# 本章将探索内核级DLL模块注入实现原理，DLL模块注入在应用层中通常会使用 CreateRemoteThread 直接开启远程线程执行即可，驱动级别的注入有多种实现原理，而其中最简单的一种实现方式则是通过劫 持EIP的方式实现，其实现原理可总结为，挂起目标进程，停止目标进程EIP的变换，在目标进程开启空 间，并把相关的指令机器码和数据拷贝到里面去，然后直接修改目标进程EIP使其强行跳转到我们拷贝进 去的相关机器码位置，执行相关代码后，然后再次跳转回来执行原始指令集。

在内核模式中实现这一过程大体可分为如下步骤；

1.通过 PsLookupProcessByProcessId 将进程 PID 转为 EProcess 结构

2.通过 KeStackAttachProcess 附加到目标进程

3.通过 GetUserModule 得到当前进程中 Ntdll.dll 模块的基址

4.通过 GetModuleExport 得到 Ntdll.dll 模块内 LdrLoadDll 函数基址

5.通过 ZwGetNextThread 得到当前线程句柄

6.通过 PsSuspendThread 暂停当前线程运行

7.此时通过 GetWow64Code 生成特定的加载代码，并放入 ZwAllocateVirtualMemory 生成的内存中

8.修改当前EIP的值指向 newAddress 内存地址

7.通过 PsResumeThread 恢复线程执行，让其执行我们的 ShellCode 代码

8.最后调用 KeUnstackDetachProcess 脱离目标进程，并释放句柄

# 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，此部分内容摘录于微软官方文档，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](mailto:me@lyshark.com)

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

// 线程结构体偏移值

#define MAXCOUNTS 0x200

#define INITIALSTACKOFFSET 0x28 #define WOW64CONTEXTOFFSET 0x1488

#define WOW64\_SIZE\_OF\_80387\_REGISTERS 80

#define WOW64\_MAXIMUM\_SUPPORTED\_EXTENSION 512

// 导出函数

NTKERNELAPI PPEB NTAPI PsGetProcessPeb(IN PEPROCESS Process);

// 定义自定义函数指针

typedef PVOID(NTAPI PPsGetThreadTeb)(IN PETHREAD Thread);

typedef PVOID(NTAPI PPsGetProcessWow64Process)(\_In\_ PEPROCESS Process); typedef NTSTATUS(NTAPI PPsResumeThread)(PETHREAD Thread, OUT PULONG PreviousCount);

typedef NTSTATUS(NTAPI PPsSuspendThread)(IN PETHREAD Thread, OUT PULONG PreviousSuspendCount OPTIONAL);

typedef NTSTATUS(NTAPI PZwGetNextThread)(\_In\_ HANDLE ProcessHandle, \_In\_ HANDLE ThreadHandle, \_In\_ ACCESS\_MASK DesiredAccess, \_In\_ ULONG HandleAttributes, \_In\_ ULONG Flags, \_Out\_ PHANDLE NewThreadHandle);



// 存放全局函数指针的变量

PPsGetThreadTeb g\_PsGetThreadTeb = NULL; PPsResumeThread g\_PsResumeThread = NULL; PPsSuspendThread g\_PsSuspendThread = NULL; PZwGetNextThread g\_ZwGetNextThread = NULL;

PPsGetProcessWow64Process g\_PsGetProcessWow64Process = NULL;

// 定义微软结构体

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;

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 \_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 \_KLDR\_DATA\_TABLE\_ENTRY

{

LIST\_ENTRY InLoadOrderLinks; PVOID ExceptionTable;



ULONG ExceptionTableSize;

PVOID GpValue; ULONG UnKnow; PVOID DllBase; PVOID EntryPoint; ULONG SizeOfImage;

UNICODE\_STRING FullDllName; UNICODE\_STRING BaseDllName; ULONG Flags;

USHORT LoadCount; USHORT Unused5; PVOID SectionPointer; ULONG CheckSum; PVOID LoadedImports;

PVOID PatchInformation;

} KLDR\_DATA\_TABLE\_ENTRY, PKLDR\_DATA\_TABLE\_ENTRY;

typedef struct \_WOW64\_FLOATING\_SAVE\_AREA

{

DWORD ControlWord; DWORD StatusWord; DWORD TagWord; DWORD ErrorOffset;

DWORD ErrorSelector; DWORD DataOffset; DWORD DataSelector;

BYTE RegisterArea[WOW64\_SIZE\_OF\_80387\_REGISTERS];

DWORD Cr0NpxState;

} WOW64\_FLOATING\_SAVE\_AREA;

typedef struct \_WOW64\_CONTEXT

{

DWORD padding; DWORD ContextFlags; DWORD Dr0;

DWORD Dr1; DWORD Dr2; DWORD Dr3; DWORD Dr6; DWORD Dr7;

WOW64\_FLOATING\_SAVE\_AREA FloatSave; DWORD SegGs;

DWORD SegFs; DWORD SegEs; DWORD SegDs; DWORD Edi; DWORD Esi; DWORD Ebx; DWORD Edx; DWORD Ecx; DWORD Eax; DWORD Ebp; DWORD Eip; DWORD SegCs;



DWORD EFlags; DWORD Esp; DWORD SegSs;

BYTE ExtendedRegisters[WOW64\_MAXIMUM\_SUPPORTED\_EXTENSION];

} WOW64\_CONTEXT, PWOW64\_CONTEXT;

// 自定义注入结构体

typedef struct \_INJECT\_BUFFER

{

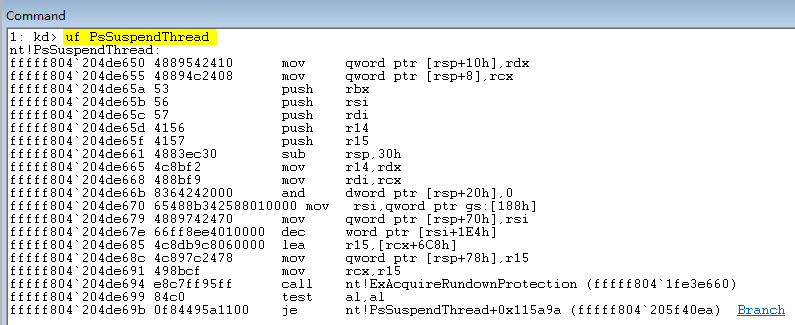
UCHAR Code[0x200]; UNICODE\_STRING Path; UNICODE\_STRING32 Path32;

wchar\_t Buffer[488]; PVOID ModuleHandle; ULONG Complete; NTSTATUS Status; ULONG64 orgRipAddress; ULONG64 orgRip;

} INJECT\_BUFFER, PINJECT\_BUFFER;

**SearchOPcode 特征码定位基址:** 在注入之前我们需要通过 SearchOPcode() 函数动态的寻找几个关键函数的基址，以 PsSuspendThread 函数的寻找为例，通过 WinDBG 我们可以定位到该函数，该函数模块在 ntoskrnl.exe 中，且无法直接通过 MmGetSystemRoutineAddress 拿到，为了能通过代码拿到该函数的入口地址，我提取 fffff804204de668 到 fffff804204de670 位置处的特征码，由于

fffff804204de668 距离 PsSuspendThread 函数开头只有 24 字节，所以直接通过 -24 即可得到。



通过调用 SearchOPcode() 并传入机器码即可直接拿到 PsSuspendThread 的入口地址，根据上述方式我们需要分别得到 PsSuspendThread ， PsResumeThread 这几个函数的内存基址，这些函数的具体作用如下所示；

# PsSuspendThread() 用于暂停或者挂起线程

PsResumeThread() 用于恢复线程

其次还需要通过 MmGetSystemRoutineAddress 函数动态的得到 ZwGetNextThread ，

PsGetThreadTeb ， PsGetProcessWow64Process 这几个函数的基址，这些函数的具体作用如下所示；

# ZwGetNextThread() 用于获取下一个活动线程

PsGetThreadTeb() 用于获取线程TEB结构

PsGetProcessWow64Process() 判断当前进程是否为32位

完整代码如下所示，运行这段代码将定位到我们所需的所有内核函数的基址信息；

// 署名权

// right to sign one's name on a piece of work

// PowerBy: LyShark

// Email: [me@lyshark.com](mailto:me@lyshark.com) #include "lyshark.h"

