



Corporação Víbora

-----[Keepers of our tradition presents]-----

```
u_char *title[]={
    "[=] + =====[####]===== + [=]\n",
    "    *** Viper Corp Collection Issue 0x02 *** \n",
    "        XxX    Viper Corp Group    XxX\n",
    "            <-> Hacker eagle[s] <->\n",
    "[=] + =====[####]===== + [=]\n\n"};
```

“Don’t spit on my mind“

--- Album: Better than raw - Helloween

```
0x01 <-> Looking for secret messages      0x07 <-> Traps and alterations of code
0x02 <-> Reaching the correct password    0x08 <-> Hie controls
0x03 <-> Understanding the codes          0x09 <-> Solution for defiance one
0x04 <-> Removing nag screens             0x0a <-> Inserting codes
0x05 <-> Creating a PATCHER                0x0b <-> Revert .NET Applications
0x06 <-> UPX unpacking                     0x0c <-> Addresses of memory
```

Extra: Reverse Engineering 13 - Defiance I

This document was originally written by **Fernando Birck** (A.k.a **F3rGO**) []’s in Brazilian Portuguese and after that properly translated and adapted by **David Firmino Siqueira** (A.k.a **6_BI4ck9_f0x6**) []’s This text was written to our older fellow **Wallace Ferreira** (A.k.a **Dark_Side**) []’s, **Jeremy Brown** (A.k.a **Rush**) []’s, **M.:M.:** (<http://www.istf.com.br>) []s **AciDmuD** (A.k.a **AciDmuD**) []’s and **Sandra Julia Firmino** (‘Aka **Little Witch**) [s] Brief comment: “Nois samo foda!!” [<O>]’s



See you there xschooler...

Tutorial - Engenharia Reversa

Parte 1

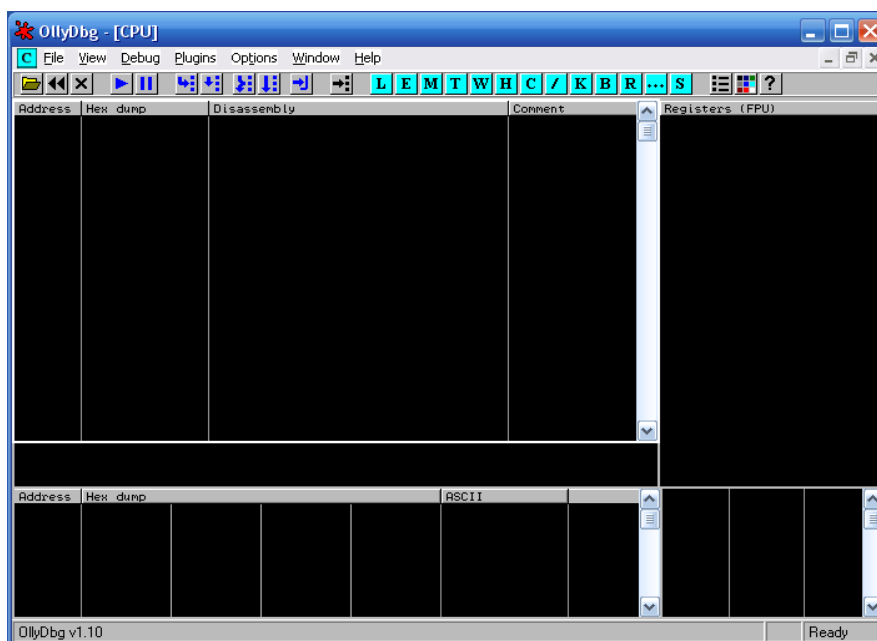


Looking for secret messages

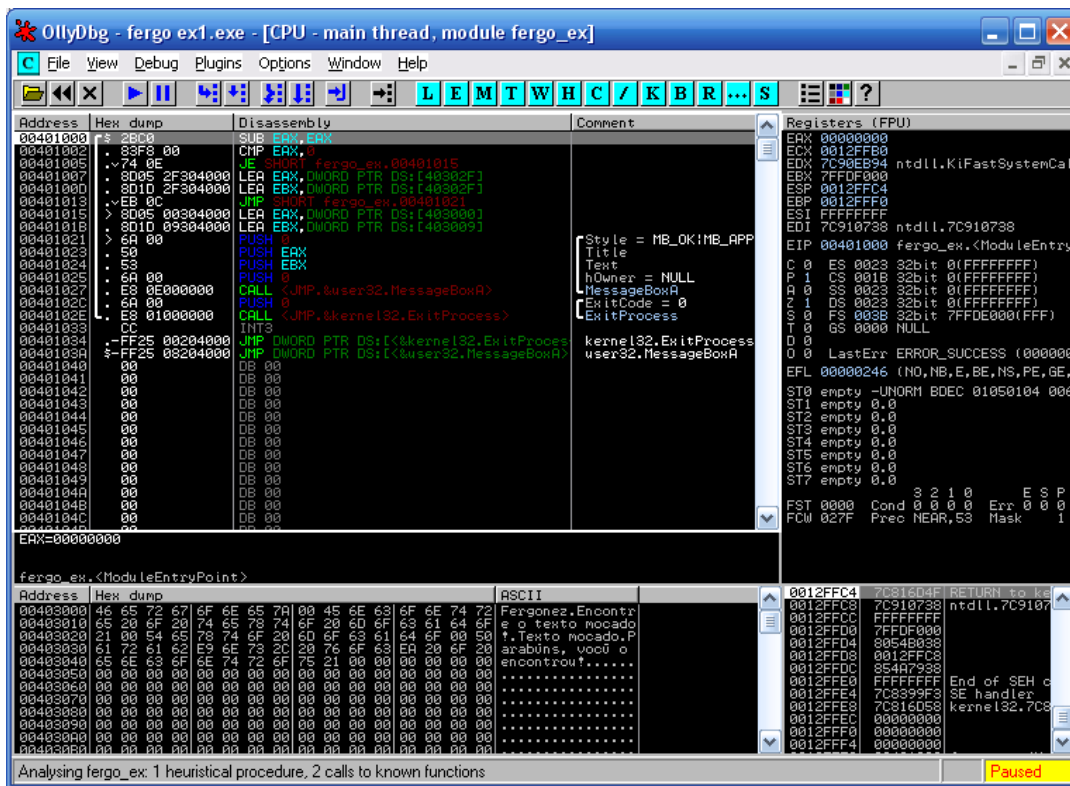
A lot of people think that writing crackers is a hard thing to do, a mistake... this is my first paper about reverse engineering to beginners; this text was written to teach beginner... If you know so much as me (almost nothing ,) this page is a good starting point. Firstly I'll explain some basic things. I will make a manual debug of a program coded in assembly language (your goal will be find the hidden phrase). The program utilized in this chapter is extremely simple, was compiled with MASM32 and written in assembly (of course), as I said. The source code is included with the executable (but don't see it before to finalize this tutorial (my international partner) ,).

-Download: [fergo_ex1.zip](#)

First of all we need to use something to convert our executable in instructions that a human will understand. To do that you need of a program called Debugger or "Disassembler". In this tutorial I'll use one of the best Debuggers, more complete actually (one of the most famous too), by the visual interface and is freeware: [OllyDbg](#) When you start the Olly you'll have a window like this (the colors can be different):



So let us open our executable to analyze the code (in machine language or assembly). Go to "File -> Open" and select "fergo ex1.exe". Instantaneously because of the size of the file it opens. It has few effective lines of code and you'll have something like this shown below:



There are many things no? Don't worry, after you'll understand that. At the main window (to the left), you will have 4 columns: Address, Hex Dump, Disassembly and Comment. Address contains the address of each instruction, using this address you will determine jumps (saltos), calls (chamadas), etc... Hex Dump is the instruction in hex format (that's not important to us now). Disassembly is the same as Hex Dump, but "translated". Comments has not relation to the code, is just a help to identify some things (calls to the functions for example). How the code is little I'll enumerate the lines to start our debugging.

```

01 00401000 2BC0      SUB EAX,EAX
02 00401002 83F8 00   CMP EAX,0
03 00401005 74 0E    JE SHORT fergo_ex.00401015
04 00401007 8D05 25304000 LEA EAX,DWORD PTR DS:[403025]
05 0040100D 8D1D 25304000 LEA EBX,DWORD PTR DS:[403025]
06 00401013 EB 0C    JMP SHORT fergo_ex.00401021
07 00401015 8D05 00304000 LEA EAX,DWORD PTR DS:[403000]
08 0040101B 8D1D 09304000 LEA EBX,DWORD PTR DS:[403009]
09 00401021 6A 00    PUSH 0 ; Style = MB_OK|MB_APPLMODAL
10 00401023 50      PUSH EAX ; Title
11 00401024 53      PUSH EBX ; Text
12 00401025 6A 00    PUSH 0 ; hOwner = NULL

```

```

13 00401027 E8 14000000 CALL <JMP.&user32.MessageBoxA> ; MessageBoxA
14 0040102C 6A 00 PUSH 0 ; ExitCode = 0
15 0040102E E8 01000000 CALL <JMP.&kernel32.ExitProcess> ; ExitProcess
16 00401033 CC INT3
17 00401034 -FF25 00204000 JMP DWORD PTR DS:[&kernel32.ExitProcess] ; kernel32.ExitProcess
18 0040103A -FF25 0C204000 JMP DWORD PTR DS:[&user32.wsprintfA] ; user32.wsprintfA
19 00401040 $-FF25 08204000 JMP DWORD PTR DS:[&user32.MessageBoxA] ; user32.MessageBoxA

```

Line 1: SUB EAX, EAX

The instruction SUB indicates a subtraction operation, followed by two arguments. EAX is a register, a place of temporary data storing which normally are laid values to compare, etc. This command is used to put in the first argument the final result of the subtraction between it and the second one. Something as “EAX = EAX – EAX”. That’s equal to zero (that’s one of the most used forms “to zero” values in ASM).

Line 2: CMP EAX, 0

CMP - “to CoMPare”. This instruction makes an operation between the first and second argument and marks the zero flag (as you will see in a debugger like Immunity) if the operation was successfully solved. In this case it was comparing EAX with 0 (something like “if (eax == 0)” in C language). In the last line, the result zero was thrown to the register EAX and now it is being compared with 0, this operation is true.

Line 3: JE SHORT fergo_ex.00401015

Jump if Equal. As the phrase said, if the arguments of the last operation were equal (if true) so is realized a jump to another region of code. In this case the operation was true, so the program will jump to the address 00401015 from the executable fergo_ex1.exe .

I’ll jump the lines of number 4, 5 and 6 for a while (let’s still play). After I’ll say about these lines.

Line 7: LEA EAX, DWORD PTR DS:[403000]

The LEA command makes the first argument to point to the second one. It does NOT receive the “value”, but just the "place" which the value is stored. In this case it will move to the EAX register the address 403000 (32 bits value (DWORD)).

Line 8: LEA EBX, DWORD PTR DS:[403009]

As you can see the same as above, the differences are: Different addresses and different variables (403009 to EBX).

Line 9: PUSH 0

Just "to push" its argument (0) to a temporary place (stack) in the memory, it doesn't realize any other command. See below.

Line 10, 11, 12: PUSH ...

Makes the same thing that the last line, the difference is the value that will be pushed on the stack. In other words is where it pushes the values of the registers EAX, EBX and after that, one more time the 0.

Line 13: CALL <JMP.&user32.MessageBoxA>

Makes a call to a function, in this case this instruction will call the MessageBoxA function contained in the user32.dll . This function is used to show a message in a message box and this message is displayed when you start the program.

MessageBoxA (dono, endereço do texto, endereço do título, tipo)

Dono (Owner) indicates the owner of the dialog box (that is not important now). Endereço do texto (Address of text) is the address of the title from the window. Tipo (type) is the type of the message (button OK/Cancel, Yes/No, etc...). But where's the arguments? The function Call in ASM gets the pushed arguments on the stack in reverse order, for instance: The "Dono" is located in the address 00401025, the text is in the 00401024. Each "Push" inserts data on the stack, to get that data is necessary to start by the reverse order. Imagine a stack of books, the last book inserted on the stack is the first to be retired. Therefore the data are "popped off" in reverse order.

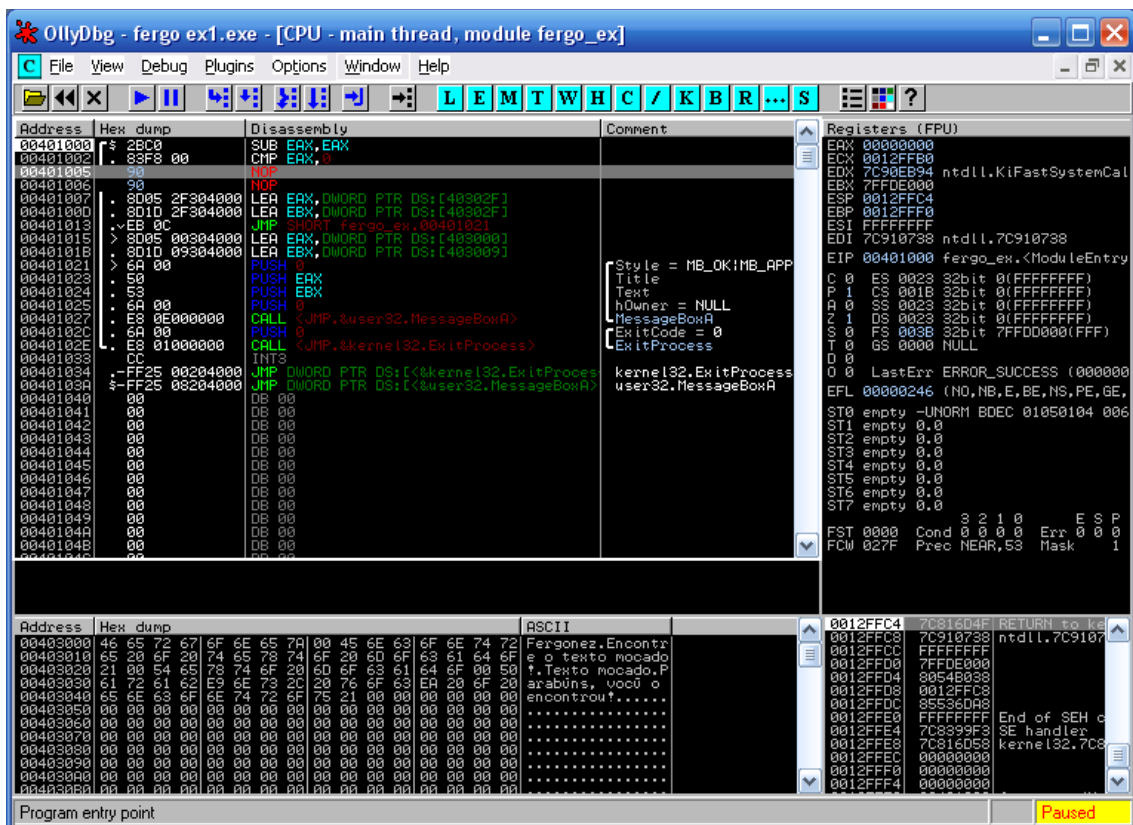
Linha 14: PUSH 0

A new value is thrown on the stack.

Line 15: CALL <JMP.&kernel32.ExitProcess>

Again is made a call to a function. Now the function called is 'ExitProcess'. This function is used to terminate the execution of the program. The argument of this function is a status number which determines if the program had success or not. As you also can see in the program, after you press OK the program closes, that's right before you threw 0 in the "stack" (that means that the function receives the value zero) so I believe you are understanding... If our program terminates when you close the window and this function stops the process with the value 0, now you know that if this function receives the value zero it closes the program after that **duh!** But where's the hidden message? Make an analysis of the program again to see in the lines 7 and 8 that the program determinates the values of EAX and EBX and afterward it calls an instruction to push this two registers (the located values in that) on the stack as text and title of the message box. Notice the lines 4 and 5, we have two LEAs, in this two LEAs happens the attribution of different values to EAX and EBX, but it doesn't made, why?

Notice that in the line 3 is realized a skip if the condition is true (is what occurs, remember?), the program doesn't execute the lines 4 and 5 because the skip occurs in the line 3 (it doesn't reach because of the instruction above). But if we change the code using the Olly? Think about what you shall do if you want to change that to make the program to execute the lines 4 and 5. That's easy, only change the condition, you can remove the jmp instruction in the line 3 or to make the result of the condition always to be false. We will remove the jmp instruction (is more easy to understand). You can't simply delete that line, because that will change all the addresses and the program shall stop! To avoid that exists the NOP command (No operation) that fill with hex 90 the addresses without destroying the program. To make the procedure press the right button of the mouse on the line 3 and after that go to "Binary->Fill with NOPS". The program now has been modified.



Now the program executes the instructions without any jump in the line 3, but now exists different values pointed to EAX and EBX (Probably our hidden message) and there's a jump in the line 6 to address 00401021 after the modification, that place starts to launch the different values of MessageBoxA. Now we'll save the code modified ("cracked"). To make that press the right button on any line and after that go to "Copy to executable -> All Modifications" and after that "Copy All". A new window will be opened, select "Save File" and save the file in some place. Now execute the program to see the message "Parabéns, você o encontrou!" or in English, of course: "Congratulations! You found!". Hold on, part two! Ahh before I forget, sorry my bad grammar, English is not my native language)

Tutorial - Engenharia Reversa

Parte 2



REACHING THE CORRECT PASSWORD

In this tutorial let's learn how to find a valid code to make the correct message to be displayed. Remember that reverse engineering is completely rulez.

-Download: [fwdv2.zip](#)

Before starting the Olly, execute the program and write any character and after that press 'Register'. The message probably was "Wrong Code". That's good! Because:

- 1) The message has been shown in a MessageBox (that's a call of API), we can localize the place in that API and set a breakpoint in all references where there is a call to the function MessageBoxA.
- 2) Across the message "Wrong Code" we can also arrive to the correct address where the arguments are located.

Let's use the first method. Start the Olly and import our target. The code is little, few lines. Press the right button on the main window and after that go to 'Search for -> All Referenced Text Strings'. A new window showing all text from the program will be opened. You can see the message we are looking for "Sorry, wrong code", but see below, there's a message saying "Success! Thanks for Playing". Probably this message is displayed when we use the correct code!

Address	Disassembly	Text string
00401000	PUSH 0	(Initial CPU selection)
00401054	PUSH v2.00403000	ASCII "Fishing with DiLA v0.2"
00401059	PUSH v2.00403017	ASCII "Sorry, wrong code!"
0040106A	PUSH v2.00403000	ASCII "Fishing with DiLA v0.2"
0040106F	PUSH v2.0040302A	ASCII "Success! Thank you for playing ;)"

Go to the "good message". You'll arrive there:

00401068	6A 40	PUSH 40
0040106A	68 00304000	PUSH v2.00403000
0040106F	68 2A304000	PUSH v2.0040302A
00401074	FF75 08	PUSH DWORD PTR SS:[EBP+8]
00401077	E8 28000000	CALL <JMP &user32.MessageBoxA>


```

00401061 . E8 3E000000 CALL <JMP.&user32.MessageBoxA> [MessageBoxA
00401066 . EB 14 JMP SHORT v2.0040107C
00401068 . 6A 40 PUSH 40
0040106A . 68 00304000 PUSH v2.00403000
0040106F . 68 2A304000 PUSH v2.0040302A [Style = MB_OK!MB_ICONASTERISK!MB_APPLMODAL
00401074 . FF75 08 PUSH DWORD PTR SS:[EBP+8] Title = "Fishing with DILA v0.2"
00401077 . E8 28000000 CALL <JMP.&user32.MessageBoxA> [MessageBoxA
0040107C . EB 0E JMP SHORT v2.0040108C [Text = "Success! Thank you for playing ;)"

```

As you also can see in the *Immunity Debugger* this string is the second argument of `MessageBoxA` (That's my sight). All right, but when this function is called? Maybe we need to find a method to arrive here. Now select the first line of this sequence of `MessageBoxA` (0041068). Now notice below to the left the message "Jump from 00401050".

```

004010B6 00
004010B7 00
004010B8 00
004010B9 00
Jump from 00401050

```

Address	Hex dump
00403000	46 69 73 68 69

In all the debuggers (Immunity and Olly). That means to the function be showed the message will need to skip in the address 00401050. Go to the address 00401050 (press CTRL+G and write the address 00401050), there we have:

```
00401050 74 16 JE SHORT v2.00401068
```

If the two elements are equal a skip to the address 00401068 (`MessageBoxA`) is realized, but where's the elements? See above the 00401050 the instruction `CMP` followed by two registers.

```

00401035 . 75 45 JMP SHORT v2.0040107C
00401037 . 6A 00 PUSH 0
00401039 . 6A 00 PUSH 0
0040103B . 68 EA030000 PUSH 3EA
00401040 . FF75 08 PUSH DWORD PTR SS:[EBP+8] [IsSigned = FALSE
00401043 . E8 56000000 CALL <JMP.&user32.GetDlgItemInt> [GetDlgItemInt
00401048 . BB 9A030000 MOV EBX, 39A [pSuccess = NULL
0040104D . 4B DEC EBX [ControlID = 3EA (1002.)
0040104E . 8B3B CMP EBX, EBX [hWnd
00401050 . 74 16 JE SHORT v2.00401068 [hOwner
00401052 . 6A 10 PUSH 10 [Style = MB_OK!MB_ICONHAND!MB_APPLMODAL
00401054 . 68 00304000 PUSH v2.00403000 [Title = "Fishing with DILA v0.2"
00401059 . 68 17304000 PUSH v2.00403017 [Text = "Sorry, wrong code!"
0040105E . FF75 08 PUSH DWORD PTR SS:[EBP+8] [hOwner
00401061 . E8 3E000000 CALL <JMP.&user32.MessageBoxA> [MessageBoxA
00401066 . EB 14 JMP SHORT v2.0040107C [Style = MB_OK!MB_ICONASTERISK!MB_APPLMODAL
00401068 . 6A 40 PUSH 40 [Title = "Fishing with DILA v0.2"
0040106A . 68 00304000 PUSH v2.00403000 [Text = "Success! Thank you for playing ;)"
0040106F . 68 2A304000 PUSH v2.0040302A [hOwner
00401074 . FF75 08 PUSH DWORD PTR SS:[EBP+8] [Style = MB_OK!MB_ICONHAND!MB_APPLMODAL
00401077 . E8 28000000 CALL <JMP.&user32.MessageBoxA> [Title = "Fishing with DILA v0.2"
0040107C . EB 0E JMP SHORT v2.0040108C [Text = "Success! Thank you for playing ;)"
0040109E=<JMP.&user32.GetDlgItemInt>

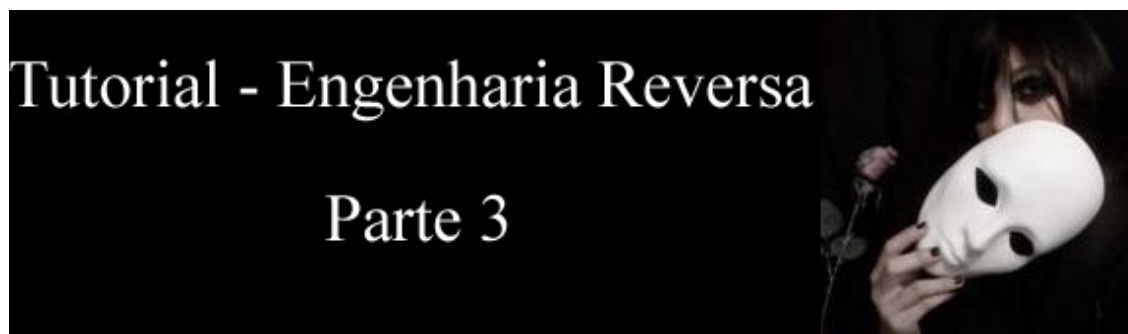
```

In 00401043 the program calls a function to get the data in a dialog box (I guess you are understanding). We haven't a misunderstood thing in our brains now! Usually the function `GetDlgItemInt` obtains the data written in a dialog box to move them after that to `EAX` register (please remember that), after that the program move to the register `EBX` the hex value 39A (decimal 922). In the line 0040104D the register `EBX` is subtracted by 1, i.e. becomes 921. There's an operation of subtraction in the 0040104E, exists an operation between 'EAX' and 'EBX', 'EBX' holds 921 and 'EAX' holds the data written by the user. If they're equal the jump is executed from the line 00401050 to our secret message, unless the result is different we always go to there ;) When the values are different is showed to us the bad message (the operation was unsuccessful).

Now write the number 921 in the dialog window and after that go to 'Register'. Tan dam! The good message was displayed, there are other methods to find the instruction that verifies the correct password. One of the most used was mentioned, another way shall be needed to put breakpoints in calls to basic functions (if the program uses them) . For instance some of this:

- `GetDlgItemText` ou `GetDlgItemText`
- `LStrCmp` ou `StrCmp`
- `GetWindowTextA` ou `GetWindowText`
- `MessageBoxA`
- `GetDlgItemInt`

The first function listed obtains inputs ("texts") written in the dialog boxes by the users, while the second one compares also text (usually called the "strings" in programming ;). The third one obtains text from the window, the fourth is used to show a message and finally the last one obtains data from the dialog boxes. Making it to accept any value, to do that you must realize a modification according to always to be a false operation. Change the JE (Jump if Equal) to JMP (jump). The program always will jump to there (To the "correct message" or "secret message" or password, but that's not important to us :). To know how to modify the codes go to the last chapter.



Understanding the codes

I went to our website and downloaded it ,)

-Download: [keygenme1.zip](#)

PART 1

Before opening the Olly run the program (it has a music ;) and think in something with at least five characters and no more than sixteen and insert a key, unless you have written a string less than four characters long, was showed something like this "Hello, Mr. Badboy!". That's a wrong key. Now open the Olly and after that run our target in that. To find in our program the place where the verification code is located, let's take a look in some functions from the application programming interfaces of the windows, used by the program. Press Alt+E and after that (in the list) press the right button on "Name = keygenme" and go to "View Names".

In the latest tutorial I talked about some interesting functions that we must give more attention to them. They are marked in red.

00404044	.rdata	Import	kernel32.CloseHandle
00404040	.rdata	Import	kernel32.CreateFileA
0040403C	.rdata	Import	kernel32.CreateThread
00404060	.rdata	Import	user32.DialogBoxParamA
00404064	.rdata	Import	user32.EndDialog
00404000	.rdata	Import	kernel32.ExitProcess
00404038	.rdata	Import	kernel32.FindResourceA
00404034	.rdata	Import	kernel32.FreeResource
00404068	.rdata	Import	user32.GetDlgItem
0040406C	.rdata	Import	user32.GetDlgItemTextA
00404004	.rdata	Import	kernel32.GetModuleHandleA
00404070	.rdata	Import	user32.GetWindowLongA
00404074	.rdata	Import	user32.LoadIconA
00404030	.rdata	Import	kernel32.LoadResource
0040402C	.rdata	Import	kernel32.LocalAlloc
00404028	.rdata	Import	kernel32.LocalFree
00404024	.rdata	Import	kernel32.LockResource
00404010	.rdata	Import	kernel32.lstrcatA
00404014	.rdata	Import	kernel32.lstrcmpA
00404018	.rdata	Import	kernel32.lstrlenA
00404078	.rdata	Import	user32.MessageBoxA
00401000	.text	Export	<ModuleEntryPoint>
00404020	.rdata	Import	kernel32.ReadFile
00404008	.rdata	Import	kernel32.RtlZeroMemory
00404084	.rdata	Import	user32.SendMessageA
00404080	.rdata	Import	user32.SetDlgItemTextA
0040401C	.rdata	Import	kernel32.SetFilePointer
0040407C	.rdata	Import	user32.SetFocus
00404058	.rdata	Import	user32.SetLayeredWindowAttributes
00404050	.rdata	Import	user32.SetWindowLongA
00404054	.rdata	Import	user32.SetWindowPos
00404088	.rdata	Import	user32.ShowWindow
00404048	.rdata	Import	kernel32.SizeofResource
0040400C	.rdata	Import	kernel32.Sleep
004040A8	.rdata	Import	winmm.waveOutClose
004040A4	.rdata	Import	winmm.waveOutGetPosition
004040A0	.rdata	Import	winmm.waveOutOpen
0040409C	.rdata	Import	winmm.waveOutPrepareHeader
00404098	.rdata	Import	winmm.waveOutReset
00404094	.rdata	Import	winmm.waveOutUnprepareHeader
00404090	.rdata	Import	winmm.waveOutWrite
0040405C	.rdata	Import	user32.wsprintfA

All that will lead us to algorithm, but one leads more rapidly: lstrcmpA. That function compares 2 strings; it compares the true key with the key you wrote. lstrcmpA also is one of the most popular between programmers (Hackers) around the world. Let's mark a breakpoint in all places where it is called. Press the right button on the line containing lstrcmpA and after that mark 'Set breakpoint on every reference'. Now you can close all the windows and get back to the default window. Run the program by the Olly, one more time insert a name and key and after that go to 'Check'. The program stops in our breakpoint (as you can see below):

```

00401315 | . 5B 73D49000 | JNB EBX, key genuine 73D49000
0040131A | . 50 | PUSH EBX
0040131B | . 50 | PUSH EBX
0040131D | . 58 6B000000 | <JMP, &kernel32.lstrcmpA>
00401321 | . 74 15 | JE SHORT key genuine, 00401338

```

Observe these 2 arguments received by the lstrcmpA before the call. EBX contains 123456 (What I thought. My efforts I promise!) and EAX holds a string ;) in a typical format of key. After that it calls the function (lstrcmpA) that will compare the two values (123456 with "Von-FF..." both prior to the lstrcmpA) if the answer is true the code make a jump to 00401338 (JE as you can see after the call). Here we go, now go to the address 00401338 to see the place which the message saying that the message is correct is showed (That's incredible!).

Go to 00401338 to see the place where is the message “Correct key”. Copy the value from the EBX and after that test it (in the keygenme) , use the same name inserted because the key is made based on the name. Bingo!



PART 2

Ok the corresponding key to the entered name has been found, they have been showed above. But we don't know how that key is generated. To find keys and their respective values in programs thereby isn't hard (As you could see above =). Remove all breakpoints (Alt + B). Now select "View Names" in the first item of the list. Let's set a breakpoint in all references where the program calls the GetDlgItemTextA (that function obtains texts in a dialog box). If you don't know how, go to the **PART 1** and see the text and its letters ; P Mark the breakpoints and go to "Play", write a name and a key and after that go to 'Check'. We are where the program obtains the text of one of the text boxes, notice that there are two calls to GetDlgItemTextA, obviously one of them (the first one ,) obtains the name and the other obtains the key. Press F9 to continue the execution and after the pause the program obtains the second value. To see more detail:

```
004010E9 6A 28 PUSH 28
004010EB 68 F8DC4000 PUSH keygenme.0040DCFB ; Armazena o nome em 0040DCFB
004010F0 68 EE030000 PUSH 3EE
004010F5 FF75 08 PUSH DWORD PTR SS:[EBP+8]
004010F8 E8 B3020000 CALL <JMP.&user32.GetDlgItemTextA> ; Chama a função para pegar o nome
004010FD 6A 28 PUSH 28
004010FF 68 F8DE4000 PUSH keygenme.0040DEF8 ; Armazena a key em 0040DEF8
00401104 68 EF030000 PUSH 3EF
00401109 FF75 08 PUSH DWORD PTR SS:[EBP+8]
0040110C E8 9F020000 CALL <JMP.&user32.GetDlgItemTextA> ; Chama a função para pegar a key
00401111 E8 F2000000 CALL keygenme.00401208
```

After that is realized a jump to 401208 (in the address 00401111). The instinct told me that the call probably generates a key based on the name (as we saw). So let's "enter" in this function to analyze what makes. Select the line 00401111 and press ENTER.

```
00401208 68 F8DC4000 PUSH keygenme.0040DCFB
0040120D E8 80010000 CALL <JMP.&kernel32.lstrlenA>
00401212 A3 86DC4000 MOV DWORD PTR DS:[48DC86],EAX
00401217 833D 86DC4000 CMP DWORD PTR DS:[48DC86],4
0040121E 0F8C 29010000 JL keygenme.00401340
00401224 833D 86DC4000 CMP DWORD PTR DS:[48DC86],32
0040122B 0F8F 1C010000 JG keygenme.00401340
00401231 33C0 XOR EAX,EAX
00401233 33DB XOR EBX,EBX
00401235 33DB XOR EDI,EDI
00401237 68 F8DC4000 MOV ECX,keygenme.0040DCFB
0040123C 8B15 86DC4000 MOV EDI,DWORD PTR DS:[48DC86]
00401242 > 0FB60439 MOVZX EAX,BYTE PTR DS:[ECX+EDI]
00401246 83E8 19 SUB EAX,19
00401249 2B08 SUB EBX,EAX
0040124B 41 INC ECX
0040124C 3BCA CMP ECX,EDX
0040124E ^75 F2 JNZ SHORT keygenme.00401242
00401250 68 F8DB4000 PUSH keygenme.0040DBF8
00401256 68 F8E04000 PUSH keygenme.0040E0F8
0040125B E8 38010000 CALL <JMP.&user32.wsprintfA>
00401260 83C4 0C ADD ESP,0C
00401263 33C0 XOR EAX,EAX
00401265 33D2 XOR EDI,EDI
00401267 33C9 XOR ECX,ECX
00401269 03C8 ADD EAX,EBX
0040126B 0FAC IMUL EAX,EBX
0040126E 03C8 ADD ECX,EAX
00401270 2B03 SUB EDI,EBX
00401272 33D0 XOR EDI,EAX
00401274 0FAD IMUL EBX,EAX
00401277 68 F8DB4000 PUSH keygenme.0040DBF8
0040127D 68 F8E14000 PUSH keygenme.0040E1F8
00401282 E8 11010000 CALL <JMP.&user32.wsprintfA>
00401289 83C4 0C ADD ESP,0C
0040128B 33C0 XOR EAX,EAX
0040128E 33D2 XOR EDI,EDI
00401290 33C9 XOR ECX,ECX
00401292 68 F8E04000 MOV EAX,keygenme.0040E0F8
00401297 03D8 ADD EBX,EAX
00401299 33C8 XOR ECX,EBX
0040129B 0FAC IMUL ECX,EBX
0040129E 2B03 SUB ECX,EAX
004012A0 51 PUSH ECX
004012A1 68 F8DB4000 PUSH keygenme.0040DBF8
004012A6 68 F8E24000 PUSH keygenme.0040E2F8
004012AB E8 E8000000 CALL <JMP.&user32.wsprintfA>
```

String = ""
lstrlenA

<%IX>
Format = "%IX"
s = keygenme.0040E0F8
wsprintfA

<%IX>
Format = "%IX"
s = keygenme.0040E1F8
wsprintfA

<%IX>
Format = "%IX"
s = keygenme.0040E2F8
wsprintfA

1
2
3

I'll talk about the important lines. As you can see the algorithm is divided into three main sequences, 3 sequences almost equals. The correct key is composed of "Bon-" followed by 3 numeric sequences. We can suppose that each one of the blocks of code (marked is red) are responsible to generate each part of key.

