本节内容

PE文件结构

1、什么是可执行文件？

可执行文件 (executable file) 指的是可以由操作系统进行加载执行的文件。

可执行文件的格式：

Windows平台：

PE(Portable Executable)文件结构

Linux平台：

ELF(Executable and Linking Format)文件结构

哪些领域会用到PE文件格式：

<1> 病毒与反病毒

<2> 外挂与反外挂

<3> 加壳与脱壳(保护与破解)

<4> 无源码修改功能、软件汉化等等

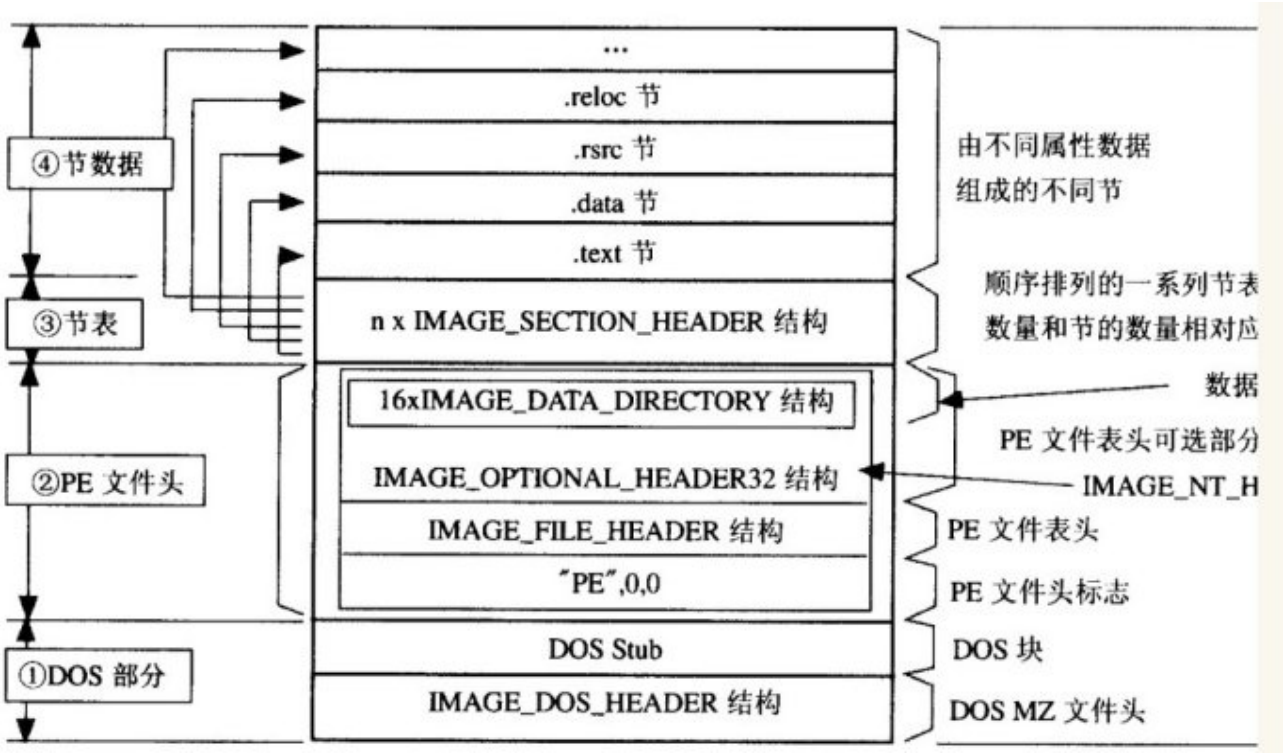
2、如何识别PE文件？

<1> PE文件的特征（PE指纹）

分别打开.exe .dll .sys 等文件,观察特征前2个字节。

<2> 不要仅仅通过文件的后缀名来认定PE文件

3、PE文件的整体结构


...
.reloc 节
.rsrc 节
.data 节
.text 节
n x IMAGE_SECTION_HEADER 结构
16xIMAGE_DATA_DIRECTORY 结构
IMAGE_OPTIONAL_HEADER32 结构
IMAGE_FILE_HEADER 结构
"PE",0,0
DOS Stub
IMAGE_DOS_HEADER 结构
④节数据
③节表
②PE 文件头
①DOS 部分
由不同属性数据
组成的不同节
顺序排列的一系列节表
数量和节的数量相对应
数据
PE 文件表头可选部分
IMAGE_NT_H
PE 文件表头
PE 文件头标志
DOS 块
DOS MZ 文件头

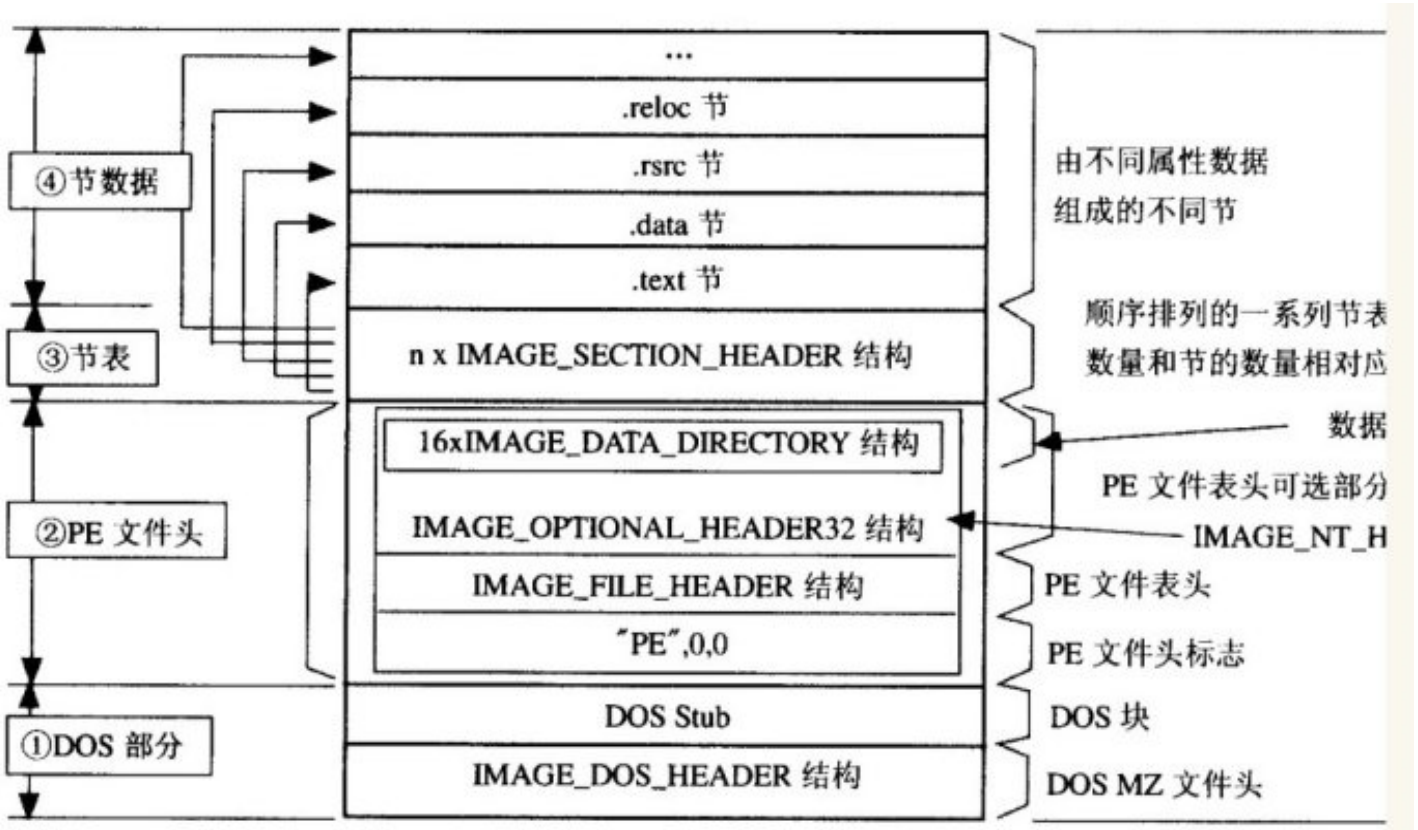
课后练习：

<线上班>学员可见

本节内容

PE文件的两种状态

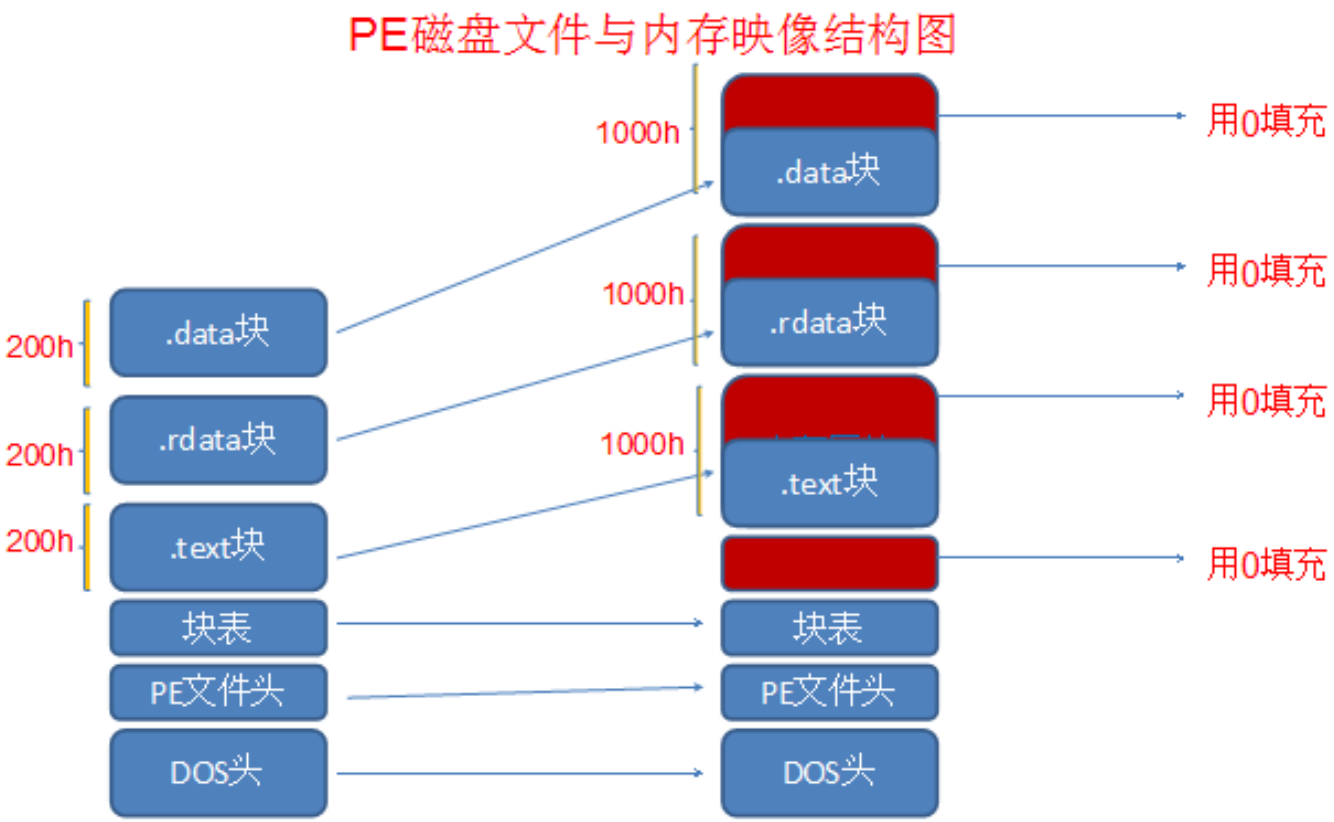
1、主要结构体



IMAGE_DOS_HEADER(64)
IMAGE_FILE_HEADER(20)

IMAGE_OPTIONAL_HEADER32)(224)
IMAGE_SECTION_HEADER)(40)

2、PE文件的两种状态


PE磁盘文件与内存映像结构图
200h
.data块
200h
.rdata块
200h
.text块
块表
PE文件头
DOS头
1000h
.data块
1000h
.rdata块
1000h
.text块
块表
PE文件头
DOS头
用0填充
用0填充
用0填充
用0填充

