

---

# 第六章

## 引言：

可执行文件格式是操作系统本身执行进制的反映，虽然研究可执行文件格式并不是程序员的首要任务，但这想种工作能够积累大量的知识，有助于对操作系统的深刻理解，掌握可执行文件的数据结构，也是研究软件安全的必修课。

## 我们为什么要学习 PE？

了解可执行文件结构，更方便的了解操作系统。

## 什么才是正确的学习方法？

本章节主要是记结构体，然后要写程序，如果没写程序，是永远学不懂的。

## 本章必须要掌握的知识点：

1. PE 头结构。
2. 文件和内存的对齐方式。
3. 添加节，扩大节，合并节。
4. 导入表，导出表，重定位表。
5. IAT 表。

## 6.1 PE

### 本节主要内容：

掌握 IMAGE\_DOS\_HEADER、IMAGE\_FILE\_HEADER、IMAGE\_OPTIONAL\_HEADER 结构。


### 老唐语录：

今天我们开始讲 PE。

1. 用 OD 随便打开一个进程，或者打开一个记事本，看一下它的内存分布（0-2G）。大家都抄过 CR3，也应该知道内存分布了。2G 以上的大部分都一样。然后点击查看选项，选

择内存，也就是说这个 CR3 里面有哪些部分构成。

如图 6-1 所示：



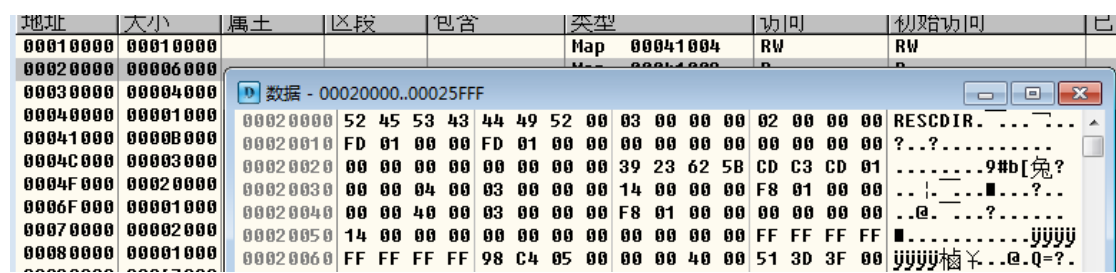
地址	大小	属主	区段	包含	类型	访问	初始访问	已映射为
00010000	00010000				Map 00041004	RW	RW	
00020000	00006000				Map 00041002	R	R	
00030000	00004000				Map 00041002	R	R	
00040000	00001000	notepad		PE 文件头	Imag 01001002	R	RWE	
00041000	00008000	notepad	.text	代码,输入表	Imag 01001002	R	RWE	
0004C000	00003000	notepad	.data	数据	Imag 01001002	R	RWE	
0004F000	000020000	notepad	.rsrc	资源	Imag 01001002	R	RWE	
0006F000	00001000	notepad	.reloc	重定位	Imag 01001002	R	RWE	
00070000	00002000				Map 00041002	R	R	

图 6-1

我们只看 地址，大小，访问，初始访问，如果没有学页的话，你根本看不懂，页怎么会有属性呢？等等一系列问题都出来了，当然我们学过就知道了，觉得理所当然。

我们看 2G 空间，前 64K 和后 64K 都是空的，不可访问的。

可以看里面的内容 双击就可以了 如图 6-2 所示：



地址	大小	属主	区段	包含	类型	访问	初始访问	已
00010000	00010000				Map 00041004	RW	RW	
00020000	00006000				Map 00041002	R	R	
00030000	00004000				Map 00041002	R	R	
00040000	00001000	notepad		PE 文件头	Imag 01001002	R	RWE	
00041000	00008000	notepad	.text	代码,输入表	Imag 01001002	R	RWE	
0004C000	00003000	notepad	.data	数据	Imag 01001002	R	RWE	
0004F000	000020000	notepad	.rsrc	资源	Imag 01001002	R	RWE	
0006F000	00001000	notepad	.reloc	重定位	Imag 01001002	R	RWE	
00070000	00002000				Map 00041002	R	R	

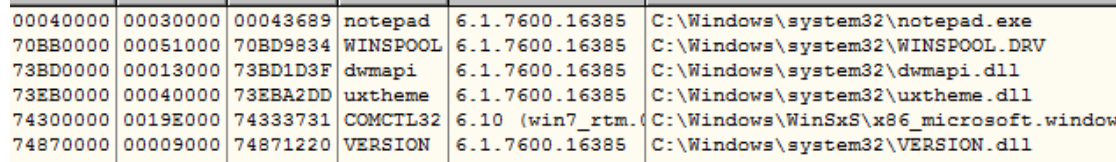
  

地址	大小	属主	区段	包含	类型	访问	初始访问	已
00010000	00010000				Map 00041004	RW	RW	
00020000	00006000				Map 00041002	R	R	
00030000	00004000				Map 00041002	R	R	
00040000	00001000	notepad		PE 文件头	Imag 01001002	R	RWE	
00041000	00008000	notepad	.text	代码,输入表	Imag 01001002	R	RWE	
0004C000	00003000	notepad	.data	数据	Imag 01001002	R	RWE	
0004F000	000020000	notepad	.rsrc	资源	Imag 01001002	R	RWE	
0006F000	00001000	notepad	.reloc	重定位	Imag 01001002	R	RWE	
00070000	00002000				Map 00041002	R	R	

图 6-2

可以选择查看方式，数据或者反汇编以及文本等等，2G 以下在 3 环都可以看到的。你会发现有些块是有名字的，比如：.text .data 等等。

我们在查看里面看一下执行模块和内存对一下有什么区别？如图 6-3 所示：



地址	大小	名称	路径	版本
00040000	00030000	notepad	C:\Windows\system32\notepad.exe	6.1.7600.16385
70BB0000	00051000	WINSPOOL	C:\Windows\system32\WINSPOOL.DRV	6.1.7600.16385
73BD0000	00013000	dwmapi	C:\Windows\system32\dwmapi.dll	6.1.7600.16385
73EB0000	00040000	uxtheme	C:\Windows\system32\uxtheme.dll	6.1.7600.16385
74300000	0019E000	COMCTL32	C:\Windows\WinSxS\x86_microsoft.window	6.10 (win7_rtm.0
74870000	00009000	VERSION	C:\Windows\system32\VERSION.dll	6.1.7600.16385

图 6-3

学过 API 就知道啦，页的属性是可以改变的，有一个 API (VirtualProtect)可以改变页的属性。

模块列表和内存列表的区别：

1. 内存列表是相同属性放在一起，模块列表是很多块放在一起了，也就是说当模块里面没有的那些内存都是用 virtualAlloc 分配的，如果说是有模块的那段内存都是在磁盘上有对应的 EXE 的。

- 2. 有模块就是有对应的 EXE。
- 3. 一个 EXE 在磁盘中叫做文件，在内存中叫模块。
- 4 每个模块都有一个格式就是一个结构体，windows 把它叫做 PE。

练习：

用 winhex 分别打开不同的文件格式

比如： .exe .txt .doc .jpg .ink .htm .dll .pdf 等等 打开 10 个 看看前四个字节是什么，写在纸上。

- 1. 在 winhex 里面鼠标拖动最后面看下多大，然后在磁盘上看下这个文件多大

如图 6-4 所示：

0002C1D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0002C1E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0002C1F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

图 6-4

如图 6-5 所示：

位置：	C:\
大小：	176 KB (180,736 字节)
占用空间：	180 KB (184,320 字节)

图 6-5

我们发现一样大。

大家发现没有，相同的文件前面几个字节是一样的。

.doc .mp3 .jpg .dll 等等都是有文件格式的，我们只需要记 .exe 文件格式，因为 .exe 文件是可执行文件格式，其他的文件是不可执行的，不能在 CPU 上运行的，每一种操作系统它最重要的格式就是它的可执行文件格式，因为操作系统就是为了支持这些文件而生成的，内核里面有很多机制，也是配合这种文件格式设计的。换句话说，这种文件格式也是适合操作系统设计的。

比如： PE 它是 windows 下的文件格式，是 MZ 打头的（4D5A）只有两个字节，后面很大一片就是对这个结构体的管理，比如：声音在什么位置，图像在什么位置，文字在什么位置，在前面这一片都是有记录的，也就是说，前面开头不只是标志而已，前面这一片是一个结构体。

