第一章 计算机病毒概述

“人为的特制程序”是任何计算机病毒的固有本质属性

- 程序性的客观性决定了计算机病毒的可防治性和可清除性
- 人为性的主观性导致计算机病毒各异多变

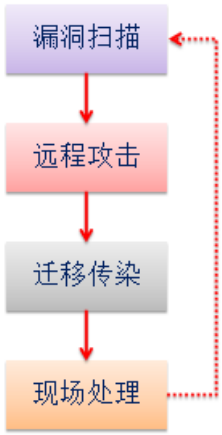
病毒的基本特征

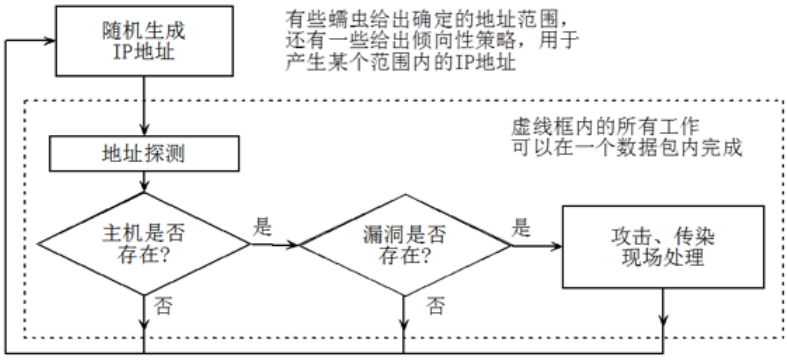
1. 传染性：是指计算机病毒把自身复制到其他程序的能力。是否具有传染性是判别一个程序是否为计算机病毒的首要条件
2. 隐蔽性：利用社会工程学进行伪装、传染过程的隐蔽性、病毒存在的隐蔽性
3. 可触发性：因某个事件或数值的出现，触发病毒实施感染或攻击破坏，即隐藏又保持破坏力
4. 破坏性：数据破坏、计算机功能破坏、经济损失
5. 针对性：特定软件、操作系统、硬件平台（也反映了其程序性的客观事实）
6. 不可预见性：对未知病毒的预测难度（反病毒软件预防措施和技术手段往往滞后于病毒产生速度）
7. 其他特征：欺骗性（欺骗用户触发、激活病毒）、非授权性（窃取系统控制权）、寄生性（依附于宿主程序）、衍生性（病毒变种）、持久性（数据恢复困难、病毒清除困难）

蠕虫

定义：独立的可执行程序，不需要寄生在宿主程序中，通过网络分发自己的副本

工作方式：蠕虫的工作方式一般是“扫描→攻击→复制”





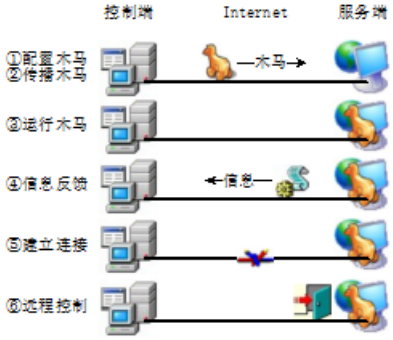
蠕虫和病毒的区别：

	病　　毒	蠕　　虫
存在形式	寄生	独立个体
复制机制	插入到宿主程序(文件)中	自身的拷贝
传染机制	宿主程序运行	系统存在漏洞(Vulnerability)
搜索机制(传染目标)	主要是针对本地文件	主要针对网络上的其它计算机
触发传染	计算机使用者	程序自身
影响重点	文件系统	网络性能、系统性能
计算机使用者角色	病毒传播中的关键环节	无关
防治措施	从宿主程序中摘除	为系统打补丁(Patch)

特洛伊木马

定义：在远程计算机之间建立连接，使得远程计算机能通过网络控制本地计算机的非法程序。

木马系统软件一般由木马配置程序、控制程序和木马程序(服务端)三部分组成。其入侵的方式与一般病毒存在区别，而且，自身一般没有传染性。



第二章 病毒基础-代码初始

调试器的**监视**功能，可观察高级语言表达式（sizeof(gi),&gi）和变量

调试器的**内存**查看方法，以及修改方法

整数的大小端机表示

0x12345678小端机：78 56 34 12，大端机：12 34 56 78

JMP跳转的偏移量=目的地址-JMP的下一条指令地址

理由：CPU执行指令的过程为①将当前指令放入指令队列②EIP指向下一条指令③执行当前指令

篡改执行流程病毒模型

1. 找到normalfunc入口的位置，写入e9 xx xx xx xx.其中，xx xx xx xx代表转跳到virusfunc中打印指令的偏移量
2. 找到virusfunc中的打印指令，在其后修改指令，填充内容为normalfunc中被第一步jmp指令覆盖的3条完整指令
3. 在第2步填充的3条指令后填写jmp 指令使其跳回第1步覆盖的3条指令后

注：在文件中，为了找到normalfunc的位置，我们应该寻找其中的标志性机器码，那就是包含被打印字串的地址的指令68 3c 57 41 00

第三章 病毒基础-文件系统

- 1个FAT9个扇区，引导扇区1个扇区，根目录区有224条记录（一个记录32字节），这些参数都都可以在引导扇区的引导记录中设定
- 引导扇区的结束标志为 AA55H
- 引导记录最开始的 JMP 指令跳转至引导代码

  EB 3C表示偏移为60字节（3CH）：转跳到的地址 = JMP指令写入内存的地址 + 2(JMP长2) + 3C
- BIOS在自检完成后，会根据用户指定的顺序从磁盘或光盘启动

  如果以软盘启动，BIOS会将第一个扇区（0头0道1扇区）加载到内存7C00H处，并跳到该处执行，这段代码就是系统引导代码，它会运行操作系统加载器，加载我们的操作系统
- FAT 表的本质：磁盘簇分配情况的数据表示

FAT表格式

- 在文件系统中，扇区被组成一个更大的单位簇；文件分配的最小单位是簇，哪怕只有一个字节也会分配一簇；簇由几个扇区组成在引导扇区的引导记录中定义
- 在FAT12文件系统中，FAT表以3个半字节（3*0.5 Byte = 1.5 Byte = 12 bit）来记录一个簇的相关情况，这也是"FAT12"文件系统中命名12的原因
- 查找机制：FAT表是一个数组，数组中每个元素是1.5字节的整数，为了查找FAT表中簇号为N的元素，只需用N作为索引查找FAT[N]元素
- 根据FAT12缺省设置，引导区占1扇区，FAT1紧跟其后，所以在偏移512，即200h处开始
- FAT表开始3个字节没用于用户文件分配，3字节有2组12bits所以，占用了0,1两个簇号，用户的数据从簇2开始分配。

根目录表

每条记录占 32 字节，其中包含注文件名（查找需要）（文件名 8B，扩展名 3B，名字的结束以空格表示，即 0x20）和首簇（遍历需要）（首簇号从该记录首部偏移0x1A，占两字节）这两个重要字段

在引导扇区（0头0道1扇区）的引导记录中，有一个字段记录了根目录区的最大记录数目：这个字段在引导扇区偏移17字节处，2字节大小

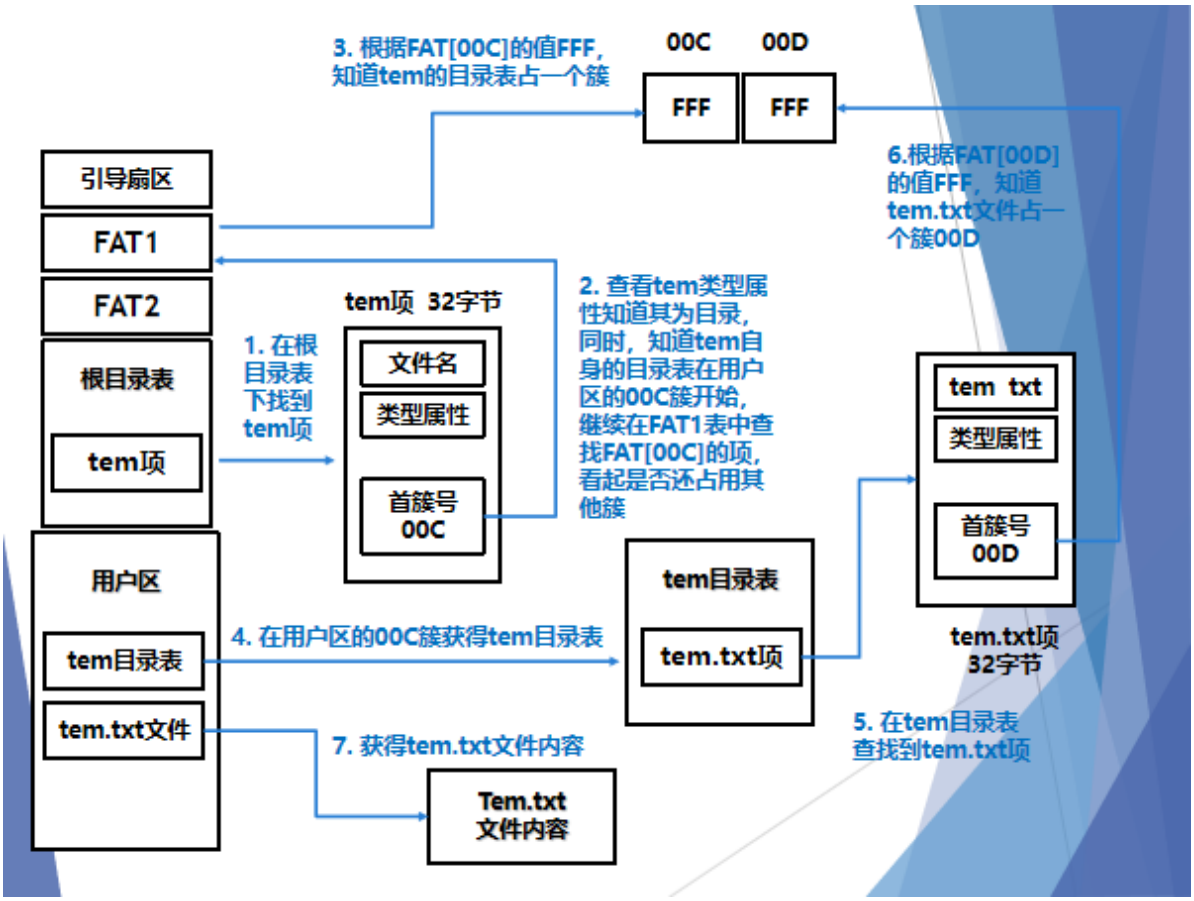
查找根目录表的算法

- 从磁盘0头0道1扇区读出引导区（Boot）512字节，从其中引导记录获取相关信息：Boot区占用扇区数，FAT表数目，每个FAT表的扇区数，1个扇区的字节数。
- 根目录区的起始位置为：[1（Boot区扇区数）+2（FAT数目）*9（FAT扇区数）] * 512 = 0x2600

根目录区的一个特殊项，用来记录软盘卷标

目录项有字段可以标识文件的类型，如卷标、目录、文件等

多层级目录定位文件



文件查找算法：

1. 根据文件路径的第一项（1），先查看根目录表，是否有匹配的项，如果有，通过对应项的首簇段（0x1A）获取该子目录表（1）的首簇号
2. 通过首簇号和FAT表获得子目录表（1）的全部内容，根据文件路径的第二项（2），遍历子目录表（1），一次偏移32字节用名字匹配的方法查找记录项（2），如果找到，则类似STEP1和STEP2继续查找文件路径中的下一项，否则说明找不到，结束
3. 如果在最后一层目录表（路径的倒数第二项，最后一项是文件名）中找到了被查文件的项，从中获取首簇号，即可通过FAT表访问该文件整个相关簇

