北京郵電大學
Beijing University of Posts and Telecommunications

Merak
hacking securily

汇编语言与逆向工程
Assembly Language and Software Reverse Engineering

北京邮电大学
付俊松

2024年4月

第六章 从PE结构到LoadLibrary

- 一. PE简介
- 二.检查PE格式
- 三.内存映像结构
- 四.基址重定位
- 五.导入表
- 六.导出表

第六章 从PE结构到LoadLibrary

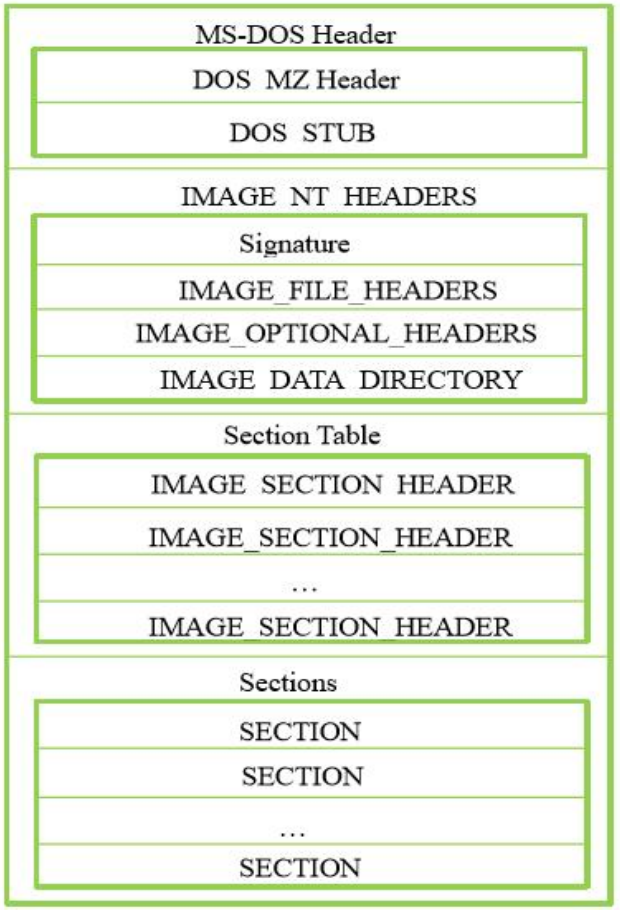
（1）PE简介

- PE（Portable and Executable File Format）
  - PE是Windows平台主流可执行文件格式, .exe , .dll, .sys, .com文件都是PE格式
  - 32位的PE文件称为PE32，64位的称为PE32+
  - PE文件格式在winnt.h头中有着详细的定义
  - PE文件头包含了一个程序在运行时需要的所有信息
    - 包括了如何将文件加载到内存、开辟多大的堆栈空间、调用哪些DLL以及相关函数、从何处开始运行，这些信息都以结构体的形式存储在PE头中

第六章 从PE结构到LoadLibrary

（1）PE简介

- PE格式的大致布局
  - PE文件包括四个组成部分
    - MS-DOS（Disk Operation System）头
    - NT头（New Technology）
    - Section table表中包含了所有的section头
    - 所有的section实体（段实体）



第六章 从PE结构到LoadLibrary

（1）PE简介

- 实现LoadLibrary，即把PE文件加载到内存中需要经过四步
  - 判定输入文件是否是PE格式
  - 将PE文件按照内存映像结构分块放在内存中
  - 在IAT（Import Address Table，导入地址表）中，填入其依赖的导入函数地址
  - 利用重定位表修复需要重定位的值

第六章 从PE结构到LoadLibrary

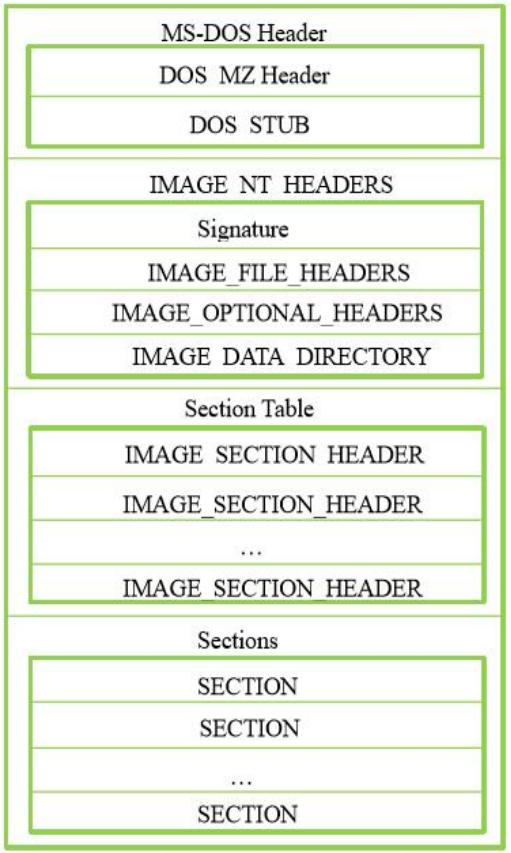
- 一. PE简介
- 二.检查PE格式
- 三.内存映像结构
- 四.基址重定位
- 五.导入表
- 六.导出表

（2）检查PE格式

- PE文件加载，首先要检查的是该文件是否为PE格式，还需要检查该PE文件是否为DLL
  - 对于PE格式的检测，需要检查的部分是MS-DOS头中“MZ”关键字和NT头中“PE/0/0”关键字
  - 对于DLL的检测，则需要检查NT头中的IMAGE_FILE_HEADER的Characteristics字段下IMAGE_FILE_DLL信息位

（2）检查PE格式

- （1）MS-DOS头
- （2）NT头
- （3）DLL检查（只针对DLL）



（2）检查PE格式

- （1） MS-DOS头
  - MS-DOS头是微软为了考虑PE文件对DOS文件的兼容性而添加的。
  - 大多数情况下由编译器自动生成，通常把DOS MZ头与DOS stub（模拟对象接口）合称为DOS文件头
  - IMAGE_DOS_HEADER结构体如下图
    - IMAGE_DOS_HEADER结构体共64字节，其中两个字段比较重要，分别是e_magic和e_lfanew。

(2) 检查PE格式

```
typedefstruct _IMAGE_DOS_HEADER{
+0h              WORD     e_magic; //DOS signature : 4D5A("MZ")
+2h              WORD     e_cblp;
+4h              WORD     e_cp;
+6h              WORD     e_crlc;
+8h              WORD     e_cparhdr;
+ah              WORD     e_minalloc;
+ch              WORD     e_maxalloc;
+eh              WORD     e_ss;
+10h             WORD     e_sp;
+12h             WORD     e_csum;
+14h             WORD     e_ip;
+16h             WORD     e_cs;
+18h             WORD     e_lfarlc;
+1ah             WORD     e_ovno;
+1ch             WORD     e_res[4];
+24h             WORD     e_oemid;
+26h             WORD     e_oeminfo;
+28h             WORD     e_res2[10];
+3ch             DWORD    e_lfanew; //Relative address of NT header
} IMAGE_DOS_HEADER, *PIMGAE_DOS_HEADER;
```

（2）检查PE格式

- e_magic需要被设置为0x5A4D，其ASCII值为“MZ”，为DOS签名，标志着DOS头的开始
- e_lfanew字段是NT头的相对偏移，其指出NT头的文件偏移位置，共占用四个字节，位于文件开始偏移0x3C字节中

(2) 检查PE格式

- （2） NT头
  - 在DOS stub后的是NT头（IMAGE_NT_HEADERS）

IMAGE_NT_HEADERS结构体

```
typedefstruct _IMAGE_NT_HEADERS{
+0h             DWORD Signature; // PE Signature : 50450000("PE\0\0")
+4h             IMAGE_FILE_HEADER          FileHeader;
+18h            IMAGE_OPTIONAL_HEADER32    OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

（2）检查PE格式

– Signature

➢在一个有效的PE文件中，Signature字段必须被设置为0x00004550，对应于ASCII字符“PE\0\0”。