-- Analysis of first sequence (00401208)

Let's start our analysis by the address 00401208. The program firstly obtains the size of the name and holds in EAX, after that moves the stored value to address [40DC86]. In the following lines the program compares the size of the name (located in [40DC86]) with four (4) and after that with thirty-two 32 (50 in decimal). If the result is larger than fifty (50) or less than four (4) then the program makes a jump to a error message. Presupposing you wrote a string with the size $4 \leq \text{nome} \leq 50$ let's continue. In 00401231 starts the first sequence of key. The program fills with zero inside of the registers EAX, EBX and ECX (XOR X, X). After that moves the name, the size of the name (EDI) and in the next line starts a "loop". Look the sequence.

```

00401242 0FB60439 MOVZX EAX, BYTE PTR DS:[ECX+EDI] ; Move to EAX the byte pointed by ECX ( initially zero )
00401246 83E8 19 SUB EAX, 19 ; Subtracts 19 ( 25 in decimal ) from EAX ( EAX = EAX - 25 )
00401249 2BD8 SUB EBX, EAX ; Subtract EAX from EBX ( EBX = EBX - EAX )
0040124B 41 INC ECX ; Increases ECX ( ECX = ECX + 1 )
0040124C 3BCA CMP ECX, EDX ; Compares ECX with EDX ( that contains the size of the name )
0040124E 75 F2 JNZ SHORT keygenme.00401242 ; If the operation is false ( ECX different of EDX ), get back to the start
of the loop ( 1242 )

```

Summarizing the previous code: It gets the ASCII value of each character, takes thirty-five and after that subtracts that founded value by EBX, something like:

For i = 1 to Tamanho do nome

 Sequencial = Sequencial - ASC(Caractere(i)) - 25

Next

After the calculate in the first sequence the program calls a function to format the text and stores it in determined address. In this case the program will move the result (stored in EBX) to the position of memory [0040E0F8]

- Another sequence (00401263)

Again the program starts filling with zero the registers EAX, EDX and ECX .

```

00401269 03C3 ADD EAX, EBX ; EAX = EBX ( i.e. EAX will have the value of the sequence from the last algo, storage in
EBX )
0040126B 0FAFC3 IMUL EAX, EBX ; EAX = EAX * EBX ( With the two registers the same value )
0040126E 03C8 ADD ECX, EAX ; EAX = EAX * EBX ( With the two registers the same value )
00401270 2BD3 SUB EDX, EBX ; EDX = EDX - EBX ( EDX has the negative value from the EBX ) – Unuseful instruction
00401272 33D0 XOR EDX, EAX ; EDX receive the result of the binary operation XOR between EDX and EAX - Unuseful
instruction
00401274 0FAFD8 IMUL EBX, EAX ; EBX = EBX * EAX ( EBX is "multiplicated" with EAX ( that has been "multiplicated" with
EBX )

```

Make an analyze of the last instruction with prudence to discover that the second sequence it's the generated value in the first "elevated to cube", thus:

Sequencia2 = Sequencia1 * Sequencia1 * Sequencia1

The value of the sequence is stored in the same form as the first sequence, but in the memory address [0040E1F8]

-Another sequence (0040128A)

As always it starts filling with zeros the registers EAX, EBX, EDX, ECX

```

00401292  B8 F8E04000  MOV EAX, keygenme.0040E0F8
00401297  03D8         ADD EBX, EAX
00401299  33CB         XOR ECX, EBX
0040129B  0FAFCB      IMUL ECX, EBX
0040129E  2BC8         SUB ECX, EAX

```

This last sequence has a particular feature. Observe that any of these "variables" are variables **duh!** (WTF?!). The program never makes something according to the written name. The value from the EAX in 00401292 go to 0040E0F8 always, because the address of the instruction never is changed. Making all necessary calculus (use the calculator from the Windows), you'll arrive in the value '41720F48', located in the position [0040E2F8]. Sequencia3 = 41720F48 . We understand entirely the algorithm. The program mounts a complete string (join all), so the rest is not important to us, but if you want to understand it let's go: The program makes that initially in the line 004012C5. Remember also that is increased a "Bon-" firstly to the beginning of the key. Now you must choose a language to program your generator. I'll make a generator using the Visual Basic.

- Algorithm to generate (Generator), in VB

```

Private Sub Gerar_Click()
    Dim seq1 As Double, seq2 As Double
    Dim tamanho As Integer, i As Integer

    'obtains the size of the name
    tamanho = Len(txtNome.Text)
    If (tamanho < 4) Or (tamanho > 50) Then Exit Sub

    'sequencia 1
    For i = 1 To tamanho
        seq1 = seq1 - (Asc((Mid(txtNome.Text, i, 1))) - 25)
    Next


    'sequencia 2
    seq2 = seq1 ^ 3

    txtSerial.Text = "Bon-" & Hex(seq1) & "-" & Hex(seq2) & "-41720F48"
End Sub

```


Tutorial - Engenharia Reversa

Parte 4

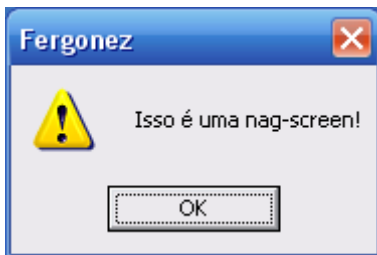


REMOVING NAG SCREENS


This tutorial it's easier to understand and was made especially to people working with reverse engineering for the first time. The main focus (our tee) is to remove a kind of message called "Nag Screen", they're boring windows showed when you run some programs. However in this case is simpler than others (covered in a next tutorial, of course), so let's go. Download our target (programmed by F3rGO and translated by 6_Bl4ck9_f0x6 me **duh!**)

-Download: [fergo_nag.zip](#)

Run our target to see a "MessageBox" saying something about nag-screens =) and after that go to "OK", after that we are in the program.



This is a nag-screen!

All right, open it (the Olly) and load the executable. We've several forms to arrive in the place where the message box is showed, one of that is to search for all the contained strings in the program (Right button -> Search For -> All referenced text string). Another method is to find the call to MessageBox, that moves directly to the call of nag-screen, because exists a unique MessageBox in all software. Now press Alt+E and in the list click on "fergonag" using the right button and after that go to "View Names". In the list of all function used by the program go to MessageBox. Let's set a breakpoint in the place where the function is called. Click using the right button of the mouse on "user32.MessageBoxA" and select "Set Breakpoint on every reference". Now run the program (Press F9 to run our program or click on the play button ). The program stops before the call to MessageBox (how we know yet ,).

```
00401023 | . 6A 30 | PUSH 30 | Style = MB_OK!MB_ICONEXCLAMATION!MB_APPLMODAL
00401025 | . 50 | PUSH EAX | Title -> "Fergonez"
00401026 | . 53 | PUSH EBX | Text -> "Isso é uma nag-screen!"
00401027 | . 6A 00 | PUSH 0 | hWndex = NULL
00401029 | . E8 B0010000 | JMP <.user32.MessageBoxA> | MessageBoxA
```

```

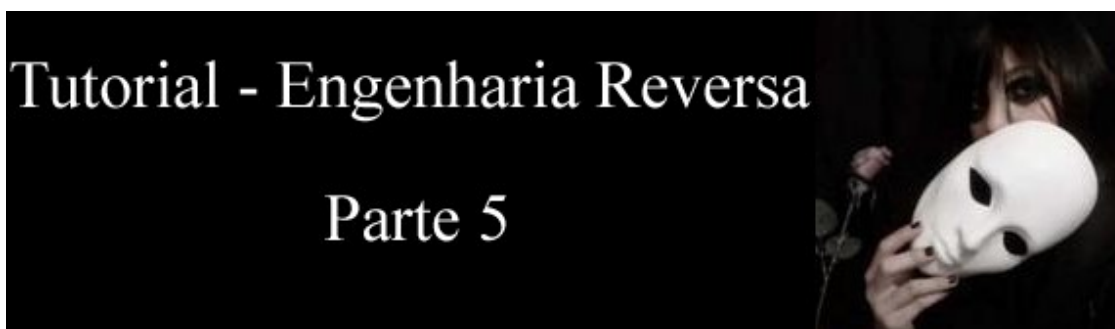
00401023  6A 30      PUSH 30      ; Pushes to stack the type of the msgbox (Exclamation Icon)
00401025  50        PUSH EAX    ; Pushes the title
00401026  53        PUSH EBX    ; Pushes the text
00401027  6A 00     PUSH 0      ; Owner of the window
00401029  EB B0010000 CALL <JMP.&user32.MessageBoxA> ; Calls the function to use the 4 pushed arguments

```

Let's remove our msgbox. We can also simply modify the content of that one (saying "register at http://*");, but it's so easy to remove! "To remove" a line use the NOP command (No OPERATION), select the lines corresponding to the addresses of MsgBox (five lines), start the process of selection in the address 00401023 and stop in the 00401029, after that click using the right button and go to the option "Binary -> Fill with Nops".

00401023	90	NOP	Style Title => "Fergonez" Text => "Isso � uma nag-screen!" hOwner MessageBoxA
00401024	90	NOP	
00401025	90	NOP	
00401026	90	NOP	
00401027	90	NOP	
00401028	90	NOP	
00401029	90	NOP	
0040102A	90	NOP	
0040102B	90	NOP	
0040102C	90	NOP	
0040102D	90	NOP	

Now run the program, as you can see the nag-screen disappears! If you wanna save the modifications in a modified executable "aside", as always Right button -> Copy to executable file -> All Modifications" and after that "Copy All". In the new window click again using the right button of the mouse and go to "Save File" as fg_cracked.exe.



CREATING A PATCHER

This tutorial teaches how to modify bytes from the executable to modify its behavior without using the OllyDbg. I'll use the same executable (Chapter/Tutorial #4).

-Download: [fergo_nag.zip](#)

INTRODUCTION

If you read the last chapter #4 (was needed :) you maybe remember that we should to fill with NOPs some lines to cancel a function. In that case the hex code (OpCode) is 90, equivalent to one byte, but the problem is: How to do that without understanding what is the RCE and/or without using the OllyDbg? That's easy; you compile a program/patcher to modify the target! We had this situation in the last chapter...

```

00401023 .: 6A 30      PUSH 30
00401025 .: 50        PUSH EAX
00401026 .: 53        PUSH EBX
00401027 .: 00        PUSH 0
00401029 .: E8 B0010000 CALL <JMP.&user32.MessageBoxA>
0040102E .: 56        PUSH ESI
00401030 .: 40        PUSH 40
00401035 .: 40        PUSH 40
00401037 .: E8 BA010000 CALL <JMP.&kernel32.GlobalAlloc>

```

```

Style = MB_OK|MB_ICONEXCLAMATION|MB_APPLMODAL
Title => "Fergonez"
Text => "Isso é uma nag-screen!"
hOwner = NULL
MessageBoxA

MemSize = 400 (1024.)
Flags = GPTR
GlobalAlloc

```

I need to explain two things; each instruction in ASM has a value that generally occupies 1 byte followed by an argument. The second column of the Olly contains the value of the instruction (OpCode) in hexadecimal, and the third column shows what we call “Mnemonic”, that is the OpCode “translated” . Notice the line 00401029. We’ve the OpCode “E8 B0010000” and its corresponding mnemonic, referring to the OpCode: "CALL <SMP.&user32.MessageBoxA". In this case E8 indicates the CALL and the others 4 bytes "B0010000" (Please friend never forget that we are working using hexadecimal values and they can vary - 00 to FF (0 to the 255). Two hex values occupy 1 byte in the memory) represents the function MessageBoxA. Now I’ll talk about “Address”, the number of the line. That indicates the location of each instruction that will be executed, in hexadecimal. Address starts in 0 and grow up in each instruction, for example I have a command line in 00 and that occupies 2 bytes in the memory (A PUSH instruction followed by a constant, for instance: PUSH 69), the next instruction will be located in the address 02, i.e. 00 with two bytes occupied by the last instruction that means that the instruction will be located in the 02. Across the address we’ll write the Patch, but... there’s a detail: The executable file doesn’t start using the commands, before starting the commands it has a header that indicates several features, etc. That header has a size equal to 1024 byte (400 in Hex) and just after that the instructions are allocated/started, i.e. the command will never be allocated in the byte 0, but in the byte 1025 (401 in Hex). Let’s back to the program. Observe the line 00401023. There the four necessary elements to the CALL are pushed on the stack. In the last tutorial we changed all corresponding to the MsgBox. When you fill with NOP the lines the Olly laid the value 90 (NOP) in the byte 29 (where the E8 (CALL) was stored) and laid also several others in “new lines”. Why? Because to fill with zeros the byte 29 the program also cancel the following 4 bytes (remember: E8 B1 01 00 00) because of that the program inserted in the bytes 29, 2A, 2B, 2C, 2D some NOPs too. If we want to code a patcher we must cancel all referring to text message. See in the table below the original bytes from the file and the bytes that we will modify:

	Bytes originais	Bytes Modificados
23	6A (PUSH)	90
24	30	90
25	50 (PUSH EAX)	90
26	53 (PUSH EBX)	90
27	6A (PUSH)	90
28	00	90
29	E8 (CALL)	90
2A	B1	90
2B	01	90
2C	00	90
2D	00	90
2E	56 (PUSH ESI)	90

CREATING A PATCHER

Before creating the patcher let's remember what and where we need to modify. We need to fill eleven (11) bytes (initially the patcher will start to mark where's the first PUSH, in the address 00401023 to the 00 in the address 0040102D) using the value \x90, that indicates none operation. Let's insert that in the executable (of course). Remember guy be aware: NEVER, I said NEVER start to insert initially in the byte 23 (35 in decimal), 24 (36 in decimal), etc; exists 1024 (400 in hex. To convert hex numbers use the calculator of the Windows) composing the header of executable, that means that we must write from the 423 to the 42D (1059 to the 1069). I'll write again the code using VB, it's more easy to understand. I could write in a compressed form, but will be misunderstood by some people. Add the Command Button named as cmbPatch). The patcher need to stay in the same folder of the target.

Code

```
Private Sub cmbPatch_Click()
    Dim bytFile() As Byte
    Dim strFile As String
    Dim i As Integer

    strFile = App.Path & "\fergonag.exe"

    ReDim bytFile(FileLen(strFile) - 1) As Byte

    Open strFile For Binary As #1
        Get #1, , bytFile()
    Close #1

    For i = &H423 To &H42D
        bytFile(i) = &H90
    Next

    Open strFile For Binary As #1
        Put #1, , bytFile()
        MsgBox "Alvo alterado com sucesso", vbInformation, "Wee"
    Close #1
End Sub
```

---=[Commented]=---

Finished, now compile it! I'm not sure (at the time of this writing ;), but maybe that run using VB script (I believe – “at the time of this writing”). If you want download the source code to test in the “default” of the VB, download in my place of storage – link: [patch_src.zip](#).

Tutorial - Engenharia Reversa

Parte 6



UPX UNPACKING

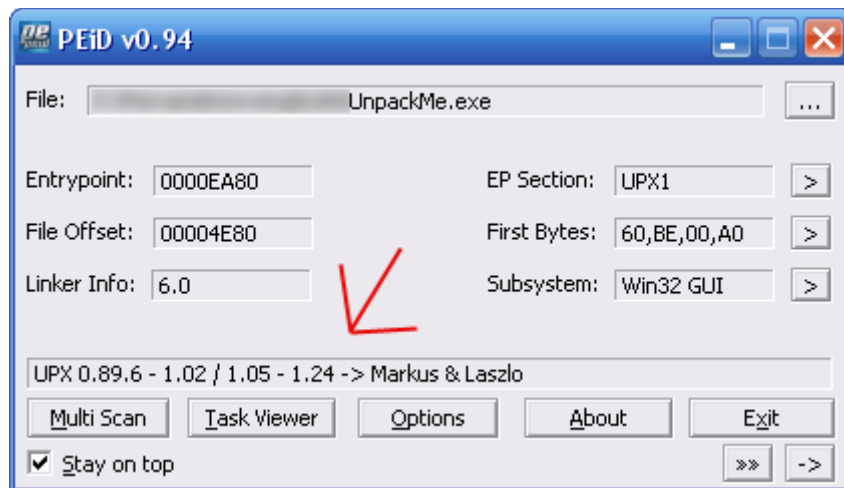
Several applications normally are distributed compressed, the reasons: to reduce the size of the executable file, or just “to encrypt it”, for that reason the debugging becomes hard (extreme difficulties for **beginners** – OF COURSE). One of that (one of the most famous compressors) is called UPX, acronym for “ Ultimate Packer for eXecutables”. It has a good compression and unpack the original executable file rapidly. The UPX is OP, acronym for Open Source in a “Cracker language” and you can download it freely at <http://upx.sourceforge.net/> . There are many forms to unpack executable files. You can use the “UPX unpacker” or can use other unpacker or simply to do the act manually (we will do that). First of all I’ll explain how a UPX compressor works. When you utilize a “packer” like UPX, the EP (Entry Point, where starts the code of the program) is redirected (deviates) to another address which contains what’s called “Loader”. The function of the loader is only to unpack the executable file to the memory and after that to call the OEP (Original Entry Point, containing the true code unpacked). The sequence is like this:

EP -> loader -> unpacks the contents to the memory -> move EP to the OEP

The manual process to unpack UPX is almost always the same, follow the same sequence to unpack. I’ll not explain what happens with the loader (how it works), let see just how to unpack the starting point of the code (OEP) and find the compressed executable in the memory. To do that, let’s use the Olly and also another application that remount the base of Imports of the executable files (To know more about that, find by “format of executable files”) called ImportREC. What follows is a simple executable coded in VB and packed with UPX.

-Download: unpackme.zip

To know if the executable file has some protection I like to use the PEiD (that’s a excellent program, so much more useful than others at the moment of this writing, as always brother), it makes analysis and detects almost all the kinds of compressors. Download and unpack in the folder of the Olly the dll of the plug-in (“plugin”) [OllyDump](#), we will use later, for a while see this image below.



This program above was compiled with the UPX as you can see (the version is not important, the process is the same for all). Open the Olly and load our executable. A warning saying that it was compressed will be showed, and still asks if we wish to continue the execution, the answer is Yes (an easy answer to this question ;). Now we are in the Entry Point (of the loader still):

0040EA80 60 PUSHAD

Remember this line above, that's a typical instruction of UPX. All the compressed files with it will start using a PUSHAD. As I said, I will not explain why this process is being made, I just will indicate what you must do. Press F7, the instruction PUSHAD will be executed and a skip to the next line (EA81) is realized. See in the window of registers (to the right), the ESP is colored, that means that it changed (the value contained in that). Let's set a hardware breakpoint in the content of the first byte of ESP, because it is accessed only in the end of our loader, that means that we will stop near of the place where the loader ends, and it calls the OEP, what we need. Click using the right button of the mouse on the value of ESP and go to "Follow in Dump". Now we are monitoring the content of ESP (lower side of the screen). Click using the right button on the first byte in "dump" (38) and select "Breakpoint -> Hardware, on access -> DWord". This breakpoint will lead to the end of the loader, near of the call to OEP (to know why to use these breakpoint, you need to study about behavior of the loader. That is not the objective of this tutorial).

Address	Hex dump	ASCII
0012FFA4	38 07 91 7C FF FF FF FF F0 FF 12 00 C4 FF 12 00	8*! - + - +
0012FFB4	00 A0 FD 7F 94 EB 90 7C B0 FF 12 00 00 00 00 00	.a?00E! +... .
0012FFC4	4F 60 81 7C 38 07 91 7C FF FF FF FF 00 A0 FD 7F	Om!8*! .a? .
0012FFD4	38 B0 54 80 C8 FF 12 00 A8 CD 60 85 FF FF FF FF	%T% +.='a
0012FFE4	F3 99 83 7C 58 60 81 7C 00 00 00 00 00 00 00 00	%0a!Xm!.....
0012FFF4	00 00 00 00 80 EA 40 00 00 00 00 00	...000.....

Press F9 to continue the execution of the program. Almost instantaneously I am in the marked breakpoint.


```

0040EC06  61          POPAD
0040EC07  8D4424 80    LEA EAX,DWORD PTR SS:[ESP-80] ; Paramos aqui
0040EC0B  6A 00     PUSH 0
0040EC0D  39C4     CMP ESP,EAX
0040EC0F  75 FA     JNZ SHORT UnpackMe.0040EC0B
0040EC11  83EC 80    SUB ESP,-80
0040EC14  E9 7725FFFF JMP UnpackMe.00401190

```

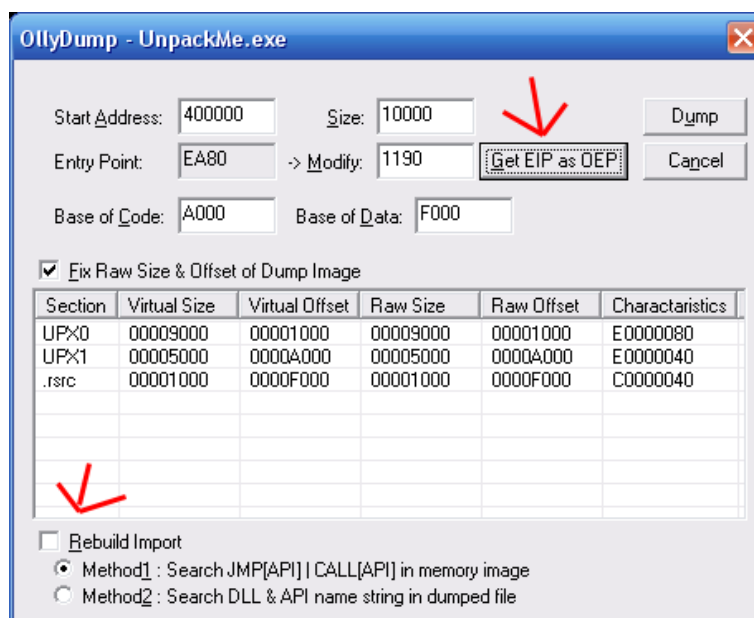
See the POPAD, that's typical thing of UPX, starting with PUSHAD and ending POPAD. Notice in the last line an obligatory jump to the address 1190. That jump indicates the end of the loader, the unpacked application was copied to the memory, that jump will lead us until the original starting point (seeing the jump we know that the OEP is 00401190). Lay a common breakpoint in the line of the jump (0040EC14) pressing F2. F9 to continue until we reach the jump and after that press F7 one more time to realize a jump to the original entry point.

```

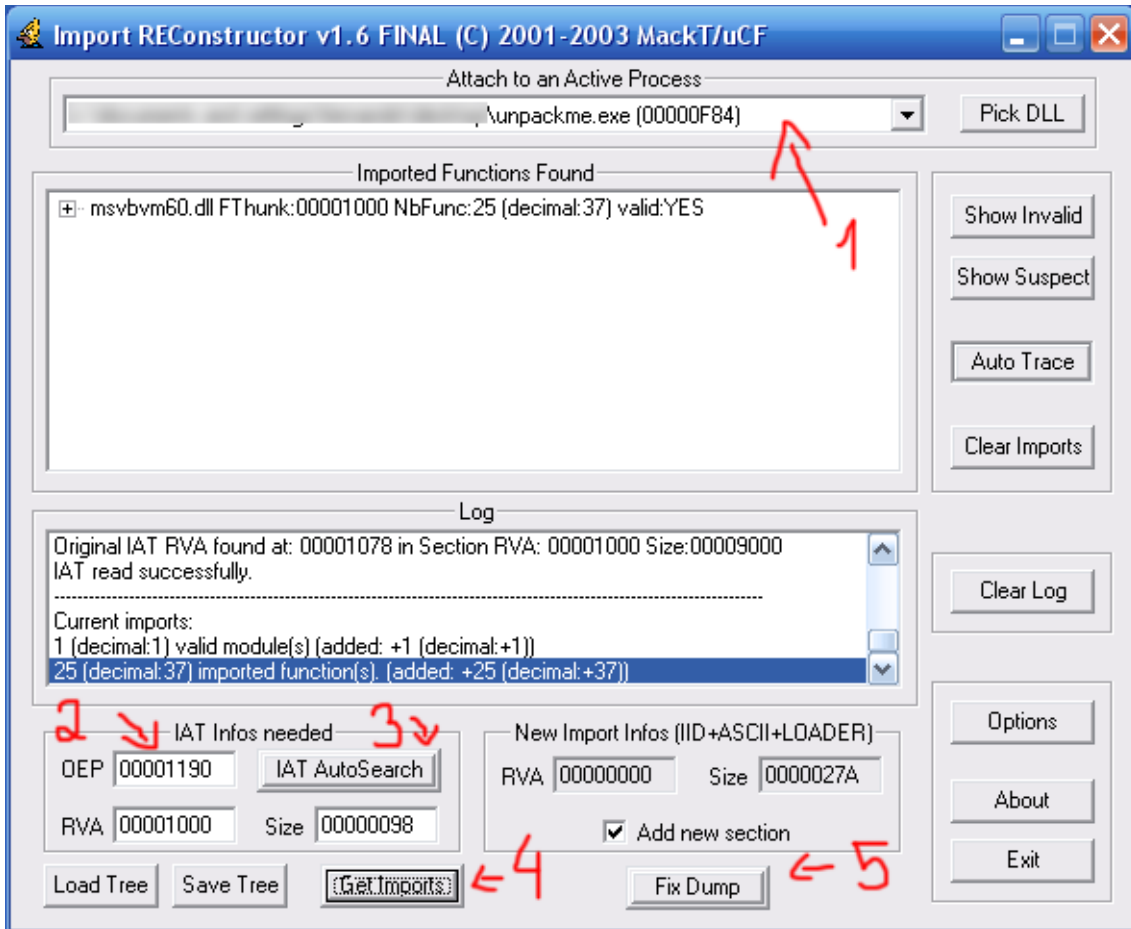
00401190  68 A09E4000 PUSH UnpackMe.00409EA0
00401195  E8 EFFFFFFF CALL UnpackMe.00401188
0040119A  0000     ADD BYTE PTR DS:[EAX],AL

```

If you made all correctly now you are in the address 00401190 that shows the original code of the application, which was unpacked to the memory. Let's obtain that application in the memory (uncompressed application) and after that to mount a new executable by it (based on it), indicating the correct starting point (1190). We'll use the OllyDump. With the program stopped in the address 1190, go to "Plugins -> OllyDump -> and finally Dump debugged process".



The Olly make all the calculations (the dirty work :) for you, inclusive the OEP, but if you want to guarantee click on "Get EIP as OEP". Unmark also the text box "Rebuild Import"; we will not use other programs to remount the "base of Imports" (or "Import base" - I'm Brazilian, I'm afraid to translate in an illegible form to you ;). Click on "Dump" and choose the name and the place where you wish to save the unpacked executable file ("Dumped.exe" was my name of choice). The error displayed when you start the generated file occurs because of the UPX to organize the import section from the executables, we just need to remount them. To do that let's use the ImpRec. Run the original .exe (included in the .zip) and let the Olly opened too. Start the ImpRec and use these settings:



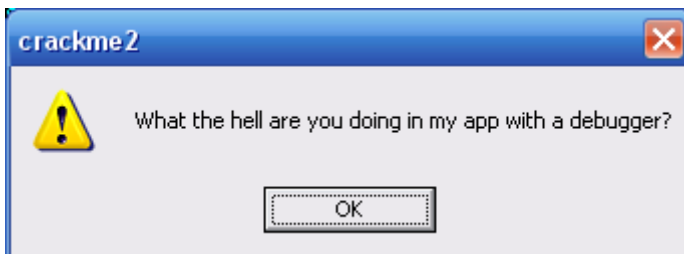
Where the number two is marked you need to insert the values obtained in the text box "Modify", in the Dump Window of the Olly. When we click on Get Imports (4) appears some items in the list "Imported Functions Found". All the items need be showed as "YES" in the end; else the import table will not be correctly mounted. Now click on [Fix Dump] (5). When you click on the program it requests an executable file. Select that program created by the Olly named as "Dumped.exe". After you determine the file the program will generate an unpacked executable definitive (with a underline in the end of the name: Dumped_ .exe) and also will show in the Logs (from the ImpRec) a message. Close all the other programs and run "Dumped_ .exe". A good message was sent, the file was uncompressed, make a new scan with the PEiD in the "Dumped_ .exe" to make sure or simply compare the sizes.

Tutorial - Engenharia Reversa

Parte 7

TRAPS AND ALTERATIONS OF CODE

Some application has protections against debuggers (such as OllyDBG, SoftICE, WinDbg...), our target too, that utilized protection generates an exception (deviates the code that many times generates an error) that shows a message saying we are using a debugger and after that closes the program.



Our target:

Download: [crackme2.zip](#)
(coded by **mucki**)

Open the target using Olly and after that press F9 to start the execution. The program stops in the address 401047. See the status bar the following message:

Singlestep event at crackme2.00401047 -use Shift+F7/F8/F9 to pass exception to program

That indicates that the program found an exception and stopped before showing the window. If you press F9 the program will continue and will show a MessageBox (as you can see above) and after that closes the program. The Olly allows you to skip that exception and continue running the program normally, just press the keys Shift+F9. After the “bypassing” we made on the “anti-debugging system” let’s modify our code. The application begs a name and password, compares and checks the information. The program shows good or bad messages according to the data. Now Press Ctrl+N to open a window showing names of functions used by the program. See the function "lstrcmp".

