# PE 可执行文件通用加密工具的设计与实现

## Design and Implementation of General Encryption Tool for Portable Executable File

张建明　(益阳 湖南城市学院计算机科学系 413049、长沙 湖南大学计算机与通信学院 410082)
林亚平　(长沙 湖南大学计算机与通信学院 410082)

**摘要**:深入分析了 PE 文件的格式及其在 Windows 操作系统下的装入机制,提出了 PE 可执行文件的加密方法。设计了给任意 PE 可执行文件进行加密、加口令的工具,并给出了实现要点。提出了壳程序中调用 API 函数的两种寻址方法。
**关键词**:PE 文件　加密与解密　口令　导入表

## 1　引言

Microsoft 在 1993 年发行 Windows NT 这一全新 32 位操作系统时推出了一种新的文件格式 Portable Executable File Format(可移植执行体文件格式,简称为 PE 格式)。该格式是迄今所有 32 位 Windows 系列操作系统下可执行文件、动态链接库文件的格式;目标文件、静态库文件以及 NT 中的设备驱动程序也是 PE 格式的。软件一般以可执行文件形式存在。目前使用的主流操作系统是 Windows 平台,为了对软件进行保护,必须深入剖析 PE 文件的格式。

## 2　PE 文件的格式及装入机制

从某种意义上说,操作系统的可执行文件格式是这个操作系统的一面镜子。可执行文件中包含了操作系统装入此可执行文件去运行时所需的各种控制信息。PE 格式在很多地方继承了 VMS 和 Unix 中的 COFF 格式,在头文件 WINNT.H 中可找到用 C 语言说明的有关 PE 文件组成部分的类型定义。

PE 文件中的许多域是用相对虚拟地址(RVA)来定义的。RVA 是某对象相对于文件映射到内存的开始地址(基地址)的偏移量。为了将 RVA 转换为可用的指针,只需将 RVA 加上模块的基地址。

### 2.1　MS－DOS 头

MS－DOS 头占据了 PE 文件最开始的 64 个字节,最开始的两个字节的值为"MZ"。MS－DOS 头中的 e_lfanew 域(相对文件开始,偏移为 60 字节处)保存了实际 PE 头(IMAGE_NT_HEADERS)的 RVA。

接下来是 MS－DOS 桩(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"。桩的大小是不确定的。

### 2.2　PE 头

接下来的是 PE 头,包括三部分,首先是一个 32 位的签名(Signature),表明可执行文件的类型,PE 格式的值为 50h、45h、00h、00h(PE\0\0);后面依次是文件头(IMAGE_FILE_HEADER)和可选头(IMAGE_OPTIONAL_HEADER)。这里只介绍那些真正有用的域。

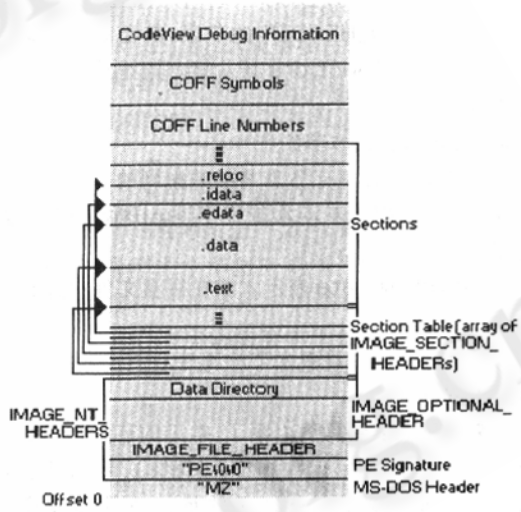


图 1　PE 文件的结构[1]

文件头中的 NumberOfSections 域表明这个 PE 文件中有多少个节(Section)。如果我们增加或减少了节数时候,就必须修改这个值。

可选头中的 AddressOfEntryPoint 域是用 RVA 表示的装入器(Loader)装入后程序开始执行的地址,如 WinMain()、LibMain()的位置。