```
000000E0  50 45 00 00 4C 01 03 00  87 52 02 48 00 00 00 00  PE  L   ‡R H
000000F0  00 00 00 00 E0 00 0F 01  0B 01 07 0A 00 78 00 00      à        x
00000100  00 88 00 00 00 00 00 00  9D 73 00 00 00 10 00 00  ˆ         s
```

(2) 检查PE格式

– IMAGE_FILE_HEADER

- IMAGE_FILE_HEADER结构包含了PE文件的基本信息，最重要的是其中一个字段指出了IMAGE_OPTIONAL_HEADER的大小

```
typedefstruct _IMAGE_FILE_HEADER{
+4h    WORD              Machine;
+6h    WORD              NumberOfSections;
+8h    DWORD             TimeDateStamp;
+ch    DWORD             PointerToSymbolTable;
+10h   DOWRD             NumberOfSymbols;
+14h   WORD              SizeOfOptionalHeader;
+16h   WORD              Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

（2）检查PE格式

- 其中Machine， NumberOfSections，SizeOfOptionalHeader， Characteristics如果出现错误，将导致该PE文件无法正常执行
- #1 Machine
  - 该字段说明了可执行文件的目标CPU类型，每类CPU都有唯一的Machine码

(2) 检查PE格式

– 常见的一些Machine码

Machine	Value
Inteli386	14Ch
MIPS R3000	162h
MIPS R4000	166h
Alpha AXP	184h
Power PC	1F0h

(2) 检查PE格式

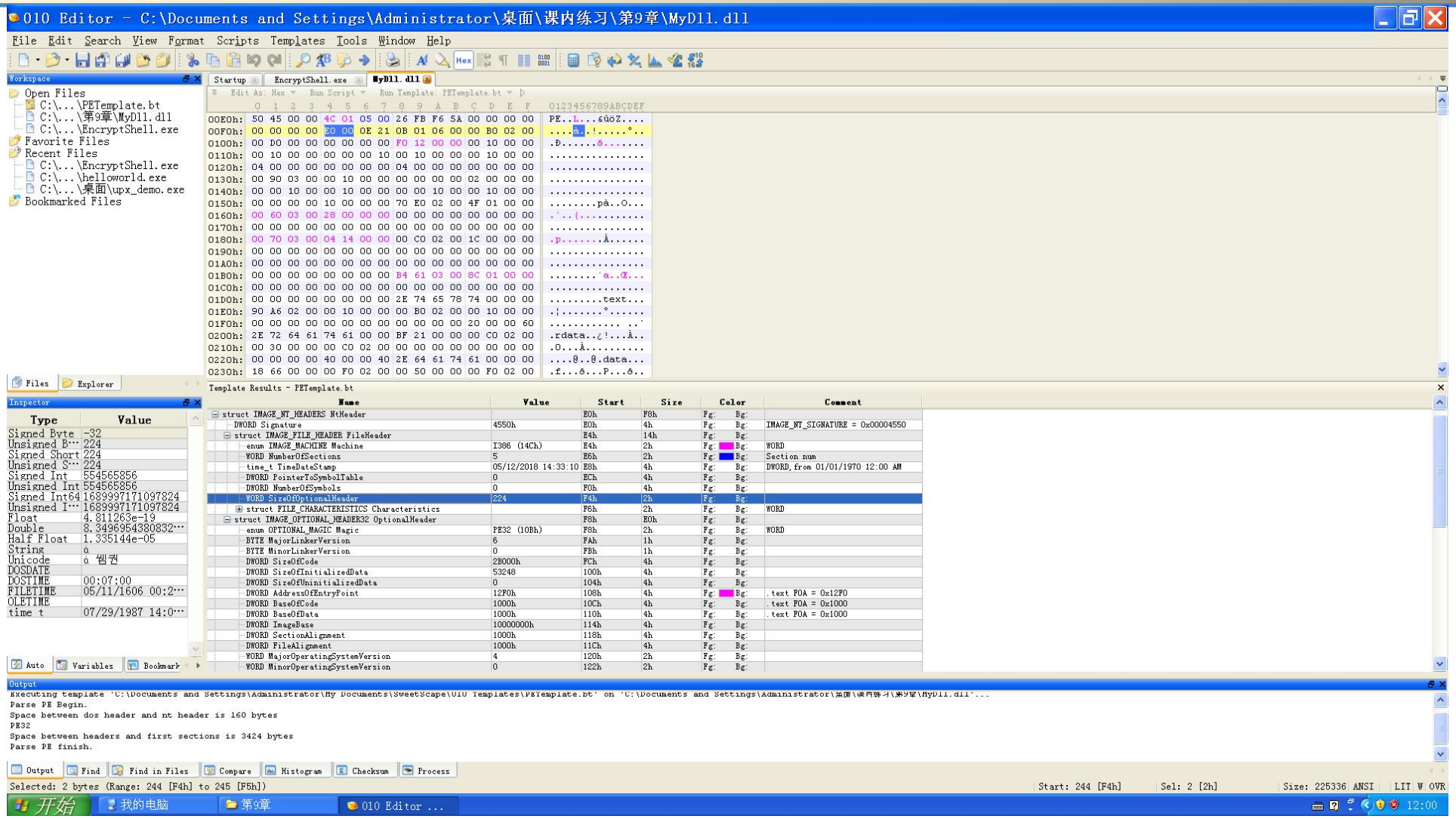
- #2 NumberOfSections
  - 该字段说明了在这个PE文件中节区（Section）的数目
- #3 TimeDateStamp
  - 该字段说明了该PE文件是何时被创建的
- #4 PointerToSymbolTable
  - 该字段说明了COFF符号表(基本用不到)的文件偏移位置，COFF符号表在PE文件中较为少见，通常其值为0

(2) 检查PE格式

- #5 NumberOfSymbols
  - 如果存在COFF符号表，该字段说明了其中的符号数目
- #6 SizeOfOptionalHeader
  - 该字段说明了紧跟在IMAGE_FILE_HEADER后的数据大小
  - 对于32位文件，该字段通常为0x00E0，对于64位文件，该字段通常为0x00F0

第六章 从PE结构到LoadLibrary

(2) 检查PE格式



（2）检查PE格式

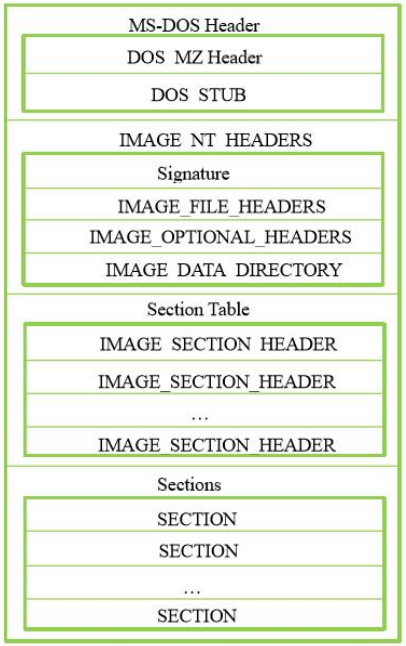
– #7 Characteristics

- 该字段用于标识文件的属性，文件是否可执行，是否为DLL等文件信息，这些信息以比特位的方式组合起来

(2) 检查PE格式

– IMAGE_OPTIONAL_HEADER

- IMAGE_OPTIONAL_HEADER虽然叫做可选头，但是仅有IMAGE_FILE_HEADER并不足以定义PE文件的属性
- IMAGE_OPTIONAL_HEADER定义了更多的PE文件的属性，两者结合起来描述了一个完整的PE文件



(2) 检查PE格式

- #1 Magic
  - 当IMAGE_OPTIONAL_HEADER为IMAGE_OPTIONAL_HEADER32（32位）时，Magic为0x10B。当其为IMAGE_OPTIONAL_HEADER64时，Magic为0x20B。
- #2 AddressOfEntryPoint
  - 该字段的值为相对虚拟地址（加载到内存的地址），该值表明了程序最先执行的代码的启始地址，即程序入口点。
- #3 ImageBase
  - 该字段表明了PE文件被加载进内存时，文件将被优先装入的虚拟内存的地址。对于EXE来说，ImageBase通常为0x00400000；对于DLL来说，ImageBase通常为0x10000000。装载后，EIP = ImageBase + AddressOfEntryPoint。

