

PE文件讲解

转载 杨航 AI 于 2013-03-12 22:51:05 发布 阅读量3.9k 收藏 1 点赞数

我们大家都知道，在Windows 9x、NT、2000下，所有的可执行文件都是基于Microsoft设计的一种新的文件格式Portable Executable File Format（可移植的执行体），即PE格式。有一些时候，我们需要对这些可执行文件进行修改，下面文字试图详细的描述PE文件的格式及对PE格式文件的修改。

PE文件框架构成

DOS MZ header

DOS Stub

PE header

Section table

Section 1

Section 2

Section...

Section n

上表是PE文件结构的总体层次分布。所有 PE文件(甚至32位的 DLLs) 必须以一个简单的 DOS MZ header开始，在偏移0处有DOS下可执行文件的“MZ标志”，有了它，一旦程序在DOS下执行，DOS就能识别出这是有效的执行体，然后运行紧随 MZ header之后的DOS Stub。紧接着DOS Stub的是PE header。PE header是PE相关结构IMAGE_NT_HEADERS的简称，其中包含了许多PE装载器用到的重要域。可执行文件在支持PE文件结构的操作系统中执行时，PE装载器将从DOS MZ header的偏移3CH处找到PE header的起始偏移量。因而跳过了DOS Stub直接定位到真正的文件头PE header。

小知识：DOS Stub实际上是个有效的EXE，在不支持PE文件格式的操作系统中，它将简单显示一个错误提示，类似于字符串“This program cannot run in DOS mode”或者程序员可根据自己的意图实现完整的DOS代码。通常DOS Stub由汇编器/编译器自动生成，对我们的用处不是很大，它简单调用中断21h服务9来显示字符串“This program cannot run in DOS mode”。

PE文件的真正内容划分成块，称之为Sections（节）。每节是一块拥有共同属性的数据，比如“.text”节等，那么，每一节的内容都是什么呢？实际上PE格式的文件把具有相同属性的内容放入同一个节中，而不必关心类似“.text”、“.data”的命名，其命名只是为了便于识别，所有，我们如果对PE格式的文件进行修改，理论上讲可以写入任何一个节内，并调整此节的属性就可以了。

PE header 接下来的数组结构Section table（节表）。每个结构包含对应节的属性、文件偏移量、虚拟偏移量等。如果PE文件里有5个节，那么此结构数组内就有5个成员。

以上就是PE文件格式的物理分布，下面将总结一下装载一PE文件的主要步骤：

- 1.PE文件被执行，PE装载器检查DOS MZ header里的PE header偏移量。如果找到，则跳转到PE header。
- 2.PE装载器检查PE header的有效性。如果有效，就跳转到PE header的尾部。

- 3.紧跟 PE header的是节表。PE装载器读取其中的节信息，并采用文件映射方法将这些节映射到内存，同时附上节表里指定的节属性。
4. PE文件映射入内存后，PE装载器将处理PE文件中类似Import table（引入表）逻辑部分。

PE文件头定义

我们可以在Winnt.h这个文件中找到关于PE文件头的定义：

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    //PE文件头标志：“PE/0/0”。在开始DOS header的偏移3CH处所指向的地址开始
    IMAGE_FILE_HEADER FileHeader;    //PE文件物理分布的信息
    IMAGE_OPTIONAL_HEADER32 OptionalHeader; //PE文件逻辑分布的信息
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
typedef struct _IMAGE_FILE_HEADER {
    WORD Machine;    //该文件运行所需要的CPU，对于Intel平台是14Ch
    WORD NumberOfSections;    //文件的节数目
    DWORD TimeDateStamp;    //文件创建日期和时间
    DWORD PointerToSymbolTable;    //用于调试
    DWORD NumberOfSymbols;    //符号表中符号个数
    WORD SizeOfOptionalHeader;    //OptionalHeader 结构大小
    WORD Characteristics;    //文件信息标记，区分文件是exe还是dll
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
typedef struct _IMAGE_OPTIONAL_HEADER {
    WORD Magic;    //标志字(总是010bh)
    BYTE MajorLinkerVersion;    //连接器版本号
    BYTE MinorLinkerVersion;    //
    DWORD SizeOfCode;    //代码段大小
    DWORD SizeOfInitializedData;    //已初始化数据块大小
    DWORD SizeOfUninitializedData;    //未初始化数据块大小
    DWORD AddressOfEntryPoint;
```

PE装载器准备运行的PE文件的第一个指令的RVA，若要改变整个执行的流程，可以将该值指定到新的RVA，这样新RVA处的指令首先被执行（以往许多文章都有介绍RVA，请大家先了解）。

```
DWORD BaseOfCode;    //代码段起始RVA
DWORD BaseOfData;    //数据段起始RVA
DWORD ImageBase;    //PE文件的装载地址
DWORD SectionAlignment;    //块对齐
DWORD FileAlignment;    //文件块对齐
```

```

WORD MajorOperatingSystemVersion; //所需操作系统版本号
WORD MinorOperatingSystemVersion; //
WORD MajorImageVersion; //用户自定义版本号
WORD MinorImageVersion; //
WORD MajorSubsystemVersion; //win32子系统版本。若PE文件是专门为Win32设计的
WORD MinorSubsystemVersion; //该子系统版本必定是4.0否则对话框不会有3维立体感
DWORD Win32VersionValue; //保留
DWORD SizeOfImage; //内存中整个PE映像体的尺寸
DWORD SizeOfHeaders; //所有头+节表的大小
DWORD CheckSum; //校验和
WORD Subsystem; //NT用来识别PE文件属于哪个子系统
WORD DllCharacteristics; //
DWORD SizeOfStackReserve; //
DWORD SizeOfStackCommit; //
DWORD SizeOfHeapReserve; //
DWORD SizeOfHeapCommit; //
DWORD LoaderFlags; //
DWORD NumberOfRvaAndSizes; //
IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
//IMAGE_DATA_DIRECTORY 结构数组。每个结构给出一个重要数据结构的RVA，比如引入地址表等
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;

```

```

typedef struct _IMAGE_DATA_DIRECTORY {
DWORD VirtualAddress; //表的RVA地址
DWORD Size; //大小
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;

```

PE文件头后是节表，在winnt.h下如下定义