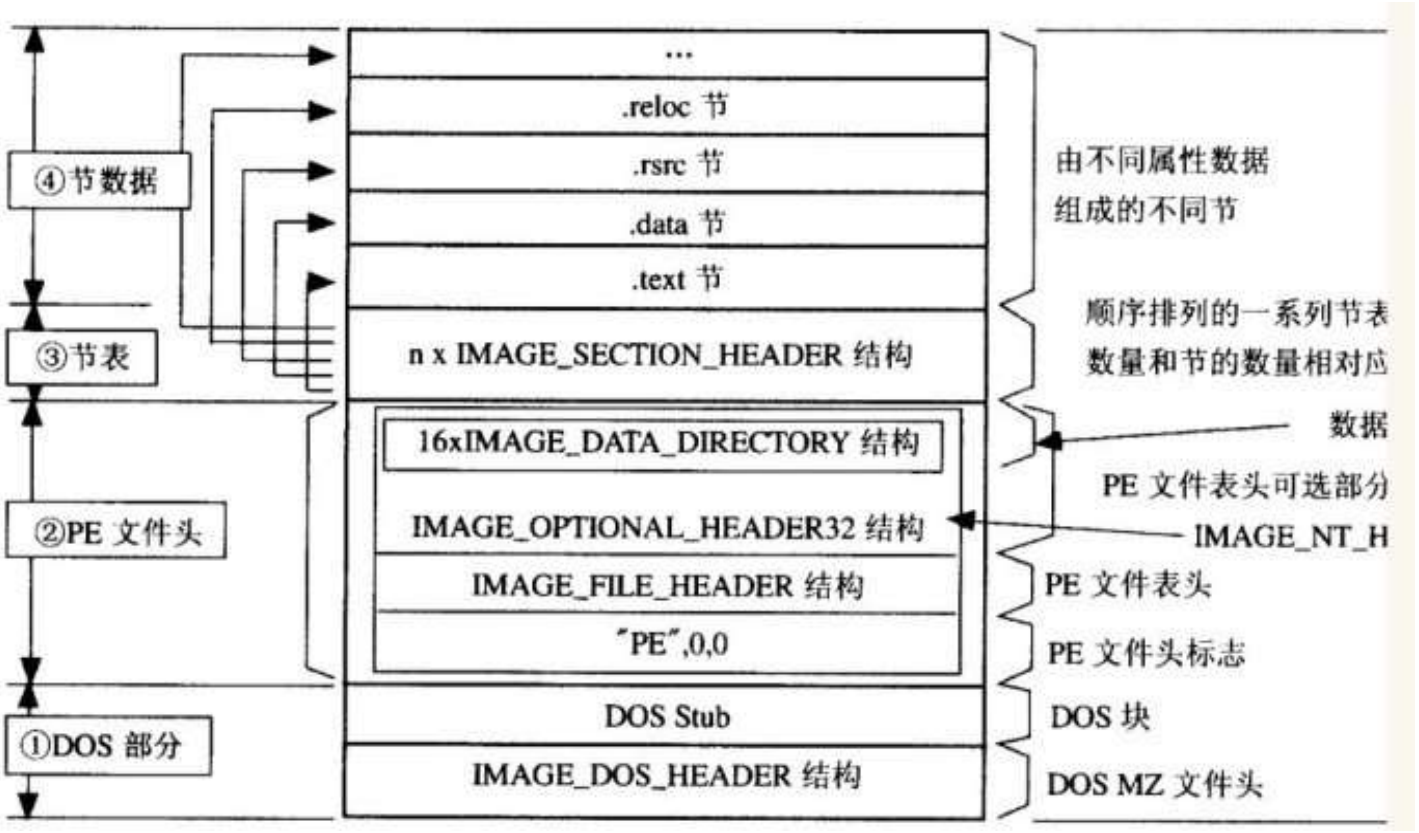
课后练习：

<线上班>学员可见

本节内容

DOS头属性说明

1、PE文件的整体结构


...
.reloc 节
.rsrc 节
.data 节
.text 节
n x IMAGE_SECTION_HEADER 结构
16xIMAGE_DATA_DIRECTORY 结构
IMAGE_OPTIONAL_HEADER32 结构
IMAGE_FILE_HEADER 结构
"PE",0,0
DOS Stub
IMAGE_DOS_HEADER 结构
④节数据
③节表
②PE 文件头
①DOS 部分
由不同属性数据
组成的不同节
顺序排列的一系列节表
数量和节的数量相对应
数据
PE 文件表头可选部分
IMAGE_NT_H
PE 文件表头
PE 文件头标志
DOS 块
DOS MZ 文件头

2、DOS MZ头

```
typedef struct _IMAGE_DOS_HEADER {      // DOS .EXE header
    WORD   e_magic;                     // Magic number
    WORD   e_cblp;                      // Bytes on last page of file
    WORD   e_cp;                        // Pages in file
    WORD   e_crlc;                      // Relocations
    WORD   e_cparhdr;                   // Size of header in paragraphs
    WORD   e_minalloc;                  // Minimum extra paragraphs needed
    WORD   e_maxalloc;                  // Maximum extra paragraphs needed
    WORD   e_ss;                        // Initial (relative) SS value
    WORD   e_sp;                        // Initial SP value
    WORD   e_csum;                      // Checksum
    WORD   e_ip;                        // Initial IP value
    WORD   e_cs;                        // Initial (relative) CS value
    WORD   e_lfarlc;                    // File address of relocation table
    WORD   e_ovno;                      // Overlay number
    WORD   e_res[4];                    // Reserved words
    WORD   e_oemid;                     // OEM identifier (for e_oeminfo)
    WORD   e_oeminfo;                   // OEM information; e_oemid specific
    WORD   e_res2[10];                  // Reserved words
    LONG   e_lfanew;                    // File address of new exe header
  } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

3、DOS 块

<1> DOS块的位置

<2> DOS块的内容

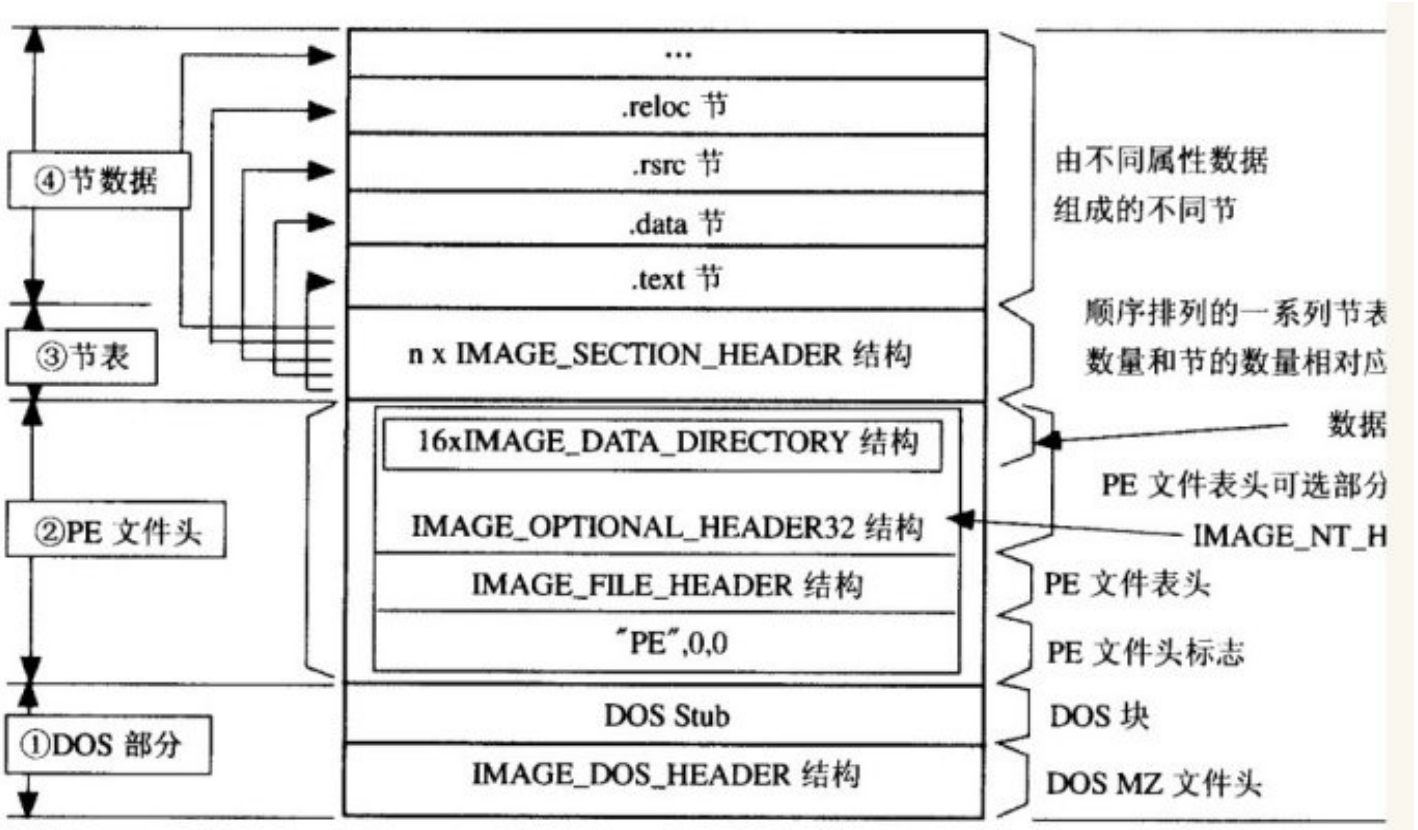
课后练习：

<线上班>学员可见

本节内容

标准PE头属性说明

1、PE文件的整体结构


...
.reloc 节
.rsrc 节
.data 节
.text 节
n x IMAGE_SECTION_HEADER 结构
16xIMAGE_DATA_DIRECTORY 结构
IMAGE_OPTIONAL_HEADER32 结构
IMAGE_FILE_HEADER 结构
"PE",0,0
DOS Stub
IMAGE_DOS_HEADER 结构
④节数据
③节表
②PE 文件头
①DOS 部分
由不同属性数据
组成的不同节
顺序排列的一系列节表
数量和节的数量相对应
数据
PE 文件表头可选部分
IMAGE_NT_H
PE 文件表头
PE 文件头标志
DOS 块
DOS MZ 文件头

2、PE头

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;                                    //PE标识
    IMAGE_FILE_HEADER FileHeader;                       //标准PE头
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;             //扩展PE头
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

PE标识：

PE标识不能破坏，操作系统在启动一个程序的时候会检测这个标识。

3、标准PE头

```
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;                    //可以运行在什么样的CPU上   任意：0    Intel 386以及后续：14C   x64：8664
    WORD    NumberOfSections;           //表示节的数量
    DWORD   TimeDateStamp;              //编译器填写的时间戳 与文件属性里面(创建时间、修改时间)无关
    DWORD   PointerToSymbolTable;       //调试相关
    DWORD   NumberOfSymbols;            //调试相关
    WORD    SizeOfOptionalHeader;       //可选PE头的大小(32位PE文件：0xE0  64位PE文件：0xF0)
    WORD    Characteristics;            //文件属性
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

4、文件属性：IMAGE_FILE_HEADER.Characteristics

数据位	常量符号	为 1 时的含义
0	IMAGE_FILE_RELOCS_STRIPPED	文件中不存在重定位信息
1	IMAGE_FILE_EXECUTABLE_IMAGE	文件是可执行的
2	IMAGE_FILE_LINE_NUMS_STRIPPED	不存在行信息
3	IMAGE_FILE_LOCAL_SYMS_STRIPPED	不存在符号信息
4	IMAGE_FILE_AGGRESSIVE_WS_TRIM	调整工作集
5	IMAGE_FILE_LARGE_ADDRESS_AWARE	应用程序可处理大于 2GB 的地址
6		此标志保留
7	IMAGE_FILE_BYTES_REVERSED_LO	小尾方式
8	IMAGE_FILE_32BIT_MACHINE	只在 32 位平台上运行
9	IMAGE_FILE_DEBUG_STRIPPED	不包含调试信息
10	IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP	不能从可移动盘运行
11	IMAGE_FILE_NET_RUN_FROM_SWAP	不能从网络运行
12	IMAGE_FILE_SYSTEM	系统文件（如驱动程序），不能直接运行
13	IMAGE_FILE_DLL	这是一个 DLL 文件
14	IMAGE_FILE_UP_SYSTEM_ONLY	文件不能在多处理器计算机上运行
15	IMAGE_FILE_BYTES_REVERSED_HI	大尾方式