可选头中的 ImageBase 域是链接器建立可执行文件时,假设此 PE 文件将被映射到的内存基址。若实际装入时不为此地址,则要重定位。

可选头中的 SectionAlignment 域是内存中节对齐的粒度。即要求每节开始的虚拟地址是此域值的倍数。为支持分页,缺省值为 0x1000,即 4K。

可选头中的 FileAlignment 是文件中节对齐的粒度。

缺省值为 0x200,即 512 字节。

可选头中的 SizeOfImage 是内存中整个 PE 映像体的大小,它是经 SectionAlignment 对齐处理后的所有头和节的大小之和。

可选头中的 SizeOfHeaders 是 DOS 头、PE 头、节表的大小。可以以此值作为 PE 文件第一个节的文件偏移量。

可选头中的数据目录 DataDirectory 是一个由 IMAGE_DATA_DIRECTORY 结构组成的数组。每个结构给出本文件中一个重要部分的 RVA 和大小。如第一个数组元素为导出表的 RVA 和大小,如第二个数组元素为导入表的 RVA 和大小。

### 2.3 节表(Section Table)

在 PE 头(IMAGE_NT_HEADERS)之后,就是元素类型为 IMAGE_SECTION_HEADER 的节表,其元素个数是由文件头中的 NumberOfSections 域决定。这里只介绍几个有一点用的域。

Name 域用来标识本节的名字。

VirtualAddress 表示本节将被映射到的 RVA。

SizeOfRawData 表示本节经 FileAlignment 对齐后在磁盘文件中的大小。

PointerToRawData 表示本节在磁盘文件中保存的偏移量。

Characteristics 设置本节的属性。

### 2.4 预定义的节

接下来就是各节的具体内容了,常见的节包括代码节(.text 或 CODE)、已初始化数据节(.data)、导入节(.idata)、导出节(.edata)。对软件保护而言,需要着重研究的是导入节(导入表)和导出节。

在 PE 文件中,当调用另一模块中的函数时,编译器发出的 CALL 指令不直接将控制传递到 DLL 中的函数。实际上,CALL 指令将控制转到同在.text 节的指令 JMP DWORD PTR[XXXXXXXX],此 JMP 指令通过一个在导入节.idata 的 DWORD 变量间接寻址。这个导入节的 DWORD 变量才包含操作系统函数入口点的真实地址。图 2 给出了在应用程序的.text 节中调用其他模块中包含的 API 的具体过程。

利用一个地址来汇集对某个 DLL 函数的所有调用,这样装入器不必对调用 DLL 函数的每条指令进行重定位。PE 装入器要做的只是从被调用 DLL 的导出节中将目标函数的正确地址存入被装入文件映像的导入节的 DWORD 类型的变量中。

导入节是以 IMAGE_IMPORT_DESCRIPTOR 结构为元素的数组,最后一个元素的各个域全为 NULL。每个数组元素对应 PE 文件非显示链接的一个 DLL。如图 3 所示。

Characteristics 域(有时叫 OrginalFirstThunk)包含一个叫 HintName 数组的 RVA。HintName 数组是原始

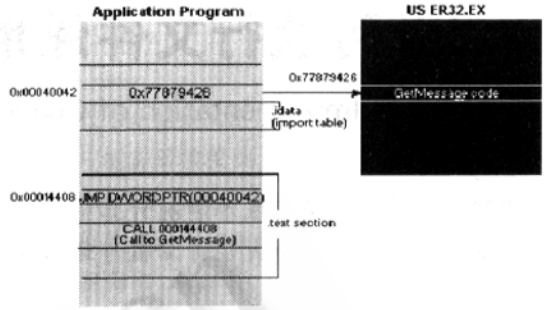


图 2　调用 API 函数的过程[1]

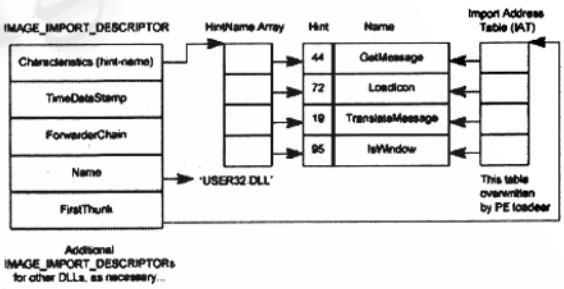


图 3　导入节元素的结构(双平行数组)[2]