```

typedef struct _IMAGE_SECTION_HEADER {
BYTE Name[IMAGE_SIZEOF_SHORT_NAME]; //节表名称,如 ".text"
union {
    DWORD PhysicalAddress; //物理地址
    DWORD VirtualSize; //真实长度
} Misc;
DWORD VirtualAddress; //RVA
DWORD SizeOfRawData; //物理长度

```

```

DWORD PointerToRawData;    //节基于文件的偏移量
DWORD PointerToRelocations; //重定位的偏移
DWORD PointerToLinenumbers; //行号表的偏移
WORD  NumberOfRelocations; //重定位项数目
WORD  NumberOfLinenumbers; //行号表的数目
DWORD Characteristics;    //节属性
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;

```

以上结构就是在Winnt.h中关于PE文件头的定义，如何我们用C/C++来进行PE可执行文件操作，就要用到上面的所有结构，它详细的描述了PE文件头的结构。

修改PE可执行文件

现在让我们把一段代码写入任何一个PE格式的可执行文件，代码如下：

```

-- test.asm --
.386p
.model flat, stdcall
option casemap:none

include /masm32/include/windows.inc
include /masm32/include/user32.inc
includelib /masm32/lib/user32.lib

.code

start:
    INVOKE MessageBoxA,0,0,0,MB_ICONINFORMATION or MB_OK
    ret
end start

```

以上代码只显示一个MessageBox框，编译后得到二进制代码如下：

```

unsigned char writeline[18]=;

```

好，现在让我们看看该把这些代码写到那。现在用Tdump.exe显示一个PE格式得可执行文件信息，可以发现如下描述：

Object table:

#	Name	VirtSize	RVA	PhysSize	Phys off	Flags
01	.text	0000CCC0	00001000	0000CE00	00000600	60000020 [CER]
02	.data	00004628	0000E000	00002C00	0000D400	C0000040 [IRW]
03	.rsrc	000003C8	00013000	00000400	00010000	40000040 [IR]

Key to section flags:

C - contains code

E - executable

I - contains initialized data

R - readable

W - writeable

上面描述此文件中存在3个段及每个段的信息，实际上我们的代码可以写入任何一个段，这里我选择“.text”段。用光盘中提供的代码可以得到一个PE格式可执行文件的头信息。

由于在PE格式的文件中，所有的地址都使用RVA地址，所以一些函数调用和返回地址都要经过计算才可以得到。

2.1 引言

通常Windows下的EXE文件都采用PE格式。PE是英文Portable Executable的缩写，它是一种针对于微软Windows NT、Windows 95和Win32s系统，由微软公司设计的可执行的二进制文件（DLLs和执行程序）格式，目标文件和库文件通常也是这种格式。这种格式由TIS（Tool Interface Standard）委员会（Microsoft、Intel、Borland、Watcom、IBM等）在1993进行了标准化。显然，它参考了一些UNIXes和VMS的COFF（Common Object File Format）格式。

认识可执行文件的结构非常重要，在DOS下是这样，在Windows系统下更是如此。了解了这种结构后就可以对可执行程序进行加密、加壳和修改等，一些黑客也利用了这些技术。为了使读者对PE文件格式有进一步的认识，本章从一个程序员的角度出发再次介绍PE文件格式。如果已经熟悉这方面的知识，可以跳过这一章。

2.2 PE文件格式概述

认识PE文件，既要懂得它的结构布局，又要知道它是如何装载到计算机内存中的。下面分别对它们进行说明。

2.2.1 PE文件结构布局

找到文件中某一结构信息有两种定位方法。第一种是通过链表方法，对于这种方法，数据在文件的存放位置比较自由。第二种方法是采用紧凑或固定位置存放，这种方法要求数据结构大小固定，它在文件中的存放位置也相对固定。在PE文件结构中同时采用以上两种方法。

因为在PE文件头中的每个数据结构大小是固定的，因此能够编写计算程序来确定某一个PE文件中的某个参数值。在编写程序时，所用到的数据结构定义，包括数据结构中变量类型、变量位置和变量数组大小都必须采用Windows提供的原型。图2.1所示的PE文件结构的总体层次分布如下：

PE文件结构总体层次分布

· DOS MZ Header

所有 PE文件（甚至32位的DLLs）必须以简单的DOS MZ header开始，它是一个IMAGE_DOS_HEADER结构。有了它，一旦程序在DOS下执行，DOS就能识别出这是有效的执行体，然后运行紧随MZ Header之后的DOS Stub。

DOS Stub

DOS Stub实际上是个有效的EXE，在不支持PE文件格式的操作系统中，它将简单显示一个错误提示，类似于字符串“This program requires Windows”或者程序员可根据自己的意图实现完整的DOS代码。大多数情况下DOS Stub由汇编器/编译器自动生成。

PE Header

紧接着DOS Stub的是PE Header。它是一个IMAGE_NT_HEADERS结构。其中包含了很多PE文件被载入内存时需要用到的重要域。执行体在支持PE文件结构的操作系统中执行时，PE装载器将从DOS MZ header中找到PE header的起始偏移量。因而跳过DOS Stub直接定位到真正的文件头 PE header。

Section Table

PE Header之后是数组结构Section Table（节表）。如果PE文件里有5个节，那么此Section Table结构数组内就有5个（IMAGE_SECTION_HEADER）成员，每个成员包含对应节的属性、文件偏移量、虚拟偏移量等。排在节表中的最前面的第一个默认成员是text，即代码节头。通过遍历查找方法可以找到其他节表成员（节表头）。

Sections

PE文件的真正内容划分成块，称为Sections（节）。每个标准节的名字均以圆点开头，但也可以不以圆点开头，节名的最大长度为8个字节。Sections是以其起始位址来排列，而不是以其字母次序来排列。通过节表提供的信息，可以找到这些节。程序的代码，资源等就放在这些节中。

节的划分是基于各组数据的共同属性，而不是逻辑概念。每节是一块拥有共同属性的数据，比如代码/数据、读/写等。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，唯有节的属性设置决定了节的特性和功能。

2.2.2 PE文件内存映射

在Windows系统下，当一个PE应用程序运行时，这个PE文件在磁盘中的数据结构布局和内存中的数据结构布局是一致的。系统在载入一个可执行程序时，首先是Windows装载器（又称PE装载器）把磁盘中的文件映射到进程的地址空间，它遍历PE文件并决定文件的哪一部分被映射。其方式是将文件较高的偏移位置映射到较高的内存地址中。磁盘文件一旦被装入内存中，其某项的偏移地址可能与原始的偏移地址有所不同，但所表现的是一种从磁盘文件偏移到内存偏移的转换，如图2.2所示。

PE文件内存映射

当PE文件被加载到内存后，内存中的版本称为模块（Module），映射文件的起始地址称为模块句柄（hModule），可以通过模块句柄访问内存中的其他数据结构。这个初始内存地址也称为文件映像基址（ImageBase）。载入一个PE程序的主要步骤如下：

（1）当PE文件被执行时，PE装载器首先为进程分配一个4GB的虚拟地址空间，然后把程序所占用的磁盘空间作为虚拟内存映射到这个4GB的虚拟地址空间中。一般情况下，会映射到虚拟地址空间中0x400000的位置。装载一个应用程序的时间比一般人所设想的要少，因为装载一个PE文件并不是把这个文件一次性地从磁盘读到内存中，而是简单地做一个内存映射，映射一个大文件和映射一个小文件所花费的时间相差无几。当然，真正执行文件中的代码时，操作系统还是要把存在于磁盘上的虚拟内存中的代码交换到物理内存（RAM）中。但是，这种交换也不是把整个文件所占用的虚拟地址空间一次性地全部从磁盘交换到物理内存中，操作系统会根据需要和内存占用情况交换一页或多页。当然，这种交换是双向的，即存在于物理内存中的一部分当前没有被使用的页，也可能被交换到磁盘。

（2）PE装载器在内核中创建进程对象和主线程对象以及其他内容。

(3) PE装载器搜索PE文件中的Import Table（引入表），装载应用程序所使用的动态链接库。对动态链接库的装载与对应用程序的装载方法完全类似。

(4) PE装载器执行PE文件首部所指定地址处的代码，开始执行应用程序主线程。

2.2.3 Big-endian和Little-endian

PE Header中IMAGE_FILE_HEADER的成员Machine 中的值，根据winnt.h中的定义，对于Intel CPU应该为0x014c。但是用[十六进制](#) 编辑器打开PE文件时，看到这个WORD显示的却是4c 01。其实4c 01就是0x014c，只不过由于Intel CPU是Little-endian，所以显示出来是这样的。对于Big-endian和Little-endian，请看下面的例子。一个整型int变量，长度为4个字节。当这个整型变量的值为0x12345678时，对于Big-endian来说，显示的是{12, 34, 45, 78}，而对于Little-endian来说，显示的却是{78, 45, 34, 12}。注意Intel使用的是Little-endian。

2.2.4 3种不同的地址

PE文件的各种结构中，涉及到很多地址、偏移。有些是指在文件中的偏移，有些 是指在内存中的偏移。以下的第一种是指在文件中的地址，第二、三种是指在内存中的地址。

第一种，文件中的地址。比如用十六进制编辑器打开PE文件，看到的地址（偏移）就是文件中的地址，使用某个结构的文件地址，就可以在文件中找到该结构。

第二种，当文件被整个映射到内存时，例如某些PE分析软件，把整个PE文件映射到内存中，这时是内存中的虚拟地址（VA）。如果知道在这个文件中某一个结构的内存地址的话，那么它等于这个PE文件被映射到内存的地址加上该结构在文件中的地址。

第三种，当执行PE时，PE文件会被载入器载入内存，这时经常需要的是RVA。例如知道一个结构的RVA，那么程序载入点加上RVA就可以得到该结构的内存地址。比如，如果PE文件装入虚拟地址（VA）空间的0x400000处，某一结构的RVA为0x1000，那么其虚拟地址为0x401000。

PE文件格式要用到RVA，主要是为了减少PE装载器的负担。因为每个模块都有可能被重定位到任何虚拟地址空间，如果让PE装载器修正每个重定位项，这肯定是个梦魇。相反，如果所有重定位项都使用RVA，那么PE装载器就不必操心那些东西了，即它只要将整个模块重定位到新的起始VA。这就像相对路径和绝对路径的概念：RVA类似相对路径，VA就像绝对路径。

注意，RVA和VA是指内存中，不是指文件中。是指相对于载入点的偏移而不是一个内存地址，只有RVA加上载入点的地址，才是一个实际的内存地址。

2.3 PE文件结构

在win32 SDK的文件winnt.h中有PE文件格式的定义。本文所用到的变量，如果没有特别说明，都在文件winnt.h中定义。

有关一些PE头文件结构一般都有32位和64位之分，如IMAGE_NT_HEADERS32和IMAGE_NT_HEADERS64等，除了在64位版本中的一些扩展域外，这些结构总是一样的。是采用32位还是64位，需要用#define _WIN64来定义，如果没有这种定义，则采用的是32位的文件结构。编译器将根据此定义选择相应的编译模式。

2.3.1 MS-DOS 头部

MS-DOS头部占据了PE文件的头64个字节，描述它内容的结构如下：

