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

Anti Reverse Engineering Uncovered

Nicolas Brulez*

E-Mail: 0x90@Rstack.org

* Corresponding Author

Received: 07. Mar. 2005, Accepted: 12. Mar. 2005, Published: 13. Mar. 2005

This work has been previously published at the Honeynet Project, Scan of the Month 33.

Abstract

Rather than doing another complete analysis of the binary, i will rather present the techniques i have used in the challenge, and how i have implemented them. The Scan of the Month 33 was released by the Honeynet Project in November 2004. I invite everyone to read the excellent submissions we received this month once they have read my paper. I am presenting the binary from the protection author point of view, while they presented it from the analyst point of view. You will learn the methods and techniques used to Protect / Unprotect a binary with this month's challenge. A lot of weaknesses were left on purpose in this binary and they will be presented here.

Keywords: *Software Protection; Reverse Code Engineering; Linux; Anti-Debugging; Anti-Anti-Debugging*

1. Introduction

This month's challenge is to analyze an unknown binary, in an effort to reinforce the value of reverse engineering, and improve (by learning from the security community) the methods, tools and procedures used to do it. This challenge is similar to SotM 32. However, this binary has mechanisms implemented to make the binary much harder to analyze, to protect against reverse engineering.

Skill Level: Advanced/Expert

All we are going to tell you about the binary is that it was 'found' on a WinXP system and has now be sent to you for analysis. You will have to analyse it in-depth and get as much information as possible about its inner working, and what is the goal of the binary. The main goal of this challenge is to teach people how to analyse heavily armored binaries. Such techniques could be used in the future, and its time to get used to them.

2. Identify and explain any techniques in the binary that protect it from being analyzed or reverse engineered

Many techniques have been used in order to slow down analysis and break reverse engineers tools:

- **PE Header Modifications**

  Many fields of the PE header were modified in order to disturb analysing tools, and thus, the Reverse Engineer. I will quickly cover the most important changes:

```
->Optional Header
   Magic:                        0x010B  (HDR32_MAGIC)
   MajorLinkerVersion:           0x02
   MinorLinkerVersion:           0x19  -> 2.25
   SizeOfCode:                   0x00000200
   SizeOfInitializedData:        0x00045400
   SizeOfUninitializedData:      0x00000000
   AddressOfEntryPoint:          0x00002000
   BaseOfCode:                   0x00001000
   BaseOfData:                   0x00002000
   ImageBase:                    0x00DE0000  <--- "Non Standard" ImageBase
   SectionAlignment:             0x00001000
   FileAlignment:                0x00001000
   MajorOperatingSystemVersion:  0x0001
   MinorOperatingSystemVersion:  0x0000  -> 1.00
   MajorImageVersion:            0x0000
   MinorImageVersion:            0x0000  -> 0.00
   MajorSubsystemVersion:        0x0004
   MinorSubsystemVersion:        0x0000  -> 4.00
   Win32VersionValue:            0x00000000
   SizeOfImage:                  0x00049000
   SizeOfHeaders:                0x00001000
   CheckSum:                     0x00000000
   Subsystem:                    0x0003  (WINDOWS_CUI)
   DllCharacteristics:           0x0000
   SizeOfStackReserve:           0x00100000
   SizeOfStackCommit:            0x00002000
   SizeOfHeapReserve:            0x00100000
   SizeOfHeapCommit:             0x00001000
   LoaderFlags:                  0xABDBFFDE <--- Bogus Value
   NumberOfRvaAndSizes:          0xDFFFDDDE <--- Bogus Value
```

The "standard" **ImageBase** usually is 400000 for Win32 applications and Reverse Engineers are used to analyse programs with such an ImageBase. While it isn't a protection by itself, this simple modification will confuse some Reverse Engineers, because they aren't used to such memory addresses.

"Anti" OllyDbg:

LoaderFlags and **NumberOfRvaAndSizes** were modified.. I have Reverse Engineered OllyDBG and Soft ICE to find a few tricks that could slow down the analysis of a binary. With those two modifications, Olly will pretend that the binary isn't a good image and will eventually run the application without breaking at its entry point. This could be a bad thing if you wanted to debug a malware on your computer, because you would get infected.

Anti Soft ICE : Blue Screen of Death and no Chocolate:

The **NumberOfRvaAndSizes** field has been modified in order to reboot any computer running a recent version of Soft ICE. While Disassembling the PE Loader of Soft ICE, i found a very critical vulnerability in Soft ICE that allows one binary to crash any computer running Soft ICE without any code execution. This vulnerability (bug) has been reported to Compuware and should be fixed in the next version. Apparently it didn't happen on some of the authors of the submissions for some reasons. Oh well.

Here is the disassembly of Soft ICE PE loader to find out why it reboots your computer:

```
.text:000A79FE
.text:000A79FE loc_A79FE:                               ; CODE XREF:
sub_A79B9+31j
.text:000A79FE                                          ; sub_A79B9+3Cj
.text:000A79FE                                          ; DATA XREF:
.text:00012F9Bo
.text:000A79FE                sti
.text:000A79FF                mov     esi, ecx
.text:000A7A01                mov     ax, [esi]
.text:000A7A04                cmp     ax, 'ZM'
.text:000A7A08                jnz     not_PE_file
.text:000A7A08
.text:000A7A0E                mov     edi, [esi+_IMAGE_DOS_HEADER.e_lfanew]
.text:000A7A11                add     edi, esi
.text:000A7A13                mov     ax, [edi]
.text:000A7A16                cmp     ax, 'EP'
.text:000A7A1A                jnz     not_PE_file
.text:000A7A1A
.text:000A7A20                movzx   ecx,
[edi+IMAGE_NT_HEADERS.FileHeader.NumberOfSections]
.text:000A7A24                or      ecx, ecx
.text:000A7A26                jz      not_PE_file
.text:000A7A26
.text:000A7A2C                mov     eax,
[edi+IMAGE_NT_HEADERS.OptionalHeader.NumberOfRvaAndSizes]
.text:000A7A2F                lea     edi,
[edi+eax*8+IMAGE_NT_HEADERS.OptionalHeader.DataDirectory]
.text:000A7A33                mov     eax, ecx
.text:000A7A35                imul    eax, 28h
.text:000A7A38                mov     al, [eax+edi]   ; CRITICAL BUG! One
can force EAX+EDI to be equal to zero. Reading at [0] in ring 0 isn't nice eh
;-)
.text:000A7A3B
.text:000A7A3B loc_A7A3B:                               ; DATA XREF:
.text:00012FA5o
.text:000A7A3B                cli
```

```
.text:000A7A3C                 call    sub_15C08
.text:000A7A3C
.text:000A7A41                 mov     byte_FA259, 0
.text:000A7A48                 push    eax             ; Save EAX
.text:000A7A49                 mov     eax, dword_16B56F ; EAX is modified by
a saved dword
.text:000A7A4E                 mov     dr7, eax        ; Debug Register 7
take the value in EAX
.text:000A7A51                 pop     eax             ; EAX is restored
.text:000A7A52                 mov     dword_FC6CC, esp
.text:000A7A58                 mov     esp, offset unk_FBABC
.text:000A7A5D                 and     esp, 0FFFFFFFCh
.text:000A7A60                 xor     al, al          ; AL is zeroed? Why
this mov al, [eax+edi] then ?
.text:000A7A60                                         ; I don't see the
point. old code?
.text:000A7A62                 call    sub_4D2EB
.text:000A7A62
.text:000A7A67                 call    sub_36AC1
.text:000A7A67
.text:000A7A6C                 xor     edx, edx
.text:000A7A6E
.text:000A7A6E loc_A7A6E:                              ; CODE XREF:
sub_A79B9+124j
.text:000A7A6E                 call    sub_74916
.text:000A7A6E
```

As you can see from the code above, we can force Soft ICE to read at memory location [0] or something similar using a special value inside the PE header. For this binary i didn't bother calculating the exact value to read at address [0], that's may explain why it didn't crash for some people.I won't explain how to calculate this special value because it is trivial and i don't want Darklords to use that trick without a little brainstorming.

To fix this problems, one needs to patch the value in the PE Header. The standard value for NumberOfRvaAndSizes is 0x10.Just patch this value in the PE Header and the Soft ICE wrecking will be gone. The OllyDBG problem as well, because it is based on BOTH fields modifications. You can also nullify the other field if you want.

