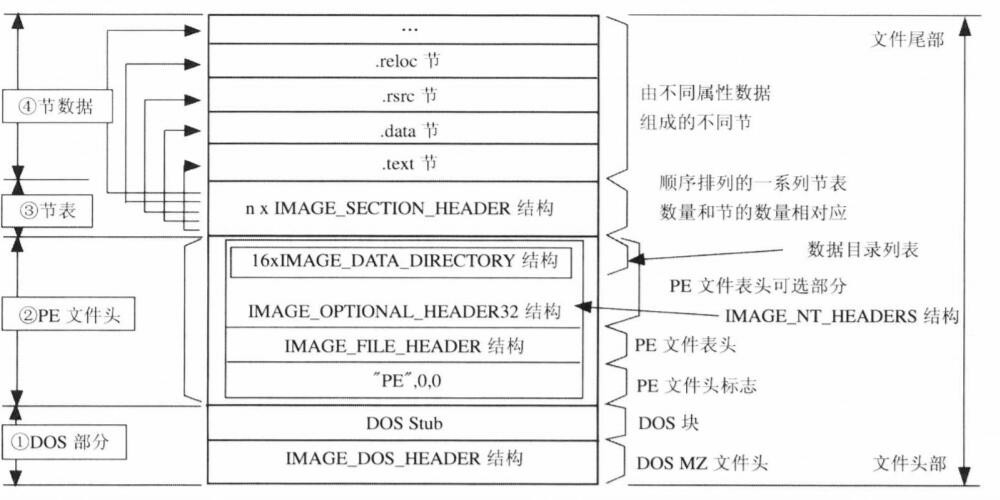
在笔者上一篇文章 《驱动开发：内核解析PE结构导出表》 介绍了如何解析内存导出表结构，本章将继续延 申实现解析PE结构的PE头，PE节表等数据，总体而言内核中解析PE结构与应用层没什么不同，在上一篇 文章中 LyShark 封装实现了 KernelMapFile() 内存映射函数，在之后的章节中这个函数会被多次用

# 到，为了减少代码冗余，后期文章只列出重要部分，读者可以自行去前面的文章中寻找特定的片段。

Windows NT 系统中可执行文件使用微软设计的新的文件格式，也就是至今还在使用的PE格式，PE文件的基本结构如下图所示：



在PE文件中,代码,已初始化的数据,资源和重定位信息等数据被按照属性分类放到不同的 Section(节区/或

简称为节) 中,而每个节区的属性和位置等信息用一个 IMAGE\_SECTION\_HEADER 结构来描述,所有的

IMAGE\_SECTION\_HEADER 结构组成了一个节表 (Section Table) ,节表数据在PE文件中被放在所有节数

# 据的前面.

上面PE结构图中可知PE文件的开头部分包括了一个标准的DOS可执行文件结构，这看上去有些奇怪，但 是这对于可执行程序的向下兼容性来说却是不可缺少的，当然现在已经基本不会出现纯DOS程序了，现 在来说这个 IMAGE\_DOS\_HEADER 结构纯粹是历史遗留问题。

**DOS头结构解析：** PE文件中的DOS部分由MZ格式的文件头和可执行代码部分组成，可执行代码被称为

DOS块(DOS stub) ，MZ格式的文件头由 IMAGE\_DOS\_HEADER 结构定义，在C语言头文件 winnt.h 中有对这个DOS结构详细定义,如下所示：

|  |  |  |  |
| --- | --- | --- | --- |
| typedef struct \_IMAGE\_DOS\_HEADER WORD e\_magic;  WORD e\_cblp; | { | //  // | DOS的头部  Bytes on last page of file |
| WORD e\_cp; |  | // | Pages in file |
| WORD e\_crlc; |  | // | Relocations |
| WORD e\_cparhdr;  WORD e\_minalloc; |  | //  // | Size of header in paragraphs  Minimum extra paragraphs needed |
| WORD e\_maxalloc; |  | // | Maximum extra paragraphs needed |
| WORD e\_ss; |  | // | Initial (relative) SS value |
| WORD e\_sp;  WORD e\_csum; |  | //  // | Initial SP value  Checksum |
| WORD e\_ip; |  | // | Initial IP value |
| WORD e\_cs; |  | // | Initial (relative) CS value |
| WORD e\_lfarlc;  WORD e\_ovno; |  | //  // | File address of relocation table  Overlay number |
| WORD e\_res[4]; |  | // | Reserved words |
| WORD e\_oemid; |  | // | OEM identifier (for e\_oeminfo) |



