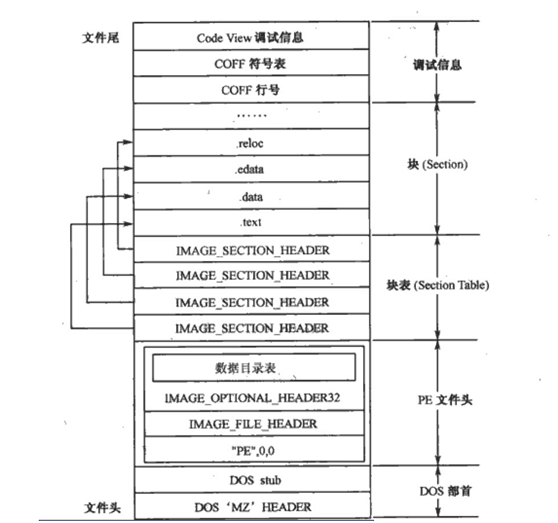
一个pe文件主要由 DOS MZ , DOS 块, PE 文件头,节表,节等模块组成.



1. DOS MZ：首先将一个exe文件载入，判断一个文件是否是PE文件。根据文件的前两个字节是否为4D 5A，也就是’MZ’，然后看第四排四个字节指向的地址00 00 00 D0是否为50 45，也就是’PE’，满足这两个条件也就满足了PE文件的格式，简称PE指纹

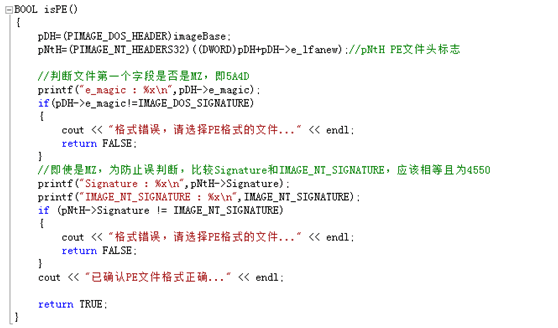
(4D5A: 4即0100，D即1101，5即0101，A即1010。一个字节8bit，所以分为01001101和01011010，分别对应77（M），90（Z）)。

DOS MZ我们需要熟悉的是e\_magic成员和e\_lfanew成员，前者是标识PE指纹的一部分，后者则是寻找PE文件头的部分。

第一个成员e\_magic,对应4D 5A两个字节

最后一个成员e\_lfanew,对应 00 00 00 D0 (小尾方式) 最后一个成员是保存了新的文件格式地址.所以我们看到D0的位置,正好是PE

那么我们如果不是在16位程序下使用,那么上面除了最后一个成员,其余的位置都可以随便改.如果是16位系统下,那么你这个DOS头记录的信息就有用了(保存了页大小,页的个数,SS段.IP执行位置,校验和等等)。



2. DOS块：e\_lfanew指向PE文件头，我们可以通过它来寻找PE文件头，而DOS块的部分自然就是DOS MZ文件头和PE文件头中间的部分，这部分是由链接器所写入的，可以随意进行修改，并不影响程序的运行。DOS MZ中还有一个成员 WORD e\_lfarlc; 记录的是DOS的代码执行位置（即DOS块）,这块区域属于DOS的代码执行区域。主要作用是,在16位系统下,不能运行32位程序,如果运行,则利用中断,显示 This is Program cannot be run in Dos Mode (这个程序不能运行在DOS系统)。如果在32位系统下使用,那么这一段是没有任何作用的。

3.PE文件头由PE文件头标志，标准PE头，扩展PE头三部分组成。

typedef struct \_IMAGE\_NT\_HEADERS {

    DWORD Signature;                                      //PE文件头标志 => 4字节

    IMAGE\_FILE\_HEADER FileHeader;                  //标准PE头 => 20字节

    IMAGE\_OPTIONAL\_HEADER32 OptionalHeader; //扩展PE头 => 32位下224字节(0xE0) 64位下240字节(0xF0)

} IMAGE\_NT\_HEADERS32, \*PIMAGE\_NT\_HEADERS32;