```
// 此结构包含于WINNT.H中

//

typedef struct _IMAGE_DOS_HEADER { // DOS的.EXE头部

    WORD e_magic;    // 魔术数字

    WORD e_cblp;     // 文件最后页的字节数

    WORD e_cp;       // 文件页数

    WORD e_crlc;     // 重定义元素个数

    WORD e_cparhdr;  // 头部尺寸，以段落为单位

    WORD e_minalloc; // 所需的最小附加段

    WORD e_maxalloc; // 所需的最大附加段

    WORD e_ss;       // 初始的SS值(相对偏移量)

    WORD e_sp;       // 初始的SP值

    WORD e_csum;     // 校验和

    WORD e_ip;       // 初始的IP值

    WORD e_cs;       // 初始的CS值(相对偏移量)

    WORD e_lfarlc;   // 重分配表文件地址

    WORD e_ovno;     // 覆盖号

    WORD e_res[4];   // 保留字

    WORD e_oemid;    // OEM标识符(相对e_oeminfo)

    WORD e_oeminfo;  // OEM信息

    WORD e_res2[10]; // 保留字

    LONG e_lfanew;   // 新exe头部的文件地址

} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```


|

其中第一个域e_magic，被称为魔术数字，它用于表示一个MS-DOS兼容的文件类型。所有MS-DOS兼容的可执行文件都将这个值设为0x5A4D，表示ASCII字符MZ。MS-DOS头部之所以有的时候被称为MZ头部，就是这个缘故。还有许多其他的域对于MS-DOS操作系统来说都有用，但是对于Windows NT来说，这个结构中只有一个有用的域——最后一个域e_lfnew，一个4字节的文件偏移量，PE文件头部就是由它定位的。