WORD e\_oeminfo;

WORD e\_res2[10]; LONG e\_lfanew;

} IMAGE\_DOS\_HEADER,

// OEM information; e\_oemid specific

// Reserved words

// 指向了PE文件的开头(重要) PIMAGE\_DOS\_HEADER;

在DOS文件头中，第一个字段 e\_magic 被定义为 MZ ,标志着DOS文件的开头部分,最后一个字段

# e\_lfanew 则指明了PE文件的开头位置，现在来说除了第一个字段和最后一个字段有些用处，其他字段几乎已经废弃了，这里附上读取DOS头的代码。

void DisplayDOSHeadInfo(HANDLE ImageBase)

{

PIMAGE\_DOS\_HEADER pDosHead = NULL;

pDosHead = (PIMAGE\_DOS\_HEADER)ImageBase;

printf("DOS头： %x\n", pDosHead->e\_magic);

printf("文件地址： %x\n", pDosHead->e\_lfarlc);

printf("PE结构偏移： %x\n", pDosHead->e\_lfanew);

}

**PE头结构解析：** 从DOS文件头的 e\_lfanew 字段向下偏移 003CH 的位置，就是真正的PE文件头的位置， 该文件头是由 IMAGE\_NT\_HEADERS 结构定义的，定义结构如下：



IMAGE\_OPTIONAL\_HEADER32 OptionalHeader;

} IMAGE\_NT\_HEADERS32, PIMAGE\_NT\_HEADERS32;

// PE文件标识字符

typedef struct \_IMAGE\_NT\_HEADERS {

DWORD Signature; IMAGE\_FILE\_HEADER FileHeader;

如上PE文件头的第一个DWORD是一个标志，默认情况下它被定义为00004550h也就是 P,E 两个字符另外加上两个零，而大部分的文件属性由标志后面的 IMAGE\_FILE\_HEADER 和 IMAGE\_OPTIONAL\_HEADER32 结构来定义，我们继续跟进 IMAGE\_FILE\_HEADER 这个结构：



typedef struct \_IMAGE\_FILE\_HEADER {

WORD Machine; // 运行平台

WORD NumberOfSections; // 文件的节数目

DWORD TimeDateStamp; // 文件创建日期和时间DWORD PointerToSymbolTable; // 指向符号表(用于调试) DWORD NumberOfSymbols; // 符号表中的符号数量

WORD SizeOfOptionalHeader; // IMAGE\_OPTIONAL\_HANDLER32结构的长度 WORD Characteristics; // 文件的属性 exe=010fh dll=210eh

} IMAGE\_FILE\_HEADER, PIMAGE\_FILE\_HEADER;

# 继续跟进了：

IMAGE\_OPTIONAL\_HEADER32

结构，该结构体中的数据就丰富了，重要的结构说明经备注好

typedef struct \_IMAGE\_OPTIONAL\_HEADER {

WORD BYTE BYTE DWORD DWORD

DWORD

Magic; MajorLinkerVersion; MinorLinkerVersion; SizeOfCode; SizeOfInitializedData;

SizeOfUninitializedData;

// 连接器版本

// 所有包含代码节的总大小

// 所有已初始化数据的节总大小

// 所有未初始化数据的节总大小

DWORD AddressOfEntryPoint; // 程序执行入口RVA DWORD BaseOfCode; // 代码节的起始RVA



DWORD BaseOfData; // 数据节的起始RVA

DWORD ImageBase; // 程序镜像基地址

DWORD SectionAlignment; // 内存中节的对其粒度

DWORD FileAlignment; // 文件中节的对其粒度WORD MajorOperatingSystemVersion; // 操作系统主版本号WORD MinorOperatingSystemVersion; // 操作系统副版本号

WORD MajorImageVersion; // 可运行于操作系统的最小版本号

WORD MinorImageVersion;

WORD MajorSubsystemVersion; // 可运行于操作系统的最小子版本号

WORD MinorSubsystemVersion; DWORD Win32VersionValue;

DWORD SizeOfImage; // 内存中整个PE映像尺寸

DWORD SizeOfHeaders; // 所有头加节表的大小

DWORD CheckSum; WORD Subsystem;

WORD DllCharacteristics;

DWORD SizeOfStackReserve; // 初始化时堆栈大小

DWORD SizeOfStackCommit; DWORD SizeOfHeapReserve; DWORD SizeOfHeapCommit; DWORD LoaderFlags;