课后练习：

<线上班>学员可见

本节内容

扩展PE头属性说明

1、PE头

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;


typedef struct _IMAGE_NT_HEADERS64 {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER64 OptionalHeader;
} IMAGE_NT_HEADERS64, *PIMAGE_NT_HEADERS64;
```

2、扩展PE头

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    WORD    Magic;                          //PE32：10B    PE32+：20B
    BYTE    MajorLinkerVersion;             //链接器版本号
    BYTE    MinorLinkerVersion;             //链接器版本号
    DWORD   SizeOfCode;                     //所有代码节的总和  文件对齐后的大小  编译器填的  没用
    DWORD   SizeOfInitializedData;          //包含所有已经初始化数据的节的总大小  文件对齐后的大小 编译器填的  没用
    DWORD   SizeOfUninitializedData;        //包含未初始化数据的节的总大小 文件对齐后的大小 编译器填的  没用
    DWORD   AddressOfEntryPoint;            //程序入口
    DWORD   BaseOfCode;                     //代码开始的基址，编译器填的   没用
    DWORD   BaseOfData;                     //数据开始的基址，编译器填的   没用
    DWORD   ImageBase;                      //内存镜像基址，ImageBase+AddressOfEntryPoint  断点调试常用，可以被加密
    DWORD   SectionAlignment;               //内存对齐
    DWORD   FileAlignment;                  //文件对齐
    WORD    MajorOperatingSystemVersion;    //标识操作系统版本号 主版本号
    WORD    MinorOperatingSystemVersion;    //标识操作系统版本号 次版本号
    WORD    MajorImageVersion;              //PE文件自身的版本号
    WORD    MinorImageVersion;              //PE文件自身的版本号
    WORD    MajorSubsystemVersion;          //运行所需子系统版本号
    WORD    MinorSubsystemVersion;          //运行所需子系统版本号
    DWORD   Win32VersionValue;              //子系统版本的值，必须为0
    DWORD   SizeOfImage;                    //内存中整个PE文件的映射的尺寸，可比实际的值大，必须是SectionAlignment的整数倍
    DWORD   SizeOfHeaders;                  //所有头+节表按照文件对齐后的大小，否则加载会出错
    DWORD   CheckSum;                       //校验和，一些系统文件有要求.用来判断文件是否被修改.
    WORD    Subsystem;                      //子系统          驱动程序(1)  图形界面(2)  控制台、DLL(3)
    WORD    DllCharacteristics;             //文件特性 不是针对DLL文件的
    DWORD   SizeOfStackReserve;             //初始化时保留的栈大小
    DWORD   SizeOfStackCommit;              //初始化时实际提交的大小
    DWORD   SizeOfHeapReserve;              //初始化时保留的堆大小
    DWORD   SizeOfHeapCommit;               //初始化时实践提交的大小
    DWORD   LoaderFlags;                    //调试相关
    DWORD   NumberOfRvaAndSizes;            //目录项数目
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

3、IMAGE_OPTIONAL_HEADER.DllCharacteristics

数据位	常量符号	为1时的含义
0		保留，必须为0
1		保留，必须为0
2		保留，必须为0
3		保留，必须为0
6	IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE	DLL可以在加载时被重定位
7	IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY	强制代码实施完整性验证
8	IMAGE_DLLCHARACTERISTICS_NX_COMPAT	该映像兼容DEP
9	IMAGE_DLLCHARACTERISTICS_NO_ISOLATION	可以隔离，但并不隔离此映像
10	IMAGE_DLLCHARACTERISTICS_NO_SEH	映像不使用SEH（第10章）
11	IMAGE_DLLCHARACTERISTICS_NO_BIND	不绑定映像
12		保留，必须为0
13	IMAGE_DLLCHARACTERISTICS_WDM_DRIVER	该映像为一个WDM driver
14		保留，必须为0
15	IMAGE_DLLCHARACTERISTICS_TERMINAL_SERVER_AWARE	可用于终端服务器

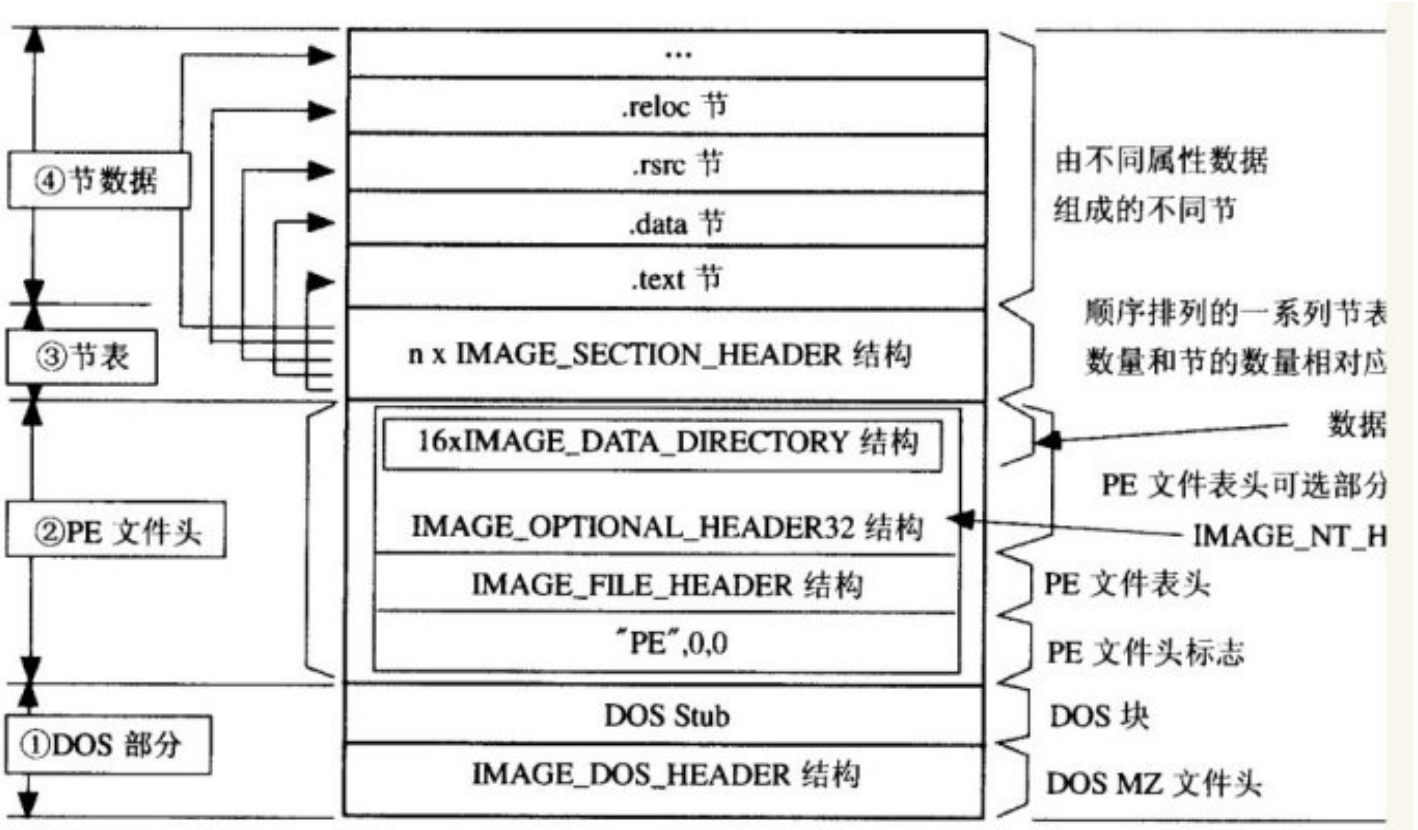
课后练习：

<线上班>学员可见

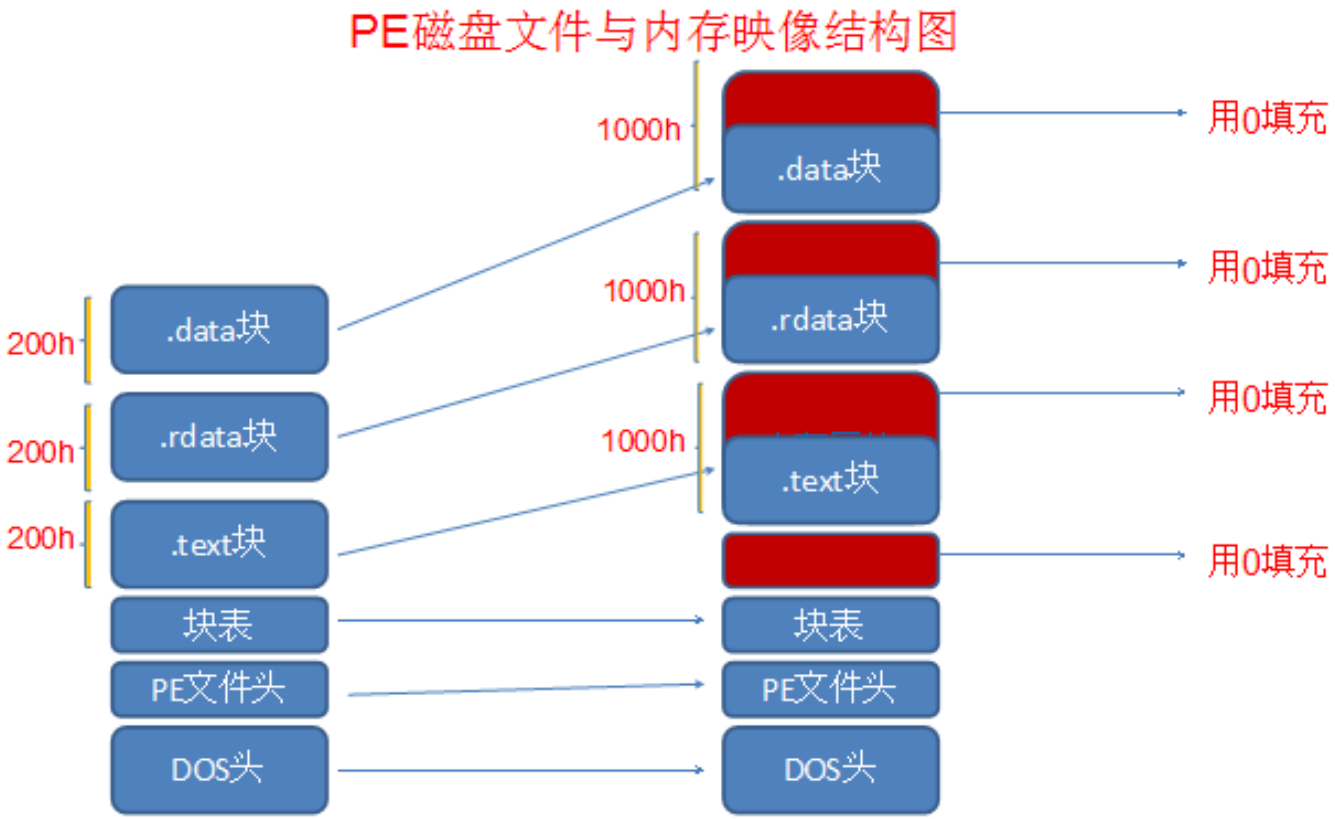
本节内容

PE节表

PE文件的整体结构


...
.reloc 节
.rsrc 节
.data 节
.text 节
n x IMAGE_SECTION_HEADER 结构
16xIMAGE_DATA_DIRECTORY 结构
IMAGE_OPTIONAL_HEADER32 结构
IMAGE_FILE_HEADER 结构
"PE",0,0
DOS Stub
IMAGE_DOS_HEADER 结构
④节数据
③节表
②PE 文件头
①DOS 部分
由不同属性数据
组成的不同节
顺序排列的一系列节表
数量和节的数量相对应
数据
PE 文件表头可选部分
IMAGE_NT_H
PE 文件表头
PE 文件头标志
DOS 块
DOS MZ 文件头

1、PE文件的分节结构


PE磁盘文件与内存映像结构图
1000h
.data块
用0填充
1000h
.rdata块
用0填充
1000h
.text块
用0填充
用0填充
200h
.data块
200h
.rdata块
200h
.text块
块表
PE文件头
DOS头
块表
PE文件头
DOS头

2、节表数据结构说明

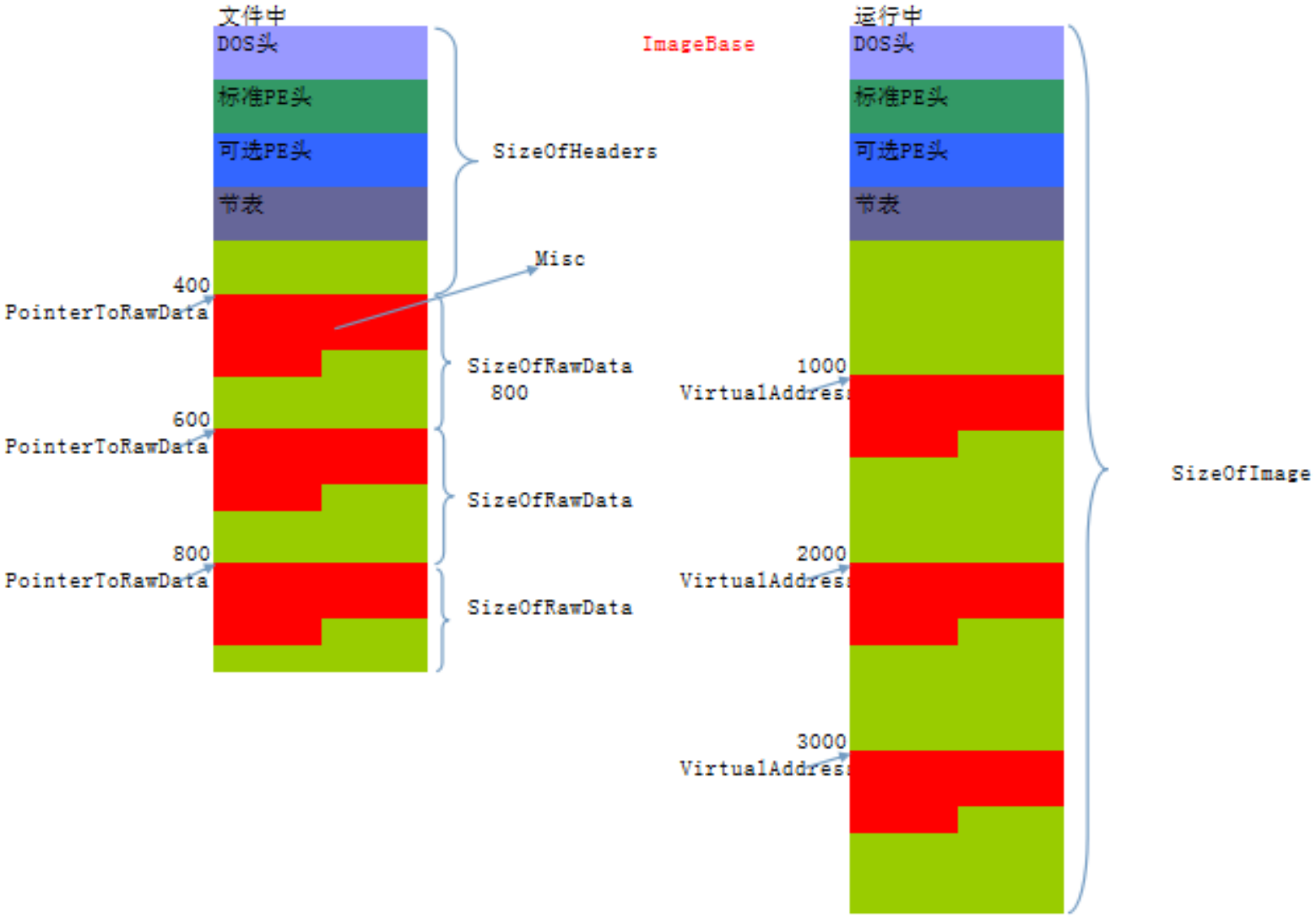
全局变量未赋初始值时，在文件中不在空间，在内存中必须分配空间