(1)PE文件头标志自然是50 45 00 00，也就是’PE’。

(2)标准PE头结构如下，有20个字节，我们可以从PE文件头标志后20个字节找到它

typedef struct \_IMAGE\_FILE\_HEADER {

    WORD    Machine;                             //可以运行在什么平台上 任意:0 ,Intel 386以及后续:14C x64:8664

    WORD    NumberOfSections;            //节的数量

    DWORD   TimeDateStamp;                //编译器填写的时间戳

    DWORD   PointerToSymbolTable;   //调试相关

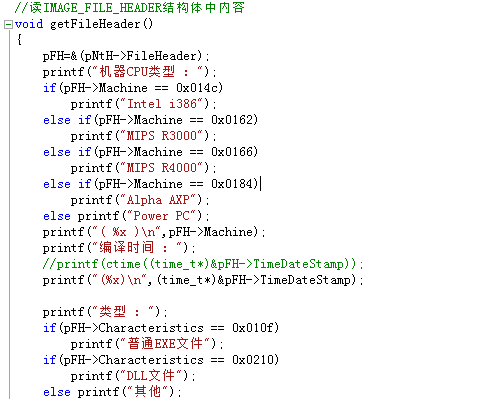
    DWORD   NumberOfSymbols;          //调试相关

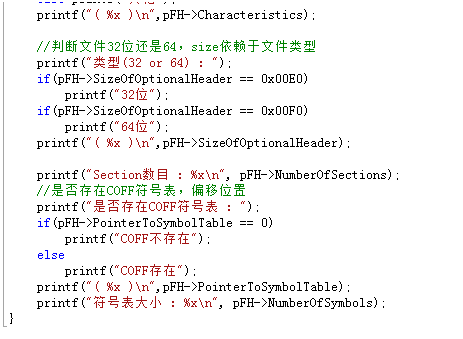
    WORD    SizeOfOptionalHeader;   //标识扩展PE头大小

    WORD    Characteristics;        //文件属性 => 16进制转换为2进制根据哪些位有1,可以查看相关属性

} IMAGE\_FILE\_HEADER, \*PIMAGE\_FILE\_HEADER;

核心代码截图





(3)扩展PE头在32位和64位系统上大小是不同的，在32位系统上有224个字节（32位大小是E0，64位大小是F0）

typedef struct \_IMAGE\_OPTIONAL\_HEADER {

    //

    // Standard fields.

    //

    WORD    Magic;                                         //PE32: 10B PE64: 20B

    BYTE    MajorLinkerVersion;

    BYTE    MinorLinkerVersion;

    DWORD   SizeOfCode;                              //所有含有代码的区块的大小 编译器填入 没用(可改)

    DWORD   SizeOfInitializedData;         //所有初始化数据区块的大小 编译器填入 没用(可改)

    DWORD   SizeOfUninitializedData;     //所有含未初始化数据区块的大小 编译器填入 没用(可改)

    DWORD   AddressOfEntryPoint;        //程序入口RVA

    DWORD   BaseOfCode;                             //代码区块起始RVA

    DWORD   BaseOfData;                              //数据区块起始RVA

    //

    // NT additional fields.

    //

    DWORD   ImageBase;                                      //内存镜像基址(程序默认载入基地址)

    DWORD   SectionAlignment;                            //内存中对齐大小

    DWORD   FileAlignment;                                  //文件中对齐大小(提高程序运行效率)

    WORD    MajorOperatingSystemVersion;

    WORD    MinorOperatingSystemVersion;

    WORD    MajorImageVersion;

    WORD    MinorImageVersion;

    WORD    MajorSubsystemVersion;

    WORD    MinorSubsystemVersion;

    DWORD   Win32VersionValue;

    DWORD   SizeOfImage;                                   //内存中整个PE文件的映射的尺寸,可比实际值大,必须是SectionAlignment的整数倍

    DWORD   SizeOfHeaders;                               //所有的头加上节表文件对齐之后的值

    DWORD   CheckSum;                                      //映像校验和,一些系统.dll文件有要求,判断是否被修改

    WORD    Subsystem;

    WORD    DllCharacteristics;                      //文件特性,不是针对DLL文件的,16进制转换2进制可以根据属性对应的表格得到相应的属性

    DWORD   SizeOfStackReserve;

