

# 课后作业

## ◆ 简述JNZ和JZ指令的用法及区别。

- ▶▶ **JNZ**: 条件转移指令，根据**ZF**决定是否跳转。结果不为零（或不相等）则转移，此时**ZF=0**。
- ▶▶ **JZ**: 条件转移指令，根据**ZF**决定是否跳转。结果为零（或相等）则转移，此时**ZF=1**。

## ◆ 简述TEST指令的用法。

- ▶▶ **Test**命令将两个操作数进行逻辑与运算，并根据运算结果设置相关的标志位。结果是**0**，**ZF**标志位为**1**；结果不是**0**，**ZF**为**0**。
  - 若两操作数相等，则**TEST**只有在其为**0**的情况下结果才为**0**
- ▶▶ **Test**命令的两个操作数不会被改变。运算结果在设置过相关标记位后会被丢弃。

# 课后作业

## ◆ 简述MOVSX指令的用法。

- ▶▶ **MOVSX**是带符号扩展传送指令。符号扩展的意思是：当计算机存储某一个有符号数时，符号位位于该数的第一位，所以，当扩展一个负数的时候需要将扩展的高位全赋为1；对于正数而言，符号扩展和零扩展**MOVZX**是一样的，将扩展的高位全赋为0。

## ◆ 简述进行程序爆破的思路（如何定位要爆破的指令？定位后如何爆破？）。

- ▶▶ 运行程序，找到相关提示字符串；
- ▶▶ 用分析工具（如**OlllyDbg**等）在代码中找到对应字符串；
- ▶▶ 跟踪程序中使用字符串的指令；
- ▶▶ 分析（附近）指令序列的逻辑，利用断点确认判断指令；
- ▶▶ 用**NOP**指令填充该指令；
- ▶▶ 测试运行

# 课后作业

- ◆ 在“面向汇编的逐句解析”的示例中，有一条注册码判断指令为`cmp byte ptr ss:[esp+0x9],0x2D`。请解释：
  - ▶▶ a) 这条指令是如何访问内存的；
    - `byte ptr`，以字节为单位进行访问
    - `ss:[esp+0x9]`，`ss`为栈基址，`esp`为栈顶，`0x9`为偏移量
  - ▶▶ b) 这条指令的含义是什么。
    - 取出栈顶`0x9`偏移量处的内存的一个字节，将其内容与`0x2D`进行对比
    - 具体含义需要结合上下文

# 课后作业

## ◆ 简述MD5算法的流程。

- ▶▶ 首先要对需加密消息进行数据填充，使消息的长度对512取模后等于448，消息长度  $\text{mod } 512 = 448$ 。不足位数在消息（字符串）后面进行填充，填充第一位为1，其余为0。即第一个字节为0x80,其余字节为0x00。
- ▶▶ 再填充上原消息的长度，例如欲加密字符为16个ASCII，即16字节 $\times 8$ 位=128位，128即16进制的0x80。但它是一个QWORD值，占用8个字节(64位)。在此步骤进行完毕后，最终消息长度就是512的整数倍。(448 + 64 = 512)
- ▶▶ 4个种子值：A = 0x67452301, B = 0xEFCDAB89, C = 0x98BADCFE, D = 0x10325476;  
4个函数：F(X,Y,Z)=(X & Y) | ((~X) & Z); G(X,Y,Z)=(X & Z) | (Y & (~Z)); H(X,Y,Z)=X ^ Y ^ Z; I(X,Y,Z)=Y ^ (X | (~Z));  
把消息分以512位为一分组进行处理，每一个分组进行4轮变换，以上面所说4个常数为起始变量进行计算，重新输出4个变量，以这4个变量再进行下一分组的运算，如果已经是最后一个分组，则这4个变量为最后的结果，即MD5值。