“.”和“..”：

“.”和“..”是两个目录别名，分别代表当前目录名和上级目录名，它们可以让我们很容易地访问当前目录和回溯上级目录。

“.”和“..”的目录项中最关键的就是首簇字段（项的 0x1A 处），0 代表根目录

删除文件

让目录文件中的记录项无效，所对应的目录项第一个字节被修改成E5代表删除

同时进行簇的回收：从被删除文件的首簇开始，在 FAT 表中获得簇链，将链上的每个 FAT 项复原成 0

恢复文件

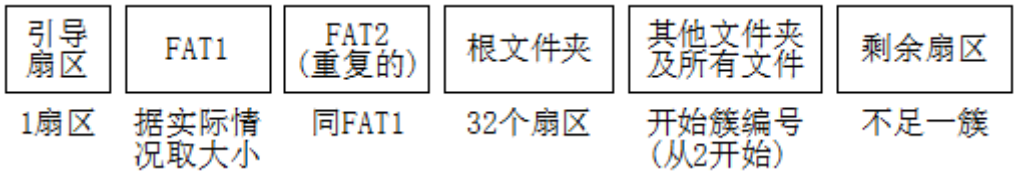
（只考虑连续存放）因为不涉及多个簇，在目录表中找到E5开始的相关文件名，将目录项的第一个字符（E5）修改为其他任意字符，再从该项的首簇字段（0x1A）获得首簇，然后，在 FAT 表中对应的簇项改成 FFF 即可

创建文件

1. 创建文件时，需要首先定位到文件所在的目录文件，然后查找目录项（以32字节为偏移递增），如果第一个字节为00或5E表示可用
2. 根据文件大小计算出需要的簇数目，然后从FAT首部开始（第4个字节，簇2），查看值为00的项，如果是则可用，找到第一个为00的项，将其簇号写入1找到的目录项的首簇字段（0x1A处）
3. 继续在FAT表中继续搜寻为00的项，找到后，将其簇号写入到为前一步找到的项中，这样形成一个簇链，如果是最后一个簇，把FFF写入自己的项中
4. 填写文件目录项的创建时间，属性，大小等字段

FAT16和FAT32文件系统

FAT16



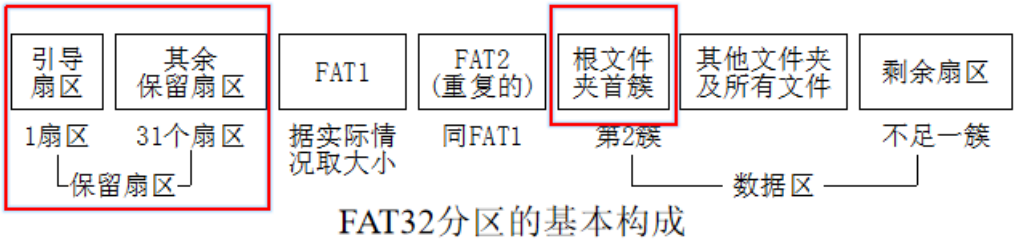
它和FAT12的主要区别在于FAT表项不是12bits（1.5个字节）而是16bits（2字节）

既然FAT16以2字节表示簇号，那么簇号最大为0xFFFF，也就是65535，以32KB为簇大小（单簇空间过大会造成存储空间大量浪费），FAT16最多管理32KB*65535=2048MB=2GB，所以，FAT16不支持超过2GB的分区。

FAT表以“F8 FF FF FF”开头，此4字节为介质描述单元，并不参与FAT表簇链关系。

FAT32

FAT32最早是出于FAT16不支持大分区、单位簇容量大以至于空间急剧浪费等缺点设计的



FAT32分区的基本构成

保留区（引导区）加上FAT区后就是数据区，没有专门的根目录区

在引导区的引导记录中，可以找到：保留扇区数B，FAT表个数F，每FAT的扇区数S等信息

数据区起始扇区计算为：B + F * S=36 + 2 * 8158=16352扇区

长名项：

- FAT32为每个登录的名字（文件名或目录名）都会分配一个短名记录项，其中包括了首簇号，文件大小等基本属性

- 当文件名不足11个字节时，除了短名记录项，同时，还会分配2个长名记录项（第一个字节用E5表示，但它不是表示删除，而是没有生效的长名项）

  当文件名大于11个字节时，会按文件名的字节数分配长名记录项

- 长名记录项放在短名记录项之前
- 对于长文件名，在其所对应的记录项中，倒数第一个依然为短名记录项，但是，它会将原来得文件名截断，使得该短名记录项中文件名的最后两字节为”~1”，比如123456789.txt会截断为123456~1.txt

  如果我们用dir 123456~1.txt的命令是可以成功查找到该文件

  并在长名记录项中用Unicode存放从第一个字符开始的文件名。一个长名记录项（总共32字节）可以存13个字符，占13*2=26字节，不够继续从后往前分配长名记录项

- 和FAT12不同，FAT32有两个字段表示首簇号

  高两字节存储在记录项的0x14~15处，低2字节存储在0x1A ~1B处，FAT12和FAT16不使用高两字节，设定为0

四字节为一个FAT项，FAT32最高半字节是0，而不是F，即：0x0FFFFFFF

FAT32和FAT12的不同：

1. FAT32系统的FAT表项一个长4字节（32位）（0x0FFFFFFF表示无后续扇区）
2. FAT32的根目录区在用户区，而FAT12有专门的根目录区。FAT32引导记录中有一个指向根目录区的首簇字段，一般为簇2，FAT12则没有该字段
3. FAT32支持长名文件，由扩展的32字节记录项构成
4. FAT32的记录项中的首簇号，由两个字段构成，分别代表高位两字节和低位两字节
5. FAT32的引导区有保留区

第四章 病毒基础-硬盘数据结构

硬盘也有本身的结构，硬盘结构（分区等）是建立在文件系统之前，而文件系统是建立在硬盘结构之上。

硬盘分区的作用：

建立硬盘数据结构（分区表，扩展分区等），将硬盘划分成多个逻辑上的分区（每个分区模拟一个独立的硬盘），再将文件系统（比如FAT32）整个放入某个分区，这样就可以有多个逻辑盘了，而且不同分区还可以放入不同的文件系统。

硬盘分区的执行：

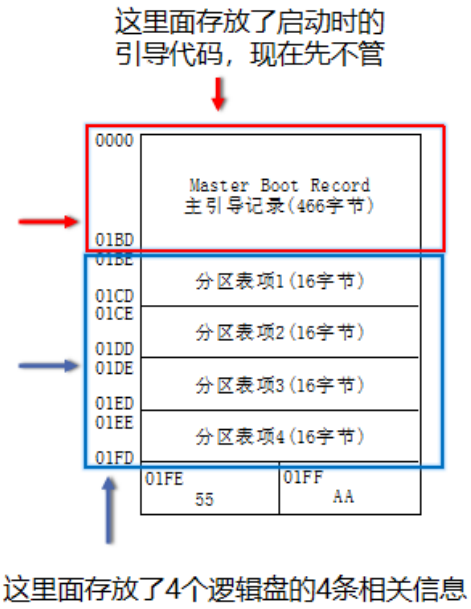
分区划分以柱面为单位，一个柱面不能属于两个分区。

主引导扇区（第一个扇区）

主引导扇区包含整个硬盘分区信息，位于 0 头 0 道 1 扇区，512 字节

其中446字节为主引导记录

后面存放了一个分区表，分区表共4条记录，能最多分4个区（逻辑盘）

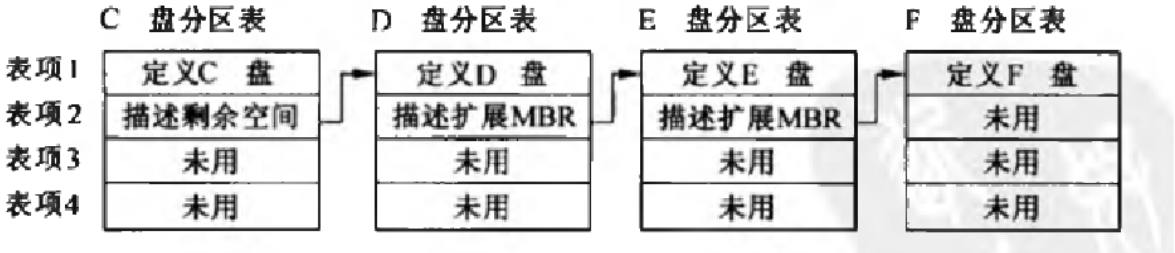


通过相对扇区数找到分区入口

分区1是系统的引导分区，那么它第一个扇区（开始处）应该是包含EB机器码的JMP指令，并且引导扇区最后两字节为55 AA

虚拟扩展分区（Extended MBR，即EBR）技术

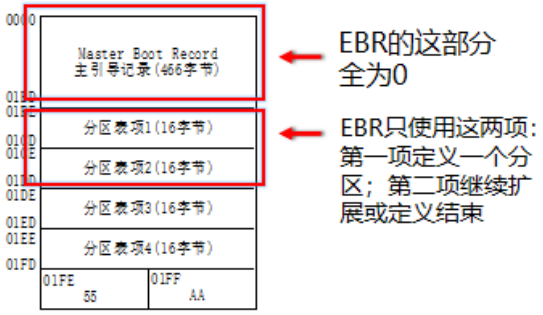
其核心想法是形成一个分区链，如图所示的分区中，MBR定义的主分区表本来有4条分区记录，用第一条描述自己分区的信息，用第2条指向下一个分区。下一分区也如此处理，形成链。



为了完全兼容MBR的格式，EBR完全复用了引导扇区的格式，即起始446字节给引导记录EBR，但在EBR中，这些内容全为0

同样的，从446偏移开始为分区表，16字节为一项，共4项，后两条无效（但占空间），EBR定义的扇区最后两字节也是结束标识55 AA

主分区表中最后一个有效主分区记录指向的磁盘空间（也叫主扩展分区）将用EBR进行划分，分成N个逻辑盘，形成EBR链。



定位主分区和扩展分区的关键区别：

- 主分区表项的偏移量，就是相对物理盘开始（MBR）的偏移，且这些主分区前面没有EBR，根据偏移得到的直接就是盘区（开始就是该分区的引导扇区和文件系统），除了主分区表项中的最后一项偏移指向的是扩展分区的开始处
- 扩展分区表项的偏移量，是相对于包含该分区表的扩展区开始处而言（扩展分区开始处为其EBR）
- 在扩展分区的分区表中，第一项的偏移指向的是盘区，第二项的偏移指向的是下一个扩展分区开始处（也就是下一个扩展分区的EBR）

硬盘引导过程

1. 开机加电自检：开机，CPU跳到内存FFFF:0000处，由该处的一条JMP指令跳到BIOS的自检程序（POST），自检通过后，加载引导程序，与操作系统无关的MBR
2. 读主引导扇区：将主引导扇区MBR读入到内存的0000:7C00，执行权交给MBR，MBR会扫描主分区表，搜索激活分区，分区表项第一个字节为0x80表示激活分区
3. 读激活分区引导扇区：如果有多个激活分区或没有，报错结束。否则读取激活分区引导扇区到0000:7c00
4. 操作系统引导代码引导系统并读取操作系统初始化文件

操作系统相关的引导代码在激活分区的引导扇区，而 MBR 的引导程序选择激活分区

MBR为什么要拷贝自己：

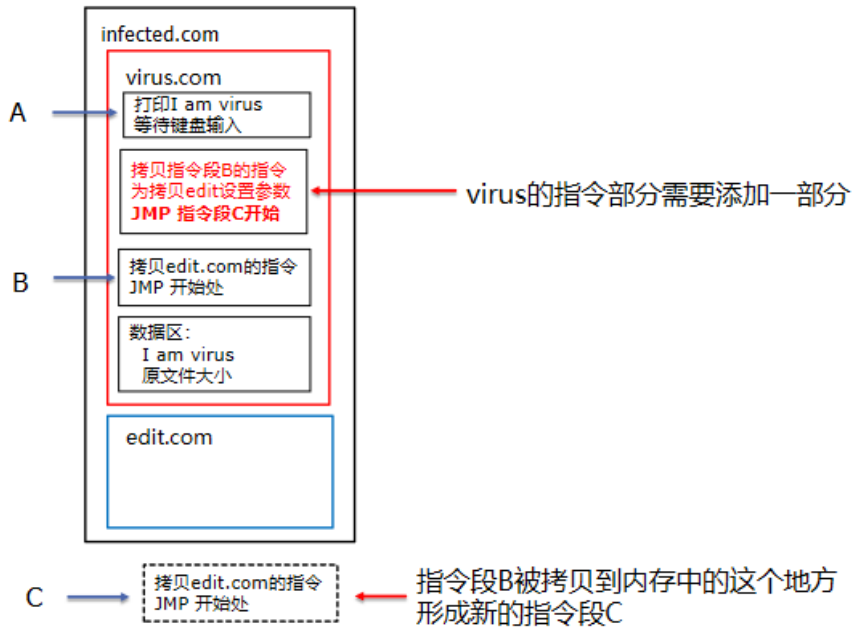
MBR的这段代码会被加载到7C00处，这段代码的主要任务是找到激活分区（如C盘）并将真正C盘的引导扇区加载到7C00处（也就是拷贝），这必然覆盖已经在内存7C00处的MBR自己，当自己正在拷贝的另外的指令覆盖自己时，必然破坏了自己的指令执行逻辑。所以，要错开MBR引导程序和激活分区引导程序占用的内存空间范围，让MBR的引导程序在600h执行，而它拷贝的激活分区引导程序将在7C00h执行