    DWORD   SizeOfStackCommit;

    DWORD   SizeOfHeapReserve;

    DWORD   SizeOfHeapCommit;

    DWORD   LoaderFlags;

    DWORD   NumberOfRvaAndSizes;

    IMAGE\_DATA\_DIRECTORY DataDirectory[IMAGE\_NUMBEROF\_DIRECTORY\_ENTRIES]; //数据目录表,结构体数组

} IMAGE\_OPTIONAL\_HEADER32, \*PIMAGE\_OPTIONAL\_HEADER32;

PS：程序的真正入口点 = ImageBase + AddressOfEntryPoint

重点：其中的DataDirectory数据目录，这是一个数组，数组中的每一项对应一个特定的数据结构，包括导入表、导出表等等。数据目录表（结构体数组）

其中每一项的定义如下：

typedef struct \_IMAGE\_DATA\_DIRECTORY {

    DWORD   VirtualAddress;

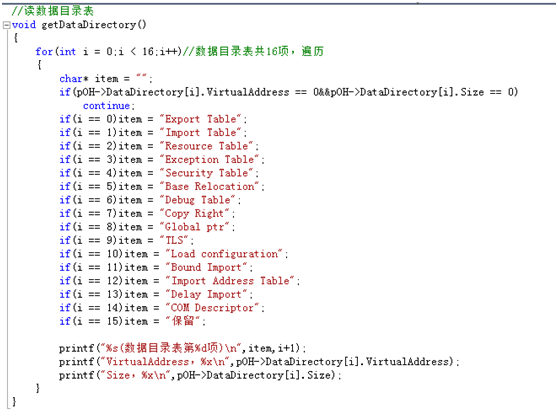
    DWORD   Size;

} IMAGE\_DATA\_DIRECTORY, \*PIMAGE\_DATA\_DIRECTORY;

VirtualAddress定义了该项的相对虚拟地址

Size定义了该项的大小

核心代码



在这些成员中比较重要的就是前两个，导出表和导入表

导出表：Windows在加载一个程序后会在内存中为该程序开辟一个单独的虚拟地址空间，程序用到的函数会加载到其地址空间运行。而有一些函数会有很多程序都能用到，因此采用将一些函数封装成动态链接库，程序在需要时加载动态链接库的方式可以大大节约内存空间。

导出表是用来记载动态链接库的一些导出信息的结构，其主要成分是一个表格，内含函数名称、输出序数等等。结构体定义如下：

typedef struct \_IMAGE\_EXPORT\_DIRECTORY {

    DWORD   Characteristics;

    DWORD   TimeDateStamp;

    WORD    MajorVersion;

    WORD    MinorVersion;

    DWORD   Name; // 指针指向该导出表文件名字符串

    DWORD   Base; // 导出函数起始序号

    DWORD   NumberOfFunctions; // 所有导出函数的个数

    DWORD   NumberOfNames; // 以函数名字导出的函数个数

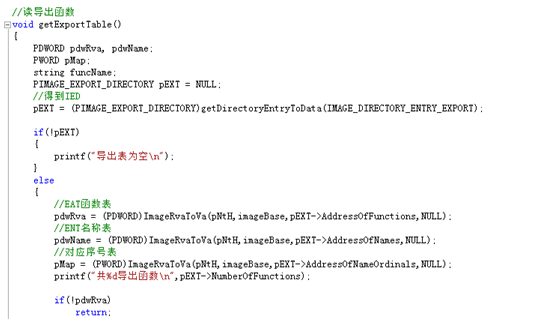
    DWORD   AddressOfFunctions;     // 指针指向导出函数地址表RVA

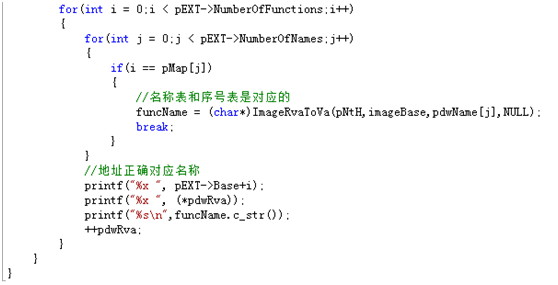
    DWORD   AddressOfNames;         // 指针指向导出函数名称表RVA