2.3.2 IMAGE_NT_HEADER头部

PE Header是紧跟在MS-DOS头部和实模式程序残余之后的，描述它内容的结构 如下：

|

```
typedef struct _IMAGE_NT_HEADERS {  
    DWORD Signature;           // PE文件头标志:"PE/0/0"  
  
    IMAGE_FILE_HEADER FileHeader;    // PE文件物理分布的信息  
  
    IMAGE_OPTIONAL_HEADER32 OptionalHeader; // PE文件逻辑分布的信息  
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

紧接PE文件头标志之后是PE文件头结构，由20个字节组成，它被定义为：

|


```
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;
```

```
#define IMAGE_SIZEOF_FILE_HEADER 20
```

|

其中请注意这个文件头部的大小已经定义在这个包含文件之中了，这样一来，想要得到这个结构的大小就很方便了。

Machine：表示该程序要执行的环境及平台，现在已知的值如表2.1所示。

 应用程序执行的环境及平台代码

IMAGE_FILE_MACHINE_I386 (0x14c)	Intel 80386 处理器以上
0x014d	Intel 80486 处理器以上
0x014e	Intel Pentium 处理器以上
0x0160	R3000(MIPS)处理器, big endian
IMAGE_FILE_MACHINE_R3000(0x162)	R3000(MIPS)处理器, little endian
IMAGE_FILE_MACHINE_R4000(0x166)	R4000(MIPS)处理器, little endian
IMAGE_FILE_MACHINE_R10000(0x168)	R10000(MIPS)处理器, little endian
IMAGE_FILE_MACHINE_ALPHA(0x184)	DEC Alpha AXP处理器
IMAGE_FILE_MACHINE_POWERPC(0x1f0)	IBM Power PC, little endian

NumberOfSections：节的个数。

TimeStamp：文件建立的时间。可用这个值来区分同一个文件的不同的版本，即使它们的商业版本号相同。这个值的格式并没有明确的规定，但是很显然地大多数的C编译器都把它定为从1970.1.1 00:00:00以来的秒数（time_t）。这个值有时也被用做绑定输入目录表。注意：一些编译器将忽略这个值。

PointerToSymbolTable及NumberOfSymbols：用在调试信息中，用途不太明确，不过它们的值总为0。

SizeOfOptionalHeader：可选头的长度（sizeof IMAGE_OPTIONAL_HEADER），可以用它来检验PE文件的正确性。

Characteristics：是一个标志的集合，其大部分位用于OBJ或LIB文件中。

文件头下面就是可选择头，这是一个叫做IMAGE_OPTIONAL_HEADER的结构，由224个字节组成。虽然它的名字是“可选头部”，但是请确信：这个头部并非“可选”，而是“必需”的。可选头部包含了很多关于可执行映像的重要信息。例如，初始的堆栈大小、程序入口点的位置、首选基地址、操作系统版本、段对齐的信息等。IMAGE_OPTIONAL_HEADER结构

如下:

|

```
#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES    16
```

```
typedef struct _IMAGE_OPTIONAL_HEADER {
```

```
    //
```

```
    // 标准域
```

```
    //
```

```
    WORD    Magic;
```

```
    BYTE    MajorLinkerVersion;
```

```
    BYTE    MinorLinkerVersion;
```

```
    DWORD    SizeOfCode;
```

```
    DWORD    SizeOfInitializedData;
```

```
    DWORD    SizeOfUninitializedData;
```

```
    DWORD    AddressOfEntryPoint;
```

```
    DWORD    BaseOfCode;
```

```
    DWORD    BaseOfData;
```

```
    //
```

```
    // NT附加域
```

```
    //
```

```
    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_HEADER, *PIMAGE_OPTIONAL_HEADER;

|
```

其中参数含义如下所述。

Magic：这个值好像总是0x010b。

MajorLinkerVersion及MinorLinkerVersion：链接器的版本号，这个值不太可靠。

SizeOfCode：可执行代码的长度。

SizeOfInitializedData：初始化数据的长度（数据段）。

SizeOfUninitializedData：未初始化数据的长度（bss段）。

AddressOfEntryPoint：代码的入口RVA地址，程序从这儿开始执行，常称为程序的原入口点OEP（Original Entry Point）。

BaseOfCode：可执行代码起始位置。

BaseOfData：初始化数据起始位置。

ImageBase：载入程序首选的RVA地址。这个地址可被Loader改变。

SectionAlignment：段加载后在内存中的对齐方式。

FileAlignment：段在文件中的对齐方式。

MajorOperatingSystemVersion及MinorOperatingSystemVersion：操作系统版本。

MajorImageVersion及MinorImageVersion：程序版本。

MajorSubsystemVersion及MinorSubsystemVersion：子系统版本号，这个域系统支持。例如，程序运行于NT下，子系统版本号如果不是4.0，对话框不能显示3D风格。

Win32VersionValue：这个值总是为0。

SizeOfImage：程序调入后占用内存大小（字节），等于所有段的长度之和。

SizeOfHeaders：所有文件头长度之和，它等于从文件开始到第一个段的原始数据之间的大小。

Checksum：校验和，仅用在驱动程序中，在可执行文件中可能为0。它的计算方法Microsoft不公开，在imagehelp.dll中的ChecksumMappedFile()函数可以计算它。

Subsystem：一个标明可执行文件所期望的子系统的枚举值。

DllCharacteristics：DLL状态。

SizeOfStackReserve：保留堆栈大小。

SizeOfStackCommit：启动后实际申请的堆栈数，可随实际情况变大。

SizeOfHeapReserve：保留堆大小。

SizeOfHeapCommit: 实际堆大小。

LoaderFlags: 目前没有用。

NumberOfRvaAndSizes: 下面的目录表入口个数, 这个值也不可靠, 可用常数IMAGE_NUMBEROF_DIRECTORY_ENTRIES来代替它, 这个值在目前Windows版本中设为16。注意, 如果这个值不等于16, 那么这个数据结构大小就不能固定下来, 也就不能确定其他变量位置。

DataDirectory: 是一个IMAGE_DATA_DIRECTORY数组, 数组元素个数为IMAGE_NUMBEROF_DIRECTORY_ENTRIES, 结构如下:

```
|
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;    // 起始RVA地址
    DWORD   Size;              // 长度
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