DWORD NumberOfRvaAndSizes; // 数据目录的结构数量

IMAGE\_DATA\_DIRECTORY DataDirectory[IMAGE\_NUMBEROF\_DIRECTORY\_ENTRIES];

} IMAGE\_OPTIONAL\_HEADER32, PIMAGE\_OPTIONAL\_HEADER32;

# IMAGE\_DATA\_DIRECTORY数据目录列表，它由16个相同的IMAGE\_DATA\_DIRECTORY结构组成，这16

个数据目录结构定义很简单仅仅指出了某种数据的位置和长度，定义如下：



typedef struct \_IMAGE\_DATA\_DIRECTORY {

DWORD VirtualAddress; // 数据起始RVA DWORD Size; // 数据块的长度

} IMAGE\_DATA\_DIRECTORY, PIMAGE\_DATA\_DIRECTORY;

上方的结构就是PE文件的重要结构，接下来将通过编程读取出PE文件的开头相关数据，读取这些结构也 非常简单代码如下所示。

// 署名权

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

// PowerBy: LyShark

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

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

{

DbgPrint("hello lyshark.com \n");

NTSTATUS status = STATUS\_SUCCESS; HANDLE hFile = NULL;

HANDLE hSection = NULL; PVOID pBaseAddress = NULL;

UNICODE\_STRING FileName = { 0 };

// 初始化字符串

RtlInitUnicodeString(&FileName, L"\\??\\C:\\Windows\\System32\\ntdll.dll");

// 内存映射文件

status = KernelMapFile(FileName, &hFile, &hSection, &pBaseAddress); if (!NT\_SUCCESS(status))

{

return 0;

}

// 获取PE头数据集

PIMAGE\_DOS\_HEADER pDosHeader = (PIMAGE\_DOS\_HEADER)pBaseAddress; PIMAGE\_NT\_HEADERS pNtHeaders = (PIMAGE\_NT\_HEADERS)((PUCHAR)pDosHeader +

pDosHeader->e\_lfanew);

PIMAGE\_FILE\_HEADER pFileHeader = &pNtHeaders->FileHeader;

DbgPrint("运行平台: %x\n", pFileHeader->Machine);

DbgPrint("节区数目: %x\n", pFileHeader->NumberOfSections);

DbgPrint("时间标记: %x\n", pFileHeader->TimeDateStamp);

DbgPrint("可选头大小 %x\n", pFileHeader->SizeOfOptionalHeader);

DbgPrint("文件特性: %x\n", pFileHeader->Characteristics);

DbgPrint("入口点： %p\n", pNtHeaders-

>OptionalHeader.AddressOfEntryPoint);

DbgPrint("镜像基址： %p\n", pNtHeaders->OptionalHeader.ImageBase);

DbgPrint("镜像大小： %p\n", pNtHeaders->OptionalHeader.SizeOfImage);

DbgPrint("代码基址： %p\n", pNtHeaders->OptionalHeader.BaseOfCode);

DbgPrint("区块对齐： %p\n", pNtHeaders->OptionalHeader.SectionAlignment);

DbgPrint("文件块对齐： %p\n", pNtHeaders->OptionalHeader.FileAlignment);

DbgPrint("子系统： %x\n", pNtHeaders->OptionalHeader.Subsystem);

DbgPrint("区段数目： %d\n", pNtHeaders->FileHeader.NumberOfSections);

DbgPrint("时间日期标志： %x\n", pNtHeaders->FileHeader.TimeDateStamp);

DbgPrint("首部大小： %x\n", pNtHeaders->OptionalHeader.SizeOfHeaders);

DbgPrint("特征值： %x\n", pNtHeaders->FileHeader.Characteristics);

DbgPrint("校验和： %x\n", pNtHeaders->OptionalHeader.CheckSum);

DbgPrint("可选头部大小： %x\n", pNtHeaders->FileHeader.SizeOfOptionalHeader);

DbgPrint("RVA 数及大小： %x\n", pNtHeaders-

>OptionalHeader.NumberOfRvaAndSizes);

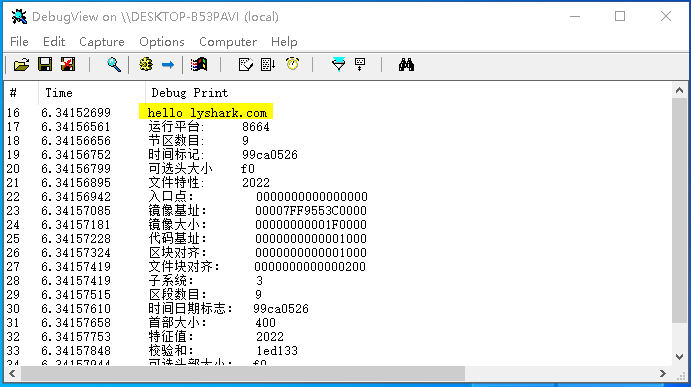
ZwUnmapViewOfSection(NtCurrentProcess(), pBaseAddress); ZwClose(hSection);