（2）检查PE格式

- #4 SectionAlignment， FileAlignment
  - PE文件的Body部分划分为若干节区，FileAlignment制定了节区在文件系统中的最小单位，SectionAlignment则指定了节区在内存中的最小单位
  - 磁盘文件或内存的节区大小必定为FileAlignment或SectionAlignment的整数倍
- #5 SizeOfImage
  - 加载PE文件时，SizeOfImage指定了PE Image在虚拟内存中所占的空间大小
- #6 SizeOfHeaders
  - 该字段表明了整个PE文件头部的大小，该值必须是FileAlignment的整数倍

（2）检查PE格式

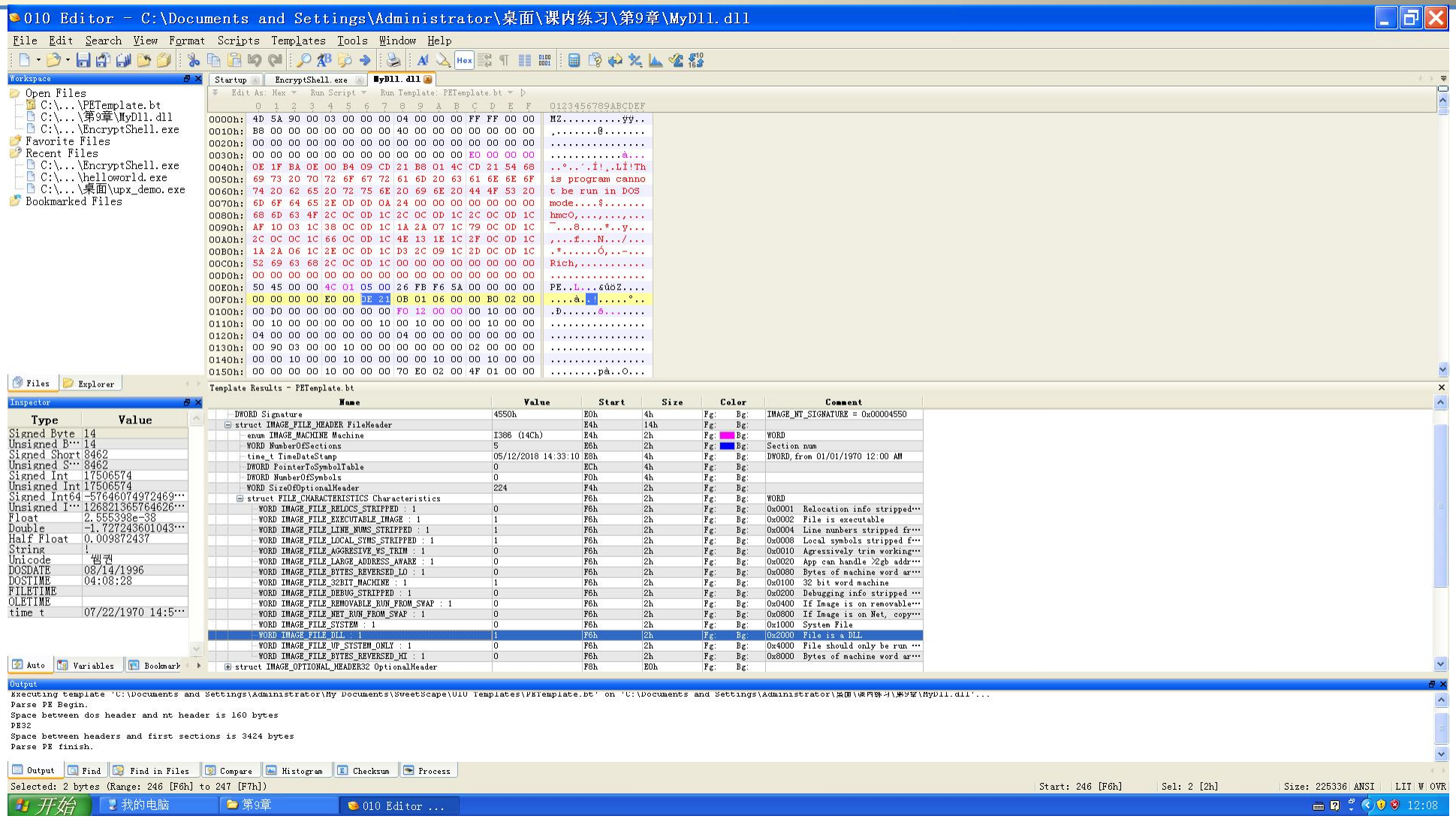
- #7 Subsystem
  - 该字段用于区分系统驱动文件与普通可执行文件。
- #8 NumberOfRvaAndSizes
  - 该字段表明了下面出现的DataDirectory数组的个数。一般来说该值为16。
- #9 DataDirectory
  - DataDirectory是由IMAGE_DATA_DIRECTORY结构体构成的数组，数组的每项都有不同的意义

（2）检查PE格式

- （3）PE格式检查
  - PE格式检查主要针对于MS-DOS头和NT头。要求MS-DOS头和NT头 的签名与规定相同。其中MS-DOS头的签名为0x4D5A即ASCII码的”MZ”。
  - 通过MS-DOS头中的e_lfanew成员变量找到NT头。检查NT头签名为0x50450000即ASCII的”PE\0\0”
  - 根据NT头中FileHeader中的Characteristics中的IMAGE_FILE_DLL位可以判断该PE文件是否为DLL 。

第六章 从PE结构到LoadLibrary

(2) 检查PE格式



第六章 从PE结构到LoadLibrary

- 一. PE简介
- 二.检查PE格式
- 三.内存映像结构
- 四.基址重定位
- 五.导入表
- 六.导出表

（3）内存映像结构

- 在检查了PE文件格式之后，第二步是将PE文件从硬盘中映射到内存映像结构

（3）内存映像结构

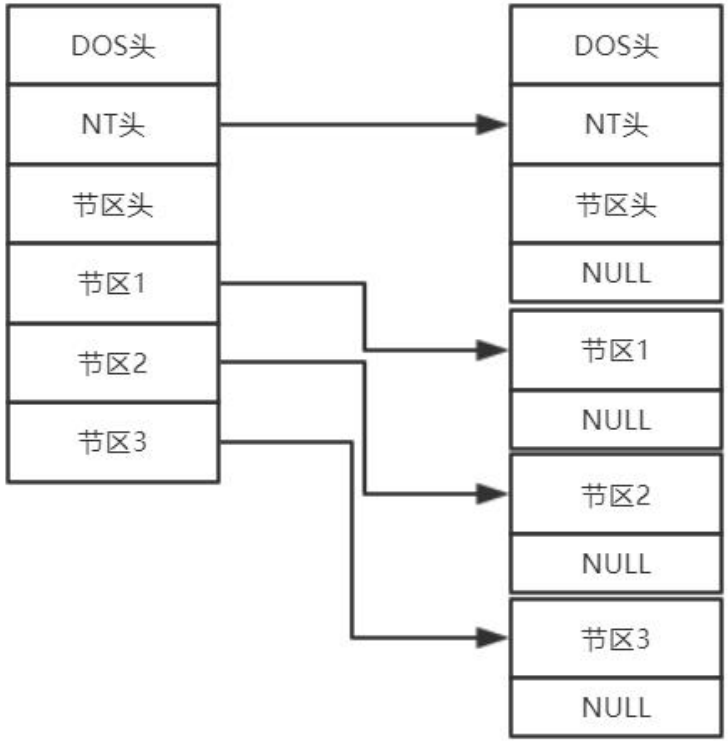
- 1.最小基本单元
- 2.程序处理
- 3.RVA&VA
- 4.从文件偏移到相对虚拟地址

（3）内存映像结构 — 最小基本单元

- 1.最小基本单元
  - 计算机中，为了提高处理文件过程中，内存的效率，使用“最小基本单元”这一概念
  - PE文件映射到内存后节区的起始位置应该在最小基本单元的倍数上
  - 在最小基本单元中空余的空间填NULL