// 内核特征码定位函数封装

// 参数1：传入驱动句柄

// 参数2：传入驱动模块名

// 参数3：传入节表名称

// 参数4：传入待搜索机器码字节数组

// 参数5：传入机器码长度

// 参数6：基址修正字节数

PVOID SearchOPcode(PDRIVER\_OBJECT pObj, PWCHAR DriverName, PCHAR sectionName, PUCHAR opCode, DWORD len, DWORD offset)

{

PVOID dllBase = NULL; UNICODE\_STRING uniDriverName; PKLDR\_DATA\_TABLE\_ENTRY firstentry;

// 获取驱动入口

PKLDR\_DATA\_TABLE\_ENTRY entry = (PKLDR\_DATA\_TABLE\_ENTRY)pObj->DriverSection;

firstentry = entry; RtlInitUnicodeString(&uniDriverName, DriverName);

// 开始遍历

while ((PKLDR\_DATA\_TABLE\_ENTRY)entry->InLoadOrderLinks.Flink != firstentry)

{

if (entry->FullDllName.Buffer != 0 && entry->BaseDllName.Buffer != 0)

{

// 如果找到了所需模块则将其基地址返回

if (RtlCompareUnicodeString(&uniDriverName, &(entry->BaseDllName), FALSE) == 0)

{

dllBase = entry->DllBase; break;

}

}

entry = (PKLDR\_DATA\_TABLE\_ENTRY)entry->InLoadOrderLinks.Flink;

}

if (dllBase)

{

try

{

// 载入模块基地址

PIMAGE\_DOS\_HEADER ImageDosHeader = (PIMAGE\_DOS\_HEADER)dllBase; if (ImageDosHeader->e\_magic != IMAGE\_DOS\_SIGNATURE)

{

return NULL;

}

// 得到模块NT头以及Section节头

PIMAGE\_NT\_HEADERS64 pImageNtHeaders64 = (PIMAGE\_NT\_HEADERS64)

((PUCHAR)dllBase + ImageDosHeader->e\_lfanew);

PIMAGE\_SECTION\_HEADER pSectionHeader = (PIMAGE\_SECTION\_HEADER)

((PUCHAR)pImageNtHeaders64 + sizeof(pImageNtHeaders64->Signature) + sizeof(pImageNtHeaders64->FileHeader) + pImageNtHeaders64-

>FileHeader.SizeOfOptionalHeader);

PUCHAR endAddress = 0; PUCHAR starAddress = 0;

i++)

+ 1) == 0)

// 寻找符合条件的节

for (int i = 0; i < pImageNtHeaders64->FileHeader.NumberOfSections;

{

if (memcmp(sectionName, pSectionHeader->Name, strlen(sectionName)

{

starAddress = pSectionHeader->VirtualAddress +

(PUCHAR)dllBase;



endAddress = pSectionHeader->VirtualAddress + (PUCHAR)dllBase

+ pSectionHeader->SizeOfRawData;

break;

}

pSectionHeader++;

}

if (endAddress && starAddress)

{

// 找到会开始寻找特征

for (; starAddress < endAddress - len - 1; starAddress++)

{

// 验证访问权限

if (MmIsAddressValid(starAddress))

{

DWORD i = 0;

for (; i < len; i++)

{

// 判断是否为通配符' ' if (opCode[i] == 0x2a)

{

continue;

}

// 找到了一个字节则跳出

if (opCode[i] != starAddress[i])

{

break;

}

}

// 找到次数完全匹配则返回地址

if (i == len)

{

return starAddress + offset;

}

}

}



}

}

except (EXCEPTION\_EXECUTE\_HANDLER) {}

}

return NULL;

}

NTSTATUS UnDriver(PDRIVER\_OBJECT driver)

{

return STATUS\_SUCCESS;

}

NTSTATUS DriverEntry(IN PDRIVER\_OBJECT Driver, PUNICODE\_STRING RegistryPath)

{

DbgPrint("Hello LyShark.com \n");

/

0: kd> uf PsSuspendThread nt!PsSuspendThread:

fffff804`204de650 4889542410 mov qword ptr [rsp+10h],rdx fffff804`204de655 48894c2408 mov qword ptr [rsp+8],rcx fffff804`204de65a 53 push rbx

fffff804`204de65b 56 push rsi

fffff804`204de65c 57 push rdi

fffff804`204de65d 4156 push r14

fffff804`204de65f 4157 push r15 fffff804`204de661 4883ec30 sub rsp,30h fffff804`204de665 4c8bf2 mov r14,rdx

fffff804`204de668 488bf9 mov rdi,rcx fffff804`204de66b 8364242000 and dword ptr [rsp+20h],0 fffff804`204de670 65488b342588010000 mov rsi,qword ptr gs:[188h] fffff804`204de679 4889742470 mov qword ptr [rsp+70h],rsi fffff804`204de67e 66ff8ee4010000 dec word ptr [rsi+1E4h] fffff804`204de685 4c8db9c8060000 lea r15,[rcx+6C8h] fffff804`204de68c 4c897c2478 mov qword ptr [rsp+78h],r15 fffff804`204de691 498bcf mov rcx,r15

fffff804`204de694 e8c7ff95ff call nt!ExAcquireRundownProtection (fffff804`1fe3e660)

|  |  |  |  |
| --- | --- | --- | --- |
| fffff804`204de699 | 84c0 | test | al,al |
| fffff804`204de69b | 0f84495a1100 | je | nt!PsSuspendThread+0x115a9a |

(fffff804`205f40ea) Branch

/

UCHAR SuspendOpCode[] = { 0x48, 0x8b, 0xf9, 0x83, 0x64, 0x24, 0x20, 0x00, 0x65, 0x48, 0x8b, 0x34, 0x25, 0x88, 0x01 };

/

0: kd> uf PsResumeThread nt!PsResumeThread:

|  |  |  |  |
| --- | --- | --- | --- |
| fffff804`204c7ab0  fffff804`204c7ab5 | 48895c2408  4889742410 | mov  mov | qword ptr [rsp+8],rbx  qword ptr [rsp+10h],rsi |
| fffff804`204c7aba | 57 | push | rdi |
| fffff804`204c7abb | 4883ec20 | sub | rsp,20h |
| fffff804`204c7abf | 488bda | mov | rbx,rdx |
| fffff804`204c7ac2 | 488bf9 | mov | rdi,rcx |

fffff804`204c7ac5 e8ee4fa5ff call nt!KeResumeThread (fffff804`1ff1cab8)



fffff804`204c7aca 65488b142588010000 mov rdx,qword ptr gs:[188h]

fffff804`204c7ad3 8bf0 mov esi,eax

fffff804`204c7ad5 83f801 cmp eax,1 fffff804`204c7ad8 7521 jne nt!PsResumeThread+0x4b

(fffff804`204c7afb) Branch

/

UCHAR ResumeOpCode[] = { 0x48, 0x8b, 0xf9, 0xe8, 0xee, 0x4f, 0xa5, 0xff, 0x65, 0x48, 0x8b, 0x14, 0x25, 0x88 };

// 特征码检索PsSuspendThread函数基址

g\_PsSuspendThread = (PPsSuspendThread)SearchOPcode(Driver, L"ntoskrnl.exe", "PAGE", SuspendOpCode, sizeof(SuspendOpCode), -24);

DbgPrint("PsSuspendThread = %p \n", g\_PsSuspendThread);

// 特征码检索PsResumeThread基址

g\_PsResumeThread = (PPsResumeThread)SearchOPcode(Driver, L"ntoskrnl.exe", "PAGE", ResumeOpCode, sizeof(ResumeOpCode), -18);

DbgPrint("PsResumeThread = %p \n", g\_PsResumeThread);