That function compares 2 strings and the program “sets” a ‘Zero Flag’ as true. Select "lstrcmp" in the list and press Enter. A new window appears showing the times it was used (in this case 2), let’s to attempt one of the two to verify if we are in the correct place. Select the first and press Enter again. We are here:

0040121F	· 2BC1	SUB EAX,ECX	<XIX>
00401221	· 0FFFC0	IMUL EAX, EAX	<XIX>
00401224	· 50	PUSH EAX	Format = "CM2-%IX-%IX"
00401225	· 51	PUSH ECX	s = crackme2.004062B6
00401226	· 68 E1604000	PUSH crackme2.004060E1	wsprintfA
0040122B	· 68 B6624000	PUSH crackme2.004062B6	Count = 4B (75.)
00401230	· E8 9B010000	CALL <JMP.&user32.wsprintfA>	Buffer = crackme2.004060BA
00401235	· 68 4B	PUSH 4B	ControlID = 2
00401237	· 68 9A604000	PUSH crackme2.0040609A	hWnd
0040123C	· 6A 02	PUSH 2	GetDlgItemTextA
0040123E	· FF75 08	PUSH DWORD PTR SS:[EBP+8]	String2 = ""
00401241	· E8 A2010000	CALL <JMP.&user32.GetDlgItemTextA>	String1 = ""
00401246	· 68 BA604000	PUSH crackme2.004060BA	CM2-%IX
0040124B	· 68 B6624000	PUSH crackme2.004062B6	lstrcmpA
00401250	· 75 18	JNZ <JMP.&kernel32.lstrcmpA>	
00401255	· 6A 00	PUSH 0	Style = MB_OK MB_APPLMODAL
00401257	· 68 00604000	PUSH crackme2.00406000	Title = "crackme2"
0040125E	· 68 3D604000	PUSH crackme2.0040603D	Text = "Valid serial - now write a keygen!"
00401263	· FF75 08	PUSH DWORD PTR SS:[EBP+8]	hOwner
00401266	· E8 89010000	CALL <JMP.&user32.MessageBoxA>	MessageBoxA
0040126B	· EB 14	JMP SHORT crackme2.00401281	
0040126D	· 6A 10	PUSH 10	Style = MB_OK MB_ICONHAND MB_APPLMODAL
0040126F	· 68 0A604000	PUSH crackme2.0040600A	Title = "crackme2"
00401274	· 68 60604000	PUSH crackme2.00406060	ASCII "Wrong serial - try again!"
00401279	· FF75 08	PUSH DWORD PTR SS:[EBP+8]	hOwner
0040127C	· E8 73010000	CALL <JMP.&user32.MessageBoxA>	MessageBoxA
00401281	· 90	POP EBX	
00401282	· 90	POP ESI	
00401283	· 90	POP EDI	

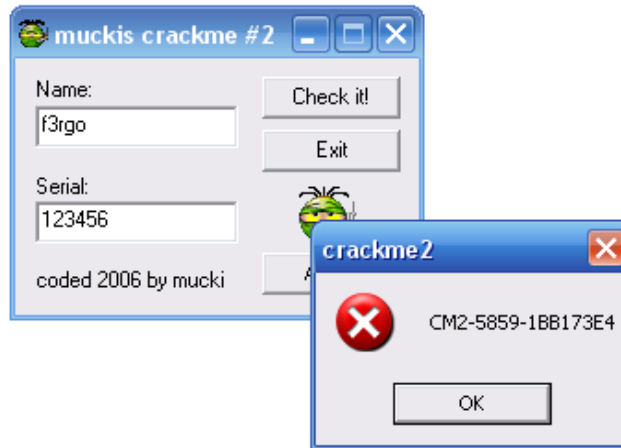
See the functions around, we have several typical functions, you can see manipulation of a text in a serial format (wsprintfA), GetDlgItemTextA (obtain our serial to compare) and the two MsgBoxes indicating if the serial is true or not. Let’s change the code of MsgBox? The program will show a good message. Notice in the function “lstrcmpA” again, that function needs of two arguments to start the process, one of the strings obviously it’s the serial we wrote and the other is the valid serial. One of the strings is located in the address 004060BA (first argument “pushed”) and the other is located in the 004062B6 (second one). Where is the arguments? You can see the code of generation of the key (above) or to make many attempts (test a determined address and after that other). The true serial is located in the address 4062B6 and the serial we wrote is located in the 4060BA, we need to make a modification in the text of the MsgBox; so instead the message "Wrong Serial - try again!" the good message is showed. The best way to do that is simply go the correct place where the bad text is pushed on the stack (401274):

```
00401274 68 60604000 PUSH crackme2.00406060 ; ASCII "Wrong serial - try again!"
```

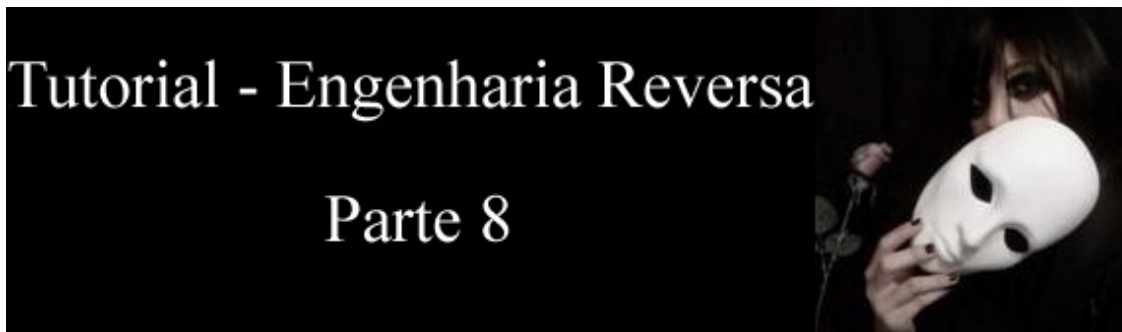
Double Click on "PUSH crackme2.00406060" and change, "PUSH 004062B6”:

```
00401274 68 60604000 PUSH crackme2.004062B6
```

We made the following modification: Instead the call to the address 406060 containing the bad message, we call the address containing the correct serial (4062B6). To test all modifications just save all modifications into a new executable file, click using the right button on the main window, go to "Copy to Executable -> All Modifications -> Copy All". In the new window, one more time click using the right button and select "Save File". Save the new file, run, and write a name and a serial to see the following message.



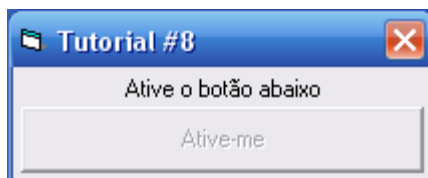
That's the serial use it! The bad message disappears as you can see ;) I've a simple task, decipher the algorithm of the serial (starts in the address 004011F6). Simply is realized many basic operations with each character of the name you wrote. The code could be modified without any manipulation of the exception.



Hi controls

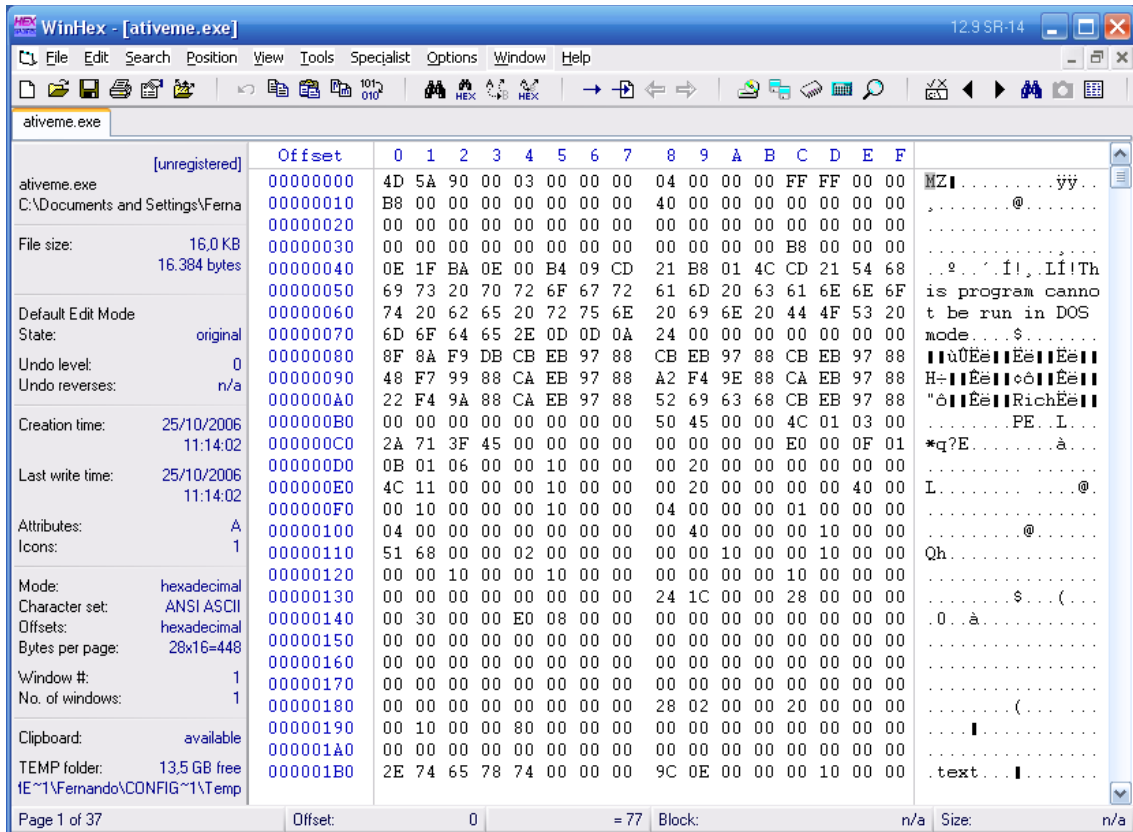
I will explain in this tutorial how to modify the state of a specific control (a command button, in this case) in any application coded in Visual Basic. I will not utilize the Olly (you also can use it), we will use a hex editor (can be any hex editor), it must have a tool for searching. I like to use the WinHex. As always download the file (the file was coded in VB) in the following link:

Download: [ativeme.zip](#)



Before starting the process I'll tell you a little tale about the code of the VB. Unlike most people think VB application if compiled in Native Code generates normal codes (in assembly), they can be opened by any type of debugger. The difference resides beyond the application, it utilize a "linker", a DLL (MSVBM60.dll) which contains all

the functions utilized by the VB. Thus when you make an operation using “compare” between strings for instance, it calls the function "vbaStrCmp" from the MSVBM60.dll. Because of this connection between the executable and DLL they normally (when compiled in Native Code) are more difficult for debugging. To convert assembly codes generated by the VB is necessary a little more time because they are more difficult, but it has a feature which facilitates the editing. All the information about controls, as state, name, color, text and size are stored into the executable as text. That means that if we search for text of buttons using a hex editor (or notepad.exe) we can change its basic structure. Our target enables a Command Button as you can see; now open it using a hex editor.



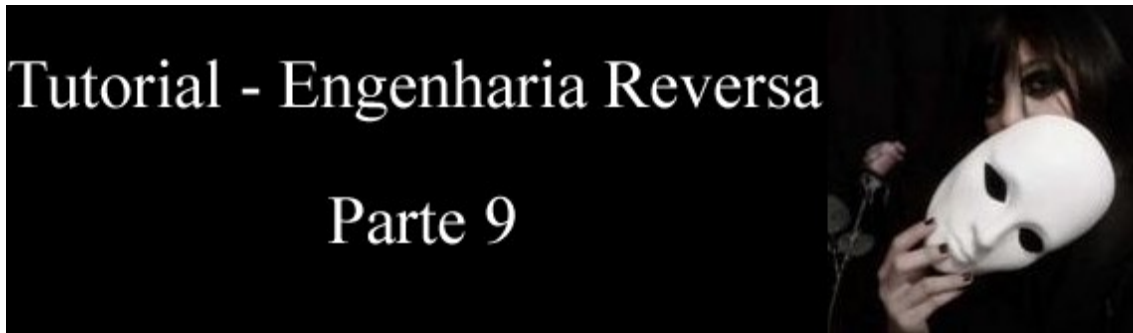
As I said the program has information of controls inside of it. Let’s find the text of the button (Active-me). Go to “search -> Find Text”. Select the ASCII mode and find the word “Active-me”. We are in the offset 125A (line 1250, column A).

```
00001250 | 41 74 69 76 65 00 04 01 08 00 41 74 69 76 65 2D | Ative....Active-
00001260 | 6D 65 00 04 3C 00 2C 01 C7 0B EF 01 08 00 11 00 | me.<...Ç.i....
```

In the center you can see the hex values of each byte of the file, to the right the ASCII corresponding to these values. Notice in the numbers 08 00. The “state” of a button follows after the height, followed by the byte 08 (ALWAYS). The height of the button is indicated by the "EF 01" (in reverse order of byte 01EFh = 495d), followed by the byte 08 and 00. The number 00 indicates that the button was disable (Enabled = False), and 01 indicates an active button (Enabled = True). To enable just change the 00 to 01.

Antes	Depois
EF 01 08 00	EF 01 08 01

Save the file using the WinHex ("File -> Save") and run our target again to see the final result.

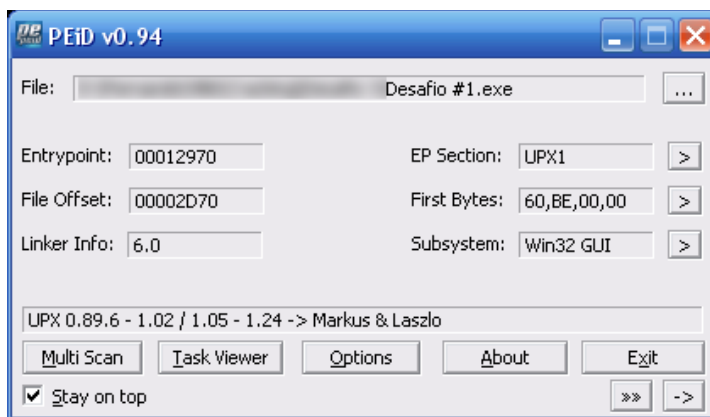


Solution for defiance one

At this moment I'll explain one of the two methods to elucidate the first challenge (can be seen in the table of contents - index). Open the file to make a general analysis of it. As you can see the verifier button is disable. When we close the program a nag screen is showed, we need to remove it. Let's see if there is some packer in it.

Part 1 – Unpacking

Open it using the PEiD to see that it was compressed with the UPX (Ultimate Packer for eXecutables).



If you read the tutorials the process will be easy. Firstly unpack the file, open it with Olly. The entry point of the packer is positioned (See also the PUSHAD). Press F7 to see the register ESP modified, use the right button on the value of the ESP and select "Follow in Dump". We are monitoring the address stored by the ESP in the dump window (Olly). Go to the first byte in the dump (38), select "Breakpoint -> Hardware, on write -> DWord".

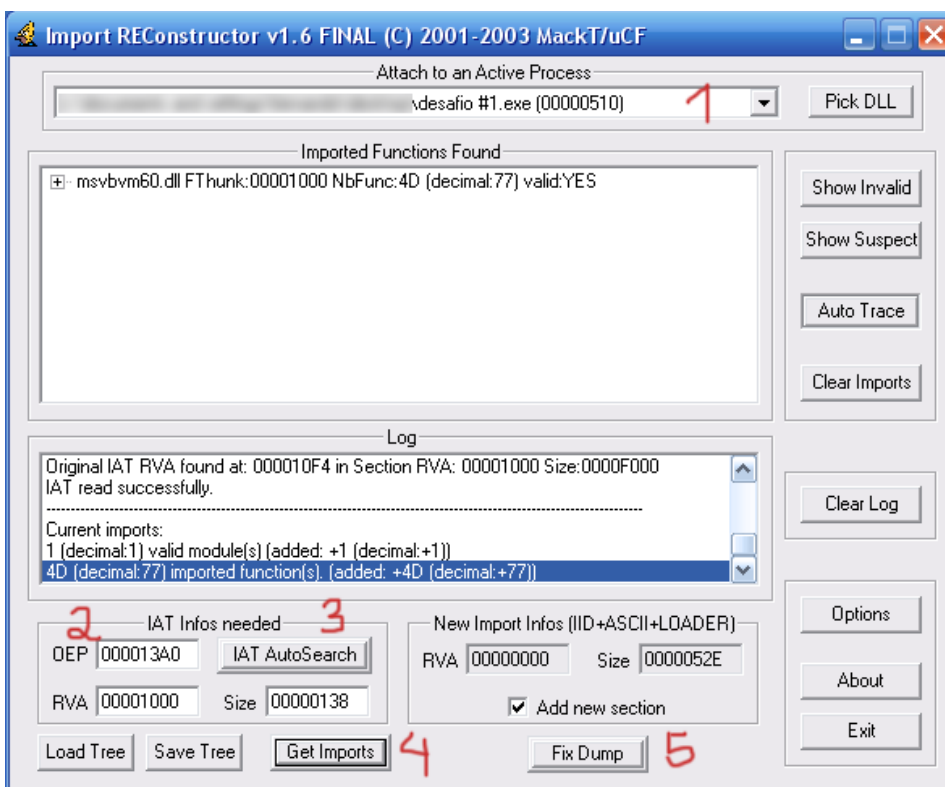
When the target writes something in this address the Olly stops the program, after that we are on the final address (near of it). F9 to continue the process.

```

00412AF6  61          POPAD
00412AF7  8D4424 80   LEA EAX,DWORD PTR SS:[ESP-80]
00412AFB  6A 00      PUSH 0
00412AFD  39C4      CMP ESP,EAX           ; Paramos aqui
00412AFF  75 FA      JNZ SHORT Desafio_00412AFB
00412B01  83EC 80    SUB ESP,-80
00412B04  E9 97E8FEFF JMP Desafio_004013A0 ; Como visto no tutorial de UPX, este jump nos leva ao EP original do programa

```

To make the dumping (unpacked program) we need to execute this last jump. Add a breakpoint (F2) in the last jump (00412B04) and press F9. Great! We stopped in the jump, press F7, we are in the beginning - OEP (Original Entry Point) of the unpacked program. Let's extract the unpacked program and send it to an executable file. In the OllyDump go to "Plugins -> OllyDump -> Dump Debugged Process" and after that click in "Get EIP as OEP" and unset "Rebuild Import". Now save the file. Let's remount the "base of imports", to do that let's use the ImpRec (don't close the olly).



In the step 5 select the file created by the olly. The final executable file is named "DUMPED_.exe", it is unpacked (extracted from the memory). To know more about the process step by step please see the chapter #six. This chapter is a simple overview which you can use as a fast catalog, as you can see in the subsequent chapter.

Part 2 - Enable controls of the programs...

Open the DUMPED_.exe with the WinHex and search for the ASCII ("Search -> Find Text") corresponding to the disable button: "Registrar". We are in the Offset 6CC3.

```
00006CC0 | 01 09 00 52 65 67 69 73 74 72 61 72 00 04 54 06 | ...Registrar..T.  
00006CD0 | EC 04 EB 05 77 01 08 00 11 04 00 FF 03 1F 00 00 | i.ë.w.....ÿ....
```

The important bytes are marked with the color violet. The byte 08 indicates the feature "Enabled" and the 00 indicates "disable" (Enabled = False). To revert that feature change the bytes 08 00 -> 08 01 and save the file with the WinHex ("File -> Save"). For more information read the chapter #eight

Part 3 – Removing protection against the Olly

Open the DUMPED_.exe modified, a message saying about "EP out of module of the program" is displayed. In some cases the process becomes difficult because of that, in our case the unique problem is that the program doesn't allow to save all the modifications we made to the executable, just the selections. Press F9 again to see a text message indicating that we are using the Olly, disable it. Restart the target using the Olly (Ctrl + F2) and type "bpx rtcMsgBox" in the command line ("Plugins -> Command Line"). That command sets a breakpoint in the call of the function MsgBox. Start the execution (F9), the Olly stops where the message "Anti-Olly" appears (4008B3A). We have to find a method to remove the message. Notice in the address 408AF1, we have an additional jump instruction which jumps to a region after showing up the msg. If the jump is realized the MsgBox disappears. We can force the program to jump "always" to another region of memory. Just modify the JNZ -> JMP:

```
00408AF1 75 6E JNZ SHORT DUMPED_.00408B61
```

Change for (select the line and press space):

```
00408AF1 EB 6E JMP SHORT DUMPED_.00408B61
```

Select the modified line with the right button of the mouse and move the cursor position to "Copy to Executable -> Selection" ; P, in the next window click again using the right button and follow to "Save File", select the executable (DUMPED_.exe). Now press Ctrl+F2 to reset the application. Cracked...

Parte 4 – Finding the correct password...

Type "bpx __vbaStrCmp" in the Command Line to set a breakpoint in the function that does operations between the data. Write something more than 4 characters and a serial number (I wrote F3rGO! and 132456), go the "Registrar".

We stop here after pressing "Registrar":

```
00408621 51 PUSH ECX
00408622 52 PUSH EDX
00408623 FF15 8C104000 CALL DWORD PTR DS:[&msvbum60._vbaStrCmp] msvbum60._vbaStrCmp
```

The program pushes two arguments and after that call the function. These arguments will be compared (in this case the program will compare ECX with EDX). See the registers (to the right). In my case, ECX holds 123456 and EDX holds 418. The program compares 123456 (our false serial) with 418 (the true serial, generated according to the name). Bingo! The serial to the name F3rGO! is 418.

Part 5 - Nag Screen

We found the serial number yet, now we will remove the Nag-screen which is showed when the program terminates. Type again "bpx rtcMsgBox" to set breakpoints in the places where the function "rtcMsgBox" is called. The olly stops before the execution, now fill with NOPs the lines referring to the MsgBox.

```
00408C51 50 PUSH EAX
00408C52 8D55 CC LEA EDX,DWORD PTR SS:[EBP-34]
00408C55 51 PUSH ECX
00408C56 52 PUSH EDX
00408C57 8D45 DC LEA EAX,DWORD PTR SS:[EBP-24]
00408C5A 6A 10 PUSH 10
00408C5C 50 PUSH EAX
00408C5D FF15 5C104000 CALL DWORD PTR DS:[&msvbum60.rtcMsgBox] msvbum60.rtcMsgBox
```

The line: 'CALL DWORD PTR...' in the address 408C5D. There are five arguments (as you can see...), select "CALL DWORD..." (408C5D) and go to "Binary -> Fill with NOPs". Fill with NOPs also the arguments of it...

```
00408C51 90 NOP
00408C52 8D55 CC LEA EDX,DWORD PTR SS:[EBP-34]
00408C55 90 NOP
00408C56 90 NOP
00408C57 8D45 DC LEA EAX,DWORD PTR SS:[EBP-24]
00408C5A 90 NOP
00408C5B 90 NOP
00408C5C 90 NOP
00408C5D 90 NOP
00408C5E 90 NOP
00408C5F 90 NOP
00408C60 90 NOP
00408C61 90 NOP
00408C62 90 NOP
```

Save the modifications, using the SHIFT button select the address 00408C51 until the 00408C62. Click using the right button on the selection and follow to "Copy to Executable -> Selection", use the right button again and go to "Save File". As I said this chapter is a simple overview.

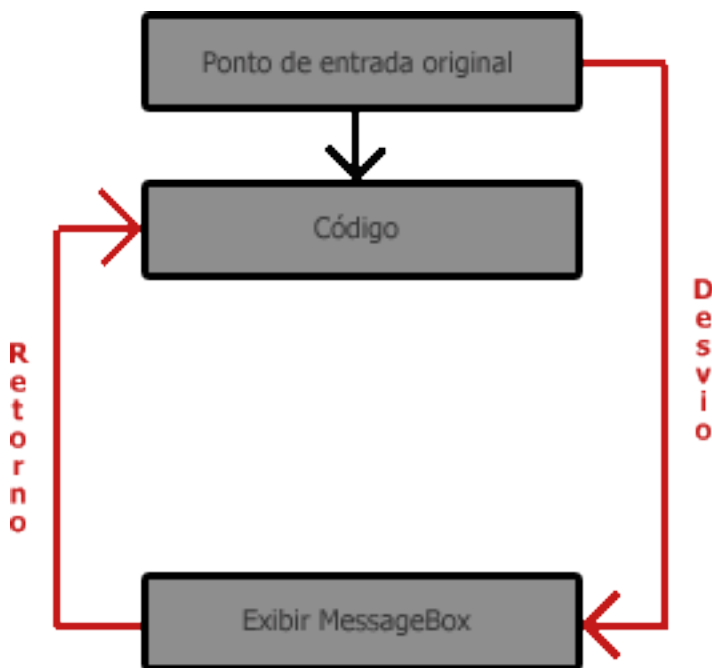
Tutorial - Engenharia Reversa

Parte 10

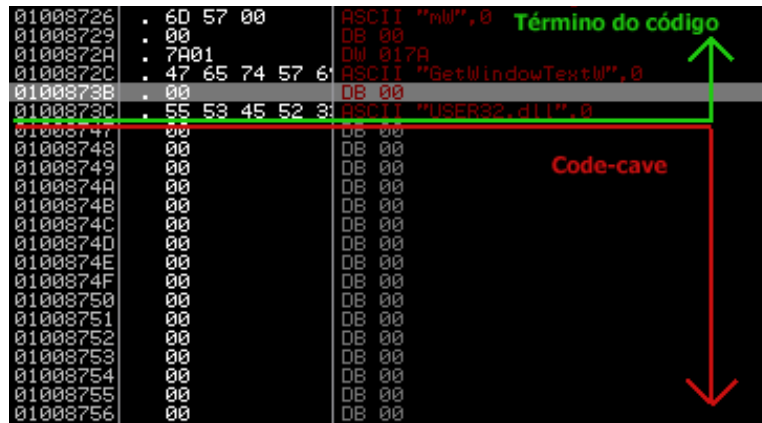


Inserting codes

Let's see how to inject new codes into the compiled executable file. We will inject codes into the notepad.exe. Our goal is to inject a message box into the program, is just an easy demonstration. We must redirect the beginning of the code to another region of memory, display a message and after that go back to the starting point. That's as easy as stupid (The simplicity is dangerous ;)



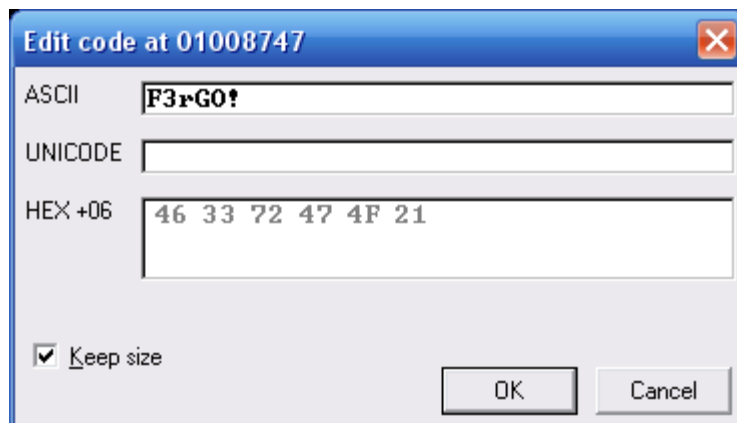
So to do that we need to find a section in the notepad.exe, a void place which does not have power to modify our program, that's called "Code-cave". Open the notepad.exe using the Olly and find an offset like that (for instance: 8747), there are many bytes 00 in this offset as you can see, these null bytes compose the "Code-cave" of our program which can be found frequently in programs. These procedures are easy to work with.



We will use this region above to add a message. See the parameters of the function MessageBox...

```
int MessageBox (
    HWND hWnd, // handle of owner window
    LPCTSTR lpText, // address of text in message box
    LPCTSTR lpCaption, // address of title of message box
    UINT uType // style of message box
);
```

To call a function we need to throw on the stack 4 arguments, hWnd indicates the owner of window, lpText and lpCaption correspond to the text and title being used, finally the uType indicates the style of message (Yes/No, Ok, Icons, etc...). We need to insert the text and title of the window in some place, in the code cave! ;P Firstly the title. Select the amount of space to store the title, each line occupies 1 character (one byte) in the memory. I'll use the title "F3rGO!" which contains 6 characters. Select the lines referring to the string using the SHIFT button (of course), click on the selection using the right button and after that go to "Binary -> Edit", mark the checkbox "Keep Size" and type your message in ASCII, the statement may look like this:



Press OK to see strange instructions in the dead listing. Press 'Ctrl+A' and the Olly will analyse the code again and you will see the correct instruction in the dead listing. To insert the text is the same procedure. Select the lines after the title. I inserted the text "Notepad modificado!" which contains nineteen (19) characters. The spaces count as characters too (of course). In the dead listing you'll have something like this:

```
01008738 . 00 DB 00
0100873C . 55 53 45 52 3: ASCII "USER32.dll",0
01008747 . 46 33 72 47 4: ASCII "F3rG0!",0
0100874E . 4E 6F 74 65 7: ASCII "Notepad modifca"
0100875E . 64 6F 21 00 ASCII "do!",0
01008762 00 DB 00
01008763 00 DB 00
```

The strings are located in the code cave yet (The yard of the wishes ;). Insert the function... The arguments are pushed in reverse order, we need to push the items on the stack firstly in this order: uType -> hWnd. Choose a offset of the CV, I used 01008763 and write the code indicating the addresses of the title and text (01008747 and 0100874E in my case). To write the code just click two times on theline and insert the command (As you can see the comments are in brazilian portuguese).

```
01008747 . 46 33 72 47 4: ASCII "F3rG0!",0
0100874E . 4E 6F 74 65 7: ASCII "Notepad modifca"
0100875E . 64 6F 21 00 ASCII "do!",0
01008762 00 DB 00
01008763 6A 00 PUSH 0
01008765 68 47870001 PUSH NOTEPAD.01008747
01008767 68 4E870001 PUSH NOTEPAD.0100874E
01008769 6A 00 PUSH 0
01008771 E8 957DD576 CALL USER32.MessageBoxA
01008776 00 DB 00
01008777 00 DB 00
01008778 00 DB 00
```

```
uType -> 0 -> Somente um botao OK
ASCII "F3rG0!"
ASCII "Notepad modificado!"
hWnd -> 0 -> Numero da janela dona da msgbox
Chama a funcao da msgbox
```

We need to redirect the beginning of the program 01008763 to the call of the function MessageBox and after that we'll need to return to the beginning of the program. For a while we will not add a JMP of return after the call, I 'll explain that later. See the address 01008763, memorize it, we'll redirect to it (the beginning of the function); right button "Goto -> Origin *". We are in the **EntryPoint** of the program.

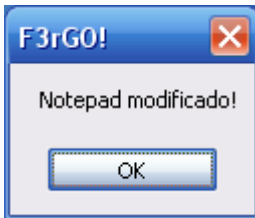
```
0100739D 6A 70 PUSH 70
0100739F 68 98180001 PUSH NOTEPAD.01001898
010073A4 E8 BF010000 CALL NOTEPAD.01007568
010073A9 33DB XOR EBX,EBX
```

Modify the PUSH 70, insert a um jump to the address 01008763. Before modifying save/memorize these two commands (PUSH 70 e PUSH NOTEPAD...). Modify the instruction 'PUSH 70' to the instruction 'JMP 01008763' (remember to use the option 'Fill with NOP's '). The redirection was made, but we lost two commands because of the jump occupies the space of the instruction PUSH 70 and part of the other PUSH, thus filling with NOPs the remaining bytes. We must add these two lost commands correctly. Where? In the code cave friend, dantan! ;P After the call to the function MessageBoxA.

Save the offset of the 'CALL NOTEPAD.01007568' (010073A4), after the message we will jump to this address. Get back to the region of the MessageBox and add the two commands (they were memorized ;) . As you will see in the screen we push the addresses which the strings are located. Insert a new instruction JMP to the address of memory 010073A4.

01008720	. 47 65 74 57 61	ASCII "GetWindowText",0	
01008730	. 00	DB 00	
0100873C	. 55 53 45 52 31	ASCII "USER32.dll",0	
01008747	. 4E 60 72 47 41	ASCII "F3rGO!",0	
0100874E	. 74 65 71	ASCII "Notepad modifica"	
0100875E	. 64 0F 21 00	ASCII ".dot",0	
01008763	00	DB 00	
01008765	6A 00	PUSH 0	uType -> 0 -> Somente um botao OK
01008766	68 47870001	PUSH NOTEPAD.01008747	ASCII "F3rGO!"
0100876A	68 4E870001	PUSH NOTEPAD.0100874E	ASCII "Notepad modificado!"
0100876F	6A 00	PUSH 0	hWnd -> 0 -> Numero da janela dona da msgbox
01008771	E8 95700576	CALL USER32.MessageBoxA	Chama a funcao da msgbox
01008776	6A 70	PUSH 70	Primeiro comando movido
01008778	68 98180001	PUSH NOTEPAD.01001898	Segundo comando movido
0100877D	^E9 22ECFFFF	JMP NOTEPAD.010073A4	Retorno ao endereço de execucao normal
01008782	00	DB 00	
01008783	00	DB 00	
01008784	00	DB 00	

Save your file now "Right button -> Copy to Exetuable -> All Modifications -> Copy All -> Botão Direito -> Save File" and test it.



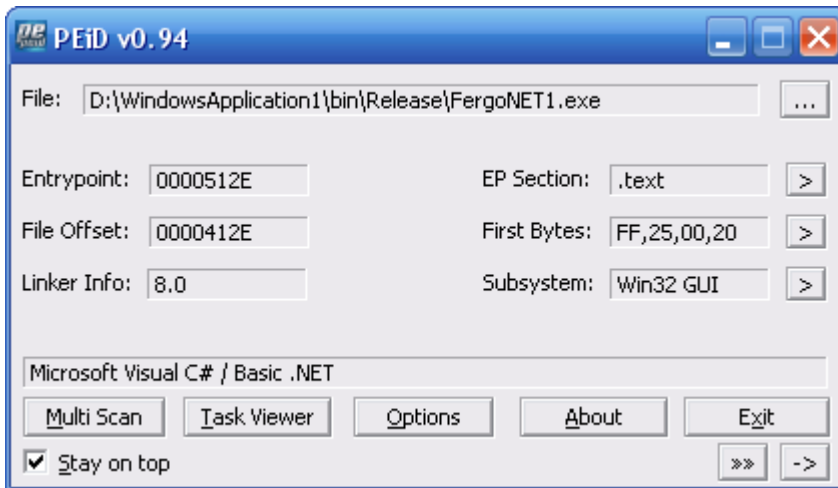
Revert .Net Applications

Let's analyze an application in '.NET'. This kind of application facilitates the reverse engineering, we can easily see the source codes. These kind of executables aren't compiled in native code (converted in assembly), instead it uses a language called IL (Intermediate-Level), that is the "disassembled code" structured in units in order to allow easy access to each line (one command per line). When you run this kind of program the Framework need to compile it in execution-time (on the fly), because of that the programs compiled with this Framework are slow.