# 可执行文件格式

# 本次课程支撑的毕业要求指标点

## ◆ 毕业要求3-3:

充分理解信息安全领域软硬件系统的基础上，能够设计或开发满足特定需求和约束条件的信息安全系统、模块或算法流程，并能够进行系统级优化。

▶▶ 紧跟映像文件头后面的是可选映像头-是必须的！

```
typedef struct _IMAGE_OPTIONAL_HEADER {  
    // 标准域:  
    //  
    WORD    Magic;                // 0x18, 一般是0x010B  
    BYTE    MajorLinkerVersion;   // 0x1a, 链接器的主/次版本号,  
    BYTE    MinorLinkerVersion;  // 0x1b, 这两个值都不可靠  
    DWORD    SizeOfCode;          // 0x1c, 可执行代码的长度  
    DWORD    SizeOfInitializedData; // 0x20, 初始化数据的长度(数据节)  
    DWORD    SizeOfUninitializedData; // 0x24, 未初始化数据的长度(bss节)  
    DWORD    AddressOfEntryPoint; // 0x28, 代码的入口RVA地址, 程序从这开始执行  
    DWORD    BaseOfCode;          // 0x2c, 可执行代码起始位置, 意义不大  
    DWORD    BaseOfData;          // 0x30, 初始化数据起始位置, 意义不大  
    // NT 附加域:  
    //  
    DWORD    ImageBase;           // 0x34, 载入程序首选的VA地址  
    DWORD    SectionAlignment;    // 0x38, 加载后节在内存中的对齐方式-节的大小  
    DWORD    FileAlignment;       // 0x3c, 节在文件中的对齐方式-节的大小  
};
```

(待续)

# EXE文件的格式

## PE文件格式

(续前)

```
WORD    MajorOperatingSystemVersion; // 0x3e, 操作系统主/次版本,
WORD    MinorOperatingSystemVersion; // 0x40, Loader并没有用这两个值
WORD    MajorImageVersion;           // 0x42, 可执行文件主/次版本
WORD    MinorImageVersion;           // 0x44
WORD    MajorSubsystemVersion;       // 0x46, 子系统版本号
WORD    MinorSubsystemVersion;       // 0x48
DWORD    Win32VersionValue;          // 0x4c, Win32版本, 一般是0
DWORD    SizeOfImage;               // 0x50, 程序调入后占用内存大小(字节)
DWORD    SizeOfHeaders;            // 0x54, 文件头的长度之和
DWORD    Checksum;                 // 0x58, 校验和
WORD     Subsystem;                // 0x5c, 可执行文件的子系统GUI或CUI
WORD     DllCharacteristics;         // 0x5e, 何时DllMain被调用, 一般为0
DWORD    SizeOfStackReserve;         // 0x60, 初始化线程时保留的堆栈大小
DWORD    SizeOfStackCommit;         // 0x64, 初始化线程时提交的堆栈大小
DWORD    SizeOfHeapReserve;          // 0x68, 进程初始化时保留的堆大小
DWORD    SizeOfHeapCommit;           // 0x6c, 进程初始化时提交的堆大小
DWORD    LoaderFlags;                // 0x70, 装载标志, 与调试相关
DWORD    NumberOfRvaAndSizes;        // 0x74, 数据目录的项数, 一般是16
IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```



# EXE文件的格式

## DataDirectory: 数据目录表

```
typedef struct  
_IMAGE_DATA_DIRECTORY {  
    DWORD VirtualAddress;  
    DWORD Size;  
} IMAGE_DATA_DIRECTORY,  
*PIMAGE_DATA_DIRECTORY;
```

- ◆ 是一个 **IMAGE\_DATA\_DIRECTORY** 结构数组，有16个这样的元素。
- ◆ 数据目录表的每个结构给出一个重要数据结构的起始RVA和大小信息。
- ◆ 如果把节表看作是PE文件各节的根目录，那也可以认为数据目录表是存储在这些节里的逻辑元素的根目录。

目录表查看器

	RVA	大小
输出表:	00000000	00000000
输入表:	000073A4	000000B4
资源:	0002D000	00004348
例外:	00000000	00000000
安全性:	00089BB8	00001558
基址重定位:	00000000	00000000
调试:	00000000	00000000
版权:	00000000	00000000
全局指针:	00000000	00000000
TLS 表:	00000000	00000000
载入配置:	00000000	00000000
输入表范围:	00000000	00000000
输入地址表:	00007000	0000028C

# EXE文件的格式

## 重要数据结构

- ◆ 什么重要数据结构？  
如：引入函数表 (import table)  
一个引入函数是被某模块调用但又不在于调用模块中的函数，位于一个或者更多的DLL里，因而要保留一些函数信息，包括函数名及其驻留的DLL名。
- ◆ 怎么样获得PE文件中重要数据结构？

目录表查看器

	RVA	大小
输出表: Export Table:		00000000
输入表: Import Table:		000000B4
资源: Recourse:		00004348
例外: Exception:		00000000
安全性: Security:		00001558
基址重定位: Baserelease:		00000000
调试: Debug:		00000000
版权: Copyright:		00000000
全局指针: Globalptr:		00000000
TLS 表: Tls Table:		00000000
载入配置: Load Config:		00000000
输入表范围: Bound Import:		00000000
输入地址表: Import Address Table:		0000028C

# EXE文件的格式

## 怎么样获得PE文件中重要数据结构？

1. 从 DOS header 定位到 PE header
2. 从 optional header 读取 data directory 的地址。
3. IMAGE\_DATA\_DIRECTORY 结构尺寸  
  乘上找寻结构的索引号。例如，欲获取import table的位置信息，必须用  
  IMAGE\_DATA\_DIRECTORY 结构尺寸(8 bytes)乘上1（import table在  
  data directory中的索引号）。
4. 将上面的结果加上data directory地址，  
  就得到包含所查询数据结构信息的  
  IMAGE\_DATA\_DIRECTORY 结构项

目录表查看器

	RVA	大小
输出表:	Export Table:	00000000
输入表:	Import Table:	00000B4
资源:	Resource:	0004348
例外:	Exception:	00000000
安全性:	Security:	0001558
基址重定位:	BaseReloc:	00000000
调试:	Debug:	00000000
版权:	Copyright:	00000000
全局指针:	Globalptr:	00000000
TLS 表:	Tls Table:	00000000
载入配置:	Load Config:	00000000
输入表范围:	Bound Import:	00000000
输入地址表:	Import Address Table:	000028C

# EXE文件的格式

## PE文件格式

- ◆ 节表是紧挨着NT映像头的一结构数组，其成员的数目由映像头中NumberOfSections决定

