \n");
}

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegPath)
{
    DbgPrint("Hello LyShark \n");

    // 获取SSDT表基址
    KeServiceDescriptorTable =
(PSYSTEM_SERVICE_TABLE)GetKeServiceDescriptorTable64(DriverObject);

    // 得到进程PID
    HANDLE processid = GetProcessID("x32.exe");
    DbgPrint("进程PID = %d \n", processid);

    // 附加执行注入
    AttachAndInjectProcess(processid, L"C:\\hook.dll");
    DriverObject->DriverUnload = Unload;
    return STATUS_SUCCESS;
}
```

运行如上这段代码片段，将编译好的DLL文件放入到`C: \\hook.dll`目录下，并运行`x32.exe`程序，手动加载驱动即可注入成功，输出效果图如下所示；



```
#    Time          Debug Print
17   7.05571222    Hello LyShark
18   7.06106091    索引 [ 0000000000000AF9 ] 函数名 [ _stricmp ] 相对RVA [ 000000000019D8D...
19   7.06234884    进程PID = 2452
20   7.06236649    [*] 注入32位DLL
21   7.06349659    线程函数地址: FFFFF8051B62D420 --> 开始执行地址: 00000000004B0000
```

回到应用层进程中，可看到我们的DLL已经被注入到目标进程内了，效果图如下所示；



hello lyshark

参考文献

Blackbone OpenSource