不同的文件有不同的结构体，我们只需要学习 windows 下 .exe 文件格式。

---

那前面这个结构体是什么了？这个结构体头文件有定义。

**练习：**

在 VC6 中输入 `#include "winnt.h"` 在上面点右键打开就行了

然后搜索 `IMAGE_DOS_HEADER` 或者在程序里面输入 `IMAGE_DOS_HEADER` 按 F12 进去

```
typedef struct _IMAGE_DOS_HEADER {  
  
    WORD    e_magic;  
  
    WORD    e_cblp;  
  
    WORD    e_cp;  
  
    WORD    e_crlc;  
  
    WORD    e_cparhdr;  
  
    WORD    e_minalloc;  
  
    WORD    e_maxalloc;  
  
    WORD    e_ss;  
  
    WORD    e_sp;  
  
    WORD    e_csum;  
  
    WORD    e_ip;  
  
    WORD    e_cs;  
  
    WORD    e_lfarlc;  
  
    WORD    e_ovno;  
  
    WORD    e_res[4];  
  
    WORD    e_oemid;  
  
    WORD    e_oeminfo;  
  
    WORD    e_res2[10];  
  
    LONG    e_lfanew;  
  
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

可以用 winhex 打开这个文件，对着这个结构体拆分，写出他们的数值。

比如：e\_magic 的值是多少，写在纸上。

备注：这个结构体里面的每个成员都要背。

我们看最后一个成员 e\_lfanew（新的文件位置）用 winhex 看下一个成员在什么

---

地方 (0x3c) 在看里面的值指向的是哪个位置 (0xE0) 我们看一下 0xE0 的位置 (备注: 我打开的是记事本) (000000E0 45500000 (PE)), 所以把这个文件格式叫做 PE。

紧接着 PE 后面又是一个结构体, 那么这个结构在哪里了? 那个结构叫做 IMAGE\_FILE\_HEADER 同样的在 VC6 中 F12 进去。

```
typedef struct _IMAGE_FILE_HEADER
{
    WORD    Machine;

    WORD    NumberOfSections;

    DWORD   TimeDateStamp;

    DWORD   PointerToSymbolTable;

    DWORD   NumberOfSymbols;

    WORD    SizeOfOptionalHeader;

    WORD    Characteristics;

} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

**备注: 有的东西我说一遍, 你没记住也没用, 所以说要先记住, 我再说一遍, 才能印象深刻。**

紧接着在 IMAGE\_FILE\_HEADER 后面又是一个结构体, 那个结构叫做

IMAGE\_OPTIONAL\_HEADER (F12 进去)

```
typedef struct _IMAGE_OPTIONAL_HEADER
{
    WORD    Magic;

    BYTE    MajorLinkerVersion;

    BYTE    MinorLinkerVersion;

    DWORD   SizeOfCode;

    DWORD   SizeOfInitializedData;

    DWORD   SizeOfUninitializedData;

    DWORD   AddressOfEntryPoint;

    DWORD   BaseOfCode;

    DWORD   BaseOfData;

    DWORD   ImageBase;
```

---

```
    DWORD    SectionAlignment;

    DWORD    FileAlignment;

    WORD     MajorOperatingSystemVersion;

    WORD     MinorOperatingSystemVersion;

    WORD     MajorImageVersion;

    WORD     MinorImageVersion;

    WORD     MajorSubsystemVersion;

    WORD     MinorSubsystemVersion;

    DWORD    Win32VersionValue;

    DWORD    SizeOfImage;

    DWORD    SizeOfHeaders;

    DWORD    CheckSum;

    WORD     Subsystem;

    WORD     DllCharacteristics;

    DWORD    SizeOfStackReserve;

    DWORD    SizeOfStackCommit;

    DWORD    SizeOfHeapReserve;

    DWORD    SizeOfHeapCommit;

    DWORD    LoaderFlags;

    DWORD    NumberOfRvaAndSizes;

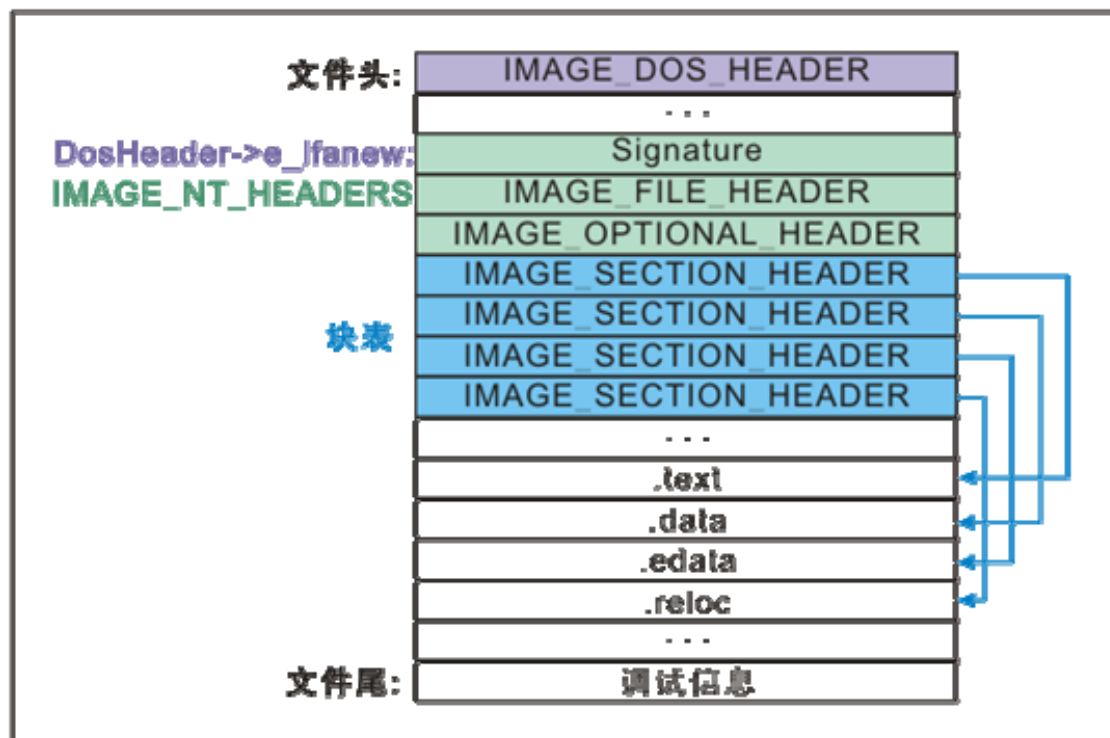
    IMAGE_DATA_DIRECTORY

DataDirectory[ IMAGE_NUMBEROF_DIRECTORY_ENTRIES ];

    } IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

就记这些结构体就行了

**课后理解:**



课后疑问：

本节没有疑问

课后总结：

PE 的格式是 0x4d5a。

课后练习：

记住：

`IMAGE_DOS_HEADER` 、 `IMAGE_FILE_HEADER` 、 `IMAGE_OPTIONAL_HEADER`  
这些结构体。

---

## 6.2 PE结构分布

本节主要内容：

1. 掌握 IMAGE\_DOS\_HEADER、IMAGE\_FILE\_HEADER、IMAGE\_OPTIONAL\_HEADER 结构里面的每个成员。

老唐语录：

结构体成员都记住了没有？我们现在开始一个一个讲：

WORD e\_magic; //MZ 标识 0x4d5a 这 0x4d5a 其实上就是微软开发人员的名字，他的名字就是 MZ,所以这是程序员的爱好，把自己的名字写上去了。

WORD e\_cp; //表示这个文件占了多少页，这个页是表示在磁盘上的（0x200 为单位）

问题来了，如果这个页不是整数倍了，那怎么办？我们看下一个成员。

WORD e\_cblp; //表示最后一页多少占了多少字节

例子：

e\_cp = 2 e\_cblp = 5 共占多少字节。

$2 * 200 + 5 = 0x405$  字节（磁盘上的）

所以说只能出现 0 - 511 不可能出现 512 的。

WORD e\_crlc //重定位的意思，当一个 EXE 文件放到内存里面去，有可能放在不同的位置，关于全局变量的位置需要修正，这个就是表示修正的信息放在哪里。

表示项数

WORD e\_cparhdr //表示这个文件头有多大 0x10 为单位（比如放个人信息的）

WORD e\_minallo //表示这个 DOS 程序最少要分配多少内存（这些都是老的格式）

WORD e\_maxalloc //表示这个 DOS 程序最大要分配多少内存（期望值）

WORD e\_ss;

WORD e\_sp;

WORD e\_cs;



---

```
WORD    e_ip;
```

就是程序初始化的时候 这些值是多少。

```
WORD    e_csum //检验和的意思：检验和就把整文件两个字节两个字节加起来，并且是带进位加的，结果放在这里来，没什么用处，可以随便改，操作系统不检查。
```