```
#define IMAGE_SIZEOF_SHORT_NAME 8
typedef struct _IMAGE_SECTION_HEADER {
    UCHAR Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        ULONG PhysicalAddress;
        ULONG VirtualSize;
    } Misc;
    ULONG VirtualAddress;
    ULONG SizeOfRawData;
    ULONG PointerToRawData;
    ULONG PointerToRelocations;

    ULONG PointerToLinenumbers;
    USHORT NumberOfRelocations;
    USHORT NumberOfLinenumbers;
    ULONG Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

本节的实际字节数 如388H字节

本节的相对虚拟地址 如为1000H，  
而PE文件装载基地址400000H，？

// OBJ文件中表示本节物理地址

// EXE文件中表示节的实际字节数

// 本节的RVA

// 本节经过文件对齐后的尺寸

// 本节原始数据在文件中的位置

// OBJ文件中表示本节重定位信

// 息的偏移，EXE文件中无意义

// 行号偏移

// 本节需重定位的数目

// 本节在行号表中的行号数目

// 节属性

# EXE文件的格式

## PE文件格式

- ◆ 节表是紧挨着NT映像头的一结构数组，其成员的数目由映像文件头中NumberOfSections决定

```
#define IMAGE_SIZEOF_SHORT_NAME 8
typedef struct _IMAGE_SECTION_HEADER {
    UCHAR Name[IMAGE_SIZEOF_SHORT_NAME]; // 节名
    union {
        ULONG PhysicalAddress;
        ULONG VirtualSize;
    } Misc;
    ULONG VirtualAddress; // 本节的RVA
    ULONG SizeOfRawData; // 本节经过文件对齐后的尺寸
    ULONG PointerToRawData; // 本节原始数据在文件中的位置
    ULONG PointerToRelocations; // OBJ文件中表示本节重定位信息的偏移，EXE文件中无意义
    ULONG PointerToLinenumbers; // 行号偏移
    USHORT NumberOfRelocations; // 本节需重定位的数目
    USHORT NumberOfLinenumbers; // 本节在行号表中的行号数目
    ULONG Characteristics; // 节属性
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