的未被装入器绑定的导入地址表(IAT)。

Name 域表示这个结构所对应 DLL 的文件名。

FirstThunk 域也指向一个 IAT。在 PE 文件装入内存前,这一个数组的内容和 HintName 数组的内存完全一样的。但是在装入后,Windows 的装入器会把 API 函数的虚拟地址(VA)放在其中。

## 3　PE 文件通用加口令工具的设计

程序主体分为两个部分:第一个部分是"装配"程序,主要负责把加密外壳程序装配到目标程序之中,并且把目标程序的一些重要的块进行加密。设置原程序的一些装入信息。第二部分是加密外壳程序,除反跟踪程序和合法性验证代码之外,还有一些原程序的数据和文件头信息。

### 3.1 外壳程序设计

外壳式加密就是加密软件把一段加密代码附加到可执行程序上,并把程序入口指向附加的代码中。被加密的程序装入内存之后,附加代码首先执行,检查是否有跟踪程序存在,如果没有再检查密匙是否正确,正确才转入原来的程序中执行。主要步骤如下:

(1) 反跟踪,检查内存中是否存在动态调试软件。

(2) 合法性验证,检查密匙是否正确。可以采取如加密狗,加密卡等高可靠、高速的介质。

(3) 通过合法性检查后就认为使用是合法的了。现在要作的工作就是对原程序的一些节进行解密还原。装配时

我们已经保存了被加密的节数、节的长度、起始 RVA 地址。由被加密节的 RVA 地址就可以找到该节的密文在内存中的首址。再使用装配加密时的密匙并根据保存的该节的长度进行循环解密。

由于在装配的时候已经把被加密块的块属性设置了可写属性(80000000h),所以在解密还原时的写操作不会产生普通保护模式错误。

(4) 由于原程序的导入表被加密,所以解密还原后必须进行人工装入[3]。人工装入基本原理是:根据导入表的指示找到外部模块的文件名,再使用函数 GetModuleHandleA 获得该模块在内存中的句柄;如果没在内存中就使用 LoadLibraryA 函数装入该模块。随后使用获得的模块句柄调用函数 GetProcAddress 获得该模块中导入表中指定函数的实际地址,加上装入基址,并且填入导入表的 FirstThunk 所指向的指针数组中,完成该模块的一个函数的人工装入。重复上述过程对其他函数、其他模块进行装入。

(5) 转入原程序执行。在完成了对原程序一些关键块的解密还原和 Import 表的人工填写后,原程序已经被完全还原。将保存的原程序的入口 RVA 加上装入基址,就是原程序的实际入口地址,使用一条 JMP 指令跳入原程序执行之。

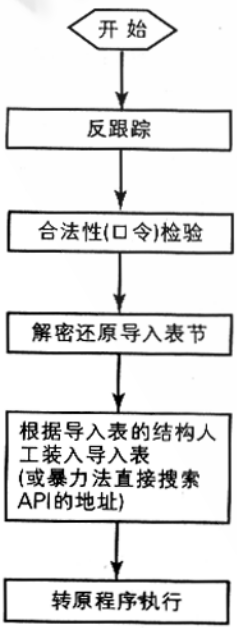


图 4　外壳程序流程

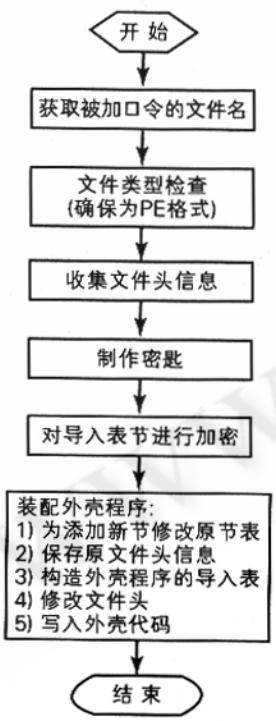


图 5　装配程序流程

### 3.2 装配程序设计

光有加密外壳程序是不能工作的,还需要有一段程序把它和被加密的程序有机的结合起来。这部分的工作就需要装配程序来进行。除了上述功能之外,程序还要收集被加密程序的文件头信息、构造外壳程序的 Import 表、产生密匙等功能。该段程序是整个程序的核心,其中主要是对 PE 可执行文件的操作。主要步骤如下:

(1) 打开文件进行类型检查,确认是 PE 格式。收集文件头信息。

(2) 密匙制作。读加密狗,获得密匙,置于验证代码之中。

(3) 对导入表等重要的节进行加密。对节加密的目的是为了防止对原程序的非法修改和静态分析。保存被加密程序的导入表的 RVA。

(4) 添加新节并进行对齐,修改节表。被加密程序中原来的节已经是对齐了的,这里需要按节对齐(SectionAlignment)的是外壳程序节的 RVA 地址和节大小;需要按文件对齐(FileAlignment)的是外壳程序大小和整个目标文件的大小。

对最后一节末的 RVA 进行节对齐的结果就是外壳程序节的 RVA;对外壳程序末进行文件对齐的结果就是文件的尾指针;对外壳程序大小节对齐的结果就是该节的 VirtualSize;对整个程序大小节对齐的结果就是整个程序的 SizeOfImage。以上结果都需要存到新的文件头或外壳程序数据区中。对齐结束后,把新建的外壳程序节对应的节表表项追加到内存中原节表的末尾。

(5) 修改文件头信息。在程序的前面各部分都有关于文件头信息的修订操作。一些原有的文件头信息需要保存在外壳程序之中。例如要保存原来程序入口的 RVA 地址、设置新的入口地址、修改 SizeOfImage 等。

(6) 构造外壳程序的导入表。外壳程序中必然要使用到一些 API 调用。如:人工填写导入表的工作就是使用 GetProcAddress、GetModuleHandleA、LoadLibraryA 函数调用;密匙验证需要用到 CreateFileA、DeviceIoControl 功能。由于我们的外壳程序是被装配上去的,而不是由程序直接编译链结的,相应的导入表不会自己生成。所以外壳程序的导入表就必须人工构造,以满足外壳程序对外来 DLL 函数调用的需求。

我们设计外壳程序只使用了 Kernel32.dll 和 User32.dll,故只需构造与之分别对应的两个 IMAGE_IMPORT_DESCRIPTOR 结构,每个这样的结构还需要三个辅助数组,参考图 3。在知道了导入表的构造方法后,可以在外壳程序中使用任何 API 调用,以丰富外壳程序的功能。

(7) 写入文件头信息和外壳程序代码。通过前面的步骤,在内存的数据区已经修改好了一切信息,现在需要把新的文件头写入目标文件。使用 SetFilePointer 函数移动指针到 PE 文件头处,调用 WriteFile 写入之。在我们的外壳程序中,已经保存了需要的原文件的文件头信息、新构造

的导入表和其他重要信息。我们使用同样的方法把该段数据和代码写入目标文件的末尾。

## 4 PE 文件通用加口令工具实现要点

在对安全性要求不太高的场合,例如限制他人未经允许使用家里或办公室计算机上安装的某个软件,给 PE 可执行文件加个口令即可。

基本思想是在被加口令的 PE 文件节表末尾加入一个新的节表表项,并正确设置它的值。在文件末尾的地方加入新节的内容,新节实现了一个用来验证口令的对话框。然后,修改 PE 头中的 AddressOfEntryPoint 域,使 PE 文件执行的时候就会先去执行加入的新节中的内容。

### 4.1 暴力法搜索 API 函数地址

任何一个? DLL 都可以用? LoadLibraryA? 来装入,然后通过? GetProcAddress? 来取得这个? DLL 中函数的地址。但是如何能够获得这两个? API? 的地址呢? 可以在整个 4GB? 的线性地址空间中暴力搜索? Kernel32.dll? 的基地址,再从 Kernel32.dll 的导出表中取得? LoadLibraryA? 和? GetProcAddress? 的地址。

