

本章提要

- PE 文件格式概述
- PE 文件结构
- 如何获取 PE 文件中的 OEP
- 如何获取 PE 文件中的资源
- 如何修改 PE 文件使其显示 MessageBox 的实例

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：

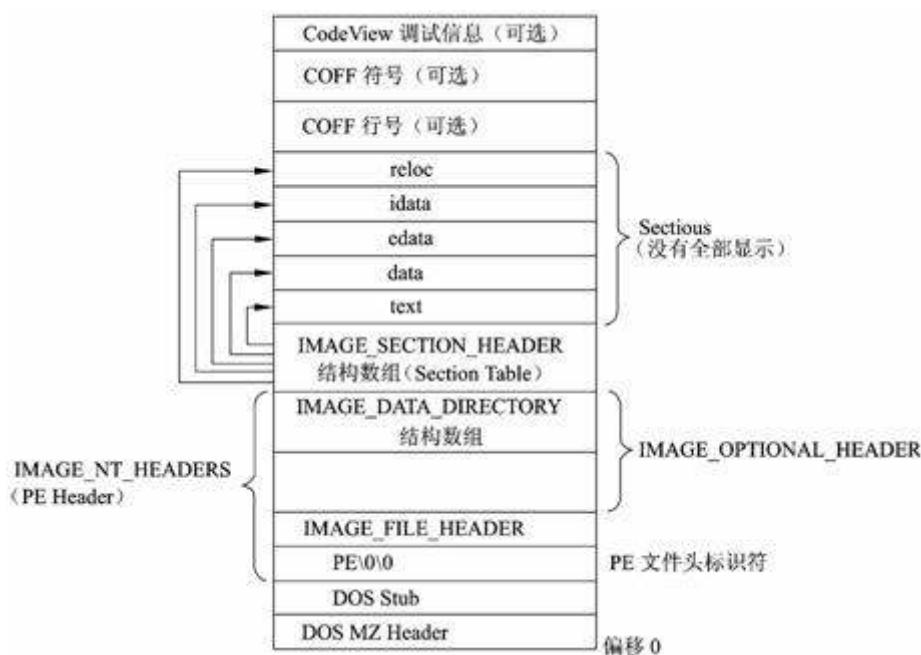


图 2.1 PE 文件结构总体层次分布

- 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。

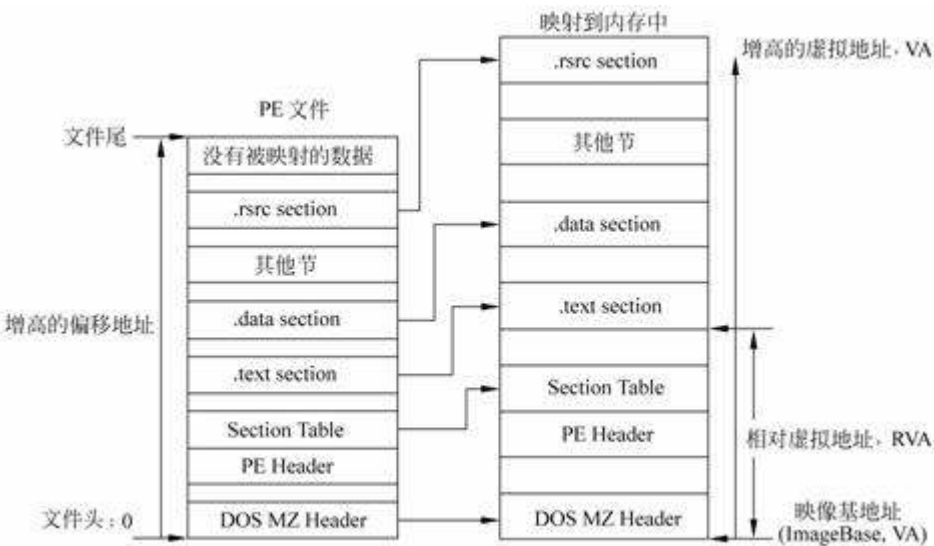


图 2.2 PE 文件内存映射

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。有些是指文件中的偏移，有些是指内存中的偏移。以下的第一种是指文件中的地址，第二、三种是指内存中的地址。

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载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：

1

// 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:

1

```
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:

1

```
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
```

```
1
```

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。

 表 2.1 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：段的个数。

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：

```
1

#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;

```

1

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:

1

```
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:

1

```
#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;

1

```

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：

```
1

// 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;
}

// 把文件头映像存入 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：

```

1

// oep.cpp:读取 OEP 的实例

//

#include <windows.h>

#include <stdio.h>

BOOL ReadOEPbyMemory(LPCSTR szFileName);

BOOL ReadOEPbyFile(LPCSTR szFileName);

void main()

{

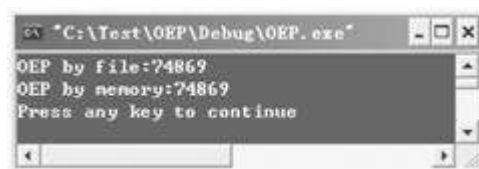
    ReadOEPbyFile("../\\calc.exe");