PE装载器通过本域找到节的位置

## PE文件格式

### ◆ 节

- ▶▶ PE文件的真正内容划分成块，称之为**Section(节)**，紧跟在节表之后
- ▶▶ 每个节是一块拥有共同属性的数据，比如代码/数据、读/写等
- ▶▶ 可以把PE文件想象成一逻辑磁盘，**PE header**是磁盘的**Boot**扇区，节表就是根目录，而**Section**就是各种文件，每种文件自然就有不同属性如只读、系统、隐藏、文档等等
- ▶▶ 节的划分是基于各组数据的共同属性而不是逻辑概念——如果PE文件中的数据/代码拥有相同属性，它们就能被归入同一节中
- ▶▶ 节名称仅仅是个区别不同节的符号而已，类似“**data**”、“**code**”的命名只为了便于识别，惟有节的属性设置决定了节的特性和功能



## PE文件格式

### ▶▶ 代码节.text

- **Windows NT**默认的做法是将所有的可执行代码组成了一个单独的节，名为“**.text**”或“**.code**”

### ▶▶ 引入函数节.idata

- 包含有从其它**DLL**中引入的函数
- 该节开始是一个成员为**IMAGE\_IMPORT\_DESCRIPTOR**结构的结构数组，也叫引入表，数据目录表表项结构成员**VirtualAddress**包含引入表地址
- 引入函数节可能被病毒用来直接获取**API**函数地址

# EXE文件的格式

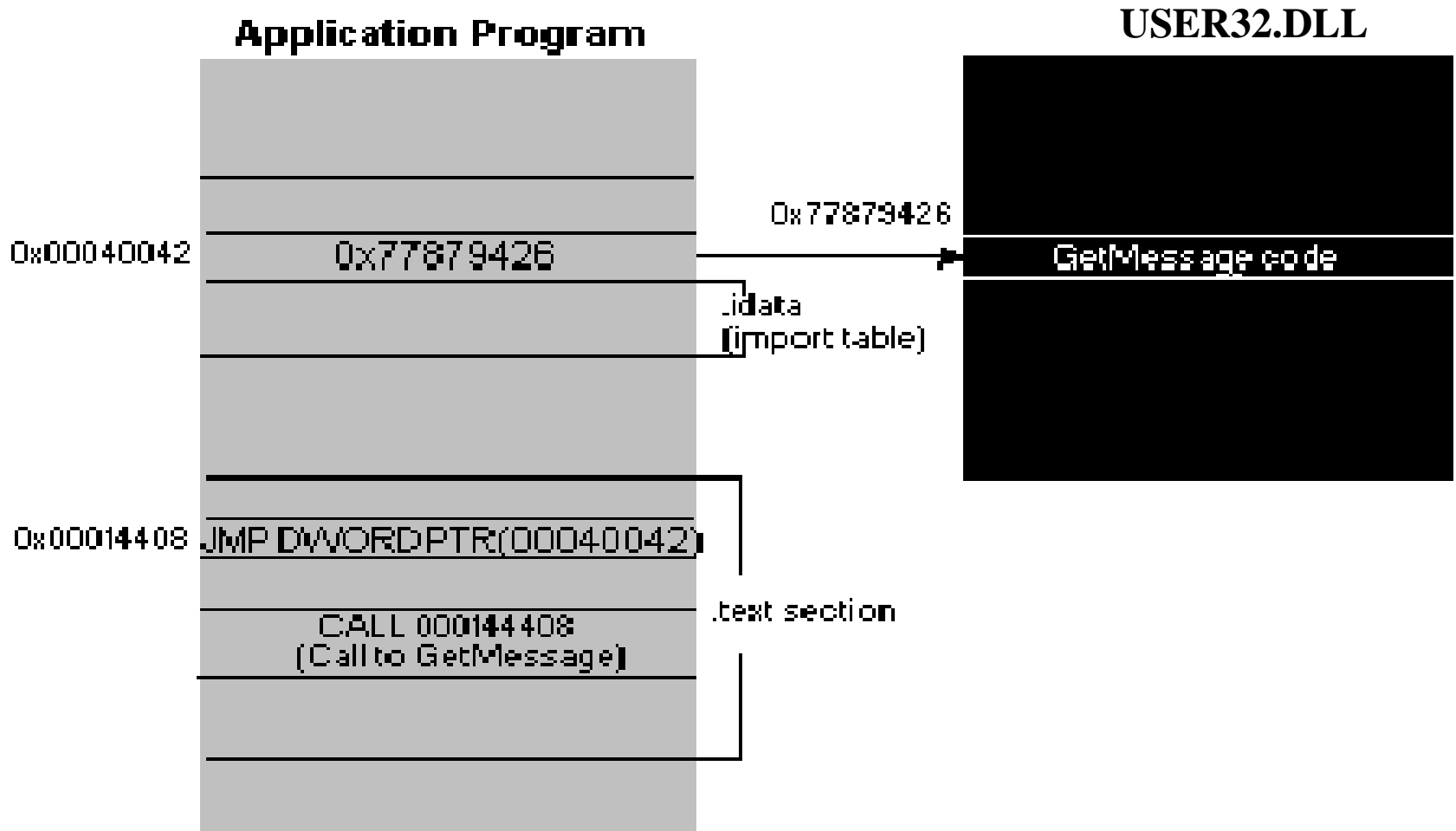
FFFFFFFFFH	操作系统使用 不可读写 (2GB)
80000000H 7FFFFFFFFFH	用于防止跨用户/系统边界传输数据 不可读写 (64KB)
7FFF0000H 7FFFEFFFH	进程私有空间 128KB~2GB
00010000H 0000FFFFH	用户捕捉NULL指针用 不可读写 (64KB)
00000000H	

## Windows进程的地址空间

进程需要用到的DLL都会载入自己的私用地址空间  
那么如何找到DLL中的函数呢？



一个程序调用外部DLL中的函数时并不直接调用那个DLL中的函数。相反，CALL指令转到了同一个.text节中的JMP DWORD PTR [XXXXXXXX]类型的指令。这种JMP指令查找并且将控制权转移到的地址是实际的目标地址。PE文件的.idata节包含了加载器用以确定目标函数的地址并且在可执行映像中修正它们所需的信息。



## 引入表 import table

- ◆ Data Directory数组第二项的VirtualAddress包含引入表地址。
- ◆ 引入表实际上是一个**IMAGE\_IMPORT\_DESCRIPTOR**结构数组。
- ◆ 每个结构包含PE文件引入函数的一个相关DLL的信息。
- ◆ 比如，如果该PE文件从10个不同的DLL中引入函数，那么这个数组就有10个成员。该数组以一个全0的成员结尾。

目录表查看器

	RVA	大小
输出表: Export Table:		00000000
输入表: Import Table:		000000B4
资源: Recourse:		00004348
例外: Exception:		00000000
