Hakin9

Exploiting Software

Vol.2 No.10
Issue 10/2012(14) ISSN: 1733-7186

70+ pages

A Manual To REVERSE ENGINEERING

HOW TO ANALYZE APPLICATIONS WITH OLLY DEBUGGER?

HOW TO IDENTIFY AND BYPASS ANTI-REVERSING TECHNIQUES?

SOCAT AND WIRESHARK FOR PRACTICAL SSL PROTOCOL REVERSE ENGINEERING

DISASSEMBLE AND DEBUG EXECUTABLE PROGRAMS ON LINUX, WINDOWS AND MAC OS X

PLUS

JSCRAMBLER – PROTECT YOUR CODE (REVIEW)

MODERN WEBSITES, WHICH USE WEB 2.0 AND AJAX, OFTEN GENERATE HTML AND JAVASCRIPT CODE ON THE FLY

|w| http://www.sysmoth.com
|e| info@sysmoth.com
|p| +92 333 2319192

sysmoth

Cloud & Virtualization | Server Administration | Security & Compliance

Cloud & Virtualization

- Cloud & Virtualization Consultancy
- Building Virtualized Infrastructure
- Infrastructure on Public Cloud
- Building Private Cloud
- Cloud Management Setups
- Big Data Setups
- Infrastructure Management and Support

Server Administration

- Server Setups
- Control Panels Setups
- Server/Network Monitoring Setups
- Site Migration
- Server Optimization
- Email Setups
- Version Control Setups
- Server Automation
- Server Management & Support
- Load Balancing, FailOver and
- Geo Distribution Solutions
- Storage Solutions
- Special Purpose Appliance Building

Security & Compliance

- Server & Network Security Setups
- Security Testing, Audit and Compliance
- Incident Response
- Managed Security Service

Source HDD
Target HDD
Atola
Busy
Power
Power On
Power Off
Power

Exploiting Software team

Editor in Chief: Krzysztof Samborski
krzysztof.samborski@hakin9.org

Editorial Advisory Board: John Webb, Marco Hermans, Visvaprakash, Guillermo Lozano, Elia Pinto, JI PB

Proofreaders: Jeff Smith, Nick Baronian

Special thanks to our Beta testers and Proofreaders who helped us with this issue. Our magazine would not exist without your assistance and expertise.

Publisher: Paweł Marciniak

CEO: Ewa Dudzic
ewa.dudzic@hakin9.org

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

Art. Director: Ireneusz Pogroszewski
ireneusz.pogroszewski@hakin9.org

DTP: Ireneusz Pogroszewski

Marketing Director: Krzysztof Samborski
krzysztof.samborski@hakin9.org

Publisher: Software Press sp. z o.o. SK
02-682 Warszawa, ul. Bokserska 1
Phone: 1 917 338 3631
www.hakin9.org/en

Dear Readers,
Reverse Engineering is a process of the exploration of a product (computer program, device) which is conducted to find out how this product really works and how it was made. The process is usually applied in order to create an equivalent of the already existing product or to ensure interoperability with other products.

The issue you are reading touches upon the topic of Reverse Engineering. We decided to supply you with this publication in response to your request for the subject to be covered in this month's issue of Exploiting Software.

We grouped the articles published in the issue into thematic sections. These are called: Tools (the articles by Jaromir Horejsi, Jacek Adam Piasecki and Shane R. Spencer), Reaching The Code (Adam Kujawa and Eoin Ward's publications) and Reverse It Yourself (in the papers of Lorenzo Xie and Raheel Ahmad). The latter closes the issue with his review of JScrambler product in the Hakin9 Extra section.

Seizing the opportunity of publishing this December's issue of Exploiting Software, we wanted to wish a Merry Christmas and a Happy New Year to all our readers and followers. May this special time of the year be peaceful and cheerful for You and Your Families.

MERRY CHRISTMAS!

Regards,

Krzysztof Samborski
and Hakin9 Team

CONTENTS

THE TOOLS

How to Analyze Applications With Olly Debugger? **06**

BY JAROMIR HOREJSI

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.