- **Section Modification: Or how to kill many tools.**

```
->Section Header Table
   1. item:
    Name:                   CODE
    VirtualSize:            0x00001000
    VirtualAddress:         0x00001000
    SizeOfRawData:          0x00001000
    PointerToRawData:       0x00001000
    PointerToRelocations:   0x00000000
    PointerToLinenumbers:   0x00000000
    NumberOfRelocations:    0x0000
    NumberOfLinenumbers:    0x0000
    Characteristics:        0xE0000020
    (CODE, EXECUTE, READ, WRITE)

   2. item:
    Name:                   DATA
    VirtualSize:            0x00045000
    VirtualAddress:         0x00002000
    SizeOfRawData:          0x00045000
    PointerToRawData:       0x00002000
    PointerToRelocations:   0x00000000
```

```
     PointerToLinenumbers:  0x00000000
     NumberOfRelocations:   0x0000
     NumberOfLinenumbers:   0x0000
     Characteristics:       0xC0000040
     (INITIALIZED_DATA, READ, WRITE)

    3. item:
     Name:                  NicolasB
     VirtualSize:           0x00001000
     VirtualAddress:        0x00047000
     SizeOfRawData:         0xEFEFADFF   <--- BIG Size of section on the disk.
     PointerToRawData:      0x00047000
     PointerToRelocations:  0x00000000
     PointerToLinenumbers:  0x00000000
     NumberOfRelocations:   0x0000
     NumberOfLinenumbers:   0x0000
     Characteristics:       0xC0000040
     (INITIALIZED_DATA, READ, WRITE)

    4. item:
     Name:                  .idata
     VirtualSize:           0x00001000
     VirtualAddress:        0x00048000
     SizeOfRawData:         0x00001000
     PointerToRawData:      0x00047000
     PointerToRelocations:  0x00000000
     PointerToLinenumbers:  0x00000000
     NumberOfRelocations:   0x0000
     NumberOfLinenumbers:   0x0000
     Characteristics:       0xC0000040
     (INITIALIZED_DATA, READ, WRITE)
```

From those informations, we can conclude a few things. First, the binary doesn't seem to be compressed, because the Virtual Address and Size matche the Raw Offset and Size at one exception, the NicolasB section. This section has an extremly big size of raw data, which will crash a few tools and make a few others very very slow.

IDA will try to allocate a LOT of memory because it thinks that the section is THAT big, turning your computer into a very slow turtle ;-). Eventually, it will load the file, or run out of memory, depending of the computer you are using to do the analysis.

This modification will also create havoc with many tools such as Objdump, PE editor, some memory dumpers etc. It is very easy to fix this problem, you need to correct the Raw Size. If you look at the section following this special one, you will find that it starts at the very same Raw Offset. This means that the other section is actually null on the disk. You can therefore, safely replace the big value by zero.

Protection Weakness:

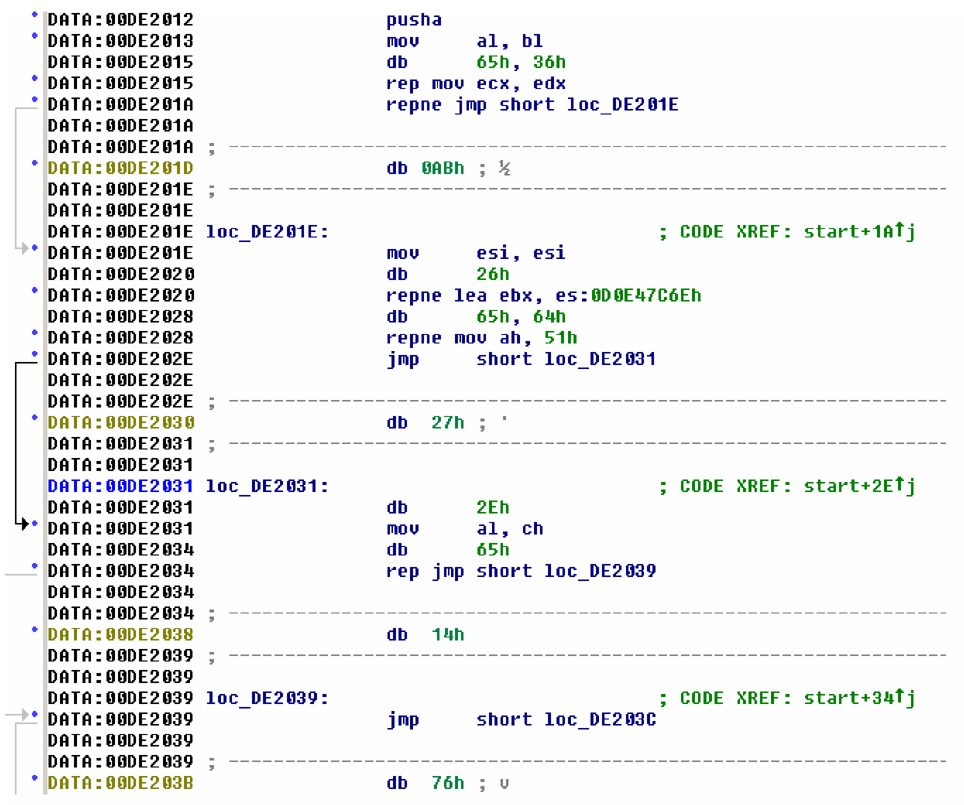
While writing this binary, i knew people were going to patch the PE header but i didn't do any integrity checks on purpose. Originally i wanted to use the value in the PE Header as ***keys*** *to decrypt a few layers of the protection, and the result would have been an unworking binary if this one had been changed.*

I have also changed a few other things in the PE header, but nothing of real interest here. (who said Cosmetic?)

- **Junk Code**

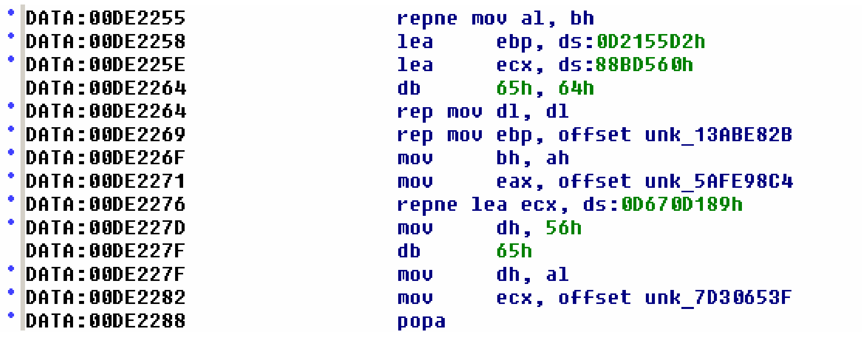
All along the binary, i have added junk code between real instructions, in order to make the analysis a little harder. The junk code are long blocks of code that does nothing but fancy operations to disturb the analyst , especially when he choose to do a static analysis of the binary. **Each** block of Junk Code is **different** and have been generated by a personal tool. A **Thrash generator** which creates macros to be inserted in the code source around real instructions.

Here is how it looks inside a disassembler:



```
DATA:00DE2012                 pusha
DATA:00DE2013                 mov     al, bl
DATA:00DE2015                 db      65h, 36h
DATA:00DE2015                 rep mov ecx, edx
DATA:00DE201A                 repne jmp short loc_DE201E
DATA:00DE201A
DATA:00DE201A ; ---------------------------------------------------------------------------
DATA:00DE201D                 db 0ABh ; ½
DATA:00DE201E ; ---------------------------------------------------------------------------
DATA:00DE201E
DATA:00DE201E loc_DE201E:                             ; CODE XREF: start+1A↑j
DATA:00DE201E                 mov     esi, esi
DATA:00DE2020                 db      26h
DATA:00DE2020                 repne lea ebx, es:0D0E47C6Eh
DATA:00DE2028                 db      65h, 64h
DATA:00DE2028                 repne mov ah, 51h
DATA:00DE202E                 jmp     short loc_DE2031
DATA:00DE202E
DATA:00DE202E ; ---------------------------------------------------------------------------
DATA:00DE2030                 db  27h ; '
DATA:00DE2031 ; ---------------------------------------------------------------------------
DATA:00DE2031
DATA:00DE2031 loc_DE2031:                             ; CODE XREF: start+2E↑j
DATA:00DE2031                 db      2Eh
DATA:00DE2031                 mov     al, ch
DATA:00DE2034                 db      65h
DATA:00DE2034                 rep jmp short loc_DE2039
DATA:00DE2034
DATA:00DE2034 ; ---------------------------------------------------------------------------
DATA:00DE2038                 db  14h
DATA:00DE2039 ; ---------------------------------------------------------------------------
DATA:00DE2039
DATA:00DE2039 loc_DE2039:                             ; CODE XREF: start+34↑j
DATA:00DE2039                 jmp     short loc_DE203C
DATA:00DE2039
DATA:00DE2039 ; ---------------------------------------------------------------------------
DATA:00DE203B                 db  76h ; v
```

The junk code starts with a pushad (save all registers states onto the stack) and finish with a popad (restore register states).Here is the end of a block of junk:



```
DATA:00DE2255                 repne mov al, bh
DATA:00DE2258                 lea     ebp, ds:0D2155D2h
DATA:00DE225E                 lea     ecx, ds:88BD560h
DATA:00DE2264                 db      65h, 64h
DATA:00DE2264                 rep mov dl, dl
DATA:00DE2269                 rep mov ebp, offset unk_13ABE82B
DATA:00DE226F                 mov     bh, ah
DATA:00DE2271                 mov     eax, offset unk_5AFE98C4
DATA:00DE2276                 repne lea ecx, ds:0D670D189h
DATA:00DE227D                 mov     dh, 56h
DATA:00DE227F                 db      65h
DATA:00DE227F                 mov     dh, al
DATA:00DE2282                 mov     ecx, offset unk_7D30653F
DATA:00DE2288                 popa
```

Protection Weakness:

The thrash generator isn't perfect (at least with the options i have used here ;) and it is easy to find the start and the end of a block of junk code. The junk code is bounded by pushad/popad. When i wrote this binary i was aware of this problem, but this is a perfect real life example of protection weakness. It allows Reverse Engineers to practice IDA/Ollydbg scripting. Very interesting scripts were found in the submissions. I invite you to have a look at them if you didn't know how to write one. When i wrote the binary, i already had a better version of my Thrash generator that doesn't use any pushad/popad around the blocks of useless code, but we will keep it for another challenge, if any.