安全性: Security:		00001558
基址重定位: Baserebase:		00000000
调试: Debug:		00000000
版权: Copyright:		00000000
全局指针: Globalptr:		00000000
TLS 表: Tls Table:		00000000
载入配置: Load Config:		00000000
输入表范围: Bound Import:		00000000
输入地址表: Import Address Table:		0000028C

### ◆ 引出函数节.edata

- ▶▶ 引出函数节是本文件向其他程序提供的可调用函数列表
- ▶▶ 这个节一般用在DLL中，EXE文件中也可以有这个节，但通常很少使用
- ▶▶ 当PE装载器执行一个程序，它将相关DLLs都装入该进程的地址空间。然后根据主程序的引入函数信息，查找相关DLLs中的真实函数地址来修正主程序。PE装载器搜寻的是DLLs中的引出函数。

- ▶▶ **DLL/EXE要引出**一个函数给其他**DLL/EXE**使用，有两种实现方法
  - 通过函数名引出
  - 通过函数名引出或者仅仅通过序数引出。比如某个**DLL**要引出名为"**GetSysConfig**"的函数，如果它以函数名引出，那么其他**DLLs/EXEs**若要调用这个函数，必须通过函数名-**GetSysConfig**。
  - 仅仅通过序数引出
  - 什么是序数呢？序数是唯一指定**DLL**中某个函数的**16**位数字，在所指向的**DLL**里是独一无二的。例如在上例中，**DLL**可以选择通过序数引出，假设是**16**，那么其他**DLLs/EXEs**若要调用这个函数必须以该值作为**GetProcAddress**调用参数。这就是所谓的仅仅靠序数引出。
- ▶▶ 引出函数节对病毒来说，非常重要的

- 引出函数节的开始，是一个  
**IMAGE\_EXPORT\_DIRECTORY**结构

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD Characteristics;           // 一般为0
    DWORD TimeDateStamp;             // 文件生成时间
    WORD MajorVersion;               // 主版本号
    WORD MinorVersion;               // 次版本号
    DWORD Name;                      // 指向DLL的名字
    DWORD Base;                      // 基数，加上序数就是函数地址数组的索引值
    DWORD NumberOfFunctions;         // AddressOfFunctions数组的项数
    DWORD NumberOfNames;             // AddressOfNames数组的项数
    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;
```