第五章 Dos下的病毒-头插入+尾插入+逆插入

头插入

步骤：

1. 扩充病毒文件的大小
2. 将正常文件往后拷贝，拷贝时要从后往前拷贝，以防止产生自我覆盖
3. 将病毒文件拷贝到正常文件前面的区域
4. 病毒尾部的 ret 会将执行权交还 DOS 系统，使得 normal 程序无法运行：去掉头部的 ret【第一大难题】
5. 去掉 ret 指令后，系统无法区分代码和数据：用 jmp 跳过【第二大难题】
6. normal 的加载偏差：把 normal 再拷贝回到 100H，拷贝时要从前往后拷贝，以防止产生自我覆盖【第三大难题】
7. 如果原文件过大，可能覆盖病毒的 copy 代码：把病毒的 copy 代码移动至安全区域执行【第四大难题】

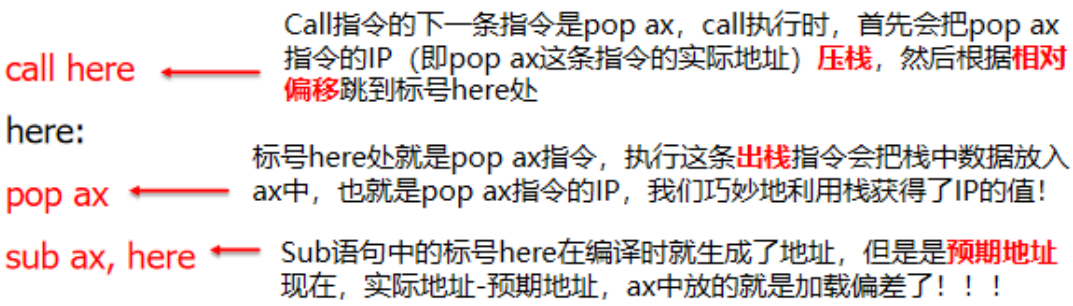


尾插入

步骤：

1. 构造一个打印的正常代码normal.com
2. 构造一个寄生在normal.com尾部的病毒代码virus.com
3. 用DOS的拷贝命令将两个编译好的程序粘起来
4. 自定位后执行病毒代码
5. 将normal.com开始的3字节作为数据保存到virus.com的代码某部分，然后将normal.com开始的3字节修改为JMP XXXX（跳到virus.com指令处）。JMP的偏移量计算方法为：normal.com的size – 3（其中3是因为偏移量从JMP下一条指令开始算，JMP是3字节，所以减3）
6. virus.com执行后，将normal.com开始的3字节进行还原，并JMP到normal.com开始，将执行权限交给normal.com

自定位技术/重定位技术：



virus中的所有的实际地址就=加载偏差+预期地址（编译后）

需要重定位的原因：预计加载地址和实际加载地址不同

重定位的关键技术：获得加载偏差

逆插入

步骤：

1. 获取原文件大小，后续写入需要
2. 将原文件扩容，增加hdrvirusSize（头病毒部分长度）个字节，通过在原文件的尾部写字节完成
3. 原文件向后拷贝hdrvirusSize长度，腾出空间给头病毒部分。（注意：原文件应从尾至头完成拷贝，如果采取从头至尾的拷贝，则当原文件大小>移动长度时，就会产生覆盖）
4. 将HdrVirus部分写入头部
5. 将EndVirus部分写入尾部（不用事先扩容，写直接完成扩容）

头病毒需要自定位：头病毒在原病毒中的位置和在被感染文件中的位置是不同，存在加载偏差，需要自定位

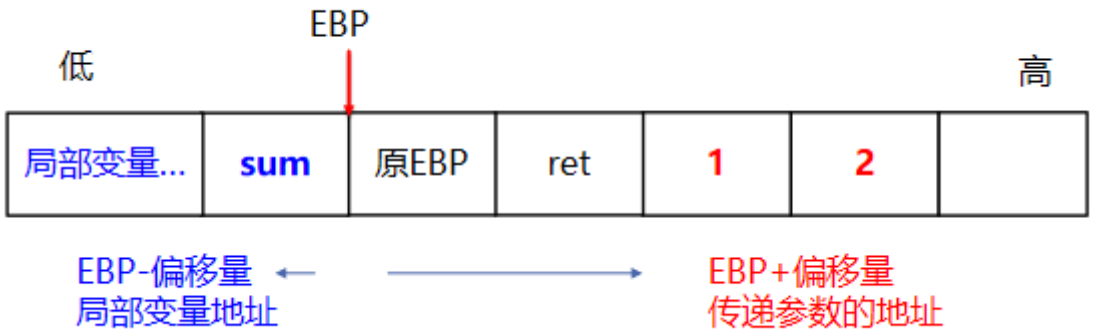
尾病毒也存在加载偏差，原则上也是需要进行自定位的，但由于参数已经在头病毒设好，因此只需简单的将正常程序拷贝到程序头部并跳转即可

尾病毒加载地址=程序加载地址+头病毒大小+原文件大小

第六章 理解栈-函数调用

程序中发生函数调用时，计算机（32位）做如下操作：

1. 首先把指令寄存器EIP（它指向当前CPU将要运行的下一条指令的地址）中的内容压入栈，作为程序的返回地址（一般用RET表示）；
2. 之后放入栈的是基址寄存器EBP（保持之前的值，调用后恢复）；
3. 然后把EBP设为栈顶指针ESP，作为新的基地址；
4. 最后为动态存储分配留出一定空间，即把ESP减去一个适当的数值



在使用了EBP寻址的函数中，EBP+偏移量就是参数的地址，EBP-偏移量就是局部变量的地址。

函数如何返回：通过调用 ret 指令，将当前栈顶存放的内容作为返回的地址。被调函数的 ret 语句将返回地址从栈中弹出，并放入 EIP 寄存器。

函数返回值如何传递：一般都是通过寄存器eax传递，如果有两个返回值，还会用edx寄存器。如果返回值太多或太大（如结构体），寄存器不够用，需要创建一个临时存储空间然后，当被调函数传递返回值时：①通过[EBP+偏移]获得这个返回值的存放位置②把返回值写入到这个内存③把返回值的地址写入到EAX（mov eax, [ebp+偏移]）。

清栈工作可以由被调函数来完成，也可以由主调函数完成。

第七章 Dos下的病毒-引导+中断+链式

引导型病毒

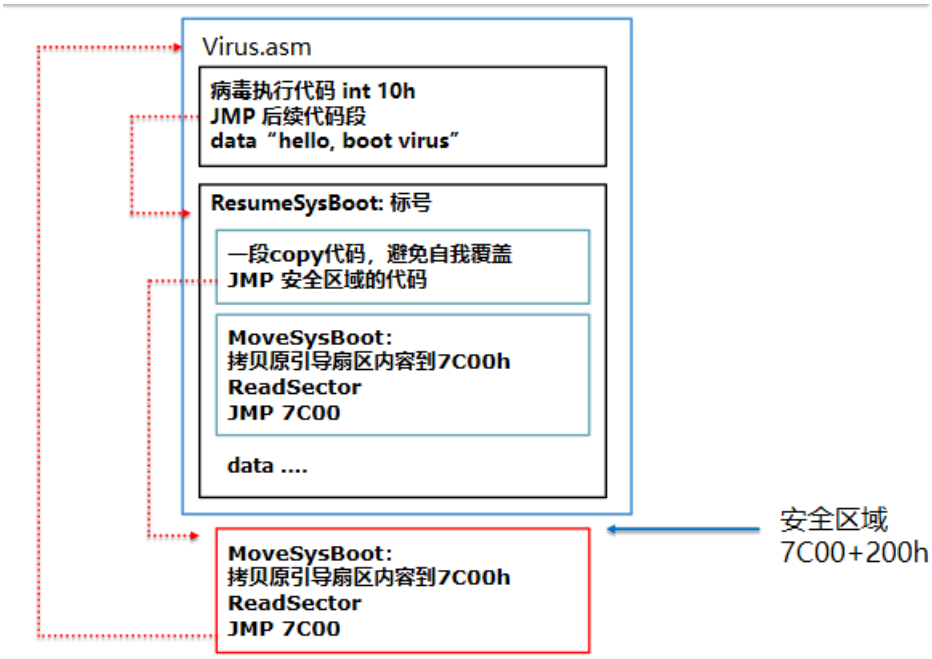
执行部分

原理:

感染引导区，替换引导区原始的引导代码，从而获得执行。之后还原被修改的引导区，并将执行权限交给原来的引导代码，从而保持正常的工作

步骤:

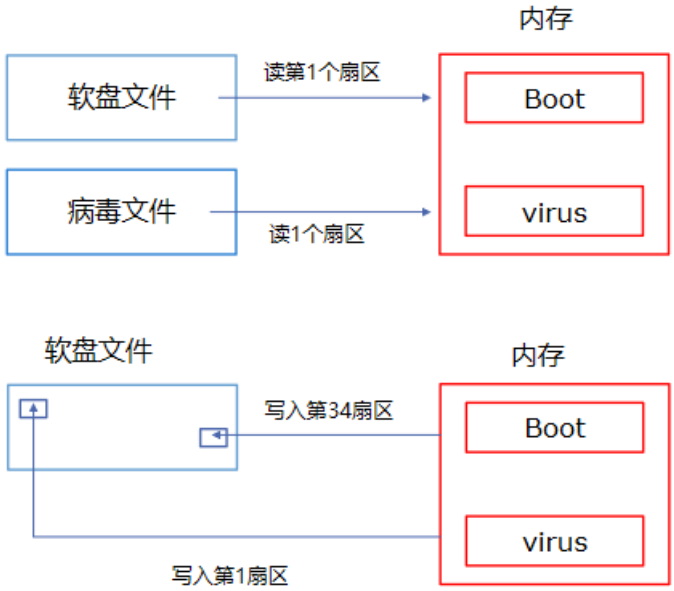
1. 病毒感染时替换0头0道1扇区（512个字节），并将原引导扇区内容保存到用户数据区第一个扇区，也即簇2（假定簇2为空）
2. 病毒先把自己加载到 7C00h，病毒执行完后，从簇2对应扇区读出原引导扇区内容，将其恢复到内存7c00h处，并跳到7c00h将执行权限交给原来的引导程序。这一过程会出现自我覆盖问题，所以需要将病毒中执行拷贝的指令段移出被覆盖的区域，我们可以将它后移一个扇区 7e00h 处。
3. 感染会将原引导扇区的内容一直放到簇 2，如何避免后续使用占用簇 2，可以修改 FAT1 和 FAT2 表，将簇 2 的项改成不可使用，如果改为已占用 FFF，但却没有对应的目录项，是可疑的，因此，可以改为坏簇 FF7



磁盘访问 int 13h 中断：int 13h 采用的是 CHS 的硬盘寻址方式，会将线性寻址方式转化为硬盘寻址方式：（一道 18 个扇区，分为上下两个头）

- 绝对扇区号/18 得到余数和商N
- 扇区号：余数+1，磁道：N>>1，磁头：N & 1

感染部分



引导型病毒设计过程中出现的问题

因为病毒只是简单地覆盖了整个引导扇区，从而破坏了很多重要的引导记录（包括 FAT 有多少扇区，磁盘的介质类型等），导致 DOS 无法识别盘

解决方案：

保留引导记录，利用引导扇区的头的 3 个字节（一条 JMP 指令和一个 NOP），使其跳过引导记录，跳到引导程序处，因此，我们的病毒指令可以只覆盖引导记录后面的部分。

步骤：

1. 在病毒指令之前加入填充字节（字节数为引导记录占用的字节数）
2. 用 fseek 定位到引导记录后才拷贝，而老代码是从一开始
3. 另外，在 fwrite 时，新代码也只从引导记录后开始复制，复制长度减少了引导记录的长度（即仅仅复制病毒执行部分的长度）