// 动态获取内存中的ZwGetNextThread基址

UNICODE\_STRING ZwGetNextThreadString = RTL\_CONSTANT\_STRING(L"ZwGetNextThread");

g\_ZwGetNextThread =

(PZwGetNextThread)MmGetSystemRoutineAddress(&ZwGetNextThreadString); DbgPrint("ZwGetNextThread = %p \n", g\_ZwGetNextThread);

// 动态获取内存中的PsGetThreadTeb基址

UNICODE\_STRING PsGetThreadTebString = RTL\_CONSTANT\_STRING(L"PsGetThreadTeb"); g\_PsGetThreadTeb =

(PPsGetThreadTeb)MmGetSystemRoutineAddress(&PsGetThreadTebString);

DbgPrint("PsGetThreadTeb = %p \n", g\_PsGetThreadTeb);

// 动态获取内存中的PsGetProcessWow64Process基址

UNICODE\_STRING PsGetProcessWow64ProcessString = RTL\_CONSTANT\_STRING(L"PsGetProcessWow64Process");

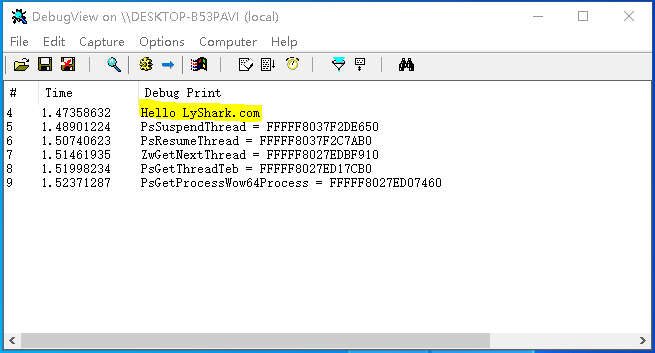
g\_PsGetProcessWow64Process = (PPsGetProcessWow64Process)MmGetSystemRoutineAddress(&PsGetProcessWow64ProcessStr ing);

DbgPrint("PsGetProcessWow64Process = %p \n", g\_PsGetProcessWow64Process);

Driver->DriverUnload = UnDriver; return STATUS\_SUCCESS;

}

# 编译并运行如上代码片段，则会输出我们所需函数的入口地址，输出效果图如下所示；



**GetUserModule 获取模块基址:** 此函数的功能是获取到当前内核下特定模块的基址，函数接收三个参数，在入口 DriverEntry 位置通过 KeStackAttachProcess 附加到进程空间内，如果是32位进程则通过 PsGetProcessWow64Process 得到进程的PEB结构，如果是64位则通过 PsGetProcessPeb 得到PEB进程环境块的目的是为了解析 PLIST\_ENTRY32 链表，通过 RtlCompareUnicodeString 对比模块是否符合要求，如果符合则在此链表中取出 LdrDataTableEntry32->DllBase 模块基址并返回给调用者，其完整代码片段如下所示；

1.通过 KeStackAttachProcess 附加到用户层进程空间内

# 2.通过各种函数获取到进程 PEB 进程环境块

3.遍历 PLIST\_ENTRY32 链表，判断 ModuleName 是否所需

4.获取 LdrDataTableEntry32->DllBase 中的模块基址

// 署名权

// right to sign one's name on a piece of work

// PowerBy: LyShark

// Email: [me@lyshark.com](mailto:me@lyshark.com)

#include "lyshark.h"

// 得到当前用户进程下的模块基址

// 参数1：传入用户EProcess结构

// 参数2：传入模块名

// 参数3：是否32位

PVOID GetUserModule(IN PEPROCESS EProcess, IN PUNICODE\_STRING ModuleName, IN BOOLEAN IsWow64)

{

if (EProcess == NULL) return NULL;

try

{

// 执行32位

if (IsWow64)

{

// 获取32位下的PEB进程环境块

PPEB32 Peb32 = (PPEB32)g\_PsGetProcessWow64Process(EProcess);

if (Peb32 == NULL)

return NULL;

if (!Peb32->Ldr) return NULL;

// 循环遍历链表 寻找模块

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);

// 初始化模块名

RtlUnicodeStringInit(&UnicodeString, (PWCH)LdrDataTableEntry32-

>BaseDllName.Buffer);

// 对比模块名是否符合

if (RtlCompareUnicodeString(&UnicodeString, ModuleName, TRUE) ==

0)

return (PVOID)LdrDataTableEntry32->DllBase;

}

}

// 执行64位

else

{

// 得到64位PEB进程环境块

PPEB Peb = PsGetProcessPeb(EProcess); if (!Peb)

return NULL;

if (!Peb->Ldr) return NULL;

// 开始遍历模块

for (PLIST\_ENTRY ListEntry = Peb->Ldr->InLoadOrderModuleList.Flink; ListEntry != &Peb->Ldr->InLoadOrderModuleList;

ListEntry = ListEntry->Flink)

{

// 得到表头

PLDR\_DATA\_TABLE\_ENTRY LdrDataTableEntry =

CONTAINING\_RECORD(ListEntry, LDR\_DATA\_TABLE\_ENTRY, InLoadOrderLinks);

// 判断是否是所需要的模块

if (RtlCompareUnicodeString(&LdrDataTableEntry->BaseDllName, ModuleName, TRUE) == 0)

return LdrDataTableEntry->DllBase;

}

}

}

except (EXCEPTION\_EXECUTE\_HANDLER){}

return NULL;



}

NTSTATUS UnDriver(PDRIVER\_OBJECT driver)

{

return STATUS\_SUCCESS;

}

NTSTATUS DriverEntry(IN PDRIVER\_OBJECT Driver, PUNICODE\_STRING RegistryPath)

{

DbgPrint("Hello LyShark.com \n");

// 动态获取内存中的PsGetProcessWow64Process基址

UNICODE\_STRING PsGetProcessWow64ProcessString = RTL\_CONSTANT\_STRING(L"PsGetProcessWow64Process");

g\_PsGetProcessWow64Process = (PPsGetProcessWow64Process)MmGetSystemRoutineAddress(&PsGetProcessWow64ProcessStr ing);

DbgPrint("PsGetProcessWow64Process = %p \n", g\_PsGetProcessWow64Process);

PEPROCESS pEprocess = NULL; DWORD pid = 6084;

// 根据PID得到进程Eprocess结构

if (NT\_SUCCESS(PsLookupProcessByProcessId((HANDLE)pid, &pEprocess)))

{

// 初始化结构

UNICODE\_STRING ntdllString = RTL\_CONSTANT\_STRING(L"Ntdll.dll");

KAPC\_STATE kApc = { 0 };

// 附加到进程内

KeStackAttachProcess(pEprocess, &kApc);

// 获取NTDLL的模块基地址

PVOID ntdll\_address = GetUserModule(pEprocess, &ntdllString, TRUE); if (ntdll\_address != NULL)

{

DbgPrint("[ ] Ntdll Addr = %p \n", ntdll\_address);

}

// 取消附加

KeUnstackDetachProcess(&kApc);

// 递减计数

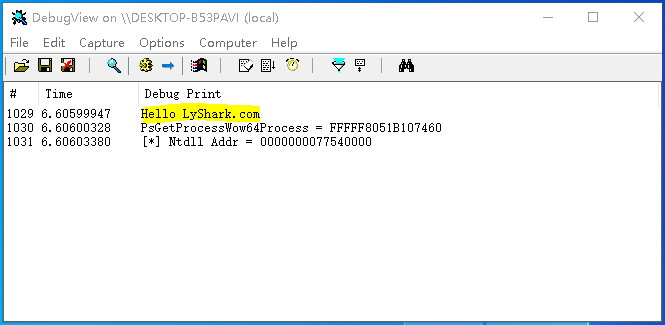
ObDereferenceObject(pEprocess);

}

Driver->DriverUnload = UnDriver; return STATUS\_SUCCESS;