# EXE文件的格式

## PE文件格式

导出函数序号的起始值，将AddressOfFunctions字段指向的入口地址表的索引号加上该起始值，就是对应函数的导出号。

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD Characteristics;           // 一般为0
    DWORD TimeDateStamp;             // 文件生成时间
    WORD MajorVersion;               // 主版本号
    WORD MinorVersion;               // 次版本号
    DWORD Name;                      // 指向DLL的名字
    DWORD Base;                      // 基数，加上序数就是函数地址数组的索引值
    DWORD NumberOfFunctions;         // AddressOfFunctions数组的项数
    DWORD NumberOfNames;             // AddressOfNames数组的项数
    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;
```

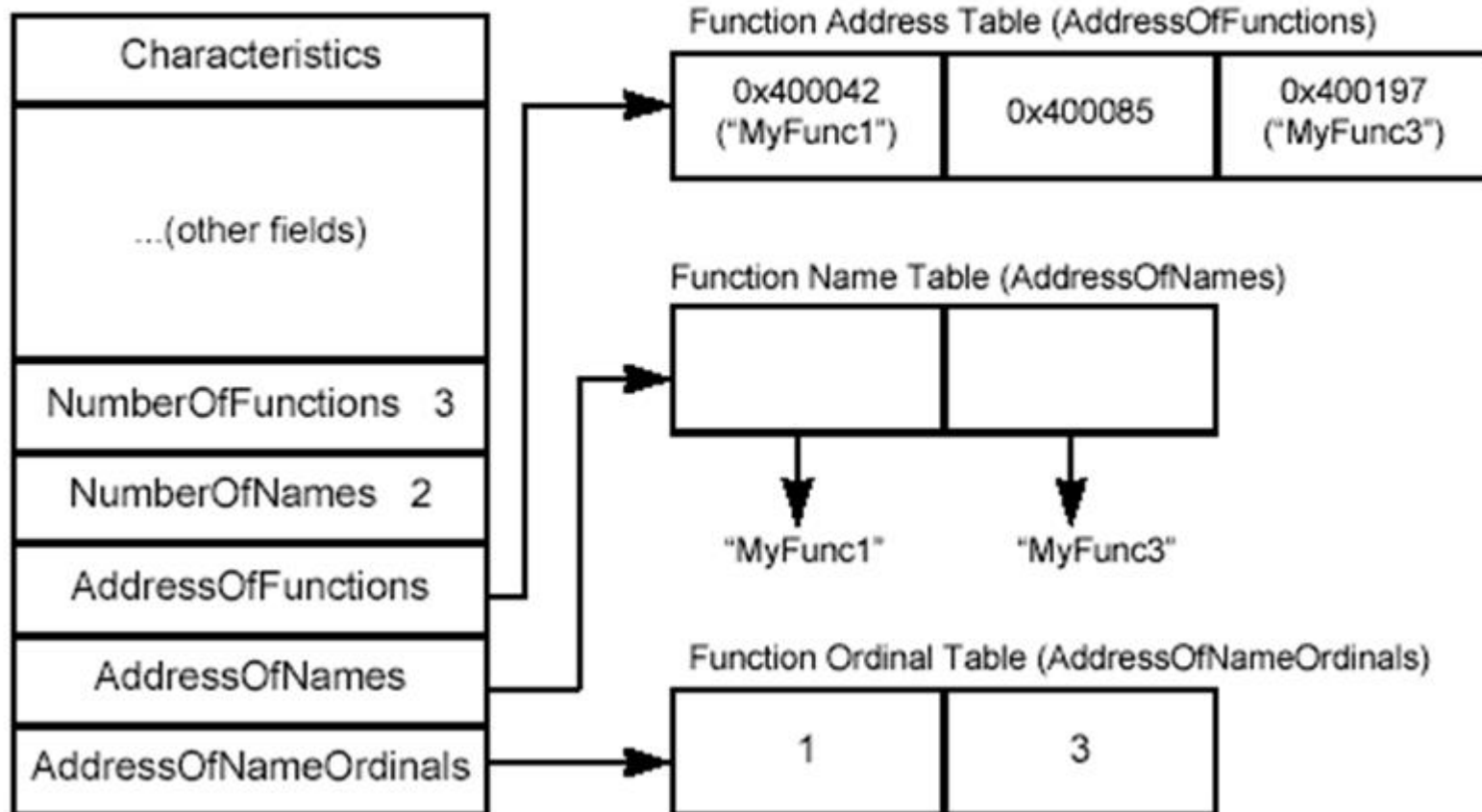
# EXE文件的格式

## PE文件格式

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD Characteristics;           // 一般为0
    DWORD TimeDateStamp;             // 文件生成时间
    WORD MajorVersion;               // 模块中所有函数的RVA都保存在一个数组里，该域就是指向该数组的
    WORD MinorVersion;              // 指向所有函数名的RVA都保存在一个数组里，该域就是指向该数组的
    DWORD NamePointer;               // 指向包含上述AddressOfNames数组中相关函数序数的16位数组；项的
    DWORD BasePointer;              // 值代表函数入口地址表的索引。与AddressOfNames一一对应。
    DWORD NumberOfFunctions;         // AddressOfFunctions数组的项数
    DWORD NumberOfNames;             // AddressOfNames数组的项数
    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;
```

# 引出函数示例

## IMAGE\_EXPORT\_DIRECTORY





# EXE文件的格式

## PE文件格式

已知引出函数名，获取函数地址的一般步骤：

1. 定位到PE header
2. 从数据目录表读取引出表的虚拟地址
3. 定位引出表获取名字数目 (NumberOfNames)，并行遍历AddressOfNames和AddressOfNameOrdinals指向的数组匹配名字。如果在AddressOfNames指向的数组中找到匹配名字，从AddressOfNameOrdinals指向的数组中提取索引值。