- ## SEH - Structured Exception Handling

Windows SEH were used extensively in this binary. It allows one to access the context structure of the current application, and therefore, access privileged registers such as Debug Registers. Those registers are used by Hardware Breakpoints (BPM). If you can access them, you can also erase the hardware breakpoints.

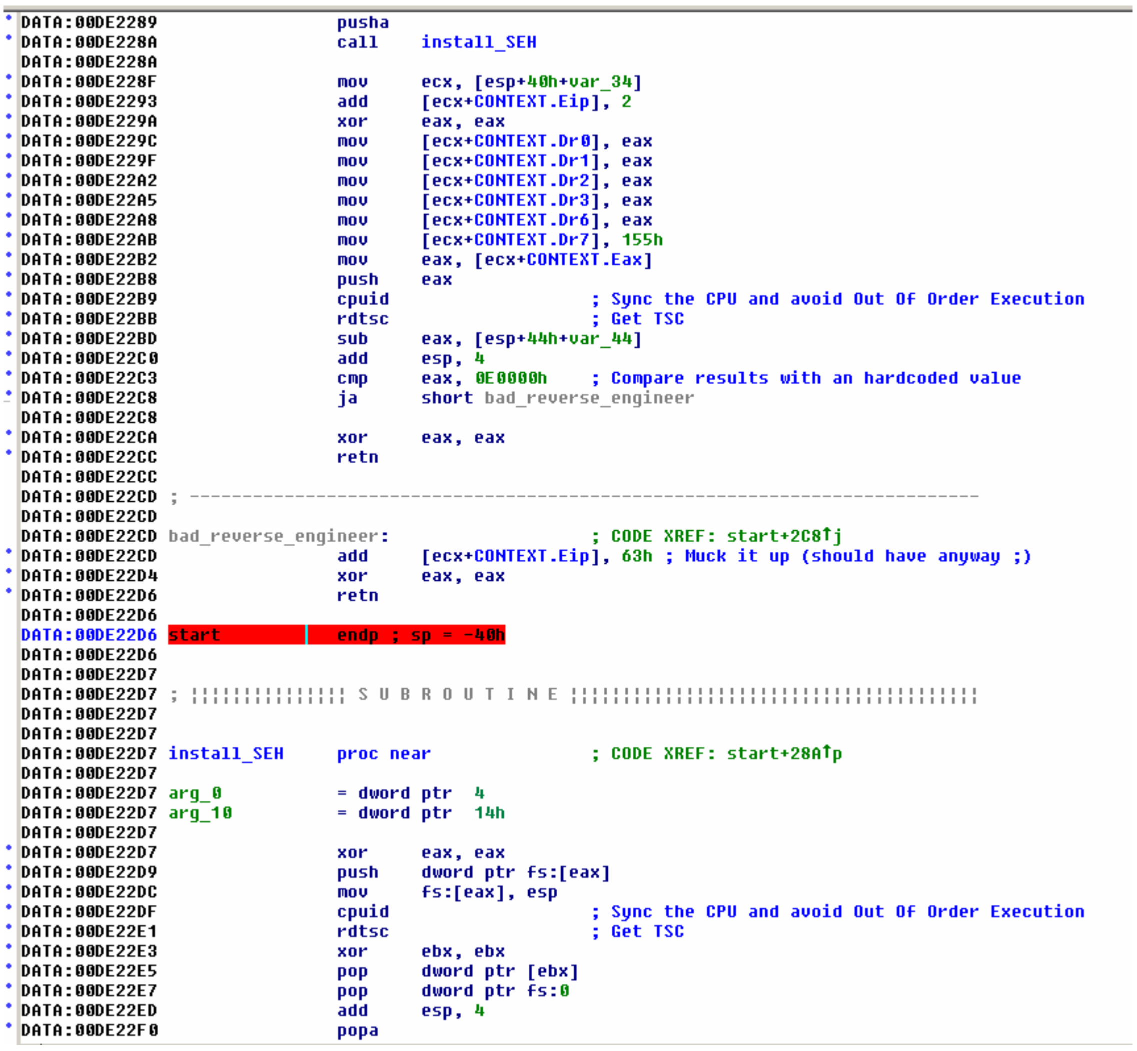
- ## Timing Detection Through SEH

Here is a little detection i invented to detect debuggers. If we merge SEH (And access to context structure) with the known Timing Detection Technique, we can detect a lot of Ring 3 debuggers and Tracers. The idea is to read the Time Stamp Counter using RDTSC (number of cycles executed by the CPU basically) and then generating an Exception.

In the exception handler, we can access the EAX register (previously modified by RDTSC) in the Context Structure, which contains the TSC. In the Exception Handler, we use RDTSC one more time, to get the current TSC value. Now, we can compare both TSC to see whether the program has been debugged/traced or not. If such an action has occured, the difference of cycles will be huges, thus triggering the Payload. In this binary, i just modified EIP through the context structure. The application resumes at a different location skipping mandatory instructions.The application crashes eventually. It seems that on some version on Windows, it doesn't work as expected because of the utilisation of the CPUID instruction, that will modify the ECX register.

The detection became less stealth because of this "bug", but it would still have been a matter of time until someone discovered it anyway. Many people wondered why i used CPUID in the program before RDTSC. The reason is that on recent CPU such as P4, there is a feature called: Out of Order Execution. The CPUID is a synchronization instruction which tells to the CPU not to use Out Of Order execution, avoiding False Positives in the debugger detection. If you don't tell to the CPU not to use OOO execution, you don't know in which order the CPU is going to execute your code. It can be different from your source code. Sometimes, it will create a false positive and your program will crash for no reason.

Here is the code of this detection:



```
DATA:00DE2289                 pusha
DATA:00DE228A                 call    install_SEH
DATA:00DE228A
DATA:00DE228F                 mov     ecx, [esp+40h+var_34]
DATA:00DE2293                 add     [ecx+CONTEXT.Eip], 2
DATA:00DE229A                 xor     eax, eax
DATA:00DE229C                 mov     [ecx+CONTEXT.Dr0], eax
DATA:00DE229F                 mov     [ecx+CONTEXT.Dr1], eax
DATA:00DE22A2                 mov     [ecx+CONTEXT.Dr2], eax
DATA:00DE22A5                 mov     [ecx+CONTEXT.Dr3], eax
DATA:00DE22A8                 mov     [ecx+CONTEXT.Dr6], eax
DATA:00DE22AB                 mov     [ecx+CONTEXT.Dr7], 155h
DATA:00DE22B2                 mov     eax, [ecx+CONTEXT.Eax]
DATA:00DE22B8                 push    eax
DATA:00DE22B9                 cpuid                   ; Sync the CPU and avoid Out Of Order Execution
DATA:00DE22BB                 rdtsc                   ; Get TSC
DATA:00DE22BD                 sub     eax, [esp+44h+var_44]
DATA:00DE22C0                 add     esp, 4
DATA:00DE22C3                 cmp     eax, 0E0000h    ; Compare results with an hardcoded value
DATA:00DE22C8                 ja      short bad_reverse_engineer
DATA:00DE22C8
DATA:00DE22CA                 xor     eax, eax
DATA:00DE22CC                 retn
DATA:00DE22CC
DATA:00DE22CD ; ---------------------------------------------------------------------------
DATA:00DE22CD
DATA:00DE22CD bad_reverse_engineer:                   ; CODE XREF: start+2C8↑j
DATA:00DE22CD                 add     [ecx+CONTEXT.Eip], 63h ; Muck it up (should have anyway ;)
DATA:00DE22D4                 xor     eax, eax
DATA:00DE22D6                 retn
DATA:00DE22D6
DATA:00DE22D6 start           endp ; sp = -40h
DATA:00DE22D6
DATA:00DE22D7
DATA:00DE22D7 ; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
DATA:00DE22D7
DATA:00DE22D7
DATA:00DE22D7 install_SEH     proc near               ; CODE XREF: start+28A↑p
DATA:00DE22D7
DATA:00DE22D7 arg_0           = dword ptr  4
DATA:00DE22D7 arg_10          = dword ptr  14h
DATA:00DE22D7
DATA:00DE22D7                 xor     eax, eax
DATA:00DE22D9                 push    dword ptr fs:[eax]
DATA:00DE22DC                 mov     fs:[eax], esp
DATA:00DE22DF                 cpuid                   ; Sync the CPU and avoid Out Of Order Execution
DATA:00DE22E1                 rdtsc                   ; Get TSC
DATA:00DE22E3                 xor     ebx, ebx
DATA:00DE22E5                 pop     dword ptr [ebx]
DATA:00DE22E7                 pop     dword ptr fs:0
DATA:00DE22ED                 add     esp, 4
DATA:00DE22F0                 popa
```