（3）内存映像结构 — 最小基本单元

– 内存映像结构



（3）内存映像结构 — 程序处理

2.程序处理

- 首先将PE文件的MS-DOS头，NT头以及节区头拷贝到开辟的内存空间的首地址处。
- 下面的代码中pFileBuf存储了从硬盘中读取的PE数据，pFileBuf_New为依据SizeofImage开辟的新内存空间
- 头部大小可由可选头中的SizeOfHeaders成员变量获得

```
m_dwSizeOfHeader = m_pNtHeader->OptionalHeader.SizeOfHeaders;
memcpy(pFileBuf_New,m_pFileBuf,m_dwSizeOfHeader);
```

（3）内存映像结构 — 程序处理

- Windows提供了一个宏IMAGE_FIRST_SECTION，可以根据NT头直接返回第一个节区头的指针
- 由每个节区头中的PointerToRawData（指针指向的原始数据），VirtualAddress以及SizeOfRawData成员变量，可以获知每个节区的数据在pFileBuf中的首地址，该数据应该被放在pFileBuf_New的地址加上VirtualAddress

```
PIMAGE_SECTION_HEADER pSectionHeader=IMAGE_FIRST_SECTION(m_pNtHeader);
for(DWORD i=0;i<m_dwSectionNum;i++,pSectionHeader++){
        memcpy(pFileBuf_New+pSectionHeader->VirtualAddress,
                m_pFileBuf+pSectionHeader->PointerToRawData,
                pSectionHeader->SizeOfRawData);
}
```

（3）内存映像结构 — RVA&VA

3. RVA&VA

- RVA是在PE文件中为了避免使用确定的内存地址，出现了相对虚拟地址（Relative Virtual Address，简称RVA）
- RVA是内存中相对于PE文件装入地址的偏移位置，是一个“相对地址”，或称为“偏移量”
- VA指的是进程装入内存后实际的内存地址，被称为虚拟地址（Virutal Address，简称VA）
- VA=Image Base + RVA

（3）内存映像结构 — RVA&VA

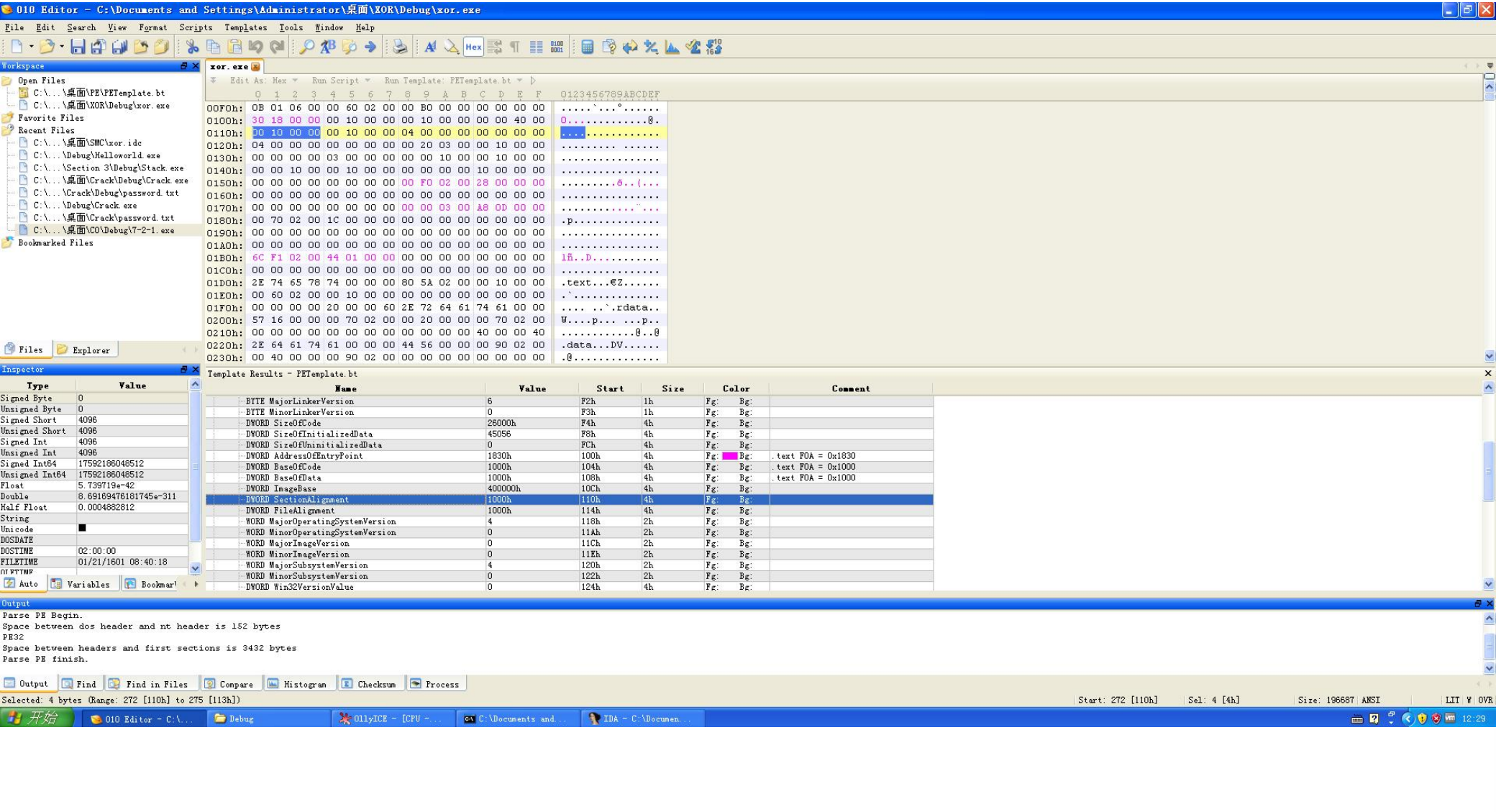
- 其中基地址是PE文件通过Windows加载器装入内存后，该模块的初始内存地址就被称为基地址

（3）内存映像结构 — 从文件偏移到相对虚拟地址

- 4.从文件偏移到相对虚拟地址
  - 在以上小节的地址计算中，都是在文件映射到内存之后进行的
  - 但是PE文件在存储时为了减少体积，FileAlignment通常小于SectionAlignment
  - 当文件被映射到内存中后，同一数据在文件中的偏移量与在内存中的偏移量是不一样的，这样就存在这从文件偏移地址（RAW）到相对虚拟地址(RVA)之间的转换
    - 如果需要对存储在硬盘中的PE文件进行操作，需要将RVA转换为RAW

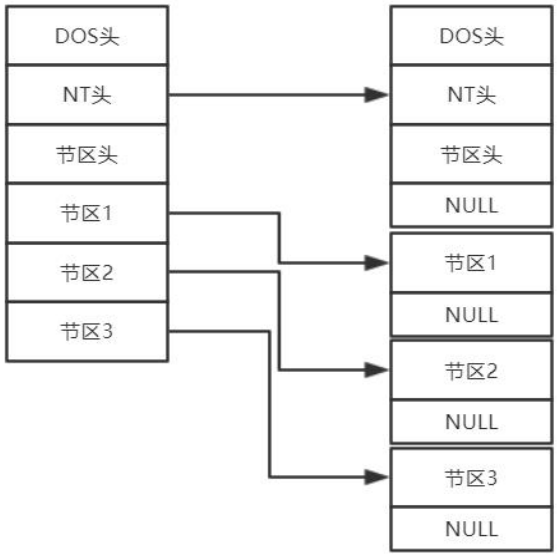
第六章 从PE结构到LoadLibrary

（3）内存映像结构 — 从文件偏移到相对虚拟地址



（3）内存映像结构 — 从文件偏移到相对虚拟地址

- 由于应用程序的映射是以节区为单位做的映射，一个节区内数据的地址相对于节区的地址是不变的，因此只需要计算各节区在磁盘与内存中起始地址的差值即可



（3）内存映像结构 — 从文件偏移到相对虚拟地址

将该差值以 θ 表示，RAW 与 RVA 之间的关系如下所示

$$RAW = RVA - \theta$$

由于

$$RVA = VA - ImageBase$$

RAW 的公式也可改写为下式

$$RAW = VA - ImageBase - \theta$$

（3）内存映像结构 — 从文件偏移到相对虚拟地址

– 以notepad为例，其差值如表

节区	RVA	RAW	差值
.text	1000h	400h	0C00h
.data	9000h	7C00h	1400h
.rsrc	B000h	8400h	2C00h