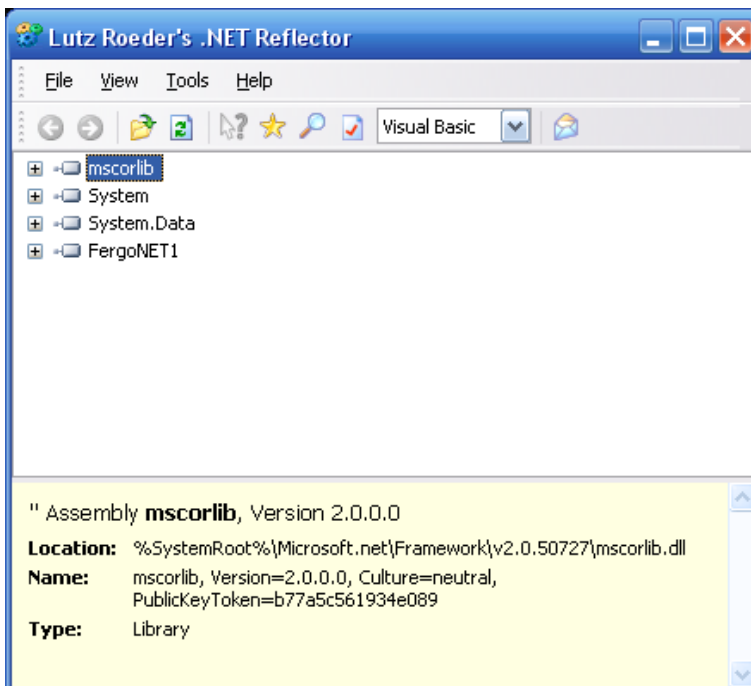
The language is not import (VB, C#, ...), the IL-Code is the same. .NET Fuscator (Obfuscator) doesn't solve the problem, there are many application able to convert "criptografy". We will just analyze the code. Download your copy.

-Download: [fergonet.zip](#)

The Olly cannot convert applications .NET, let's use a program called [.NET Reflector](#). This program converts the codes in native langue. As always see in the PEiD more informations about the executable file.

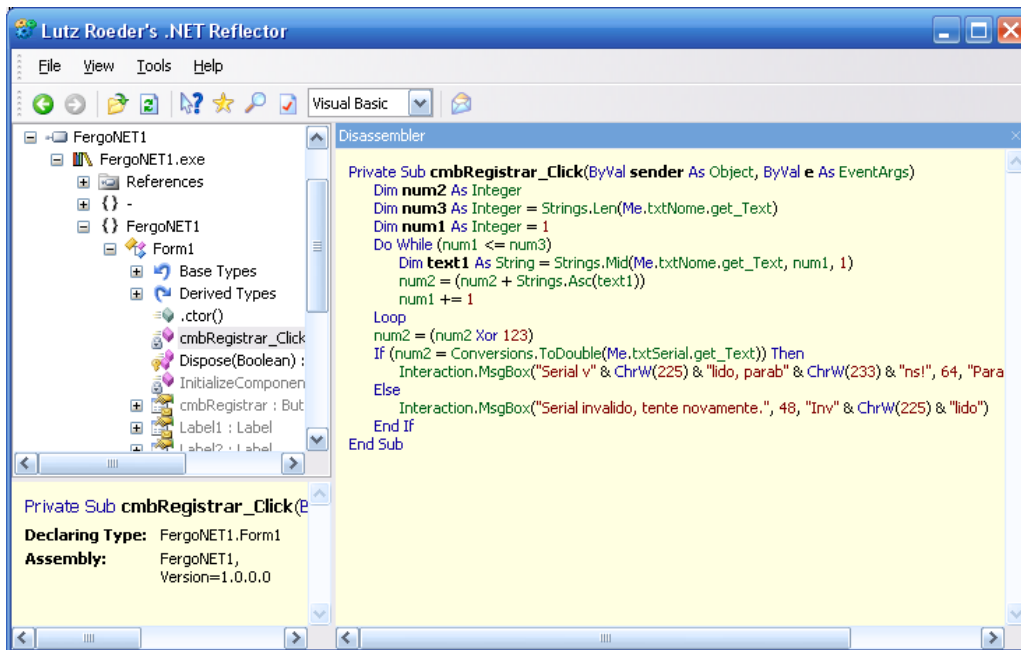


Start the .NET Reflector and open our program inside of it (if necessary select the version 2.0).

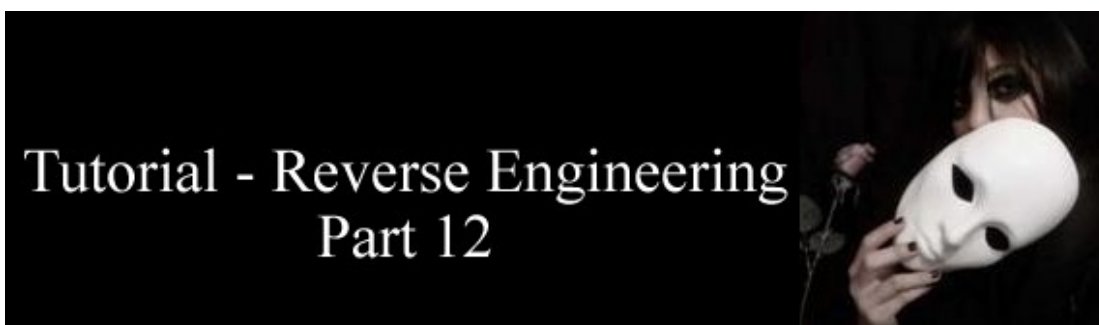


Expand the tree until Form1, which contains all the events, controls, and some more informations (FergoNET1\FergoNET1.exe\FergoNET1\Form1). That's interesting, we can see all controls and events (inclusive with its original names). Between the elements of the tree we can see: cmbRegistrar_Click(Object, EventArgs) : Void

Click with the right button on it and select "Disassembly" (if necessary click on 'Skip' in a message box). The codes are there, written in native language (you can select one in the drop-down – toolbar. I chose 'Visual Basic').



This little detail is simply incredible (None right reserved ;) ... Copy this code to the VisualStudio, make some basic modifications and you will have a keygen ; P The serial is generated because of the operation with the ASCII values (each character of the name), and by the XOR instruction. If you want to see the code in IL select the option IL in the drop-down ('toolbar'), the debugging becomes much more easy.



Addresses of memory

There is a difference between registers and memory addresses. The registers are little pieces of memory referring to the processor, they are used to store data temporarily.

In x86 processors they can store values of 32 bits (4 bytes) and are used basically for calculation during run-time of the programs. The memory addresses are responsible to store more data than that, like texts (strings).

```
00401000 B9 05000000 MOV ECX,5
00401005 BE 04304000 MOV ESI,pont.00403004 ; ASCII "abcde"
0040100A BF 0C304000 MOV EDI,pont.0040300C
0040100F 49 DEC ECX
00401010 8B1C31 MOV EBX,DWORD PTR DS:[ECX+ESI]
00401013 891F MOV DWORD PTR DS:[EDI],EBX
00401015 47 INC EDI
00401016 0BC9 OR ECX,ECX
00401018 77 F5 JA SHORT pont.0040100F
```

The first command moves the value 5 to the register ECX, in the second one (MOV ESI,pont.00403004) the 'MOV' moves to the register ESI the value 00403004. The Oly identifies a memory address (for that reason you will see -> "pont.00403004"); this address is pointing to the first byte of the string "abcde":

Endereço	Conteúdo (ASCII)
00403004	a (97)
00403005	b (98)
00403006	c (99)
00403007	d (100)
00403008	e (101)

As you can see ESI is pointing to the address 403004 (beginning of the string "abcde") and EDI -> 40300C which is a void space. The value of the ECX is decreased; the value was 5 and now -> 4.

```
00401010 8B1C31 MOV EBX,DWORD PTR DS:[ECX+ESI] ; sentenca 1
```

"DWORD PTR" indicates that EBX is receiving the content of the address pointed (ECX+ESI). The values between '[' and ']' indicate the address, in this case ECX+ESI. We saw that ECX holds 4 and ESI holds 00403004, the first byte of the string "abcde".

∴ 00403004 + 4 = 00403008.

The register EBX is receiving the contents stored in the address 00403008, the character "e" (101), EBX = 101.

```
00401013 891F MOV DWORD PTR DS:[EDI],EBX ; sentenca 2
```

As we saw the register EDI points to a void place in the memory 0040300C; that address will receive the value stored in the register EBX (101). The address 0040300C contains the number 101.

The instruction 'MOV EDI, EBX' moves to the register EDI the value stored in the register EBX and not to the memory address pointed by the EDI. The register EDI is increased (EDI++); the EDI had the value 0040300C, now the value is 0040300D. In the next line is realized a logic operation OR with ECX (4) to verify if the register ECX is null; if the answer wasn't true the process is repeated using modified values. The register ECX is decreased one more time (4 -> 3), ECX+ESI, instead 00403004 + 4 -> 00403004 + 3, corresponding to the address which the character "d" (100) is located. The value is stored in the address pointed by the register EDI, now -> 0040300D. The algorithm simply inverts the string, inserts the result in a place of the memory. The procedure starts moving the content of the address 403008 to the address 40300C.



Thanks to: |



- #include <x00. <http://www.milw0rm.com>.....
- #include <x01. <http://www.packetstormsecurity.com>.....
- #include <x02. <http://www.thebugmagazine.org>.....
- #include <x03. <http://www.kernelhacking.com>.....
- #include <x04. <http://jbrowsec.blogspot.com>.....
- #include <x05. <http://www.elhacker.net>.....
- #include <x06. <http://undergroundsecurityresources.blogspot.com>.....
- #include <x07. <http://www.hackemate.com.ar/e-zines/fatal/index.html>.....
- #include <x08. <http://www.hackpalace.com/e-zines/BR-axur05/index.html>.....



I like to remember in my spare time (my waste of time), the first season of my heart, my first blow, my efforts, the nostalgia never die and the time belong to us (my love)...

[]'s

By

6_B14ck9_f0x6 – Viper Corp Group

Se eu quizer voltar a fazer defacements, nao vai ser pq os outros acham q isso eh errado que eu nao vou fazer. Eu crio minhas proprias leis eticas ao contrario de uns que simplesmente se deixam ser controlados pelo unsexurity team. Pergunta: Se eles sao tao certinhos assim, por que usam mascaras nakele video etico deles?? Do que tem medo se eles sao os kra do bem? Eles devem simpatisar com terroristas e se escondem debaixo desta etica tosca. Ou estavam com medo de algum conhecido assistir akilo e ficar com vontade de espanca-los (como todos os q assistiram). Nos meus textos voce nao encontra coisas do tipo: "Se voce eh um scriptkiddie entao va se fuder e nao leia isso". Isso pq eu nao me considero superior a ninguem que esteje lendo o que escrevi, portanto nao posso dizer quem deve e nao deve ler isso. Se os sk nao lerem, nunca vao aprender para deixarem de ser kiddies. Eu nao falo nos meus textos o que voces devem ou nao fazer para serem considerados hackers. Eu simplesmente tento explicar tal assunto e voce use do jeito que achar melhor. Pra q ser hacker?? Voce vai fazer tudo direitinho como manda o nashleon pq se nao vai ser considerado hacker. Se nao voce vai ser conhecido como lamer, ou sk, ou vibora, etc... Isso sao apenas rotulos que algum imbecil inventou para classificar o nivel de conhecimento, mas esta sendo usado para classificar o nivel etico das pessoas. Entao se essas palavras ja perderam totalmente o sentido pra que dar importancia a elas? Hacker ja lhe pareceu coisa seria alguma vez? Provavelmente sim. Mas todo o respeito que voce ja teve por esta palavra, esqueca. Por que hoje os hackers sao akeles escravos de uma etica ridicula, ou akelas crianças que brincam com exploits em perl que rodam em windows. O termo hacker nao impoe mais o respeito de qdo nos eramos novatos e nao conheciamos direito a comunidade. Agora "ser hacker" eh motivo riso. O povo sabe que isso eh fato e querem mudar isso impondo regras eticas para ser classificado como hacker. Nao sei pq dessa necessidade de querer ser conhecido como hacker. Sera que tem gente q nao dorme direito enquanto nao provar para os outros que eh um hacker? A um tempo atraz o kra queria provar q era hacker então fazia um deface barato num .org winnt e durmia direito. Mas agora isso eh coisa de lamer. Ser hacker agora eh ficar na sua e aprender coisas para nao ter que usar. Devem achar que eh como karate. Voce aprende a lutar para se defender e nao para atacar. Entao se um loko espanca sua mulher, voce espera ate ele te atacar para dar uma surra nele. Besteira! Se voce sabe lutar, lute! Se sabe hackear, hackeie! Faca o que quizer com o conhecimento que voce adquiriu com seu suor. Depois arque com as consequencias mas nao deixe que alguem lhe diga o que eh certo e o que eh errado. Tenha seu proprio codigo de honra. Este eh um dos poucos textos em que tento conscientizar o povo. So escrevo isso pq muitas pessoas andam se influenciando demais pelo hacker etico imposto pela unsexurity. Se voce apoia as ideias nazistas e ego-centricas do nashleon eu nao tenho nada a ver com a sua vida. Faca o que quizer dela. Novamente, soh estou escrevendo isto porque o pessoal da unsek me baniram do knal sem me conceder direito de resposta as criticas que fizeram nas minhas costas. Mas isto eh etico para eles. Nem todos os membros da unsexurity sao troxas, nao preciso dizer a quais este texto se refere. Ae... consegui o log daquele dia. Tirem suas proprias conclusoes...
Nota: cortei umas partes do log para vc nao ter que ler merda e encher o saco.

6_B14ck9_f0x6 says: Ui, medo.

0x00000002

(#behindthescene.Unsecurity)

```
<module> agora vou abrir
<module> um pouco
<coracaodeleao> quero meu direito de resposta..:)
<module> pra o pessoal dar opiniao propria
<module> por favor
<module> nash
<module> vc vai ter
<module> calma kct
<module> tentem nao zonear
<module> please
*** module sets mode: -m
<Emmanuele> mandei minhas opinioes pra list hoje..e acho que chega d
    explicacoes e replicas e treplicas. vamos ao unsek com
    cara mais transparente e regras novas.
<xf> eh bom diminuir o tempo de +m
*** flash is now known as flash_crakao_do_mal
* snape ugh!
<xf> Emmanuele isso!!
<hts> mais tolerancia.
<Blind_Bard> voces ja leram "os meninos da rua paulo"? =DDD
*** snape is now known as snape_banda_podre
<module> ae, esse snape nao eh um defacer?
<hts> cada lado aumenta e diminui
<hts> quando covem..
<module> oq ele faz aqui?
<hts> e isso nunca acaba
<xf> hahaha :) esses nicks tao ironicos
*** snape_banda_podre is now known as snp\defacer
<Sock> module deve ser
<xf> quem sao eles?
<snp\defacer> module, nao fale muito
*** cezinha is now known as cezinha_kiddie
<snp\defacer> pois ja vi falcatruas suas tb
<snp\defacer> :)
*** snp\defacer is now known as snape_do_mao
<xf> a banda podre veio participar tb
<module> snp\defacer: tu eh um lame meu pobre
*** hak is now known as hak_podre
<flash_crakao_do_mal> eh
*** snape_do_mao was kicked by module (module)
*** module sets mode: +b *!*@200.193.115.UNsecurity-secures-me-3629
<Blood_Sucker> eu acho q estamos aki a 3 horas e ateh agora soh falamu
    de nash e blind
<cezinha_kiddie> ou
<cezinha_kiddie> bane eu tb
<cezinha_kiddie> vai
<cezinha_kiddie> vai tio
<cezinha_kiddie> hehe
<module> ae
<module> ok cezinha_kiddie
<Emmanuele> blind..vc botou sua merda na list e agora vem fazer
    discurso populista.vai pro pdt que fica melhor.
<module> infelicamente
<luciphher> module: lê o manual do ircd, existe um modo de canal que
    nao deixa mudar de nick :-)
<module> se vcs querem assim
```

0x00000003

<module> pra mim
<xf> hehehe seria falta de respeito.
<cezinha_kiddie> EH
<module> nao faz diferenca nenhuma
<module> =)
<cezinha_kiddie> HAIL HITTLER!
*** hts sets mode: +b *!*@200.173.116.UNsecurity-secures-me-16
*** cezinha_kiddie was kicked by hts (hts)
<nibble> Blood_Sucker: discordo
<hak_podre> aieuhoiueh
<Emmanuele> ban mesmo
<hak_podre> uau
<xf> vcs se delongam muito em papo burocratico
<hak_podre> lá vou eu também
<nibble> Blood_Sucker: mais concordo q perdemos muito tempo com isso
<module> hak_podre: se vc quiser..
<coracaodeleao> vcs tao vendo?
<alex_> Emmanuele o muieh.. vai defende seo mitnick
*** hts sets mode: -bb *!*@200.173.116.UNsecurity-secures-me-16
!@200.193.115.UNsecurity-secures-me-3629
<coracaodeleao> pq querem esse pessoal na scene?
<Emmanuele> hummmm
<Emmanuele> jah defendi
*** snape_do_mao has joined #behindthescene
<coracaodeleao> pessoal q nao tah nem aih?
<xf> ele ta livre
<module> poiseh nash
*** snape_do_mao is now known as snape
<module> nao sei quem quer nao
*** hak_podre is now known as hak
<hak> olha cara
<hak> vou dizer o que eu penso
<Emmanuele> to esperando outro bode expiatorio
<Emmanuele> ta a fim??
<hak> tudo isso acontece por falta de compreensao e tolerancia
<klogd> eu num quero, jah tem que sair tirando essa merda daqui
<alex_> Emmanuele nah... sou ex-programador php
<snape> dialogo eh a melhor solucao
<Emmanuele> alex..vc respeite ou caia fora
<alex_> Emmanuele pq vc traiu o dernerval?
<dum_dum> coracaodeleoa Eu acho q vc nao esta sendo muito Anti-Etico,
o Blind falou de vc com educaçao e sem te atingir
verbalmente
alias ate te elogiou! Ja vc pelo contrario o hcamou de
vibora e o atacou! Nao da pra para com essa CRISE ?? Pq nao
resolvemos isso como adultos coerentes e voltamos a ficar
todos em uma boa ?? :)
<Emmanuele> o q???
<Emmanuele> pirou??
<Emmanuele> derneval eh amigo que se hospeda na minha casa
<Emmanuele> que lama eh esta??
<module> puts
<alex_> sei la
<F22> eu tb acho que o blind e o nash devem ficar nuba boa
<alex_> ocuparam meo teclado ake
<Emmanuele> dum dum larga de ser infantil e vai ler a list
<snape> porque os dois nao conversam em pvt

0x00000004

<snape> e pronto
*** module sets mode: +m
<module> realmente
<module> eh complicado
<module> engracado
<module> tem um cara aqui do meu lado
<klogd> fala o q c q module?
<module> e disse "bah, nao sei como vc consegue ! eu faco isso pq eu gosto, to cagando e andando pra esse monte de ente brigando!"
<module> exatamente oq eu nao penso eh isso
<module> realmente
<klogd> eu to do seu lado tombem
<module> seria FACILIMO
*** module sets mode: -o klogd
<module> mto facil
<module> eu ME FUDER
<module> mas nao
<module> eu realmente
<module> pelo menos tento
<module> fazer algo
<module> pra todos
<module> claro
<module> que nao da pra ser perfeito
<module> ora
<module> quem sou eu
<module> mas eu prefiro ficar aqui
<module> no meio dessa guerra
<module> tentando ajudar de alguma maneira
<module> a ficar na minha
*** module sets mode: -m
*** ChanServ sets mode: +o klogd
*** klogd sets mode: +b *!*@*.250.193.UNsecurity-secures-me-12777
*** alex_ was kicked by klogd (klogd)
<snape> voces conhecem o ditado
*** klogd sets mode: +b *!*@*.177.52.UNsecurity-secures-me-12084
*** flash_crakao_do_mal was kicked by klogd (klogd)
<Emmanuele> nao admito que um cara qualquer venha falar de mim e de minha relacao com derneval
<snape> briga de marido e mulher, ninguem mete a colher.
<module> Emmanuele: quem falou?
<Emmanuele> chega de largarem mentiras e merdas no ventilador alheio
<klogd> quem mais vai ficar com graca
<Emmanuele> alex falou
<xf> pessoal, nada de ataques pessoais.
<Emmanuele> alias..quem eh este???
<module> ele saiu ja neh?
<module> poiseh
<module> ataque pessoal
<Emmanuele> ahan
<module> vai ser kill da rede
<Emmanuele> ataques pessoais sao o fim mesmo
<klogd> ** klogd sets ban on *!*@*.250.193.UNsecurity-secures-me-12777
<klogd> *** klogd has kicked alex_ from #behindthescene (klogd)
<klogd> jah se foi :)
<klogd> adoro isso
<module> puta meu
<module> sinceramente
<module> to cansado viu

0x00000005

<coracaodeleao> meu direito de resposta, please.
<Emmanuele> fiquei 4 anos de minha vida dando sangue e meu dinheiro por um hacker.nao admito falta de respeito com isto.
<xf> coracaodeleao manda ver
<Emmanuele> fale nash
<module> fala nash
<Emmanuele> to magoada. mas vou ficar ate o fim disto aqui.
<coracaodeleao> sete aih o +m +v
*** hts sets mode: +m
<module> haha, calma emma
*** hts sets mode: +v coracaodeleao
<hts> fale, mas nao fale muito.
<coracaodeleao> Emmanuele tem todo motivo para estar magoada tanto quanto eu...
<coracaodeleao> o unsek sao vcs, e eu chamo o blind de vibora
<coracaodeleao> pq pra mim eh isso q ele representa, jah q quem o conhece ha algum tempo
<coracaodeleao> sabe o quanto ele foi falso, mentiroso e astucioso.
<coracaodeleao> ele sabia da kimera desde janeiro..
<coracaodeleao> mas soh falou sobre ela quando por ironia do destino
<coracaodeleao> eu disse num pvt ao xf q jamais participaria de um projeto com ele(ou seja, ele nao iria entrar na kimera).
<coracaodeleao> no q se refere as normas...
<coracaodeleao> vcs querem gente como essas
<coracaodeleao> q ficam brincando e sao elitistas dentro do unsek?
<coracaodeleao> q nao se importam a minima
<coracaodeleao> q passam anos sem aparecer e quando aparecem, eh pra decidir o futuro do unsek?
<coracaodeleao> soh pq o blind as convidou?
<coracaodeleao> nao acho q eh por aih, pessoal..
<coracaodeleao> o unsek sao vcs.. e seu eu digo
<coracaodeleao> q sou diferente, eh pq eu nao quero
<coracaodeleao> amanha ser comparado com viboras, pessoas q prendem hackers eticos
<coracaodeleao> script kiddies q mudam hp ou destroem dados alheios..
<coracaodeleao> eles tem prejudicado a gente(hackers eicos) ha tanto tempo..
<coracaodeleao> pq permiti-los no nosso meio?
<coracaodeleao> sim.. eu sei q temos q concientizar esse pessoal..
<coracaodeleao> mas depois de concientizado eles sao bem vindo a opinar dentro do unsek..
<coracaodeleao> mas um cara q muda hp
<coracaodeleao> tem algum moral para defender o hacking?
<coracaodeleao> ou mesmo um elitista tem?
<coracaodeleao> sem lutamos contra a elitizacao, e isso q o blind disse
<coracaodeleao> de q eu sou a figura central...
<coracaodeleao> quantas vezes eu nao mudei de nick
<coracaodeleao> para evitar estar hj aqui falando isso?
<coracaodeleao> mas ele e outros sempre fizeram
<coracaodeleao> questao de atar meus nicks
<coracaodeleao> como uma corda q chega ao nash leon.
<coracaodeleao> e no q se refere a elitizacao..
<coracaodeleao> tem aqui um monte de testemunhas q viram quando
<coracaodeleao> eu me asfatei no meio do ano passado..
<coracaodeleao> o blind querer q os nicks voltassem para a pagina
<coracaodeleao> e q rolasse sim uma diferenciacao

0x00000006

<coracaodeleao> das pessoas.(q igualdade eh essa?)
<coracaodeleao> eu acho sim q o unsek eh bem maior q nash e blind..
<coracaodeleao> e q a scene nao precisa de nenhum de nos dois..
<coracaodeleao> mas o q eu quero deixar eh q
<coracaodeleao> se vc abre espaco a banda podre e aos kiddies..
<coracaodeleao> vai ocorrer o q estah correndo hj em maior escala..
<coracaodeleao> "a perda do objetivo original".
<coracaodeleao> sem mais, pessoal.
*** hts sets mode: -m
*** hts sets mode: -v coracaodeleao
<module> olha
<klogd> otimo
<module> acho que agora
<module> isso tem de terminar
<Emmanuele> faco minhas as palavras de nash
<module> os dois lados
<module> falaram
<module> antes
<module> eu acho
<module> que soh seria justo
<module> xf falar
<module> pq ele foi falado
<module> comentaram sobre ele
<module> e ele nao falou nada
<module> e olha
<module> depois dele
<module> o assunto termina
<module> pq temos
<module> que andar com o unsek
<module> nao ficarmos parados
<module> nos probelmas
<xf> +m +v
<module> isso
<module> da +v
*** hts sets mode: +m
*** hts sets mode: +v xf
<xf> vou falar pouco.
<lucipher> eu concordo com as palavras do nash, nao com as ofensas ao
Blind_Bard.. porque nao sei o que se passou
<xf> <coracaodeleao> ele sabia da kimera desde janeiro..
<xf> <coracaodeleao> mas soh falou sobre ela quando por ironia do
destino
<xf> <coracaodeleao> eu disse num pvt ao xf q jamais
<xf> <coracaodeleao> participaria de um projeto com ele(ou seja, ele
nao iria entrar na kimera).
<lucipher> ou seja, me mantenho neutro
<xf> primeiro, nao repassei oq vc falou pra mim pra ele
<xf> foi uma acusacao e suspeita injusta
<xf> nao manipulei ninguem para fazer nada
<xf> coracaodeleao eh melhor dar nome aos bois
<xf> e ser claro..
<xf> -v
<module> terminou xf?
*** hts sets mode: -v xf
<module> ok
<module> soh uma coisa
<module> pra terminar
<module> de vez
<module> soh uma coisa

0x00000007

<module> pro blind nao precisar esponder tudo
<module> e continuar com esse loop sem fim
<module> ele soh pediu
<module> pra avisar
<module> que nao convidou nnguem pro canal
<module> hj
<module> ele disse isso pra mim no pvt
<module> e pediu pra eu avisar isso
<module> agora
<module> vamos voltar ao unsek
<module> que eh o importante pra todos aqui
<hts> isso
<module> tem uns itens interessantes aqui
<module> soh q cada janela
<module> de pvt aqui
<module> tem menos de meio cm
<module> hehe
<module> se alguem tiver algo pra falar p.eqto
<module> pode falar
<module> em qto eu tento achar algo
<module> de util
<module> pra por aqui
<hts> enquanto isso
<module> pra serdiscutido
<module> ...
<hts> enquanto isso..
<hts> vamos voltar a questao que ainda nao foi resolvida ate agora...
<lucipher> posso falar o que eu acho ?
<hts> Temos que decidir a questao do website(o problema de limite de transferencia da virtualave).
<hts> se alguem tem uma sugestao pra esse problema
<hts> me mande no pvt.
<module> ah eh
<module> uma coisa q falaram mto
<module> tava ficando de lado
<module> forum deve voltar.. concordam?
*** hts sets mode: -m
<hts> concordo!
<Emmanuele> para mostrar como funcionam as coisas por aqui..preciso colar uma coisa
<module> concordo
<marcelo> c
<coracaodeleao> concordo, mas nao o q estah!
<module> Emmanuele
<Emmanuele> posso????
<module> concorda ou nao
<module> depois vc fala
<Sock> concordo
<klogd> concordo
<klogd> e bye
<klogd> vou nessa abraços, trampo SUX
<hts> ninguem tem sugestoes ?
<Blood_Sucker> falow klogd
<module> ok foru
<module> espera
<module> tem varias
<module> no pvt
<module> mas eh mta coisa

0x00000008

<module> dexa eu procurar
<xf> agree com o forum
<F22> concordo com o forum
*** klogd has quit IRC (Quit: [x]chat)
<dum_dum> concordo com FORUM
<struck> soh uma coisa... o assunto ja acabou mas... achu q mesmo sendo um grupo, vcs nao precisam seguir todos a mesma etica... 6 sao nazistas por acaso?
<coracaodeleao> sai fora struck script kiddie.
<struck> ui
<struck> medo
<coracaodeleao> vai mudar hp q eh a unica coisa q tu sabe fazer.
<Blood_Sucker> putz, olha quem ta no canal meu
<F22> ?
<xf> q tristeza
<F22> quem eh struck ?
<AnJiNh0`> onde chegamos
<Blood_Sucker> lucifer, da um jeito nisso meu
<AnJiNh0`> ..
<AnJiNh0`> struck: ...
<snape> deixem ele ae
*** hts sets mode: +m
<module> struck: parece que vc nao viu o comeco da conversa que eh basica pra que possa entender oq acontece
<module> entao please, nao opine
<module> vamos comecar com a aplicar um pouco das regras aqui?
<module> tem certas pessoas que todos sabem que sao completamente fora da nossa realidade
<module> da nossa intensao
<module> filosofica
<module> entao essas pessoas
<module> vou ser mais claro
<module> alguns mudadores de sites
<module> e afins
<module> devem ser banidos?
<module> concordam?
*** module sets mode: -m
<hak> nao
<surfer> sim
<F22> sim
<module> concordo
<Blood_Sucker> com certeza sim
<dum_dum> sim!
<Emmanuel> concordo
<hts> sim
<dum_dum> concordo!
<hak> lucifer: o cara, pode chingar meu irmao, mas nao minha mae.
<scuzzy> concordo
<hak> nao concordo
<coracaodeleao> concordo
<coracaodeleao> kiddie = ban!
<module> hak: uma resposa soh
<module> please
<module> ninguem mais opina?
<Blood_Sucker> nao soh os defacers, mas qualquer pessoa q se mostre contra o q pregamos e valorizamos
<marcelo> concordo!
<hak> module: foi mal

0x00000009

<hak> achei que nao tinha pego
*** snape was kicked by module (module)
*** snape has joined #behindthescene
*** module sets mode: +b *!*@200.193.115.UNsecurity-secures-me-3629
*** struck was kicked by module (module)
*** struck has joined #behindthescene
*** module sets mode: +b *!*@200.228.65.UNsecurity-secures-me-3592
<Blind_Bard> discordo... quem vai dizer o q eh certo pra eles?
*** snape was kicked by module (module)
<hak> penso a mesma coisa que o Blind_Bard
<hak> somos superiores a eles?
<coracaodeleao> q historia eh essa?
*** struck was kicked by module (module)
<hts> abrirao os olhos quando eles quiserem..
<coracaodeleao> negativo..
<xf> leiam a biblia
<Blood_Sucker> estaremos aki
<hak> hts: o que faz voce pensar que esta certo?
<surfer> talvez diferentes
<coracaodeleao> quando abrirem os olhos serao bem vindos.
<hts> hak, o que faz pensar que eles vao mudar ?
<Blood_Sucker> quando eles acordarem serao bem vindos
<hts> eles ja estao no nosso meio faz tempo
<hts> e nao mudaram.
<Blood_Sucker> sim, saun uns aproveitadores
<Blood_Sucker> q soh vem ateh o unsekurity pra conseguir informacao
<dum_dum> Eh verdade!
<coracaodeleao> eh isso q tem q ser coibido!
<coracaodeleao> como?
<coracaodeleao> com essas teias normas aih..
<coracaodeleao> quem for contra, ban!
<dum_dum> Podiamos banir os e-mails desses defacers q sao bem conhecidos neh ??
<Blind_Bard> ei...
<F22> sim
<coracaodeleao> um struck desse com infos farah
<hts> maillist privada ?
<Blind_Bard> mas se eles leem um texto
<coracaodeleao> bobagem amanha
<F22> tirar eles da lista ...
<Blind_Bard> de 100 paginas
<coracaodeleao> e nos seremos responsaveis
<Blind_Bard> pra fazer coisas
<dum_dum> Nao mail-list a mesma!
<F22> eh
<Blind_Bard> entao nao sao mais kiddies
<coracaodeleao> se formos coniventes.
<module> haha
<dum_dum> Mas e-maisl q sao vistos em defacer e estao na mail-list devem ser descadastrados! :)
<hts> acho que maillist privada eh melhor.
<Blood_Sucker> eu acho assim, quem demonstrar qualquer tipo de objecao àquilo q a gente prega, BAN!
<hts> teremos forum, canal e site.
<F22> eh
<hts> a maillist sera a unica privada.
<Blood_Sucker> hts, eu concordo
<coracaodeleao> nao se pode desvincular etica de técnica