2.3.3 IMAGE_SECTION_HEADER头部

PE文件格式中, 所有的节头部位于可选头部之后。每个节头部为40个字节长, 并且没有任何填充信息。节头部被定义为以下的结构:

```
|
#define IMAGE_SIZEOF_SHORT_NAME 8

typedef struct _IMAGE_SECTION_HEADER {
    BYTE   Name[IMAGE_SIZEOF_SHORT_NAME]; // 节表名称,如".text"
    union {
        DWORD   PhysicalAddress;    // 物理地址
        DWORD   VirtualSize;        // 真实长度
    } Misc;
    DWORD   VirtualAddress;          // RVA
    DWORD   SizeOfRawData;           // 物理长度
    DWORD   PointerToRawData;        // 节基于文件的偏移量
```

```

    DWORD   PointerToRelocations;      // 重定位的偏移

    DWORD   PointerToLinenumbers;      // 行号表的偏移

    WORD    NumberOfRelocations;      // 重定位项数目

    WORD    NumberOfLinenumbers;      // 行号表的数目

    DWORD   Characteristics;          // 节属性
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;

|

```

其中IMAGE_SIZEOF_SHORT_NAME等于8。注意，如果不是这个值，那么这个数据结构大小就不能固定下来，也就不能确定其他变量位置。

2.4 如何获取PE文件中的OEP

OEP（Original Entry Point）是每个PE文件被加载时的起始地址，如何获得这个地址很重要，因为修改程序中的这个值是文件加壳和脱壳时的必须步骤，一些黑客程序也是通过修改OEP值来获得对目标程序的控制权从而实施攻击。下面分别介绍如何通过文件直接访问和通过内存映射访问读取OEP值的方法，并给出完整的程序代码。

2.4.1 通过文件读取OEP值

获得OEP值的最简单方法是，直接从一个PE文件中读取OEP。根据以上对PE文件结构的介绍可知，OEP是PE文件的IMAGE_OPTIONAL_HEADER结构的AddressOfEntryPoint成员，在偏移此结构头40个字节处。而IMAGE_OPTIONAL_HEADER在PE文件的起始位置由IMAGE_DOS_HEADER的e_lfanew成员来计算。注意，以上两个结构在PE文件中不是紧跟在一起的，它之间是DOS Stub，而在每个PE文件DOS Stub的长度可能不一定相等。在PE文件的头部是IMAGE_DOS_HEADER结构，读取这个结构可以得到e_lfanew的值，因而可以得到IMAGE_OPTIONAL_HEADER在PE文件中的位置，也就得到了OEP值。以下是通过文件访问的方法读取OEP的程序代码，即：

```

|

// 通过文件读取OEP值

BOOL ReadOEPbyFile(LPCSTR szFileName)

{
    HANDLE hFile;

    // 打开文件

    if ((hFile = CreateFile(szFileName, GENERIC_READ,

```

```
FILE_SHARE_READ, 0, OPEN_EXISTING,  
FILE_FLAG_SEQUENTIAL_SCAN, 0)) == INVALID_HANDLE_VALUE)  
{  
    printf("Can't not open file.\n");  
    return FALSE;  
}  
  
DWORD dwOEP,cbRead;  
IMAGE_DOS_HEADER dos_head[sizeof(IMAGE_DOS_HEADER)];  
if (!ReadFile(hFile, dos_head, sizeof(IMAGE_DOS_HEADER), &cbRead, NULL)){  
    printf("Read image_dos_header failed.\n");  
    CloseHandle(hFile);  
    return FALSE;  
}  
  
int nEntryPos=dos_head->e_lfanew+40;  
SetFilePointer(hFile, nEntryPos, NULL, FILE_BEGIN);  
  
if (!ReadFile(hFile, &dwOEP, sizeof(dwOEP), &cbRead, NULL)){  
    printf("read OEP failed.\n");  
    CloseHandle(hFile);  
    return FALSE;  
}
```



```

// 关闭文件

CloseHandle(hFile);


// 显示OEP地址

printf("OEP by file:%d/n",dwOEP);

return TRUE;

}

```

2.4.2 通过内存映射读取OEP值

获得OEP值的另一种方法是通过内存映射来实现，此方法也需要熟悉PE的文件结构。与直接访问PE的方法不同，内存映射的方法首先把PE文件映射到计算机的内存，再通过内存的基指针获得IMAGE_DOS_HEADER的头指针，由此再获得IMAGE_OPTIONAL_HEADER指针，这样就可以得到AddressOfEntryPoint的值。下面是通过内存映射获得OEP值的方法：

```

|

// 通过文件内存映射读取OEP值

BOOL ReadOEPbyMemory(LPCSTR szFileName)

{

    struct PE_HEADER_MAP

    {

        DWORD signature;

        IMAGE_FILE_HEADER _head;

        IMAGE_OPTIONAL_HEADER opt_head;

        IMAGE_SECTION_HEADER section_header[6];

    } *header;

```

```
HANDLE hFile;
```

```
HANDLE hMapping;
```

```
void *basepointer;
```

```
// 打开文件
```

```
if ((hFile = CreateFile(szFileName, GENERIC_READ,  
    FILE_SHARE_READ,0,OPEN_EXISTING,  
    FILE_FLAG_SEQUENTIAL_SCAN,0)) == INVALID_HANDLE_VALUE)
```

```
{  
    printf("Can't open file.\n");  
    return FALSE;
```

```
}
```

```
// 创建内存映射文件
```

```
if (!(hMapping = CreateFileMapping(hFile,0,PAGE_READONLY|SEC_COMMIT, 0,0,0)))
```

```
{  
    printf("Mapping failed.\n");  
    CloseHandle(hFile);  
    return FALSE;
```

```
}
```

```
// 把文件头映象存入basepointer
```

```
if (!(basepointer = MapViewOfFile(hMapping,FILE_MAP_READ,0,0,0)))
```

```
{

    printf("View failed./n");

    CloseHandle(hMapping);

    CloseHandle(hFile);

    return FALSE;

}

IMAGE_DOS_HEADER * dos_head =(IMAGE_DOS_HEADER *)basepointer;


// 得到PE文件头

header = (PE_HEADER_MAP *)((char *)dos_head + dos_head->e_lfanew);


// 得到OEP地址.

DWORD dwOEP=header->opt_head.AddressOfEntryPoint;


// 清除内存映射和关闭文件

UnmapViewOfFile(basepointer);

CloseHandle(hMapping);

CloseHandle(hFile);


// 显示OEP地址

printf("OEP by memory:%d/n",dwOEP);

return TRUE;
```

```
}
```