How to Disassemble and Debug Executable Programs on Linux, Windows and Mac OS X? **18**

BY JACEK ADAM PIASECKI

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.

How to use Socat and Wireshark for Practical SSL Protocol Reverse Engineering? **30**

BY SHANE R. SPENCER

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.

REACHING THE CODE

How to Defeat Code Obfuscation While Reverse Engineering? **36**

BY ADAM KUJAWA

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.

How to Identify and Bypass Anti-reversing Techniques? **46**

BY EOIN WARD

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.

REVERSE IT YOURSELF

How to Reverse Engineer? **54**

BY LORENZO XIE

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.

How to Reverse the Code? **60**

BY RAHEEL AHMAD

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).

HAKIN9 EXTRA

JScrambler Protect Your Code (Review) **66**

BY RAHEEL AHMAD

Modern websites, which use Web 2.0 and AJAX, often generate HTML and JavaScript code on the fly. This means that standard static code analysers cannot fully scan the source code and locate client-side JavaScript issues, since the source code itself does not yet include the entire HTML and JavaScript code.

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.

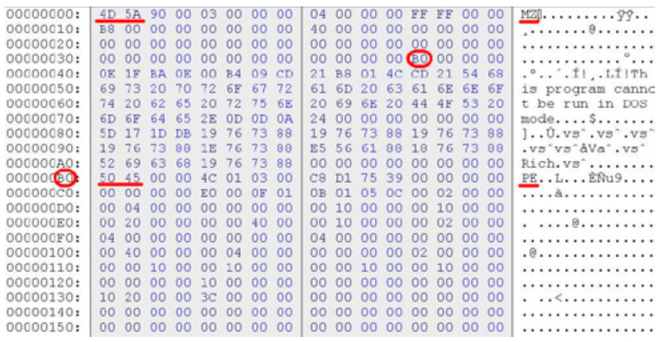
After reading the article, you should be able to open a program in Olly debugger and start analyzing it. If necessary, you should be able to make your own patch or reverse simple algorithms.

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.vs^.vs^.vs^
00000090: 19 76 73 88 1E 76 73 88  E5 56 61 88 18 76 73 88  .vs^vs^åVa^.vs^
000000A0: 52 69 63 68 19 76 73 88  00 00 00 00 00 00 00 00  Rich.vs^........
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