    DWORD   AddressOfNameOrdinals;  // 指针指向导出函数序号表RVA

} IMAGE\_EXPORT\_DIRECTORY, \*PIMAGE\_EXPORT\_DIRECTORY;

核心代码





导入表：在PE文件加载时，会根据这个表里的内容加载依赖的DLL，并填充所需函数的地址。

导入表结构如下：

typedef struct \_IMAGE\_IMPORT\_DESCRIPTOR {

    union {

        DWORD   Characteristics;            // 0 for terminating null import descriptor

        DWORD   OriginalFirstThunk;         // RVA 指向 INT (PIMAGE\_THUNK\_DATA结构数组)

    } DUMMYUNIONNAME;

    DWORD   TimeDateStamp;                  // 0 if not bound,

                                            // -1 if bound, and real date\time stamp

                                            //     in IMAGE\_DIRECTORY\_ENTRY\_BOUND\_IMPORT (new BIND)

                                            // O.W. date/time stamp of DLL bound to (Old BIND)

    DWORD   ForwarderChain;                 // -1 if no forwarders

    DWORD   Name; //RVA指向dll名字,以0结尾

    DWORD   FirstThunk;                     // RVA 指向 IAT (PIMAGE\_THUNK\_DATA结构数组)

} IMAGE\_IMPORT\_DESCRIPTOR;

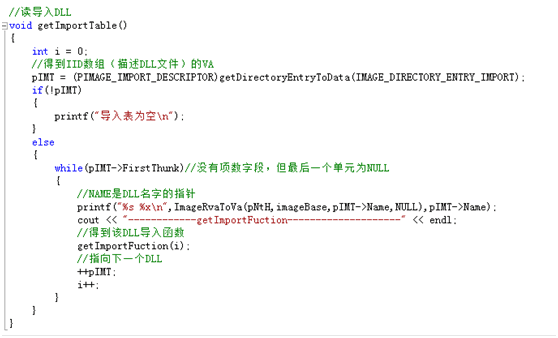
typedef IMAGE\_IMPORT\_DESCRIPTOR UNALIGNED \*PIMAGE\_IMPORT\_DESCRIPTOR;

FirstThunk和OriginalFirstThunk都指向IMAGE\_THUNK\_DATA数组，但是却是不同的。OriginalFirstThunk指向的是单独的一项，而且不能被修改，成为INT。FirstThunk指向的事实上是由PE加载器重写的。

在PE文件加载以前或者说在导入表未处理以前，FirstThunk所指向的数组与OriginalFirstThunk中的数组虽不是同一个，但是内容却是相同的，都包含了导入信息，而在加载之后，FirstThunk中的Function开始生效，他指向实际的函数地址，因为FirstThunk实际上指向IAT中的一个位置，IAT就充当了IMAGE\_THUNK\_DATA数组，加载完成后，这些IAT项就变成了实际的函数地址，即Function的意义。

PE加载器首先搜索OriginalFirstThunk，找到之后加载程序迭代搜索数组中的每个指针，找到每个IMAGE\_IMPORT\_BY\_NAME结构所指向的输入函数的地址，然后加载器用函数真正入口地址来代替由FirstThunk数组中的一个入口，因此称为输入地址表IAT。

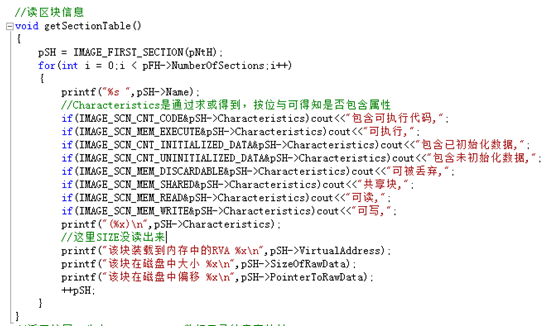
核心代码：



4.节表

节表由一系列的IMAGE\_SECTION\_HEADER结构排列而成，每个结构用来描述一个节。其排列顺序和节在文件中的排列顺序一致。节表以一个空的IMAGE\_SECTION\_HEADER结构作为结尾，因此节表中IMAGE\_SECTION\_HEADER结构的总数等于节的数量+1。