}

运行如上这段程序，则会取出进程ID为 6084 中 Ntdll.dll 的模块基址，输出效果图如下所示；



**GetModuleExport 取导出表函数基址:** 此函数的功能是获取到当前内核下特定模块中的特定函数（内存中）基址，函数接收两个参数，在入口 DriverEntry 位置通过 KeStackAttachProcess 附加到进程空间内，通过解析 IMAGE\_DIRECTORY\_ENTRY\_EXPORT 导出表取出导出函数名，此处需要注意如果函数名指针小于等于 0xFFFF 则说明是序号导出，如果大于 0xFFFF 则说明是名字导出，判断名字是否一致，如果一致则返回当前内存的 ModuleBase 模块基址加上 pAddressOfFuncs[OrdIndex] 相对偏移，从而获取到内存中的绝对地址，完整代码片段如下所示；

// 署名权

// right to sign one's name on a piece of work

// PowerBy: LyShark

// Email: [me@lyshark.com](mailto:me@lyshark.com)

#include "lyshark.h"

// 根据函数名得到导出表地址

// 参数1：传入模块入口地址

// 参数2：传入导出函数名

PVOID GetModuleExport(IN PVOID ModuleBase, IN PCCHAR FunctionName)

{

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;

try

{

// 判断是否是DOS头

if (ImageDosHeader->e\_magic != IMAGE\_DOS\_SIGNATURE)

{

return NULL;

}

// 获取PE结构节NT头

ImageNtHeaders32 = (PIMAGE\_NT\_HEADERS32)((PUCHAR)ModuleBase + ImageDosHeader->e\_lfanew);

ImageNtHeaders64 = (PIMAGE\_NT\_HEADERS64)((PUCHAR)ModuleBase +

ImageDosHeader->e\_lfanew);

// 判断是否是64位

if (ImageNtHeaders64->OptionalHeader.Magic == IMAGE\_NT\_OPTIONAL\_HDR64\_MAGIC)

{

// 如果是64位则执行如下

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

{

// 如果32位则执行如下

ImageExportDirectory = (PIMAGE\_EXPORT\_DIRECTORY)(ImageNtHeaders32-

>OptionalHeader.DataDirectory[IMAGE\_DIRECTORY\_ENTRY\_EXPORT].VirtualAddress + (ULONG\_PTR)ModuleBase);

ExportDirectorySize = ImageNtHeaders32-

>OptionalHeader.DataDirectory[IMAGE\_DIRECTORY\_ENTRY\_EXPORT].Size;

}

// 取出导出表Index，名字，函地址等

PUSHORT pAddressOfOrds = (PUSHORT)(ImageExportDirectory-

>AddressOfNameOrdinals + (ULONG\_PTR)ModuleBase);

PULONG pAddressOfNames = (PULONG)(ImageExportDirectory->AddressOfNames + (ULONG\_PTR)ModuleBase);

PULONG pAddressOfFuncs = (PULONG)(ImageExportDirectory-

>AddressOfFunctions + (ULONG\_PTR)ModuleBase);

// 循环导出表

for (ULONG i = 0; i < ImageExportDirectory->NumberOfFunctions; ++i)

{

USHORT OrdIndex = 0xFFFF; PCHAR pName = NULL;

// 说明是序号导出

if ((ULONG\_PTR)FunctionName <= 0xFFFF)

{

// 得到函数序号

OrdIndex = (USHORT)i;

}

// 说明是名字导出

else if ((ULONG\_PTR)FunctionName > 0xFFFF && i < ImageExportDirectory->NumberOfNames)

{

// 得到函数名

pName = (PCHAR)(pAddressOfNames[i] + (ULONG\_PTR)ModuleBase); OrdIndex = pAddressOfOrds[i];

}

else

return NULL;

// 判断函数名是否符合

if (((ULONG\_PTR)FunctionName <= 0xFFFF && (USHORT) ((ULONG\_PTR)FunctionName) == OrdIndex + ImageExportDirectory->Base) ||

((ULONG\_PTR)FunctionName > 0xFFFF && strcmp(pName, FunctionName)

== 0))

{

// 得到完整地址

FunctionAddress = pAddressOfFuncs[OrdIndex] +

(ULONG\_PTR)ModuleBase;



break;

}

}

}

except (EXCEPTION\_EXECUTE\_HANDLER){}

return (PVOID)FunctionAddress;

}

NTSTATUS UnDriver(PDRIVER\_OBJECT driver)

{

return STATUS\_SUCCESS;

}

NTSTATUS DriverEntry(IN PDRIVER\_OBJECT Driver, PUNICODE\_STRING RegistryPath)

{

DbgPrint("Hello LyShark.com \n");

PEPROCESS pEprocess = NULL; DWORD pid = 6084;

// 根据PID得到进程Eprocess结构

if (NT\_SUCCESS(PsLookupProcessByProcessId((HANDLE)pid, &pEprocess)))

{

KAPC\_STATE kApc = { 0 };

// ntdll.dll模块基址

PVOID ntdll\_address = (PVOID)0x0000000077540000;

// 附加到进程内

KeStackAttachProcess(pEprocess, &kApc);

// 取模块中LdrLoadDll函数基址

PVOID LdrLoadDllAddress = GetModuleExport(ntdll\_address, "LdrLoadDll");

DbgPrint("[ ] LdrLoadDllAddress = %p \n", LdrLoadDllAddress);

// 取消附加

KeUnstackDetachProcess(&kApc);

// 递减计数

ObDereferenceObject(pEprocess);

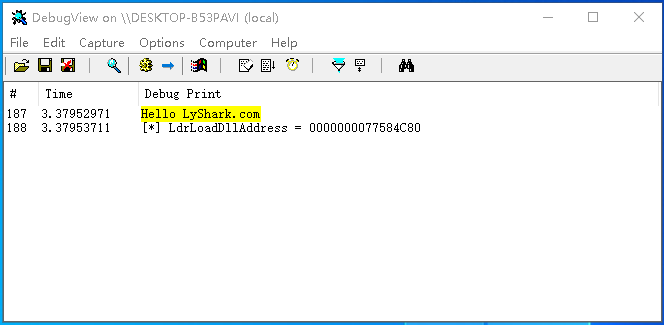
}

Driver->DriverUnload = UnDriver;

return STATUS\_SUCCESS;

}

编译并运行如上代码片段，即可获取到进程 6084 号， ntdll.dll 模块中 LdrLoadDll 的内存地址，其输出效果图如下所示；



**GetCurrentContext 获取当前线程上下文:** 此函数的功能是获取附加进程内当前线程的上下文地址，函数接收一个参数，内部通过 PsLookupProcessByProcessId 得到进程 EProcess 结构体，通过

KeStackAttachProcess 附加到进程内，调用 g\_ZwGetNextThread 获取当当前线程上下文，函数

ObReferenceObjectByHandle 用于将 Handle 转换为线程对象，之后再通过 g\_PsSuspendThread 暂停线程后，即可通过各类函数获取到该线程的绝大部分信息，最终在调用结束时记得调用

g\_PsResumeThread 恢复线程的运行，并 KeUnstackDetachProcess 脱离附加，解析上下文环境完整代码如下所示；



// 署名权

// right to sign one's name on a piece of work

// PowerBy: LyShark

// Email: [me@lyshark.com](mailto:me@lyshark.com)

#include "lyshark.h"

// ShellCode 注入线程函数

NTSTATUS GetCurrentContext(ULONG pid, PVOID allcateAddress)

{

PEPROCESS pEprocess = NULL;

// 根据PID得到进程Eprocess结构

if (NT\_SUCCESS(PsLookupProcessByProcessId((HANDLE)pid, &pEprocess)))

{

KAPC\_STATE kApc = { 0 };

// 附加到进程内

KeStackAttachProcess(pEprocess, &kApc);

HANDLE threadHandle = NULL;

// 得到当前正在运行的线程上下文

if (NT\_SUCCESS(g\_ZwGetNextThread((HANDLE)-1, (HANDLE)0, 0x1FFFFF, 0x240,



0, &threadHandle)))

{

PVOID threadObj = NULL;