```
WORD    e_lfarlc //这个表格的起始地址（重定位）e_crlc 表示有多少项
```

```
WORD    e_res[4] //保留的
```

```
WORD    e_oemid
```

```
WORD    e_oeminfo //拉风信息标签 比如：公司名字之类的 0.0
```

DOS 下文件格式 MZ。

Win3.x 文件格式 NE。

win9.x 文件格式 LE。

winnt 文件格式 PE。

LONG e\_lfanew 表示新的文件格式从哪里开始，未必是执行 PE 头，只是我们现在学的是 PE。

PE -> Portable Executable file format (可移植的执行体)。

在 IMAGE\_NT\_HEADERS 里面有一个成员。

```
DWORD Signature //文件标识 四个字节 为了对齐。
```

```
WORD    Machine; //CPU 类型，在哪个 CPU 上跑（14C 代表 386CPU）。
```

```
#define IMAGE_FILE_MACHINE_I386    0x014c // Intel 386。
```

```
WORD    NumberOfSections; //文件的节表数，PE 是分节的表示有几个节（以后会讲）。
```

```
DWORD    TimeDateStamp; //时间戳（文件创建的日期与时间）
```

```
DWORD    PointerToSymbolTable; //指向符号表（用于调试）
```

```
DWORD    NumberOfSymbols; //符号表中的符号个数（用于调试）
```

```
WORD    SizeOfOptionalHeader; IMAGE_OPTIONAL_HEADER //结构大小
```

```
WORD    Characteristics; //文件属性
```

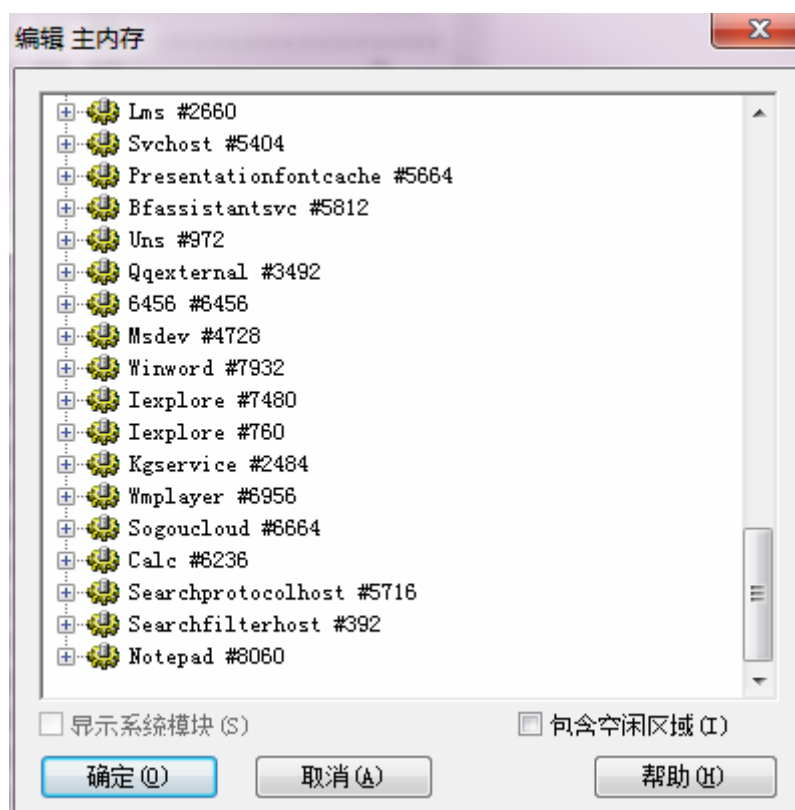
3. 用 winhex 打开一个记事本，对比一下磁盘和内存的区别

如图 6-6 所示：文件格式

notepad.exe						
Offset	0	1	2	3	4	5
00000000	4D	5A	90	00	03	00
00000016	B8	00	00	00	00	00
00000032	00	00	00	00	00	00
00000048	00	00	00	00	00	00
00000064	0E	1F	BA	0E	00	B4
00000080	69	73	20	70	72	6F
00000096	74	20	62	65	20	72

图 6-6

如图 6-7 所示：内存格式



---

Notepad: Notepad.exe									
	0	1	2	3	4	5	6	7	
]	4D	5A	90	00	03	00	00	00	
]	B8	00	00	00	00	00	00	00	
]	00	00	00	00	00	00	00	00	
]	00	00	00	00	00	00	00	00	
]	0E	1F	BA	0E	00	B4	09	CD	
]	69	73	20	70	72	6F	67	72	
]	74	20	62	65	20	72	75	6E	
]	6D	6F	64	65	2E	0D	0D	0A	

图 6-7 内存中

练习：我们发现文件的地址和内存的地址是不一样的，现在就两种状态对比，一个字节一个字节的比，把不同的地方抄在纸上。

**备注：**比完了以后才知道 **PE** 设计的理念，才知道它为什么这样设计。

例子 定义一个全局变量

```
char buffer[0x10000];

int main(int argc, char* argv[])
{
    buffer[0] = 0;

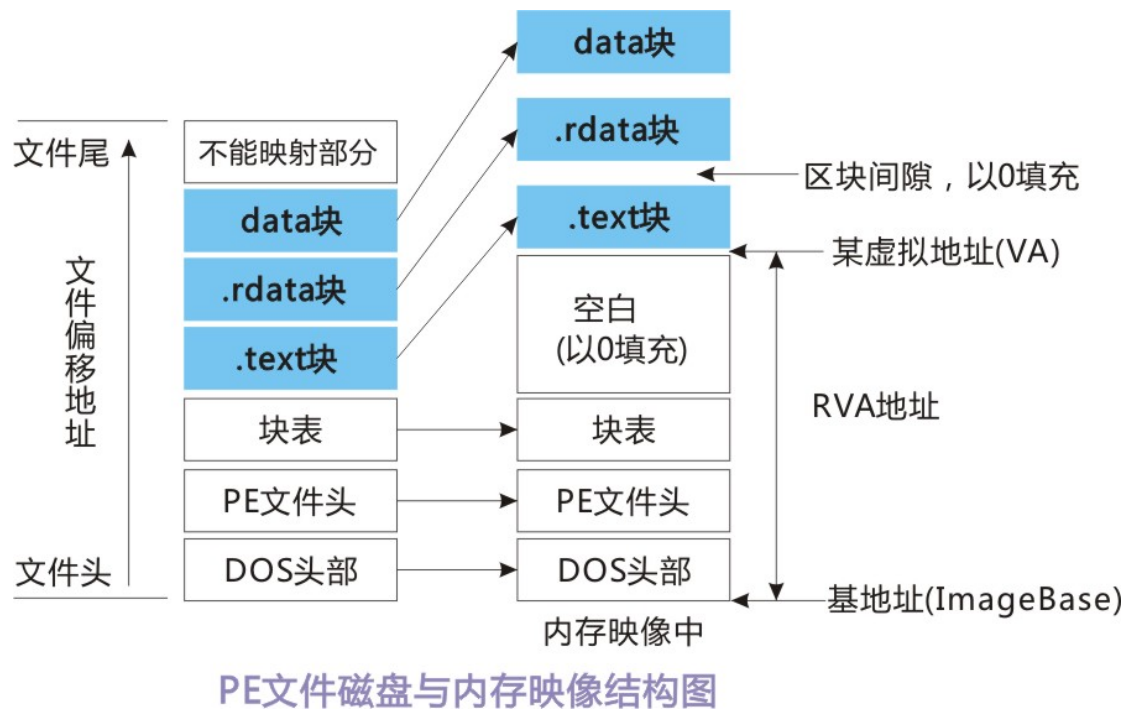
    printf("Hello World!\n");

    getchar();