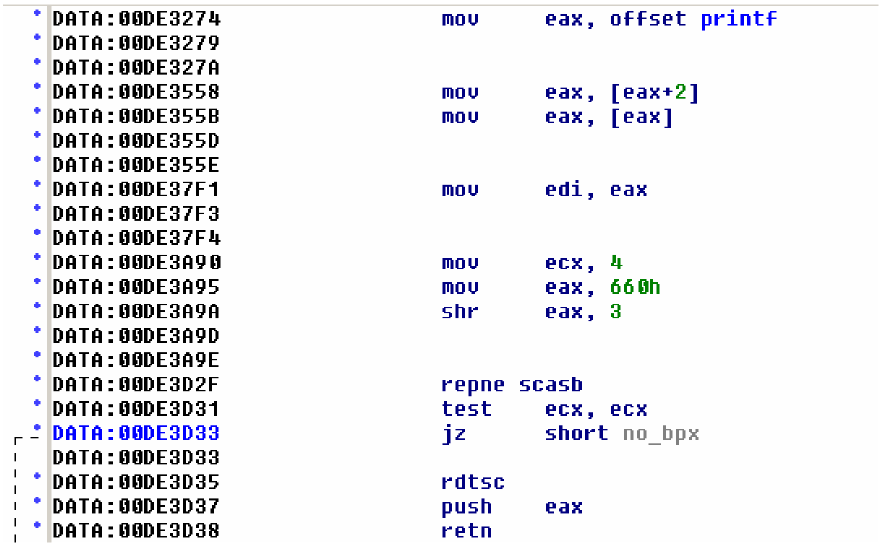
E0000h is the maximum cycles difference accepted by this detection. If the number is bigger, then a debugger is most likely running and debugging our application.

Protection Weakness:

I have used a fixed value for the number of cycles: E0000h. I could have (Actually i can do it with my layer generator) used a random value rather than a constant and therefore, making the scan for this constant useless. I could also have used different instructions for each SEH to make the creation of a generic pattern difficult. The biggest weakness of this detection is the constant and the usage of the same instructions for every checks. It is also possible to write a Kernel Module Driver to catch every execution of RDTSC (See Intel documentation for further informations) and return very similar values, thus bypassing the detection completely.

- **BPX Detection:**

As we are going to use API functions, We have to protect them from beeing BPX'ed by an attacker. Rather than Using GetProcAddress to get the API address and then to check for an int 3 opcode (0xCC) in the API function code, i have used a different method. I directly access the Import Table , more precisely, the Import Address Table to read the API function address and then start to search for breakpoints.



```
DATA:00DE3274      mov     eax, offset printf
DATA:00DE3279
DATA:00DE327A
DATA:00DE3558      mov     eax, [eax+2]
DATA:00DE355B      mov     eax, [eax]
DATA:00DE355D
DATA:00DE355E
DATA:00DE37F1      mov     edi, eax
DATA:00DE37F3
DATA:00DE37F4
DATA:00DE3A90      mov     ecx, 4
DATA:00DE3A95      mov     eax, 660h
DATA:00DE3A9A      shr     eax, 3
DATA:00DE3A9D
DATA:00DE3A9E
DATA:00DE3D2F      repne scasb
DATA:00DE3D31      test    ecx, ecx
DATA:00DE3D33      jz      short no_bpx
DATA:00DE3D33
DATA:00DE3D35      rdtsc
DATA:00DE3D37      push    eax
DATA:00DE3D38      retn
```

The int 3 opcode is 0xCC and is known by Reverse Engineers. In order to make a little less obvious, i have obfuscated the breakpoint check using a "SHR" (Shift Right) instruction: 0x660 shr 3 = 0xCC ;-). The program will then check four bytes at API function entry point, looking for a breakpoint. If a breakpoint is found, i have used a funny way to crash the application. Im using RDTSC to generate a pseudo random number and i put this number onto the stack. To modify EIP, i simply use the RET instruction, which will transfer us to random memory address, crashing our application. Each time a detection occurs, the address is different, thus hard to monitor. The crash occurs far from the detection code and Soft ICE's FAULT ON won't catch it either.

Protection Weakness:

First, the Imports aren't protected, therefore anyone can read the Imported functions from the binary. From The import table we can see that printf, GetCommandLineA and ExitProcess are used. This is a weakness. A Reverse Engineer can put breakpoints on those functions, or at least, guess they are going to be used at some point. In the case of our binary, one can guess that the application is waiting for a special command line. A solution would be to load the Import Table manually.

For this we could use a home made GetProcAddress function to browse the Export Table of the dlls we want to import functions from, and then, get the address of the API function from there. A Kernel32 address is always on the stack when a binary is started, so we could have used this value to get the dll's ImageBase (Or use the PEB, SEH chaining etc..). We would have everything needed to get the address of Loadlibrary which allows us to Load ANY dll, and thus, to get the address of ANY API function. With this method, we don't need any Import Table at all.

Well actually, this isn't true. There is a mandatory thing to do to keep compatibility with all versions of Windows. We have to create a very small Import Table, with at least ONE import from Kernel32, else the binary won't run on Windows 2000. The Windows 2000 PE Loader is different from the one in Windows XP. XP doesn't care whether there is any import table or not.

The small Import Table is just for compatibility issue, the real import table is encrypted and will be decrypted at runtime by the protection.

Then, it is just a matter of loading the Imports mimicing the Operating System. We need to put the API address in the Import Address Table (of the decrypted Import Table) manually. The Reverse Engineer has no clue about the API functions used by the binary until he gets to the part of the code that will decrypt and load the Imports.

The BPX protection has a few weaknesses. I only check for four bytes at API entry point, which can be easily bypassed, if the API has many instructions. One could put a breakpoint to the first instruction after the 4 bytes boundary.A Better check would use a Length Disassembler Engine (LDE) which tells us the size of the instructions. With this, we can safely scan a lot of instructions without triggering any false positive.

A genuine instruction can contain the byte 0xCC and yet not beeing a breakpoint. Eg: Mov eax, 0x4010CC. The detection would trigger a false positive on this instruction, because of the 0xCC inside of it. On the other hand, a LDE would tell us the size of this instruction (5 Bytes). An int 3 (breakpoint) is either one or two bytes (0xCC or 0xCD 0x03). We would therefore skip the current instruction and check the following one.

Also, the BPX check is only done once per API at a given location in the binary.Once we have stepped over those checks , we can put a breakpoint on any API function without triggering any error. This weakness wasn't fixed on purpose because this is a common error in Protection Systems.

There is another kind of BPX detection that will be described in the next section

- **The Crazy Layers**

Here is a little more challenging protection. In order to protect the binary from beeing disassembled, i have written an Encryption Layer generator, that will generate the number of layers i want. For this binary, i used 175 layers. The Layer Generator has many options. Here are the options from the config file: (0 means disabled)

```
SEH=1
RANDOM_LAYER_SIZE=0
RANDOM_REGISTERS=1
RANDOM_ENCRYPTION=0
ENCRYPTED_RETURN_ADDRESS=1
TIMING_DETECTION=1
RANDOM_CONSTANT=0
JUNKS=0
PUSHAD_POPAD=1
RANDOM_ORDER=0
USE_DIFFERENT_LOOP_CODE=0
RANDOM_FIRST_BLOCK=0
NUMBER=175
```

I will comment each options below:

SEH:

This tells to my layer generator to use (or not) SEH inside the layers.

RANDOM LAYER SIZE:

This tells to my layer generator to use a different size for each layer. This option wasn't enabled to simplify the analysis.

RANDOM REGISTERS:

If this option is enabled, all the layers are using different registers. Some kind of "polymorphism". This option was enabled.

RANDOM ENCRYPTION:

When this option is enabled, Each layer will have a different encryption algorithm. I didn't enable this option. Therefore all the layers have a static encryption code. (Default layer)

ENCRYPTED RETURN ADDRESS

This option will encrypt the return address inside the layer. It avoids a simple patch to skip the SEH.This option was enabled

TIMING_DETECTION:

Tells whether the layers must use Timing Detection or not. I enabled this option.

RANDOM_CONSTANT:

The Random constant is to tell whether we want to use a static value for the timing detection or not. This option wasn't enable. All layers were using the defaut value: E0000h. Enabling this option will also modify the code that checks for the Difference between both TSC.

JUNKS:

Enable of Disable Junks in the Layers. I disabled this option because the layers are WAY biggers when it is enabled. The resulting binary is too huge and slow if you use a big number of layers.