<hts> e podera tb nao precisar mais de moderacao
<coracaodeleao> como tah sendo esse forum atual..
<coracaodeleao> quem nao deseja ler sobre etica nao merece
<Blood_Sucker> liberdade de informacao sim, mas tem limites!
<coracaodeleao> obter infos tecnicas.
<xf> vcs nao temem elitizacao?
<hts> nao vai ser elitizacao
<hts> vai entrar na lista quem faz parte da comunidade
<Blind_Bard> pergunto de novo
<Blood_Sucker> nao vai ser elitizacao, vai ser PRECAUCAO!
<hts> e nao quem quer apenas bisbilhotar
<F22> nao
<dum_dum> hts, mas privada ja eh eletizar! :)
<Blind_Bard> quem le um txt de 100 paginas
<Blind_Bard> pra fazer algo
<Blind_Bard> nao eh kiddie =)
<xf> devia ser aberto pra atingir mais gente
<hts> dum_dum, nao eh nao
<Blood_Sucker> prefiro ser chamado de elitista do q dar bomba na mao de terrorista!!!!
<dum_dum> Eu nao queria ser tao radicar na minha ideia, apenas banir os SK conhecidos! :)
<coracaodeleao> isso Blood_Sucker!
<lucipher> *** alex_ (mensan@200.250.193.UNsecurity-secures-me-12777)
Quit (Ping timeout: 180 seconds)
<lucipher> *** hak (demolay@200.250.193.UNsecurity-secures-me-12777)
Quit (Ping timeout: 180 seconds)
<Emmanuele> que abuso
<Blind_Bard> =DD
<F22> :)
<F22> eh mesmo
<lucipher> eu pus regra na firewall
<Emmanuele> o mesmo cara
<F22> hehehehe
<hts> dum_dum, eles entram e nao mencionam uma palavra
<lucipher> para o alex_ nao entrar
<lucipher> merda do servico nao funfa directo :-/
<Emmanuele> hak e alex a mesam coisa
<Emmanuele> impressionante..que cinismo
<dum_dum> hts, como assim entram e nao mencionam uma palavra ?
<module> <module> ok
<module> <struck> tipo...
<module> <struck> os q falam de mim sao soh os q nao me conhecem....
<module> <module> perae
<module> <module> lmin
<module> ae
<module> vo abrir aqui
<lucipher> <alex_> man
<lucipher> <lucipher> vai embora
<lucipher> <alex_> calma cara
<lucipher> <lucipher> voce ofende a emma... isto e' um servidor
<lucipher> <lucipher> para discussao de etica nao para elites
<lucipher> <alex_> po.. nao trocam conceitos tecnicos aki?
<lucipher> <alex_> cara
<lucipher> <alex_> ela ainda nao me deu um conceito sobre etica
<lucipher> <alex_> O_O
<module> pra ele contar a versao de historia dele
<module> no minimo
<module> vai mostrar a ideia de mais uma pessoa

0x0000011

<module> ok:?
<coracaodeleao> ou!!!
<hts> k
<module> fala
<coracaodeleao> vc nao conhece o struck?
<module> nao
<coracaodeleao> script kiddie
<lucipher> falaram
<coracaodeleao> q era da tdk
<Blood_Sucker> struck era dos knights pow
<Blind_Bard> =DD
<lucipher> que ele era elite
<coracaodeleao> mudou muita pagina...
<hts> eu conheco o struck de 1 ano atras.
<module> hmmm
<module> sim
<coracaodeleao> prejudicou legal o termo hacker..
<module> tipo
<module> nash
<coracaodeleao> olha lah no zine dele
<Blind_Bard> kiddie?
<module> pera
<coracaodeleao> o q ele chama de etica hacker..
<module> pera
<module> pera
<module> pperai
<module> dexa ele falar
<coracaodeleao> e verah q ele nao se importa a minima.
<module> soh pra gente ver
<module> oq ele fala
<module> ok?
<Blind_Bard> ahahahaha
<module> ele ta falando
<module> q nao eh isso q dizem
<xf> vcs tem medo de serem influenciados?
<module> ele ta dizendo aqui no pvt
<Blood_Sucker> meu, eu acho besteira perder tempo com ele
<xf> :)
<module> q nao eh isso
<Blood_Sucker> ele nao eh e nunca foi da unsek
<Blood_Sucker> pra q ouvirmos ele?
<Blind_Bard> meu... q engracado
<Emmanuele> perda de tempo
<hts> dexa ele entrar
<coracaodeleao> pra q isso?
<Emmanuele> jah chega por hoje
<module> sei ka
<hts> e vamos ouvir ele
<Blind_Bard> Blood_Sucker: boa tb!
<coracaodeleao> dar fama a esse pessoal?
<module> vcs decidam
<hts> faz 1 ano que nao vejo uma palavra dele :)
<coracaodeleao> ban logo...
<module> concordo com o hts
<hts> se for assim, me banam
<module> coracaodeleao
<module> sei la
<hts> eu ja fiz deface uma vez

<module> ok
<F22> gente
<module> entao nao?
<module> eu tenho q avisar pra ele
<module> hehehe
<F22> vamo decidir a scene aqui ...
<marcelo> otro "elite" ae: n4rfy
(bruninha@200.215.32.UNsecurity-secures-me-3469)
<Emmanuele> nao deixa nao..chega destes caras se metendo por aqui de repente
<F22> :/
<Blind_Bard> sim
<hts> deixa ele entrar.
<xf> module ouça vc entao e nao deixa ele entrar aqui.. mas se vc acha q deve ouvir, ouca
<Emmanuele> nao
<marcelo> ae
<module> marcelo: esse eh um kid
<marcelo> n4rfy.com.br
<module> lame
<r0oT> caraio
<Blind_Bard> cade a tal liberdade
<r0oT> ateh o narfy
<marcelo> tem as fotos desse zeh!
<r0oT> puts
<Blind_Bard> de expressao, cacete?
<module> marcelo: esse eh bostao
<Peace> posso perguntar ?
<module> nao esquenta
<Blind_Bard> nao falaram q o cara era kiddie?
<marcelo> hehehehe
<coracaodeleao> Blind_Bard: vc nao se importa
<Blind_Bard> deixa ele falar agora...
<hts> chega nash.
<coracaodeleao> pq vc nao se importa com o termo hacker..
*** module sets mode: +m
<module> chega nash
<module> chega blind
<module> os dois
<module> vamos fazer como tava antes
<module> o cara entra no canal
<module> pra falar o lado dele
<module> concordam
<module> ou nao
*** module sets mode: -m
<Emmanuele> nao
<module> me abstenho
<Blind_Bard> sim
<hts> sim
<coracaodeleao> nao
<xf> tenho medo de concordar pq posso ser expulso daqui
<module> haha
<xf> mas ele tem q falar.
<Emmanuele> nao tem nao
<F22> gente, e o site da unsek?
<F22> vamo ver ai
<Blood_Sucker> eu acho mais perda de tempo
<Emmanuele> ta fazendo o que aqui este cara???
<Emmanuele> vamos ver o site

0x0000013

<coracaodeleao> quero ver se ele vai pedir
<coracaodeleao> desculpas pelo menos.
<coracaodeleao> cade o outro? narfy?
<Blind_Bard> 3 x 2? struck fala?
<Emmanuele> q desculpas o que..pra quem?? vai se redimir?
<Emmanuele> q ingenuidade
<r0oT> narfy = bozo
<stdfk> coracaodeleao, voz
<r0oT> sai fora
<coracaodeleao> e aih?
<xf> vamos lembrar de jesus q deu chance aos pecadores
<coracaodeleao> vai deixar esse kiddie falar tb?
<bozo> q ceis tao falando
<Blood_Sucker> a gente, a gente ta perdendo mais tempo ainda discutindo SE VAMOS DEIXAR UM CARA FALAR OU NAO
<Emmanuele> sou contra
<xf> os pecadores nao podem se arrepender se nao derem chance
<coracaodeleao> soh querem sugar infos pra fazer defaces.
<F22> VAMOS OLHAR O NEGOCIO DO SITE DA UNSEKURITY !!!!!!!!!!!!!!!
<Emmanuele> chega de conversa firada
<coracaodeleao> deixa o cara falar, entao...
<Emmanuele> a reuniao nao foi feita pra kiddies
<coracaodeleao> veremos se vai pedir desculpas
<Blind_Bard> eles nao vao sugar info, vao falar...
<xf> se nao pedir ban
<coracaodeleao> por todo mau q causou ao hacking.
<Emmanuele> blind...eh tua turma?
<dum_dum> EH, esquece esses caras .. e vamo logo discuti logo, o resto q falta!
<Blind_Bard> =D
<Blind_Bard> pergunta a eles
<asap> me derrubaram
<asap> haha
<Blind_Bard> a quanto tempo
<xf> Emmanuele paciencia emma :*
<asap> nao eh legal
<Blind_Bard> nao os vejo
<coracaodeleao> como o blind
*** asap is now known as _module
*** bozo is now known as n4rfy
<F22> :/
<coracaodeleao> tao infiltraod apenas pq querem infos.
<Emmanuele> jah fui ofendida por um ai que usava dois nicks..chega
<_module> ae
<_module> podem banir ala vonte
<xf> Emmanuele eh :(
<F22> n4rfy es u tal juliano carneiro ?
<xf> tem razao
<Blood_Sucker> gente
<n4rfy> sou o jose augusto fernandes
<n4rfy> caso nao tenha visto no site :P
<AnJiNh0`> ...
<F22> ah ..
<xf> n4rfy q site?
<Blood_Sucker> f22, nao, mas eh da mesma raca do mesmo!
<xf> quem eh q vai perder tempo em site inutil?
<F22> ixi ...
<Blind_Bard> raca?

0x0000014

```

<F22> :/
<n4rfy> Blood_Sucker oi l33t0
<hts> calma vcs ai
<Emmanuele> hahahahhahhaa
<Blind_Bard> uhuuuu... o q o struck tinha falado sobre nazismo?
<n4rfy> hts :*
<Emmanuele> q bagunca
<xf> haha
<xf> :)
<Blood_Sucker> Blind_Bard, vc entendeu =]
<stdfk> parem! )=
<n4rfy> cade a bicha do struck ?
<stdfk> heh
<Emmanuele> blind bard..vc foi nazista e chamou o slater de facista
que eu me lembre
<dum_dum> n4rfy sai fora! Ow senao fica quieto! Tem um pah de coisa pra
discutir! E ate agora tao embaçando pra saber se entra ou nao
esses defacers! :/
*** _module sets mode: +b *!*@200.215.32.UNsecurity-secures-me-3469
<Blind_Bard> Emmanuele: quando fui nazista?
*** n4rfy was kicked by _module (bye)
<Emmanuele> portanto...alias..vc nao ia embora??
<Blind_Bard> uai
*** hts sets mode: +m
<_module> <n4rfy> te conheco ?
<_module> haha
<_module> eh hilario
<_module> eh hilario
<_module> como esses menininhos se acham os tais pq aparecem na midia
<hts> entao
<hts> Maillist Privada.
<hts> Quem concorda e quem discorda ?
*** hts sets mode: -m
<Blood_Sucker> gente
<dum_dum> Aleluia!
(...)

```

A partir dae comecaram a discutir os assuntos internos da unsek. Mas pelo visto minha opiniao tambem teve peso mesmo eu nao sendo do team. Agora entao vamos a algumas respostas: Eu nao me desculpo pelos dfcs que ja cometi, nem condeno nenhum deface de scriptkiddies. Voces estao tendo uma visao muito unilateral das coisas. Voces tem os SK como inimigos mas defendem os admin vagabundos que nao estudaram um minimo de seguranca e estao ganhando seus salarios deixando suas maquinas tao abertas que ate' as criancas conseguem entrar. Estes admins que deveriam ser seus inimigos pois eles estao tirando os seus empregos. Voces estao lendo e aprendendo seguranca antes mesmo de comecarem a trabalhar, enquanto esses caras, nem depois de estarem endo pagos para deixar o sistema seguro, fazem isso. E voces defendem estes e condenam as criancas que estao começando a aprender. Estes scriptkiddies podem ser irritantes pois acham que sao hackers somente executando programas de terceiros. Mas os admins invadidos nao aprenderam nem mesmo o que um scriptkid aprendeu e mesmo assim estao sendo pagos. Isso nao eh mais revoltante? Isso nao lhe da motivos para querer dar uma licao a este admin e faze-lo merecer seu salario ou ate' mesmo lhe tirar o emprego e dar a chance de alguem que estudou desde cedo possa receber algo por isso?

Este deveria ser o pensamento do pessoal que comecou a publicar os primeiros exploits: Criar scriptkiddies para aumentar a necessidade de admins competendes. Logo os scriptkiddies que estao fazendo o trabalho sujo para nos garantirmos nossos empregos. Mais um detalhe: Eu sou contra os grupos tentarem pregar a sua etica a todos. Os grupos deveriam ser criados a partir de membros que seguem as mesmas ideias e nao chamar pessoas para depois converte-las. Alem disso acho muito errado tentar catequizar a todos. Se forem liberar informacoes estejam cientes de que lamers, sks e defacers irão le-las e nao precisam concordar com suas ideias. Acho que a etica deve ser mantida ao grupo e nao a todos. Expor as ideias eticas tudo bem, mas nao tentem obrigar o publico a concordar com o que dizem. E qdo voces assumem essa possicao, de querer que todos tenham a mesma cabeça, qdo um fizer uma cagada a culpa sera jogada em cima dos hackers e nao apenas do grupo responsavel. Aceitem que pessoas tem ideias diferentes e nao precisam concordar com as de voces. Issu ae. Espero que a etica de voces seja repensada e criada em cima de um pensamento em comum e nao apenas dos membros de maior importancia ao grupo, pois como nota-se no log ainda ha mta oposicao no grupo.

<-> **End Of File** <->

6_Bl4ck9_f0x6 says: Entao manoh, vou mandar meu aval aih pra galera...

Nome: Glaudson Ocampos da Silva
A.k.a: Nash Leon (Coração de leão - tiozim...)
Data Nasc.: 29/05/1978
CPF: 88771369368
RG: 99010363520 SSP-CE
Data Nasc.: 29/05/1978
Titulo de Eleitor: 50241890736
Carteira de Motorista: CE111313104 Validade 26/06/2012

Mãe: Felciana Ocampos da Silva
Pai: Salatiel Tavares da Silva

Endereço: Rio Parnaíba 191 - Barra do Ceara
CEP 60330020
Cidade: Fortaleza CE

0x0000016

Tutorial - Engenharia Reversa

Parte 1

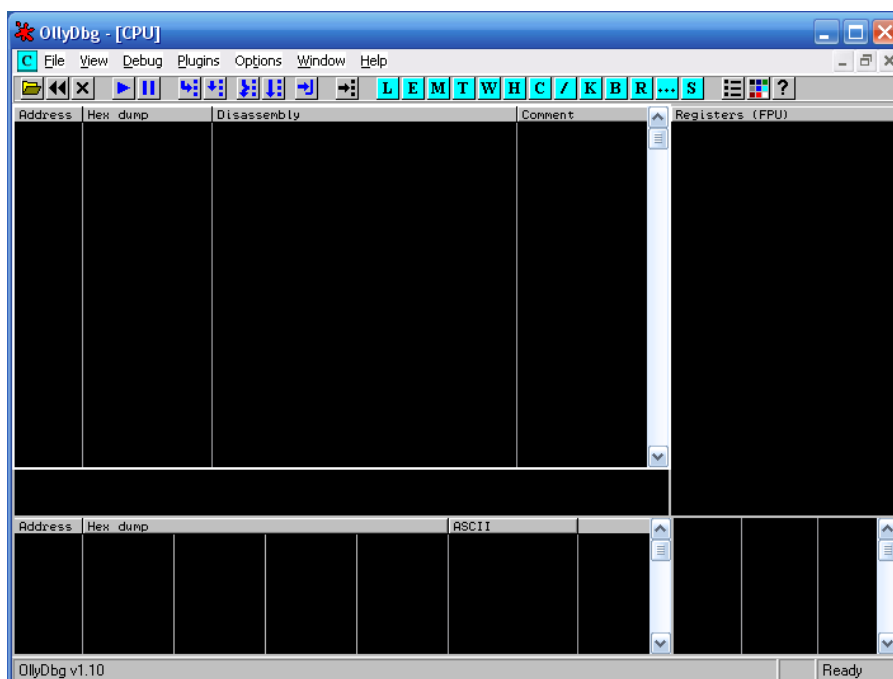


ENCONTRANDO A MENSAGEM SECRETA

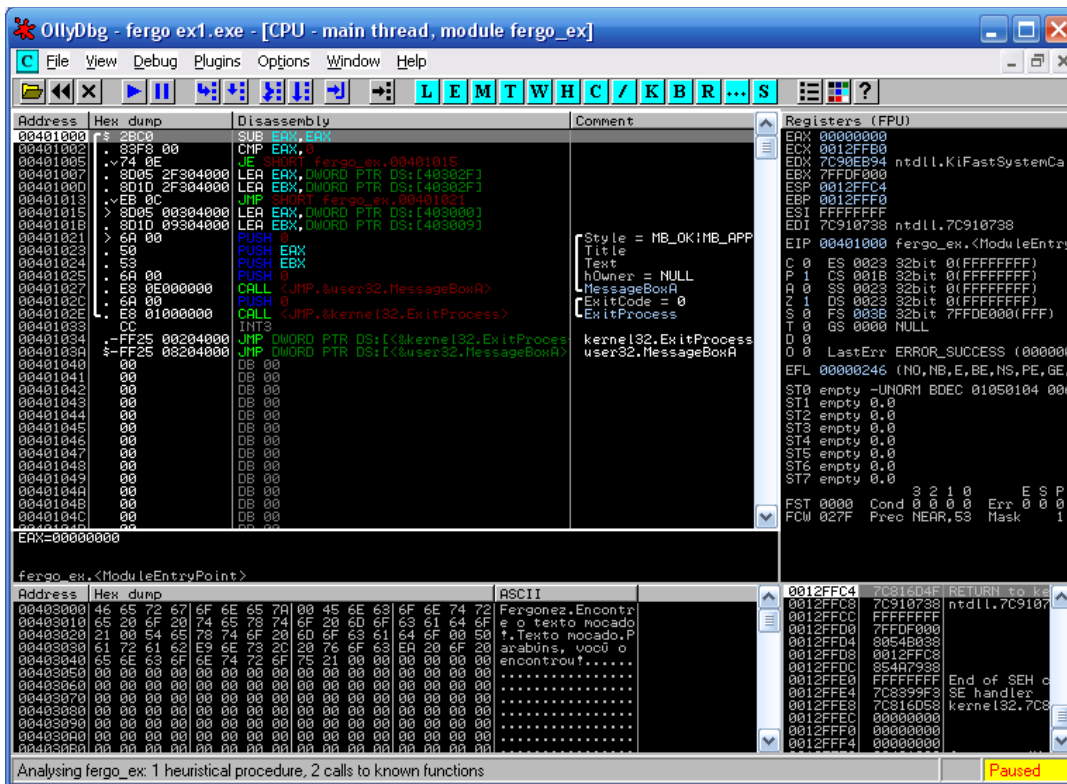
Esse é meu primeiro tutorial sobre engenharia reversa. Se você sabe tanto quanto eu (ou seja, quase nada), essa página é um bom começo. Vou explicar algumas coisas básicas e fazer um "debug" manual de um programa feito em assembly (seu objetivo será encontrar a frase escondida). O programa que eu vou utilizar nesse tutorial é extremamente simples, feito em assembly e compilado no MASM32. O código está incluso junto com o executável (mas não olhe o código antes de finalizar este tutorial).

-Download [fergo_ex1.zip](#)

Antes de tudo, precisamos de algo que transforme o nosso executável em uma linguagem que o ser humano possa entender (ou ao menos tentar). Para isso, voce precisa de um Debugger ou Disassembler. Neste tutorial eu vou utilizar um dos Debuggers mais completos atualmente (um dos mais famosos também), por ter uma interface mais visual e ser freeware: OllyDbg Ao iniciar o Olly, você terá uma tela semelhante a essa (as cores podem variar, dependendo da configuração do usuário)



Vamos abrir então o nosso executável para analisar o seu código (em linguagem de máquina, assembly (asm)). Vá em "File -> Open" e seleccione "fergo ex1.exe". Por se tratar de um executável pequeno, ele abre instantaneamente e tem poucas linhas de código efetivas. Você terá algo mais ou menos assim:



Quanta coisa né? Mas não esquite, com o tempo voce fica mais familiarizado. Na janela principal (superior esquerda), você tem 4 colunas: Adress, Hex Dump, Disassembly, Comment. Adress contem o endereço de cada instrução, é por esse endereço que você vai determinar saltos (jumps), chamadas (calls), etc... Hex Dump é a instrução no formato hexadecimal (não interessa agora). Disassembly é o mesmo que o Hex Dump, mas "traduzido" em letras, digamos assim. Comments não tem relação com o código, apenas ajuda a identificar algumas coisas (chamadas de função por exemplo). Como o código é pequeno, eu vou numerar as linhas para começarmos o nosso "debug"

```

01 00401000 2BC0 SUB EAX,EAX
02 00401002 83F8 00 CMP EAX,0
03 00401005 74 0E JE SHORT fergo_ex.00401015
04 00401007 8D05 25304000 LEA EAX,DWORD PTR DS:[40302F]
05 0040100D 8D1D 25304000 LEA EBX,DWORD PTR DS:[40302F]
06 00401013 EB 0C JMP SHORT fergo_ex.00401021
07 00401015 8D05 00304000 LEA EAX,DWORD PTR DS:[403000]
08 0040101B 8D1D 09304000 LEA EBX,DWORD PTR DS:[403009]
09 00401021 6A 00 PUSH 0
10 00401023 50 PUSH EAX
11 00401024 53 PUSH EBX
12 00401025 6A 00 PUSH 0

```

Vamos ao nosso debug então.

Linha 1: SUB EAX, EAX

SUB indica uma operação de subtração, seguida de seus 2 argumentos. EAX é um registrador, um local de armazenamento de dados temporário, onde normalmente são colocados valores para comparação, etc. Esse comando, mais especificamente, coloca no seu primeiro argumento, a subtração dele mesmo com o segundo elemento, algo como "EAX = EAX - EAX". Sim, isso dá zero (é uma das maneiras de zerar um valor em ASM).

Linha 2: CMP EAX, 0

CMP significa Compare. Ele compara seu primeiro argumento com o segundo. Se a comparação for verdadeira, ele seta uma "flag" indicando que é verdadeira. Nesse caso, ele está comparando EAX com 0 (algo como "if (eax == 0)" em C). Na linha anterior, EAX foi zerado, e agora ele está sendo comparado com 0, então, essa comparação é verdadeira.

Linha 3: JE SHORT fergo_ex.00401015

Jump if Equal. Como o nome já diz, se os argumentos da comparação anterior forem iguais (comparação verdadeira), ele realiza um salto para outra região do código. Nesse caso, como a comparação foi verdadeira, ele vai pular para o endereço 00401015 do executável fergo_ex1.exe.

Vou pular as linhas 4, 5 e 6 para não matar a charada logo no começo. Falaremos dela depois

Linha 7: LEA EAX, DWORD PTR DS:[403000]

O comando LEA faz com que o primeiro argumento aponte para o segundo argumento. Ele não recebe o valor do segundo argumento, recebe apenas o "local" onde está esse valor. Nesse caso, ele vai mover para o registrador EAX, o endereço 403000 (que corresponde a um valor de 32 bits (DWORD)).

Linha 8: LEA EBX, DWORD PTR DS:[403009]

Mesma coisa que o comando de cima, só que ele move um endereço diferente para uma variável diferente (403009 para EBX)

Linha 9: PUSH 0

Apenas "puxa" o seu argumento (0) para um local temporário, não realiza nenhum comando. Veja mais abaixo.

Linha 10, 11, 12: PUSH ...

Faz a mesma coisa que a linha anterior, só alterando o seu argumento. Ele vai puxar a variável EAX, EBX e depois novamente um 0.

Linha 13: CALL <JMP.&user32.MessageBoxA>

Faz uma chamada para uma função qualquer. Nesse caso, ele vai chamar a função MessageBoxA, contida na DLL user32.dll. Como você já deve ter imaginado, essa função exibe uma mensagem de texto. Você deve ter imaginado também que essa é a mensagem de texto que aparece quando você inicia o programa. Tá, mas onde ela pega o conteúdo para exibir? Vamos dar uma olhada nos argumentos que a função MessageBoxA recebe (procure no google caso queira saber da onde eu tirei isso):

MessageBoxA (dono, endereço do texto, endereço do título, tipo)

Dono indica o dono da janela, não importa agora. Endereço do texto e endereço do título é o que o próprio nome já diz. Tipo é o tipo da mensagem (botão OK/Cancel, Yes/No, etc...). Mas onde são passados esses argumentos no nosso código? Toda a função Call em ASM vai pegar os argumentos que você "puxou" na ordem reversa. Ou seja, o Dono ele vai pegar do endereço 00401025, o Texto ele pega do 00401024 e assim por diante. Cada "Push" que você deu, ele colocou o valor no topo de uma pilha, sendo que para pegar os valores dessa pilha, você começa pelo último valor (o último Push). Imagine você empilhando livros e depois pegando eles para organizar numa ordem ;D

Linha 14: PUSH 0

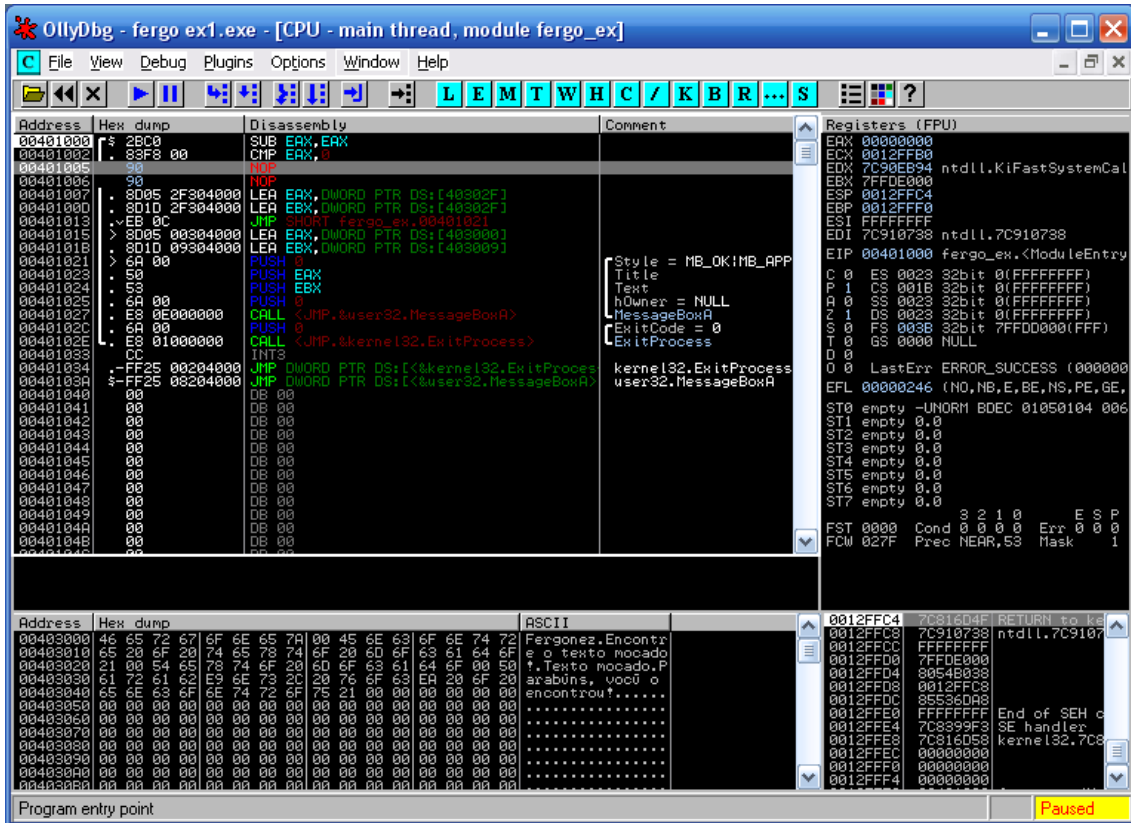
Um novo valor é posto no "stack" (agora você já deve imaginar que deve vir alguma outra função que fará o uso desse valor). E vem mesmo!

Linha 15: CALL <JMP.&kernel32.ExitProcess>

Novamente é feita uma chamada a uma função. Desta vez, a função é ExitProcess. Você já deve ter percebido que essa função encerra o programa caso o seu argumento possua um determinado valor. Você notou quando rodou o executável que assim que você clica em OK, o programa encerra, então faz sentido. Na linha anterior você colocou um valor 0 no "stack". Essa função recebe esse valor zero. Se o seu programa encerra quando você fecha a janela e a função de finalizar o processo recebe o valor 0, você sabe que caso a função receba o valor 0, ela encerra o programa.

Certo, o programa encerra por aqui. Mas e a minha mensagem escondida, onde está? Re-analise o código. Repare que na linha 7 e 8, você determina valores para o EAX e EBX, e logo depois você chama uma função que coloca esses 2 registradores como sendo Texto e o Título da mensagem. Agora ficou claro que essas 2 linhas colocam nos registradores o título e o texto da mensagem. Repare agora na linha 4 e 5. Também temos 2 LEA que fazem praticamente a mesma coisa. Oras, nessas 2 linhas ele deve atribuir a mensagem escondida para EAX e EBX, mas porque ele não faz isso? Veja que na linha 3 ele realiza um salto caso a condição seja verdadeira (é o que ocorre, lembra?). Como o salto ocorre, ele sequer passa por essas 2 linhas para poder atribuir a mensagem secreta. Agora pense no que você poderia fazer para que ele passasse por essas 2 linhas de código? Bom, tem várias alternativas, mas provavelmente vem na sua cabeça simplesmente tirar aquele jump da linha 3 ou invertê-lo, fazer com que caso a condição NÃO seja verdadeira, ele realize o salto (o pulo nunca vai ocorrer, visto que a comparação de 0 com 0 sempre vai ser verdadeira).

Nós vamos "retirar" aquele jump, pois é mais simples (na verdade o trabalho é o mesmo :P). Você não pode "deletar" uma linha, pois isso alteraria o endereço de todas as instruções, e o programa iria parar de funcionar. Felizmente existe o comando "NOP" (No Operation), que "anula" a linha sem alterar nenhum endereço. Para fazer isso, clique com o botão direito na linha 3, vá em "Binary->Fill with NOPS". Pronto, você anulou o pulo.



Repare que agora ele vai chegar na linha 3 e vai continuar seu caminho, sem um salto. Ele vai atribuir um valor ao EAX e ao EBX (provavelmente nossa mensagem escondida) e em seguida vai simplesmente pular (JMP, na linha 6) para o endereço 00401015 (00401021 após a alteração), que é justamente onde ele começa a puxar os valores para a chada da função MessageBoxA (repare que fazendo isso, ele evita que os valores originais das mensagens sejam re-atribuídos a EAX e EBX).

Que tal testar o que a gente fez? Clique com o botão direito sobre qualquer linha, vá em "Copy To Executable->All Modifications" e em seguida "Copy All". Uma nova janela se abrirá. Clique novamente com o botão direito sobre ela, selecione "Save File" e salve seu arquivo alterado. Pronto, agora execute o seu arquivo recém salvo e você vai ver que a mensagem secreta era "Parabéns, você o encontrou!". Espero ter dado o ponta pé inicial para aqueles que não sabiam por onde começar ou não sabiam o significado das instruções básicas do Assembly.

Tutorial - Engenharia Reversa

Parte 2



BUSCANDO A SENHA CORRETA

Neste tutorial vamos aprender como encontrar um código válido para que a mensagem correta apareça. É bom lembrar que engenharia reversa é completamente legal desde que não envolva softwares comerciais ou que não viole direitos autorais. Como nesse caso nós estaremos utilizando um aplicativo que foi criado exclusivamente para esse tipo de estudo, não tem problema.

Antes de tudo baixe o nosso alvo:

-Download: fwdv2.zip

Vamos lá. Antes de iniciar o Olly, execute o programa, entre com um código qualquer e clique em 'Register'. Hum, apareceu a mensagem de Wrong Code. Isso, por incrível que pareça é bom, por alguns motivos, sendo 2 deles:

1) A mensagem aparece numa MessageBox (que é uma chamada de API). Podemos localizar a região onde é feito o cálculo do código correto colocando um breakpoint em todos os locais onde é feita uma chamada para a função MessageBoxA

2) Através dessa mensagem de "Wrong Code", podemos também descobrir o local correto vendo onde ela é utilizada (provavelmente em conjunto com a MessageBox).

Vamos pelo segundo método. Inicie o Olly e abra o nosso alvo. O código é bem pequeno, poucas linhas. Bom para nós. Clique com o botão direito sobre a janela principal, vá em 'Search for->All Referenced Text Strings'. Uma nova janela vai abrir mostrando todos os textos utilizados no programa. Logo de cara você já encontra a tal mensagem "Sorry, wrong code". Mas veja logo abaixo. Tem uma mensagem "Success! Thanks for Playing". Oras, provavelmente essa mensagem é usada quando acertamos o código.