    return 0;
}
```

在 winhex 中 对比一下磁盘和内存 发现区别 ， 把不同的写在纸上。

课后理解：



课后疑问：

节表的磁盘对齐方式和内存的对齐方式一样吗？

1. 不一样，磁盘是 0x200 对齐的，内存是 0x1000 对齐的，不过现在编译器生成的程序，磁盘和内存的对齐方式是一样的，都是 0x1000。

课后总结：

文件的地址和内存的地址是不一样的。

课后练习

自己写程序把 PE 这些结构体都打印出来

需要用到的 API    CreateFile    GetFileSize    Virtualalloc    ReadFile  
CloseHandle

备注：要打开一个 EXE 文件。

# 6.3 PE节表

本节主要内容：

- 1. 掌握 PE 节表

老唐语录：

内存中和磁盘中核心的东西，发现了没有了？

如图 6-8 所示：

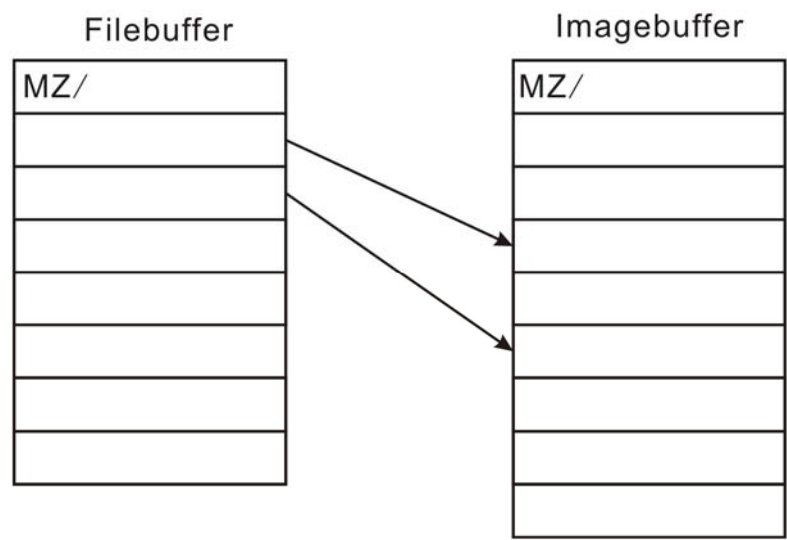


图 6-8

本质原因： FileBuffer 和 ImageBuffer 地址是错开的。

1. 当你比到 0x400 的时候，内存中在 0x1000 地方，整块都错开了，从错开的就可以感觉，这个文件在内存中是分块的。

现在我们讲，微软为什么要分块？不分块不好吗？其实本质的原因是（QQ 要开小号）一个应用程序要启多份。 当一个程序，它里面肯定有代码和数据，虽然代码和数据可以放在一起，但是微软为了让你能够开小号，把代码和数据分开了，代码分成一块，数据分成一块，为什么这样做了，学过页就知道了，当你的程序在内存当中有可能只执行，只读不可写。比如代码区通常是不可写的，数据区也有不可写的，比如：字符串，你通常不会去改它的。也有需要改写的地方，如果你能做到一点，把你的数据分的很开，把可写的放到一块，把可读的放到一块，并且保证页对齐。如果一个程序启两份的话，只读的这个页，你可以共用一

份，这样不是省了一段空间了，如果一个程序里面，有 10 个只读的页，有 20 个可读写的页，如果你启一份的时候它的读写没有什么关系，但是如果你启一百份了，明显的这里面 1000 个页变成 10 个页了，100 份可以共用一份。

**本质原因：一个应用程序可以启多份，为了节约内存空间。**

既然是分块，但也没必要错开啊，为什么还要错开了？本质原因还是省空间，分块是为了省内存空间，错开也是为了省内存，为什么？大家发现没有，昨天要你开 `char buffer[0x100000]` 在内存里面占的空间是 1M，在磁盘上占的空间是 0。也就是说分块的时候就算你填 0 对齐，那也有错开的可能，因为这一块空间，在内存中占 1M，在磁盘为 0，为什么还要 1M 了，这不是浪费磁盘空间吗？

**错开的本质是节省磁盘空间。**

磁盘是以 0x200 对齐的

内存是以 0x1000 对齐的

实际上我们自己写的程序也是 0x1000 对齐的。早期的是 0x200 对齐的

我们用 OD 打开一个记事本看一下，是分块的。如图 6-9 所示：

00050000	00001000	notepad		PE 文件头	Imag	R	RWE
00051000	00008000	notepad	.text	代码,输入表	Imag	R	RWE
0005C000	00003000	notepad	.data	数据	Imag	R	RWE
0005F000	00020000	notepad	.rsrc	资源	Imag	R	RWE
0007F000	00001000	notepad	.reloc	重定位	Imag	R	RWE

图 6-9

这些块对应着一个结构体，这个结构体在哪里了？

在 vc 中输入 `IMAGE_SECTION_HEADER` 按 F12 进去

```
typedef struct _IMAGE_SECTION_HEADER {  
  
    BYTE    Name[ IMAGE_SIZEOF_SHORT_NAME ]; // 给每一个块启一个名字  
  
    union {  
  
        DWORD    PhysicalAddress;  
  
        DWORD    VirtualSize; // 在内存中大小  
    } Misc;  
  
    DWORD    VirtualAddress; // 内存中的位置  
  
    DWORD    SizeOfRawData; // 文件中的大小  
  
    DWORD    PointerToRawData; // 文件中的位置  
  
    DWORD    PointerToRelocations;
```

```
DWORD   PointerToLinenumbers;

WORD     NumberOfRelocations;

WORD     NumberOfLinenumbers;

DWORD   Characteristics;//属性

} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER
```

我们分块就是区别属性

这四个成员是干什么用的了？

```
DWORD   PointerToRelocations;

DWORD   PointerToLinenumbers;

WORD     NumberOfRelocations;

WORD     NumberOfLinenumbers;
```

就是说这个结构体未必只给 PE 用，还会给其他的格式用，其他的文件也会用到这个结构体，别的格式可能会用到这几个成员。PE 格式没有用到这几个成员。

编译器生成 PE 的时候，编译器会自动生成一张表（关键是内存大小，内存位置，文件位置，文件大小）。

如 6-1 表所示：

表 6-1

NAME	FileAddr	FileSize	MemAddr	MemSize	Attributes

具体几项是看 NumberOfSections 这个成员。

课后理解：

将 notepad.exe 复制到 c 盘根目录下，用 winhex 打开，修改 0x3c 处为 0。保存后，用 ida 打开，光标放置 near，按空格键，查看代码：

```

00 ; ===== S U B R O U T I N E =====
00
00 ; Attributes: noreturn
00
00         public start
00 start    proc near
00         push    cs
01         pop     ds
02         assume ds:seg000
02         mov     dx, 0Eh
05         mov     ah, 9
07         int     21h                ; DOS - PRINT STRING
07                                     ; DS:DX -> string terminated by "$"
09         mov     ax, 4C01h
0C         int     21h                ; DOS - 2+ - QUIT WITH EXIT CODE (EXIT)
0C start|   endp                    ; AL = exit code
0C
0C ; -----
0E aThisProgramCan db 'This program cannot be run in DOS mode.',0Dh,0Dh,0Ah
0E             db '$',0
3A             align 8
40             db 0ECh, 85h, 5Bh, 0A1h, 0A8h, 0E4h, 35h, 0F2h, 0A8h, 0E4h
40             db 35h, 0F2h, 0A8h, 0E4h, 35h, 0F2h, 6Bh, 0EBh, 3Ah, 0F2h
40             db 0A9h, 0E4h, 35h, 0F2h, 6Bh, 0EBh, 55h, 0F2h, 0A9h, 0E4h
40             db 35h, 0F2h, 6Bh, 0EBh, 68h, 0F2h, 0BBh, 0E4h, 35h, 0F2h
40             db 0A8h, 0E4h, 34h, 0F2h, 63h, 0E4h, 35h, 0F2h, 6Bh, 0EBh
40             db 6Bh, 0F2h, 0A9h, 0E4h, 35h, 0F2h, 6Bh, 0EBh, 6Ah, 0F2h
40             db 0BFh, 0E4h, 35h, 0F2h, 6Bh, 0EBh, 6Fh, 0F2h, 0A9h, 0E4h

```

2. Int 21h:dos 下的 API, AH 为主功能, AL 为子功能, 主功能为 9, 输出字符串;  
dx 指向字符串; 若主功能为 4c, 子功能为 01, 退出程序。
3. 早期磁盘大小有限, 按 0x200 对齐。

分块原因: 有的应用程序被打开多次

## 课后疑问:

PE 里面的节表可以随便乱放吗?

可以的。

## 课后总结:

一个应用程序可以启多份, 为了节约内存空间

## 课后练习:

把 PE 结构的每个成员打印出来



# 6.4 拷贝节表

本节主要内容：

1. 拷贝节表

老唐语录：

现在理解 PE 为什么这么一块一块的吗？其实 PE 错开的可能非常大，只是我们见到的它没错开，我们见到的 PE 它有两个状态 一个是文件状态（FileBuffer） 一个是内存状态（ImageBuffer）。

如图 6-10 所示：

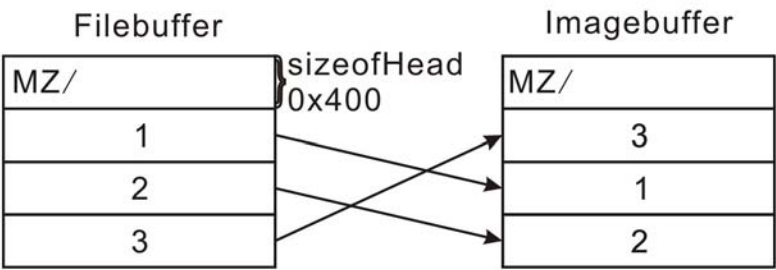


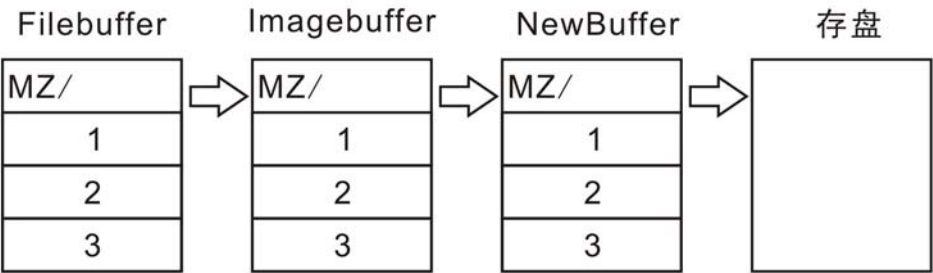
图 6-10

- 1. 文件中，节表是可以交叉 可以重叠，可以乱序。
- 2. 内存中不可以交叉乱序重叠。

课堂练习写程序：

今天就做一件事，首先开一个缓冲区，根据 sizeofImage 先拷贝头(sizeofheader) 再拷贝节表，一个 for 循环就拷贝完了，然后在开一个缓冲区，从内存拷贝在文件，根据节表考，然后在新建一个文件，把整个存盘。

如图 6-11 所示：



---

图 6-11

备注：读文件时要一次性读进来，这个文件有多大缓冲区就开多大。

### 课后理解：

从文件拷贝到内存：