```
#define IMAGE_SIZEOF_SHORT_NAME 8
typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];   //ASCII字符串 可自定义  只截取8个 可以8个字节都是名字
    union {                                  //Misc  双字 是该节在没有对齐前的真实尺寸,该值可以不准确
          DWORD   PhysicalAddress;           // 真实长度，这两个值是一个联合结构，可以使用其中的任何个
                DWORD     VirtualSize;       //一般是取后一个

    } Misc;
    DWORD   VirtualAddress;                  //在内存中的偏移地址,加上ImageBase才是在内存中的真正地址
    DWORD   SizeOfRawData;                   //节在文件中对齐后的尺寸
    DWORD   PointerToRawData;                //节区在文件中的偏移
    DWORD   PointerToRelocations;            //调试相关
    DWORD   PointerToLinenumbers;
    WORD    NumberOfRelocations;
    WORD    NumberOfLinenumbers;
    DWORD   Characteristics;                 //节的属性
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

SizeOfRawData 和 联合体VirtualSize的值 的关系
联合体的大小可能大于 SizeOfRawData 。可能等于 也可能小于
没有初始化的变量在文件中是不分配内存的 ， 如果没有初始化的全局变量特别多的话，联合体的值可能会SizeOfRawData大

他们谁大 按谁的来

3、PE文件的两种状态


文件中
DOS头
标准PE头
可选PE头
节表
400
PointerToRawData
600
PointerToRawData
800
PointerToRawData
SizeOfHeaders
Misc
SizeOfRawData
800
SizeOfRawData
SizeOfRawData
ImageBase
运行中
DOS头
标准PE头
可选PE头
节表
1000
VirtualAddres
2000
VirtualAddres
3000
VirtualAddres
SizeOfImage

4、节属性说明

数据位	常量符号	位为1时的含义
5	IMAGE_SCN_CNT_CODE 或 00000020h	节中包含代码
6	IMAGE_SCN_CNT_INITIALIZED_DATA 或 00000040h	节中包含已初始化数据
7	IMAGE_SCN_CNT_UNINITIALIZED_DATA 或 00000080h	节中包含未初始化数据
8	IMAGE_SCN_LNK_OTHER 或 00000100h	保留供将来使用
25	IMAGE_SCN_MEM_DISCARDABLE 或 02000000h	节中的数据在进程开始以后将被丢弃，如 .reloc
26	IMAGE_SCN_MEM_NOT_CACHED 或 04000000h	节中的数据不会经过缓存
27	IMAGE_SCN_MEM_NOT_PAGED 或 08000000h	节中的数据不会被交换到磁盘
28	IMAGE_SCN_MEM_SHARED 或 10000000h	表示节中的数据将被不同的进程所共享
29	IMAGE_SCN_MEM_EXECUTE 或 20000000h	映射到内存后的页面包含可执行属性
30	IMAGE_SCN_MEM_READ 或 40000000h	映射到内存后的页面包含可读属性
31	IMAGE_SCN_MEM_WRITE 或 80000000h	映射到内存后的页面包含可写属性

课后练习：

<线上班>学员可见

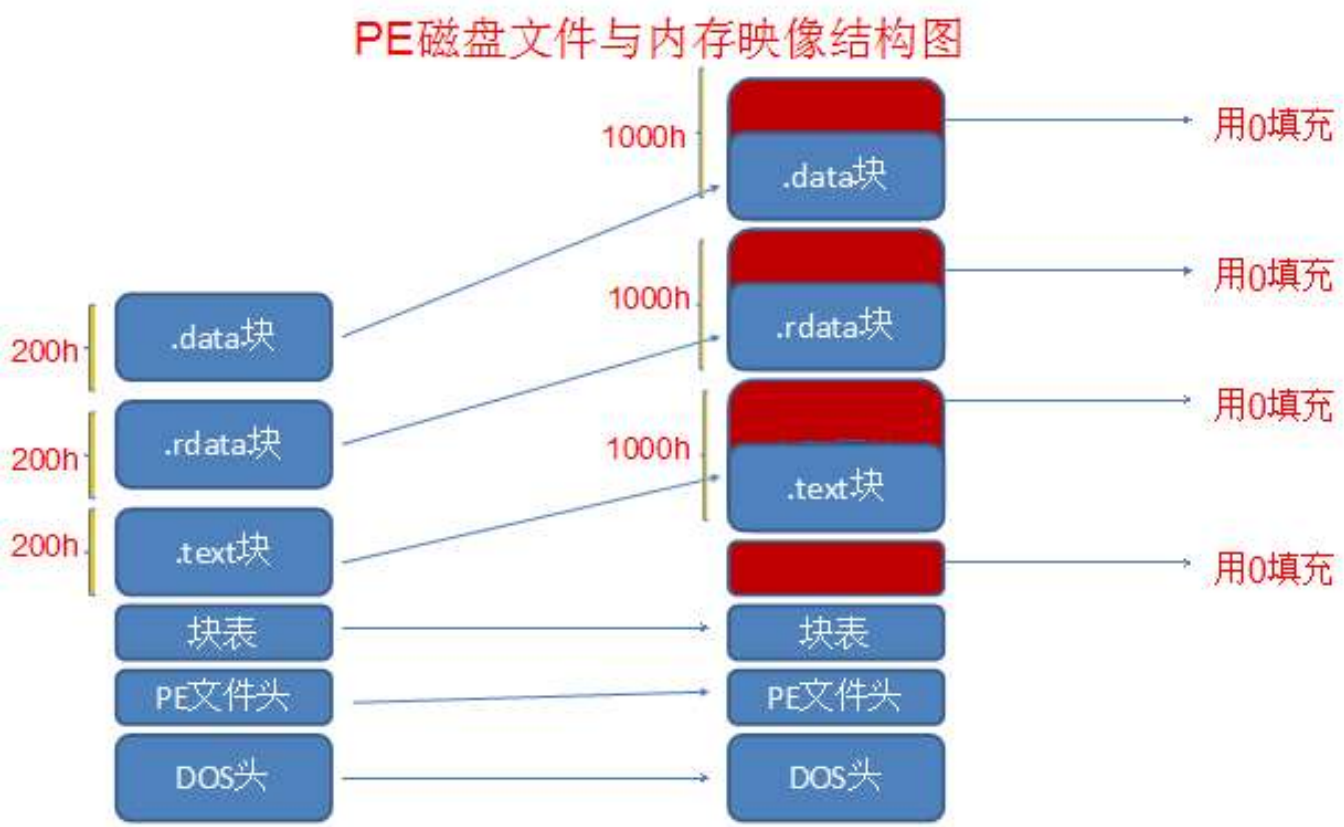
本节内容

RVA与FOA的转换

引出问题：

如果想改变一个全局变量的初始值，该怎么做?

1、面临的问题是什么？


PE磁盘文件与内存映像结构图
200h
.data块
200h
.rdata块
200h
.text块
块表
PE文件头
DOS头
1000h
.data块
1000h
.rdata块
1000h
.text块
块表
PE文件头
DOS头
用0填充
用0填充
用0填充
用0填充

2、RVA到FOA的转换：

<1> 得到RVA的值：内存地址 - ImageBase

<2> 判断RVA是否位于PE头中，如果是：FOA == RVA

<3> 判断RVA位于哪个节：

RVA >= 节.VirtualAddress
RVA <= 节.VirtualAddress + 当前节内存对齐后的大小

差值 = RVA - 节.VirtualAddress；

<4> FOA = 节.PointerToRawData + 差值；

课后练习：

<线上班>学员可见

本节内容

空白区添加代码

1、实现思路：

<1> 在PE的空白区构造一段代码

<2> 修改入口地址为新增代码

<3> 新增代码执行后，跳回入口地址

2、实现步骤：

<1> 获取MessageBox地址，构造ShellCode代码

<2> E8 E9计算公式

<3> 在空白区添加代码

<4> 修改OEP，指向ShellCode

<5> 指向完ShellCode后跳回OEP

课后练习：

<线上班>学员可见

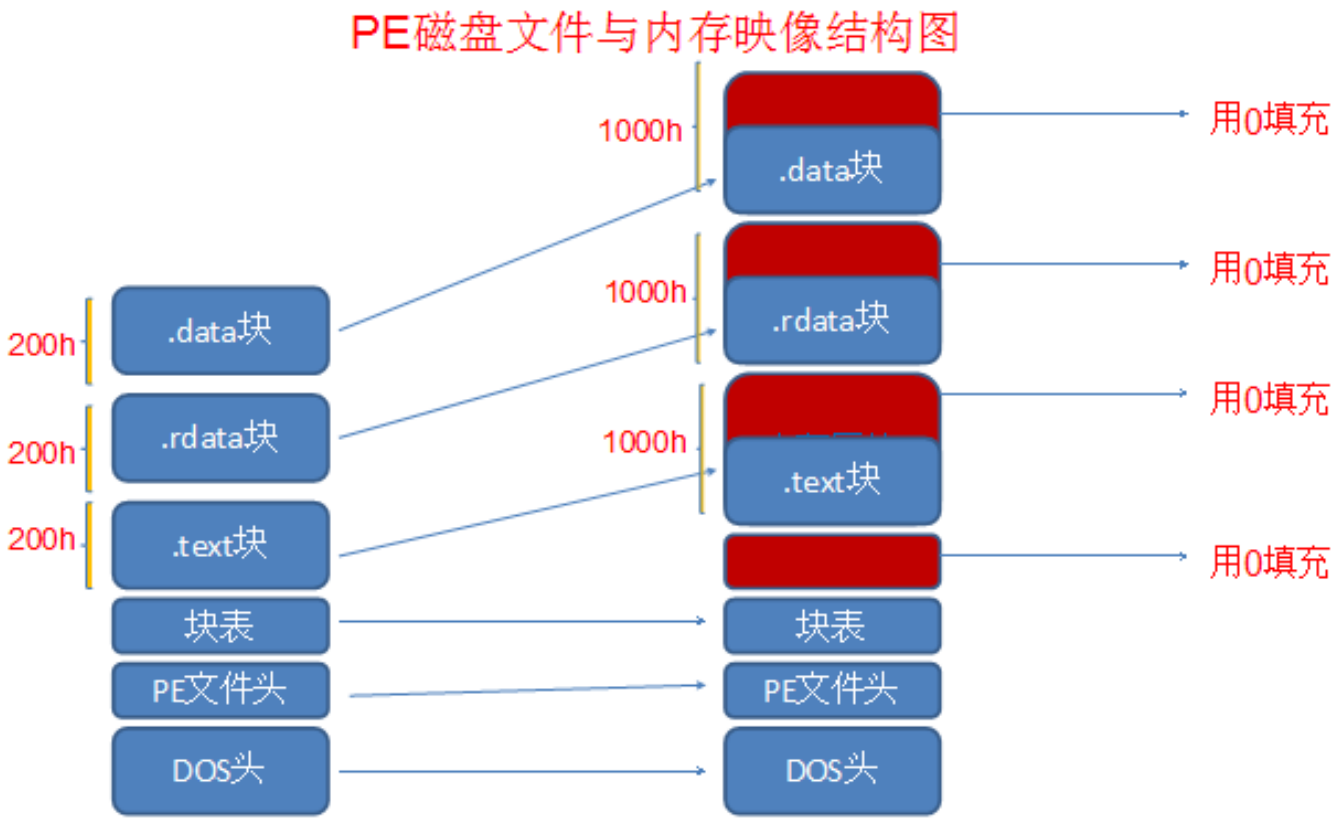
本节内容

扩大节

1、为什么要扩大节?