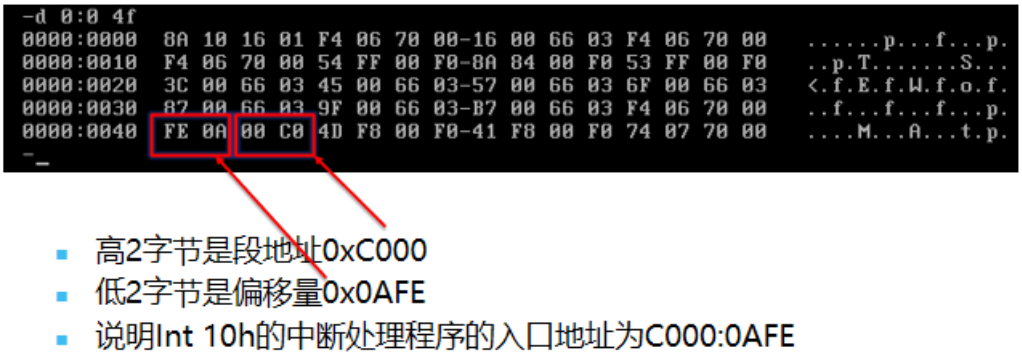
中断型病毒

中断向量表

中断引发后，就会去调用一段处理程序，叫中断处理程序（例程），系统找到中断处理程序的入口地址，从而执行中断。通过一个叫中断向量表的东西，中断处理程序的入口地址就存放在这个表中。

中断向量表就在内存00:00处，每4个字节为1个项（这个项的索引就是中断向量号），存放一个中断处理程序的入口地址，其中，高2字节是段地址[4 * x + 2]，低2字节是段内偏移[4 * x]。

一个中断触发指令int xxh，其中的xx就是中断向量号，4 * xx就是中断xx的入口地址在中断向量表中存放的位置。比如int 10h其入口地址就存放在中断向量表中的4*10h = 40h的位置。



非驻留式中断替换

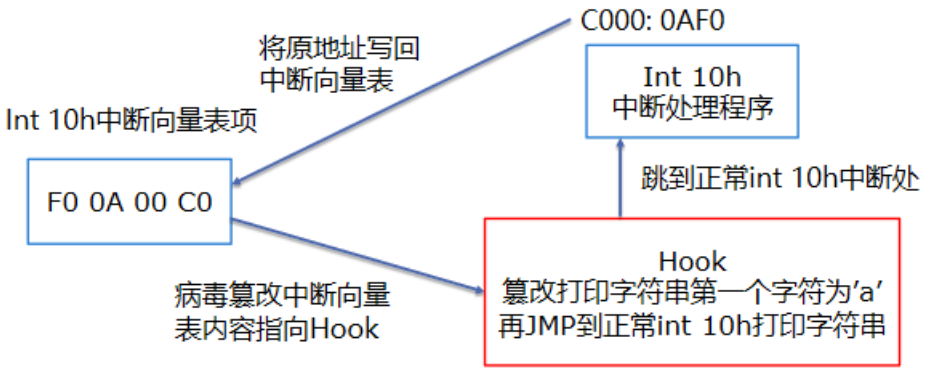
驻留程序：一直在内存中不退出（例如系统的中断处理程序要为所有程序服务，所以它的特点就是不退出一直驻留在内存中）

我们的病毒为了获得执行，会修改中断处理程序的入口地址（即修改中断向量表）指向病毒提供的一段程序，这样，只要调用相应的中断，就会去执行这段程序。但是，这段病毒程序必须也是驻留程序，否则病毒结束后，内存回收，这段程序也不在了，导致中断向量表指向无效。因此，在非驻留式的中断替换中，我们还需要在病毒程序调用后恢复中断向量表

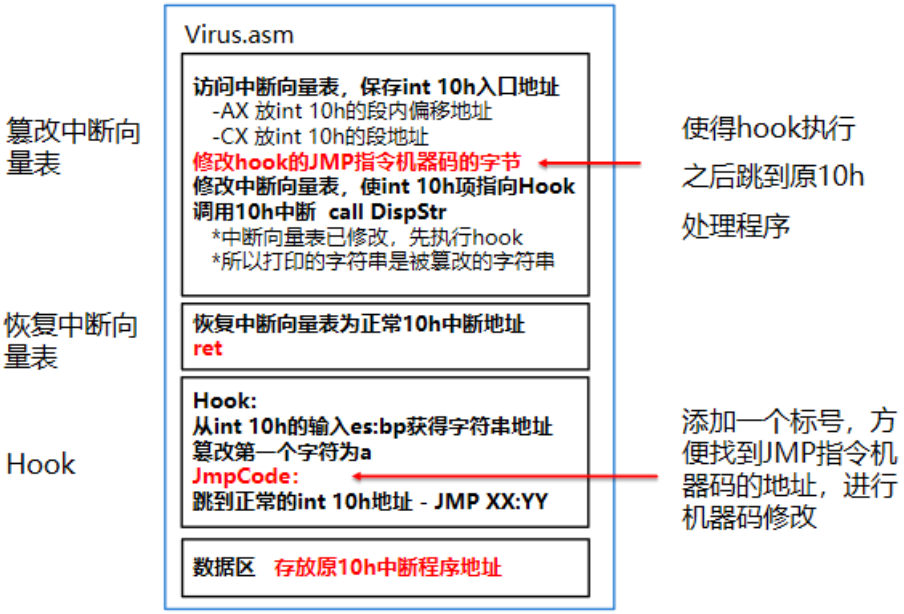
步骤：

（以修改 int 10h 打印字符串首个字符为例）

1. 篡改中断向量表，指向钩子程序 hook
2. 篡改打印字符串第一个字符为'a'，再 JMP 到正常 int 10h 打印字符串
3. 执行原中断处理程序
4. 将原地址写回中断向量表



跨段跳转的问题：病毒需要由自己段跳到原中断向量程序所在的段，要跨段跳转，就要采用 JMP XX:YY 的形式，但是 JMP CX:AX 的语法是不支持的，我们可以先采用两个立即数来确定JMP指令的形式（比如 JMP 00:00），然后我们再来定位到JMP指令的机器码，进行按字节的细粒度修改。



驻留式中断替换

调用了DOS提供的驻留退出中断Int 27h

链式病毒

链式病毒只保留一份病毒拷贝，利用文件目录项，将受感染文件的头簇指向病毒。

感染过程：

1. 如首次感染，将病毒保存在某个空闲扇区
2. 将被感染文件（COM文件）首簇存目录项保留段
3. 修改首簇号指向病毒的首簇

执行过程：

1. 执行被感染文件则启动病毒，加载的是病毒的首簇，并执行
2. 病毒获取当前执行程序的名字，获取对应目录项。从其中保留字段获取原文件首簇号，并遍历FAT簇链加载它们。
3. 跳到加载的原文件内存中执行

感染部分：

1. 被感染文件的真实起始扇区号写到目录表项的保留区（目录项的保留区从目录项头第13个字节即偏移0ch开始，共10字节）
2. 修改被感染文件的目录项的起始扇区字段指向病毒文件的首簇
3. 目录项中的文件大小字段也要修改成病毒的真实大小，这样才能保证病毒能被完整加载
4. 将原来病毒文件的目录项全部32字节改为0，这样从外部看就不存在这个病毒文件，也没有对应的目录项了

执行部分：

1. 病毒运行后，先获取被感染程序的名字
2. 然后从根目录寻找被感染程序的目录项
3. 找到后从该目录项的保留区获取被感染程序的首簇号
4. 找到被感染程序所在簇（即扇区），加载该扇区到内存

因为被感染文件是COM文件，将其加载到内存100h处，并将执行权交还给被感染文件。当把原文件加载到100h时，会产生自我覆盖的问题，和头感染相似，因此，病毒代码需要把执行拷贝功能的代码段移到一个安全区域（即不会被覆盖的区域）。

获取执行文件名

从COM文件的ds:2c处获得环境块首址的段地址

环境块就在段地址：0000处

环境块内容：PATH=……COMSPEC=C:\COMMAND.COM\0….00 xxxx

环境块开始是PATH等0字符结尾的串，最后是两个00字符。然后有两个字节，可能是数目，之后就是执行程序的名字

第八章 Win病毒-虚拟地址

动态链接库DLL

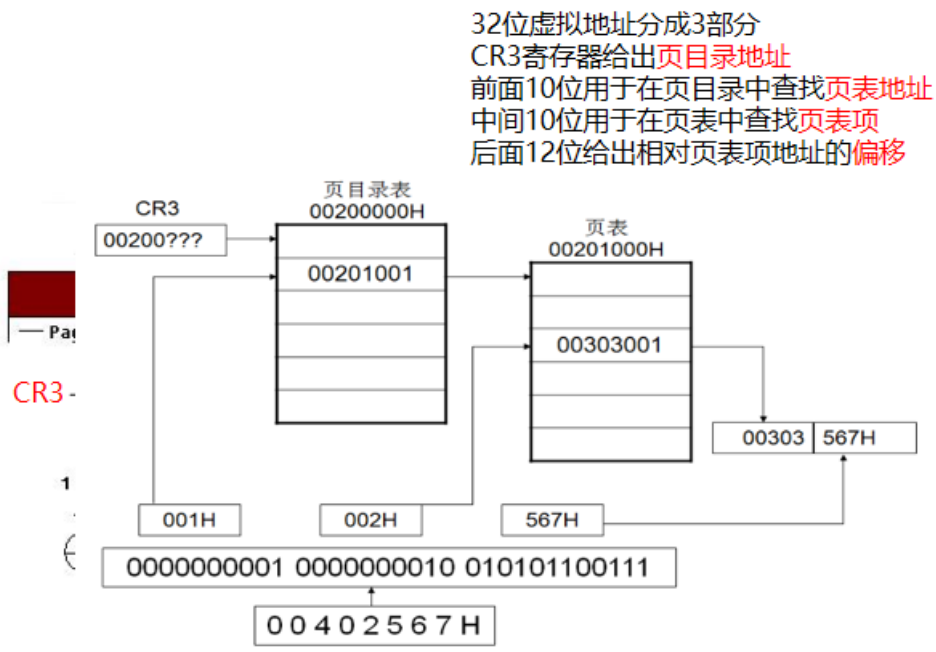
动态链接库（Dynamic Link Libraries）为模块化应用程序提供了一种方式，使得更新和重用程序更加方便。

1. 动态链接库是应用程序的一部分，作为模块被进程加载到自己的空间地址
2. 动态链接库在程序编译时并不会被插入到可执行文件中，在程序运行时整个库的代码才会调入内存，这就是所谓的“动态链接”

保护模式

保护模式最大优点是对内存寻址从机制上提供了保护，将系统的执行空间按权限进行了划分，防止应用程序非法访问其他应用程序的地址空间(任务间保护)，防止应用程序非法访问操作系统地址空间(系统保护)。

虚拟地址

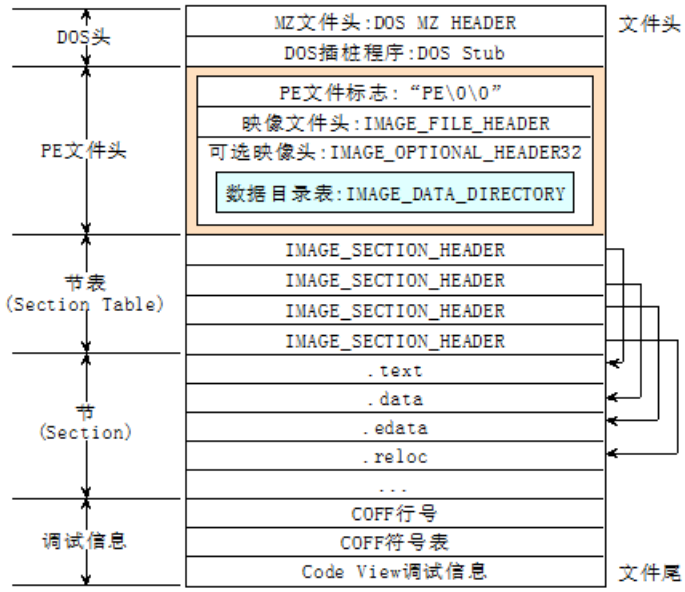


第九章 Win病毒-PE结构+末节寄生

PE格式

- 在Win32位平台可执行文件命名为可移植的可执行文件(Portable Executable File)，该文件的格式就是PE格式。
- PE文件的加载要完成虚拟地址（内存）和PE文件（硬盘）之间的映射关系，所以又被称为映像文件。当真正执行某个内存页中的指令或访问某一个页的数据时，这个页面才会真正读入内存。这种机制使得文件装入的速度和文件的大小没有太大的关系。
- RVA是内存的相对位置，相对的是加载到内存的基地地址；FOA是文件中的相对位置，相对的是文件开始位置（即0）。