PUSHAD_POPAD:

This option tells the Layer Generator to use (or not) Pushad/Popad around the Junk Code. The layer generator directly use the Thrash Generator (external tool) i have programmed. I was using pushad popad in the junk code, that's why it is enabled. This option does nothing if the Junks option is disabled.

RANDOM_ORDER:

Each layer use a table to access part of its code. If this option is enabled, Each layer has a random order of execution. I didn't enable this one on purpose.

USE_DIFFERENT_LOOP_CODE:

Each layer loops a given number of time. With this option, one can use different code to test the end of the loop. It makes it harder for the reverse engineer to find removal pattern. This option wasn't enabled. A defaut checking code was used.

RANDOM_FIRST_BLOCK:

This option allows one to use random value inside the first elements of the layers tables. You will see in some submissions that the static value were used to bypass the layers. I didn't enable this option to see whether someone was going to use it or not.

NUMBER:

This is the number of the layer, the generator must use. I used 175 layers in this challenge. I can generate 65000 layers in a few seconds because the generator engine is programmed in Assembly Language.

Presentation of the encryption layers:

Layer Selector

```
xor esi,esi                                   ; ESI = 0
mad_loop175_1:                                ; Loop label
inc esi                                       ; ESI++
mov edi,dword ptr [ebp+(esi*4)+EIPtable175_1] ; Grab block address
mov ebx,dword ptr [ebp+(esi*4)+RETable175_1]  ; Grab "Encrypted"
                                              ; Return address

Add ebx, [ebp+_startloader]                   ; Add Base.
push ebx                                      ; Save Return Address
                                              ; from the stack
Call tricky_call175_1                         ; Fake call
db 0EBh,01,0E8h                               ; Some junk crap

fake_ret175_1:                                ; fake return address label.
Add edi, [ebp+_startloader]                   ; Add EDI Base. EDI now
                                              ; contains address of a block
                                              ; inside the layer.
jmp edi                                       ; Execute that block.

return_addy175_1:
cmp esi, 4                                    ; When we get back from the
                                              ;block, we check whether we
                                              ;have done every blocks.
jnz mad_loop175_1                             ; if we didn't, loop!

bpxcheck175_1:                                ; Label used for BPX check.
jmp @layer175_1

tricky_call175_1:
pop ebx                                       ; Ret address is in EBX

jmp fake_ret175_1                             ; Jmp to fake return address.
@layer175_1:                                  ; end of the layer.
```

This is the main part of a layer. This part loops through the layer blocks using some obfuscated ways. It prepares the stack with return addresses, and fake a call. If you step over with your debugger on this call, the binary won't break and it will run. If you were debugging a malware, you would get infected. And if you were analysing the binary, you would need to restart from scatch. (Except if you have dumped your position regulary).

Layers Blocks

```
include   obfuscation/junk198.inc          ; I have added a few junks macro
                                           ; manually in order to add a
                                           ; little fun :)

dec_loader175_1:                           ; Decrypt label
xor byte ptr [edx],cl                      ; Defaut options were used.
                                           ; Very simple encryption.
inc edx                                    ; Code to decrypt++
dec ecx                                    ; Loop index--
test ecx, ecx                              ; is ECX = 0 ?
jnz dec_loader175_1                        ; no :( therefore we continue
                                           ; to decrypt.

; This encryption can be different for each layer if you enable the option in
the layer Generator.

lea edx , [ebp+bpxcheck175_1]              ; Grab address of BPX check.
cmp byte ptr [edx],0CCh                    ; Any break point ?
jnz return175_1                            ; no. Good boy.

rdtsc                                      ; Ah.. he did put a bpx..
                                           ; EAX = random value
push eax                                   ; push eax on stack
ret                                        ; Return to it :) Crash the
                                           ; poor guy.

return175_1:                               ; on return block
include   obfuscation/junk199.inc          ; a few junk
include   obfuscation/junk19A.inc          ; ditto.
SEHBLOCK 66137317 28513829                 ; SEH block macro with
                                           ; keys in parameters.
ret                                        ; return

inst175_2_1:                               ; another block of code
add dword ptr [esp], 41952561              ; fix return address and return.
ret

inst175_3_1:                               ; Another block.
mov ecx, (offset _end174_1- @layer175_1)   ; Get Size of layer
add dword ptr [esp], 13007360              ; Fix return address and return
ret

inst175_1_1:                               ; Another block
lea edx, [ebp+@layer175_1]                 ; Get Layer address
add dword ptr [esp], 30560857              ; Fix the return address
                                           ; and return.
ret

EIPtable175_1 dd 000DEADh, (offset inst175_1_1 - offset startloader), (offset
inst175_2_1 - offset startloader), (offset inst175_3_1 - offset startloader),
(offset _end174_1 - offset startloader)
```

```
; This is a table of offset used to redirect the code.

RETable175_1 dd 0031000h, (offset return_addy175_1 - offset startloader -
30560857) , (offset return_addy175_1 - offset startloader - 41952561),(offset
return_addy175_1 - offset startloader - 13007360),(offset return_addy175_1 -
offset startloader - 37623488)

; This is a table of return address with a little "encryption".

; You can notice the first member of the tables : DEADh and 31000h. Those
values are constants and can be random using the RANDOM_FIRST_BLOCK
; option in the layer generator.
```

The layer presented above has been generated by the little Layer generator Engine i have programmed. I have added comments for the readers.

Protection Weakness:

Those layers have a few weaknesses. You can use BPM (Hardware Break Point) on the next layer once you have passed the SEH that is going to clear the debug registers. Another weakness is the static size of the layer. Using this information, one can pass the layers rather quickly with a few Soft ICE macros for instance. I didn't turn the random size option on, on purpose to allow such attacks.

Those layers always use the same encryption algo, which can allow one to write scripts to decrypt the binary. And as you can read in a few submissions, some people did it. I did put this weakness on purpose as well. In a challenge i had done in the past, i had used random encryption for each layers, this time i choose not to use it. It is possible to bypass the 175 layers in a few seconds easily as well using a live approach. As we know wich API functions are going to be used, we can set a break point after the BPX checks have occured.Another possibility is to create a little utility that will PATCH the system dll in memory (each application has a copy of the dll) and to redirect them to a place that you contol. This way you can put breakpoints without triggering any Detection code.

Talking of patching the Windows dll files, it is possible to patch ntdll to avoid the Debug Registers access in the context structure, by hooking the Exception Handling Mechanism of Windows. This allows one to put Hardware Breakpoints anywhere without ever having problems, never seeing his debug breakpoints beeing erased etc. The cool thing is you don't even need a Kernel Mode Driver to do that. I leave this as an exercice for interested people.

- **Virtual Machine**

The final protection of the binary is a complete Virtual Machine i wrote for the challenge. I have designed a Virtual CPU that will interpret my own Assembly language. The Virtual Machine is quite simple to understand and isn't very complex.

Virtual Machines seem to be a new trend in protection systems, so i thought it could be a good thing to write one for such a challenge. The instruction encoding is very trivial, and could have been a lot harder to understand. The first Version i had in mind was a lot more complex. I wanted not only to have a pseudo language, but also to program the instructions handlers emulating real x86 instructions. Each handler would be a few hundred instructions long and a lot harder to analyse.

A small program has been written with this Virtual Machine Assembly language, and it was used to authenticate the user running the binary.

Read next part for further informations

3. Something uncommon has been used to protect the code from beeing reverse engineered, can you identificate what it is and how it works?

Even though, a few protection systems are using some kind of Virtual Machines, those aren't very common. Especially in Malwares and other exploits.

Virtual CPU description and Inner working:

Registers:

```
REGISTERS STRUC
        R0_             dd      ?  ; 000
        R1_             dd      ?  ; 001
        R2_             dd      ?  ; 002
        COUNTER_ dd     ?  ; 003
        EIP_            dd      ?  ; 004 -> reserved
        STATE_          dd      ?  ; 005
REGISTERS ENDS
```

This is the original structure from my code source. Every registers is a DWORD. Some registers weren't used because they are reserved for futur version of the Virtual Machine. One can read "EIP_". I planned to add another information per instruction, but i didn't do it, because i didn't want it to be too complex. I will add the ability to change the Instruction pointer for any instruction. The result will be a completely mad code flow. The instruction order in the file will have nothing to do with the real execution flow.

The *STATE* Register is some kind of mini Eflags. This register changes depending of other instructions.

The *COUNTER* Register is used for loop instructions. Similar to the ECX register when we use the LOOP instruction.

```
regs    REGISTERS       <>

R0      equ     000b
R1      equ     001b
R2      equ     010b
COUNTER equ     011b
EIP     equ     100b
STATE   equ     101b
```

Here are a few other definitions used in my program.I started to represent the registers in binary because i wanted to do complex opcode decoding. I will do that for another version ;)

Registers Initialisation:

```
mov      dword ptr [regs.R0_],"livE"       ; Registers are initialized with a
                                           ; Slayer Song Title.
..
mov      dword ptr [regs.R1_],"saH "       ; Evil Has No Boundaries!
..
mov      dword ptr [regs.R2_]," oN "
..
mov      dword ptr [regs.COUNTER_],"nuoB"
..
mov      dword ptr [regs.EIP_],"irad"
..
mov      dword ptr [regs.STATE_],"! se"
```

At the start of the VM, i first begin to initialise my own registers with the song's title of a thrash metal band. This title was selected because i planned to do real evil things with the Virtual Machine. It isn't as hard as the initial version, but still evil enough to keep that funny string ;-).