// 在对象句柄上提供访问验证，如果可以授予访问权限，则返回指向对象的正文的相应指针。

NTSTATUS state = ObReferenceObjectByHandle(threadHandle, 0x1FFFFF, PsThreadType, KernelMode, &threadObj, NULL);

if (NT\_SUCCESS(state))

{

// 暂停线程

g\_PsSuspendThread(threadObj, NULL);

try

{

// 得到TEB

PVOID pTeb = g\_PsGetThreadTeb(threadObj); if (pTeb)

{

DbgPrint("[+] 线程环境块TEB = %p \n", pTeb);

// 得到当前线程上下文

/ WOW64CONTEXTOFFSET = TlsSlots + 8

0: kd> dt \_TEB nt!\_TEB

+ 0x000 NtTib : \_NT\_TIB

+ 0x1258 StaticUnicodeString : \_UNICODE\_STRING

+ 0x1268 StaticUnicodeBuffer : [261] Wchar

+ 0x1472 Padding3 : [6] UChar

+ 0x1478 DeallocationStack : Ptr64 Void

+ 0x1480 TlsSlots : [64] Ptr64 Void

+ 0x1680 TlsLinks : \_LIST\_ENTRY

+ 0x1690 Vdm : Ptr64 Void

+ 0x1698 ReservedForNtRpc : Ptr64 Void

+ 0x16a0 DbgSsReserved : [2] Ptr64 Void

/

PWOW64\_CONTEXT pCurrentContext = (PWOW64\_CONTEXT)( (ULONG64 )((ULONG64)pTeb + WOW64CONTEXTOFFSET));

DbgPrint("[-] 当前上下文EIP = %p \n", pCurrentContext-

>Eip);

// 检查上下文是否可读

ProbeForRead((PVOID)pCurrentContext, sizeof(pCurrentContext), sizeof(CHAR));

UCHAR Code[] = {

0xb8, 0x0, 0x0, 0x0, 0x0, // mov eax, orgEip 0x58, // pop eax

0xc3 // ret

};

// 将ShellCode拷贝到InjectBuffer中等待处理RtlCopyMemory(allcateAddress, Code, sizeof(Code)); DbgPrint("[ ] 拷贝 [%p] 内存 \n", allcateAddress);;

>Eip;

>Eip);

// 修改代码模板，将指定位置替换为我们自己的代码

(ULONG )((PUCHAR)allcateAddress + 1) = pCurrentContext-

DbgPrint("[ ] 替换 [ %p ] 跳转地址 \n", pCurrentContext-

>Eip);

// 执行线程

pCurrentContext->Eip = (ULONG)(ULONG64)(allcateAddress); DbgPrint("[ ] 执行 [ %p ] 线程函数 \n", pCurrentContext-

}

}

except (EXCEPTION\_EXECUTE\_HANDLER) {}



// 恢复线程g\_PsResumeThread(threadObj, NULL); ObDereferenceObject(threadObj);

}

NtClose(threadHandle);

}

// 关闭线程KeUnstackDetachProcess(&kApc); ObDereferenceObject(pEprocess);

}

return STATUS\_SUCCESS;

}

NTSTATUS UnDriver(PDRIVER\_OBJECT driver)

{

UNREFERENCED\_PARAMETER(driver);

return STATUS\_SUCCESS;

}

NTSTATUS DriverEntry(IN PDRIVER\_OBJECT Driver, PUNICODE\_STRING RegistryPath)

{

DbgPrint("Hello LyShark.com \n");

UNREFERENCED\_PARAMETER(RegistryPath);

// 初始化基址

InitAddress(Driver);

ULONG ProcessID = 4904; PVOID AllcateAddress = NULL; DWORD create\_size = 1024;

// 申请堆 《内核远程堆分配与销毁》核心代码

NTSTATUS Status = AllocMemory(ProcessID, create\_size, &AllcateAddress);

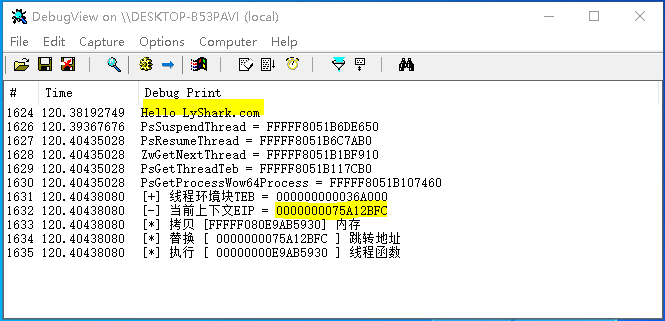
// 执行ShellCode线程注入

Status = GetCurrentContext(ProcessID, &AllcateAddress);

Driver->DriverUnload = UnDriver; return STATUS\_SUCCESS;

}

# 运行如上代码片段，则将输出进程 ID=4904 的当前进程内，线程上下文RIP地址，输出效果如下图所示；



**KernelInjectDLL 驱动注入:** 如上代码中我们已经找到了驱动注入时所需用到的关键函数，那么实现代码就变得很容易了，驱动注入的实现方式有很多种，不论哪一种其实现的难度并不在于代码本身，而在于 某些结构如何正确的被找到，一旦结构被找到原理方面的代码可以说非常容易获取到，如下这段完整代 码则是驱动注入的一个简化版，如果你觉得不方便完全可以自行添加IOCTL控制器让其更易于使用，此处 为了节约篇幅不在增加冗余代码，代码已做具体分析和备注。

此注入驱动核心实现代码如下所示，其中 SearchOPcode 用于在内核模块中寻找符合条件的内存地址， GetNativeCode 则用于生成一段可被调用的 ShellCode 代码，此代码执行的目的就是将DLL动态装载到对端内存中， SetThreadStartAddress 则用于填充执行线程结构信息， GetUserModule 用户获取进程内特定模块的基址， GetModuleExport 用于在模块内寻找特定函数的基址， KernelInjectDLL 则是最终注入函数，其首先将线程暂停，并注入生成的 ShellCode ，然后恢复线程让 ShellCode 跑起来，当ShellCode 跑起来后将会自动的将特定目录下的DLL拉起来，以此来实现动态加载的目的。

// 署名权

// right to sign one's name on a piece of work

// PowerBy: LyShark

// Email: [me@lyshark.com](mailto:me@lyshark.com)

#include "lyshark.h"

// 内核特征码定位函数封装

PVOID SearchOPcode(PDRIVER\_OBJECT pObj, PWCHAR DriverName, PCHAR sectionName, PUCHAR opCode, int len, int offset)

{

PVOID dllBase = NULL; UNICODE\_STRING uniDriverName; PKLDR\_DATA\_TABLE\_ENTRY firstentry;

// 获取驱动入口

PKLDR\_DATA\_TABLE\_ENTRY entry = (PKLDR\_DATA\_TABLE\_ENTRY)pObj->DriverSection;

firstentry = entry; RtlInitUnicodeString(&uniDriverName, DriverName);

// 开始遍历

while ((PKLDR\_DATA\_TABLE\_ENTRY)entry->InLoadOrderLinks.Flink != firstentry)

{

// 如果找到了所需模块则将其基地址返回

if (entry->FullDllName.Buffer != 0 && entry->BaseDllName.Buffer != 0)

{

if (RtlCompareUnicodeString(&uniDriverName, &(entry->BaseDllName), FALSE) == 0)

{

dllBase = entry->DllBase; break;

}

}

entry = (PKLDR\_DATA\_TABLE\_ENTRY)entry->InLoadOrderLinks.Flink;

}

if (dllBase)

{

try

{

// 载入模块基地址

PIMAGE\_DOS\_HEADER ImageDosHeader = (PIMAGE\_DOS\_HEADER)dllBase; if (ImageDosHeader->e\_magic != IMAGE\_DOS\_SIGNATURE)

{

return NULL;

}

// 得到模块NT头

PIMAGE\_NT\_HEADERS64 pImageNtHeaders64 = (PIMAGE\_NT\_HEADERS64)

((PUCHAR)dllBase + ImageDosHeader->e\_lfanew);