例如，若发现匹配名字的RVA存放在AddressOfNames数组的第6个元素，那就提取AddressOfNameOrdinals数组的第6个元素作为索引值。如果遍历完NumberOfNames个元素，说明当前模块没有所要的名字。从AddressOfNameOrdinals数组提取的数值作为AddressOfFunctions数组的索引。也就是说，如果值是5，就必须读取AddressOfFunctions数组的第5个元素，此值就是所要函数的RVA

已知函数的序数，获取函数地址的一般步骤：

1. 定位到PE header 从数据目录表读取引出表的虚拟地址
2. 定位引出表获取Base值
3. 减掉Base值得到指向AddressOfFunctions数组的索引
4. 将该值与NumberOfFunctions作比较，大于等于后者则序数无效
5. 通过上面的索引就可以获取AddressOfFunctions数组中的RVA

## PE文件格式

### ◆ 数据节:.data、.bss

- ▶ 已初始化数据节.data中存放的是在编译时已经确定的数据，这个节有时也叫DATA
- ▶ 未初始化数据节.bss存放的是没有初始化的全局变量和静态变量

### ◆ 资源节:.rsrc

- ▶ 资源节.rsrc存放程序要用到的菜单、字符串表和对话框等资源。资源节是树形结构，有一个主目录，主目录下有子目录，子目录下又是子目录或数据

### ◆ 重定位节:.reloc

- ▶ 重定位节存放有一个重定位表。若装载器不是把程序装载到程序编译时默认的基地址时，就需要这个重定位表来做一些调整

# EXE文件的格式

## PE文件格式

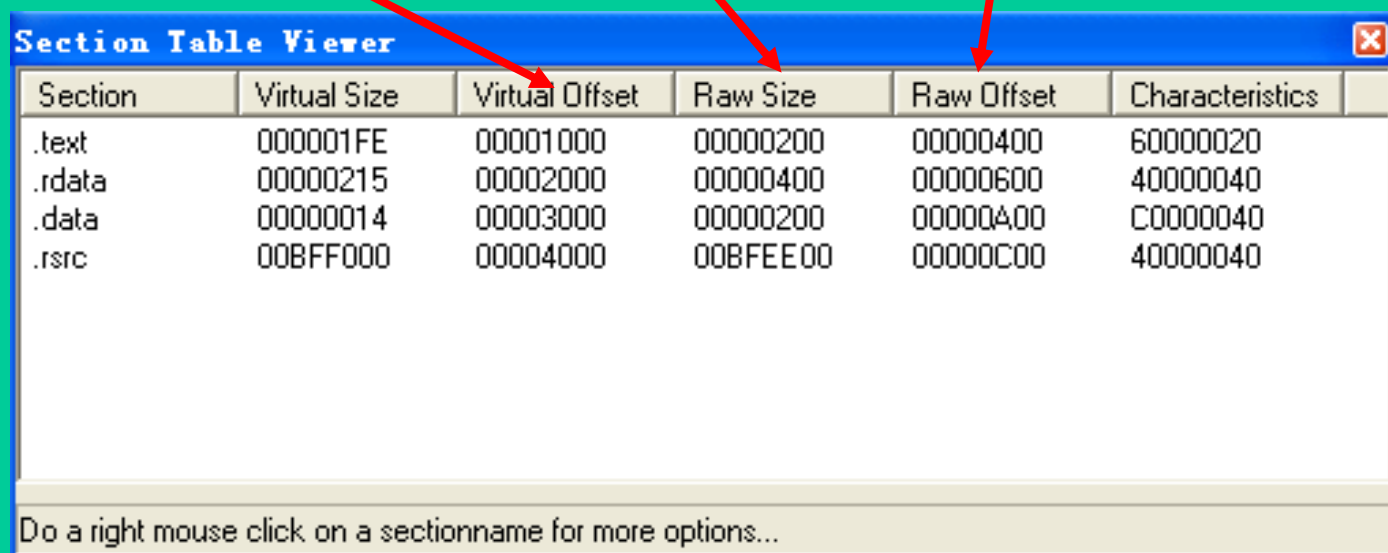
**NumberOfSections**知道有几个节

**SizeOfHeaders**知道节表在什么地方开始

遍历节表，**PointerToRawData**知道节在文件中偏移量

**SizeOfRawData**来决定映射内存的字节数

**VirtualAddress**加上**ImageBase**知道节的起始虚拟地址



Section	Virtual Size	Virtual Offset	Raw Size	Raw Offset	Characteristics
.text	000001FE	00001000	00000200	00000400	60000020
.rdata	00000215	00002000	00000400	00000600	40000040
.data	00000014	00003000	00000200	00000A00	C0000040
.rsrc	00BFF000	00004000	00BFEE00	00000C00	40000040

Do a right mouse click on a sectionname for more options...