Address	Disassembly	Text string
00401000	PUSH 0	(Initial CPU selection)
00401054	PUSH v2.00403000	ASCII "Fishing with DILA v0.2"
00401059	PUSH v2.00403017	ASCII "Sorry, wrong code!"
0040106A	PUSH v2.00403000	ASCII "Fishing with DILA v0.2"
0040106F	PUSH v2.0040302A	ASCII "Success! Thank you for playing ;)"

Dê um duplo clique sobre a "boa mensagem". Nós vamos ser levados ao local onde ela é utilizada. Você vai parar aqui:

```

00401068  6A 40      PUSH 40
0040106A  68 00304000  PUSH v2.00403000
0040106F  68 2A304000  PUSH v2.0040302A
00401074  FF75 08      PUSH DWORD PTR SS:[EBP+8]
00401077  E8 28000000  CALL <JMP.&user32.MessageBoxA>

```

Como você pode ver, ela é o segundo argumento da função MessageBoxA. Certo, mas quando que essa função é chamada? Precisamos descobrir como que nós podemos parar aqui. Para isso, selecione a primeira linha dessa sequencia do message box (0041068). Agora bem em baixo da janela do código, apareceu um texto "Jump from 00401050".

```

00401066  00
00401067  00
00401068  00
00401069  00
Jump from 00401050

```

Address	Hex dump
00403000	46 69 73 68 69

Isso significa que para essa mensagem aparecer, um salto no endereço 00401050 vai ter que ser efetuado. Vá até o endereço 00401050 (aperte CTRL G e digite o endereço ou simplesmente procure no olho mesmo, já que o código é pequeno. No endereço 00401050 temos:

```
00401050 74 16 JE SHORT v2.00401068
```

Se 2 elementos comparados forem iguais, ele pula para 00401068 (que é o local onde a MessageBox contendo a mensagem de sucesso aparece). Quais elementos? Os elementos da linha anterior. Dê uma olhada geral nas linhas anteriores à esse jump:

```

00401035  75 45      JRC SHORT v2.0040107C
00401037  6A 00      PUSH 0
00401039  6A 00      PUSH 0
0040103B  68 E0300000  PUSH 3EA
00401040  FF75 08      PUSH DWORD PTR SS:[EBP+8]
00401043  E8 56000000  CALL <JMP.&user32.GetDlgItemInt>
00401048  B8 9A030000  MOV EBX, 39A
0040104D  4B        DEC EBX
0040104E  3BC3      CMP EAX, EBX
00401050  74 16      JE SHORT v2.00401068
00401052  6A 10      PUSH 10
00401054  68 00304000  PUSH v2.00403000
00401059  68 17304000  PUSH v2.00403017
0040105E  FF75 08      PUSH DWORD PTR SS:[EBP+8]
00401061  E8 3E000000  CALL <JMP.&user32.MessageBoxA>
00401066  EB 14      JMP SHORT v2.0040107C
00401068  > 6A 40      PUSH 40
0040106A  68 00304000  PUSH v2.00403000
0040106F  68 2A304000  PUSH v2.0040302A
00401074  FF75 08      PUSH DWORD PTR SS:[EBP+8]
00401077  E8 28000000  CALL <JMP.&user32.MessageBoxA>
0040109E=<JMP.&user32.GetDlgItemInt>

```

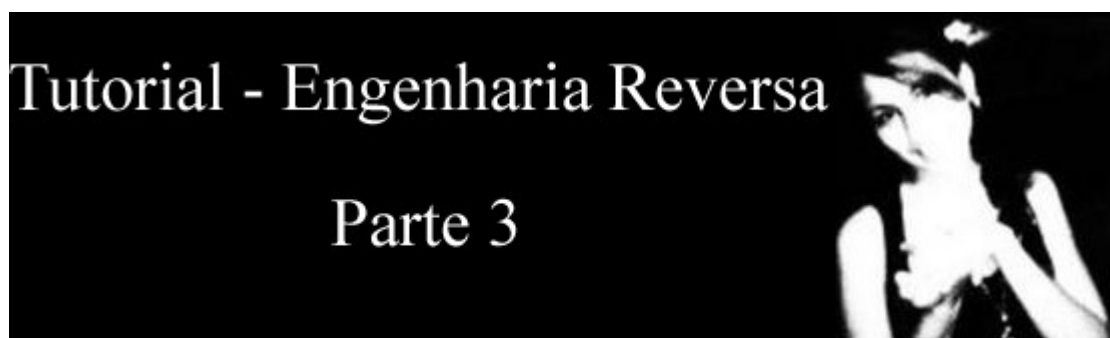
Em 00401043 ele chama uma função que pega um número digitado em uma caixa de texto (já desconfia que número é esse que ele pega né?). Normalmente essa função coloca o seu resultado (o valor), no registrador EAX (lembre-se disso). Depois ele move para o registrador EBX o valor hexadecimal 39A (922 em decimal). Na linha seguinte ele decrementa o EBX (subtrai 1 de EBX).

Como EBX tinha o valor 922, agora ele passa a ser 921. No endereço 0040104E ele compara EBX com EAX. EBX é 921, mas e o EAX, quanto é? Algumas linhas acima eu disse que o valor pego na caixa de texto é armazenado no registrador EAX, logo, ele vai comparar o EAX (921) com o texto digitado. Se os 2 forem iguais, ele vai executar o salto (da linha 00401050) que nos leva até a mensagem da senha correta, caso contrário, ele continua sem fazer o pulo e exibe a mensagem de senha incorreta. Matamos a charada. Se o valor digitado for igual a EBX (921), ele exibe a mensagem de senha válida, caso contrário, receberemos a mensagem de senha incorreta. Experimente digitar 921 na caixa de texto do programa e clicar em 'Register'. Voilá! Descobriu a senha de acesso :) Tem algumas outras maneiras de localizar o local onde está o algoritmo que verifica a senha correta. Um deles é este que nós usamos. Outra maneira seria colocar breakpoints em chamadas de funções clássicas (caso o programa use-as). Abaixo algumas delas:

- `GetDlgItemText` ou `GetDlgItemText`
- `LStrCmp` ou `StrCmp`
- `GetWindowTextA` ou `GetWindowText`
- `MessageBoxA`
- `GetDlgItemInt`

O nome das funções já dizem tudo. A primeira pega textos digitados em caixas de diálogo. A segunda compara strings. A terceira pega texto de janela. A quarta exibe messagebox (como foi o caso deste tutorial), e a última pega um número digitado na caixa de texto (também neste tutorial).

Outro método que poderia ser utilizado é alterar o código para que ele aceite qualquer valor que você digite. Para fazer isso, você tem que obrigar o programa a realizar o salto, não somente quando a comparação for verdadeira. Você consegue isso alterando o JE (Jump if Equal) para simplesmente JMP (Jump). Ele vai sempre pular para a MsgBox correta, não interessando se o valor digitado está certo ou não. Leia meu tutorial anterior (http://fergo.5gigs.com/reveng/tut1/tut_engrev1.html) para saber como alterar o código e salvar o arquivo novamente.



ENTENDENDO ALGORÍTMOS

Lembrete: Engenharia reversa é completamnetne legal desde que não envolva softwares comerciais ou que não viole direitos autorais.

Como nesse caso nós estaremos utilizando um aplicativo que foi criado exclusivamente para esse tipo de estudo, não tem problema.

Como sempre, baixe o nosso alvo:

-Download: [keygenme1.zip](#)

PARTE 1

Antes de abrir o Olly, rode o programa (tem até musiquinha), chute um nome qualquer (de referência nunca chute nomes com menos de 5 caracteres e mais que 16) e chute uma key qualquer. A não ser que você seja largo, mas largo mesmo, deve ter aparecido a mensagem "Hello, Mr. Badboy!". Erramos a key. Ok, abra o Olly e abra o nosso alvo nele. Para encontrar o local onde é feita a verificação, vamos dar uma olhada nas funções que o programa utiliza das APIs do windows. Aperte ALT E. Na lista que apareceu, clique com o botão direito na primeira (Name = keygenme) e depois View Names. No tutorial anterior eu falei sobre algumas funções interessantes que devemos dar maior atenção. São as marcadas em vermelho:

00404044	.rdata	Import	kernel32.CloseHandle
00404040	.rdata	Import	kernel32.CreateFileA
0040403C	.rdata	Import	kernel32.CreateThread
00404060	.rdata	Import	user32.DialogBoxParamA
00404064	.rdata	Import	user32.EndDialog
00404000	.rdata	Import	kernel32.ExitProcess
00404038	.rdata	Import	kernel32.FindResourceA
00404034	.rdata	Import	kernel32.FreeResource
00404068	.rdata	Import	user32.GetDlgItem
0040406C	.rdata	Import	user32.GetDlgItemTextA
00404004	.rdata	Import	kernel32.GetModuleHandleA
00404070	.rdata	Import	user32.GetWindowLongA
00404074	.rdata	Import	user32.LoadIconA
00404030	.rdata	Import	kernel32.LoadResource
0040402C	.rdata	Import	kernel32.LocalAlloc
00404028	.rdata	Import	kernel32.LocalFree
00404024	.rdata	Import	kernel32.LockResource
00404010	.rdata	Import	kernel32.lstrcatA
00404014	.rdata	Import	kernel32.lstrcmpA
00404018	.rdata	Import	kernel32.lstrlenA
00404078	.rdata	Import	user32.MessageBoxA
00401000	.text	Export	<ModuleEntryPoint>
00404020	.rdata	Import	kernel32.ReadFile
00404008	.rdata	Import	kernel32.RtlZeroMemory
00404084	.rdata	Import	user32.SendMessageA
00404080	.rdata	Import	user32.SetDlgItemTextA
0040401C	.rdata	Import	kernel32.SetFilePointer
0040407C	.rdata	Import	user32.SetFocus
00404058	.rdata	Import	user32.SetLayeredWindowAttributes
00404050	.rdata	Import	user32.SetWindowLongA
00404054	.rdata	Import	user32.SetWindowPos
00404088	.rdata	Import	user32.ShowWindow
00404048	.rdata	Import	kernel32.SizeofResource
0040400C	.rdata	Import	kernel32.Sleep
004040A8	.rdata	Import	winmm.waveOutClose
004040A4	.rdata	Import	winmm.waveOutGetPosition
004040A0	.rdata	Import	winmm.waveOutOpen
0040409C	.rdata	Import	winmm.waveOutPrepareHeader
00404098	.rdata	Import	winmm.waveOutReset
00404094	.rdata	Import	winmm.waveOutUnprepareHeader
00404090	.rdata	Import	winmm.waveOutWrite
0040405C	.rdata	Import	user32.wsprintfA

Todas elas vão acabar nos levando ao algoritmo, mas uma delas em especial nos leva mais rapidamente a ele: lstrcmpA. Essa função faz uma comparação entre 2 strings, e é amplamente utilizada para comparar uma key verdadeira com a key que voce digitou. Vamos setar um breakpoint em todos os locais onde ela é chamada. Clique com o botão direito sobre a linha que contém a lstrcmpA e marque 'Set breakpoint on every reference'.

Pode fechar essas janelas (tanto a das funções quanto à das dlls utilizadas) e voltar para a tela padrão do código. Rode o programa pelo Olly, digite novamente um nome e uma key e aperte 'Check'. Pimba, paramos no nosso breakpoint bem na chamada da função. Certo, estamos parados bem na hora em que ele vai comparar 2 strings.

```

00401313 .: 68 13DC4000 PUSH EBX keygenme.0040DCF8
0040131A .: 68 13DC4000 PUSH EBX
0040131B .: 68 13DC4000 PUSH EBX
0040131D .: E8 6B000000 CALL <JMP.&kernel32.lstrcmpA>
00401321 .: 74 15 JE SHORT keygenme.00401338

```

Agora repare nos 2 argumentos que essa função lstrcmpA recebe (as 2 linhas que antecedem essa chamada). EBX contém 123456 (que foi o serial que eu chutei) e EAX contém uma outra string, num formato BEM típico de key. Após isso ele chama a função lstrcmpA que vai comparar os 2 valores (123456 com "Von-FF...") e se a comparação for verdadeira, ele pula para 00401338 (JE abaixo do CALL). Se você for até o endereço 00401338, você vai ver que é o local onde ele exibe a mensagem de que a key está correta. Experimente anotar o valor de EBX (String1), e testá-la nesse KeyGenMe (utilize o mesmo nome de antes, pois a key é gerada a partir do nome). BINGO! Você acaba de descobrir a key correta para o seu nome. :)

PARTE 2

Certo, encontramos a key verdadeira para o seu nome, mas que tal seguirmos adiante e descobrir como essa key é gerada? Delete os breapoints atuais (aperte ALT B e delete todos). Volte a janela dos módulos (ALT E), selecione "View Names" no primeiro item da lista. Vamos setar um breakpoint em todos os locais onde ele chama a função GetDlgItemTextA (pega um valor digitado na caixa de texto). Caso não lembre como fazer isso, releia o início deste tutorial. Depois de setar o breakpoints, clique em "Play", digite um nome e uma key e novamente clique em 'Check'. Certos, paramos no local onde ele adquire o texto de uma das textbox. Note que tem 2 chamadas dessa função, uma logo após a outra. Obviamente em uma vai pegar o nome e a outra vai pegar a key. Aperte F9 para continuar a execução do programa e novamente nós paramos na chamada da função (agora para pegar o valor do segundo textbox)

```

004010E9 6A 28 PUSH 28
004010EB 68 F8DC4000 PUSH keygenme.0040DCF8 ; Armazena o nome em 0040DCF8
004010F0 68 EE030000 PUSH 3EE
004010F5 FF75 08 PUSH DWORD PTR SS:[EBP+8]
004010F8 E8 B3020000 CALL <JMP.&user32.GetDlgItemTextA> ; Chama a função para pegar o nome
004010ED 6A 28 PUSH 28
004010FF 68 F8DE4000 PUSH keygenme.0040DEF8 ; Armazena a key em 0040DEF8
00401104 68 EF030000 PUSH 3EF
00401109 FF75 08 PUSH DWORD PTR SS:[EBP+8]
0040110C E8 9F020000 CALL <JMP.&user32.GetDlgItemTextA> ; Chama a função para pegar a key
00401111 E8 F2000000 CALL keygenme.00401208

```

Após ter pego os 2 textos e armazenados nos determinados locais, ele realiza um pulo para 00401208 (no endereço 00401111).

O instindo nos diz que essa chamada provavelmente gera uma key a partir do nome que é comparada posteriormente (como vimos agora pouco). Vamos "entrar" nessa função para analisar o que ela faz. Selecione a linha 00401111 e aperte ENTER.

```

00401208  .: 68 F8DC4000  PUSH keygenne.0040DCF8
0040120D  .: E8 80010000  CALL <JMP.&kernel32.lstrlenA>
00401212  .: A3 86DC4000  MOV DWORD PTR DS:[40DC86],EAX
00401217  .: 833D 86DC4000  CMP DWORD PTR DS:[40DC86],4
0040121E  .: 0F8C 29010000  JL keygenne.0040134D
00401224  .: 833D 86DC4000  CMP DWORD PTR DS:[40DC86],32
0040122B  .: 0F8F 1C010000  JG keygenne.0040134D
00401231  .: 33C0        XOR EAX,EAX
00401233  .: 33DB        XOR EBX,EBX
00401235  .: 33C9        XOR ECX,ECX
00401237  .: BF F8DC4000  MOV EDI,keygenne.0040DCF8
0040123C  .: 8B15 86DC4000  MOV EDX,DWORD PTR DS:[40DC86]
00401242  .: 0FB60439    MOVZX EAX,BYTE PTR DS:[ECX+EDI]
00401246  .: 83E8 19      SUB EAX,19
00401249  .: 2BD8        SUB EBX,EAX
0040124B  .: 41          INC ECX
0040124C  .: 3BCA        CMP ECX,EDX
0040124E  .: 75 F2       JNZ SHORT keygenne.00401242
00401250  .: 58          PUSH EBX
00401251  .: 68 F8DB4000  PUSH keygenne.0040DBF8
00401256  .: 68 F8E04000  PUSH keygenne.0040E0F8
0040125B  .: E8 38010000  CALL <JMP.&user32.wsprintfA>
00401263  .: 83C4 0C      ADD ESP,0C
00401266  .: 33C0        XOR EAX,EAX
00401268  .: 33DB        XOR EBX,EBX
0040126A  .: 33C9        XOR ECX,ECX
0040126C  .: 03C3        ADD EAX,EBX
0040126E  .: 0FACF3     IMUL EAX,EBX
00401270  .: 2BD8        SUB EBX,EAX
00401272  .: 33DB        XOR EBX,EBX
00401274  .: 0FADF8     IMUL EBX,EAX
00401277  .: 58          PUSH EBX
00401278  .: 68 F8DB4000  PUSH keygenne.0040DBF8
0040127D  .: 68 F8E14000  PUSH keygenne.0040E1F8
00401282  .: E8 11010000  CALL <JMP.&user32.wsprintfA>
00401287  .: 83C4 0C      ADD ESP,0C
0040128A  .: 33C0        XOR EAX,EAX
0040128C  .: 33DB        XOR EBX,EBX
0040128E  .: 33C9        XOR ECX,ECX
00401290  .: 33C9        XOR EDX,EDX
00401292  .: B8 F8E04000  MOV EAX,keygenne.0040E0F8
00401297  .: 03D8        ADD EBX,EAX
00401299  .: 33CB        XOR ECX,EBX
0040129B  .: 0FACFB     IMUL ECX,EBX
0040129E  .: 2BC8        SUB ECX,EAX
004012A0  .: 51          PUSH ECX
004012A1  .: 68 F8DB4000  PUSH keygenne.0040DBF8
004012A6  .: 68 F8E24000  PUSH keygenne.0040E2F8
004012AB  .: E8 E8000000  CALL <JMP.&user32.wsprintfA>

```

Devido ao comprimento do código, não vou explicar exatamente o que cada linha faz, vou ressaltar as mais importantes. Logo de cara você percebe que existe uma certa "simetria" digamos assim. Você consegue ver que o algoritmo fica dividido em 3 sequencias semelhantes. Caso não tenha reparado, a key correta é composta por um "Bon-" seguido de 3 sequências numéricas. Podemos então supor que cada um dos blocos de código (marcados em vermelho), são responsáveis por gerar cada parte da key.

-Analisando a primeira sequência (00401208)

Vamos então iniciar a análise pelo endereço 00401208. Ele primeiramente pega o tamanho do nome e armazena em EAX. Depois joga o valor de EAX para o endereço [40DC86]. Nas linhas seguintes ele compara o tamanho do nome (que está em [40DC86]) com 4 e depois com 32 (50 em decimal). Se for maior que 50 ou menor que 4 ele faz um salto para indicar que o nome está ou muito grande ou muito pequeno. Supondo que você tenha digitado um nome que tem um tamanho $4 \leq \text{nome} \leq 50$, podemos continuar. Em 00401231 começa o cálculo da primeira sequencia da key. Ele zera EAX, EBX e ECX (XOR X, X zera o valor X). Depois move para EDI o nome digitado e para EDX o tamanho do nome.

Na próxima linha começa um "Loop", ou seja, uma sequência que se repete até que determinada condição seja alcançada. Veja a sequência

```

00401242 0FB60439 MOVZX EAX, BYTE PTR DS:[ECX+EDI] ; Move to EAX the byte pointed by ECX ( initially zero )
00401246 83E8 19 SUB EAX, 19 ; Subtracts 19 ( 25 in decimal ) from EAX ( EAX = EAX - 25 )
00401249 2BD8 SUB EBX, EAX ; Subtract EAX from EBX ( EBX = EBX - EAX )
0040124B 41 INC ECX ; Increases ECX ( ECX = ECX + 1 )
0040124C 3BCA CMP ECX, EDX ; Compares ECX with EDX ( that contains the size of the name )
0040124E 75 F2 JNZ SHORT keygenme.00401242 ; If the operation is false ( ECX different of EDX ), get back to the start
of the loop ( 1242 )

```

Resumindo o algoritmo, ele pega o valor ASCII de cada caractere, tira 25 e depois subtrai esse valor encontrado de EBX, algo como:

```

For i = 1 to Tamanho do nome
  Sequencia1 = Sequencia1 - ASC(Caractere(i)) - 25
Next

```

Após o cálculo da primeira sequência, ele chama uma função que formata o texto e armazena ele em determinado endereço. Nesse caso, ele vai mover o resultado (que está em EBX) para a posição de memória [0040E0F8]

-Analisando a primeira sequência (00401263)

Novamente ele começa zerando EAX, EDX e ECX.

```

00401269 03C3 ADD EAX, EBX ; EAX = EBX ( i.e. EAX will have the value of the sequence from the last algo, storage in EBX )
0040126B 0FAFC3 IMUL EAX, EBX ; EAX = EAX * EBX ( With the two registers the same value )
0040126E 03C8 ADD ECX, EAX ; EAX = EAX * EBX ( With the two registers the same value )
00401270 2BD3 SUB EDX, EBX ; EDX = EDX - EBX ( EDX has the negative value from the EBX ) – Unuseful instruction
00401272 33D0 XOR EDX, EAX ; EDX receive the result of the binary operation XOR between EDX and EAX - Unuseful instruction
00401274 0FAFD8 IMUL EBX, EAX ; EBX = EBX * EAX ( EBX is "multiplied" with EAX ( that has been "multiplied" with EBX )

```

Analisando essa última instrução com cuidado, você descobre que a sequência 2 nada mais é do que o valor gerado na primeira sequência, elevado ao cubo. Assim:

$$\text{Sequencia2} = \text{Sequencia1} * \text{Sequencia1} * \text{Sequencia1}$$

O valor da sequência 2 é armazenado da mesma forma que na primeira sequencia, mas no endereço de memória [0040E1F8]

-Analisando a primeira sequência (0040128A)

Como sempre, ela começa zerando EAX, EBX, EDX, ECX

```

00401292  B8 F8E04000  MOV EAX, keygenme.0040E0F8
00401297  03D8         ADD EBX, EAX
00401299  33CB         XOR ECX, EBX
0040129B  0FAFCB      IMUL ECX, EBX
0040129E  2BC8         SUB ECX, EAX

```

Esta última sequência tem algo em particular. Repare que nenhuma das "variáveis" é variável (WTF?!). Em nenhum momento ele usa algo que possa variar de acordo com o nome digitado. O valor de EAX na instrução 00401292 vai ser sempre 0040E0F8, pois o endereço da instrução não vai mudar nunca. Fazendo todos os cálculos necessários (use a calculadora do Windows), você vai chegar no valor '41720F48', que é então armazenado na posição [0040E2F8].

Sequencia3 = 41720F48

Já deciframos todo o algoritmo. Como ele monta a string completa (junta tudo) não nos interessa, mas caso queria entender, ele faz isso a partir da linha 004012C5. Lembre-se também que ele adiciona um "Bon-" no início da key. Agora o que você precisa fazer é escolher uma linguagem para programar o seu gerador. Eu vou fazer em VisualBASIC porque todo mundo entende.

-Codigo do gerador, em VB

```

Private Sub Gerar_Click()
    Dim seq1 As Double, seq2 As Double
    Dim tamanho As Integer, i As Integer

    'obtains the size of the name
    tamanho = Len(txtNome.Text)
    If (tamanho < 4) Or (tamanho > 50) Then Exit Sub

    'sequencia 1
    For i = 1 To tamanho
        seq1 = seq1 - (Asc(Mid(txtNome.Text, i, 1))) - 25)
    Next

    'sequencia 2
    seq2 = seq1 ^ 3


    txtSerial.Text = "Bon-" & Hex(seq1) & "-" & Hex(seq2) & "-41720F48"
End Sub

```

É isso ae. Este é o fim do tutorial!

Tutorial - Engenharia Reversa

Parte 4

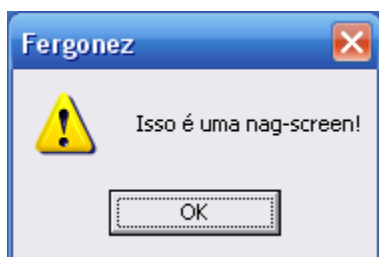


REMOVENDO NAG SCREENS

Este com certeza é meu tutorial mais fácil, feito especialmente para aqueles que estão trabalhando com engenharia reversa pela primeira vez. O foco principal é remover algo que chamamos de "Nag Screen", aquelas janelinhas chatas que aparecem quando você inicia determinados programas. Nesse caso, é o mais simples possível, então vamos lá. Baixe o nosso alvo (programado em ASM, por mim mesmo)

-Download: [fergo_nag.zip](#)

Rode o nosso alvo. Logo de cara aparece uma MessageBox alertando sobre a nag, e somente depois de clicar OK nos conseguimos entrar no "programa".



Certo, vamos ao Olly. Abra o Olly e carregue o executável. Temos diversas maneiras de chegar no local onde a messagebox é exibida. Um deles é procurando por todas as strings contidas no programa (Botão Direito->Search For->All referenced text string). Outro e rocurar pela chamada da função MessageBox. A segunda, neste caso, nos leva diretamente a chamada da nagscreen, pois só existe uma MessageBox no programa. Aperte ALT E, na lista que aparecer, clique com o botão direito sobre o primeiro item da lista (fergonag) e selecione View Names. Uma nova lista apareceu, contendo todas as funções que o executável utiliza. Repare que uma delas é a MessageBox. Vamos setar um breakpoint no local onde a função é chamada. Clique com o botão direito sobre "user32.MessageBoxA" e selecione "Set Breakpoint on every reference". Ao rodar o programa, ele vai congelar na hora em que for chamada a função Messagebox. Aperte F9 (ou clique no botão Play la em cima).

Assim que você clicar no botão play, ele já atinge a chamada da função e o Olly exhibe o local onde paramos:

00401023	6A 30	PUSH 30	
00401025	50	PUSH EAX	
00401026	53	PUSH EBX	
00401027	6A 00	PUSH 0	
00401029	E8 B0010000	CALL <JMP.&user32.MessageBoxA>	

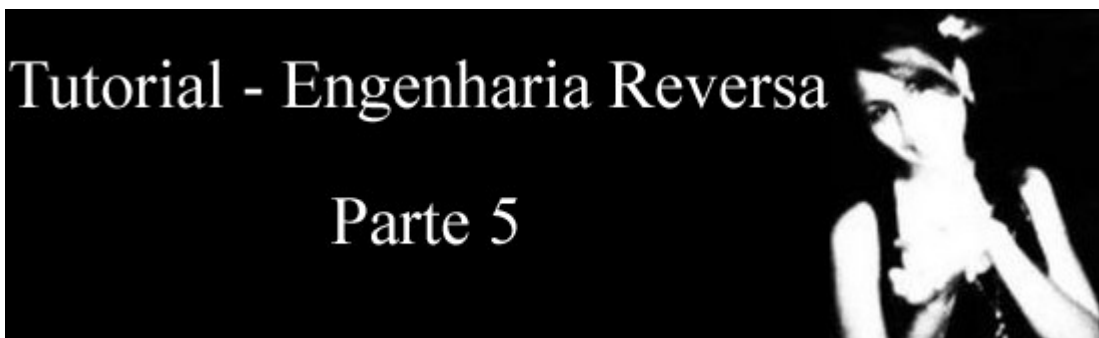
; Puxa o tipo da msgbox para janela com icone de exclamação
 ; Puxa o título
 ; Puxa o texto da msgbox
 ; Puxa o dono da janela
 ; Chama a função, que usa como argumentos os 4 valores puxados anteriormente

Certo, qual era o nosso objetivo mesmo? Remover essa msgbox. Bom, podemos simplesmente o conteúdo referente a msgbox. Mas é tão fácil assim mesmo? Pior que é. Para anular uma linha, usamos o comando NOP (No Operation). Selecione todas as 5 linhas referentes à MsgBox (do endereço 00401023 até 00401029), clique com o botão direito, vá em "Binary->Fill with Nops".

00401023	90	NOP	
00401024	90	NOP	
00401025	90	NOP	
00401026	90	NOP	
00401027	90	NOP	
00401028	90	NOP	
00401029	90	NOP	
0040102A	90	NOP	
0040102B	90	NOP	
0040102C	90	NOP	
0040102D	90	NOP	

Depois de alterar a linha, continue a execução do programinha (F9) e pimba! A nag-screen sumiu! Alcançamos nosso objetivo. Se quiser salvar as alterações, clique com "Botão Direito->Copy do Executable->All Modifications" e depois em "Copy All". Na janela que abrir, clique com o botão direito novamente e vá em "Save File".

F3rGO!



CRIANDO UM PATCHER

Este tutorial ensina como alterar os bytes de um executável para alterar o seu comportamento sem precisar do OllyDbg. Vou utilizar o executável do tutorial anterior (Tutorial #4).

-Download: [fergo_nag.zip](#)

INTRODUÇÃO

Se você leu o tutorial #4, você deve estar lembrado que tivemos que anular uma linha, para que uma mensagem de texto não fosse exibida. No caso, preenchemos a chamada da função messagebox com NOPs (cujo código é 90 em hexadecimal e ocupa somente 1 byte). Mas como fazer isso sem que o usuário tenha conhecimento de RCE e/ou OllyDbg? Você faz um outro utilitário que modifica o alvo! Vamos lá. Tínhamos a seguinte situação no tutorial 4



The screenshot shows the OllyDbg interface. On the left, the assembly window displays the following code:

```
00401023 .: 90 30          PUSH 30
00401025 .: 90 30          PUSH EAX
00401026 .: 90 30          PUSH EBX
00401027 .: 90 30          PUSH 30
00401029 .: E8 B0010000   CALL <JMP.&user32.MessageBoxA>
0040102E .: 90 30          PUSH ESI
0040102F .: 90 30          PUSH EDI
00401030 .: 90 30          PUSH 30
00401035 .: 90 30          PUSH 30
00401037 .: E8 BA010000   CALL <JMP.&kernel32.GlobalAlloc>
```

On the right, a MessageBox dialog box is open with the following properties:

- Style = MB_OK|MB_ICONEXCLAMATION|MB_APPLMODAL
- Title = "Ferguson"
- Text = "Isso é uma nag-screen!"
- hOwner = NULL
- MessageBoxA
- MemSize = 400 (1024.)
- Flags = GPTR
- GlobalAlloc

Preciso explicar algumas coisas. Cada instrução em ASM tem um valor, que geralmente ocupa 1 byte, seguido de algum argumento. A segunda coluna do Olly, contém o valor da instrução (OpCode) em Hexadecimal, e a terceira coluna mostra o que chamamos de Mnemônico, ou seja, o "apelido" ao OpCode, que facilita o nosso entendimento. Repare na linha 00401029. Temos o OpCode "E8 B0010000" e o mnemônico referente ao OpCode: "CALL <SMP.&user32.MessageBoxA". Nesse caso, E8 indica um CALL e os outros 4 bytes "B0010000" (lembre-se que estamos trabalhando com valores hexadecimais, que variam de 00 a FF (0 a 255) e 2 algarismos ocupam 1 byte na memória) representam a função MessageBoxA.

Agora vou falar um pouco do "Address", o número da linha. O número da linha indica simplesmente a localização de cada instrução a ser executada, em hexadecimal. Ela inicia em 0 e vai aumentando a cada instrução. Digamos que eu tenha um comando no endereço 00, e esse comando ocupa 2 bytes (um PUSH seguido de uma constante, por exemplo: PUSH 69). A próxima instrução vai estar no endereço 00 mais os 2 bytes que a instrução anterior ocupa, logo, ela vai estar no endereço 02. É através deste endereço que nós vamos nos localizar para fazer o Patch, mas tem um porém: o arquivo executável não começa logo com os comandos. Antes de iniciar os comandos, ele tem todo um cabeçalho, que indica diversas características, etc. Esse cabeçalho tem um tamanho de 1024 bytes (400 em Hex) e somente após ele que começam as instruções. Ou seja, o primeiro comando não vai estar no byte 0, mas sim no byte 1025 (401 em Hex).

Voltando ao programa. Repare na linha 00401023. Ali começam a ser puxados todos os 4 argumento que o CALL necessita para exibir a MessageBox.

No tutorial anterior, nós anulamos tudo referente a MsgBox. Quando você mandou preencher as linhas com NOPs, por exemplo, ele colocou o valor 90 (NOP) no byte 29 (que continha o E8 (CALL)), mas também incluiu diversos outros NOPs em linhas que nem existiam, sabe porque? Porque não basta ele anular o byte 29, ele tem que anular os outros 4 bytes seguintes também (lembre-se que era E8 B1 01 00 00).