// 获取节表头

PIMAGE\_SECTION\_HEADER pSectionHeader = (PIMAGE\_SECTION\_HEADER)

((PUCHAR)pImageNtHeaders64 + sizeof(pImageNtHeaders64->Signature) + sizeof(pImageNtHeaders64->FileHeader) + pImageNtHeaders64-

>FileHeader.SizeOfOptionalHeader);

PUCHAR endAddress = 0; PUCHAR starAddress = 0;

i++)

+ 1) == 0)

// 寻找符合条件的节

for (int i = 0; i < pImageNtHeaders64->FileHeader.NumberOfSections;

{

// 寻找符合条件的表名

if (memcmp(sectionName, pSectionHeader->Name, strlen(sectionName)

{

// 取出开始和结束地址

starAddress = pSectionHeader->VirtualAddress +

(PUCHAR)dllBase;

endAddress = pSectionHeader->VirtualAddress + (PUCHAR)dllBase

+ pSectionHeader->SizeOfRawData;

break;

}

// 遍历下一个节

pSectionHeader++;

}

if (endAddress && starAddress)



{

// 找到会开始寻找特征

for (; starAddress < endAddress - len - 1; starAddress++)

{

// 验证访问权限

if (MmIsAddressValid(starAddress))

{

int i = 0;

for (; i < len; i++)

{

// 判断是否为通配符' ' if (opCode[i] == 0x2a)

continue;

// 找到了一个字节则跳出

if (opCode[i] != starAddress[i]) break;

}

// 找到次数完全匹配则返回地址

if (i == len)

{

return starAddress + offset;

}

}

}

}

}

except (EXCEPTION\_EXECUTE\_HANDLER) {}

}

return NULL;

}

// 生成64位注入代码

PINJECT\_BUFFER GetNativeCode(PVOID LdrLoadDll, PUNICODE\_STRING DllFullPath, ULONGLONG orgEip)

{

SIZE\_T Size = PAGE\_SIZE;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| PINJECT\_BUFFER InjectBuffer | = | NULL; |  | | |
| UCHAR Code[] = {  0x41, 0x57, |  |  | // | push | r15 |
| 0x41, 0x56, |  |  | // | push | r14 |
| 0x41, 0x55, |  |  | // | push | r13 |
| 0x41, 0x54,  0x41, 0x53, |  |  | //  // | push  push | r12  r11 |
| 0x41, 0x52, |  |  | // | push | r10 |
| 0x41, 0x51, |  |  | // | push | r9 |
| 0x41, 0x50,  0x50, |  |  | //  // | push  push | r8  rax |
| 0x51, |  |  | // | push | rcx |
| 0x53,  0x52, |  |  | //  // | push  push | rbx  rdx |
| 0x55, |  |  | // | push | rbp |
| 0x54, |  |  | // | push | rsp |
| 0x56, |  |  | // | push | rsi |

0x57, // push rdi

0x66, 0x9C, // pushf

0x48, 0x83, 0xEC, 0x26, // 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 +38

0x49, 0xB9, 0, 0, 0, 0, 0, 0, 0, 0, // mov r9, ModuleHandle

offset +48

0x48, 0xB8, 0, 0, 0, 0, 0, 0, 0, 0, // mov rax, LdrLoadDll

offset +58

0xFF, 0xD0, // call rax

0x48, 0xBA, 0, 0, 0, 0, 0, 0, 0, 0, // mov rdx, COMPLETE\_OFFSET

offset +70

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 +86

0x89, 0x02, // mov [rdx], eax

0x48, 0x83, 0xC4, 0x26, // add rsp, 0x28

0x66, 0x9D, // popf

0x5F, // pop rdi

0x5E, // pop rsi

0x5C, // pop rsp

0x5D, // pop rbp

0x5A, // pop rdx

0x5B, // pop rbx

0x59, // pop rcx

0x58, // pop rax

0x41, 0x58, // pop r8

0x41, 0x59, // pop r9

0x41, 0x5A, // pop r10

0x41, 0x5B, // pop r11

0x41, 0x5C, // pop r12

0x41, 0x5D, // pop r13

0x41, 0x5E, // pop r14

0x41, 0x5F, // pop r15

0x50, // push rax

0x50, // push rax

0x48, 0xB8, 0, 0, 0, 0, 0, 0, 0, 0, // mov rax, orgEip offset +130 0x48, 0x89, 0x44, 0x24, 0x08, // mov [rsp+8],rax

0x58, // pop rax

0xC3 // ret

};

// 在当前进程内分配内存空间

if (NT\_SUCCESS(ZwAllocateVirtualMemory(ZwCurrentProcess(), &InjectBuffer, 0, &Size, MEM\_COMMIT, PAGE\_EXECUTE\_READWRITE)))

{

// 初始化路径变量与长度参数

PUNICODE\_STRING UserPath = &InjectBuffer->Path; UserPath->Length = DllFullPath->Length;

UserPath->MaximumLength = DllFullPath->MaximumLength; UserPath->Buffer = InjectBuffer->Buffer;

RtlUnicodeStringCopy(UserPath, DllFullPath);

// 将ShellCode拷贝到InjectBuffer中等待处理



memcpy(InjectBuffer, Code, sizeof(Code));

// 修改代码模板，将指定位置替换为我们自己的代码

(ULONGLONG )((PUCHAR)InjectBuffer + 38) = (ULONGLONG)UserPath;

(ULONGLONG )((PUCHAR)InjectBuffer + 48) = (ULONGLONG)& InjectBuffer-

>ModuleHandle;

(ULONGLONG )((PUCHAR)InjectBuffer + 58) = (ULONGLONG)LdrLoadDll;

(ULONGLONG )((PUCHAR)InjectBuffer + 70) = (ULONGLONG)& InjectBuffer-

>Complete;

(ULONGLONG )((PUCHAR)InjectBuffer + 86) = (ULONGLONG)& InjectBuffer-

>Status;

(ULONGLONG )((PUCHAR)InjectBuffer + 130) = orgEip;

return InjectBuffer;

}

return NULL;

}

// 生成32位注入代码

PINJECT\_BUFFER GetWow64Code(PVOID LdrLoadDll, PUNICODE\_STRING DllFullPath, ULONG orgEip)

{

SIZE\_T Size = PAGE\_SIZE;

PINJECT\_BUFFER InjectBuffer = NULL;

UCHAR Code[] = {

0x60, // pushad

0x9c, // pushfd

0x68, 0, 0, 0, 0, // push ModuleHandle

offset +3

0x68, 0, 0, 0, 0, // push ModuleFileName

offset +8

0x6A, 0, // push Flags

0x6A, 0, // push PathToFile

0xE8, 0, 0, 0, 0, // call LdrLoadDll

offset +17

0xBA, 0, 0, 0, 0, // mov edx, COMPLETE\_OFFSET

offset +22

0xC7, 0x02, 0x7E, 0x1E, 0x37, 0xC0, // mov [edx], CALL\_COMPLETE

0xBA, 0, 0, 0, 0, // mov edx, STATUS\_OFFSET

offset +33

0x89, 0x02, // mov [edx], eax

0x9d, // popfd

0x61, // popad

0x50, // push eax

0x50, // push eax

0xb8, 0, 0, 0, 0, // mov eax, orgEip

0x89, 0x44, 0x24, 0x04, // mov [esp+4],eax

0x58, // pop eax

0xc3 // ret

};

/

如下代码中通过定义Code并写入调用模块加载的汇编指令集，通过运用ZwAllocateVirtualMemory在 当前进程也就是附加到对端以后的进程内动态开辟了一块长度为Size的内存空间并赋予了



PAGE\_EXECUTE\_READWRITE读写执行属性，

由于Code代码无法直接使用，则此处调用RtlCopyMemory将指令拷贝到了InjectBuffer其目的是用于 后续的填充工作，最后通过 (ULONG )((PUCHAR)InjectBuffer + 3)的方式将需要使用的函数地址，