ZwClose(hFile);

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

}

运行如上这段代码，即可解析出 ntdll.dll 模块的核心内容，如下图所示；



# 接着来实现解析节表，PE文件中的所有节的属性定义都被定义在节表中，节表由一系列的

IMAGE\_SECTION\_HEADER 结构排列而成，每个结构邮过来描述一个节，节表总被存放在紧接在PE文件头的地方，也即是从PE文件头开始偏移为 00f8h 的位置处，如下是节表头部的定义。



} IMAGE\_SECTION\_HEADER, PIMAGE\_SECTION\_HEADER;

// 节区属性字段

// 节区RVA

// 在文件中对齐后的尺寸

// 在文件中的偏移

// 在OBJ文件中使用

VirtualAddress; SizeOfRawData; PointerToRawData; PointerToRelocations; PointerToLinenumbers; NumberOfRelocations; NumberOfLinenumbers;

Characteristics;

} Misc; DWORD DWORD DWORD DWORD DWORD WORD WORD

DWORD

typedef struct \_IMAGE\_SECTION\_HEADER { BYTE Name[IMAGE\_SIZEOF\_SHORT\_NAME];

union {

DWORD PhysicalAddress;

DWORD VirtualSize; // 节区尺寸

解析节表也很容易实现，首先通过 pFileHeader->NumberOfSections 获取到节数量，然后循环解析直到所有节输出完成，这段代码实现如下所示。

// 署名权

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

// PowerBy: LyShark

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

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

{

DbgPrint("hello lyshark.com \n");

NTSTATUS status = STATUS\_SUCCESS; HANDLE hFile = NULL;

HANDLE hSection = NULL; PVOID pBaseAddress = NULL;

UNICODE\_STRING FileName = { 0 };



// 初始化字符串

RtlInitUnicodeString(&FileName, L"\\??\\C:\\Windows\\System32\\ntdll.dll");

// 内存映射文件

status = KernelMapFile(FileName, &hFile, &hSection, &pBaseAddress); if (!NT\_SUCCESS(status))

{

return 0;

}

// 获取PE头数据集

PIMAGE\_DOS\_HEADER pDosHeader = (PIMAGE\_DOS\_HEADER)pBaseAddress; PIMAGE\_NT\_HEADERS pNtHeaders = (PIMAGE\_NT\_HEADERS)((PUCHAR)pDosHeader +

pDosHeader->e\_lfanew);

PIMAGE\_SECTION\_HEADER pSection = IMAGE\_FIRST\_SECTION(pNtHeaders); PIMAGE\_FILE\_HEADER pFileHeader = &pNtHeaders->FileHeader;

DWORD NumberOfSectinsCount = 0;

// 获取区块数量

NumberOfSectinsCount = pFileHeader->NumberOfSections;

DWORD64 difA = NULL; // 虚拟地址开头

DWORD64 difS = NULL; // 相对偏移(用于遍历)

difA = ExAllocatePool(NonPagedPool, NumberOfSectinsCount sizeof(DWORD64)); difS = ExAllocatePool(NonPagedPool, NumberOfSectinsCount sizeof(DWORD64));

DbgPrint("节区名称 相对偏移\t虚拟大小\tRaw数据指针\tRaw数据大小\t节区属性\n");

for (DWORD temp = 0; temp<NumberOfSectinsCount; temp++, pSection++)

{

DbgPrint("%10s\t 0x%x \t 0x%x \t 0x%x \t 0x%x \t 0x%x \n",

pSection->Name, pSection->VirtualAddress, pSection->Misc.VirtualSize, pSection->PointerToRawData, pSection->SizeOfRawData, pSection-

>Characteristics);

difA[temp] = pSection->VirtualAddress;

difS[temp] = pSection->VirtualAddress - pSection->PointerToRawData;

}

ZwUnmapViewOfSection(NtCurrentProcess(), pBaseAddress); ZwClose(hSection);

ZwClose(hFile);

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

}

运行驱动程序，即可输出 ntdll.dll 模块的节表信息，如下图；