  文件在硬盘存储时的对齐方式和在内存加载时的对齐方式是不同的。在文件中，每一个节往往按512B（200H）的粒度对齐；而在内存中，通常以4096B（1000H）的粒度对齐。由于对齐的方式不同，有些节，它在内存的RVA就和它在文件的FOA就不一致了。Windows装载器在装载DOS部分、PE文件头部分和节表部分时往往不进行任何处理，因此这部分的文件偏移和内存偏移是一致的；相应的，第一个节的RVA和它的文件位置往往是一致的。



1. DOS头

   该头部的第一个字段e_magic，就是两个字符MZ，代表DOS文件。 计算机病毒就是通过“MZ”和后面的“PE”标志，来初步判断文件是否为PE文件，从而确定是否进行感染寄生。

   最后一个字段e_lfanew是偏移量，就是从文件开始到PE文件头（NT头）的偏移量。

2. PE文件头/NT头

   ①PE文件标志（Signature）

   Signature字段头两个字节是“PE”，表明该文件是PE格式的文件。因此，病毒判断一个文件是否是PE格式往往可以通过：先判断文件头2字节是否为“MZ”，再判断NT头的Signature是否为“PE”。

   ②映像文件头（IMAGE_FILE_HEADER）

   NumberOfSections：文件中节的个数（病毒查找一个节的时候需要此字段）

   ③可选映像文件头（IMAGE_OPTIONAL_HEADER32）

   AddressOfEntryPoint：代码入口RVA，第一条指令的RVA

   ImageBase：载入程序的首选RVA

   SectionAlignment：节在内存中对齐方式

FileAlignment：节在文件中对齐方式

SizeOfImage：内存中整个PE文件的总大小（按内存对齐）

Import_RVA：导入表RVA

Export_RVA：导出表RVA

Rel_RVA：重定位表RVA

3. 节表

   PointerToRawData：本节原始数据在文件中的位置

   SizeOfRawData：对应节的文件大小（按文件对齐）

   VirtualSize：加载到内存的大小（文件原始大小，未对齐）

   注：SizeOfRawData和VirtualSize可能不同，文件大小可以大于内存也可以小于内存。小于内存时，将在内存补0

4. 节
5. 调试信息

利用入口RVA实现病毒执行

如何从一个PE文件中找到程序入口的文件地址：

1. 在 PE 头的可选映像头中定位 AddressOfEntryPoint 字段，读取其内容，即程序入口点的 RVA
2. 找到此 RVA 所属的节（节的RVA<=入口点的RVA<节的Virtual Size，说明入口点在这个节），如果该节的起始 RVA 和该节的起始文件偏移相等，那么就直接用 1690 作为文件偏移
3. 若有差别，则：入口点的 RVA（AddressOfEntryPoint）- 节的 RVA = 入口点的 FOA - 节的起始文件位置（PointerToRawData）

注：程序入口地址VA=ImageBase+AddressOfEntryPoint

让病毒加载到内存——增加病毒代码到最后一节

如果该节内存大小 < 文件大小，我们就在文件中将指令加到该节的多余部分（对齐后的空洞），然后修改节表SectionHeader中的VirtualSize字段（加载到内存的字节数）为修改后的大小，而对齐后的文件大小SizeOfRawData保持不变。

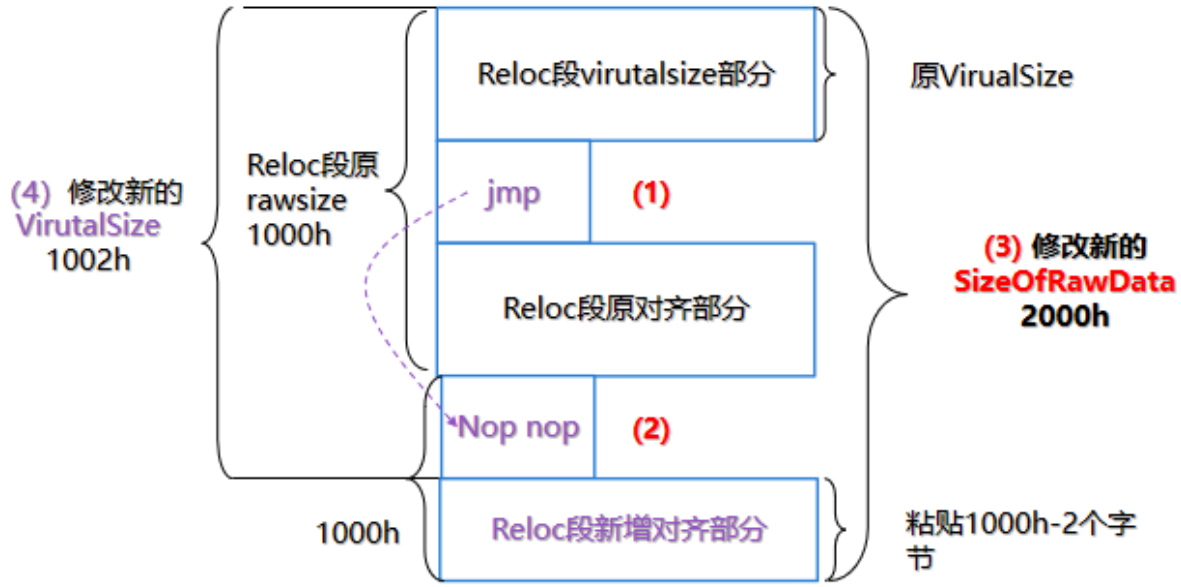
寄生后未改变文件大小的步骤：

1. 首先，我们要找到reloc节在文件的位置，即其节表项的PointerToRawData字段
2. 找到节中的寄生位置，就是reloc节有效长度（实际字节数），即VirtualSize字段的后面
3. 找到在文件中的寄生位置：PointerToRawData + VirtualSize
4. 用UE在文件偏移到该处进行修改
5. 将reloc节相应的SectionHeader中的VirutalSize修改，比原值加4（修改的指令的大小）
6. SizeOfRawData不用修改
7. 加载到内存的实际字节数发生了改变，SizeOfImage 可能发生变化，SizeOfImage =（reloc 节的 PointerToRawData + reloc 节现在的大小 VirtualSize）/ SectionAlignment，向上取整
8. 最 后 修 改 AddressOfEntryPoint ， 即 为 reloc 节 的 RVA + reloc 节原来的大小 VirtualSize

寄生后使文件长度变大的步骤： （模拟：reloc 节后加两条 nop）

1. 在reloc节原VirutalSize后添加JMP xx xx xx xx，在DOS部分已知机器码偏移量为两字节E9 xx xx，而 windows 下器码偏移量为四字节E9 xx xx xx xx。JMP指令偏移量 = 原SizeOfRawData – 原Virutalsize – JMP指令长（5字节）。
2. 在reloc节后添加两个NOP指令
3. 修改reloc节头的SizeOfRawData，加一个FileAlignment
4. 修改reloc节头的VirtualSize为原SizeOfRawData+2(两个nop)，现在NOP才是实际结尾

5. 修改可选映像头的SizeOfImage = (relocRVA + 新VirutalSize) / SectionAlign 向上取整
6. 在NOP后手动增加1000h-2个字节，内容不论，为对齐后填充内容，以前是编译器自动填充
7. 修改入口点RAV（AddressOfEntryPoint）



第十章 Win病毒-末节寄生程序设计

1. 将病毒寄生在末节空洞的程序设计中，下列说法不正确的是（　　）

A. <windows.h>和<winnt.h>提供了PE文件头相关的结构体定义

B. 病毒程序想访问AddressOfEntryPoint时，可以先将NT头信息由文件读到相应的结构体变量中

C. 病毒程序需要生成寄生到原文件的病毒代码

D. 寄生完成后，病毒程序需要修改结构体变量中的成员virtualSize和imageSize来确保寄生的病毒代码被加载到内存

答案：D

第十一章 Win病毒-EPO 入口点不在代码节的问题

入口点模糊技术（EPO）：病毒代码隐藏自己入口点，避免被查杀的一种技术

属性

怎样附加一个属性：或操作（要添加的为1，其他都为0）

怎样判断有没有一个属性：与操作（要判断的为1，其他都为0），判断是否等于原来的值，如果相等就有

怎样去掉一个属性：与操作（要去掉的为0，其他都为1）

如何判断一个节是否为代码节：遍历所有节表项，判断节的节表项的 characteristics 的属性是否为 20

感染在代码节的空洞

不感染最后一节，直接感染代码节，病毒代码附着在代码节的尾部，再修改入口点。这样虽然修改了入口点，但让入口点处于代码节

缺陷：本方法有个缺陷，真正的入口一般都在代码节的前部，而我们感染的是尾部，这样修改后的入口点太靠代码节的后部，这也会使得入口点看起来异常，从而被一些杀毒软件查杀；此外，感染代码节时，代码节往往不是最后节，如果代码节空洞不够，就必须增加代码节的节长，则后续节都要修改 RVA，文件偏移等字段，非常麻烦。

步骤：（假设病毒感染代码小于代码节对齐后的空洞长度）

1. 生成需要寄生的病毒代码
2. 获得被感染文件的 NT 头
3. 找到代码节并判断是否具有空洞
4. 修改 VirtualSize 写入病毒
5. 修改 SizeofImage 和 EntryPoint

感染最后节，替换入口指令（不修改入口点）

不修改入口点，但将入口点所在的指令替换成一条JMP指令，跳往到寄生的病毒代码

感染时，先将原入口5字节保存，替换成JMP跳到寄生代码。病毒执行后，将入口5字节还原，然后跳回到原入口

缺陷：即使没有修改入口点，而是修改入口点的代码本身，杀毒软件一般还是能发现。只要杀毒软件检查了开始的指令，发现有JMP，再算出JMP跳到的位置，目标位置在最后节，非code，而且从一个节(.text)跳到另一个节(.reloc)，基本上都是坏人；就算将代码寄生到代码节.text的空洞，而不是.reloc节，也必然是寄生在代码节的尾部附件了，程序第一条指令就JMP到尾部上，跳转的偏移太大，也很可疑，必然会被抓出来。

步骤：（仅包括主函数涉及的内容）

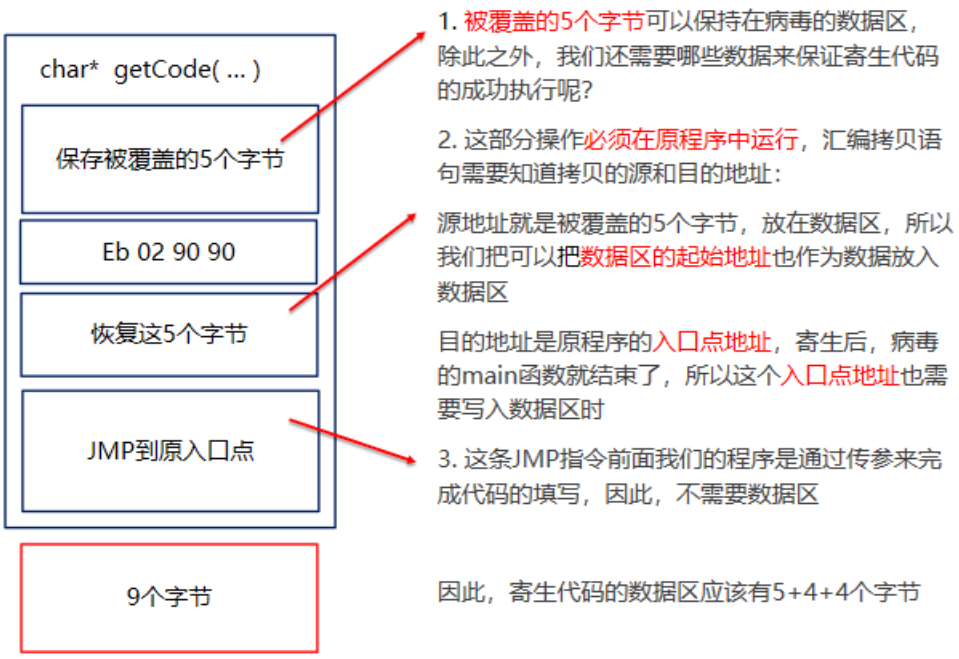
1. 篡改入口点的 5 个字节：找到入口点所在的文件位置；先将原入口5字节保存，替换成JMP跳到寄生代码。
2. 修改相关字段 VirtualSize
3. 写入寄生的病毒代码
4. 修改字段 SizeOfImage