```
CopyMemory(pNFileBuffer,pImageInMem,new_pImageOptionalHeader->
SizeOfHeaders);

for (i = 0 ; i < new_pImageFileHeader->NumberOfSections ; i++)
{
    CopyMemory(pNFileBuffer+new_pImageSectionHeader->PointerToRawD
ata , pImageInMem + new_pImageSectionHeader->VirtualAddress ,
new_pImageSectionHeader->SizeOfRawData);
    new_pImageSectionHeader++;}
```

### 课后疑问：

文件的地址怎么求？

用文件 FileBuffer + PointerToRawData。PointerToRawData 只是一个相对地址。

### 课后总结：

文件节表 是可以交叉 可以重叠，可以乱序。内存中不可以交叉乱序重叠。

### 课后练习：

首先开一个缓冲区，根据 sizeofImage 先拷贝头(sizeofheader) 在拷贝节表，一个 for 循环就拷贝完了，然后在开一个缓冲区，从内存拷贝在文件，根据节表考，然后在新建一个文件，把整个存盘。

---

## 6.5 PE拷贝节

本节主要内容：

1. 在节表里面添加内容

老唐语录：

写程序：

在 ImageBuffer 里面添加一段代码

```
push 0
push 0
push 0
push 0
call MessageBox
push 0
xitProcess
```

2. 不用 Call 调用，用 0xE8 调用

备注：

算函数的偏移。

课后理解：

```
(*ppImageInMemory)[addr++] = 0x6a;
(*ppImageInMemory)[addr++] = 0x00;
(*ppImageInMemory)[addr++] = 0x6a;
(*ppImageInMemory)[addr++] = 0x00;
(*ppImageInMemory)[addr++] = 0x6a;
(*ppImageInMemory)[addr++] = 0x00;
(*ppImageInMemory)[addr++] = 0x6a;
```

---

```
(*ppImageInMemory)[addr++] = 0x00;

//call MessageBoxA

(*ppImageInMemory)[addr] = 0xe8;

*(PDWORD)((*ppImageInMemory)+addr+1) =

(DWORD)(0x77D507EA -

(pImageOptionalHeader->ImageBase + addr + 5));

addr += 5;

(*ppImageInMemory)[addr++] = 0x6a;

(*ppImageInMemory)[addr++] = 0x00;

//ExitProcess

(*ppImageInMemory)[addr++] = 0xb8;

(*ppImageInMemory)[addr++] = 0x12;

(*ppImageInMemory)[addr++] = 0xcb;

(*ppImageInMemory)[addr++] = 0x81;

(*ppImageInMemory)[addr++] = 0x7c;

(*ppImageInMemory)[addr++] = 0xff;

(*ppImageInMemory)[addr++] = 0xd0;
```

### 课后疑问：

MessageBoxA 和 ExitProcess 的地址怎么获取？

在 0D 里面找（暂时先去填地址，以后会讲到）。

### 课后总结：

在节表里面写入函数，可以在这个函数里面做任何动作。

### 课后练习：

在 ImageBuffer 添加一个函数

---

## 6.6 PE添加节

本节主要内容：

1. 掌握 全局变量，变量，函数 字符串在节表的位置。
2. 添加节，在节里面添加一个函数。

老唐语录：

1. 从文件拷贝到内存

文件对齐 0x200。

内存对齐 0x1000。

2. IMAGE\_SECTION\_HEADER.NAME 没有太大的意义。

3. 自己定义一个全局变量，变量，函数 字符串把他们的地址打印出来在 winhex 分别找到他们的位置，在哪个节表中？

例子：

```
char* p = "Hello word" (rdata )
```

```
char ss[0x80];          (data)
```

```
int a = 5;               (data)
```

```
const int b = 6;         (rdata)
```

Data 数据有两部分，初始化的数据放在前面，未初始化的数据放在后面（跟编译器有关）。

练习：添加一个节程序能正常运行。

如果把节添加进去了，可以把自己的名字打印出来吗？我们今天就写一段代码进去，用硬编码写进去太麻烦了，我们就写一个函数进去。

```
void fun(int a, int b)
```

```
{
```

```
//有全局变量
```

```
char* name = "Hello word"
```

```

//有全局变量

char* buffer[0x80];

strcpy(buffer,"Hello word ");

//没有全局变量

char* aa[0x80];

aa[0] = 'H';

aa[1] = 'E';

.....

}

```

这个函数的首地址我们知道，直接拷贝就行了，拷贝多长呢？自己估计一下函数的长度，直接拷贝一个页，实在是不行用反汇编到 ret 就行了。把这个函数拷进去。

我们写简单一点，就弹出一个 MessageBox 就可以了，我们不要考虑全局变量，一个赋值就没有全局变量，只要没有全局变量，拷贝到哪里都可以执行，把这段代码拷贝到新加的节里面去就行了，把 EIP 指到函数那边去，然后你执行完了，最后还可以加参数，参数用硬编码填 push 0, push 1, call fun, 最后执行完以后跳到原来的 OEP 哪里去。

OEP -> AddressOfEntryPoint 函数入口 表示这个函数从哪里执行。

可以设置的，可以在 vc6 中设置一下如图 6-12 所示：

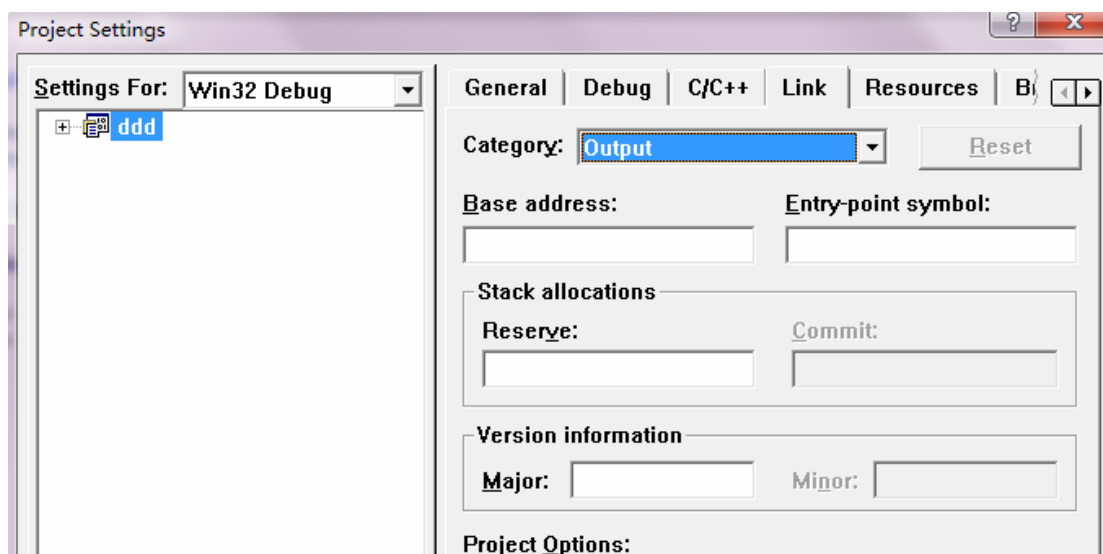


图 6-12

可以新建一个工程，设置一下，看看 OEP 变了没有。

---

程序跳过去能运行对吧，但是毕竟填这个字符串太麻烦了，一个一个填很麻烦，我们把这些麻烦全部解决了，我们只需要定义一个结构体。