2.4.3 读取OEP值方法的测试

为了检验以上两种获取OEP值方法的正确性和一致性，可以用以下的方法来测试：

```
|
```

```
// oep.cpp:读取OEP的实例
```

```
//
```

```
#include <windows.h>
```

```
#include <stdio.h>
```

```
BOOL ReadOEPbyMemory(LPCSTR szFileName);
```

```
BOOL ReadOEPbyFile(LPCSTR szFileName);
```

```
void main()
```

```
{
```

```
    ReadOEPbyFile("../calc.exe");
```

```
    ReadOEPbyMemory("../calc.exe");
```

```
}
```

```
|
```

运行以上代码后，可以得到如图2.3所示的结果。从图中可以看出，以上两种获取OEP值方法所得到的结果是一致的。

获取**OEP**值方法的测试结果

2.5 PE文件中的资源



一些PE格式（Portable Executable）的EXE文件常常存在很多资源，如图标、位图、对话框、声音等。若要把这些资源取出为自己所用，或修改这些文件中的资源，则需要对PE文件中资源数据结构有所了解。

2.5.1 查找资源在文件中的起始位置

要找出一个PE文件中的某种资源，首先需要确定资源节在PE文件中的起始位置。有两种方法来确定资源在文件中的起始位置。

第一种方法，首先根据FileHeader中的成员NumberOfSections的值，确定文件中节的数目，再根据节的数目，遍历节表数组。也就是从0到（节表数-1）的每一个节表项。比较每一个节表项的Name字段，看看是否等于“.rsrc”，如果是，就找到了资源节的节表项。这个节表项的PointerToRawData 中的值，就是资源节在文件中的位置。

第二种方法，取得PE Header中的IMAGE_OPTIONAL_HEADER中的DataDirectory数组中的第三项，也就是资源项。DataDirectory[]数组的每项都是IMAGE_DATA_DIRECTORY结构，该结构定义如下：

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

从以上结构对象取得DataDirectory数组中的第三项中的成员VirtualAddress的值。这个值就是在内存中资源节的RVA。然后根据节的数目，遍历节表数组，也就是从0 ~ （节表数-1）的每一个节表项。每个节在内存中的RVA的范围是从该节表项的成员VirtualAddress字段的值开始（包括这个值），到VirtualAddress+Misc.VirtualSize的值结束（不包括这个值）。遍历整个节表，看看所取得的资源节的RVA是否在那个节表项的RVA范围之内。如果在范围之内，就找到了资源节的节表项。这个节表项中的PointerToRawData 中的值，就是资源节在文件中的位置。如果这个PE文件没有资源 的话，DataDirectory数组中的第三项内容为0。这样也可以得到了资源在文件中开始的位置。

2.5.2 确定PE文件中的资源

得到了资源节在文件中的位置后，就可以确定某个资源类型及其二进制数据在PE文件中的位置和数据块的大小。

资源节最开始是一个IMAGE_RESOURCE_DIRECTORY结构，在winnt.h文件中有这个结构的定义。这个结构长度为16字节，共有6个参数，其结构的原型如下：

```
|  
  
typedef struct _IMAGE_RESOURCE_DIRECTORY {  
  
    DWORD Characteristics;  
  
    DWORD TimeDateStamp;  
  
    WORD MajorVersion;  
  
    WORD MinorVersion;  
  
    WORD NumberOfNamedEntries;
```

```
WORD NumberOfIdEntries;

// IMAGE_RESOURCE_DIRECTORY_ENTRY DirectoryEntries[];
} IMAGE_RESOURCE_DIRECTORY, *PIMAGE_RESOURCE_DIRECTORY;

|
```

其中各个参数的含义如下所述

Characteristics：标识此资源的类型。

TimeStamp：资源编译器产生资源的时间。

MajorVersion：资源主版本号。

MinorVersion：资源次版本号。

NumberOfNamedEntries和NumberOfIdEntries：分别为用字符串和整形数字来进行标识的IMAGE_RESOURCE_DIRECTORY_ENTRY项数组的成员个数。

紧跟着IMAGE_RESOURCE_DIRECTORY后面的是一个IMAGE_RESOURCE_DIRECTORY_ENTRY数组。这个结构长度为8个字节，共有两个字段，每个字段4个字节。其结构原型如下：

```
|
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {

    union {

        struct {

            DWORD NameOffset:31;

            DWORD NameIsString:1;

        };

        DWORD Name;

        WORD Id;

    };

    union {
```

```

    DWORD   OffsetToData;

    struct {

        DWORD   OffsetToDirectory:31;

        DWORD   DataIsDirectory:1;

    };

};

} IMAGE_RESOURCE_DIRECTORY_ENTRY, *PIMAGE_RESOURCE_DIRECTORY_ENTRY;

|

```

其中，对于第一个字段，当其最高位为1（0x80000000）时，这个DWORD剩下的31位表明相对于资源开始位置的偏移，偏移的内容是一个IMAGE_RESOURCE_DIR_STRING_U，用其中的字符串来标明这个资源类型；当第一个字段的最高位为0时，表示这个DWORD的低WORD中的值作为Id标明这个资源类型。

对于第二个字段，当第二个字段的最高位为1时，表示还有下一层的结构。这个DWORD的剩下31位表明一个相对于资源开始位置的偏移，这个偏移的内容将是一个下一层的IMAGE_RESOURCE_DIRECTORY结构；当第二个字段的最高位为0时，表示已经没有下一层的结构了。这个DWORD的剩下31位表明一个相对于资源开始位置的偏移，这个偏移的内容会是一个IMAGE_RESOURCE_DATA_ENTRY结构，此结构会说明资源的位置。对于资源标示号Id，当Id等于1时,表示资源为光标，等于2时表示资源为位图等，等于3时表示资源为图标等。在winuser.h文件中有定义。

标识一个IMAGE_RESOURCE_DIRECTORY_ENTRY一般都是使用Id，就是一个整数。但是也有少数使用IMAGE_RESOURCE_DIR_STRING_U来标识一个资源类型。这个结构定义如下：

```

|

typedef struct _IMAGE_RESOURCE_DIR_STRING_U {

    WORD Length;

    WCHAR NameString[1];

} IMAGE_RESOURCE_DIR_STRING_U, *PIMAGE_RESOURCE_DIR_STRING_U;

|