## 工具篇 -- PEInfo

- 如何编写PE格式分析工具？
- 编程思路：文件格式检查 -> FileHeader读取 -> FileOptionalHeader读取 -> 数据目录表读取 -> 节表读取 -> 输出表读取 -> 输入表读取

# 文件格式检查

- **DOS Header**

```
STRUCT IMAGE_DOS_HEADER
```

```
{
```

```
    WORD    e_magic    // DOS可执行文件标记
```

```
    ...
```

```
    DWORD   e_lfanew    // 指向PE文件头 (+3ch)
```

```
} PIMAGE_DOS_HEADER ENDS
```

```
PIMAGE_DOS_HEADER pDH;
```

- 判断 `pDH -> e_magic == 'MZ'`
- 通过 `pDH -> e_lfanew` 找到 `IMAGE_NT_HEADERS`

# 文件格式检查

```
STRUCT IMAGE_NT_HEADERS
{
    DWORD Signature
    IMAGE_FILE_HEADER FileHeader
    IMAGE_OPTIONAL_HEADER32 OptionalHeader
} PIMAGE_NT_HEADERS ENDS
PIMAGE_NT_HEADERS pNTH;
```

- ◆ 检测pNTH -> Signature == 'PE';
- ◆ 至此，我们确定他符合PE文件的特征。

# 文件格式检查

```
BOOL IsPEFile(LPVOID ImageBase)
{
    PIMAGE_NT_HEADERS pNtH;
    PIMAGE_DOS_HEADER pDH;
    pDH = (PIMAGE_DOS_HEADER)ImageBase;
    if (pDH->e_magic!=IMAGE_DOS_SIGNATURE)
        return 0; //判断是否为MZ
    pNtH = (PIMAGE_NT_HEADERS)((DWORD)pDH +
        pDH ->e_lfanew);
    if (pNtH->Signature!=IMAGE_NT_SIGNATURE)
        return 0; //判断是否为PE
    return 1;
}
```



# FileHeader 读取

```
PIMAGE_FILE_HEADER GetFileHeader(LPVOID ImageBase)
{
    PIMAGE_DOS_HEADER pDH = NULL;
    PIMAGE_NT_HEADERS pNtH = NULL;
    PIMAGE_FILE_HEADER pFH = NULL;

    if( !IsPEFile(ImageBase) )
        return NULL;
    pDH = (PIMAGE_DOS_HEADER)ImageBase;
    pNtH = (PIMAGE_NT_HEADERS)((DWORD)pDH +
        pDH -> e_lfanew);
    pFH = &pNtH -> FileHeader;
    return pFH;
}
```

# FileOptionalHeader 读取

```
PIMAGE_OPTIONAL_HEADER GetOptionalHeader(LPVOID ImageBase)
{
    PIMAGE_DOS_HEADER      pDH = NULL;
    PIMAGE_NT_HEADERS      pNtH = NULL;
    PIMAGE_OPTIONAL_HEADER pOH = NULL;

    if( !IsPEFile(ImageBase) )
        return NULL;

    pDH = (PIMAGE_DOS_HEADER)ImageBase;
    pNtH = (PIMAGE_NT_HEADERS)((DWORD)pDH + pDH->e_lfanew);
    pOH = &pNtH->OptionalHeader;
    return pOH;
}
```

# NtHeader 读取

```
PIMAGE_NT_HEADERS GetNtHeaders(LPVOID ImageBase)
{
    if( !IsPEFile(ImageBase) )
        return NULL;
    PIMAGE_NT_HEADERS pNtH;
    PIMAGE_DOS_HEADER pDH;
    pDH = (PIMAGE_DOS_HEADER)ImageBase;
    pNtH = (PIMAGE_NT_HEADERS)((DWORD)pDH +
        pDH ->e_lfanew);

    return pNtH;
}
```

# wsprintf()

- **FileHeader** 和 **FileOptionalHeader** 的信息若要以十六进制方式显示在编辑控件上，应当怎么做？
- 我们可以利用**wsprintf()** 函数，将欲显示的值进行格式化。
  - **wsprintf**在Windows编程设计中常用于格式化输出

# 感谢大家！