（3）内存映像结构 — 从文件偏移到相对虚拟地址

- 在计算某虚拟地址对应的文件偏移时，应首先查看其属于哪一节区，找到相应的差值后再进行转换
- 以上述**notepad**为例，如给定一虚拟地址**0x402854**，**ImageBase**为**0x400000**，要求计算其文件偏移地址

（3）内存映像结构 — 从文件偏移到相对虚拟地址

– 计算结果

可知其处于.text 块中，此时θ 为 0xC00。故

$$RAW = VA - ImageBase - \theta = 0x402854 - 0x400000 - 0xC00 = 0x1C54$$

第六章 从PE结构到LoadLibrary

- 一. PE简介
- 二.检查PE格式
- 三.内存映像结构
- 四.基址重定位
- 五.导入表
- 六.导出表

（4）基址重定位

- 由第二节中对可选头的描述可知，可选头的ImageBase成员变量描述了程序在装入内存时优先装入的地址
- 在生成PE文件时，EXE文件优先装入的地址是0x400000，DLL文件优先装入的地址是0x10000000

(4) 基址重定位

Template Results - PETemplate.bt

Name	Value	Start	Size	Color
struct IMAGE_OPTIONAL_HEADER32 OptionalHeader		118h	E0h	Fg: Bg
enum OPTIONAL_MAGIC Magic	PE32 (10Bh)	118h	2h	Fg: Bg
BYTE MajorLinkerVersion	14	11Ah	1h	Fg: Bg
BYTE MinorLinkerVersion	12	11Bh	1h	Fg: Bg
DWORD SizeOfCode	20400h	11Ch	4h	Fg: Bg
DWORD SizeOfInitializedData	75776	120h	4h	Fg: Bg
DWORD SizeOfUninitializedData	0	124h	4h	Fg: Bg
DWORD AddressOfEntryPoint	618Dh	128h	4h	Fg: Bg
DWORD BaseOfCode	1000h	12Ch	4h	Fg: Bg
DWORD BaseOfData	22000h	130h	4h	Fg: Bg
DWORD ImageBase	400000h	134h	4h	Fg: Bg
DWORD SectionAlignment	1000h	138h	4h	Fg: Bg
DWORD FileAlignment	200h	13Ch	4h	Fg: Bg

EXE优先装入的地址

Template Results - PETemplate.bt

Name	Value	Start	Size	Color
struct IMAGE_OPTIONAL_HEADER32 OptionalHeader		F8h	E0h	Fg: Bg:
enum OPTIONAL_MAGIC Magic	PE32 (10Bh)	F8h	2h	Fg: Bg:
BYTE MajorLinkerVersion	6	FAh	1h	Fg: Bg:
BYTE MinorLinkerVersion	0	FBh	1h	Fg: Bg:
DWORD SizeOfCode	2B000h	FCh	4h	Fg: Bg:
DWORD SizeOfInitializedData	53248	100h	4h	Fg: Bg:
DWORD SizeOfUninitializedData	0	104h	4h	Fg: Bg:
DWORD AddressOfEntryPoint	12F0h	108h	4h	Fg: Bg:
DWORD BaseOfCode	1000h	10Ch	4h	Fg: Bg:
DWORD BaseOfData	1000h	110h	4h	Fg: Bg:
DWORD ImageBase	10000000h	114h	4h	Fg: Bg:
DWORD SectionAlignment	1000h	118h	4h	Fg: Bg:
DWORD FileAlignment	1000h	11Ch	4h	Fg: Bg:

DLL优先装入的地址

（4）基址重定位

- 在xp中没有地址随机化，EXE会被默认装入基址处，通常不需要进行基址重定位
- 一个可执行程序要加载的DLL有很多，不能都加载在0x10000000处
- 当地址已经被占用时，就需要加载到未被占用的空间中，此时，程序中的一些绝对地址访问过程中，就会访问或跳转到别的地址空间中，而不是访问或跳转到预期的位置，重定位表就是为此而产生的

(4) 基址重定位

– 在vista及以上的操作系统中，开启了地址随机化保护，EXE也会被加载到别的地址，因此EXE也有了重定位表

Template Results - PETemplate.bt

Name	Value	Start	Size
struct IMAGE_SECTION_HEADER SectionHeaders[0]		1D8h	28h
struct IMAGE_SECTION_HEADER SectionHeaders[1]		200h	28h
struct IMAGE_SECTION_HEADER SectionHeaders[2]		228h	28h
struct IMAGE_SECTION_HEADER SectionHeaders[3]		250h	28h
struct IMAGE_SECTION_HEADER SectionHeaders[4]		278h	28h
BYTE Name[8]	.reloc	278h	8h
union Misc		280h	4h
DWORD VirtualAddress	37000h	284h	4h
DWORD SizeOfRawData	2000h	288h	4h
DWORD PointerToRawData	35000h	28Ch	4h
DWORD PointerToRelocations	0h	290h	4h
DWORD PointerToLinenumbers	0	294h	4h
WORD NumberOfRelocations	0	298h	2h
WORD NumberOfLinenumbers	0	29Ah	2h
struct SECTION_CHARACTERISTICS Characteristics		29Ch	4h

.reloc节区头

第六章 从PE结构到LoadLibrary

(4) 基址重定位

Template Results - PETemplate.bt

Name	Value	Start	Size	Color	Comment
struct IMAGE_DATA_DIRECTORY DataDir2		168h	8h	Fg: Bg:	IMAGE_DIRECTORY_ENTRY_RESOURCE
struct IMAGE_DATA_DIRECTORY DataDir3		170h	8h	Fg: Bg:	IMAGE_DIRECTORY_ENTRY_EXCEPTION
struct IMAGE_DATA_DIRECTORY DataDir4		178h	8h	Fg: Bg:	IMAGE_DIRECTORY_ENTRY_SECURITY
struct IMAGE_DATA_DIRECTORY DataDir5		180h	8h	Fg: Bg:	IMAGE_DIRECTORY_ENTRY_BASERELOC
DWORD VirtualAddress	37000h	180h	4h	Fg: Bg:	.idata FOA = 0x35000
DWORD Size	5124	184h	4h	Fg: Bg:	
struct IMAGE_DATA_DIRECTORY DataDir6		188h	8h	Fg: Bg:	IMAGE_DIRECTORY_ENTRY_DEBUG
struct IMAGE_DATA_DIRECTORY DataDir7		190h	8h	Fg: Bg:	IMAGE_DIRECTORY_ENTRY_ARCHITECTURE
struct IMAGE_DATA_DIRECTORY DataDir8		198h	8h	Fg: Bg:	IMAGE_DIRECTORY_ENTRY_GLOBALPTR
struct IMAGE_DATA_DIRECTORY DataDir9		1A0h	8h	Fg: Bg:	IMAGE_DIRECTORY_ENTRY_TLS
struct IMAGE_DATA_DIRECTORY DataDir10		1A8h	8h	Fg: Bg:	IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG
struct IMAGE_DATA_DIRECTORY DataDir11		1B0h	8h	Fg: Bg:	IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT
struct IMAGE_DATA_DIRECTORY DataDir12		1B8h	8h	Fg: Bg:	IMAGE_DIRECTORY_ENTRY_IAT
struct IMAGE_DATA_DIRECTORY DataDir13		1C0h	8h	Fg: Bg:	IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT
struct IMAGE_DATA_DIRECTORY DataDir14		1C8h	8h	Fg: Bg:	IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR

BASERELOC Directory

(4) 基址重定位

- 1基址重定位结构定义
- 2程序处理

(4) 基址重定位 — 基址重定位结构定义

- 1基址重定位结构定义
  - 重定位表由许多重定位块串接组成，每个重定位块中存放着4KB大小PE文件内容的重定位信息
  - 重定位块开头以IMAGE_BASE_RELOCATION开始

```
typedef struct _IMAGE_BASE_RELOCATION {
 +0h    DWORD    VirtualAddress;
 +4h    DWORD    SizeOfBlock;
} IMAGE_BASE_RELOCATION;
```

IMAGE_BASE_RELOCATION结构体

(4) 基址重定位 — 基址重定位结构定义

- #1VirtualAddress
  - 声明了该组重定位数据开始的相对虚拟地址，各项重定位地址与该值相加，才是需要进行重定位的相对虚拟地址
- #2SizeOfBlock
  - 声明了该组重定位数据的大小，其中包含了IMAGE_BASE_RELOCATION
  - 在IMAGE_BASE_RELOCATION之后紧随的是TypeOffset数组，数组每项大小为两个字节，高四位代表重定位类型，低十二位值为重定位地址
  - 最终所有的重定位块以一个VirtualAddress为0的IMAGE_BASE_RELOCATION结构结束

(4) 基址重定位 — 基址重定位结构定义

– 常见的重定位类型如下

类型	预定义值	含义
0	IMAGE_REL_BASED_ABSOLUTE	无具体含义，用于四字节对其
3	IMAGE_REL_BASED_HIGHLOW	重定位指向的地址需要被修正
10	IMAGE_REL_BASED_DIR64	出现在64位PE中，该地址需要被修正

- 对于x86文件，所有需要处理的基址重定位类型都是IMAGE_REL_BASED_HIGHLOW
- IMAGE_REL_BASED_ABSOLUTE只是用于填充，以便于四字节对齐

(4) 基址重定位 — 程序处理

- 2 程序处理
  - 重定位表在PE文件中往往单独分为1个节区，名称为“.reloc”，可以由可选头中DataDirectory中的BASERELOC Directory成员找到
  - 整个代码共有两个循环，第一个循环遍历每个重定位块，第二个循环遍历每个重定位块中的重定位信息。根据每个重定位信息的高四位确定其是否需要重定位，需要重定位时，根据程序预期存储位置ImageBase以及当前程序存储位置m_pFileBuf对其地址进行修正

（4）基址重定位 — 程序处理

```
void CPE::Base_Reloc()
{
	PIMAGE_BASE_RELOCATION pRelocBlock = PIMAGE_BASE_RELOCATION(m_pFileBuf + m_PERelocDir.VirtualAddress);
	do
	{
		int numofReloc = (pRelocBlock->SizeOfBlock - sizeof(IMAGE_BASE_RELOCATION)) / 2;
		WORD minioffset = 0;
		WORD *pRelocData = (WORD*)((char*)pRelocBlock + sizeof(IMAGE_BASE_RELOCATION));

		for (int i = 0; i < numofReloc; i++)
		{
			DWORD *RelocAddress = 0;
			if (((*pRelocData) >> 12) == IMAGE_REL_BASED_HIGHLOW)
			{
				minioffset = (*pRelocData) & 0xfff;
				RelocAddress = (DWORD*)(m_pFileBuf + pRelocBlock->VirtualAddress + minioffset);

				*RelocAddress = *RelocAddress - m_dwImageBase + (DWORD)m_pFileBuf;
			}
			pRelocData++;
		}
		pRelocBlock = (PIMAGE_BASE_RELOCATION)((char*)pRelocBlock + pRelocBlock->SizeOfBlock);
	} while (pRelocBlock->VirtualAddress);
}
```

第六章 从PE结构到LoadLibrary

- 一. PE简介
- 二.检查PE格式
- 三.内存映像结构
- 四.基址重定位
- 五.导入表
- 六.导出表

（5）导入表

- 在编写程序时，会使用到大量的库函数，由于动态链接的存在，这些函数并不会都编写进二进制文件中，而是在函数调用处填入对应的导入表地址
- 当程序加载到内存中后，Windows加载器才将相关的DLL装入，并将调用输入函数的指令和函数实际所在地址关联起来

(5) 导入表

– 调用VirtualAlloc函数时，依据二进制查看得到VirtualAlloc地址为0x47d1d8

```
.text:00401657 50                    push    eax             ; lpAddress
.text:00401658 FF 15 D8 D1 47 00     call    ds:VirtualAlloc
.text:0040165E 89 45 F4              mov     [ebp+var_C], eax
.text:00401661 83 7D F4 00           cmp     [ebp+var_C], 0
.text:00401665 75 40                 jnz     short loc_4016A7
.text:00401667 68 24 D0 46 00        push    offset aRelocAddress ; "reloc
.text:0040166C E8 AF 31 00 00        call    _printf
.text:00401671 83 C4 04              add     esp, 4
.text:00401674 6A 04                 push    4               ; flProtect
.text:00401676 68 00 30 00 00        push    3000h           ; flAllocatic
.text:0040167B 8B 4D FC              mov     ecx, [ebp+var_4]
.text:0040167E 8B 51 0C              mov     edx, [ecx+0Ch]
.text:00401681 52                    push    edx             ; dwSize
.text:00401682 6A 00                 push    0               ; lpAddress
```

（5）导入表

– 查看0x47d1d8处，如下图所示，其值在IDA中为未知值，这是因为该PE文件尚未装入内存中，没有在导入地址表中填写相应的地址

```
.idata:0047D1D8                 ; Imports from KERNEL32.dll
.idata:0047D1D8                 ;
.idata:0047D1D8                 ; Section 4. (virtual address 0007D000)
.idata:0047D1D8                 ; Virtual size                  : 000009D5 (   25
.idata:0047D1D8                 ; Section size in file          : 00001000 (   40
.idata:0047D1D8                 ; Offset to raw data for section: 0007B000
.idata:0047D1D8                 ; Flags C0000040: Data Readable Writable
.idata:0047D1D8                 ; Alignment     : default
.idata:0047D1D8                 ; ===========================================
.idata:0047D1D8
.idata:0047D1D8                 ; Segment type: Externs
.idata:0047D1D8                 ; _idata
.idata:0047D1D8                 ; LPVOID __stdcall VirtualAlloc(LPVOID lpAddress,
.idata:0047D1D8 ?? ?? ?? ??             extrn VirtualAlloc:dword
.idata:0047D1D8                                                 ; CODE XR
```

(5) 导入表

– 当其执行后，该处的值会由Windows加载器填写，如图所示，其值已变为0x74cf6970

```
.idata:0047D1D8 ; Imports from KERNEL32.dll
.idata:0047D1D8 ;
.idata:0047D1D8 ; Section 4. (virtual address 0007D000)
.idata:0047D1D8 ; Virtual size                  : 000009D5 (   2517.)
.idata:0047D1D8 ; Section size in file          : 00001000 (   4096.)
.idata:0047D1D8 ; Offset to raw data for section: 0007B000
.idata:0047D1D8 ; Flags C0000040: Data Readable Writable
.idata:0047D1D8 ; Alignment     : default
.idata:0047D1D8 ; ===========================================================================
.idata:0047D1D8
.idata:0047D1D8 ; Segment type: Externs
.idata:0047D1D8 ; _idata
.idata:0047D1D8 ; LPVOID __stdcall VirtualAlloc(LPVOID lpAddress, SIZE_T dwSize, DWORD flAl
.idata:0047D1D8 VirtualAlloc dd offset unk_74CF6970     ; CODE XREF: sub_401530+128↑p
.idata:0047D1D8                                        ; sub_401530+154↑p ...
```

(5) 导入表

- 在程序装入内存时，PE加载器完成了这些工作
- 同样，编写一个自己的LoadLibrary也需要在导入表中填入相应的函数地址

(5) 导入表

- 1 IMAGE_IMPORT_DESCRIPTOR
- 2 程序处理

(5) 导入表 — IMAGE_IMPORT_DESCRIPTOR

1 IMAGE_IMPORT_DESCRIPTOR

- PE文件头的可选映像头中，数据目录表的第二成员指向导入表
- 导入表由IMAGE_IMPORT_DESCRIPTOR（简称IID）数组构成。
- 每个被PE文件导入的DLL都有一个与之对应的IID
- IID中并无字段表明IID数组的长度大小，该数组最后的一个单元为NULL，由此可以计算出IID数组的项数