测试时出现错误：需要修改节表项属性 characteristics，80000000h 代表写

getCode函数的设计

寄生代码的数据区需要放置的内容：

被覆盖的 5 个字节、数据区的起始地址、原程序的入口点地址，共 5+4+4 字节



getCode函数的参数设计：

char* getCode(long oldEntryRVA, long virusStartRVA, long imageBase, char * oldEntryBytes)

1. 原入口点RVA:

   long oldEntryRVA ------- AddressOfEntryPoint

2. 病毒的寄生位置的RVA（用于生产最后一条JMP指令）：

   long virusStratRVA ------ 代码节的，起始RVA + virtualSize

3. 原程序的预期加载地址：

   long imageBase ------ ImageBase

4. 被覆盖5个字节需要先由main函数读出来，放到一个字符数组里面，然后，再作为参数传递给getCode函数，因此，还需要一个参数：

   char* oldEntryBytes

本函数的另一个功能：

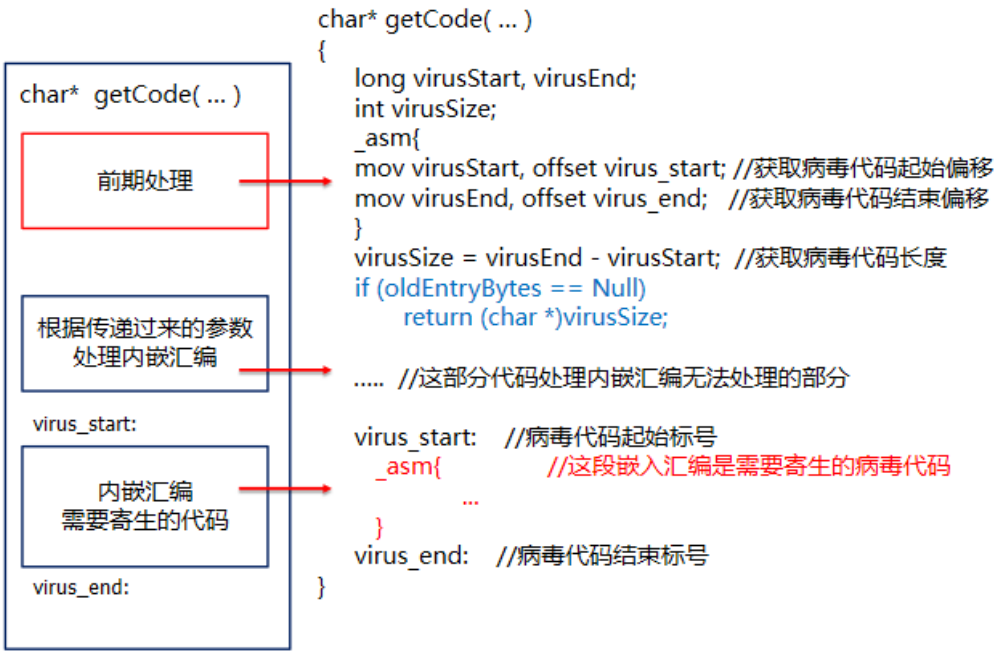
因为需要判断是否病毒能放到段对齐空余部分，所以需要在未构建病毒寄生代码时，就需要知道病毒寄生代码大小。

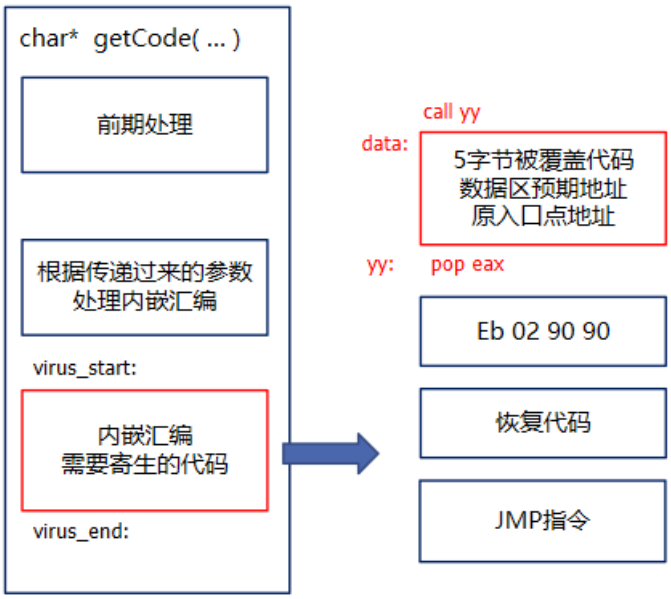
1. 如果最后一个参数为NULL，则返回病毒寄生代码的大小，却并不生成病毒代码（调用者将char * 强转成int即可）

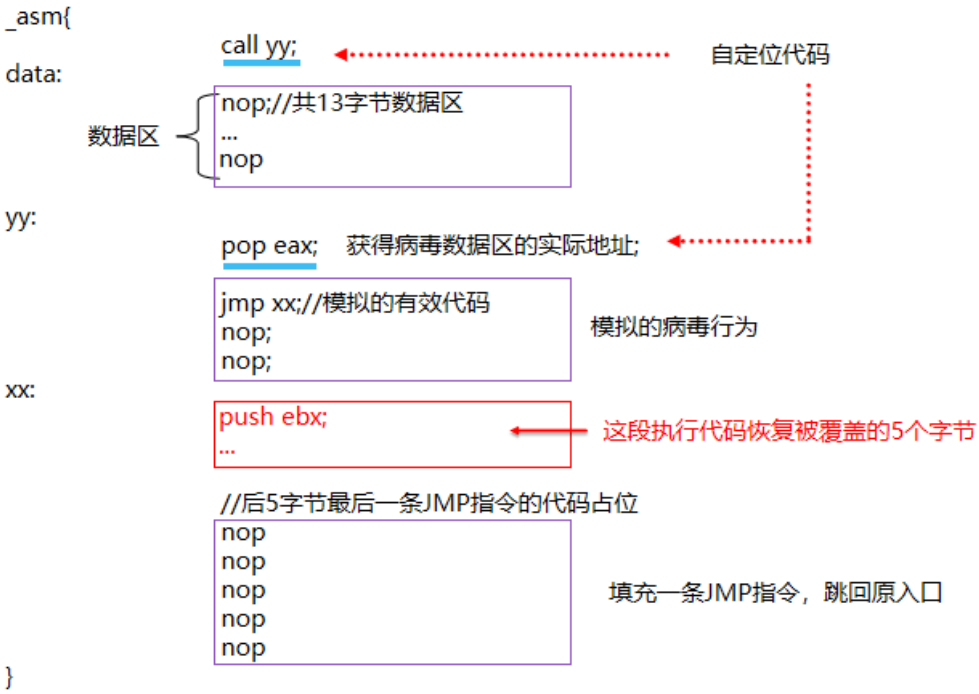
   int codeSize = (int)getCode(0, 0, 0, NULL);

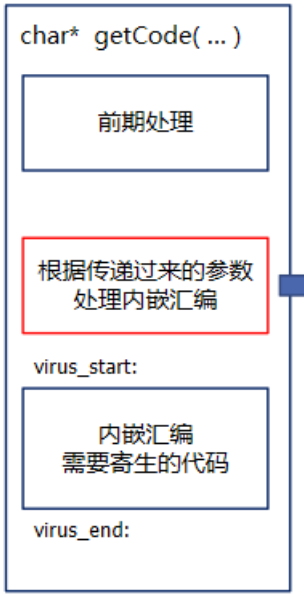
2. 如果最后一个参数不是NULL，就构建病毒寄生代码，并返回为病毒代码的首址

getCode函数代码实现：


char* getCode(...)
{
long virusStart, virusEnd;
int virusSize;
_asm{
mov virusStart, offset virus_start; //获取病毒代码起始偏移
mov virusEnd, offset virus_end; //获取病毒代码结束偏移
}
virusSize = virusEnd - virusStart; //获取病毒代码长度
if (oldEntryBytes == Null)
return (char *)virusSize;
..... //这部分代码处理内嵌汇编无法处理的部分
virus_start: //病毒代码起始标号
_asm{ //这段嵌入汇编是需要寄生的病毒代码
...
}
virus_end: //病毒代码结束标号
}
char* getCode(...)
前期处理
根据传递过来的参数
处理内嵌汇编
virus_start:
内嵌汇编
需要寄生的代码
virus_end:


char* getCode(...)
前期处理
根据传递过来的参数
处理内嵌汇编
virus_start:
内嵌汇编
需要寄生的代码
virus_end:
call yy
data:
5字节被覆盖代码
数据区预期地址
原入口点地址
yy: pop eax
Eb 02 90 90
恢复代码
JMP指令

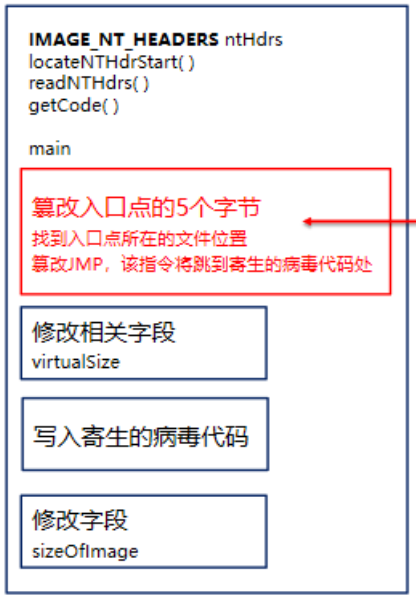

_asm{
call yy;
自定位代码
data:
数据区
nop;//共13字节数据区
...
nop
yy:
pop eax; 获得病毒数据区的实际地址;
jmp xx;//模拟的有效代码
nop;
nop;
模拟的病毒行为
xx:
push ebx;
...
这段执行代码恢复被覆盖的5个字节
//后5字节最后一条JMP指令的代码占位
nop
nop
nop
nop
nop
填充一条JMP指令，跳回原入口
}



这部分代码需要完成的工作：

1. 为**内嵌汇编代码**（即真正寄生到原文件的病毒代码）分配内存空间，生成char数组作为getCode函数的返回值，main函数将把其寄生到原文件空洞
2. 在返回前，需要修改**内嵌汇编**不能自己处理的部分，包括病毒代码里面数据区的填充工作（存放原入口5字节内容，相关地址等数据），最后一条JMP指令的填充工作

感染代码的设计



关键在于：

找到入口点所在的节

然后将入口点（内存位置）

转变为文件位置

第十二章 Win病毒-DLL+导出表

导入表：编译时，函数地址和函数名关联，形成导入表。加载时，系统解析了导入表，然后找到函数名的入口，将其地址填写如导入表项

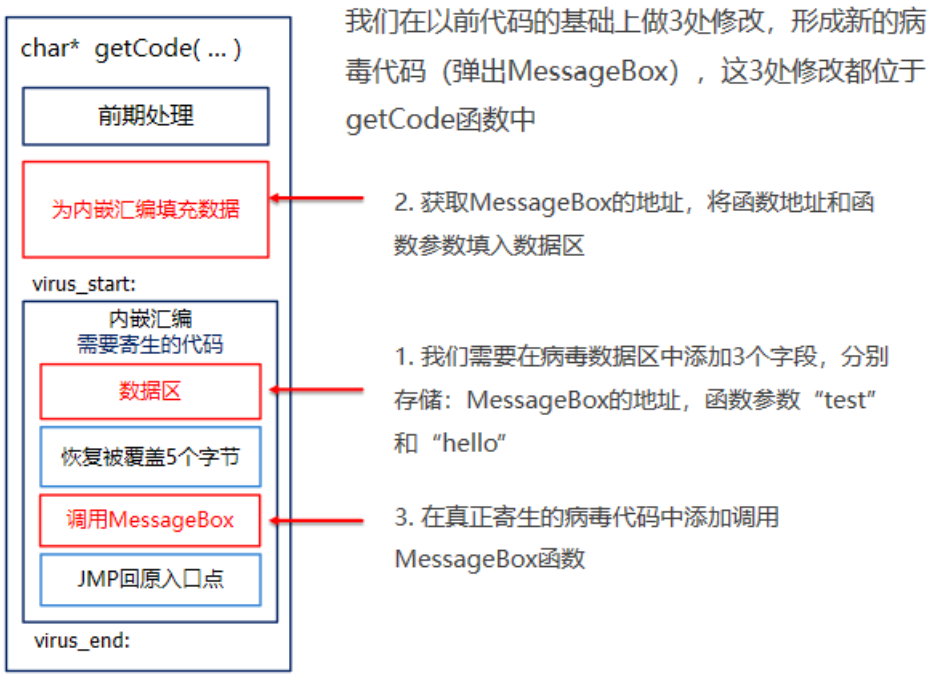
系统如何提供API：操作系统通过动态链接库（DLL）来对外提供API

如何获取API的入口地址

一种剑走偏锋的尝试

获取API的入口地址基于的前提：所有进程加载的同一个DLL的加载基址是相同的

步骤：我们只需要在C写的病毒代码中获取API函数的首址，然后把这个首址写入到我们病毒的数据区，这样病毒代码寄生到了原文件上时，就可以在自己的数据区获得该API函数的首址。