我们可以在任意空白区添加自己的代码，但如果添加的代码比较多，空白区不够怎么办?

2、扩大哪一个节呢？


PE磁盘文件与内存映像结构图
200h
.data块
200h
.rdata块
200h
.text块
块表
PE文件头
DOS头
1000h
1000h
1000h
.data块
.rdata块
.text块
块表
PE文件头
DOS头
用0填充
用0填充
用0填充
用0填充

3、节表数据结构说明

```
#define IMAGE_SIZEOF_SHORT_NAME 8
typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];   //ASCII字符串 可自定义  如果超8个系统只截取8个
    union {                                  //Misc  双字 是该节在没有对齐前的真实尺寸,该值可以不准确
            DWORD   PhysicalAddress;
            DWORD   VirtualSize;
    } Misc;
    DWORD   VirtualAddress;                  //在内存中的偏移地址,加上ImageBase才是在内存中的真正地址
    DWORD   SizeOfRawData;                   //节在文件中对齐后的尺寸
    DWORD   PointerToRawData;                //节区在文件中的偏移
    DWORD   PointerToRelocations;            //调试相关
    DWORD   PointerToLinenumbers;
    WORD    NumberOfRelocations;
    WORD    NumberOfLinenumbers;
    DWORD   Characteristics;                 //节的属性
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

4、扩大节的步骤：

<1> 分配一块新的空间，大小为S

<2> 将最后一个节的SizeOfRawData和VirtualSize改成N

N = (SizeOfRawData或者VirtualSize 内存对齐后的值) + S

<3> 修改SizeOfImage大小

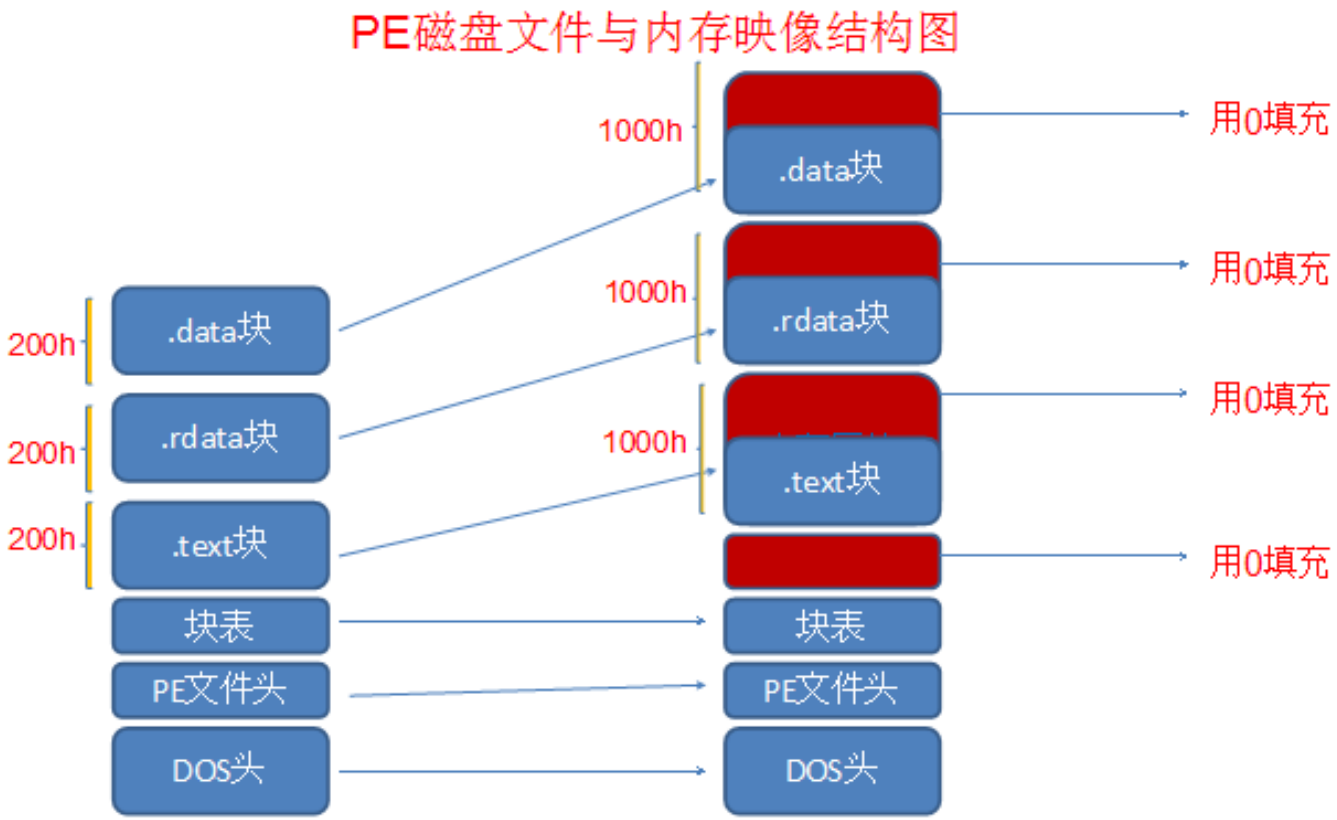
课后练习：

<线上班>学员可见

本节内容

新增节

1、节表与节


PE磁盘文件与内存映像结构图
200h
.data块
200h
.rdata块
200h
.text块
块表
PE文件头
DOS头
1000h
.data块
1000h
.rdata块
1000h
.text块
块表
PE文件头
DOS头
用0填充
用0填充
用0填充
用0填充

2、节表数据结构说明

```
#define IMAGE_SIZEOF_SHORT_NAME 8
typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];   //ASCII字符串 可自定义  如果超8个系统只截取8个
    union {                                  //Misc  双字 是该节在没有对齐前的真实尺寸,该值可以不准确
            DWORD   PhysicalAddress;
            DWORD   VirtualSize;
    } Misc;
    DWORD   VirtualAddress;                  //在内存中的偏移地址,加上ImageBase才是在内存中的真正地址
    DWORD   SizeOfRawData;                   //节在文件中对齐后的尺寸
    DWORD   PointerToRawData;                //节区在文件中的偏移
    DWORD   PointerToRelocations;            //调试相关
    DWORD   PointerToLinenumbers;
    WORD    NumberOfRelocations;
    WORD    NumberOfLinenumbers;
    DWORD   Characteristics;                 //节的属性
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

3、新增节的步骤：

<1> 判断是否有足够的空间，可以添加一个节表.

<2> 在节表中新增一个成员.

<3> 修改PE头中节的数量.

<4> 修改sizeOfImage的大小.

<5> 再原有数据的最后，新增一个节的数据(内存对齐的整数倍).

<6> 修正新增节表的属性.

课后练习：

<线上班>学员可见

本节内容

合并节

1、为什么要合并节?

如果节表没有地方插入成员了怎么办?

----打开notepad.exe观察节表

2、节表数据结构说明

```
#define IMAGE_SIZEOF_SHORT_NAME 8
typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];   //ASCII字符串 可自定义  如果超8个系统只截取8个
    union {                                  //Misc  双字 是该节在没有对齐前的真实尺寸,该值可以不准确
            DWORD   PhysicalAddress;
            DWORD   VirtualSize;
    } Misc;
    DWORD   VirtualAddress;                  //在内存中的偏移地址,加上ImageBase才是在内存中的真正地址
    DWORD   SizeOfRawData;                   //节在文件中对齐后的尺寸
    DWORD   PointerToRawData;                //节区在文件中的偏移

    DWORD   PointerToRelocations;            //调试相关
    DWORD   PointerToLinenumbers;
    WORD    NumberOfRelocations;
    WORD    NumberOfLinenumbers;
    DWORD   Characteristics;                 //节的属性
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

3、合并节的步骤：

<1> 按照内存对齐展开

<2> 将第一个节的内存大小、文件大小改成一样

Max = SizeOfRawData>VirtualSize?SizeOfRawData:VirtualSize

SizeOfRawData = VirtualSize =

最后一个节的VirtualAddress + Max - SizeOfHeaders内存对齐后的大小.

<3> 将第一个节的属性改为包含所有节的属性.

<4> 修改节的数量为1.

```
000003f0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000400h: 04 10 40 00 03 07 42 6F 6F 6C 65 61 6E 01 00 00 ; .@...Boolean...
00000410h: 00 00 01 00 00 00 00 10 40 00 05 46 61 6C 73 65 ; ........@..False
00000420h: 04 54 72 75 65 8D 40 00 2C 10 40 00 02 04 43 68 ; .True弟.,.@...Ch
00000430h: 61 72 01 00 00 00 00 FF 00 00 00 90 40 10 40 00 ; ar.....  ...怉.@.
00276000h: 00 00 00 00 00 00 00 00 00 00 00 00 02 8D 40 00 ; ..............弟.
00276010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00276020h: 00 00 00 00 32 13 8B C0 02 00 8B C0 00 8D 40 00 ; ....2.嬥..嬥.弟.
00276030h: 00 8D 40 00 00 8D 40 00 01 8D 40 00 00 00 00 00 ; .弟..弟..弟.....
002807f0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00280800h: 00 00 00 00 00 00 00 00 00 00 00 00 40 6B 28 00 ; ............@k(.
00280810h: F4 61 28 00 00 00 00 00 00 00 00 00 00 00 00 00 ; 關(.............
00280820h: A6 6E 28 00 B8 62 28 00 00 00 00 00 00 00 00 00 ;   (.廿(.........
00283c00h: 00 A0 68 00 10 A0 68 00 CC 70 67 00 10 B0 68 00 ; .燦..燦.蘑g..癵.
00283c10h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00283c20h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00283c30h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00283e00h: 00 10 00 00 70 01 00 00 00 30 16 30 28 30 3C 30 ; ....p....0.0(0<0
00283e10h: 54 30 6C 30 80 30 94 30 AC 30 C8 30 D8 30 E8 30 ; T0l0€0?????
00283e20h: F4 30 00 31 10 31 20 31 30 31 50 31 5C 31 60 31 ; ?.1.1 101P1\1`1
00283e30h: 64 31 68 31 6C 31 70 31 74 31 78 31 84 31 91 31 ; d1h1l1p1t1x1??
002a8a00h: 00 00 00 00 29 50 FE 3C 00 00 00 00 00 00 0A 00 ; ....)P?........
002a8a10h: 01 00 00 00 60 00 00 80 02 00 00 00 A8 00 00 80 ; ....`..€....?.€
002a8a20h: 03 00 00 00 B0 01 00 80 05 00 00 00 C8 01 00 80 ; ....?.€....?.€
002a8a30h: 06 00 00 00 E0 01 00 80 0A 00 00 00 50 03 00 80 ; ....?.€....P..€
002a8a40h: 0C 00 00 00 B8 06 00 80 0E 00 00 00 00 07 00 80 ; ....?.€.......€
```

本节内容

导出表

思考题：

一个可执行程序是由一个PE文件组成的?

1、如何定位导出表?

<参见课堂>

2、导出表结构

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;                // 未使用
    DWORD   TimeDateStamp;                  // 时间戳
    WORD    MajorVersion;                   // 未使用
    WORD    MinorVersion;                   // 未使用
    DWORD   Name;                           // 指向该导出表文件名字符串
    DWORD   Base;                           // 导出函数起始序号
    DWORD   NumberOfFunctions;              // 所有导出函数的个数
    DWORD   NumberOfNames;                  // 以函数名字导出的函数个数
    DWORD   AddressOfFunctions;             // 导出函数地址表RVA
    DWORD   AddressOfNames;                 // 导出函数名称表RVA
    DWORD   AddressOfNameOrdinals;          // 导出函数序号表RVA
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

3、导出

```
EXPORTS
Plus       @12
Sub        @15 NONAME
Mul        @13
Div        @16
```

课后练习：

<线上班>学员可见

本节内容

导入表：确定依赖模块

知识铺垫：

一个进程是由一组PE文件构成的：

PE文件提供哪些功能：导出表

PE文件需要依赖哪些模块以及依赖这些模块中的哪些函数：

导入表

1、如何定位导入表?

<参见课堂>

2、导入表结构

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;
        DWORD   OriginalFirstThunk;         //RVA 指向IMAGE_THUNK_DATA结构数组
    };
    DWORD   TimeDateStamp;                  //时间戳
    DWORD   ForwarderChain;
    DWORD   Name;                           //RVA,指向dll名字，该名字已0结尾
    DWORD   FirstThunk;                     //RVA,指向IMAGE_THUNK_DATA结构数组
} IMAGE_IMPORT_DESCRIPTOR;
```