    ReadOEPbyMemory("../\\calc.exe");
}

```

```
}  
  
1
```

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。



 2.3 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：

```
1
```

```
typedef struct _IMAGE_DATA_DIRECTORY {
```

```

        DWORD VirtualAddress;

        DWORD Size;

    } IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;

```

1

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：

1

```

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;
```

1

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：

1

```
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;

1

```

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：

```

1

typedef struct _IMAGE_RESOURCE_DIR_STRING_U {

    WORD Length;

    WCHAR NameString[1];

} IMAGE_RESOURCE_DIR_STRING_U, *PIMAGE_RESOURCE_DIR_STRING_U;

1

```

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,这个结构中存有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:

1

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

1

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：

```
1

;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
```

1

其中“Test”是 MessageBoxA() 对话框的标题, “Hello”是要显示的字符串。MessageBoxA() 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。



图 2.4 Msgbx.obj 文件的机器代码

把图 2.4 中所选择的计算机器代码取出变成一个命令字符串, 即:

1

```
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)

};

```

1

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 装载器首先装载以上代码；（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：

```
1

// 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 dwMessageBoxAadaddress;

};

#endif

1

```

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:

```

1

// 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：

```

1
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，用以判断可不可以写入我们的代码

    // 用此段的物理长度减去此段的真实长度就可以得到

    dwSpace=dwPhysSize-dwVirtSize;

```

```

// 得到程序的装载地址，一般为 0x400000

dwProgRAV=header->opt_head. ImageBase;

// 得到代码偏移，用代码段起始 RVA 减去此段的物理偏移

// 应为程序的入口计算公式是一个相对的偏移地址，计算公式为：

// 代码的写入地址+dwCodeOffset

dwCodeOffset=header->opt_head. BaseOfCode-dwPhysAddress;


// 代码写入的物理偏移

dwEntryWrite=header->section_header[0]. PointerToRawData+head
r->

    section_header[0]. Misc. VirtualSize;

//对齐边界

mods=dwEntryWrite%16;

if(mods!=0)

{

    dwEntryWrite+=(16-mods);

}


// 保存旧的程序入口地址

dwOldEntryAddress=header->opt_head. AddressOfEntryPoint;

// 计算新的程序入口地址

dwNewEntryAddress=dwEntryWrite+dwCodeOffset;

return;

}

```

```
}
```

```
1
```

下面的 StrOfDWord() 函数是把一个 DWORD 值转换成一个字符串，因为一个 DWORD 值占有 4 个字节，因此把一个 DWORD 值变成一个字符串，若保持数值不变，就变成了一个 4 个字节的字符串。同时把这个值的位置顺序颠倒，这是为了把一个实际的值变成按 Little-endian 的方式写入 PE 文件中，其转换方法如下：

```
1
```

```
CString CPe::StrOfDWord(DWORD dwAddress)
```

```
{
```

```
    unsigned char waddress[4]={0};
```

```
    waddress[3]=(char) (dwAddress>>24)&0xFF;
```

```
    waddress[2]=(char) (dwAddress>>16)&0xFF;
```

```
    waddress[1]=(char) (dwAddress>>8)&0xFF;
```

```
    waddress[0]=(char) (dwAddress)&0xFF;
```

```
    return waddress;
```

```
}
```

```
1
```

下面的 WriteNewEntry() 函数把新的入口点写入 PE 程序原来的入口点处，使 PE 装载器在载入程序时，直接跳入到 MessageBoxA() 的入口处，该函数的源代码如下：

```
1
```

```
BOOL CPe::WriteNewEntry(int ret, long offset, DWORD dwAddress)
```

```
{
```

```
    CString strErrMsg;
```

```

long retf;

unsigned char waddress[4]={0};

retf=_lseek(ret, offset, SEEK_SET);

if(retf==-1)

{

    AfxMessageBox("Error seek.");

    return FALSE;

}

memcpy(waddress, StrOfDWord(dwAddress), 4);

retf=_write(ret, waddress, 4);

if(retf==-1)

{

    strErrMsg.Format("Error write: %d", GetLastError());

    AfxMessageBox(strErrMsg);

    return FALSE;

}

return TRUE;

}

1

```

下面的 WriteMessageBox() 函数是把 MessageBoxA() 的机器代码写入到 PE 文件中。这个函数显示的对话框标题和显示的字符串内容和长度不是固定的。在这个函数中，首先就计算 MessageBoxA() 函数的地址和函数的返回地址，然后把重新生成的对话框代码写入到程序中。 WriteMessageBox() 函数的源代码如下：