如何判断是否是控制台程序：在PE文件的可选映像头中，有一个字段可能能够用于识别PE文件是否为控制台程序，就是subsystem，当该值为2时就是控制台程序。

获取DLL基址

主要利用PEB结构（Process Environment Block，进程环境块）查找，每个进程都对应一个PEB

步骤：

1. FS寄存器在偏移0x30处保存一个指针，指向PEB结构，FS:[0x30] -> PEB。
2. 在PEB结构的偏移0x0C处保存着另外一个指针，该指针指向一个叫PEB_LDR_DATA的结构。
3. 这个PEB_LDR_DATA 偏移0C处是加载模块链表的头指针，由8个字节组成，前4个字节指向一个LDR_MODULE结构体（LDR_MDOULE代表一个模块，每一个模块（exe,dll）都对应一个这样的结构体）。在该LDR_MODULE中，头4字节又指向下一个加载的LDR_MODULE结构体，由此组成链表。
4. 依序遍历找到对应的LDR_MODULE结构体（例如第3个就是kernel32）。**在遍历过程中识别模块的方法为：**在LDR_MODULE结构体偏移0x2C的地方，有一个成员BaseDllName，它有8个字节，其中后4字节为地址，指向一个unicode串（每个字符占2个字节），这个unicode串就是不包含路径的纯模块名。**遍历结束的条件为：**最后一个LDR_MODULE并没有包含任何真实模块的信息，除了

包含3个链表信息的头24字节，从BaseAddress开始全是0。而其开始的4字节指向了最开始我们从PEB中拿到头块地址。因此只要发现next块的头4字节是头块地址就停止遍历。

5. 在找到的结构体偏移0x18处就是所对应模块的基址。

获取DLL中的函数地址

DLL对外暴露自己的函数有两种方式：函数名和序号

- 可以通过函数名查找某函数入口，也可通过序号查找。
- 但是，需要注意的是，函数名和序号并非一一对应。

1. 序号查找的好处：快且高效

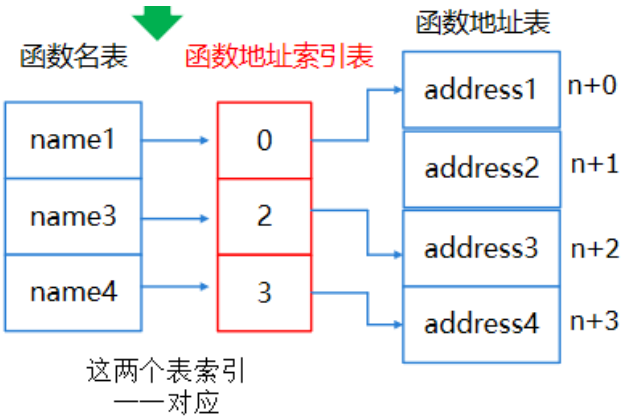
   入口地址funcEntryTb[m - n] = 数组首址funcEntryTb + 4*(m - n)（m>=n，n为数组开始序号，m为要查找的序号）

2. 用函数名查找：直观且具体

   这个最简单的办法就是一个一个函数名字串比较，找到相同的串。

DLL中有些函数是只有序号暴露，而没有函数名暴露的而函数地址表是和序号对应的。这样，就导致函数名表索引是无法和函数地址表索引形成一一对应的关系。

解决方法：增加一个函数地址索引表，用它来记录函数名表索引到函数地址表索引的对应关系



按**函数名**查找函数入口地址：

1. 找到函数名在函数名表的索引 x
2. 读函数地址索引表的第 x 项，假设该项的值为 y
3. 在函数地址表的第 y 项拿到函数地址

按**序号**查找函数入口地址：（直接根据序号计算其在函数地址表的索引）

i = 函数序号-最小的序号（n），然后在函数地址表中找到索引为 i 的项，取出地址即可

导出表表头（IMAGE_EXPORT_DIRECTORY）

- Address Table RVA：函数地址表的RVA（它每项4个字节）
- Ordinal Table RVA：函数地址索引表的RVA（PE格式叫它序号表，它每项2个字节）
- Ordinal Base：最小的序号
- Number of names：函数名表的条数
- Name Pointer Table RVA：函数名表（它每项4个字节，存放了一个RVA，指向一个函数名的字符串）

如何获取导出表的表头：在模块的可选头（NT头的可选头）中，最后有一个数据目录表，其中每一项都是一个重要的元素，包括导出表、导入表、重定位表等。第一项就是导出表。

导出函数查找算法总结

1. 从DLL加载的实际基址获取可选头，从其中数据目录表的第一项找到导出表入口RVA
2. 从导出表的表头获取Number of names，即查找的最大循环次数
3. 循环遍历函数名指针表，比对每项RVA指向的字串是否为要找的函数名

   函数名指针表1项4字节->对应的字符串地址
4. 如果找到，记下此时函数名指针表项的索引，设为 i
5. 根据索引 i，在序号表中找到对应项，获取其内容为n（序号表1项2字节）
6. 以n为索引在函数地址表中找到函数入口的RVA，加上DLL的实际基址即为函数的实际入口地址（地址表1项4字节）

注：以上算法中，所有访问实际地址的地方，就用DLL的实际加载基址+RVA即可

第十三章 Win病毒-指令Patch介绍+导入表

EPO想要不替换入口点附近的指令，而是去替换正常代码执行逻辑中的一条指令

核心问题：

1. 找到一条指令，并知道其起始边界
2. 最好这条指令必然执行，否则patch了，病毒代码也不会必然执行。

解决思路：

对导入函数的调用指令进行Patch，这些指令的可靠度很高，可以选择某些必然被调用的函数指令去Patch

程序如何调用一般函数

函数调用一般采用Call + 相对偏移（E8 xx xx xx xx）

为什么 Call 要用相对偏移：解决加载可能出现与约定地址不吻合的问题

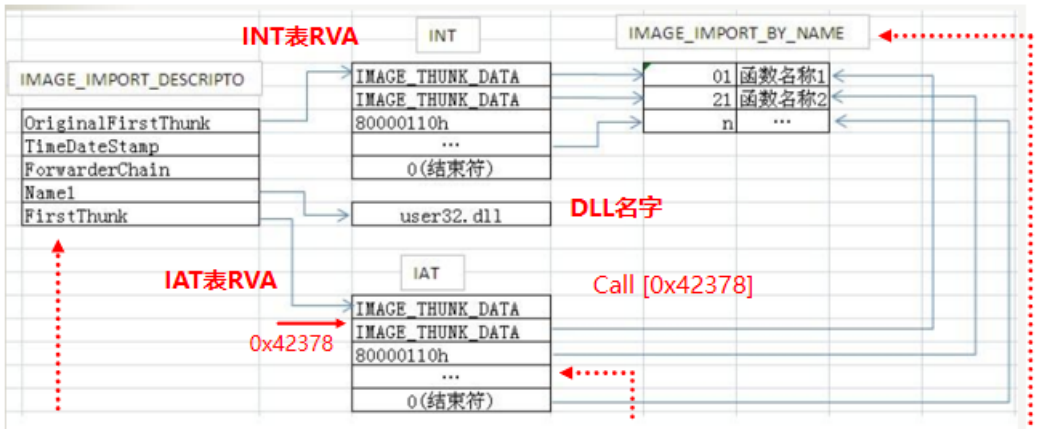
相对偏移的局限：Call 相对偏移的优点只能运用到同一个模块中（DLL，EXE），如果跨越了模块，调用DLL中的函数时，并不知道该模块会加载到哪里，因此，是不可能事先算出转跳偏移的。

解决办法：在编译器和系统协助下，利用导入表机制完成了这个工作

导入表机制

在PE文件的可选头的数据目录的第2项就是是导入表的描述，其中有导入表的RVA。RVA指向导入表（IMPORT DICTORY TABLE）起始，在这个地方，每一项是一个IMAGE_IMPORT_DESCRIPTOR结构，代表一个导入的DLL的相关信息。在这个结构中，又有INT表（Import Name Table）和IAT表（Import Address Table）的RVA。

- 加载前后，INT内容不变，就是IAT加载前的内容。可能INT并不存在，但如果有地址预绑定，就必须有INT。
- IAT是必须的，它就是好比是邮箱。编译后，IAT每项有4个字节，这4个字节是一个RVA，指向IAMGE_IMPORT_BY_NAME表中的一项，这一项就是对应函数的序号和函数名；加载后，IAT表的内容将变成函数入口地址。



IAT初始化算法：

1. 从数据目录获取导入表入口，从入口地方开始，每项代表一个被引用的DLL，从其中DLL名RVA段可获取DLL名，下面就引入此DLL
2. 系统遍历编译时生成的每个INA或IAT（如果有INT则判断INT，当预先绑定时，只能遍历INT），比如第2项，存储的值是一个RVA，指向IMAGE_IMPORT_BY_NAME中的对应项
3. 找到该项，获得其函数名串，以\0结尾，通过对应DLL的导出表找到相关函数的加载地址
4. 然后将其放入IAT第2项（此时，IAT表项的值才变为了函数的加载地址）
5. 如此遍历IAT，将所有项都填入对应函数入口地址
6. 遍历IMAGE_IMPORT_DESCRIPTOR表，对所有DLL都做2~4步处理。

为什么预先绑定时必须要有INT表：

1. 由于程序在加载时，需要在IAT表中填入函数的加载地址入口，因此，往往会比较耗时，为了减少加载时间，于是有了预先绑定技术
2. 预先绑定技术是在编译时就向IAT表中填入导入地址（即函数的入口地址，而不是指向IMAGE_IMPORT_BY_NAME表项的RVA），它是直接根据系统DLL的预期基址（ImageBase）计算出来的
3. 通过预先绑定，在实际加载时，只要系统DLL的基址没有发生改变，那么IAT表的内容就不需要再次填充，因此，加载时速度更块
4. 所以，采用预先绑定，在实际加载时，需要验证DLL是否被加载到预期地址。如果，引用的DLL没加载到预期地址，那就必须再次填充IAT表为实际的函数地址
5. 这时，已经不能使用IAT表来指向IMAGE_IMPORT_BY_NAME表项了（编译时RVA已经被覆盖为函数预期入口地址），但INT表可以，通过INT表再次执行导入机制，从而使得IAT表内容更新

注：只要INT存在，则用它检索IMAGE_IMPORT_BY_NAME，否则就用IAT表检索

用导入表调用DLL函数的指令

- CALL [xx xx xx xx] （xxxx就是IAT表项地址）

  对应的关键机器码是 FF 15 xx xx xx xx

- CALL ... JMP [xx xx xx xx] （xxxx就是IAT表项地址，CALL会到JMP [xxxx]指令，然后再跳到函数入口）

  对应的关键机器码是 FF 25 xx xx xx xx

因此，病毒代码应该是要寻找具有FF 15和FF 25这种特殊形式的指令。

当病毒使用CALL指令进行patch时。不用处理返回问题，API执行完毕，自然通过ret返回。

间接跳转指令 Call [xx] 或 JMP [xx] 的Patch方法

1. 假定我们对调用某个函数（如GetCommandLineX）的CALL和JMP感兴趣，首先查找该函数的导入表项的地址xx xx xx xx（邮箱地址）
2. 得到地址后，查找代码节以找到间接跳转指令：