课后练习：

<线上班>学员可见

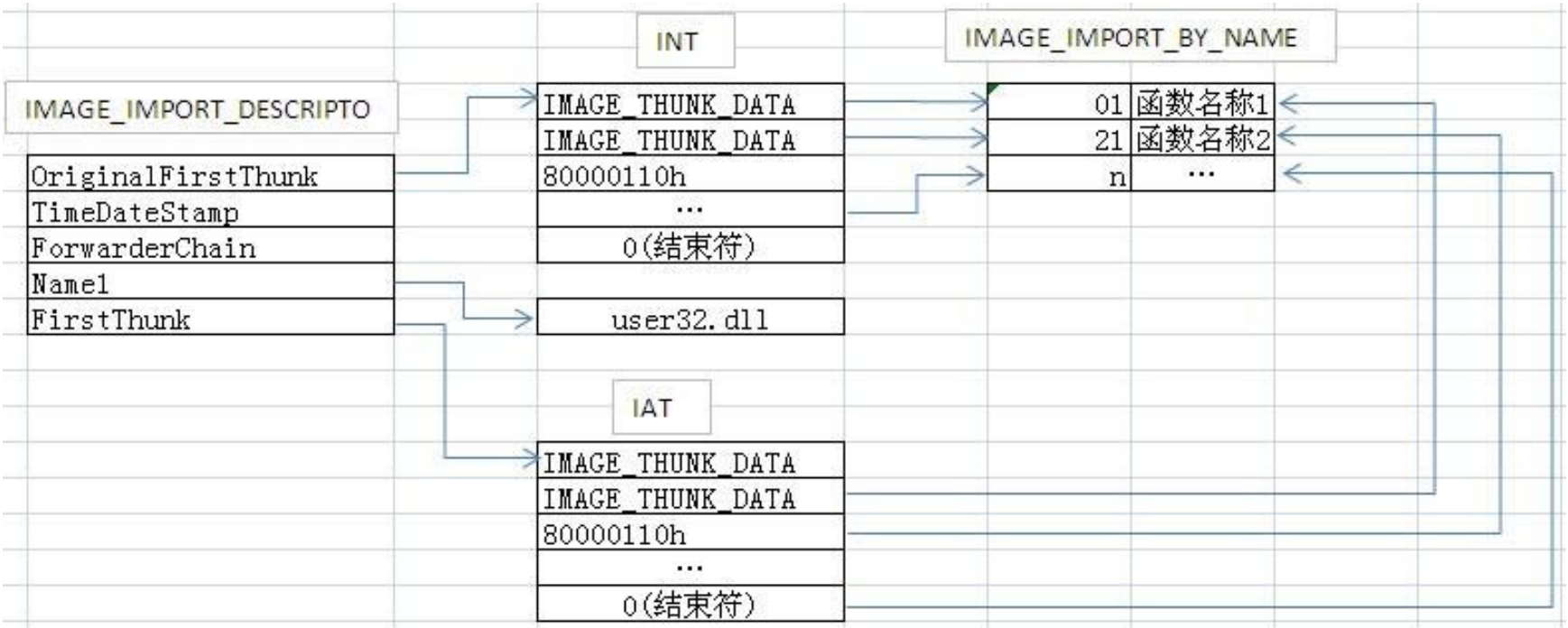
本节内容

导入表：确定依赖的函数

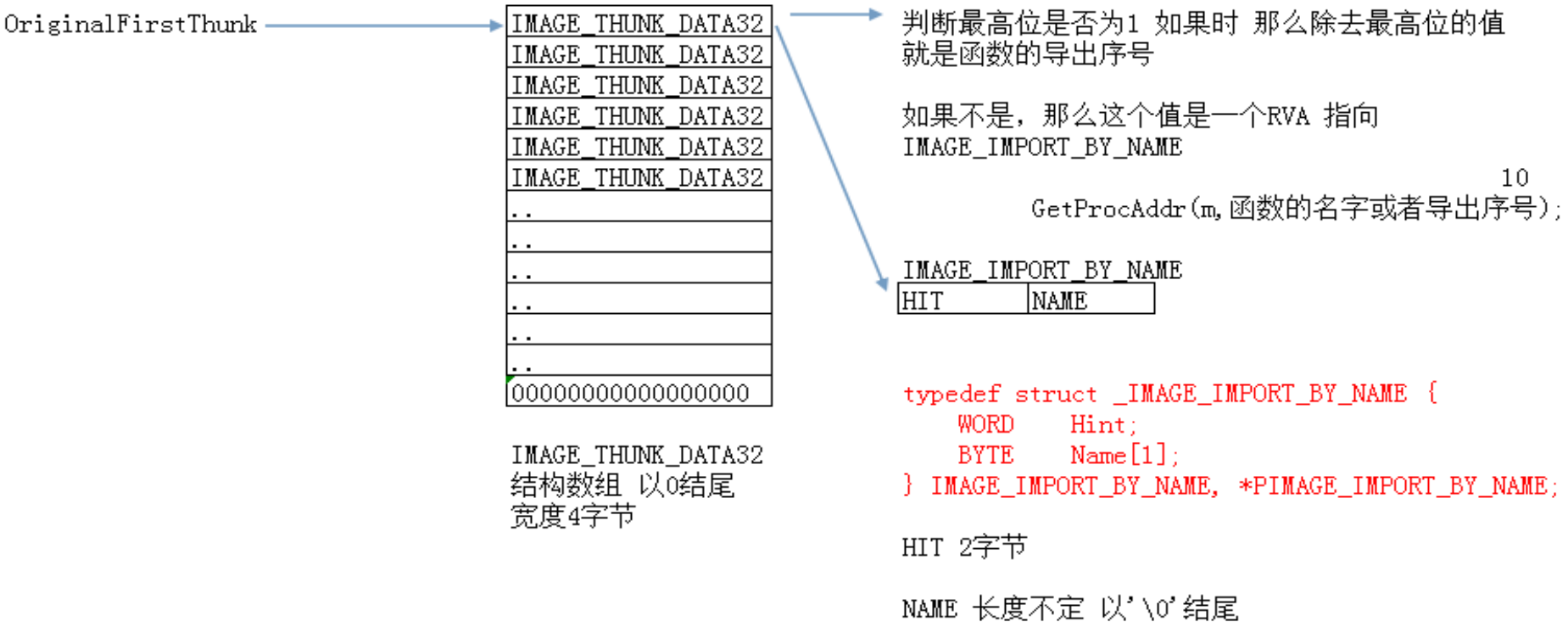
1、导入表结构

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;
        DWORD   OriginalFirstThunk;        //RVA 指向IMAGE_THUNK_DATA结构数组
    };
    DWORD   TimeDateStamp;                 //时间戳
    DWORD   ForwarderChain;
    DWORD   Name;                          //RVA,指向dll名字，该名字已0结尾
    DWORD   FirstThunk;                    //RVA,指向IMAGE_THUNK_DATA结构数组
} IMAGE_IMPORT_DESCRIPTOR;

typedef struct _IMAGE_THUNK_DATA32 {
    union {
        PBYTE  ForwarderString;
        PDWORD Function;
        DWORD Ordinal;                                  //序号
        PIMAGE_IMPORT_BY_NAME  AddressOfData;  //指向IMAGE_IMPORT_BY_NAME
    } u1;
} IMAGE_THUNK_DATA32;

typedef struct _IMAGE_IMPORT_BY_NAME {
    WORD    Hint;                                   //可能为空，编译器决定 如果不为空 是函数在导出表中的索引
    BYTE    Name[1];                                //函数名称，以0结尾
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

2、确定需要导入的函数：


INT
IMAGE_IMPORT_BY_NAME
IMAGE_IMPORT_DESCRIPTO
OriginalFirstThunk
TimeDateStamp
ForwarderChain
Name1
FirstThunk
IMAGE_THUNK_DATA
IMAGE_THUNK_DATA
80000110h
…
0(结束符)
user32.dll
01
函数名称1
21
函数名称2
n
…
IAT
IMAGE_THUNK_DATA
IMAGE_THUNK_DATA
80000110h
…
0(结束符)

3、确定需要导入的函数：


OriginalFirstThunk
IMAGE_THUNK_DATA32
IMAGE_THUNK_DATA32
IMAGE_THUNK_DATA32
IMAGE_THUNK_DATA32
IMAGE_THUNK_DATA32
IMAGE_THUNK_DATA32
..
..
..
..
..
..
00000000000000000
IMAGE_THUNK_DATA32
结构数组 以0结尾
宽度4字节
判断最高位是否为1 如果时 那么除去最高位的值
就是函数的导出序号
如果不是，那么这个值是一个RVA 指向
IMAGE_IMPORT_BY_NAME
10
GetProcAddr(m,函数的名字或者导出序号);
IMAGE_IMPORT_BY_NAME
HIT
NAME
typedef struct _IMAGE_IMPORT_BY_NAME {
WORD Hint;
BYTE Name[1];
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
HIT 2字节
NAME 长度不定 以'\0'结尾