DLL 有一个特性,当有别的程序调用它的时候,它的文件映象就会动态地映射到调用进程的虚拟地址空间。一般情况下,一个程序在运行的时候都会将 Kernel32.dll 映射到自己的地址空间,成为自己的一部分。这样,就可以在调用者的地址空间中搜索到? Kernel32.dll? 的基地址了。Kernel32.dll? 也是一个? PE? 文件,在内存中搜索的时候,只需要按照判断? PE? 文件的方法来执行就行了。在不同的操作系统下,Kernel32.dll? 的基地址是不同的,例如 Win 98 下它是? BFF70000h,Win 2K 下为 77E80000h,XP 下为 77E60000h。由于它们都在? 70000000h? 以上,为了加速搜索,可以从? 70000000h? 开始;或者是反过来,从栈顶[esp]开始往下递减,减少到? 70000000h? 为止。由于? DLL 一般是以? 1M 为边界,所以可以用? 10000h(64k)作为跨度,这样可以大大加快搜索速度。

4GB? 的地址空间并不是完全可读的。如果遇到不能读的地方就会产生一般性保护错误(GPF),但是可以用? SEH? 来解决。不过在实际的试验中,GPF 还没有真正遇到过。

### 4.2 反跟踪

反跟踪是任何加密系统的重中之重,但由于 Windows 系统的严重限制,无法编写出十分有效的反跟踪代码。

(1) 检查动态调试软件在内存中的存在。Softice 可以说是这类软件中的典型。使用 CreateFileA 打开绝对模块"\.\SICE",如果打开成功(返回非零的模块句柄)就认为 Softice 已经驻留内存。那么外壳程序就非正常的退出。

(2) 通过调试中断 int 3 给 SoftICE 发送命令让其执行,其中 SI 和 DI 寄存器中放的分别是固定值 0x4647("FG")和 0x4A4D("JM")。EAX 中存放的是子功能号,值为 0x0911,表示让 SoftICE 执行命令;此时 EDX 指向一个命令字符串如"HBOOT"等。EAX 还可以为其它子功能号,比如让 SoftICE 修改断点设置等。SoftICE 也是通过这一个方法来检验自身的是不是在内存中。但是在这一个方法中要得到 Ring 0 级的特权。

(3) 调用 Kernel32.dll 导出的 IsDebuggerPresent 函数来检测是否有调试器存在。这个函数只能检查使用 Debug API 来跟踪程序的调试器,无法检测 SoftICE 之类的系统级调试器。

### 4.3 CRC-32 加密

为了在 PE 文件不出现了用户输入口令的明文,防止他人用十六进制编辑器直接看出明文来,可以使用 CRC-32 的加密方法把明文转化成为密文。

## 5 测试和实验结论

本工具采用 32 位汇编语言编写,采用 MASM32 集成开发环境。可以运行在任意 32 位 Windows 平台,如 Windows 98 /NT /2000 /XP。

正确性测试:本 PE 文件通用加口令工具可以在 PE 文件的结尾增加一个新节(Section),从而在原程序开始运行时,先弹出一个对话框,要求输入口令。口令正确,才能运行原程序。使用者若不知道口令,则不能运行原程序。

反破解测试:

(1) 通过反跟踪技术防止了 SoftICE 等实时调试器的破解。

(2) 通过 CRC-32 加密技术防止了 WinHex 等十六进制编辑工具直接打开 PE 文件寻找口令明文。

(3) 本工具本质是一个加壳软件。因为被加密程序原有的导入表已经被加密,被加密程序刚开始执行时只有外壳程序的导入表是正确的形式。这样防止了被常见的脱壳工具如 FileInfo 3.01、ASprStripper 2.03 等跳过外壳而直接运行原程序的破解方式。

总之,经测试,该工具可以实现对任意 PE 文件的加口令,普通用户和一般的程序员是无法破解的。

### 参考文献

1 Matt Pietrek. Peering Inside the PE: A Tour of the Win32 Portable Executable File Format [J]. Microsoft Systems Journal, 1994, 9(3): 15-34.

2 Microsoft Corporation. Microsoft Portable Executable and Common Object File Format Specification (Revision 6.0 February 1999) [M/CD], MSDN.

3 看雪,加密与解密——软件保护技术及完全解决方案[M],电子工业出版社,2001。