Por isso que ele adicionou NOPs nos bytes 29, 2A, 2B, 2C, 2D também. Então, se queremos fazer um patcher, temos que anular tudo o que é referente a nossa mensagem de texto. Veja a tabela abaixo que mostra os bytes originais do arquivo e os bytes que vamos modificar:

	Bytes originais	Bytes Modificados
23	6A (PUSH)	90
24	30	90
25	50 (PUSHEAX)	90
26	53 (PUSHEBX)	90
27	6A (PUSH)	90
28	00	90
29	E8 (CALL)	90
2A	B1	90
2B	01	90
2C	00	90
2D	00	90
2E	56 (PUSHESI)	90

CRIANDO O PATCHER

Antes de iniciar o patcher, vamos lembrar o que e onde devemos alterar. Nós temos que preencher 11 bytes (desde o primeiro PUSH no endereço 00401023 até o 00 no endereço 0040102D) com o valor 90, que indica nenhuma operação. Onde vamos alterar? No arquivo executável é claro. Quais bytes? 23 (35 em decimal), 24 (36 em decimal) e assim por diante? NÃO! Lembre-se que antes de iniciar os comandos, tem 1024 bytes compondo o cabeçalho do executável. 1024 é 400 em Hexadecimal (use a calculadora do windows), então vamos ter que alterar do byte 423 até 42D (1059 até 1069). Vamos lá. Vou escrever novamente o código em VB, pois é simples de entender. Poderia ter escrito de uma maneira menor, mas ficaria mais complicado o entendimento. Adicione um Command Button com nome de cmbPatch). Lembrando que o patcher deve estar na mesma pasta do alvo

```

Private Sub cmbPatch_Click()
    Dim bytFile() As Byte
    Dim strFile As String
    Dim i As Integer

    strFile = App.Path & "\fergonag.exe"

    ReDim bytFile(FileLen(strFile) - 1) As Byte

    Open strFile For Binary As #1
        Get #1, , bytFile()
    Close #1

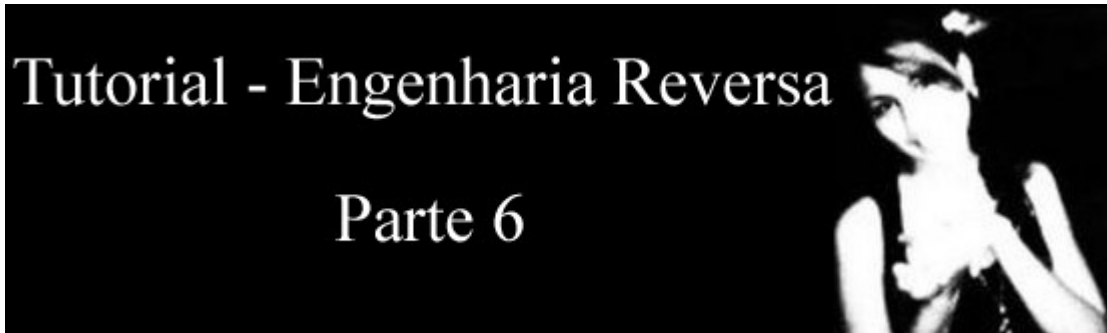
    For i = &H423 To &H42D
        bytFile(i) = &H90
    Next

    Open strFile For Binary As #1
        Put #1, , bytFile()
        MsgBox "Alvo alterado com sucesso", vbInformation, "Wee"
    Close #1
End Sub

```

~~==[Commented]==~~

Pronto, agora é só compilar e partir pro abraço. Não tenho certeza, mas talvez isso funcione até com VB Script :P Se quiser baixar o código fonte acima já nos padrões do VB clique [aqui](#)



DESCOMPACTANDO UPX

Diversos aplicativos distribuídos por aí vem compactado com algum tipo de compressor, para diminuir o tamanho do executável e dificultar o debug. Um desses compressores, talvez o mais famoso, é o UPX (Ultimate Packer for eXecutables). Possui uma ótima compressão e descompacta o executável original muito rapidamente. Ele é Open Source e pode-se baixá-lo gratuitamente em <http://upx.sourceforge.net/>

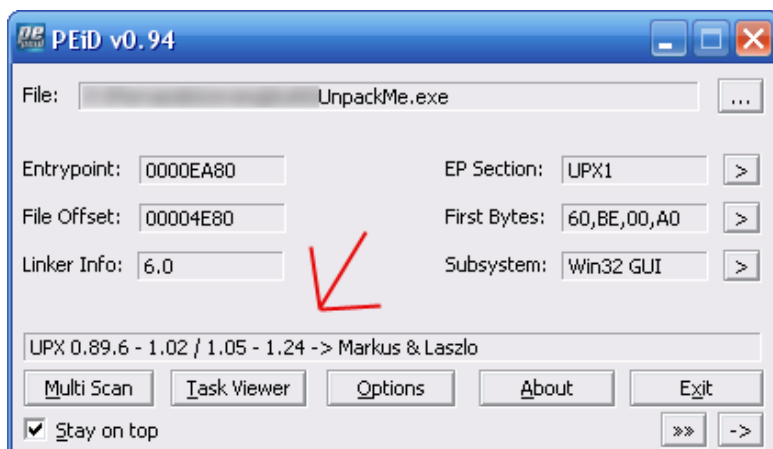
Existem diversas maneiras de descompactar executáveis. Você pode usar o próprio unpacker do UPX, usar um unpacker de terceiros ou descompactar na mão, que é o que vamos fazer nesse tutorial. Antes de tudo, vou explicar como um compressor do tipo do UPX funciona. Quando você utiliza um packer, como o UPX, o EP (Entry Point, onde inicia o código do programa) é desviado para outro endereço, que contém algo que chamamos de Loader. A função do loader é somente descompactar o executável para a memória e depois chamar o OEP (Original Entry Point, que contém o código verdadeiro e descomprimido). A sequência é mais ou menos essa:

EP -> Loader -> Descomprime o conteúdo do executável para a memória -> Move o EP para o OEP.

O processo para descompactar o UPX na mão é praticamente sempre o mesmo. É só seguir a mesma sequência que você consegue descompactar sem grandes dificuldades. Eu não vou explicar o que acontece com o loader ou como ele descomprime, nós vamos apenas descobrir onde está o ponto de partida inicial do código (OEP) e buscar o executável comprimido da memória. Para isso, vamos usar o OllyDbg e também outro utilitário que remonta a base de Imports dos arquivos executáveis (se tiver interesse em saber o que é isso, procure pelo file format de arquivos executáveis) chamado [ImportREC](#). Vamos então iniciar. Antes de tudo, nosso alvo. Um executável simples feito em VB e comprimido com UPX.

-Download: [unpackme.zip](#)

Como saber se ele está comprimido? Para ver se o executável contém algum tipo de proteção, eu uso o [PEiD](#), que analisa e detecta quase todos os tipos de compressores. Baixe e descompacte na pasta do Olly a dll do plugin [OllyDump](#), que vamos utilizar mais adiante.



Certo, ele foi comprimido com o UPX (a versão não interessa, pois o processo é o mesmo para todos). Abra o Olly e carregue nosso executável. Ele vai dar um aviso de que ele foi comprimido e pergunta se queremos continuar, clique em Yes. O Olly já nos deixa no Entry Point (do loader ainda):

0040EA80 60 PUSHAD

Guarde bem essa linha. Ele é TÍPICA de UPX. Todo arquivo comprimido com ele vai iniciar com um PUSHAD. Como eu disse, não vou explicar o porque de estarmos fazendo tal processo, etc, vou apenas indicar o que você deve fazer. Aperte F7 para que ele execute a instrução do PUSHAD e passe para a linha seguinte (EA81). Repare que na janela dos registradores (a direita), o ESP está colorido, indicando que ele mudou de valor. Nós vamos setar um breakpoint de hardware no conteúdo do primeiro byte do ESP, pois ele é acessado somente no início e no final do loader, assim nós cairemos bem perto do local onde encerra o loader e ele chama pelo OEP, que é o que nós precisamos. Clique com o botão direito sobre o valor de ESP e vá em "Follow in Dump". Agora nós estamos monitorando o conteúdo do ESP na parte de baixo da tela. Clique com o botão direito sobre o primeiro byte da janela de dump (38) e selecione "Breakpoint->Hardware, on access->DWord". Esse breakpoint vai nos levar logo no final do loader, bem perto da chamada para o OEP (para saber o motivo de setar esses breakpoints, você precisa estudar o comportamento do Loader, que não é o enfoque deste tutorial).

Address	Hex dump	ASCII
0012FFA4	38 07 91 7C FF FF FF FF F0 FF 12 00 C4 FF 12 00	8·! - - -
0012FFB4	00 A0 FD 7F 94 EB 90 7C B0 FF 12 00 00 00 00 00	·à·öü! ···
0012FFC4	4F 60 81 7C 38 07 91 7C FF FF FF FF 00 A0 FD 7F	Omú!8·! ···
0012FFD4	38 B0 54 80 C8 FF 12 00 A8 CD 60 85 FF FF FF FF	%TC· ··=·à
0012FFE4	F3 99 88 7C 58 60 81 7C 00 00 00 00 00 00 00 00	%0á!Xmú!.....
0012FFF4	00 00 00 00 80 EA 40 00 00 00 00 00 00 00 00 00ÇÜ@.....

Aperte F9 para continuar a execução do programa. Quase instantaneamente chegamos no breakpoint setado. O Olly parou aqui:


```

0040EC06  61          POPAD
0040EC07  8D4424 80   LEA EAX,DWORD PTR SS:[ESP-80] ; Paramos aqui
0040EC0B  6A 00      PUSH 0
0040EC0D  39C4      CMP ESP,EAX
0040EC0F  75 FA      JNZ SHORT UnpackMe.0040EC0B
0040EC11  83EC 80    SUB ESP,-80
0040EC14  E9 7725FFFF JMP UnpackMe.00401190

```

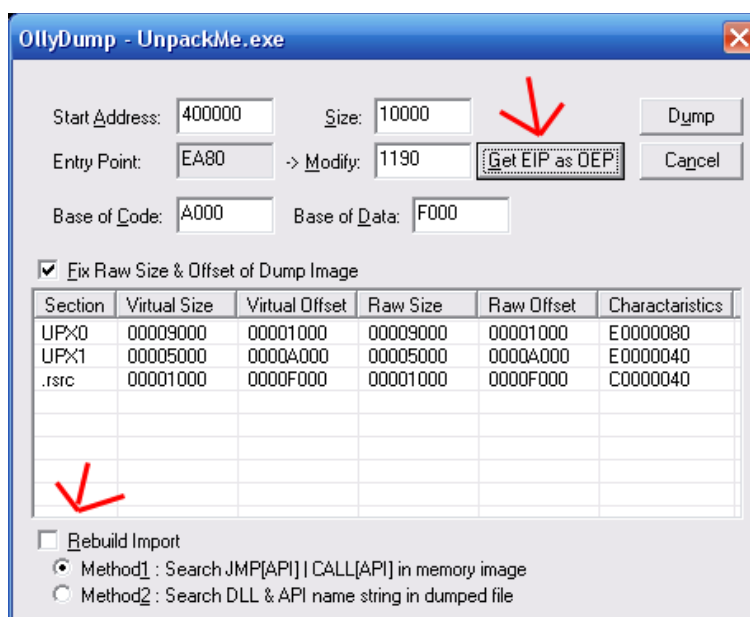
Repare no POPAD que apareceu. Como disse, isso é típico de UPX, começando com um PUSHAD e terminando com um POPAD. Repare agora na última linha, um jump obrigatório para o endereço 1190. Esse jump indica que chegamos no final do Loader, o aplicativo descomprimido já está na memória, e esse JMP vai nos levar até o ponto de partida original. (vindo pelo jump já dá pra saber que o OEP é 00401190). Coloque um Breakpoint comum na linha do jump (0040EC14) apertando F2. Aperte F9 para continuar até chegarmos no jump. Assim que chegar, aperte F7 uma vez para realizar o pulo para o entry point original.

```

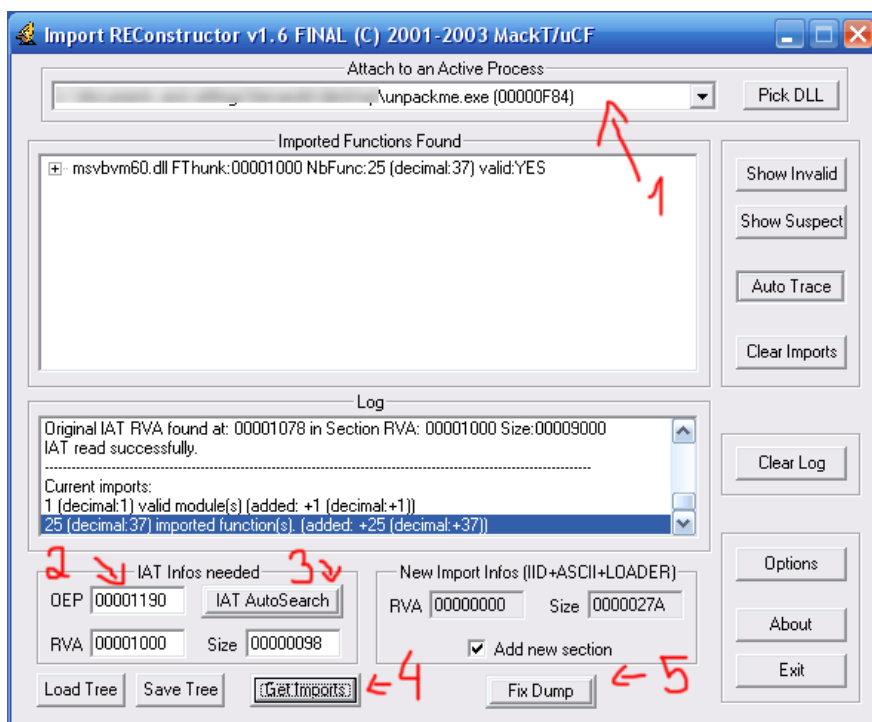
00401190  68 A09E4000 PUSH UnpackMe.00409EA0
00401195  E8 EFFFFFFF CALL UnpackMe.00401188
0040119A  0000      ADD BYTE PTR DS:[EAX],AL

```

Se você fez tudo certo, agora deve estar no endereço 00401190 que nos mostra o código original do .exe descomprimido. O que nós precisamos fazer é pegar o aplicativo descompactado da memória e remontar um novo executável a partir dele, indicando o ponto de início correto (1190). Para fazer isso, vamos utilizar o OllyDump. Com o programa ainda congelado na linha 1190, vá em "Plugins->OllyDump->Dump debugged process".



O Olly já calcula tudo sozinho para você, inclusive o OEP, mas se quiser garantir clique em "Get EIP as OEP". Desmarque também a caixa "Rebuild Import", pois nós vamos utilizar outro programa para remontar a base de Imports. Clique em "Dump" e escolha o nome e o local onde você deseja salvar o arquivo descompactado. Eu utilizei o nome "Dumped.exe". Provavelmente você ficou muito entusiasmado, já executou esse arquivo criado e se deparou com um erro de falha de inicialização. Isso ocorre porque o UPX desorganiza a seção de Imports do executável, e nós precisamos remontá-la. Para isso, vamos usar o ImpRec, que eu falei no início do tutorial. Deixe o Olly aberto da maneira que está, não feche ele. Execute também o .exe original (que veio com o zip, o mesmo que está rodando no Olly). Agora inicie o ImpRec e configure da maneira como mostrada na figura:



Em 2, você deve colocar o valor adquirido na caixa de texto "Modify", da janela de Dump do Olly. Ao clicar em Get Imports (4), deve aparecer alguns itens na lista "Imported Functions Found". Todos os itens devem aparecer com um YES no final, caso contrário ele não conseguirá montar a tabela de imports corretamente. A última coisa a fazer é clicar em Fix Dump (5). Ao clicar, ele pede por um arquivo executável. Selecione aquele que foi criado no Olly, que eu dei o nome de "Dumped.exe". Logo após selecionar o arquivo, ele já vai gerar o executável descompactado definitivo (contendo um underline no final do nome: Dumped_.exe) e vai mostrar no Log do ImpRec que o arquivo foi criado com sucesso. Agora sim você já pode fechar o Olly, todo o resto da parafernália e executar o "Dumped_.exe". Se deu tudo certo, você deve ter recebido uma mensagem animadora do unpackme :P Se quiser comprovar que foi descompactado, faça um novo scan com o PEiD no "Dumped_.exe" e veja (ou simplesmente compare o tamanho dos arquivos) ;D

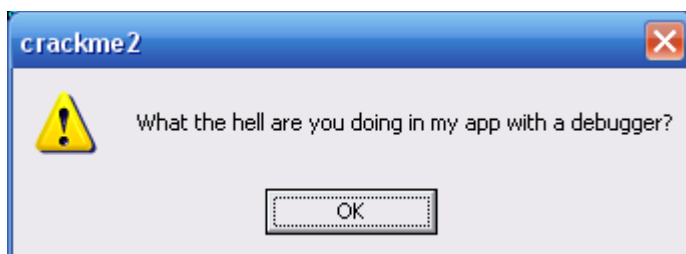
Tutorial - Engenharia Reversa

Parte 7



ARMADILHAS E ALTERAÇÃO DE CÓDIGO

Alguns aplicativos possuem certas proteções contra programas de debugging (OllyDBG, SoftICE, WinDbg, etc.), como é o caso do nosso alvo. A proteção que ele usa é bem simples, simplesmente gera uma exceção (um desvio no código, que diversas vezes gera um erro no programa) que exibe uma mensagem de texto avisando que estamos usando um debugger e encerra o programa após isso.



Nosso alvo:

Download: [crackme2.zip](#)

(créditos ao **mucki**)

Abra o alvo no Olly e aperte F9 para iniciar a execução. Logo de cara você nota que o programa não inicia e pausa em 401047. Repare no rodapé do Olly que apareceu algo assim:

Singlestep event at crackme2.00401047 -use Shift+F7/F8/F9 to pass exception to program

Isso indica que o programa encontrou uma "exception" e parou antes de ela ser executada. Se você apertar F9, ele vai continuar e vai exibir a MessageBox da imagem logo acima, e encerra o programa após isso. O Olly permite que você "pule" a exceção e continue rodando o programa normalmente, basta apertar Shift+F9. Agora você tem o programa funcionando corretamente. Já passamos o sistema anti-debugging, vamos então prosseguir e alterar o nosso código. O que o aplicativo faz é pedir um nome e senha, comparar e verificar se as informações são coerentes. Se for, ele exibe uma mensagem dizendo que o código está correto. Caso não seja, ele indica que o serial não é válido.

Ao invés de setar um breakpoint na chamada da função MessageBox (para nos levar ao local onde a key é checada), vamos usar uma outra aproximação (a da MsgBox é mais direta, mas vamos variar um pouco). Vamos procurar por outras funções clássicas. Aperte CTRL+N para abrir a janela com os nomes das funções usadas. Analise a lista e repare na função "Istrcmp". O que ela faz é comparar 2 strings, se forem iguais, ela seta a "Zero Flag" para verdadeiro. Selecione a "Istrcmp" na lista e aperte ENTER. Uma nova janela apareceu com todas as ocorrências do uso dela no código. Como só tem 2 ocorrências, é só tentar em uma das duas para ver se fomos levados para o local correto. Selecione a primeira ocorrência e aperte ENTER novamente. Caímos aqui

0040121F	. 2BC1	SUB EAX, ECX	<ZIX>
00401221	. 0FAFC0	INUL EAX, EAX	<ZIX>
00401224	. 50	PUSH EAX	Format = "CM2-%IX-%IX"
00401225	. 51	PUSH ECX	s = crackme2.004062B6
00401226	. 68 E1604000	PUSH crackme2.004060E1	wsprintfA
0040122B	. 68 B6624000	PUSH crackme2.004062B6	Count = 4B (75.)
00401230	. E8 9E010000	CALL <JMP.&user32.wsprintfA>	Buffer = crackme2.004060BA
00401235	. 6A 4B	PUSH 4B	ControlID = 2
00401237	. 68 BA604000	PUSH crackme2.004060BA	hwnd
0040123C	. 6A 02	PUSH 2	GetDlgItemTextA
0040123E	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	String2 = "
00401241	. E8 A2010000	CALL <JMP.&user32.GetDlgItemTextA>	String1 = ""
00401246	. 68 BA604000	PUSH crackme2.004060BA	Istrcmp
0040124B	. 68 B6624000	PUSH crackme2.004062B6	
00401250	. E8 0E010000	CALL <JMP.&kernel32.IstrcmpA>	
00401255	. 75 16	JNZ SHORT crackme2.00401260	
00401257	. 6A 00	PUSH 0	Style = MB_OK MB_APPLMODAL
00401259	. 68 00604000	PUSH crackme2.00406000	Title = "crackme2"
0040125E	. 68 30604000	PUSH crackme2.00406030	Text = "Valid serial - now write a keygen!"
00401263	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	hOwner
00401266	. E8 89010000	CALL <JMP.&user32.MessageBoxA>	MessageBoxA
0040126B	. EB 14	JMP SHORT crackme2.00401281	
0040126D	. 6A 10	PUSH 10	Style = MB_OK MB_ICONHAND MB_APPLMODAL
0040126F	. 68 00604000	PUSH crackme2.00406000	Title = "crackme2"
00401274	. 68 60604000	PUSH crackme2.00406060	ASCII "Wrong serial - try again!"
00401279	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	hOwner
0040127C	. E8 73010000	CALL <JMP.&user32.MessageBoxA>	MessageBoxA
00401281	. 5D	POP EBX	
00401282	. 5E	POP ESI	
00401283	. 5F	POP EDI	

Hum, parece que estamos no local certo. Repare nas funções "em volta" dela. Temos diversas funções típicas, como a formatação de um texto no formato de serial (wsprintfA), GetDlgItemTextA (pega o nosso serial para comparar), e as 2 MsgBox indicando se a serial é válida ou não.

Que tal alterar o código da MsgBox? Ao invés de ela exibir a mensagem de serial inválida, ela exibe o serial correto. Vamos fazer isso.

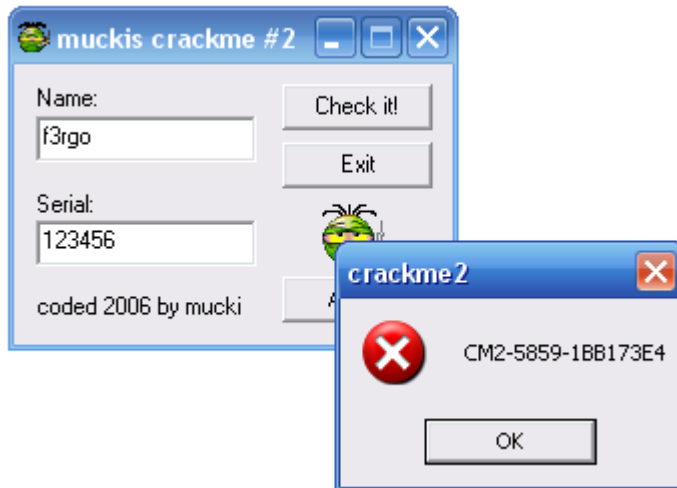
Repare na função "IstrcmpA" novamente, ela precisa de 2 argumentos (as 2 strings) para serem comparadas. Uma das strings obviamente é a serial que nós digitamos e a outra é a serial válida. Uma das strings está armazenada em 004060BA (primeiro argumento puxado) e a outra está em 004062B6 (segundo argumento). Em qual dos 2 argumentos está a serial correta? Ou você analisa o código da geração da key (logo acima) ou você vai por tentativa e erro (testa com um endereço, e depois com o outro). Eu já sei que a serial verdadeira está no endereço 4062B6 e a serial que nós digitamos está no 4060BA. Então o que nós precisamos fazer é alterar o texto da MsgBox: ao invés de aparecer "Wrong Serial - try again!", ele exibe o serial correto. Fazer isso é muito fácil, vá até o local onde ele puxa o texto "Wrong serial..." da msgbox (401274):

```
00401274 68 60604000 PUSH crackme2.00406060 ; ASCII "Wrong serial - try again!"
```

De um duplo clique sobre "PUSH crackme2.00406060" e altere para "PUSH 004062B6" :

```
00401274 68 60604000 PUSH crackme2.004062B6
```

O que nós fizemos foi: ao invés de chamar o endereço 406060 que contém "Wrong...", nós chamamos o endereço que contém a serial correta (4062B6). Para testar, basta salvar as modificações em um novo executável. Clique com o botão direito sobre a janela principal, vá em "Copy to Executable->All Modifications->Copy All". Na nova janela, clica novamente com o botão direito e escolha "Save File". Salve o novo arquivo, execute, digite um nome e um serial qualquer e veja a MessageBox que aparece:



Bingo! Ao invés de mensagem de serial incorreto, apareceu o serial verdadeiro. Anote e use-o! Como tarefa, decifre o algoritmo do serial (começa em 004011F6). Não é complicado, simplesmente realiza uma série de operações básicas com cada letra do nome digitado. Antes de finalizar, só quero lembrar que nesse caso, o código poderia ser alterado sem sequer ter controlado a exceção, só quis explicar aquilo para que em outros casos você não desista somente porque o programa parou de funcionar no debugger ;D



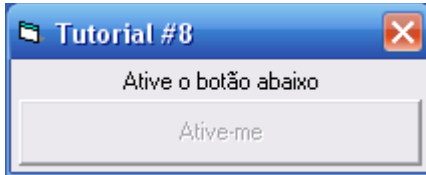
ATIVANDO CONTROLES

Este tutorial irá ensinar como alterar o estado de um controle (um Command Button, nesse caso) em qualquer aplicativo criado em Visual Basic. Não vamos utilizar o Olly (pode ser usado também, mas não vale a pena).

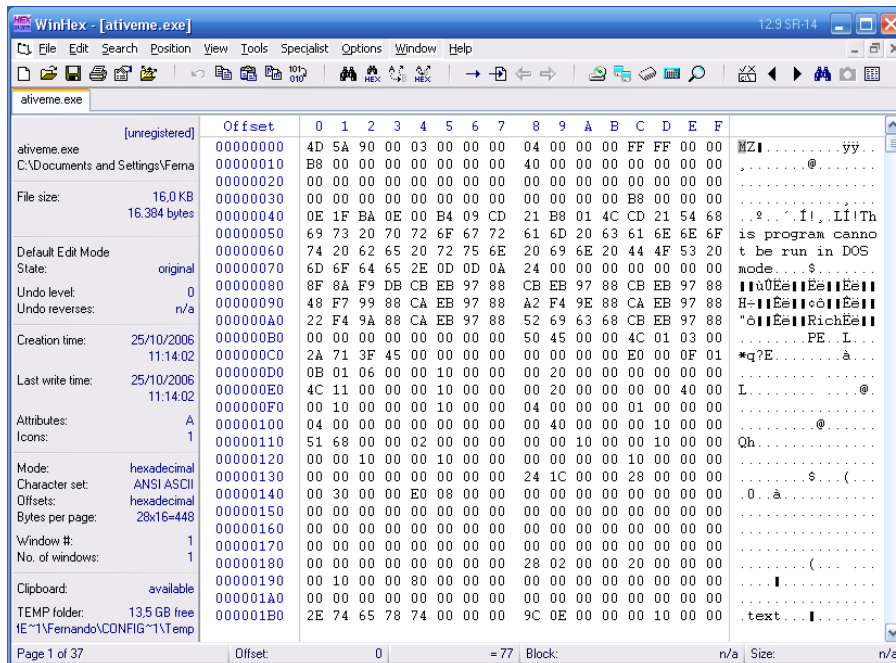
Em vez dele, vamos usar um editor hexadecimal qualquer, que tenha alguma ferramenta de busca embutida. Eu gosto do [WinHex](#)

Como sempre, baixe o alvo (codado em VB)

Download: [ativeme.zip](#)



Antes de tudo, vou explicar um pouco sobre o código do VB. Ao contrário do que muitos pensam, aplicativos em VB, se compilados em Native Code, geram códigos normais de assembly, que podem ser abertos em qualquer debugger. A diferença está que além do aplicativo, ele utiliza um "linker", uma dll (MSVBM60.dll) que contém todas as funções que o VB utiliza. Assim, quando você faz uma comparação de strings por exemplo, ele chama a função "vbaStrCmp" da MSVBM60.dll. Devido a essa conexão entre o executável e a DLL, eles são normalmente (quando compilados em Native Code), mais complicados de se fazer o debug. Apesar de ser mais complicado de reverter códigos assembly criados pelo VB, ele tem algo que também facilita a sua edição. Todas as informações sobre os controles, como o estado, nome, cor, texto e tamanho, ficam armazenados em forma de texto dentro do executável. Isso significa que se nós procurarmos pelo texto de algum botão em um editor hexadecimal (ou até mesmo abrir ele no bloco de notas e procurar), nós vamos poder alterar as suas propriedades básicas. Se você já executou o nosso alvo, deve ter visto que o objetivo é habilitar um Command Button, então vamos lá. Abra o alvo em algum editor Hexadecimal.



Como eu disse logo acima, ele armazena as informações dos controles em forma de texto no executável. Que tal procurarmos pelo texto do botão (Atime-me)? Vá em "Search->Find Text". Selecione o modo ASCII e procure por "Atime-me" (se as áspas). O WinHex deve ter o levado até o offset 125A (linha 1250, coluna A).

```
00001250 | 41 74 69 76 65 00 04 01 08 00 41 74 69 76 65 2D | Atime....Atime-
00001260 | 6D 65 00 04 3C 00 2C 01 C7 0B EF 01 08 00 11 00 | me.<...Ç.i....
```

Na parte central você pode ver os valores em hex de cada byte do arquivo, e no lado direito, os caracteres ASCII correspondentes a esses valores. Não vem ao caso explicar o que é cada pedaço ali, a parte importante mesmo é o que está marcado em vermelho. O estado do botão vem logo após o valor que define a sua altura, seguido do byte 08 (SEMPRE). A altura do botão no caso é indicada pelo "EF 01" (na ordem reversa de byte fica 01EFh = 495d), seguido do byte 08 e de um byte 00. 00 indica que o botão está desabilitado (Enabled = False), e 01 indica que está ativo (Enabled = True). Então basta alterar o byte 00 para 01



Agora é só salvar o arquivo pelo WinHex ("File->Save"). Se você executar o alvo novamente, o botão estará habilitado ;D

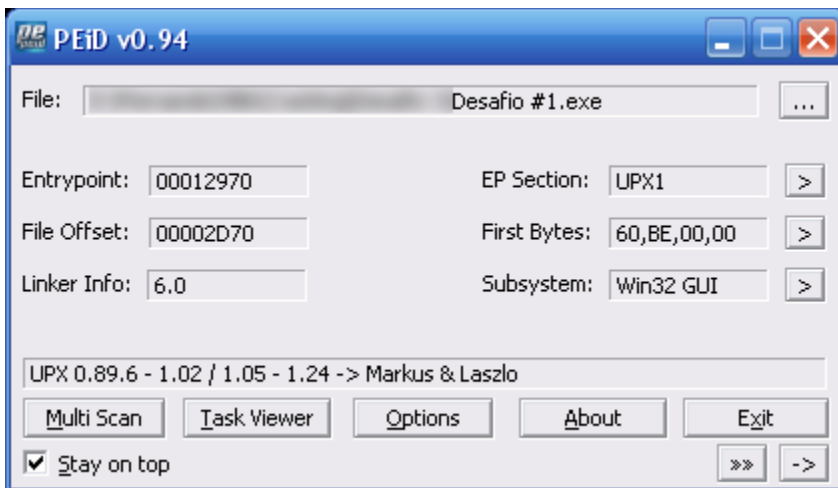


SOLUÇÃO DO DESAFIO #1

Neste tutorial vou explicar uma das formas de solucionar o primeiro desafio (pode encontrá-lo no índice de tutoriais), Vamos direto ao assunto. A primeira coisa a fazer é abrir o alvo e dar uma analisada geral. De início já notamos que o botão para verificar o serial está desabilitado. Depois, ao fechar, aparece uma Nag-screen, que devemos tirar. Vamos ver também se foi usado algum tipo de packer nele.

Parte 1 - Descompactando

Abra-o no PEiD e veja que ele foi compactado com o UPX (Ultimate Packer for eXecutables).



Se você leu todos os tutoriais, você deve conseguir resolver todos os problemas encontrados até agora. Vamos começar descomprimindo o arquivo. Abra o alvo no Olly. Ele posiciona você no entry point do packer (repare no PUSHAD característico). Aperte F7 uma vez e o valor do registrador ESP irá mudar. Clique com o botão direito no valor de ESP e selecione "Follow in Dump". Estamos monitorando o endereço armazenado pelo ESP na janela de dump do Olly (parte inferior). Clique com o botão direito no primeiro byte da janela de dump (38), selecione "Breakpoint->Hardware, on write->DWord". Quando o alvo for escrever algo neste endereço de memória, o Olly congela o programa, e isso geralmente nos leva bem próximo do final do processo de descompressão. Aperte F9 para continuar a execução e em seguida nós paramos aqui:

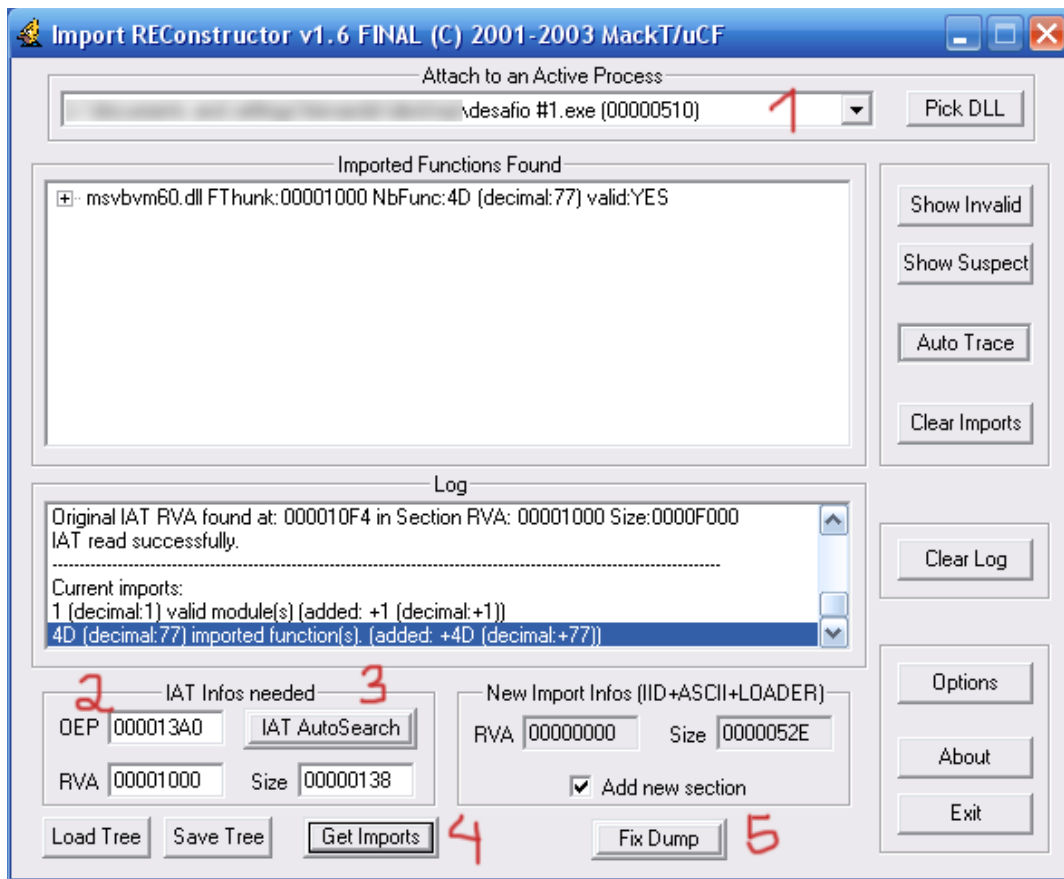
```

00412AF6  61          POPAD
00412AF7  8D4424 80    LEA EAX,DWORD PTR SS:[ESP-80]
00412AFB  6A 00      PUSH 0
00412AFD  39C4      CMP ESP,EAX           ; Paramos aqui
00412AFF  75 FA      JNZ SHORT Desafio_00412AFB
00412B01  83EC 80    SUB ESP,-80
00412B04  E9 97E8FEFF JMP Desafio_004013A0 ; Como visto no tutorial de UPX, este jump nos leva ao EP original do programa

```

Para podermos fazer o dump do programa descomprimido, precisamos chegar nesse último jump e executá-lo. Adicione um breakpoint (F2) no último jump (00412B04) e aperte F9. Ótimo, paramos no jump e agora precisamos executá-lo. Aperte F7 uma vez e nós fomos levados até o OEP (Original Entry Point), ou seja, estamos no início do código do programa descompactado. Vamos "extrair" o programa descompactado da memória e jogar em um executável a parte. Utilizando o plugin OllyDump, vá em "Plugins->OllyDump->Dump Debugged Process". Na janela que aparecer, clique em "Get EIP as OEP" e desmarque "Rebuild Import". Clique em "Dump" e salve com qualquer nome (eu utilizei DUMPED.exe, então vou me referir a ele assim).