课后练习：

<线上班>学员可见

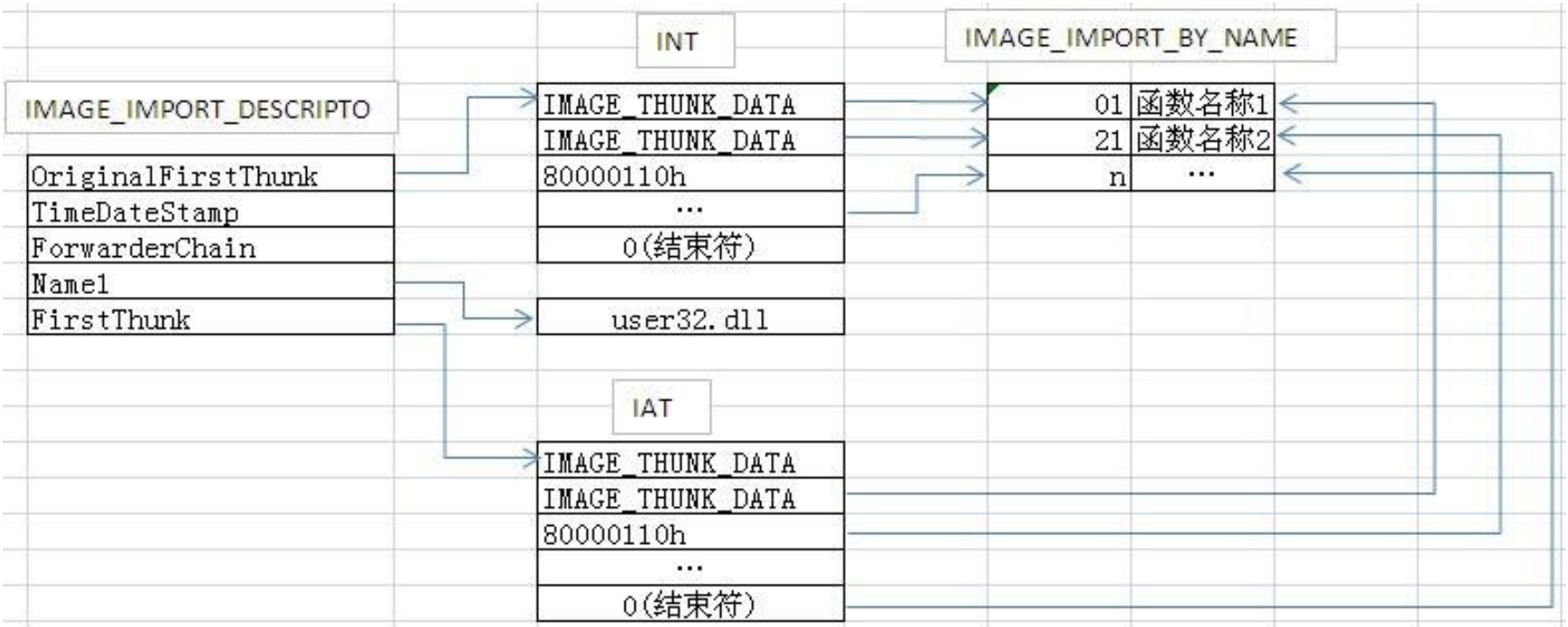
本节内容

导入表：确定函数地址

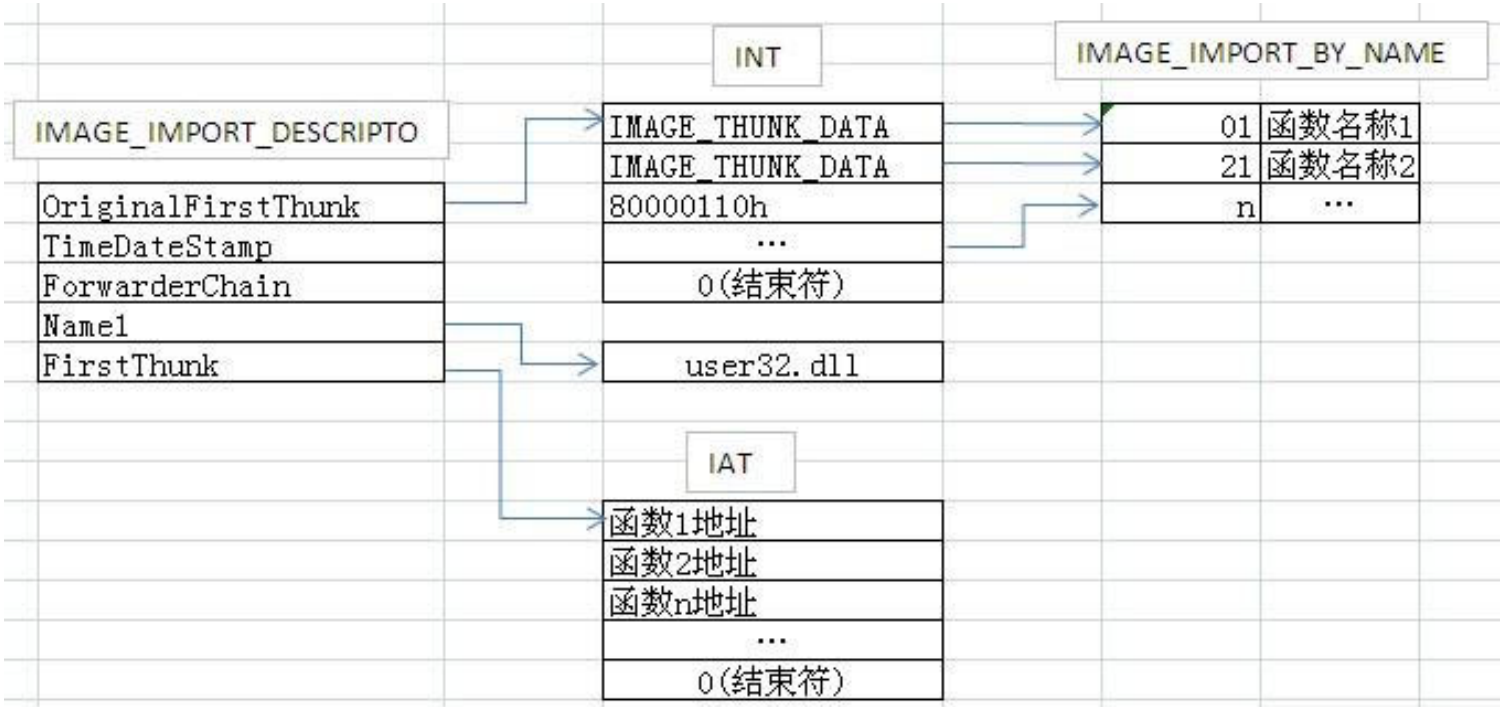
1、导入表结构

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;
        DWORD   OriginalFirstThunk;        //RVA 指向IMAGE_THUNK_DATA结构数组
    };
    DWORD   TimeDateStamp;                 //时间戳
    DWORD   ForwarderChain;
    DWORD   Name;                          //RVA,指向dll名字，该名字已0结尾
    DWORD   FirstThunk;                    //RVA,指向IMAGE_THUNK_DATA结构数组
} IMAGE_IMPORT_DESCRIPTOR;

typedef struct _IMAGE_THUNK_DATA32 {
    union {
        PBYTE  ForwarderString;
        PDWORD Function;
        DWORD Ordinal;                                  //序号
        PIMAGE_IMPORT_BY_NAME  AddressOfData;  //指向IMAGE_IMPORT_BY_NAME
    } u1;
} IMAGE_THUNK_DATA32;

typedef struct _IMAGE_IMPORT_BY_NAME {
    WORD    Hint;                                   //可能为空，编译器决定 如果不为空 是函数在导出表中的索引
    BYTE    Name[1];                                //函数名称，以0结尾
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

2、PE文件加载前：


INT
IMAGE_IMPORT_BY_NAME
IMAGE_IMPORT_DESCRIPTO
OriginalFirstThunk
TimeDateStamp
ForwarderChain
Name1
FirstThunk
IMAGE_THUNK_DATA
IMAGE_THUNK_DATA
80000110h
…
0(结束符)
user32.dll
01
函数名称1
21
函数名称2
n
…
IAT
IMAGE_THUNK_DATA
IMAGE_THUNK_DATA
80000110h
…
0(结束符)

3、PE文件加载后：


INT
IMAGE_IMPORT_BY_NAME
IMAGE_IMPORT_DESCRIPTO
IMAGE_THUNK_DATA
IMAGE_THUNK_DATA
80000110h
…
0(结束符)
01
函数名称1
21
函数名称2
n
…
OriginalFirstThunk
TimeDateStamp
ForwarderChain
Name1
FirstThunk
user32.dll
IAT
函数1地址
函数2地址
函数n地址
…
0(结束符)

课后练习：

<线上班>学员可见

本节内容

重定位表

1、PE文件的加载过程

<参见课堂>

2、为什么要用重定位表?

打开一个程序，观察一下全局变量的反汇编

3、重定位表的位置

数据目录项的第6个结构，就是重定位表.

4、重定位表的结构

```
typedef struct _IMAGE_BASE_RELOCATION {
    DWORD   VirtualAddress;
    DWORD   SizeOfBlock;
} IMAGE_BASE_RELOCATION;
typedef IMAGE_BASE_RELOCATION，* PIMAGE_BASE_RELOCATION;
```

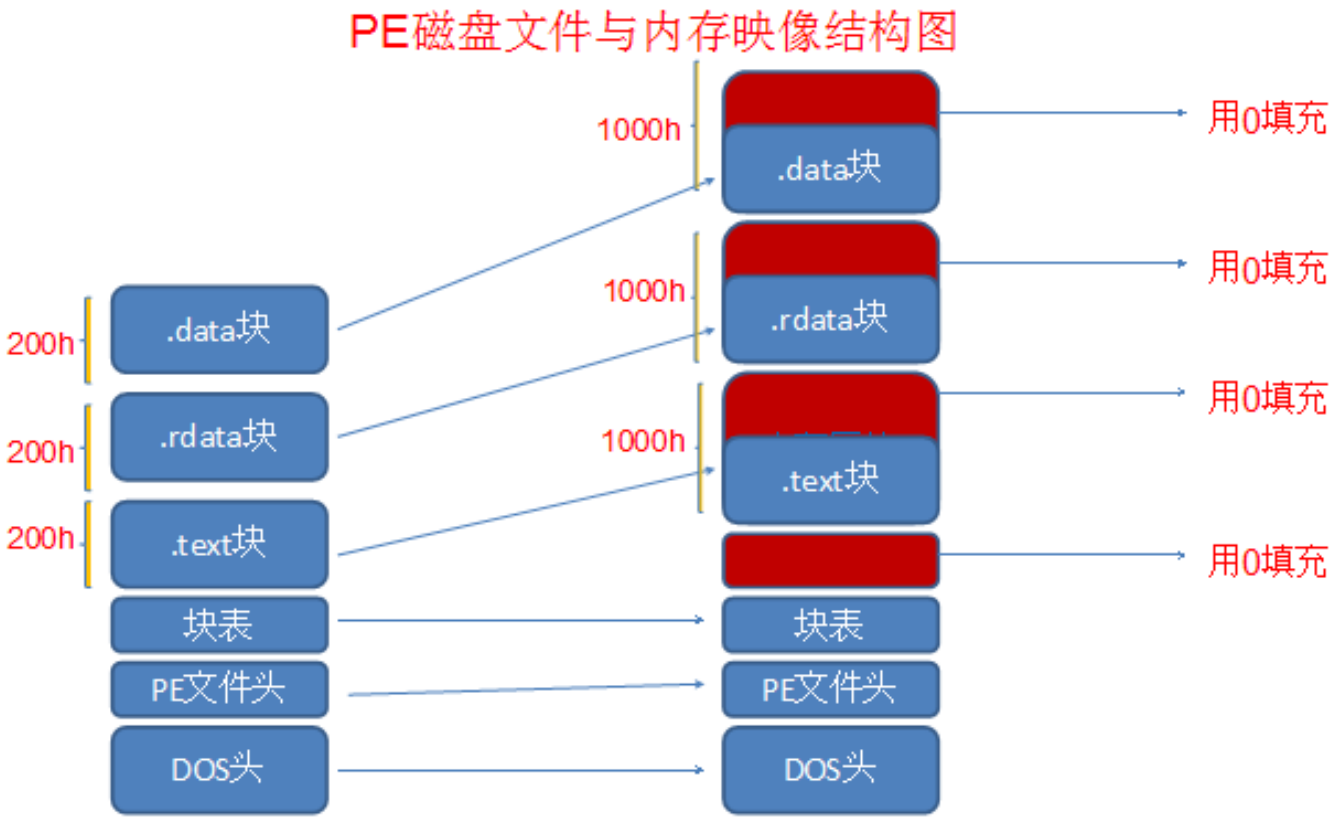
课后练习：

<线上班>学员可见

本节内容

PE头属性说明

1、PE文件的分节结构


PE磁盘文件与内存映像结构图
1000h
.data块
用0填充
1000h
.rdata块
用0填充
1000h
.text块
用0填充
用0填充
200h
.data块
200h
.rdata块
200h
.text块
块表
PE文件头
DOS头
块表
PE文件头
DOS头