(5) 导入表 — IMAGE_IMPORT_DESCRIPTOR

IMAGE_IMPORT_DESCRIPTOR的定义

```
typedefstruct _IMAGE_IMPORT_DESCRIPTOR{
+0h             union{
                        DWORD    Characteristics;
                        DWORD    OriginalFirstThunk;        //INT address
                }
+4h             DWORD    TimeDateStamp;
+8h              DWORD    ForwarderChain;
+ch             DWORD    Name;
+10h            DWORD    FirstThunk;                  //IAT address
} IMAGE_IMPORT_DESCRIPTOR;
```

(5) 导入表 — IMAGE_IMPORT_DESCRIPTOR

- #1 OriginalFirstThunk(Characteristics)
  - 包含指向导入名称表的RVA
- #2 TimeDataStamp
  - 一个32位的时间标志
- #3 ForwarderChain
  - 当程序引用一个DLL中的API，而这个API又引用别的DLL的API时使用
- #4 Name
  - DLL名字的指针。名称字符串以\0结尾
- #5 FirstThunk
  - 包含指向导入地址表的RVA

(5) 导入表 — IMAGE_IMPORT_DESCRIPTOR

– 导入名称表（INT）的结构

```
typedefstruct _IMAGE_IMPORT_BY_NAME {
+0h     WORD     Hint;
+2h     CHAR    Name[1];
} IMAGE_IMPORT_BY_NAME,
*PIMAGE_IMPORT_BY_NAME;
```

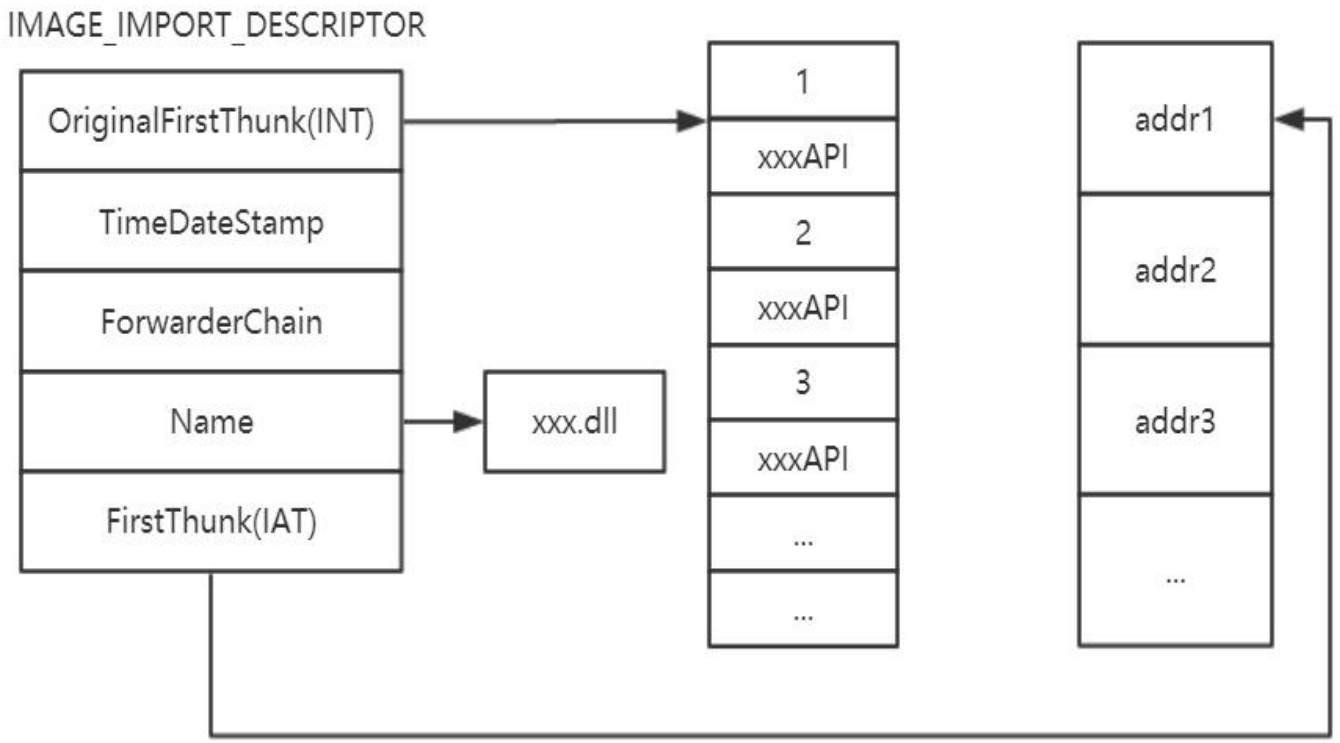
- **#1 Hint**
  - 该字段表明本函数在DLL中的导出表序号。
- **#2 Name**
  - 该字段为函数名，是一个ASCII字符串，以NULL结尾

(5) 导入表 — IMAGE_IMPORT_DESCRIPTOR

- 导入地址表（IAT）中填写对应函数的虚拟地址

(5) 导入表 — IMAGE_IMPORT_DESCRIPTOR

– 导入表整体结构



(5) 导入表 — 程序处理

- 2 程序处理
  - 与基址重定位表类似的，从可选头中的DataDirectory中的IMPORT Directory成员找到
  - 使用LoadLibrary载入所有关联的DLL，使用GetProcAddress获取所有函数地址，填入IAT中
  - 程序共有两个循环，第一个循环为遍历所有需要导入的DLL，第二个循环为遍历每个DLL中需要导入的函数。将获取到的函数地址填入IAT表中即可

（5）导入表 — 程序处理

- 为调用函数，需要执行GetProcAddress获取函数地址
  - 为了使其他函数可调用本PE文件的函数，需要实现GetProcAddress
  - GetProcAddress的实现，则依赖于EAT（Export Address Table，导出地址表）的读取
  - 根据EAT中读取到的地址，以及加载的模块的基址，可计算出所加载的函数的地址

（5）导入表 — 程序处理

```
BOOL CPE::FillIAT()
{
	PIMAGE_IMPORT_DESCRIPTOR pIIDBlock = PIMAGE_IMPORT_DESCRIPTOR(m_pFileBuf + m_PEImportDir.VirtualAddress);
	while (pIIDBlock->Characteristics)
	{
		char* dllname = (char*)(m_pFileBuf + pIIDBlock->Name);
		printf("%s\n", dllname);
		DWORD* pIAT = (DWORD*)(m_pFileBuf + pIIDBlock->FirstThunk);
		DWORD* pINT = (DWORD*)(m_pFileBuf + pIIDBlock->OriginalFirstThunk);
		HMODULE hMod = LoadLibrary(dllname);
		do
		{
			PIMAGE_IMPORT_BY_NAME INT = PIMAGE_IMPORT_BY_NAME(*pINT + m_pFileBuf);
			DWORD ProcAddr = DWORD(GetProcAddress(hMod, (char*)INT->Name));
			printf("hint %x name:%s len:%d IATaddress %x\n", INT->Hint, INT->Name, strlen((char*)INT->Name), ProcAddr);
			*pIAT = ProcAddr;
			pIAT++;
			pINT++;
		} while (*pINT);
		pIIDBlock++;
	}
	return 0;
}
```

(5) 导入表

- 至此，自制的LoadLibrary就已经完成
- 如果是在vista及以上操作系统，使用上面的代码加载EXE文件，去掉对DLL位的校验，通过可选头获取程序入口点，将控制权交给加载好的EXE，即可正常执行EXE程序，可以称之为LoadPE

第六章 从PE结构到LoadLibrary

- 一. PE简介
- 二.检查PE格式
- 三.内存映像结构
- 四.基址重定位
- 五.导入表
- 六.导出表

(6) 导出表

- 上面LoadLibrary已经完成了，接下来需要完成的是与LoadLibrary配套的GetProcAddress
- 仅仅把DLL加载到内存中是不够的，无法得到DLL导出函数的地址，这个DLL就是无效的
- 在DLL中，DataDirectory比EXE中多了一项EXPORTDirectory，通过EXPORT Directory可以找到DLL中的导出地址表