There is about 34 Instructions in the Virtual Machine. (I count instructions having different utilisation as unique)

I will present a few instruction handlers to explain the inner working of the Virtual Machine, but not every instruction will be presented here.

Pcode Fectcher:

The first thing the Virtual Machine does after the Register Init is to Fetch the Pcode entry point and jmp to the first Pcode handler.

```
movzx    eax, byte ptr [esi]               ; ESI is Pcode Entry Point. This code
                                           ; gets the first instruction Prefix.

mov      edi, dword ptr [eax*4+poffset]    ; It uses it with the offset table to
                                           ; find the Pcode family it has to
                                           ; execute.

movzx    eax, byte ptr [esi+1]             ; get second byte, use it as an
                                           ; index into last table.
                                           ; The VM now knows what instruction it
                                           ; has to emulate and goes to it.

JMPNEXT                                    ; Emulate a jmp  dword ptr [eax*4+edi]
                                           ; with Exception Handling and
                                           ; Context Manipulation.
                                           ; Jmp to the next Pcode
                                           ; instruction handler
```

Examples of Instructions implemented inside the Virtual Machine:

Before i start with those examples, i would like to say that a few instructions present in the Virtual Machine weren't used and were left as decoy.Three of them are using Self modifying code. People are reporting that they don't work, but they should. The off by one difference is because the opcode is beeing called from other instruction handlers. Two instructions are modifying one instruction on the fly as they need to execute a particular piece of code. They then restore the instruction state. I am too lazy to check whether those instructions are really bugged or if they didn't use the good parameters. One of the _unused_ instruction HAS a bug, and i am glad some people noticed it. The instruction isn't used therefore, it is just a decoy instruction. The instruction is supposed to be a Virtual BSWAP, but it doesn't save the result of the swaping. Another unused instruction is the INT 3. This instruction allows one to put breakpoint in his Pcode program and trace with his debugger from that instruction. I left this instruction in the final Virtual Machine and im glad some people found it and abused it!.

STOPVM

The first instruction i will present here is a very simple one. It tells to the Virtual Machine to stops and the program will get back to normal x86 assembly program.

```
@STOPVM:
        pop     dword ptr fs:[0]        ; Im using SEH to jmp from handlers to
                                        ;handlers in the VM.
                add     esp,4           ; Therefore i need to remove the handler
                                        ; installed before i do anything.
                popad                   ; i restore the registers..
                ..
                push    dword ptr [Pret] ; Put the Return Address (to get out of
                                        ; the VM) on the stack.
                ..
                xor     [esp],'HAX0     '; Decrypt it with a funny string:
                                        ;HAXO(R)
                ..
                ret                     ; Get out of the VM.
```

This instruction is not using any bytecode fetcher because it doesn't need to jmp to another handler. I will now present a real instruction. A Virtual PUSH:

LOAD

```
@Load:
        pop  dword ptr fs:[0]
        add  esp,4
        popad

        ; Same as every handler, remove SEH and restore registers.

        mov     eax,dword ptr [esi+2]   ; Get into EAX the first operand
                                        ; of the instruction.

        xor     eax,37195411h           ; Decrypt it.
        push    eax                     ; Push it onto the stack.
        mov     eax,0FFFFFF3Fh          ; EAX = FFFFFF3Fh
        not     eax                     ; EAX = not(EAX) = C0h
        shr     eax,5                   ; EAX = EAX shr 5 = 6 :
                                        ; This is the instruction length
        lea     esi, [esi+eax]          ; ESI = Instruction Pointer.
                                        ; Deplace the Instruction Pointer
                                        ; 6 bytes further.
        movzx   eax, byte ptr [esi]     ; ESI now points to the new
                                        ; instruction to be executed.
        mov     edi, dword ptr [eax*4+poffset]    ; It uses it with the offset
                                                  ; table to find the Pcode family
                                                  ;it has to execute.

        movzx   eax, byte ptr [esi+1]   ; get second byte, use it as an
                                        ; index into last table.
                                        ; The VM now knows what instruction it
```

```
                                          ; has to emulate and goes to it.

        JMPNEXT                           ; Emulate a jmp  dword ptr [eax*4+edi]
                                          ; with Exception Handling and
                                          ; Context Manipulation.
                                          ; Jmp to the next Pcode
                                          ; instruction handler
```

As you can see from this little handler, the instruction is 6 bytes long. It takes only one parameter and it is placed 2 bytes after the start of the instruction. (ESI+2). The parameter is encrypted with 37195411h. The decrypted parameter is pushed on the stack and then the Virtual Machine calls the next instruction.

From this, we can say that this instruction is a push. since push is already a x86 instruction, i named my virtual push : LOAD.

One can use it like this: "LOAD number"

VMXOR

```
@VMXORDISPATCHER:
        pop     dword ptr fs:[0]
        add     esp,4
        popad
; Same as every handler, remove SEH and restore registers.

        movzx   eax, byte ptr [esi+2]           ; Get the Index Register to acces
                                                ; the Virtual CPU registers.
        mov     eax, dword ptr [regs+eax*4]     ; edi = Register value to know
                                                ; which register is going to be
                                                ; concerned (RO, R1 , R2)
                                                ; EAX = value used by the XOR.

        movzx   ecx, byte ptr [esi+3]           ; ECX = type of XOR.
                                                ; Byte ptr ? Word Ptr ? or
                                                ; Dword Ptr..

        jmp     dword ptr [xortable+ecx*4]      ; Jmp to the good handler
                                                ; accordingly.

@VMXORBPTR:

        movzx   ecx, byte ptr [esi+4]           ; Get Index Register for the
                                                ; destination.

        mov     ecx, dword ptr [regs+ecx*4]     ; edi = Register value to know
                                                ; which register is going to be
                                                ; used (RO, R1 , R2)

        xor     byte ptr [ecx],al               ; XOR BYTE PTR

        add     esi,5                           ; Instruction Length is 5

        movzx   eax, byte ptr [esi]             ; ESI now points to the new
                                                ; instruction to be executed.

        mov     edi, dword ptr [eax*4+poffset]  ; It uses it with the offset
                                                ; table to find the Pcode family
                                                ; it has to execute.
        movzx   eax, byte ptr [esi+1]             ; get second byte, use it as an
                                                ; index into last table.
                                                ; The VM now knows what
                                                ; instruction it has to emulate
                                                ; and goes to it.

        JMPNEXT                                   ; Emulate a jmp  dword ptr
                                                ; [eax*4+edi] with Exception
                                                ; Handling and Context
                                                ; Manipulation.
                                                ; Jmp to the next Pcode
                                                ; instruction handler


@VMXORWPTR:

        movzx   ecx, byte ptr [esi+4]             ; Get Index Register for the
                                                ; destination

        mov     ecx, dword ptr [regs+ecx*4]       ; edi = Register value to know
                                                ; which register is going to be
                                                ; used (RO, R1 , R2)

        xor     word ptr [ecx],ax                 ; XOR WORD PTR

        add     esi,5                             ; Instruction Length is 5

        movzx   eax, byte ptr [esi]               ; ESI now points to the new
instruction to be executed.

        mov     edi, dword ptr [eax*4+poffset]    ; It uses it with the offset
                                                ; table to find the Pcode family
                                                ; it has to execute.

        movzx   eax, byte ptr [esi+1]             ; get second byte, use it as an
                                                ; index into last table.
                                                ; The VM now knows what
                                                ; instruction it has to emulate
                                                ; and goes to it.

        JMPNEXT                                   ; Emulate a jmp  dword ptr
                                                ; [eax*4+edi] with Exception
                                                ; Handling and Context
                                                ; Manipulation.
                                                ; Jmp to the next Pcode
                                                ; instruction handler

@VMXORDPTR:

        movzx   ecx, byte ptr [esi+4]             ; Get Index Register for the
                                                ; destination

        mov     ecx, dword ptr [regs+ecx*4]       ; edi = Register value to know
                                                ; wich register is going to be
                                                ; used (RO, R1 , R2)

        xor     dword ptr [ecx],eax               ; XOR DWORD PTR

        add     esi,5                             ; Instruction Length is 5

        movzx   eax, byte ptr [esi]               ; ESI now points to the new
                                                ; instruction to be executed.

        mov     edi, dword ptr [eax*4+poffset]    ; It uses it with the offset
                                                ; table to find the Pcode family
                                                ; it has to execute.
        movzx   eax, byte ptr [esi+1]           ; get second byte, use it as an
                                                ; index into last table.
                                                ; The VM now knows what
                                                ; instruction it has to emulate
                                                ; and goes to it.

        JMPNEXT                                 ; Emulate a jmp  dword ptr
                                                ; [eax*4+edi] with Exception
                                                ; Handling and Context
                                                ; Manipulation.
                                                ; Jmp to the next Pcode
                                                ; instruction handler
```

From this piece of code we can learn many things. The XOR instructions are coded with 5 Bytes. It has two parameters.One register has the value used to do the XOR and One Register has a pointer to the location to be xored.It also have a byte saying whether it is a XOR BYTE PTR, a XOR WORD PTR or a XOR DWORD PTR.