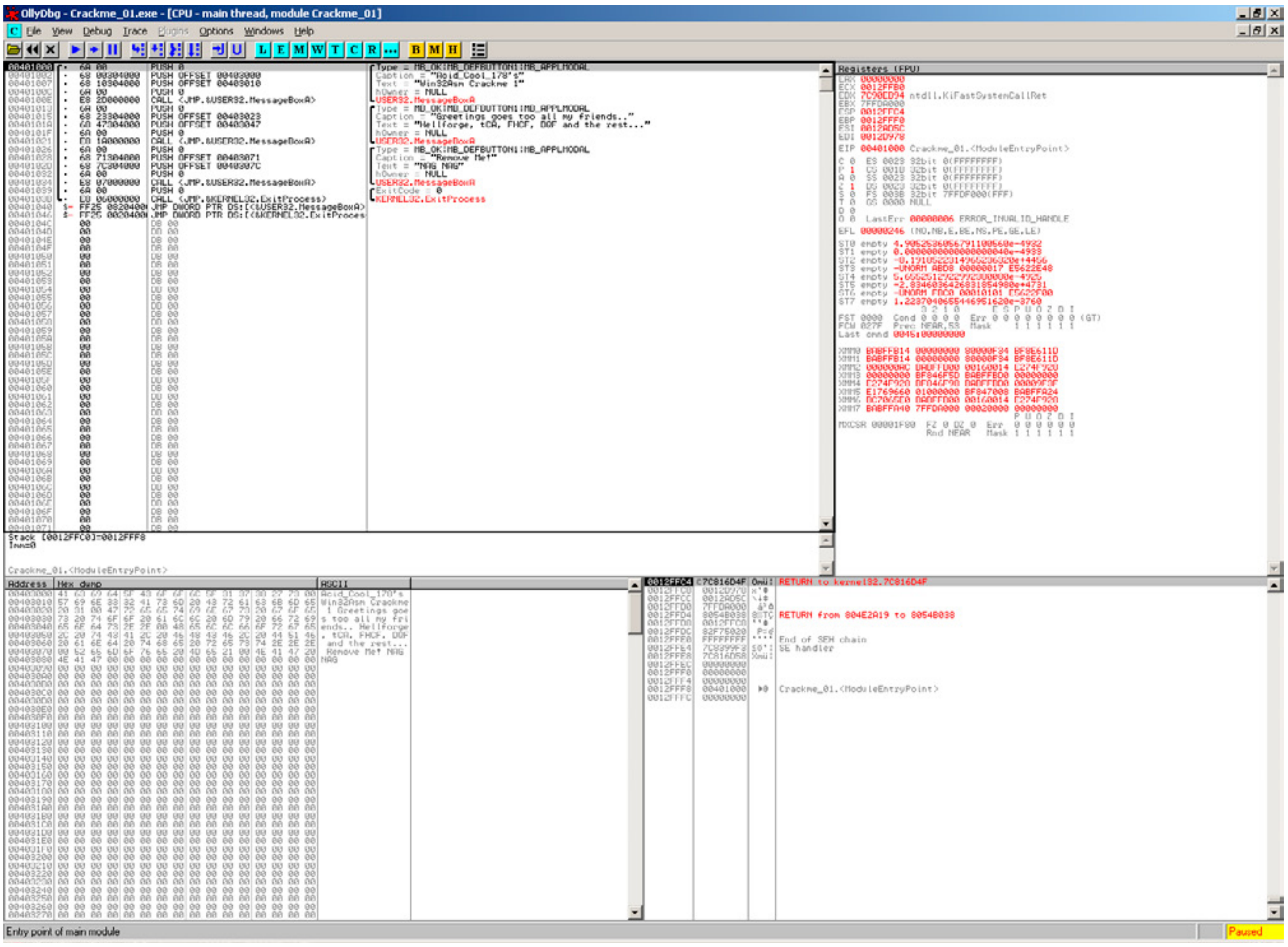
When you execute ollydbg.exe and drag and drop any executable file on it (in my case I used crack-me_01.exe), you will notice four sub-windows – disassembly (upper left), registers (upper right), dump (bottom left) and stack (bottom right) (see Figure 2). We will say a little bit about each of these 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]
C File View Debug Trace Plugins Options Windows Help
L E M W T C R ... B M H

```
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, DOF 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 08204000 JMP DWORD PTR DS:[<&USER32.MessageBoxA>]
00401046   $- FF25 00204000 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
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
...
0040106B      00            DB 00
0040106C      00            DB 00
0040106D      00            DB 00
0040106E      00            DB 00
0040106F      00            DB 00
00401070      00            DB 00
00401071      00            DB 00
Stack [0012FFC0]=0012FFF8
Imm=0

Crackme_01.<ModuleEntryPoint>
```

```
Registers (FPU)
EAX 00000000
ECX 0012FFB0
EDX 7C90ED94 ntdll.KiFastSystemCallRet
EBX 7FFDA000
ESP 0012FFC4
EBP 0012FFF0
ESI 0012AD5C
EDI 0012D978
EIP 00401000 Crackme_01.<ModuleEntryPoint>
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 00000006 ERROR_INVALID_HANDLE
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty 4.9052536056791100560e-4932
ST1 empty 0.0000000000000000040e-4933
ST2 empty -0.1910522014966206020e+4456
ST3 empty -UNORM ABD8 00000017 E5622E48
ST4 empty 5.6652512722972000000e-4925
ST5 empty -2.8346036426831854980e+4731
ST6 empty -UNORM FDC0 00010101 E5622F00
ST7 empty 1.2237040655446951620e-3760
               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
Last cmnd 0045:00000000

XMM0 BABFFB14 00000000 80000F34 BF8E611D
XMM1 BABFFB14 00000000 80000F34 BF8E611D
XMM2 000000AC DA0FFD00 00160014 E274F920
XMM3 00000000 BF846F5D BABFFBD0 00000000
XMM4 E274F920 BF046F90 BABFFBD0 00000F0F
XMM5 E1769660 01000000 BF847008 BABFFA24
XMM6 BC706CE0 BABFFD00 00160014 E274F920
XMM7 BABFFA40 7FFDA000 00020000 00000000
                                P U O Z D I
MXCSR 00001F80  FZ 0 DZ 0  Err  0 0 0 0 0 0
                Rnd NEAR   Mask 1 1 1 1 1 1
```

```
Address  Hex dump                                          ASCII
00403000 41 63 69 64 5F 43 6F 6F 6C 5F 31 37 38 27 73 00  Acid_Cool_178's
00403010 57 69 6E 33 32 41 73 6D 20 43 72 61 63 6B 6D 65  Win32Asm Crackme
00403020 20 31 00 47 72 65 65 74 69 6E 67 73 20 67 6F 65   1 Greetings goe
00403030 73 20 74 6F 6F 20 61 6C 6C 20 6D 79 20 66 72 69  s too all my fri
00403040 65 6E 64 73 2E 2E 00 48 65 6C 6C 66 6F 72 67 65  ends.. Hellforge
00403050 2C 20 74 43 41 2C 20 46 48 43 46 2C 20 44 4F 46  , tCA, FHCF, DOF
00403060 20 61 6E 64 20 74 68 65 20 72 65 73 74 2E 2E 2E   and the rest...
00403070 00 52 65 6D 6F 76 65 20 4D 65 21 00 4E 41 47 20   Remove Me! NAG
00403080 4E 41 47 00 00 00 00 00 00 00 00 00 00 00 00 00  NAG
00403090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004030A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004030B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004030C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004030D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004030E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004030F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
...
```

```
0012FFC4  7C816D4F  Omü|  RETURN to kernel32.7C816D4F
0012FFC8  0012D770  x'↕
0012FFCC  0012AD5C  \↓↕
0012FFD0  7FFDA000   á⌂
0012FFD4  8054B038  8░TÇ  RETURN from 804E2A19 to 8054B038
0012FFD8  0012FFC8  ╚ ↕
0012FFDC  82F75020   P≈é
0012FFE0  FFFFFFFF  ....  End of SEH chain
0012FFE4  7C8399F3  ≤Ö'|  SE handler
0012FFE8  7C816D58  Xmü|
0012FFEC  00000000
0012FFF0  00000000
0012FFF4  00000000
0012FFF8  00401000   ►@   Crackme_01.<ModuleEntryPoint>
0012FFFC  00000000
```