(6) 导出表

- 1 IMAGE_EXPORT_DIRECTORY
- 2 程序处理

(6) 导出表 — IMAGE_EXPORT_DIRECTORY

1 IMAGE_EXPORT_DIRECTORY

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;// RVA from base of image
    DWORD   AddressOfNames;// RVA from base of image
    DWORD   AddressOfNameOrdinals;// RVA from base of image
} IMAGE_EXPORT_DIRECTORY,*PIMAGE_EXPORT_DIRECTORY;
```

(6) 导出表 — IMAGE_EXPORT_DIRECTORY

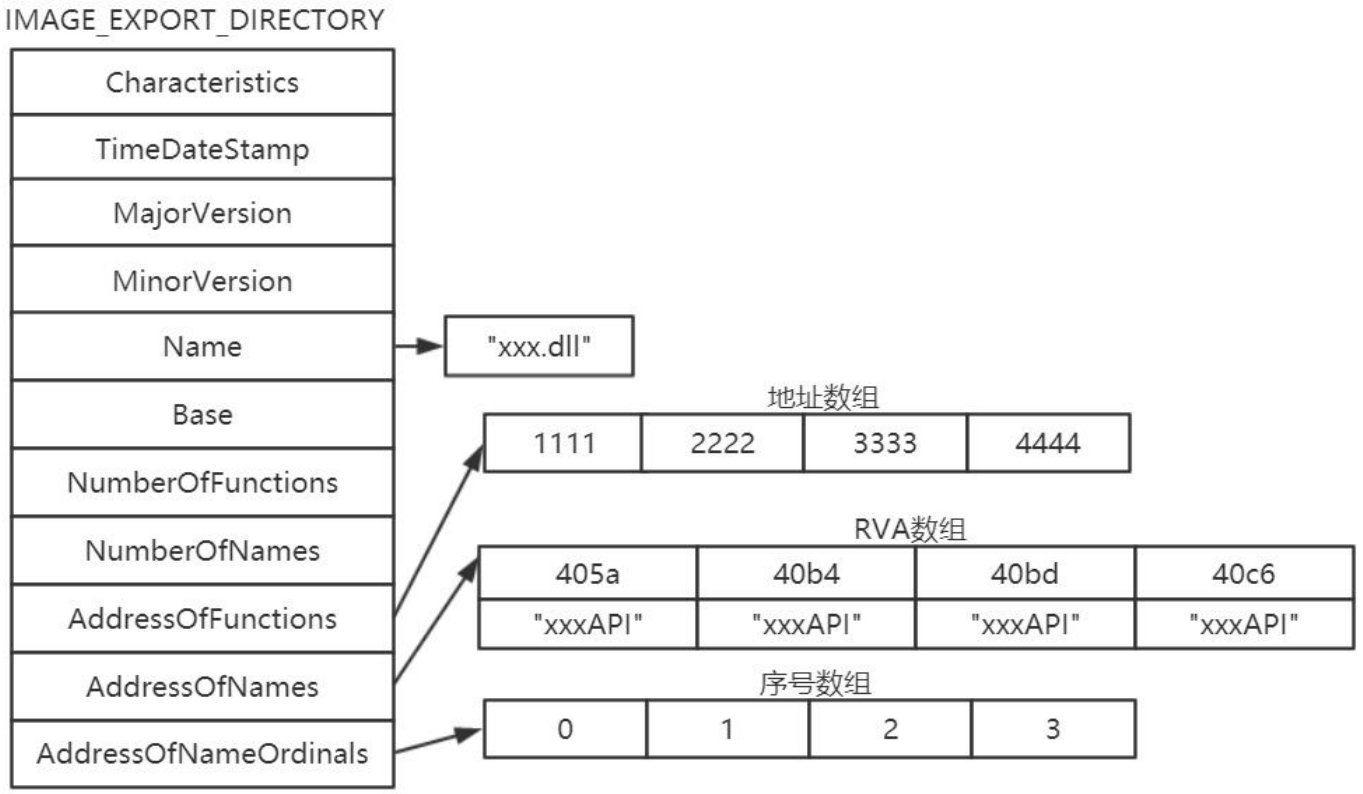
- #1 Characteristics
  - 未定义，为0
- #2 TimeDateStamp
  - 该字段表明输出表创建时间
- #3 MajorVersion
  - 该字段表明输出表的主版本号，未使用，值为0
- #4 MinorVersion
  - 该字段表明输出表的次版本号，未使用，值为0
- #5 Name
  - 该字段指向DLL名称字符串地址
- #6 Base
  - 该字段包含用于这个可执行文件输出表的起始序数值

(6) 导出表 — IMAGE_EXPORT_DIRECTORY

- #7 NumberOfFunctions
  - 该字段表明导出地址表（EAT）中的条目数量
- #8 NumberOfNames
  - 输出函数名称表的条数数量，该值小于或等于NumberOfFunctions。当函数只通过序数输出时会出现NumberOfNames小于NumberOfFunctions
- #9 AddressOfFunctions
  - 该字段指向EAT地址，EAT中存储了所有导出函数的相对虚拟地址
- #10 AddressOfNames
  - 该字段指向导出名称表（ENT）地址，ENT中存储了所有函数名称字符串的相对虚拟地址
- #11 AddressOfNameOrdinals
  - 该字段指向导出序数表地址，导出序数表中存储了所有导出函数的序数

(6) 导出表 — IMAGE_EXPORT_DIRECTORY

– IMAGE_EXPORT_DESCRIPTOR 结构图



(6) 导出表 — 程序处理

- 2 程序处理
  - 从可选头中的DataDirectory中的EXPORT Directory成员找到IMAGE_EXPORT_DIRECTORY地址
  - 从ENT中取出函数名，与要取的函数名进行对比，一致时从EAT中得到函数地址与内存中的DLL基址相加，即得到了函数真实的地址

（6）导出表 — 程序处理

```
DWORD CPE::MyGetProcAddress(char* FuncName)
{
	DWORD i;
	printf("Export table loading~\n");
	printf("Dll base %x\n",m_pFileBuf);
	PIMAGE_EXPORT_DIRECTORY pIIDBlock = PIMAGE_EXPORT_DIRECTORY(m_pFileBuf+m_PEExportDir.VirtualAddress);
	DWORD* AddressOfNames = (DWORD*)(m_pFileBuf + pIIDBlock->AddressOfNames);
	DWORD* AddressOfFunctions = (DWORD*)(m_pFileBuf + pIIDBlock->AddressOfFunctions);
	printf("%x %s %d %d\n",pIIDBlock->Name,m_pFileBuf + pIIDBlock->Name, pIIDBlock->Base, pIIDBlock->NumberOfFunctions);
	for(i=0 ;i <pIIDBlock->NumberOfFunctions; i++)
	{
		printf("%s %x\n",m_pFileBuf + AddressOfNames[i], AddressOfFunctions[i]);
		if(strcmp(FuncName, (char*)(m_pFileBuf + AddressOfNames[i]))==0)
		{
			return (DWORD)(m_pFileBuf+AddressOfFunctions[i]);
		}
	}
	return 0;
}
```

（6）导出表

- 至此，自制的GetProcAddress也已经完成，与之前编写的LoadLibrary配合，就可以实现在内存中加载DLL并获取函数地址

（7）小结

- 开发了一套加载DLL并获取函数地址的代码，可以实现对DLL的一些基本操作，可对PE格式有一个基本的了解
- 这只是一个简单版本的LoadLibrary，更加完善的LoadLibrary除了校验PE格式，将文件映射到内存，进行基址重定位，填充导入表以及执行DllMain之外，还需要在执行DllMain之前执行TLS回调，将重定位节区标为可回收等等工作
- 同时上面的程序只是使用了PE格式中最常见的一些部分。还仍有很多细节以及功能没有介绍，如Sectionheader，延迟装载DLL等
- 想要更加详尽的了解PE格式还需要广泛的查阅资料进行学习

谢　谢！