2、节表数据结构说明

```
#define IMAGE_SIZEOF_SHORT_NAME 8
typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];   //ASCII字符串 可自定义  如果超8个系统只截取8个
    union {                                  //Misc  双字 是该节在没有对齐前的真实尺寸,该值可以不准确
            DWORD   PhysicalAddress;
            DWORD   VirtualSize;
    } Misc;
    DWORD   VirtualAddress;                  //在内存中的偏移地址,加上ImageBase才是在内存中的真正地址
    DWORD   SizeOfRawData;                   //节在文件中对齐后的尺寸
    DWORD   PointerToRawData;                //节区在文件中的偏移
    DWORD   PointerToRelocations;            //调试相关
    DWORD   PointerToLinenumbers;
    WORD    NumberOfRelocations;
    WORD    NumberOfLinenumbers;
    DWORD   Characteristics;                 //节的属性
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

3、节属性说明

数据位	常量符号	位为1时的含义
5	IMAGE_SCN_CNT_CODE 或 00000020h	节中包含代码
6	IMAGE_SCN_CNT_INITIALIZED_DATA 或 00000040h	节中包含已初始化数据
7	IMAGE_SCN_CNT_UNINITIALIZED_DATA 或 00000080h	节中包含未初始化数据
8	IMAGE_SCN_LNK_OTHER 或 00000100h	保留供将来使用
25	IMAGE_SCN_MEM_DISCARDABLE 或 02000000h	节中的数据在进程开始以后将被丢弃，如 .reloc
26	IMAGE_SCN_MEM_NOT_CACHED 或 04000000h	节中的数据不会经过缓存
27	IMAGE_SCN_MEM_NOT_PAGED 或 08000000h	节中的数据不会被交换到磁盘
28	IMAGE_SCN_MEM_SHARED 或 10000000h	表示节中的数据将被不同的进程所共享
29	IMAGE_SCN_MEM_EXECUTE 或 20000000h	映射到内存后的页面包含可执行属性
30	IMAGE_SCN_MEM_READ 或 40000000h	映射到内存后的页面包含可读属性
31	IMAGE_SCN_MEM_WRITE 或 80000000h	映射到内存后的页面包含可写属性

课后练习：

<线上班>学员可见

本节内容

Virtual Table Hook

1、什么是HOOK?

<1> HOOK是用来获取、更改程序执行时的某些数据，或者是用于更改程序执行流程的一种技术。

<2> HOOK的两种主要形式：

1、改函数代码

INLINE HOOK

2、改函数地址

IAT HOOK

SSDT HOOK

IDT HOOK

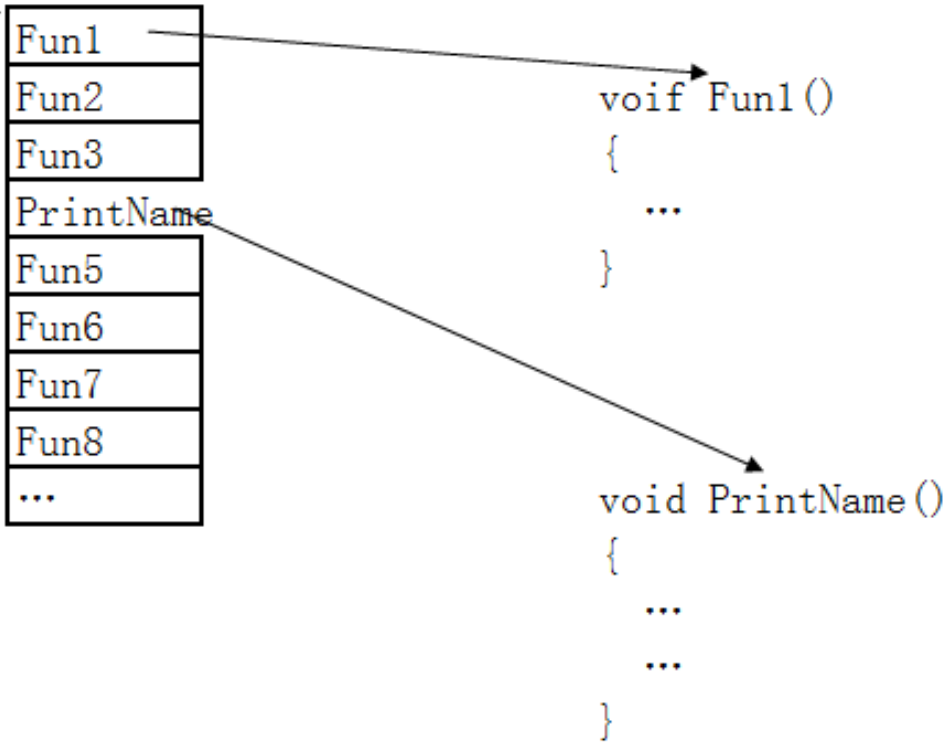
EAT HOOK

IRP HOOK

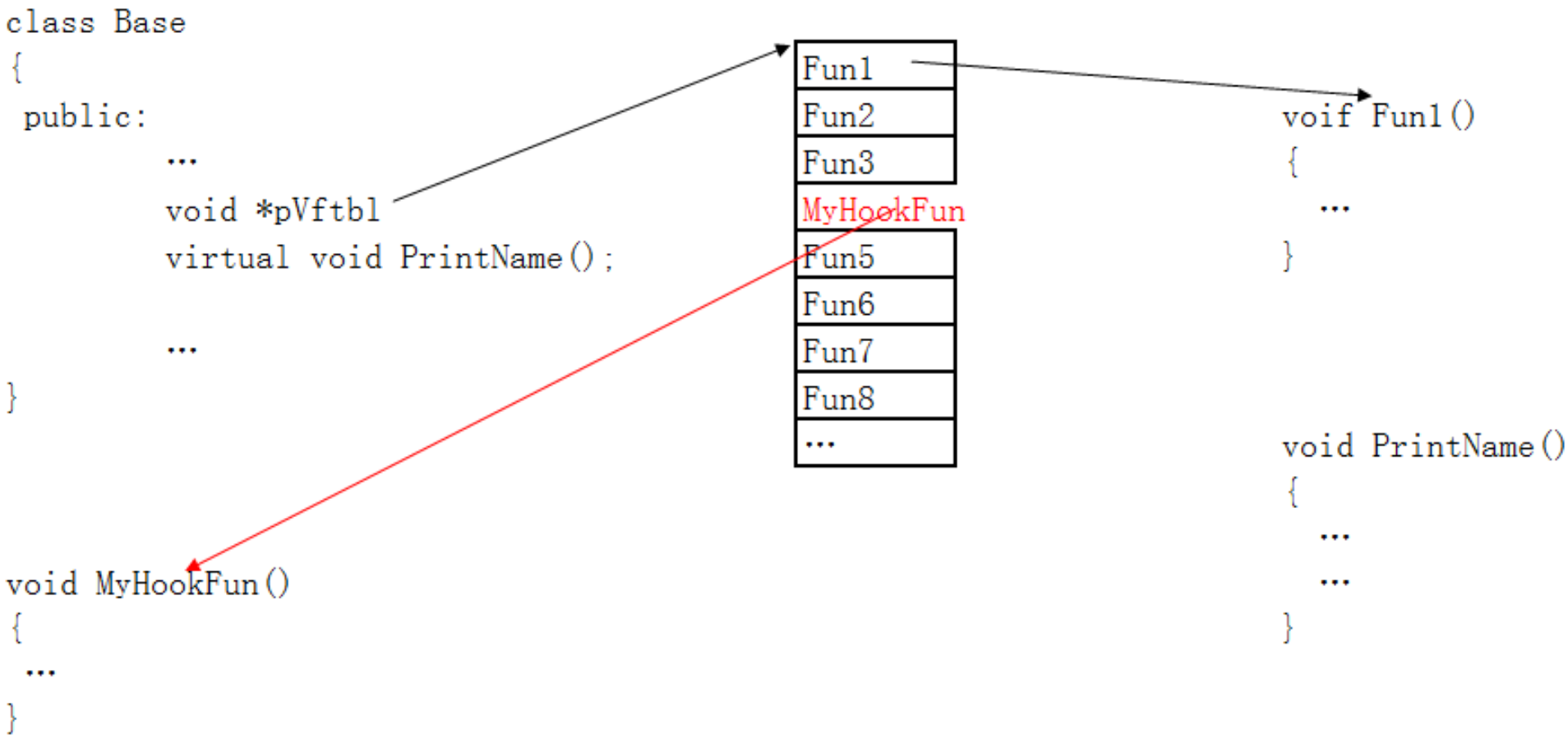
....

2、Virtual Table 是什么？

```
class Base
{
 public:
        …
        void *pVftbl
        virtual void PrintName();

        …
}
```



3、Virtual Table Hook 是什么？



课后练习：

<线上班>学员可见

本节内容

IAT HOOK

1、什么是HOOK?

<1> HOOK是用来获取、更改程序执行时的某些数据，或者是用于更改程序执行流程的一种技术。

<2> HOOK的两种主要形式：

1、改函数代码

INLINE HOOK

2、改函数地址

IAT HOOK

SSDT HOOK

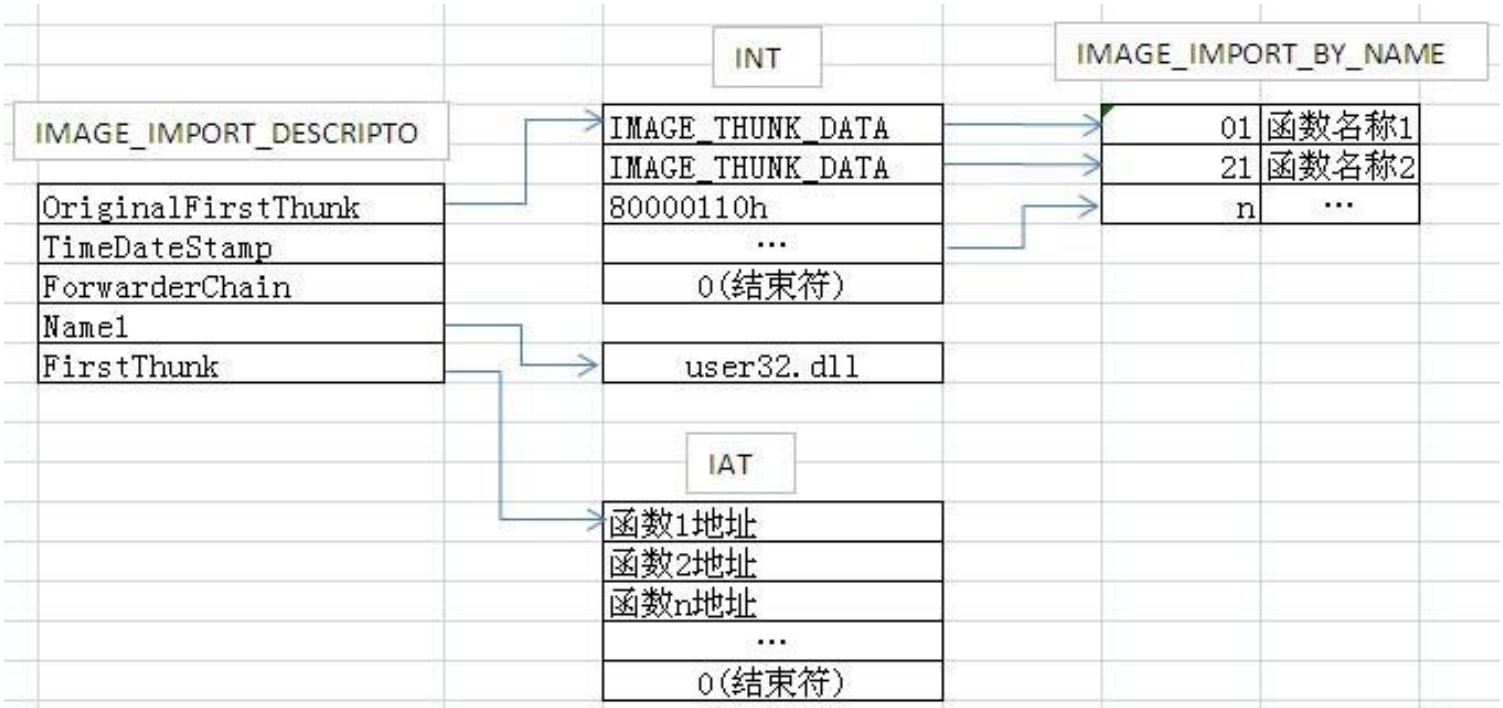
IDT HOOK

EAT HOOK

IRP HOOK

....

2、IAT表是什么?


INT
IMAGE_IMPORT_BY_NAME
IMAGE_IMPORT_DESCRIPTO
OriginalFirstThunk
TimeDateStamp
ForwarderChain
Name1
FirstThunk
IMAGE_THUNK_DATA
IMAGE_THUNK_DATA
80000110h
…
0(结束符)
01
函数名称1
21
函数名称2
n
…
user32.dll
IAT
函数1地址
函数2地址
函数n地址
…
0(结束符)

课后练习：

<线上班>学员可见

本节内容

INLINE HOOK

1、HOOK的两种主要形式：

<1> 改函数代码

INLINE HOOK

<2> 改函数地址

IAT HOOK

SSDT HOOK

IDT HOOK

EAT HOOK

IRP HOOK

....

2、IAT HOOK的缺点：

<1> 容易被检测到

<2> 只能HOOK IAT表里面的函数

3、INLINE HOOK的执行流程：

<参见课堂>

课后练习：

<线上班>学员可见

本节内容

INLINE HOOK改进版

1、下面代码的问题：

```
void __declspec(naked) NewMessageBox()
{
		_asm
		{
				//1. 保存寄存器
				pushad
				pushfd
				//2. 修改数据：esp+8
				LEA EAX,DWORD PTR DS:[szNewText]
				MOV DWORD PTR SS:[esp+0x24+8],EAX
				//3. 恢复寄存器
				popfd
				popad
				//4. 执行覆盖的代码
				MOV EDI,EDI
				PUSH EBP
				MOV EBP,ESP
				//5. 返回执行
				push dwHookAddress
				add dword ptr ds:[esp], PATCH_LENGTH
				retn
		}
}
```

2、另一种INLINE HOOK：

<1> 很多时候，防守的一方都会通过检测E9来判断自己的程序是否被HOOK了

<2> 将JMP.....JMP改为CALL + RET的方式实现

课后练习：

<线上班>学员可见

本节内容

HOOK攻防

HOOK攻防的常用手段：

阶段一：
（防）检测JMP(E9)、检测跳转范围
（破）想方设法绕

阶段二：
（防）写个线程全代码校验/CRC校验
（破）修改检测代码、挂起检测线程

阶段三. A->B->C->D->HOOK 函数
（防）先对相关API全代码校验,多个线程互相检测,并检测线程是否在活动中
（破）使用瞬时钩子/硬件钩子

课后练习：

<线上班>学员可见

本节内容

瞬时HOOK过检测

1、HOOK检测：

阶段一：
（防）检测JMP(E9)、检测跳转范围
（破）想方设法绕

阶段二：
（防）写个线程全代码校验/CRC校验
（破）修改检测代码、挂起检测线程

阶段三： A->B->C->D->HOOK 函数
（防）先对相关API全代码校验,多个线程互相检测,并检测线程是否在活动中
（破）使用瞬时钩子/硬件钩子

2、循环检测：

```
for (;;)
{
        Sleep(500);
//1.    //检测ExitProcess是否被HOOK
//2.    //检测被保护的函数是否被HOOK
        if (memcmp((LPVOID)dwAPIAddr,szAPICode,0x30)!=0)
        {
//A              bRet = VirtualProtect((LPVOID)dwAPIAddr,0x10,PAGE_READWRITE,&dwOldProtect);
                f (bRet)
                {
                        memcpy((LPVOID)dwAPIAddr,szAPICode,0x30);
                        VirtualProtect((LPVOID)dwAPIAddr,0x10,dwOldProtect,&dwOldProtect);
                }
//B              ExitProcess(0);
        }
}
```

3、解决方案：

<1> 对VirtualProtect进行HOOK，只有当调用地址为A时,对ExitProcess进行挂钩,并修正返回结果，改变执行流程。

<2> 在ExitProcess的HOOK处理函数中判断，只有当调用地址为B的时候，才将ExitProcess失效，然后卸载HOOK。

课后练习：

<线上班>学员可见