模块信息等依次填充到汇编代码的指定位置，并返回InjectBuffer指针。

/

// 在当前进程内分配内存空间

if (NT\_SUCCESS(ZwAllocateVirtualMemory(ZwCurrentProcess(), &InjectBuffer, 0, &Size, MEM\_COMMIT, PAGE\_EXECUTE\_READWRITE)))

{

// 初始化路径变量与长度参数

PUNICODE\_STRING32 pUserPath = &InjectBuffer->Path32; pUserPath->Length = DllFullPath->Length;

pUserPath->MaximumLength = DllFullPath->MaximumLength; pUserPath->Buffer = (ULONG)(ULONG\_PTR)InjectBuffer->Buffer;

// 将ShellCode拷贝到InjectBuffer中等待处理

memcpy((PVOID)pUserPath->Buffer, DllFullPath->Buffer, DllFullPath-

>Length);

memcpy(InjectBuffer, Code, sizeof(Code));

// 修改代码模板，将指定位置替换为我们自己的代码

(ULONG )((PUCHAR)InjectBuffer + 3) = (ULONG)(ULONG\_PTR)& InjectBuffer-

>ModuleHandle;

(ULONG )((PUCHAR)InjectBuffer + 8) = (ULONG)(ULONG\_PTR)pUserPath; (ULONG )((PUCHAR)InjectBuffer + 17) = (ULONG)((ULONG\_PTR)LdrLoadDll -

((ULONG\_PTR)InjectBuffer + 17) - 5 + 1);

(ULONG )((PUCHAR)InjectBuffer + 22) = (ULONG)(ULONG\_PTR)& InjectBuffer-

>Complete;

(ULONG )((PUCHAR)InjectBuffer + 33) = (ULONG)(ULONG\_PTR)& InjectBuffer-

>Status;

(ULONG )((PUCHAR)InjectBuffer + 44) = orgEip;

return InjectBuffer;

}

return NULL;

}

// 设置线程执行地址

NTSTATUS SetThreadStartAddress(PETHREAD pEthread, BOOLEAN isWow64, PVOID LdrLoadDll, PUNICODE\_STRING DllFullPath, PINJECT\_BUFFER allcateAddress)

{

try

{

// 判断是32位则执行

if (isWow64)

{

// 得到线程TEB

PVOID pTeb = g\_PsGetThreadTeb(pEthread); if (pTeb)

{

// 得到当前线程上下文

PWOW64\_CONTEXT pCurrentContext = (PWOW64\_CONTEXT)( (ULONG64 ) ((ULONG64)pTeb + WOW64CONTEXTOFFSET));

sizeof(CHAR));

// 检查上下文是否可读

ProbeForRead((PVOID)pCurrentContext, sizeof(pCurrentContext),

// 生成注入代码

PINJECT\_BUFFER newAddress = GetWow64Code(LdrLoadDll, DllFullPath, pCurrentContext->Eip);

if (newAddress)

{

>Eip);

// 替换上下文地址到内存中

newAddress->orgRipAddress = (ULONG64)& (pCurrentContext-

newAddress->orgRip = pCurrentContext->Eip; allcateAddress = newAddress;

pCurrentContext->Eip = (ULONG)(ULONG64)(newAddress);

}

return STATUS\_SUCCESS;



}

}

// 执行64位代码

else

{

// 验证地址是否可读取

if (MmIsAddressValid((PVOID) (ULONG64 )((ULONG64)pEthread + INITIALSTACKOFFSET)))

{

// 当前TID

PKTRAP\_FRAME pCurrentTrap = (PKTRAP\_FRAME)( (ULONG64 ) ((ULONG64)pEthread + INITIALSTACKOFFSET) - sizeof(KTRAP\_FRAME));

// 生成注入代码

PINJECT\_BUFFER newAddress = GetNativeCode(LdrLoadDll, DllFullPath, pCurrentTrap->Rip);

if (newAddress)

{

// 替换当前RIP地址

newAddress->orgRipAddress = (ULONG64)& (pCurrentTrap->Rip); newAddress->orgRip = pCurrentTrap->Rip;

allcateAddress = newAddress; pCurrentTrap->Rip = (ULONG64)newAddress;

}

}

return STATUS\_SUCCESS;

}

}

except (EXCEPTION\_EXECUTE\_HANDLER) {}

return STATUS\_UNSUCCESSFUL;

}

// 得到当前用户进程下的模块基址

PVOID GetUserModule(IN PEPROCESS EProcess, IN PUNICODE\_STRING ModuleName, IN BOOLEAN IsWow64)

{

if (EProcess == NULL)

return NULL;

try

{

// 执行32位

if (IsWow64)

{

// 获取32位下的PEB进程环境块

PPEB32 Peb32 = (PPEB32)g\_PsGetProcessWow64Process(EProcess); if (Peb32 == NULL)

return NULL;

if (!Peb32->Ldr) return NULL;

// 循环遍历链表 寻找模块

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);

// 初始化模块名

RtlUnicodeStringInit(&UnicodeString, (PWCH)LdrDataTableEntry32-

>BaseDllName.Buffer);

// 对比模块名是否符合

if (RtlCompareUnicodeString(&UnicodeString, ModuleName, TRUE) ==

0)

return (PVOID)LdrDataTableEntry32->DllBase;

}

}

// 执行64位

else

{

// 得到64位PEB进程环境块

PPEB Peb = PsGetProcessPeb(EProcess); if (!Peb)

return NULL;

if (!Peb->Ldr) return NULL;

// 开始遍历模块

for (PLIST\_ENTRY ListEntry = Peb->Ldr->InLoadOrderModuleList.Flink; ListEntry != &Peb->Ldr->InLoadOrderModuleList;

ListEntry = ListEntry->Flink)

{

// 得到表头

PLDR\_DATA\_TABLE\_ENTRY LdrDataTableEntry =

CONTAINING\_RECORD(ListEntry, LDR\_DATA\_TABLE\_ENTRY, InLoadOrderLinks);

// 判断是否是所需要的模块

if (RtlCompareUnicodeString(&LdrDataTableEntry->BaseDllName, ModuleName, TRUE) == 0)

return LdrDataTableEntry->DllBase;

}

}

}

except (EXCEPTION\_EXECUTE\_HANDLER){}

return NULL;

}

// 根据函数名得到导出表地址

PVOID GetModuleExport(IN PVOID ModuleBase, IN PCCHAR FunctionName)

{

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;

try

{

// 判断是否是DOS头

if (ImageDosHeader->e\_magic != IMAGE\_DOS\_SIGNATURE)

{

return NULL;

}

// 获取PE结构节NT头

ImageNtHeaders32 = (PIMAGE\_NT\_HEADERS32)((PUCHAR)ModuleBase + ImageDosHeader->e\_lfanew);

ImageNtHeaders64 = (PIMAGE\_NT\_HEADERS64)((PUCHAR)ModuleBase + ImageDosHeader->e\_lfanew);

// 判断是否是64位

if (ImageNtHeaders64->OptionalHeader.Magic == IMAGE\_NT\_OPTIONAL\_HDR64\_MAGIC)

{

// 如果是64位则执行如下

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

{

// 如果32位则执行如下

ImageExportDirectory = (PIMAGE\_EXPORT\_DIRECTORY)(ImageNtHeaders32-

>OptionalHeader.DataDirectory[IMAGE\_DIRECTORY\_ENTRY\_EXPORT].VirtualAddress + (ULONG\_PTR)ModuleBase);

ExportDirectorySize = ImageNtHeaders32-

>OptionalHeader.DataDirectory[IMAGE\_DIRECTORY\_ENTRY\_EXPORT].Size;

}

// 取出导出表Index，名字，函地址等

PUSHORT pAddressOfOrds = (PUSHORT)(ImageExportDirectory-

>AddressOfNameOrdinals + (ULONG\_PTR)ModuleBase);

PULONG pAddressOfNames = (PULONG)(ImageExportDirectory->AddressOfNames + (ULONG\_PTR)ModuleBase);