   FF 15 xx xx xx xx (CALL [xx xx xx xx]) 或 FF 25 xx xx xx xx (JMP [xx xx xx xx])

3. 分别Patch为跳转到病毒指令：

   E8 yy yy yy yy 90（CALL 偏移，NOP）或 E9 yy yy yy yy（ JMP 偏移）

第十四章 Win病毒-指令Patch实现+重定位表

如何找到需要Patch的指令

1. 首先指定一个会被大概率调用的函数名（也包括函数所在DLL的名字），如Kernel32中的GetCommandLineX，X=A/W
2. 然后通过被寄生文件（exe）的导入表找到该函数的导入表项（即IAT中的对应项地址）的地址XXXX
3. 最后去exe文件的代码中搜索所有可能的Call [XXXX] 或 JMP [XXXX]后，即 FF 15 XXXX 或 FF 25 XXXX
4. 找到后，说明该指令极有可能就是我们提供函数的调用指令，然后进行Patch，这样找到的指令的准确度对病毒来讲，完全够用

用程序实现指令Patch的步骤

1. 在文件中找到导入表的位置
2. 找到指定API函数在IAT表中的表项地址（遍历读取DLL的名字；找到DLL，获得INT或IAT表位置；查找所需的函数名）
3. 找到符合的指令进行Patch（需要先判断指令是否在代码节，即判断节的属性是否有0x20）

PE文件的重定位机制

程序因加载到非编译期约定地址时，就必需修改那些包含了绝对地址的指令（因为这些绝对地址已经不是原来的地址了），这称为程序重定位，由加载器（loader）完成

如何进行重定位：

1. 实际和预期加载地址的差x = A – B。
2. 找到需要修改的位置y
3. 读出y开始4字节的值 + x = 新地址值z
4. 将z写入y开始的4字节

如何知道哪些地方需要重定位：

在可选头的数据目录中，有一项（第6项）就是重定位表，而重定位表中就记录了所有需要进行重定位修改的位置。

从OptionHeader的数据目录项拿到重定位表首，然后遍历上面的数据表结构，获取每个重定位项，计算重定位的位置，按之前的算法重定位。（重定位表中的每一项就只需要12位（1.5个字节）来表示地址，表示它离该区域起始 RVA 的偏移，另外有0.5字节为属性，这样重定位中的每项为 2 字节）因为被修改的都是地址值（32 位机上 4 字节），那么 Loader 每次就根据这个位置定位 4 字节进行修改。

总结Patch指令引起的问题

因为我们进行Patch的原指令Call [xxxx]和Jmp [xxx]本身是包含绝对地址的指令，对于exe能重定位的情况下，正常会有指向这个绝对地址xxxx所在位置的重定位项。在我们Patch指令的过程中，我们将该指令修改为了不包含绝对地址的指令形式(即：Call 偏移，Jmp 偏移)，但并没有删除针对该指令的重定向项，那么重定位后，就会对这个重定位项指向的位置（即指向了我们Patch后的指令）进行修改，那么我们Patch后的指令就会被篡改。

解决方法：让exe重定位项失效，或删除这些被Patch指令的重定位项。

- 关闭随机基址：在OptionalHeader中有DLLCharacteristic这样的属性字段，其值0040h（Image_DallCharactersitics_Dynamic_base）表示了可以在加载时被重定位（随机基址），我们需要去掉这个属性。

- 去掉所对应的重定位项：填充补齐 reloc 节、修改重定位小表的 Size、修改数据目录中重定位项的 Size

第十六章 反病毒技术简介

计算机病毒防治技术概述

计算机病毒的防治技术分成四个方面： 病毒预防、病毒检测、病毒消除、病毒免疫

计算机病毒防范，是指通过建立合理的计算机病毒防范体系和制度，及时发现计算机病毒侵入，并采取有效的手段阻止计算机病毒的传播和破坏，恢复受影响的计算机系统和数据。计算机病毒的防范，就是要在病毒执行之前进行阻断，需要监视、跟踪系统内类似的操作，提供对系统的保护，最大限度地避免各种计算机病毒的传染破坏，往往需要基于全系统的内核级行为监控

计算机病毒的预防措施可概括为两点：勤备份、严防守

计算机检测技术分类：手工检测（可以剖析病毒、可以检测一些自动检测工具不能识别的新病毒）、自动检测（可方便地检测大量的病毒，自动检测工具的发展总是滞后于病毒的发展）

常见计算机病毒的诊断方法及原理

1. 比较诊断法：用原始的正常备份与被检测的内容(引导扇区或被检测的文件)进行比较（长度比较法、内容比较法、内存比较法、中断比较法）。
2. 校验和诊断法：根据正常文件的信息(如文件名称、大小、时间、日期及内容)，计算其校验和（checksum），计算新的校验和与原来保存的校验和是否一致。
3. 扫描诊断法：扫描法是用每一种病毒体含有的特定病毒码(Virus Pattern)对被检测的对象进行扫描。如果在被检测对象内部发现了某一种特定病毒码，就表明发现了该病毒码所代表的病毒。（特征代码扫描法、特征字扫描法）扫描法的核心是病毒特征码的选择。
4. 行为监测诊断法：利用病毒的特有行为特性监测病毒的方法，称为行为监测法（人工智能）。
5. 感染实验诊断法：利用病毒最基本的特征——感染特征，所有病毒都会进行感染，如果不感染，就不称其为病毒。将正常的文件放入异常的系统中去运行，看这些正常文件是否会被感染，如果被感染，则文件内容会发生变化（通过校验和等检测），则断言系统中存在病毒。
6. 软件模拟诊断法：多态病毒每次感染都变化其病毒代码，对付这种病毒，特征码扫描法失效。我们把使用通常特征码扫描法无法检测（或几乎很难检测）的病毒称之为多态病毒。（多态病毒常采用以下几种方式来不断变换自己：采用等价代码对原有代码替换、改变与执行次序无关的指令的次序、增加许多垃圾指令、对原有计算机病毒代码进行加密等）多态病毒的多态性通常表现在病毒代码和病毒行为的不固定性，但是，每一个多态病毒在执行时都需要还原，如先执行一段解密代码进行解密，再执行解密后的病毒代码。软件模拟诊断法就是针对解密后的病毒代码所提出来的一种诊断法。软件模拟（Software Emulation）诊断法又称为解密引擎、虚拟机执行技术或软件仿真技术，它的本质就是一种软件运行时的分析器。它通过在一个模拟的虚拟环境下运行计算机病毒，等待计算机病毒自身进行解密完成后，再对解密后的病毒代码实施特征码的识别，识别病毒种类后再进行有相关的清除和查杀工作。
7. 分析诊断法：分析法一般只被专业反病毒技术人员使用。分析法通常包括静态分析和动态调试两个步骤，Windows上一般的调试工具主要有OD、WinDBG、X64DBG等，对复杂的病毒程序，必须采用动、静结合的分析方法。

启发式代码扫描技术

启发式代码扫描技术基于给定的判断规则和定义的扫描技术，若发现被扫描程序中存在可疑的程序功能指令，则作出存在病毒的预警或判断。

启发式代码分析扫描技术与传统的检测扫描技术的对比：

1. 传统的扫描技术基于对已知病毒的分析和研究，在检测已知病毒时能够更准确、减少误报；但对没有出现过的新病毒，由于其知识库并不存在该类(种)病毒的特征数据，则有可能产生漏报的严重后果
2. 启发式代码分析技术则可以有效地应为未知的新病毒！

3. 传统扫描技术与启发式扫描技术的应该相互结合，相辅相成

虚拟机查毒技术

虚拟机查毒实际上是自动跟踪病毒入口的解密代码，当其将加密的病毒体按其解密算法进行解密后，就可以得到解密后的病毒明文

目的：为了对抗加密变形病毒

目前有两种方法可以跟踪控制病毒的每一步执行，并能够在病毒循环解密结束后从内存中读出病毒体明文：

1. 单步和断点跟踪法（和目前一些程序调试器相类似）：

   优点：它不用处理每条指令的执行；

   缺点：①易于被病毒察觉；②要求待查可执行文件真实执行

2. 虚拟执行法：

   优点：①虚拟执行不会向栈中压入单步断点和中断返回地址，不可能被病毒察觉 ②虚拟机可以完全记录每条指令执行的地址和细节并加以控制 ③虚拟机中的地址空间是在虚拟机内部，完全做到了'虚拟执行'，不会有病毒破坏的影响

   缺点：它必须在内部处理所有指令的执行，这意味着它需要编写大量的特定指令处理函数来模拟每种指令的执行效果。

反虚拟执行技术：

1. 插入特殊指令技术

原理：虚拟机是模拟CPU的执行，并不是真正的CPU，所以不可以能对整个Intel的指令集进行支持，遇到不认识的指令就会停止工作

方法：病毒插入特殊指令（如3DNoW），这些指令对病毒本身没有影响，但会干扰虚拟机的工作

应对：不需要针对每个特殊指令写专门的模拟函数，只需要构建特殊指令的指令长度表，当EIP指向特殊指令时，就跳过特殊指令长度，或者发现这些特殊指令时，交由CPU去真正执行

2. 结构化异常处理技术

原理：虚拟机仅仅模拟了CPU的工作过程，而对于异常处理等系统机制没有进行处理，虚拟机会在遇到非法指令、进入异常处理函数前停止工作

方法：将解密代码置于自己的异常处理函数，并故意引发异常处理函数

应对：为虚拟机赋予发现和记录异常的功能，并在引发异常时将控制转向异常处理函数

3. 入口点模糊技术（EPO）

原理：即便是虚拟执行，也不可能查找文件的所有代码，虚拟执行通常会在规定步数内，检查待查文件是否具有解密循环，如果没有，就会判定该文件没有携带加密变形病毒，产生漏报

方法：病毒在宿主程序执行到某个位置时再获得控制权

应对：合理的增加检查的规定步数，如果规定步数较小，极易产生漏报，但规定步数也不能盲目增加，否则会无谓增加检测时间，如何确定规定步数的大小实在是件难事

实践上，这类病毒编写技术难度较大，在没有反汇编等工具的帮助下，很难在宿主体内定位一条指令，同时保证超过规定步数，又保证一定会被执行

病毒实时监控技术

在文件打开、关闭、清除、写入等操作时检查文件是否是病毒携带者，如果是则根据用户的决定选择不同的处理方案，如清除病毒、禁止访问该文件、删除该文件或简单地忽略，从而有效地避免病毒在本地计算机上的感染传播

病毒实时监控的设计的难点：

1. 驱动程序的编写难度很大
2. 驱动程序与Ring3下客户程序的通信问题
3. 驱动程序所占用资源问题

计算机病毒免疫技术

1. 针对某一种病毒进行的计算机病毒免疫

   一个免疫程序只能预防一种计算机病毒，例如对小球病毒，在DOS引导扇区的1FCH处填上1357H，小球病毒检查到该标志就不再对它进行感染

   优点：是可以有效地防止某一种特定病毒的传染

   缺点：

   局限性，无法处理不设置感染标识或设置后不能有效判断的病毒

   病毒变种不再使用这个免疫标志、或出现新病毒时，失去作用

   某些病毒的免疫标志不容易仿制，若必须加上这种标志，则需对原文件做大的改动，例如大麻病毒；

   由于病毒的种类较多，不可能对各种病毒都加上免疫标识

   只能阻止传染，却不能阻止病毒的破坏行为，对已中毒文件无能为力

2. 基于自我完整性检查的计算机病毒免疫

   目前这种方法只能用于文件而不能用于引导扇区

   原理：为可执行程序增加一个免疫外壳（1~3KB），同时在免疫外壳中记录有关用于恢复自身的信息。执行具有这种免疫功能的程序时，免疫外壳首先得到运行，检查自身的程序大小、校验和、生成日期和时间等情况，没有发现异常后，再转去执行受保护的程序

   优点：这种方法不只是针对病毒的，由于其他原因造成的文件变化，在大多数情况下免疫外壳程序都能使文件自身得到复原

   缺点：

   - 每个受到保护的文件都要增加额外的存储空间
   - 现有一些校验码算法不能满足防病毒需要，无法检查某些病毒感染的文件
   - 无法对抗覆盖方式的文件型病毒
   - 有些类型的文件不能使用免疫外壳的防护方法，否则将不能正常执行
   - 当已被病毒感染的文件也被免疫外壳包裹，妨碍反病毒软件的检测清除