This instruction handler is therefore handling Virtual XOR instruction.

How were the virtual instruction used to create a program ?

I will now show how i did to create virtual instruction, because the x86 assembler doesn't know them and will never compile a LOAD or a VMXOR.To do so, i used a very simple way: MACRO. For each instruction, a corresponding macro has been created, and is used to encode the instruction for me. This way i can write a program with my Assembly mnemonics without caring of the opcodes representation.I will now show the 3 Macros used for the examples Virtual Instruction descrived above

```
STOPVM macro
        db      02,00
endm
```

This is the macro for the STOPVM instruction. Usage: STOPVM

```
Load macro x
        db      00,00
        dd      x xor 37195411h
endm
```

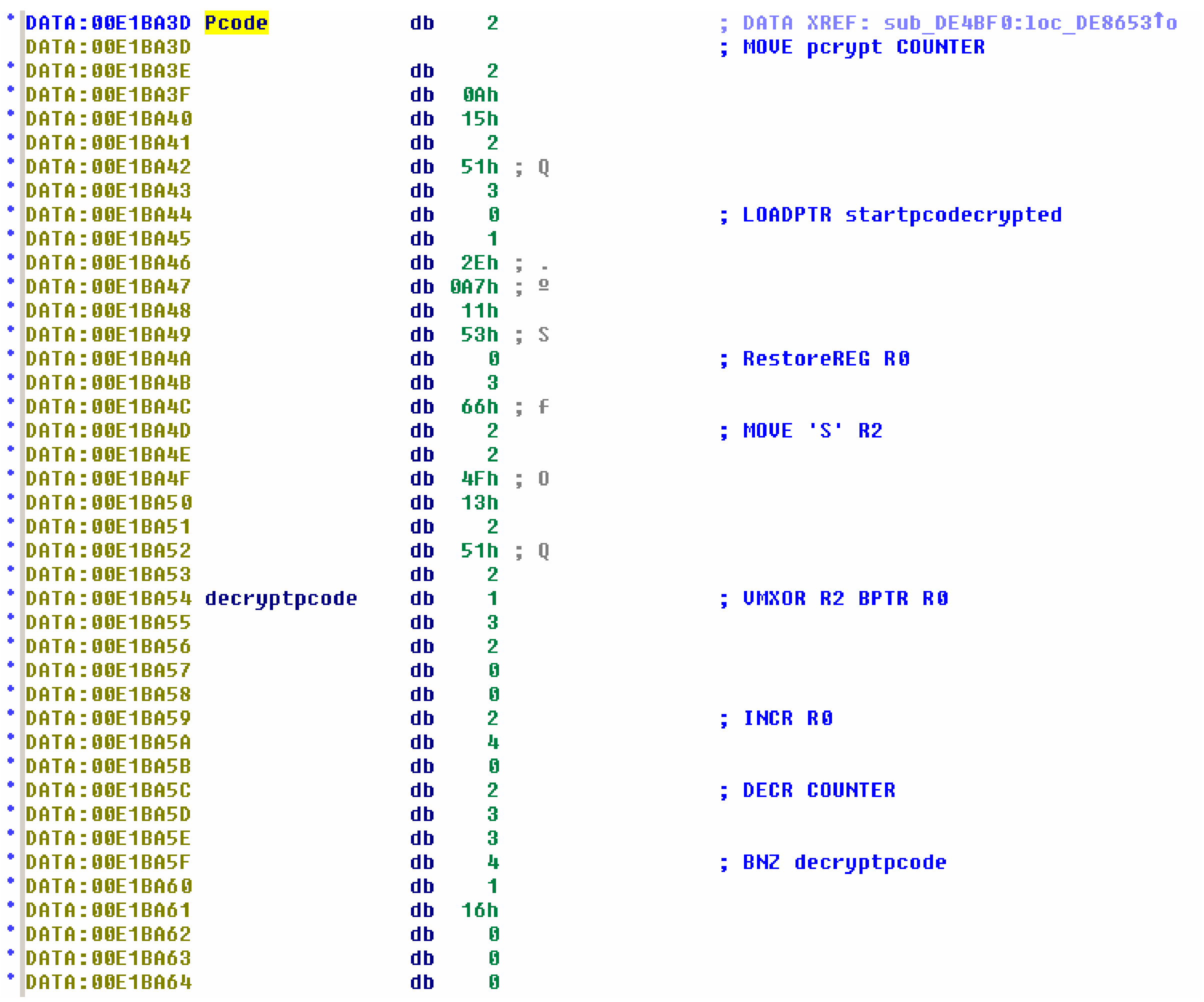
This is the macro for the LOAD instruction. Usage: LOAD x

```
VMXOR macro  reg0,kind,reg1
        db 01,03,reg0,kind,reg1
endm
```

This is the macro for the VMXOR instruction. Usage: VMXOR Rx xPTR Rx

P-code Program used in this challenge:

The P-code is my own assembly language, thus IDA doesn't know anything about it. Here is how it looks under a disassembler:



```
DATA:00E1BA3D Pcode          db    2                ; DATA XREF: sub_DE4BF0:loc_DE8653↑o
DATA:00E1BA3D                                       ; MOVE pcrypt COUNTER
DATA:00E1BA3E                db    2
DATA:00E1BA3F                db  0Ah
DATA:00E1BA40                db  15h
DATA:00E1BA41                db    2
DATA:00E1BA42                db  51h ; Q
DATA:00E1BA43                db    3
DATA:00E1BA44                db    0                ; LOADPTR startpcodecrypted
DATA:00E1BA45                db    1
DATA:00E1BA46                db  2Eh ; .
DATA:00E1BA47                db 0A7h ; º
DATA:00E1BA48                db  11h
DATA:00E1BA49                db  53h ; S
DATA:00E1BA4A                db    0                ; RestoreREG R0
DATA:00E1BA4B                db    3
DATA:00E1BA4C                db  66h ; f
DATA:00E1BA4D                db    2                ; MOVE 'S' R2
DATA:00E1BA4E                db    2
DATA:00E1BA4F                db  4Fh ; O
DATA:00E1BA50                db  13h
DATA:00E1BA51                db    2
DATA:00E1BA52                db  51h ; Q
DATA:00E1BA53                db    2
DATA:00E1BA54 decryptpcode   db    1                ; VMXOR R2 BPTR R0
DATA:00E1BA55                db    3
DATA:00E1BA56                db    2
DATA:00E1BA57                db    0
DATA:00E1BA58                db    0
DATA:00E1BA59                db    2                ; INCR R0
DATA:00E1BA5A                db    4
DATA:00E1BA5B                db    0
DATA:00E1BA5C                db    2                ; DECR COUNTER
DATA:00E1BA5D                db    3
DATA:00E1BA5E                db    3
DATA:00E1BA5F                db    4                ; BNZ decryptpcode
DATA:00E1BA60                db    1
DATA:00E1BA61                db  16h
DATA:00E1BA62                db    0
DATA:00E1BA63                db    0
DATA:00E1BA64                db    0
```