```
struct PE_T
{
    char* ss[0x80];
    DWORD OEP;
    char name[0x80];
};
```

把该放的信息全部放进去。

如图所示 6-13：



图 6-13

最后一个节扩大了是吧，先把结构体赋值好，然后拷贝进来，然后把我们的代码拷贝进来，那 OEP 入口跳到哪里了，在跳到另一个函数里面，再加一个函数，这个函数干嘛呢？只需要 PUSH 这个结构在 PE 里面的偏移，然后在 CALL 我们的函数，用 C 写也行，用汇编写也行。

```
void FunEntry()
{
    push offset PE_T
    CALL fun
}
```

然后在把 fun 函数加一个参数， fun(PE\_T\* P)。

这就不是写了一个函数进去了，是写了一个函数，在加上一个函数和一片数据。

那为什么要弄两个函数了，因为没有变量啊，没有参数。

不用重定位 exe dll 都通用。

如图 6-14 所示：

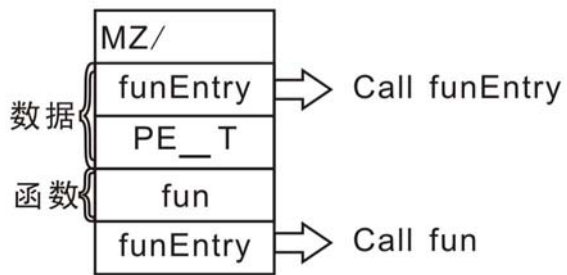


图 6-14

课后理解：

添加节部分代码实现：

```
//增加节数量计数

pImageFileHeader->NumberOfSections++;

//计算上一节内存对齐长度

if(((PIMAGE_SECTION_HEADER)(pImageSectionHeader-
1))->Misc.VirtualSize % pImageOptionalHeader->SectionAlignment)
{
    prelenInMem = (((PIMAGE_SECTION_HEADER)(pImageSectionHeader -
1))->Misc.VirtualSize / pImageOptionalHeader->SectionAlignment
+ 1) * pImageOptionalHeader->SectionAlignment;
}
Else
{
    prelenInMem = (((PIMAGE_SECTION_HEADER)(pImageSectionHeader -
1))->Misc.VirtualSize /
pImageOptionalHeader->SectionAlignment)*pImageOptionalHeader->
SectionAlignment;
}

//计算上一节磁盘对齐长度
```



---

```
if (((PIMAGE_SECTION_HEADER)(pImageSectionHeader -
1))->SizeOfRawData % pImageOptionalHeader->FileAlignment)
{
    prelenInDisk = (((PIMAGE_SECTION_HEADER)(pImageSectionHeader -
1))->SizeOfRawData / pImageOptionalHeader->FileAlignment + 1)
*pImageOptionalHeader->FileAlignment;
}
else
{
    prelenInDisk = (((PIMAGE_SECTION_HEADER)(pImageSectionHeader -
1))->SizeOfRawData / pImageOptionalHeader->FileAlignment)
*pImageOptionalHeader->FileAlignment;
}

pImageOptionalHeader->SizeOfImage += lenInMem;
```

### 课后疑问：

增加一个节后需要填属性吗？需要的话填什么  
需要，填可读可写可执行，可以参考.data 属性。

### 课后总结：

全局变量，变量，函数 字符串都在节表里面。  
添加函数时代码是代码，数据是数据，分开放。

### 课后练习：

添加一个节，在里面写入一个函数。

## 6.7 导出表结构

本节主要内容：

1. 掌握 GetProcAddress, LoadLibrary 函数。
2. 掌握导出表结构。

老唐语录：

我们上节写的调用的 messagebox 这个函数是一个 API，它是个固定的地址，换一台机器就不能跑了，就算你用 GetProcAddress 这个 API 得到，那你也要用这个 API 啊，当然要先用 LoadLibrary，只要写了这两个函数 就可以获取任何 API 了，现在我们写的程序，只能在自己的机器上跑，今天我们就来解决这个问题。

一个进程，有很多 PE 构成，其中 user32.dll、kernel32.dll、GDI32.dll 其实也是一个 PE，可以用 winhex 看一下，一个进程到底有多少 PE 构成的，其实这些 PE 之间也是有关系的，我们可以用 LoadLibrary 加载这个 DLL 再用 GetProcAddress 获取里面函数的地址。

DLL 就是提供函数给我们用的，EXE 就是用这些函数的。当然 exe 也可以提供函数给别人用。

1. 用 vc6 Toos => Depends 工具查看系统 user32.dll、kernel32.dll、GDI32.dll 用 PE 工具查看一下，这是系统提供的函数。在三个 DLL 放在 C:\Windows\System32 下的如图 6-15 所示：

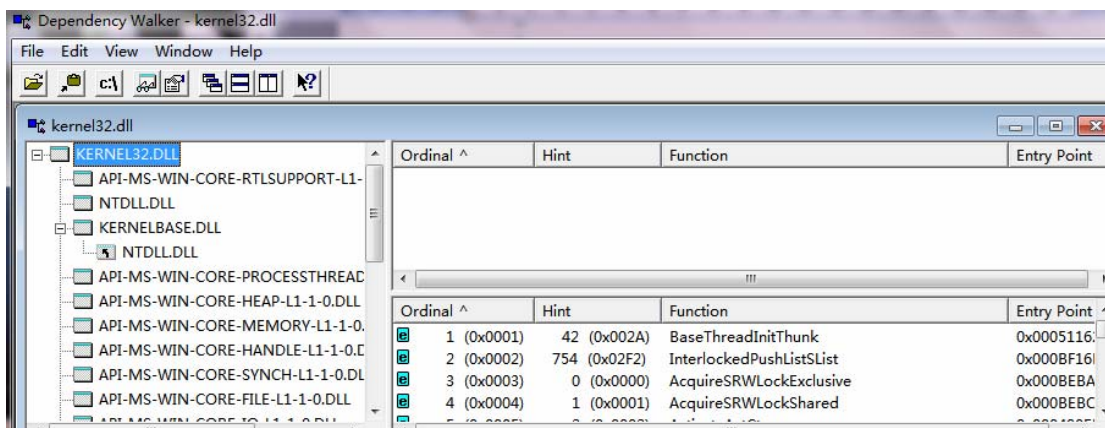


图 6-15

PE 里面为什么还可以调用 `CreateFile`，PE 除了代码数据以外，还放了一些其他的信息。

PE 它们之间的相互关系，一个 PE 可以提供函数给别人用，别的 PE 也可以提供函数，它们之间有什么关系呢？我们找自己的程序可以找到 `CreateFile`，在 `kernel32` 里也可以找的 `CreateFile`，它们之间是有某种关系的，PE 除了代码数据以外，还放了一些其他的信息比如：

编译器放的信息，比如说放的有：你用了哪些 DLL 里面的函数。用 `winhex` 打开自己的 EXE 还能搜到 `kernel32.dll` 的名字。PE 里面一定放着我们使用了哪些 DLL，并且使用了 DLL 里面的哪些 API，在 PE 里面都是有记录的，那么这些信息放在哪里了？肯定是放在节表里面了。至于这些信息是怎么放的，我们去背另一种结构。

在 `_IMAGE_OPTIONAL_HEADER` 里面有个 `IMAGE_DATA_DIRECTORY` `DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES]`。

是一个结构体数组，共有 16 项，这里面有系统的特殊数据。

```
typedef struct _IMAGE_DATA_DIRECTORY
{
    DWORD   VirtualAddress; //数据从哪里开始放
    DWORD   Size; //有多大
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