Ainda não acabamos, precisamos remontar a base de Imports, para isso vamos usar o ImpRec (não feche o Olly com o nosso programa).



No passo número 5 selecione o arquivo exportado do Olly (DUMPED.exe). Ele vai criar um arquivo DUMPED_.exe que é o nosso arquivo final, descomprimido. Caso não tenha entendido o processo utilizado, leia o tutorial #6, que explica detalhadamente cada passo utilizado aqui.

Parte 2 - Ativando o botão

Resolvi fazer esta parte primeiro para não ter que ficar trocando de programas mais tarde. Habilitando o botão agora, depois eu só preciso utilizar o Olly, mais nada. Abra o DUMPED_.exe no WinHex e faça uma busca ("Search->Find Text") no modo ASCII pelo texto do botão desativado: "Registrar" (sem as aspas). Fomos levados até o Offset 6CC3, onde tem a palavra "Registrar".

```

00006CC0 | 01 09 00 52 65 67 69 73 74 72 61 72 00 04 54 06 | ...Registrar..T.
00006CD0 | EC 04 EB 05 77 01 08 00 11 04 00 FF 03 1F 00 00 | i.ë.w....ÿ...

```

Marcado em violeta está os bytes importantes que nós procurávamos. O 08 indica a propriedade "Enabled" e o 00 em seguida indica que ele está desativado (Enabled = False). Para reverter isso, troque o 08 00 por 08 01 e salve o arquivo pelo WinHex ("File->Save"). Agora temos nosso botão habilitado.

Leia o tutorial #8 para ver uma explicação mais detalhada do processo.

Parte 3 - Desativando a proteção contra o Olly

Agora voltamos ao Olly. Apra o DUMPED_.exe já editado com o botão funcionando. Uma mensagem avisando sobre o EP fora do módulo do programa é mostrada. Dependendo do caso isso dificulta o processo, mas nesse caso a única coisa ruim é que ele não permite salvar todas as modificações no executável, apenas as seleções. Aperte F9. Uma mensagem de texto aparece indicando que estamos usando o Olly. Vamos desativá-la para que possamos fazer o debug. Reinicie o alvo no Olly (CTRL F2) e digite "bpx rtcMsgBox" na linha de comando ("Plugins->Command Line"). Isso seta um breakpoint na chamada da função MsgBox. Inicie a execução (F9) e o Olly congelou no local onde a mensagem "Anti-Olly" aparece (4008B3A). Temos que encontrar uma maneira de desviar o código para que a MsgBox não apareça. Analise o código um pouco mais acima. Repare que no endereço 408AF1 nós temos um salto condicional que pula para uma área do código logo DEPOIS da exibição da MsgBox, ou seja, se o salto for realizado, a MsgBox não aparece. Podemos então forçar para que o salto sempre ocorra. Para isso basta alterar de JNZ para JMP:

```
00408AF1 75 6E JNZ SHORT DUMPED_.00408B61
```

Troque para (selecione a linha e aperte espaço):

```
00408AF1 EB 6E JMP SHORT DUMPED_.00408B61
```

Selecione esta linha modificada, clique com o botão direito e vá em "Copy to Executable->Selection". Em seguida, na janela que aparecer, clique novamente com o botão direito, vá em "Save File" e selecione o próprio executável (DUMPED_.exe). Depois de salvar, aperte CTRL F2 para resetar o aplicativo. Se você apertar F9, o programa agora abre normalmente, não exibindo a MsgBox. ;D

Parte 4 - Buscando o serial correto

Com o programa rodando na tela onde pede por um Nome e Serial, digite "bpx __vbaStrCmp" no Command Line para setar um breakpoint na função que compara 2 strings (serial verdadeira com a serial que nós digitamos). Digite um nome qualquer com mais de 4 letras (veja mais adiante) e uma serial qualquer também (eu utilizei F3rGO! e 132456, respectivamente) e clique em "Registrar". Paramos na chamada da função.

00408621	51	PUSH ECX	
00408622	52	PUSH EDX	
00408623	FF 15 8C104000	CALL DWORD PTR DS:[&msubum60._vbaStrCmp]	msubum60._vbaStrCmp

Repare que ele puxa 2 argumentos e em seguida chama a função. Esses argumentos é o que a função vai comparar (no caso, ele vai comparar ECX com EDX). Veja na tela de registradores (lado direito). No meu caso, ECX contém 123456 e EDX contém 418. Ele está comparando 123456 (nosso serial falso) com 418 (que é o serial verdadeiro para o nome digitado).

Bingo! Para o nome F3rGO!, o serial é 418.

Quanto ao número mínimo de letras no nome, basta analisar o código mais acima. Antes de todo o processo de geração do serial começar, ele compara o texto digitado com o valor 4 (endereço 408459).

Parte 5 - Nag Screen

Já encontramos o serial. Agora só falta remover a Nag-screen que aparece quando o programa se encerra. Digite novamente "bpx rtcMsgBox" para setar breakpoints nos locais onde a função "rtcMsgBox" é chamada. Feche a janela do nosso alvo (pelo X no canto da janela mesmo). O Olly congelou novamente na chamada da função, para exibir a Nag-screen. Como essa MsgBox não depende de uma condição para ser exibida (como no caso do Anti-Debug), não podemos trocar um JNZ por JMP ou coisa do tipo. O que nós temos que fazer é anular as linhas de código que exibem a MsgBox.

```

00408C51 50          PUSH EAX
00408C52 8055 CC    LEA EDX,DWORD PTR SS:[EBP-34]
00408C55 51          PUSH ECX
00408C56 52          PUSH EDX
00408C57 8045 DC    LEA EAX,DWORD PTR SS:[EBP-24]
00408C5A 6A 10     PUSH 10
00408C5C 50          PUSH EAX
00408C5D FF15 5C104000 CALL DWORD PTR DS:[<&msvbvm60.rtcMsgBox] msvbvm60.rtcMsgBox

```

Temos que anular a linha que chama a função (CALL DWORD PTR...) no endereço 408C5D e todos os 5 argumentos que ela puxa. Para anular os comandos, temos que preencher com NOPs (NO oPeration). Clique com o botão direito sobre o "CALL DWORD..." (408C5D), vá em "Binary->Fill with NOPs". Faça o mesmo para as 5 linhas anteriores a ela que contém "PUSH"

```

00408C51 90          NOP
00408C52 8055 CC    LEA EDX,DWORD PTR SS:[EBP-34]
00408C55 90          NOP
00408C56 90          NOP
00408C57 8045 DC    LEA EAX,DWORD PTR SS:[EBP-24]
00408C5A 90          NOP
00408C5B 90          NOP
00408C5C 90          NOP
00408C5D 90          NOP
00408C5E 90          NOP
00408C5F 90          NOP
00408C60 90          NOP
00408C61 90          NOP
00408C62 90          NOP

```

Agora só precisamos salvar as alterações. Selecione do endereço 00408C51 até o 00408C62 (região que abrange todas as modificações) usando SHIFT. Clique novamente com o botão direito sobre a seleção, vá em "Copy to Executable->Selection" e na janela que abrir, clique com o botão direito e vá em "Save File". Pode sobrescrever o arquivo DUMPED_exe novamente. Pronto! Terminamos! Fizemos tudo o que foi pedido. Descompactamos, ativamos o botão, removemos a proteção anti-debug, encontramos o serial e anulamos a nag-screen. Se você conseguiu acompanhar tudo, parabéns, em breve mais desafios. Caso teve alguma dificuldade em alguma parte, leia os outros tutoriais encontrados no índice da página. Lá os assuntos vistos nesse tutorial são tratados com muito mais abrangência.

Até a próxima!

F3rGO!

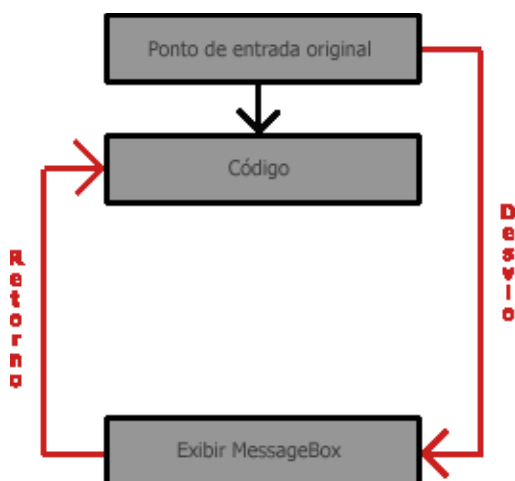
Tutorial - Engenharia Reversa

Parte 10



INJETANDO CÓDIGOS

Neste tutorial vamos ver como injetar novos códigos em um executável já compilado. Vamos modificar o bloco de notas do windows para que exiba uma simples mensagem de texto (pois é uma das função mais fáceis que o Windows fornece) quando for iniciado. Antes de tudo, faça uma cópia do bloco de notas (C:\WINDOWS\notepad.exe) para uma pasta qualquer (vamos usar essa cópia, pra preservar o original). O que nós vamos fazer é o seguinte: desviar o início do código do notepad para outro local, exibir a mensagem e depois retornar ao ponto original.



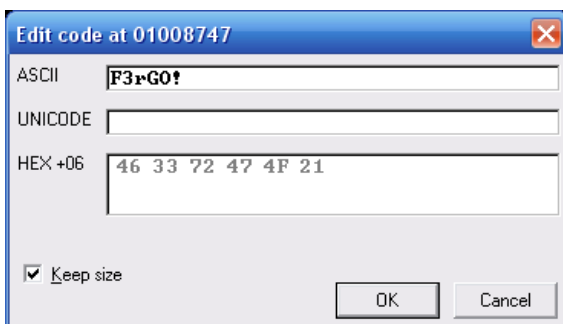
Para fazer isso, precisamos encontrar na seção do código do notepad, um local vazio, que não altere a funcionalidade do programa, o que é chamado de "Code-cave". Abra o notepad.exe no Olly e role o código para baixo. Lá pelo offset 8747 você deve encontrar o início da code-cave, uma seção sem código algum, apenas com valores 00:

```
01008726 . 60 57 00 ASCII "nul",0 Término do código
01008729 . 00 DB 00
0100872A . 7A01 DW 017A
0100872C . 47 65 74 57 61 ASCII "GetWindowText",0
0100873B . 00 DB 00
0100873C . 55 53 45 52 31 ASCII "USER32.dll",0
01008747 00 DB 00
01008748 00 DB 00
01008749 00 DB 00
0100874A 00 DB 00
0100874B 00 DB 00
0100874C 00 DB 00
0100874D 00 DB 00
0100874E 00 DB 00
0100874F 00 DB 00
01008750 00 DB 00
01008751 00 DB 00
01008752 00 DB 00
01008753 00 DB 00
01008754 00 DB 00
01008755 00 DB 00
01008756 00 DB 00
```

Essa região a partir de 8747 é a que vamos utilizar para escrever nosso código. Como disse, vamos adicionar uma MessageBox no início da execução. Vamos dar uma olhada na função MessageBoxA para ver o que nós precisamos:

```
int MessageBox (
    HWND hWnd, // handle of owner window
    LPCTSTR lpText, // address of text in message box
    LPCTSTR lpCaption, // address of title of message box
    UINT uType // style of message box
);
```

Para chamar a função, precisamos passar 4 argumentos. hWnd indica à qual janela a MsgBox pertence. lpText e lpCaption são o texto da mensagem e o título da janela, respectivamente. E por último, uType indica o estilo da janela, como número de botões e ícones. Precisamos gravar o texto e o título da janela em algum lugar, e esse lugar é a própria code-cave. Vamos ao título primeiro. A partir do início da code-cave, selecione o número de linhas que você vai usar para armazenar o título, sendo que cada linha que você selecionar vai ocupar o espaço de 1 caractere. Vou usar como título "F3rGO!", que contém 6 caracteres. Então seleciono 6 linhas a partir do início da CV (code-cave), usando SHIFT, cliço com o botão direito sobre a seleção e em seguida "Binary->Edit". Em seguida, marque a caixa "Keep Size" e digite a sua mensagem no campo ASCII:



Aperte OK. O Olly primeiramente entende o que você digitou como uma sequência de comandos, por isso a sua mensagem não apareceu corretamente na janela de código. Aperte CTRL+A para o Olly analisar o código novamente e aí sim você deve ver a sua string corretamente. Repita o mesmo processo para o texto da janela (selecionando as linhas a partir da linha que contém o seu título). Eu usei como texto "Notepad modificado!", que contém 19 caracteres (espaços contam como caracteres). Selecione as 19 linhas e faça o mesmo que você fez para o título. No final, terá algo assim:

```
0100873B . 00 DB 00
0100873C . 55 53 45 52 31 ASCII "USER32.dll",0
01008747 . 46 33 72 47 41 ASCII "F3rGO!",0
0100874E . 4E 6F 74 65 71 ASCII "Notepad modificado"
0100875E . 64 6F 21 00 ASCII "do!",0
01008762 . 00 DB 00
01008763 . 00 DB 00
```

Já temos armazenado as nossas strings, então podemos chamar a função. Lembre-se que os argumentos da função são puxados na ordem inversa, ou seja, devemos usar o comando PUSH partido da uType para o hWnd.

Escolha um offset qualquer da CV (pode ser logo abaixo das strings). Eu usei o 01008763. Escreva o código da chamada da função (na hora de puxar o título e texto, indique o endereço de cada um no código (01008747 e 0100874E no meu caso)). Para escrever o código, basta dar 2 cliques sobre a linha e entrar com o comando. Seu código deve ficar assim (os comentários eu adicionei):

```

01008747 . 46 33 72 47 41 ASCII "F3rG0*",0
0100874E . 4E 6F 74 65 71 ASCII "Notepad modifica"
0100875E . 64 6F 21 00 ASCII ".dot",0
01008762 00 DB 00
01008763 6A 00 PUSH 0
01008765 68 47870001 PUSH NOTEPAD.01008747
0100876A 68 4E870001 PUSH NOTEPAD.0100874E
0100876F 6A 00 PUSH 0
01008771 E8 957DD576 CALL USER32.MessageBoxA
01008776 00 DB 00
01008777 00 DB 00
01008778 00 DB 00

```

```

uType -> 0 -> Somente um botao OK
ASCII "F3rG0*"
ASCII "Notepad modificado*"
hWnd -> 0 -> Numero da janela dona da msgbox
Chama a funcao da msgbox

```

Já temos o código da msgbox, precisamos desviar o início do programa 01008763 para chamar a mensagem e depois retornar à sequência de código original. Ainda não vamos adicionar o JMP de retorno após a chamada por motivos que vou explicar adiante. Memorize o endereço 01008763, pois é para ele que vamos desviar (já que é onde tem início a chamada da msgbox). Clique com o botão direito em qualquer lugar do código e em seguida "Goto->Origin *". Fomos levado até o EntryPoint (ponto de entrada) do programa.

```

0100739D 6A 70 PUSH 70
0100739F 68 98180001 PUSH NOTEPAD.01001898
010073A4 E8 BF010000 CALL NOTEPAD.01007568
010073A9 33DB XOR EBX,EBX

```

Vamos ter que alterar o PUSH 70 para um jump até o endereço 01008763. Antes de alterar, anote os comandos desses 2 primeiros offsets (PUSH 70 e PUSH NOTEPAD...). Já explico o porquê. Agora dê um duplo clique sobre o PUSH 70 e altere para JMP 01008763 (marque a opção Fill with NOP's). Certo, desviamos o nosso código, mas repare que foram perdidos 2 comandos devido a esse jump (os 2 que você anotou). Isso se dá ao fato que o comando JMP ocupa o espaço do PUSH 70 e de parte do outro PUSH, completando com NOPs os bytes que sobraram. Não podemos deixar assim, precisamos de alguma maneira adicionar esses 2 comandos perdidos em algum lugar. Onde? Na code-cave após a chamada da MessageBoxA. Antes de voltar para a MessageBox, anote o offset do CALL NOTEPAD.01007568 (010073A4), pois depois da msgbox ser exibida, nós vamos ter que pular nesse endereço, para que a execução continue. Volte para a região da sua MessageBox e adicione os 2 comandos anotados (quando for adicionar o segundo comando, deixe somente o push com o endereço, remova o texto "NOTEPAD."). Agora podemos realizar o jump de retorno ao código normal. Basta adicionar um novo JMP após o fim do código injetado para o endereço 010073A4 (offset que você acabou de anotar). No final das contas, seu código deve ter ficado semelhante a isso:

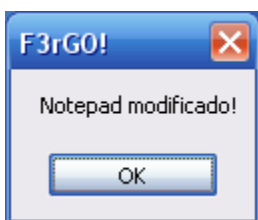
```

0100872C . 47 65 74 57 61 ASCII "GetWindowText",0
0100873B . 00 DB 00
0100873C . 55 53 45 52 31 ASCII "USER32.dll",0
01008747 . 46 33 72 47 41 ASCII "F3rGO!",0
0100874E . 4E 6F 74 65 74 ASCII "Notepad modifica"
0100875E . 54 6F 21 00 ASCII ".dot",0
01008762 00 DB 00
01008763 6A 00 PUSH 0
01008765 68 47870001 PUSH NOTEPAD.01008747
0100876A 68 4E870001 PUSH NOTEPAD.0100874E
0100876F 6A 00 PUSH 0
01008771 E8 957DD576 CALL USER32.MessageBoxA
01008776 6A 70 PUSH 70
01008778 68 88160001 PUSH NOTEPAD.01001696
0100877D ^ E9 22ECFFFF JMP NOTEPAD.010073A4
01008782 00 DB 00
01008783 00 DB 00
01008784 00 DB 00
01008785 00 DB 00

```

uType -> 0 -> Somente um botao OK
ASCII "F3rGO!"
ASCII "Notepad modificado!"
hwnd -> 0 -> Numero da janela dona da msgbox
Chama a funcao da msgbox
Primeiro comando movido
Segundo comando movido
Retorno ao endereco de execucao normal

Pode salvar seu arquivo ("Botão Direito->Copy to Exetuable->All Modifications->Copy All->Botão Direito->Save File") e testar. Se você fez tudo certo, deve ter visto isso ao iniciar o seu bloco de notas modificado



Até a próxima!

F3rGO!



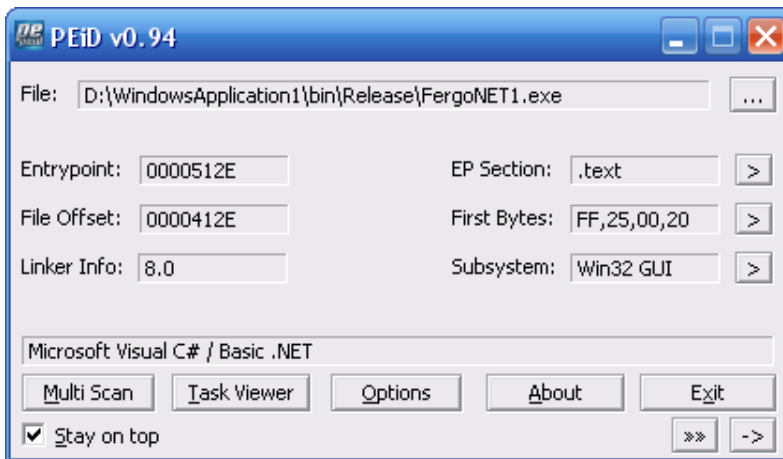
REVERTENDO APLICATIVOS .NET

Neste tutorial vamos nos distanciar um pouco do Olly para analisarmos um aplicativo feito em .NET. Apesar de facilitar muito a programação (tanto visual quanto linhas de código), aplicativos que usam o .NET Framework possuem uma grande falha: a facilidade de se reverter o código. Esses executáveis não são compilados em código nativo (transformado pra assembly), ao invés disso, é utilizada uma linguagem chamada IL (Intermediate-Level), que nada mais é do que o seu código desmembrado em 1 comando por linha. Quando você inicia um desses aplicativos, o Framework tem que compilar o aplicativo na hora (on the fly), e isso explica também a lentidão para iniciar qualquer executável programado com o Framework.

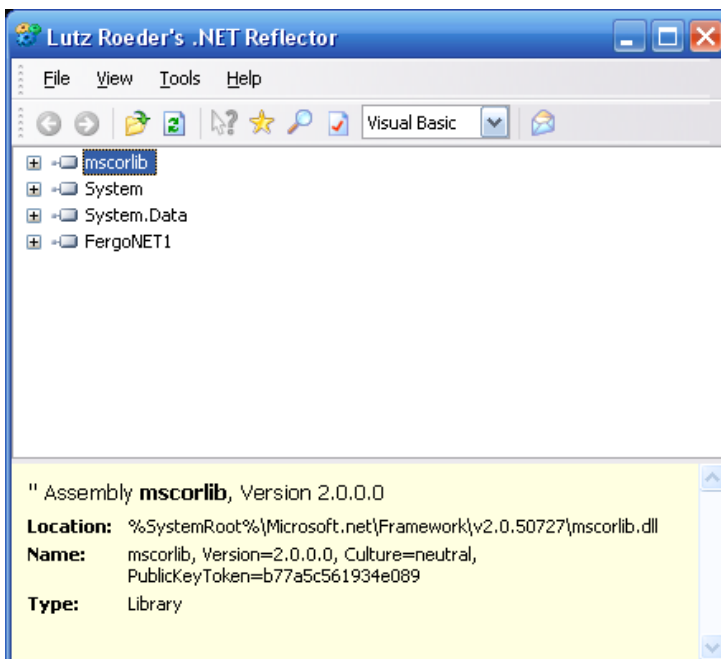
Não importa a linguagem que você utilizou (VB, C#, ...), no final das contas, tudo vira o mesmo IL-Code. O uso do .NET Fuscator (Obfuscator) também não adianta muito, já que existem dezenas de aplicativos que conseguem reverter a "criptografia". Nós não vamos fazer debug nem alteração no código desta vez, vamos só analisar. Para isso, baixe nosso alvo:

-Download: [fergonet.zip](#)

Como eu disse, o Olly não tem a capacidade de fazer o debug/disassembly de aplicativos .NET. Vamos utilizar um programa chamado .NET Reflector. Ele não tem a capacidade de fazer debug, apenas reverte o código para sua linguagem nativa (isso mesmo!). Como sempre, uma análise no PEiD para verificar a origem do executável.

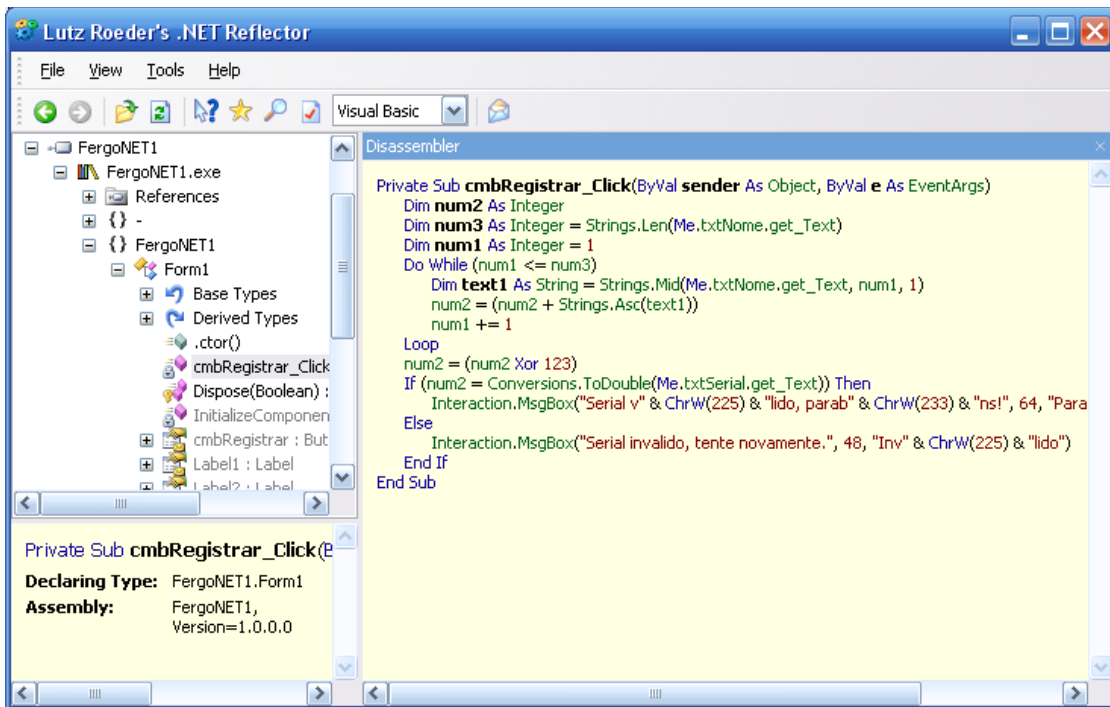


Inicie o .NET Reflector e abra o nosso alvo dentro dele. (se for questionado sobre a versão do Framework, selecione a 2.0).



Vá expandindo a árvore até chegar dentro do Form1, que contém todos os eventos, controles, etc... (FergoNET1\FergoNET1.exe\FergoNET1\Form1). Interessante, conseguimos já ver todos os controles e eventos (inclusive com seus nomes originais). Dentre os elementos da árvore, um deles chama atenção: cmbRegistrar_Click(Object, EventArgs) : Void

Clique sobre o botão direito sobre ele e selecione "Disassembly". Caso apareça alguma mensagem, simplesmente ignore-a (Skip). Se você não estava acreditando, agora vai acreditar. O código do programa está ali, escrito na linguagem nativa (voce pode selecionar uma delas no drop-down ali na toolbar (eu escolhi Visual Basic)



Incrível não? Não precisamos nem rachar a cabeça pra entender o algoritmo em assembly, ele já está numa linguagem de alto nível. Basta copiar esse código para o VisualStudio, fazer algumas alterações e temos um keygen ;D Caso não tenha entendido, o serial é gerado somando os valores ASCII de cada caractere do nome e depois executando um XOR com 123 sobre essa soma. Se quiser analisar o seu código na IL, selecione IL no drop-down da toolbar. O código não fica claro como este, mas também não chega a parecer com assembly. Compare o código IL com alguma outra linguagem para tentar entender o que cada instrução do IL representa. No próximo tutorial vou explicar um pouco melhor as instruções do IL e como fazer alterações no código.

Até a próxima!

F3rGO!

Tutorial - Reverse Engineering

Part 12



ENDEREÇOS DE MEMÓRIA

Creio que deveria ter feito este tutorial antes, pois considero de extrema importância na hora de decifrar algoritmos em assembly. Trata-se do uso dos endereços de memória. Antes de iniciar, vou fazer uma diferenciação entre os registradores e os endereços de memória. Os registradores são pequenas partes de memória do processador, para armazenamento temporário de dados. Nos computadores mais comuns (x86), eles armazenam valores de 32 bits (4 bytes) e são usados basicamente para calculos simples durante o tempo de execução do programa. Já a memória fica responsável por armazenar grande quantidade de dados, como textos (strings). Vamos ao assunto principal. Veja no código abaixo:

```
00401000  B9 05000000  MOV ECX,5
00401005  BE 04304000  MOV ESI,pont.00403004 ; ASCII "abcde"
0040100A  BF 0C304000  MOV EDI,pont.0040300C
0040100F  49          DEC ECX
00401010  8B1C31     MOV EBX,DWORD PTR DS:[ECX+ESI]
00401013  891F     MOV DWORD PTR DS:[EDI],EBX
00401015  47          INC EDI
00401016  0BC9     OR ECX,ECX
00401018  77 F5     JA SHORT pont.0040100F
```

O primeiro comando, move o valor 5 para o registrador ECX

Na segunda linha (MOV ESI,pont.00403004), o que está acontecendo? Ele está movendo para o registrador ESI o valor 00403004, que o Olly já identifica como sendo um endereço da memória do programa, por isso adicionou o "pont.00403004". O Olly também já identificou que esse endereço aponta para o primeiro byte da string "abcde", como mostra a tabela abaixo:

Endereço	Conteúdo (ASCII)
00403004	a (97)
00403005	b (98)
00403006	c (99)
00403007	d (100)
00403008	e (101)

Analisando o código, vemos que ESI então aponta para 403004

(início da string "abcde") e EDI aponta para 40300C, que é um espaço de memória vazio, não inicializado ainda. Em seguida ele decrementa o valor de ECX, que até então era 5 e passa a ser 4. Na próxima linha vem a parte principal deste tutorial.

```
00401010 8B1C31 MOV EBX,DWORD PTR DS:[ECX+ESI] ; sentenca 1
```

Esse "DWORD PTR" indica que EBX está recebendo o conteúdo do endereço que está sendo apontado (ECX+ESI). Releia essa frase quantas vezes forem necessárias. O valor que fica entre colchetes indica o endereço, que no caso é ECX+ESI. Alguns parágrafos acima vimos que ECX é 4 e ESI continha o valor 00403004, que era o primeiro byte da string "abcde". Somando os 2 valores temos: $00403004 + 4 = 00403008$. Agora as coisas estão ficando mais claras. EBX então está recebendo o conteúdo do endereço de memória 00403008. Se você olhar na tabela acima, verá que esse conteúdo corresponde a letra "e" (101). Em outras palavras, EBX = 101.

Avançando uma linha temos:

```
00401013 891F MOV DWORD PTR DS:[EDI],EBX ; sentenca 2
```

Creio que agora ficou mais fácil de entender o que está acontecendo. EDI aponta para um local de memória vazia (como vimos no início), cujo endereço é 0040300C. Esse endereço de memória recebe o valor de EBX (101). Resumindo, o local de memória 0040300C contém o número 101. Mas o que aconteceria se eu tivesse apenas "MOV EDI, EBX"? Neste caso você faria com que EDI recebesse o valor de EBX. No entanto, o que você quer fazer é mover o valor de EBX para o endereço que o EDI está apontando. Em seguida ele incrementa EDI (EDI++). EDI continha o valor 0040300C e agora passa a ser 0040300D. Na próxima linha ele realiza uma operação lógica OR com ECX (4) para verificar se ECX é nulo. Se não for, ele repete o processo, mas com valores alterados. Na segunda iteração (segunda repetição), ECX é novamente decrementado, passando de 4 para 3. A soma ECX+ESI, ao invés de ser $00403004 + 4$, vai ser $00403004 + 3$, que corresponde ao endereço do caractere "d" (100). Esse valor é armazenado no endereço apontado por EDI, que agora é 0040300D. Você já deve ter entendido o que o algoritmo faz. Caso não tenha percebido, ele simplesmente inverte uma string, colocando o resultado em um local de memória. Aí vai uma figura para ajudar (lembrando que o processo começa movendo o conteúdo de 403008 para 40300C).



Espero ter ajudado a esclarecer um pouco sobre essa questão de endereços de memória. Até a próxima!

F3rGO!

Tutorial - Engenharia Reversa

Desafio #1

DESAFIO #1

Este é o primeiro desafio que estou postando. Ele é mais como um exercício para treinar aquilo que foi ensinado nos tutoriais (do 1 ao 8). Envolve quase todos os assuntos discutidos nos tutos (descompressão de upx, nag-screen, serial fishing, anti-debug e alteração de controles). Caso seja iniciante, recomendo a leitura dos tutoriais antes de tentar resolver (principalmente dos tutoriais que falam sobre os itens que citei).

No geral, é um desafio fácil, coloquei nível 3/10 só pelo fato de envolver diversos assuntos. O alvo foi programado em VisualBasic 6, então dê uma olhada melhor nos nomes das funções, etc...

Como é o primeiro desafio, vou postar as tarefas a serem executadas (também se encontram no próprio alvo)

Tarefas:

- Descompactar UPX pelo processo manual (fazendo o dumping pelo Olly)
- Remover a simples proteção anti-debug (anti-olly na verdade)
- Ativar o botão para registrar o alvo
- Buscar pelo serial correto e/ou alterar o código e/ou entender o algoritmo (recomendo tentar os 3)
- Remover a nag-screen quando o alvo é encerrado

Info:

- Linguagem: VisualBASIC 6
- Packed: Sim
- Tamanho: 36.352 bytes (exe) - 17.511 bytes (zip)

Download: [desafio1.zip](#)