核心代码：



6.具体的节

.idata节

导入段。包含程序需要的所有DLL文件信息。

.data节

数据段（data segment）通常是指用来存放程序中已初始化的全局变量的一块内存区域。数据段属于静态内存分配。

根据示例程序中

通过010editor，我们可以在.data段找到如下信息

.text/CODE节

代码段（text segment）通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域属于只读。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。需要注意的是，borland这里叫做code，而不是text 。

.rdata节

.rdata节一般最少两种用途。通常.data节的开头是程序的Debug目录，该目录仅存在于.EXE文件当中。Debug目录是一个 IMAGE\_DEBUG\_DIRECTORY 结构体的数组，这些结构体存储了文件各种不同的Debug信息。

同时.rdata节有一部分是用来描述字符串信息。如果程序的DEF文件中存在特别指定的DESCRIPTION entry，那么这个DESCRIPTION entry就会出现在.rdata节中。

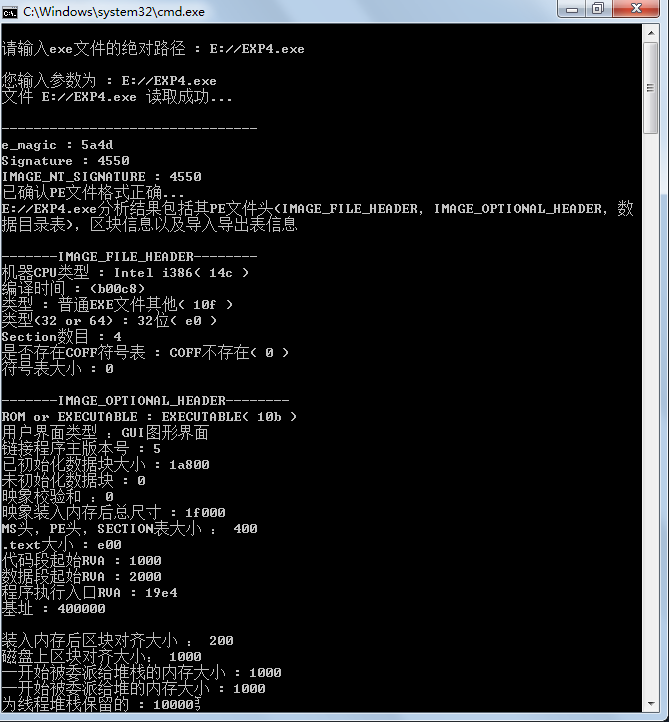
.CRT节

.CRT节是另一个 Microsoft C/C++ run-time libraries的初始化数据段。

.tls节

.tls=thread local storage，当用户使用compiler directive \_ \_declspec线程时，定义的数据会出现在当前段。当处理.tls节时，内存管理器会建立一个页表来保证进程切换线程时，一组新的物理存储页会映射到.tls节的地址空间。

运行结果截图：



总结：PE文件解析实际上就是分析结构体之间的跳转和嵌套。

需要注意的一点是里面的地址是RVA(相对虚拟地址），要得到虚拟地址，还需要调用ImageRvaToVa函数

ImageRvaToVa(NtHeaders，Base，Rva，LastRvaSection)

定位一个映射为文件的文件的图像头中的相对虚拟地址(RVA)，并返回文件中相应字节的虚拟地址。

PVOID IMAGEAPI ImageRvaToVa(

  PIMAGE\_NT\_HEADERS     NtHeaders,

  PVOID                 Base,

  ULONG                 Rva,

  PIMAGE\_SECTION\_HEADER \*LastRvaSection

);

形式参数解析

NtHeaders

一个指向IMAGE NT HEADERS结构的指针。这个结构可以通过调用ImageNtHeader函数来获得。

Base

通过调用MapViewOfFile函数映射到内存中的形象的基址。

Rva

要定位的相对虚拟地址。

LastRvaSection

一个指向IMAGE SECTION HEADER结构的指针，该结构指定最后一个RVA SECTION。这是一个可选参数。当指定时，它指向一个变量，该变量包含用于将RVA转换为VA的指定图像的最后一个section值。