其实只有 `VirtualAddress` 有意义，放的是结构体的首地址（RVA）

`Size` 放的是结构大小，`size` 对这 16 项结构，有些有意义，有些没有意义。

节表在文件中可以乱放，也可以在两个节的中间 节表只管属性。

`_IMAGE_DATA_DIRECTORY` 里面的 16 项没有任何关系，可以随便乱放，可以相互包含。

我们先看其中几项：

```
Export directory //导出 本程序提供多少函数给别人用
Import directory // 导入 本程序用了哪个 dll 里面的函数
Base Relocation Table //重定位表
```

我们来看导出表结构，在 `vc6` 中直接搜索 `IMAGE_EXPORT_DIRECTORY` 找到结构体

---

```
typedef struct _IMAGE_EXPORT_DIRECTORY
{
    DWORD Characteristics; //属性
    DWORD TimeDateStamp; //时期邮戳
    WORD MajorVersion;
    WORD MinorVersion;
    DWORD Name; //模块名字 RVA
    DWORD Base; //基数，加上序书就是函数地址数组的索引值
    DWORD NumberOfFunctions; //函数个数
    DWORD NumberOfNames; //函数名字（有的有名字，有的没名字）
    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;
```

导入表: \_IMAGE\_IMPORT\_DESCRIPTOR

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD Characteristics; //属性
        DWORD OriginalFirstThunk; //指向输入表名称表（INT）的 RVA
    };
    DWORD TimeDateStamp; //邮戳
    DWORD ForwarderChain;
    DWORD Name; //DLL 名字指针
    DWORD FirstThunk; //指向输入表地址表（IAT）的 RVA
} IMAGE_IMPORT_DESCRIPTOR;
```

一个结构体对应个 DLL（导入表）。

**课后理解：**

//打印输出表部分代码

---

```
pNameOrdinals = (PWORD)(pFileBuffer +
    FileLocationCalculator(g_pImageExportDirectoryTable->AddressOf
NameOrdinals , RVA).dwFileOffset);

pAddressOfFunctions = (PDWORD)(pFileBuffer
    +FileLocationCalculator(g_pImageExportDirectoryTable->AddressOf
fFunctions , RVA).dwFileOffset);

pENT = (PDWORD)(pFileBuffer
    +FileLocationCalculator(g_pImageExportDirectoryTable->AddressOf
fNames , RVA).dwFileOffset);

//得到输入表地址

g_pImageImportDescriptorTable = (PIMAGE_IMPORT_DESCRIPTOR)
(FileLocationCalculator(g_pImageDataDirectory->VirtualAddress
    ,
RVA).dwFileOffset +(DWORD)pFileBuffer);
```

### 课后疑问：

输入表和输出表都只有一项吗？

输出表只有一项，输入表有很多项。

### 课后总结：

DLL 就是提供函数给我们用的，EXE 就是用这些函数的

导出表：本程序提供多少函数给别人用。导入表：本程序用了哪个 dll 里面的函数。

### 课后练习：

把导入表，导出表每个成员都打印出来

---

## 6.8 导出表原理

本节主要内容：

1. 导出表的原理
2. 掌握 `GetProcAddress` 函数。

老唐语录：

将 PE 文件中结构体里面的内容打印出来。

现在已经会打印导出表了，其实我们不需要使用 `GetProcAddress`，只要使用 `LoadLibrary` 获得地址后，想要的函数都可以找到。所以现在可以自己写个 `GetProcAddress` 函数。

只相当于放了个数组，把数组的内容打印出来。系统 `dll` 加载的 `dll` 也是系统 `dll`，因为通常是一个公司产的，所以规定将最常用的 `dll` 放在固定的地址，这样访问速度就会变快。

PE 里面提供了使用了哪些函数，这些函数在哪里，函数叫什么名字，序号，这些信息都要放在一个表格（数组）里，这个表格就叫做导出表。使用过的 `dll` 都有记录，因为整个 2G 以下都是由 `exe` 和 `dll` 构成的，每个 `dll` 装载的位置可能是变的，所以只有动态的链接，`dll` 有动态链接库的意思（dynamic link library）。当对 `dll` 中的某个位置 hook 后，系统会自动复制一份，放在其他位置的物理页，再将该物理页挂到当前进程的 `pte` 里面去。所有的 `pte` 都指向相同的物理页。

练习：自己写一个 `GetProcAddress`。

为什么要用名字索引呢？

微软设计 PE 结构中的导出表，就是记录我到底提供了几个函数，是按结构体的形式提供：结构体共三项：名字，序列号，地址，合并成一个数组结构体，查找起来很方便。

名字占四个字节，名字的地址，为什么不放名字字符串呢？因为字符串的长度不固定。更浪费空间，所以使用指针指向另一个地方排成一排，岂不更好。

序列号占两个字节，地址占四个字节，共 10 个字节。

---

问题：名字多还是函数地址多？

结构体里面有函数名计数和地址计数，比如：一个线性地址只能对应一个物理地址，但是一个物理地址可以对应多个线性地址，不可能一个线性地址对应多个物理地址。同一个函数可以取多个名字，所以地址可能比名字多。但是，工程师做 dll 的时候，要留接口给外面的人用，通常我们可以通过函数名就可探知其函数功能，所以工程师可能对于某些函数不希望外人知道，将其抹掉。所以很多函数可以不用名字导出，直接给个编号就可以。

查看 msdn 的 GetProcAddress。

第一个参数是名字。

第二个参数是指针。有可能是一个数(小于 0x10000)，有可能字符串(大于 0x10000)。

在 user32.dll 里面。

以后函数都是有名字导出的，只是改成 A1, A2 或者随机数组成。

怎样根据函数名找到函数地址？

先找到名字，再找到名字的下标，根据下标找到对应序号表，再根据序号表找地址表的下标位置地址。给个序号，只需要根据序号减去基址查地址表。所以序号表只有在给函数名的时候才起作用。序号减去序号的基址才是函数地址的下标。序号表的大小和名字一样。和名字是一一对应的。序号表里没有无名函数序号。

为什么序号表不是按 1, 2, 3, 4 往上走而是乱七八糟的乱序？

名字按 a-z 排序。

名字表和地址表必须建两张，因为个数不一样。

名字表和序号表必须建两张表，是因为 4 字节对齐，容易访问。

不仅仅 dll 有导出表，exe 也含有导出表，(如 OllyDebug.exe)。

微软将 0-0x10000 和 0xFFFFefff-0xFFFFffff 设为空。就是因为编号没有大于 0x10000。这样一旦误将编号用作指针，系统可以第一时间报错。

局部变量，不管是否赋值都赋 0xCC，就是为了一旦出错，错误稳定，(如果是随机数，一旦出错，错误不稳定)。

**课后理解：**

//GetProcAddress 部分代码

```
if (pImageDataDirectory->VirtualAddress)
```

---

```
{  
    pImageExportDirectoryTable =  
        (PIMAGE_EXPORT_DIRECTORY)(pImageDataDirectory->VirtualAddress +  
(PBYTE)hModule);  
}  
  
//重要部分  
pIndex = (PWORD)((PBYTE)hModule +  
pImageExportDirectoryTable->AddressOfNameOrdinals);  
pAddressOfFunctions = (PDWORD)((PBYTE)hModule +  
pImageExportDirectoryTable->AddressOfFunctions);  
pENT = (PDWORD)((PBYTE)hModule +  
pImageExportDirectoryTable->AddressOfNames);
```

### 课后疑问：

我用自己写的 `GetProcAddress` 函数获取系统的一些 API 函数，地址不对啊，只有几个函数不对，其它的没问题，这是为什么？

因为系统对 `GetProcAddress` 做了一些特殊的处理，有的 API 是 `ntdll.dll` 里面的却放到了 `user32.dll` 所以地址不对，可以用自己写 `GetProcAddress` 函数去获取系统 `GetProcAddress` 函数，然后在得到你需要的 API 地址就行了。

### 课后总结：

函数名字表、序号表、函数地址表、各一份，最主要的原因就是节省空间。

### 课后练习：

1. 将 `GetProcAddress` 加入导出表。
2. 将导入表抹零，在自己的代码中实现。



## 6.9 PE重定位表

本节主要内容：

1. 掌握重定位结构

老唐语录：

整个操作系统都是围着 PE 转。分页机制允许当一个应用程序被启动多份可以加载到同一个线性地址中，如果没有分页，低端内存被占用了，别人就不能占用了。早期没有分页，低 2G 都是物理内存。多进程如何开启？只有第一个应用程序用第 1M，第二个应用程序用第 2M，每个 exe 占的物理地址和线性地址都不一样。分页后，所有的 exe 都可以使用相同的线性地址，只是物理地址不一样。

分页后，代码区不用修正，因为每修正一次，就要添加一个物理页，一百个程序就要使用一百个物理页。dll 也不需要修正，系统 dll 都是由微软编写，vc6 里面有设置 dll 的加载地址 (alt+F7)，如果不设置，自动加载到 0x10000000。如果 dll 设置的地址出现冲突，代码区被修正。

PE 格式不一定是 x86 (CPU)，可能是 ARM (CPU)。mov eax, 变量地址 (全局变量)。如果代码区被修正，则变量地址将被修正，pe 挪动了多少位置，变量地址就挪动多少。

编译器不仅建立了 4 个字节的修正表，还生成一个字节属性表，比如有些地址要修正，有些不需要。比如：0x500000 这个位置需要修正，0x500008，0x700020，0x700028，0x700038，0x800020，0x800040……一共占 30×5 个字节，要进行压缩，我们将 500000 或 700000 或 800000 抽出来，加一个字节偏移修正，这样形成的表格就是重定位表。

调试 dll，在 dll 加个断点，在 vc6 里面加载 dll，

在 exe 工程里面改：在 loadlibrary 前面弹个 messagebox，后面弹个 messagebox。如果 dll 出错，观察后面的 messagebox 是否弹出？loadlibrary 已经调用 dll 中的函数，所以一旦出错，后面的语句不会执行。

表 6-2：压缩前

地址	类型	地址	类型
----	----	----	----

0x50002	3	0x50004	3
0x50020	3	0x50080	3
0x50090	3	0x50100	3

表 6-3: 压缩后

0x50000	
2	3
4	3
20	3
80	3
90	3
100	3

### 课后理解:

```
//打印重定位部分代码

nCount = (g_pImageBaseRelocationTable->SizeOfBlock - 8) / 2;

p = (PWORD)(g_pImageBaseRelocationTable + 1);

printf(FORMAT_DIV , _T(" 重 定 向 表 ") ,
_T("-----\n"));

printf(FORMAT_RELOCATION_TITLE , _T("TypeOffset") , _T("高 4 位") ,
_T("低 12 位") , _T("加上 VirtualAddress") , _T("FileOffset") , _T("
重定位前地址"));
```

### 课后疑问:

重定位表是干嘛用的?

说简单点就是修改全局变量或者 JMP CALL 这些用的。

---

## 课后总结：

重定位就是修正数据的。

## 课后练习：

1. 将重定位表打印出来
2. 将重定位表抹零，在你的代码中实现。

---

## 6.10 IAT表

### 本节主要内容：

#### 1. 掌握 IAT 表

### 老唐语录：

输入表可以给软件很强大的保护。

编译器生成的 exe 是有特征串的，而通过内存拷贝得到的 exe 没有特征串。（可以乱序拷贝，称为乱序。）

重定位表被抹掉，但是可以被破解，如果将重定位表清零，备份加密，也可以被破解（将基址记住，加载两份一比较，可以得出重定位的代码位置）。

使用代码拷贝可以将 API（比如 createfile）复制一份到你分配的内存里面，再链上。

**备注：只有代码拷贝才能实现强大的功能，所以我们要学习编码。**

清零不可以根据 16 个结构体中的虚拟尺寸大小来操作，要先遍历，先清里面的字符串，再清表，不能全部清零，可能会将有用信息清除掉。里面的虚拟大小尺寸没有意义。导出表不能全挪走，里面有三张表，有一个表指向字符串，另一张表是函数表，如果将函数地址表挪走，函数没有挪。

导入表中的 IAT，填着 api 的地址，将 IAT 移走后，别的地方使用 jmp 指令访问了 IAT 的原地址，或者是 call，这些指令，调用 API 的地方未必只有一个，你怎么知道在哪里，如果是间接跳转呢？

有重定位表，可以全部找到 IAT，因为 IAT 表本身是全局变量，可以扫描到底有多少条指令访问了 IAT 表。校验。

代码加密和挪走有关系吗？

编译器生成的代码区和数据区和节表没有关系。代码和数据是混在一起的，所以只能将 IAT、导出表挪走，才能加密，因为在程序得到控制权之前，系统要使用重定位表，导出表……

---

资源表，放图标和字符串的，如果将资源项 8 个字节抹零，观察变化。

图标不见了，所以如果要加密，必须先将资源表挪走。

导入表牵扯的 dll 很可能不是系统 dll，如果是第三方 dll，拷贝后，会显示该 dll 不存在。如果将导入表抹 0，则系统不会弹出，当程序自身扫描不存在时，让操作系统完成（因为语言问题）。表考出来放到你的体内，你帮他定位，但是每一个 dll 要留一项，虽然被加过密，但是在 dll 中，操作系统会检查每一个 dll，如果找不到，就会弹出对话框，语言问题就解决了。

练习：合并节并加密，

说明：节个数是一个，节大小时 sizeofimage。属性改成可读可写可执行。要加密就要将所有表挪走（或不加密），不是表的都加密。

## 课后理解：

导入表部分程序实现：

```
for (DWORD j=0; *(DWORD*)(pImageImportDescriptor+j)!=0; j++)//
所有模块遍历
{
    pImageFirstThunkData =
        (PIMAGE_THUNK_DATA32)((DWORD)pImageDOSHeader
+ (pImageImportDescriptor+j)->OriginalFirstThunk);
    for (DWORD i=0; *((DWORD*)pImageFirstThunkData+i)!=0 &&
        (*(DWORD*)pImageFirstThunkData+i)& IMAGE_ORDINAL_FLAG32)==0;
        i++)//单模块
    {
        pImageImportByName =
            (PIMAGE_IMPORT_BY_NAME)((DWORD)(pImageFirstThunkData+i)->u1.Ad
dressOfData + (DWORD)pImageDOSHeader);
        *(WORD*)dwTemp = pImageImportByName->Hint;
        DWORD k = -1;
        do{ //函数名循环
```

---

```
k++;

*(BYTE*)(dwTemp+sizeof(WORD)+k) = pImageImportByName->Name[k];
} while (pImageImportByName->Name[k]!=0);

(pImageFirstThunkData+i)->u1.AddressOfData =
(PIMAGE_IMPORT_BY_NAME)(dwTemp - (DWORD)pImageDOSHeader);

printf("0x%04X  :%s\n",((PIMAGE_IMPORT_BY_NAME)dwTemp)->Hint,((
PIMAGE_IMPORT_BY_NAME)dwTemp)->Name);

dwTemp += sizeof(WORD);

dwTemp += (k+1)*sizeof(BYTE);
}

CopyMemory((DWORD*)dwTemp,pImageFirstThunkData,i*sizeof(IMAGE_
THUNK_DATA32));

(pImageImportDescriptor+j)->OriginalFirstThunk =dwTemp -
(DWORD)pImageDOSHeader;

dwTemp +=(i+1)*sizeof(IMAGE_THUNK_DATA32);
}

CopyMemory((DWORD*)dwTemp,
(DWORD*)(pImageNTHHeader->OptionalHeader.DataDirectory[IMAGE_DI
RECTORY_ENTRY_IMPORT].VirtualAddress+(DWORD)pImageDOSHeader),
j*sizeof(IMAGE_IMPORT_DESCRIPTOR));

pImageNTHHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_E
NTRY_IMPORT].VirtualAddress =\
(DWORD)(dwTemp - (DWORD)pImageDOSHeader);

dwTemp += j*sizeof(IMAGE_IMPORT_DESCRIPTOR);

//合并节部分代码

dwTemp = pImageSectionHeader->VirtualAddress;

pImageNTHHeader->OptionalHeader.SizeOfHeaders = 0x1000;

pImageSectionHeader->PointerToRawData = 0x1000;

dwFileSize = pImageNTHHeader->OptionalHeader.SizeOfImage;
```

---

```
pImageSectionHeader->SizeOfRawData =  
pImageNTHHeader->OptionalHeader.SizeOfImage -  
pImageNTHHeader->OptionalHeader.SizeOfHeaders;  
pImageSectionHeader->Misc.VirtualSize =  
pImageNTHHeader->OptionalHeader.SizeOfImage -  
pImageNTHHeader->OptionalHeader.SizeOfHeaders;  
pImageNTHHeader->FileHeader.NumberOfSections = 1;-
```

### 课后疑问：

不用重定位可以移到 IAT 表吗？

不行。

### 课后总结：

学会移动 IAT 表等重要的 PE 结构体才能跟有效地给文件加密。

### 课后练习

将 createfile 复制到指定代码区（可以只复制一层，如果 hook 比较深，那么复制两层）。

问题：

- 1.遇到 call 指令（0xe8）记录并修正
- 2.若分配内存大小不够，先分配 10M，拷贝失败，再扩大一倍
- 3.拷贝函数顺序