```

这个结构中将有一个Unicode的字符串，是字对齐的。这个结构的长度可变，由第一个字段Length指明后面的Unicode字符串的长度。

经过3层IMAGE_RESOURCE_DIRECTORY_ENTRY（一般是3层，也有可能更少些）最终可以找到一个IMAGE_RESOURCE_DATA_ENTRY结构，这个结构中存在相应资源的位置和大小。这个结构长16个字节，有4个参数，其原型如下：

```
|  
  
typedef struct _IMAGE_RESOURCE_DATA_ENTRY {  
  
    DWORD OffsetToData;  
  
    DWORD Size;  
  
    DWORD CodePage;  
  
    DWORD Reserved;  
  
} IMAGE_RESOURCE_DATA_ENTRY, *PIMAGE_RESOURCE_DATA_ENTRY;  
  
|
```

其中各个参数的含义如下所述。

OffsetToData：这是一个内存中的RVA，可以用来转化成文件中的位置。用这个值减去资源节的开始RVA，就可以得到相对于资源节开始的偏移。再加上资源节在文件中的开始位置，即节表中资源节中PointerToRawData的值，就是资源在文件中的位置。注意，资源节的开始RVA可以由Optional Header中的DataDirectory数组中的第三项中的VirtualAddress的值得到，或者节表中资源节那项中的VirtualAddress的值得到。

Size：资源的大小，以字节为单位。

CodePage：代码页。

Reserved：保留项。

总之，资源一般使用树来保存，通常包含3层，最高层是类型，其次是名字，最后是语言。在资源节开始的位置，首先是一个IMAGE_RESOURCE_DIRECTORY结构，后面紧跟着IMAGE_RESOURCE_DIRECTORY_ENTRY数组，这个数组的每个元素代表的资源类型不同；通过每个元素，可以找到第二层另一个IMAGE_RESOURCE_DIRECTORY，后面紧跟着IMAGE_RESOURCE_DIRECTORY_ENTRY数组。这一层的数组的每个元素代表的资源名字不同；然后可以找到第三层的每个IMAGE_RESOURCE_DIRECTORY，后面紧跟着IMAGE_RESOURCE_DIRECTORY_ENTRY数组。这一层的数组的每个元素代表的资源语言不同；最后通过每个IMAGE_RESOURCE_DIRECTORY_ENTRY可以找到每个IMAGE_RESOURCE_DATA_ENTRY。通过每个IMAGE_RESOURCE_DATA_ENTRY，就可以找到每个真正的资源。

2.6 一个修改PE可执行文件的完整实例

在下面的实例中，将把一段MessageBoxA()的计算机代码根据PE文件的格式注入到一个PE程序中。有关把代码注入到一个应用程序的技术将在后面的章节专门介绍。

2.6.1 如何获得MessageBoxA代码

要实现代码注入PE程序且能够运行，首先要做的是如何得到这段代码。为了得到这种代码，作者编写了一段汇编源程序 msgbx.asm，然后用RadASM编译器进行编译，当然也可以使用其他的方法来实现代码的注入。编写这段代码最关键的问题是如何把对话框标题字符串和显示字符串一起存放在代码段，以便提取，否则无法提取。下面是生成MessageBoxA()的源代码：

```
|  
  
;msgbx.asm 文件.  
  
;  
  
.386p  
  
.model flat, stdcall  
  
option casemap:none  
  
include      /masm32/include/windows.inc  
  
include      /masm32/include/user32.inc  
  
includelib   /masm32/lib/user32.lib  
  
.code  
  
start:  
  
    push MB_ICONINFORMATION or MB_OK  
  
    call Func1  
  
    db "Test",0  
  
Func1:  
  
    call Func2  
  
    db "Hello",0  
  
Func2:  
  
    push NULL  
  
    call MessageBoxA
```

```
; ret

end start

|
```

其中"Test"是MessageBoxA()对话框的标题, "Hello"是要显示的字符串。Message-BoxA()所用的Windows句柄为NULL。

用RadASM编译器对以上代码编译后, 可以生成一个msgbx.obj文件, 用VC++ 编辑器打开后, 如图2.4所示, 可以查看这个文件的机器代码。

Msgbx.obj文件的机器代码

把图2.4中所选择的计算机机器代码取出变成一个命令字符串, 即:

```
|
unsigned char cmdline[35]={

    0x6a,                // (1) push 命令

    0x40,                // (1) MB_ICONINFORMATION|MB_OK

    0xe8,                // (1) call命令

    0x05,0x00,0x00,0x00,    // (4) 标题字符串字节个数,包括结束位
(DWORD)

    0x54,0x65,0x73,0x74, 0x00,    // (5) "Test",0(标题)

    0xe8,                // (1) call命令

    0x06,0x00,0x00,0x00,    // (4) 标题字符串字节个数,包括结束位
(DWORD)

    0x48,0x65,0x6c,0x6c,0x6f,0x00, // (6) "Hello",0(显示字符串)

    0x6a,                // (1) push 命令

    0x00,                // (1) 窗口句柄hWnd,NULL

    0xe8,                // (1) call命令

    0x00,0x00,0x00,0x00,    // (4) MessageBoxA的地址 (DWORD)
```

```

0x1a,           // (1) 第26位,校验和

0x00,0x00,0x00,0x0b      // (4) 返回地址 (DWORD)

};

|

```

其中()中的数值表示这一行上代码的字节个数。0x6a是汇编语言中的push命令，0xe8是汇编语言中的call命令，而jmp命令为0xe9。“校验和”是从第一个push命令开始计算所得到的字节总数和（包括校验计数位），从以上代码第一个字节开始计数起到“校验和”位正好是第26位字节个数。字符串字节个数位为一个DWORD型，占4个字节，它是按Little-endian的方式存放的，要把这4个字节位的顺序颠倒才能得到实际数值，即把高位字节变成低位，把低位变换到高位。

要把以上代码注入到一个PE文件中，需要修改4个地方：（1）修改PE文件的入口地址，使PE装载机首先装载以上代码；（2）修改以上代码MessageBoxA()的地址，使以上的代码能够显示出一个对话框；（3）把“校验和”位变成跳转位，即变成jmp（0xe9）；（4）修改返回地址，把程序引入到原来的装载点上。