1

```
BOOL CPe::WriteMessageBox(int ret, long offset, CString strCap, CString
strTxt)
{
    CString strAddress1, strAddress2;

    unsigned char waddress[4]={0};

    DWORD dwAddress;

    // 获取 MessageBox 在内存中的地址

    HINSTANCE gLibMsg=LoadLibrary("user32.dll");

    dwMessageBoxAaddress=(DWORD)GetProcAddress(gLibMsg, "MessageBoxA
");

    // 计算校验位

    int nLenCap1 =strCap.GetLength()+1;    // 加上字符串后面的结束位

    int nLenTxt1 =strTxt.GetLength()+1;    // 加上字符串后面的结束位

    int nTotLen=nLenCap1+nLenTxt1+24;

    // 重新计算 MessageBox 函数的地址

    dwAddress=dwMessageBoxAaddress-(dwProgRAV+dwNewEntryAddress+nTo
t
Len-5);

    strAddress1=StrOfDWord(dwAddress);

    // 计算返回地址

    dwAddress=0-(dwNewEntryAddress-dwOldEntryAddress+nTotLen);

    strAddress2=StrOfDWord(dwAddress);

    // 对话框头代码(固定)
```

```

unsigned char cHeader[2]={0x6a, 0x40};

// 标题定义

unsigned char cDesCap[5]={0xe8, nLenCap1, 0x00, 0x00, 0x00};

// 内容定义

unsigned char cDesTxt[5]={0xe8, nLenTxt1, 0x00, 0x00, 0x00};

// 对话框后部分的代码段

unsigned char cFix[12]

    ={0x6a, 0x00, 0xe8, 0x00, 0x00, 0x00, 0x00, 0xe9, 0x00, 0x00, 0x00, 0x00
};

// 修改对话框后部分的代码段

for(int i=0;i<4;i++)

    cFix[3+i]=strAddress1.GetAt(i);

for(i=0;i<4;i++)

    cFix[8+i]=strAddress2.GetAt(i);

char* cMessageBox=new char[nTotLen];

char* cMsg;

// 生成对话框命令字符串

memcpy((cMsg = cMessageBox), (char*)cHeader, 2);

memcpy((cMsg += 2), cDesCap, 5);

memcpy((cMsg += 5), strCap, nLenCap1);

```

```

memcpy((cMsg += nLenCap1), cDesTxt, 5);

memcpy((cMsg += 5), strTxt, nLenTxt1);

memcpy((cMsg += nLenTxt1), cFix, 12);

// 向应用程序写入对话框代码

CString strErrMsg;

long retf;

retf=_lseek(ret, (long)dwEntryWrite, SEEK_SET);

if(retf==-1)

{

    delete[] cMessageBox;

    AfxMessageBox("Error seek.");

    return FALSE;

}

retf=_write(ret, cMessageBox, nTotLen);

if(retf==-1)

{

    delete[] cMessageBox;

    strErrMsg.Format("Error write: %d", GetLastError());

    AfxMessageBox(strErrMsg);

    return FALSE;

}

delete[] cMessageBox;

return TRUE;

}

```

1

下面的 WriteFile() 函数是总的写入函数。在这个函数中，先打开被修改的 PE 文件，然后调用 WriteNewEntry() 和 WriteMessageBox() 函数。WriteFile() 函数的源代码如下：

1

```
void CPe::WriteFile(CString strFileName)
{
    CString strAddress1, strAddress2;

    int ret;

    unsigned char waddress[4]={0};

    ret=_open(strFileName, _O_RDWR | _O_CREAT | _O_BINARY, _S_IREAD | _S_
IWRITE);

    if(!ret)
    {
        AfxMessageBox("Error open");

        return;
    }

    // 把新的入口地址写入文件, 程序的入口地址在偏移 PE 文件头开始第 40
位

    if(!WriteNewEntry(ret, (long)(dwPeAddress+40), dwNewEntryAddress))

    return;

    // 把对话框代码写入到应用程序中

    if(!WriteMessageBox(ret, (long)dwEntryWrite, "Test", "We are the
world!"))
    return;
```



```
        _close(ret);  
    }  
  
1
```

仅仅利用以上 CPe 类还是不能对一个 PE 文件进行注入 MessageBoxA() 代码的修改，还必须要一个“载体程序”。例如：

```
1  
  
// Pefile.cpp:修改 PE 文件实例  
  
//  
  
#include "stdafx.h"  
  
#include "Pe.h"  
  
void main()  
{  
  
    CopyFile("../\\calc.exe", "../\\calc_shell.exe", FALSE);  
  
  
    CPe a;  
  
    a.ModifyPe("../\\calc_shell.exe", "We are the world!");  
}  
  
1
```

这个修改后的 PE 文件运行时，就会先显示对话框，单击“确定”按钮后又继续执行。总之，在了解了 PE 文件格式后，就可以对某一个 PE 文件进行修改。本实例只是对 PE 文件处理的一种应用，在实际中还有更多的其他方面的应用。

2.7 本章小结



本章首先介绍了 PE 文件的基本结构，对一些容易混淆的名词进行了解释。通过介绍一个对 PE 文件注入对话框代码的实例，加强了对 PE 文件结构的认识。

本章所介绍的向 PE 文件注入代码的实例只是用来说明如何修改 PE 文件，有关如何向一个应用程序中注入代码的技术还要在以后的章节专门介绍。此外，还有其他的 技术没有介绍，例如如何提取程序中的代码，在以后的章节中对此也还要专门介绍。总之，了解了 PE 文件结构，就可以很容易地对某个应用程序进行加壳、挂钩或 捆绑。