PULONG pAddressOfFuncs = (PULONG)(ImageExportDirectory-

>AddressOfFunctions + (ULONG\_PTR)ModuleBase);

// 循环导出表

for (ULONG i = 0; i < ImageExportDirectory->NumberOfFunctions; ++i)

{

USHORT OrdIndex = 0xFFFF; PCHAR pName = NULL;

// 说明是序号导出

if ((ULONG\_PTR)FunctionName <= 0xFFFF)

{

// 得到函数序号

OrdIndex = (USHORT)i;

}

// 说明是名字导出

else if ((ULONG\_PTR)FunctionName > 0xFFFF && i < ImageExportDirectory->NumberOfNames)

{

// 得到函数名

pName = (PCHAR)(pAddressOfNames[i] + (ULONG\_PTR)ModuleBase); OrdIndex = pAddressOfOrds[i];

}

else

return NULL;

// 判断函数名是否符合

if (((ULONG\_PTR)FunctionName <= 0xFFFF && (USHORT) ((ULONG\_PTR)FunctionName) == OrdIndex + ImageExportDirectory->Base) ||

((ULONG\_PTR)FunctionName > 0xFFFF && strcmp(pName, FunctionName)

== 0))

{

// 得到完整地址

FunctionAddress = pAddressOfFuncs[OrdIndex] +

(ULONG\_PTR)ModuleBase;

break;

}

}

}

except (EXCEPTION\_EXECUTE\_HANDLER){}

return (PVOID)FunctionAddress;

}

// DLL模块注入线程函数

NTSTATUS KernelInjectDLL(ULONG pid, PUNICODE\_STRING DllFullPath, PINJECT\_BUFFER allcateAddress)



{

PEPROCESS pEprocess = NULL;

// 根据PID得到进程Eprocess结构

if (NT\_SUCCESS(PsLookupProcessByProcessId((HANDLE)pid, &pEprocess)))

{

KAPC\_STATE kApc = { 0 };

// 附加到进程内

KeStackAttachProcess(pEprocess, &kApc);

// 得到Ntdll.dll模块基址

UNICODE\_STRING ntdllString = RTL\_CONSTANT\_STRING(L"Ntdll.dll"); PVOID NtdllAddress = GetUserModule(pEprocess, &ntdllString,

g\_PsGetProcessWow64Process(pEprocess) != 0); if (!NtdllAddress)

{

// 失败了则直接脱离附加

KeUnstackDetachProcess(&kApc); ObDereferenceObject(pEprocess); return STATUS\_UNSUCCESSFUL;

}

// 得到LdrLoadDLL模块的基址

PVOID LdrLoadDll = GetModuleExport(NtdllAddress, "LdrLoadDll"); if (!LdrLoadDll)

{

KeUnstackDetachProcess(&kApc); ObDereferenceObject(pEprocess); return STATUS\_UNSUCCESSFUL;

}

HANDLE threadHandle = NULL;

// 得到当前正在运行的线程上下文

if (NT\_SUCCESS(g\_ZwGetNextThread((HANDLE)-1, (HANDLE)0, 0x1FFFFF, 0x240,

0, &threadHandle)))

{

PVOID threadObj = NULL;

// 在对象句柄上提供访问验证，如果可以授予访问权限，则返回指向对象的正文的相应指针。

NTSTATUS state = ObReferenceObjectByHandle(threadHandle, 0x1FFFFF, PsThreadType, KernelMode, &threadObj, NULL);

if (NT\_SUCCESS(state))

{

// 暂停线程

g\_PsSuspendThread(threadObj, NULL);

// 设置线程ShellCode代码

SetThreadStartAddress(threadObj, g\_PsGetProcessWow64Process(pEprocess) != 0, LdrLoadDll, DllFullPath, allcateAddress);

// 恢复线程

g\_PsResumeThread(threadObj, NULL); ObDereferenceObject(threadObj);

}

NtClose(threadHandle);

}

// 关闭线程KeUnstackDetachProcess(&kApc); ObDereferenceObject(pEprocess);

}

return STATUS\_SUCCESS;

}

NTSTATUS UnDriver(PDRIVER\_OBJECT driver)

{

UNREFERENCED\_PARAMETER(driver);

return STATUS\_SUCCESS;

}

NTSTATUS DriverEntry(IN PDRIVER\_OBJECT Driver, PUNICODE\_STRING RegistryPath)

{

DbgPrint("Hello LyShark.com \n");

UNREFERENCED\_PARAMETER(RegistryPath);

// -----------------------------------------------------------------------

// 初始化

// -----------------------------------------------------------------------

UCHAR SuspendOpCode[] = { 0x48, 0x8b, 0xf9, 0x83, 0x64, 0x24, 0x20, 0x00, 0x65, 0x48, 0x8b, 0x34, 0x25, 0x88, 0x01 };

UCHAR ResumeOpCode[] = { 0x48, 0x8b, 0xf9, 0xe8, 0xee, 0x4f, 0xa5, 0xff, 0x65, 0x48, 0x8b, 0x14, 0x25, 0x88 };

// 特征码检索PsSuspendThread函数基址

g\_PsSuspendThread = (PPsSuspendThread)SearchOPcode(Driver, L"ntoskrnl.exe", "PAGE", SuspendOpCode, sizeof(SuspendOpCode), -24);

DbgPrint("PsSuspendThread = %p \n", g\_PsSuspendThread);

// 特征码检索PsResumeThread基址

g\_PsResumeThread = (PPsResumeThread)SearchOPcode(Driver, L"ntoskrnl.exe", "PAGE", ResumeOpCode, sizeof(ResumeOpCode), -18);

DbgPrint("PsResumeThread = %p \n", g\_PsResumeThread);

// 动态获取内存中的ZwGetNextThread基址

UNICODE\_STRING ZwGetNextThreadString = RTL\_CONSTANT\_STRING(L"ZwGetNextThread");

g\_ZwGetNextThread = (PZwGetNextThread)MmGetSystemRoutineAddress(&ZwGetNextThreadString);

DbgPrint("ZwGetNextThread = %p \n", g\_ZwGetNextThread);

// 动态获取内存中的PsGetThreadTeb基址

UNICODE\_STRING PsGetThreadTebString = RTL\_CONSTANT\_STRING(L"PsGetThreadTeb"); g\_PsGetThreadTeb =

(PPsGetThreadTeb)MmGetSystemRoutineAddress(&PsGetThreadTebString); DbgPrint("PsGetThreadTeb = %p \n", g\_PsGetThreadTeb);

// 动态获取内存中的PsGetProcessWow64Process基址



UNICODE\_STRING PsGetProcessWow64ProcessString = RTL\_CONSTANT\_STRING(L"PsGetProcessWow64Process");

g\_PsGetProcessWow64Process = (PPsGetProcessWow64Process)MmGetSystemRoutineAddress(&PsGetProcessWow64ProcessStr ing);

DbgPrint("PsGetProcessWow64Process = %p \n", g\_PsGetProcessWow64Process);

// -----------------------------------------------------------------------

// 注入代码

// -----------------------------------------------------------------------

ULONG ProcessID = 984; UNICODE\_STRING InjectDllPath =

RTL\_CONSTANT\_STRING(L"C:\\Users\\lyshark\\Desktop\\hook.dll"); PINJECT\_BUFFER AllcateAddress = NULL;

// 执行线程注入

NTSTATUS Status = KernelInjectDLL(ProcessID, &InjectDllPath, &AllcateAddress);

if (Status == STATUS\_SUCCESS)

{

DbgPrint("[ ] 线程注入PID = %d | DLL = %wZ \n", ProcessID, InjectDllPath);

}

Driver->DriverUnload = UnDriver; return STATUS\_SUCCESS;

}

首先你需要自行准备好一个DLL文件，此处我的是 hook.dll 将文件放入到桌面，然后设置 ProcessID 指定进程ID，设置 InjectDllPath 指定DLL路径，签名后将驱动加载起来，此时你会看到 WinDBG 中的输出，且应用层的进程也会弹出 hello lyshark 的消息，说明注入成功了，如下图所示；