Ok, it doesn't look so good. So now, here is the complete program (copy pasted from my source) i have written with my OWN assembly language. Cool isn't it ? :-)

```
CODE@:

Pcode1:

        MOVE pcrypt COUNTER
        LOADPTR startpcodecrypted
        RestoreREG R0
        MOVE 'S' R2

decryptpcode:

        VMXOR R2 BPTR R0
        INCR R0
        DECR COUNTER
        BNZ decryptpcode
```

```
startpcodecrypted:

        MOVE 05CC80E31h R1
        APICALL GetCommandLineA
        SCANB " " 255h
        BZ youwishdude
        DLOAD R0 R2

        ADDREG R2 01D9BDC45h
        ADDREG R1 74519745h
        SUBREG R2 0AD45DFE2h
        ADDREG R1 0DEADBEEFh
        ADDREG R2 "hell"
        SUBREG R1 17854165h
        SUBREG R2 "Awai"
        ADDREG R1 "show"
        ADDREG R2 "its "
        SUBREG R1 " no "
        ADDREG R2 "driv"
        ADDREG R1 "merc"
        SUBREG R2 "nuts"
        SUBREG R1 "y!!!"
        SUBREG R2 "eh?!"
        ANDREG R2 0DFFFFFFFh

        LOADREG R2
        LOADREG R1
        CMPQ firstcheckdone
        CLEAR COUNTER
        BZ youwishdude

firstcheckdone:

        INCR R0
        ADDREG R0 2
        INCR R0
        WLOAD R0 R1

        LOADREG R1

        RESTOREREG R2
        LOADREG R0
        LOADPTR tricky-98547h
        RESTOREREG R0
        ADDREG R0 98548h
        DECR R0

        VMXOR R2 WPTR R0

        RESTOREREG R0

        BLOAD R0 R2
        ADDREG R0 2

        BLOAD R0 R1
        RADD R2 R1
        VMCALL sub_check_routine

tricky:

        ENCRYPTEDCLEAR COUNTER            ; This one get patched at run time!
```

```
cracked:
        LOADREG COUNTER
        LOADPTR congrats
        APICALL printf
        CleanStack 8
        BR outout

youwishdude:

        LOAD 0
        LOADPTR notgood
        APICALL printf
        CleanStack 8
outout:
        STOPVM


sub_check_routine:

        MOVE 'L' R1
        INCR R1
        INCR R1
        ADDREG R1 5
        DECR R1
        SUBREG R1 4
        SUBREG R2 'Z'
        CMPREG R1 R2
        BNZ youwishdude

        DECR R0
        BLOAD R0 R2
        ADDREG R0 2
        BLOAD R0 R1
        RADD R2 R1
        INCR R2
        SUBREG R2 4Eh

        LOADPTR retdecrypt-0DEADh        ; push ptr to patch
        RESTOREREG R0
        ADDREG R0 0DEACh
        INCR R0
        VMXOR R2 BPTR R0

        MOVE msgcrypt COUNTER
        LOADPTR goodboy
        RestoreREG R0
        INCR R2

decryptmsg:

        VMXOR R2 BPTR R0
        INCR R0
        DECR COUNTER
        BNZ decryptmsg

retdecrypt:

        VMRETCRYPTED
```

```
DATA@:

notgood          db      "Please Authenticate!",10,13,0


goodboy:
congrats db      "Welcome...",10,13
                 db      "Exploit for it doesn't matter 1.x Courtesy of Nicolas
Brulez",0
goodboyend:
```

This little routine is the password protection used in the binary. For more informations about it, i will let you read the submissions. As you can see, the password protection is VERY SHORT. I could have written a very complex algo with hundreds of lines to make it harder to analyse. Also this code is clear of junk. I could also have placed P-code Junk instructions inside the program. The password check was very simple and sadly, some people concentrated on the password rather than the Virtual Machine. Next time i will make it a lot more complex so people has no choice but to analyse the Virtual Machine and Instruction set.

You can compare the original P-code program here with the one inside the submissions to realize that they have done a very good job.

A Few notes regarding the Password Protection

The password is checked using a very simple algorithm, but it is also used to decrypt yet another part of the pcode program. There is a little weakness allowing one to find the correct value without any brute forcing or analysis of the Opcodes:

Here is the encrypted String :

```
0x14, 0x26, 0x2F, 0x20, 0x2C, 0x2E, 0x26, 0x6D,
0x6D, 0x6D, 0x49, 0x4E, 0x06, 0x3B, 0x33, 0x2F,
0x2C, 0x2A, 0x37, 0x63, 0x25, 0x2C, 0x31, 0x63,
0x2A, 0x37, 0x63, 0x27, 0x2C, 0x26, 0x30, 0x2D,
0x64, 0x37, 0x63, 0x2E, 0x22, 0x37, 0x37, 0x26,
0x31, 0x63, 0x72, 0x6D, 0x3B, 0x63, 0x00, 0x2C,
0x36, 0x31, 0x37, 0x26, 0x30, 0x3A, 0x63, 0x2C,
0x25, 0x63, 0x0D, 0x2A, 0x20, 0x2C, 0x2F, 0x22,
0x30, 0x63, 0x01, 0x31, 0x36, 0x2F, 0x26, 0x39,
0x43
```

Everyone knows that a C String ends with a null byte. Therefore, the value used to encrypt this string is 0x43. The key is the last byte of the encrypted string. X xor 0 = X. :-)

The other possible ways to find the good value was to look at the code structure.. We were doing a Call routine, therefore we must have an instruction to do a RET. This instruction is the Virtual RET implemented in the Virtual Machine. From this, we just had to find the opcode of this instruction to compute the key.

4. Provide a mean to "quickly" analyse this uncommon feature.

With this question, i was expecting a disassembler for my Virtual Machine. A few people sent me fully working disassemblers, so i didn't write yet another one. I invite you once again to have a look at their submissions.One of the author emailed me after the deadline with a working IDA processor module with source code included. This Processor module wasn't used in my judgement because it was sent after the deadline, but it is well worth studying it. It will be uploaded on the honeynet web site shortly after the publication of the Results.

5. Which tools are the most suited for analysing such binaries, and why?

In my opinion the best tools to analyse such binaries are Interactive Disassemblers or CPU emulators. The disassembler can be used to analyse the code statically, to remove the obfuscations, to decrypt binaries etc. If it offers possibility to write processor module, you can even write a disassembler for the Virtual Machine and thus, do a full static analysis of the whole thing. A CPU emulator can be used to quickly decrypt the code , layers etc. If it can be scripted not to show the obfuscations you have a perfect weapon. I don't like Debuggers because they aren't reliable. I could have easily written a driver to hook debug interupts to decrypt the binary for instance. Debuggers would have been useless and would have rebooted the computer if used.

6. Identify the purpose (fictitious or not) of the binary.

This binary is waiting for an user to authenticate with a password that is passed to the application through the command line. Once the user has been identified, the binary will print a little message. It looks like a fake exploit. In the real world, it could have been a real exploit protected from prying eyes.

7. What is the binary waiting from the user? Please detail how you found it.

The binary is waiting for a password through the command line. The password is used to access the real program. To find this password, you have to Reverse Engineer the binary. Decrypt every layers to access the Virtual Machine. This Virtual Machine has a virtual program used to check the password entered. One has to Reverse Engineer the Virtual Machine (or trace it blindly) in order to understand its instruction set. Then it is just a very simple algo using a few easy operations to reverse. I invite you to read submissions for details about the algo itself.

8. Bonus Question - What techniques or methods can you think of that would make the binary harder to reverse engineer?

This binary has a lot of security flaws that were left on purpose and it has a lot of things needing improvements.

- Junk Code without pushad/popad
- P-code Junk code to really drive people tracing the protection nuts: 80% of useless instructions would definitevely drive anyone mad.
- The encryption has a few weaknesses that were presented in the document. Mainly the static encryption algo and the static size of every layers.
- Random constants in the first value of the address tables used by the Layers
- Better BPX detection, it could be greatly improved.
- The SEH could be used to initialise/decrypt part of the code in order to make sure they won't be nopped out.
- Random Constants and code for the timing detection would make it harder to bypass with scripts.
- The protection has been generated by my own tools and it is a bit repetitive. More variations would make automatic removal harder.
- The Virtual Machine handlers are fairly simples. More code obfuscation (code flow and logic) could be used.
- More Instructions in the Virtual Machine would have made it longer to analyse.
- Complex Opcodes encoding would make it quite challenging to Reverse Engineer.
- Utilisation of Cryptography rather than a simple algo to check for the serial number
- More Layers and different ones. There are a lot of ways to stop ring 3 debuggers that could have been used to stop anyone trying to debug it.
- Imports Protection to make sure noone knows the API function used until he meets them in the code.
- Emulation Macros to emulate simple x86 instructions. With those macros remplacing simple instruction, it would be a lot harder to analyse. one instruction would have a 20 instructions equivalent block of code for instance.

9. Conclusion

Anti Reverse Engineering Techniques can be used to really slow down the analysis of a binary. Malwares could be using such techniques in a near futur and it is time to get used to it. Even though most of the malwares are programmed by clueless idiots without any programming skill, there is a minority able to write complex code. In the futur we could find exploit binaries on compromised systems that would be protected against Reverse Engineering to hide the vulnerability exploited. Spywares could also use such techniques to hide their activity. This binary had a lot of vulnerabilities, yet it was really challenging , even with a trivial password protection algorithm. The protection has been written within a week (a few hours per day), so with a little more effort, it can be a LOT harder.Finally, I would like to point out that Reverse Engineering isn't a pirate technique and that it is used by the Security Community on a daily basis. Some people in France doesn't seem to agree though..

Acknowledgements

I would like to thank the following people:

- The Honeynet Project and Lance Spitzner who allowed me to create the challenge.
- The authors of the submissions for taking the time to look at my binary and write a complete report. Thank you.
- People at Datarescue for their Excellent tool IDA Pro used extensively while writing this binary.
- You for reading this document

About the Author

Nicolas Brulez

Chief of Security for Digital River woking on the SoftwarePassport/Armadillo protection system, Nicolas specializes in anti-reverse engineering techniques to defend against software attacks. He has been active in researching viral threats and sharing that research with various anti-virus companies. He regularly writes for the French security magazine MISC and has authored a number of papers on reverse engineering. He currently teaches assembly programming and reverse engineering in French engineering schools.

The author has more than 7 years of Reverse Engineering Experience on Windows Operating Systems and is currently doing Research on Pocket PC devices. He plans to write a Protection system for those devices.