2.6.2 把MessageBoxA()代码写入PE文件的完整实例

根据以上的对MessageBoxA()的分析，可以直接把以上代码注入到一个PE可执行文件中。为了使程序有通用性，这里编写了一个产生显示任意长度字符的对话框的函数WriteMessageBox()。

下面是用于注入MessageBoxA()代码的头文件，取名为Pe.h，其中用#include包含了相关的文件头，定义了peHeader结构，且定义了CPe类，其源代码如下：

```

|

// Pe.h: 定义CPe类

//

#ifndef _PE_H__INCLUDED

#define _PE_H__INCLUDED

#include <io.h>

#include <fcntl.h>

#include <sys/stat.h>

typedef struct PE_HEADER_MAP

{

    DWORD signature;

```

```
    IMAGE_FILE_HEADER _head;

    IMAGE_OPTIONAL_HEADER opt_head;

    IMAGE_SECTION_HEADER section_header[6];
} peHeader;

class CPe
{
public:
    CPe();

    virtual ~CPe();

public:
    void CalcAddress(const void *base);

    void ModifyPe(CString strFileName,CString strMsg);

    void WriteFile(CString strFileName,CString strMsg);

    BOOL WriteNewEntry(int ret,long offset,DWORD dwAddress);

    BOOL WriteMessageBox(int ret,long offset,CString strCap,CString
    strTxt);

    CString StrOfDWord(DWORD dwAddress);

public:
    DWORD dwSpace;

    DWORD dwEntryAddress;

    DWORD dwEntryWrite;

    DWORD dwProgRAV;

    DWORD dwOldEntryAddress;
```

```

    DWORD dwNewEntryAddress;

    DWORD dwCodeOffset;

    DWORD dwPeAddress;

    DWORD dwFlagAddress;

    DWORD dwVirtSize;

    DWORD dwPhysAddress;

    DWORD dwPhysSize;

    DWORD dwMessageBoxAaddress;

};

#endif

|

```

其中peHeader结构是前面所讲的PE Header结构与节表（Section Table）头结构（6个表头成员）的总结构。因为它们在PE文件中是紧凑排列的，所以可以这样写。其实只用一个节表头就可以。

下面分别介绍CPe类成员函数的定义，它们包含在Pe.cpp文件中。在这个文件开始用#include包含了stdafx.h和Pe.h文件。用MFC VC++编译器编译时，必须包括stdafx.h文件，即使这个文件是空的，也需要包括它，这是编译器设置所致，除非修改MFC的编译器的默认设置。CPe类的构造和析构函数这里没有用上，对系统内存的访问和其他操作主要是通过主成员函数ModifyPe()来进行。它们的源代码如下：

```

|

// Pe.cpp: 实现 CPe类

//

#include "stdafx.h"

#include "Pe.h"

CPe::CPe()

{

```

```
}

CPe::~CPe()

{
}

void CPe::ModifyPe(CString strFileName,CString strMsg)

{
    CString strErrMsg;

    HANDLE hFile, hMapping;

    void *basepointer;

    // 打开要修改的文件

    if ((hFile = CreateFile(strFileName, GENERIC_READ|GENERIC_WRITE,

        FILE_SHARE_READ|FILE_SHARE_WRITE, 0,

        OPEN_EXISTING, FILE_FLAG_SEQUENTIAL_SCAN, 0)) == INVALID_HANDLE_VALUE)

    {

        AfxMessageBox("Could not open file.");

        return;

    }

    // 创建一个映射文件

    if (!(hMapping = CreateFileMapping(hFile, 0, PAGE_READONLY | SEC_COMMIT, 0, 0, 0)))

    {

        AfxMessageBox("Mapping failed.");

        CloseHandle(hFile);
```

```
    return;

}

// 把文件头映象存入basepointer

if (!(basepointer = MapViewOfFile(hMapping, FILE_MAP_READ, 0, 0, 0)))

{

    AfxMessageBox("View failed.");

    CloseHandle(hMapping);

    CloseHandle(hFile);

    return;

}

CloseHandle(hMapping);

CloseHandle(hFile);

CalcAddress(basepointer); // 得到相关地址

UnmapViewOfFile(basepointer);


if(dwSpace<50)

{

    AfxMessageBox("No room to write the data!");

}

else

{

    WriteFile(strFileName,strMsg); // 写文件
```

```

}

if ((hFile = CreateFile(strFileName, GENERIC_READ|GENERIC_WRITE,
    FILE_SHARE_READ|FILE_SHARE_WRITE, 0,
    OPEN_EXISTING, FILE_FLAG_SEQUENTIAL_SCAN, 0)) == INVALID_HANDLE_VALUE)
{
    AfxMessageBox("Could not open file.");
    return;
}

CloseHandle(hFile);
}

```

其中对一个PE文件进行MessageBoxA()代码的注入是通过ModifyPe()函数进行，它的入口参数是要被修改的PE可执行文件名。在这个函数中，首先创建所修改文件的句柄，然后创建映射文件，再通过映射文件的句柄获得这个PE文件的文件头指针，最后把这个指针传给函数CalcAddress()。通过CalcAddress()函数来计算PE Header的开始偏移、保存旧的程序入口地址、计算新的程序入口地址和计算PE文件的空隙空间等。

CalcAddress()函数的源代码如下：

```

|
void CPe::CalcAddress(const void *base)
{
    IMAGE_DOS_HEADER * dos_head =(IMAGE_DOS_HEADER *)base;

    if (dos_head->e_magic != IMAGE_DOS_SIGNATURE)
    {
        AfxMessageBox("Unknown type of file.");
        return;
    }
}

```



```
}

peHeader * header;

// 得到PE文件头

header = (peHeader *)((char *)dos_head + dos_head->e_lfanew);

if(IsBadReadPtr(header, sizeof(*header)))

{

    AfxMessageBox("No PE header, probably DOS executable.");

    return;

}

DWORD mods;

char tmpstr[4]={0};

if(strstr((const char *)header->section_header[0].Name, ".text")!=

NULL)

{

    // 此段的真实长度

    dwVirtSize=header->section_header[0].Misc.VirtualSize;

    // 此段的物理偏移

    dwPhysAddress=header->section_header[0].PointerToRawData;

    // 此段的物理长度

    dwPhysSize=header->section_header[0].SizeOfRawData;
```

```
// 得到PE文件头的开始偏移
```

```
dwPeAddress=dos_head->e_lfanew;
```

```
// 得到代码段的可用空间，用以◆%
```

广告