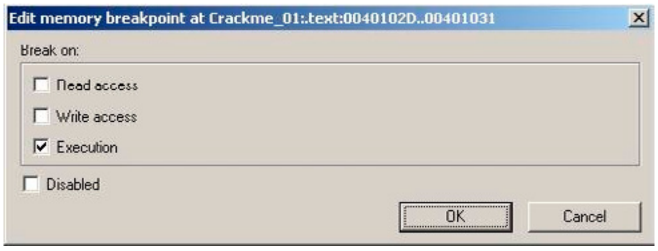
Entry point of main module   Paused

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).

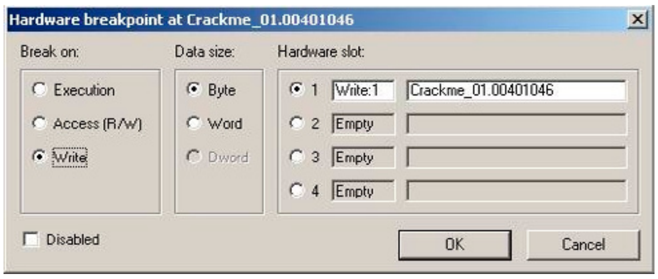
That’s all for the description of the four basic sub-windows. However, if you need to display more information, you can click on View menu and select any of those options to display optional sub-windows – see Figure 3.

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.



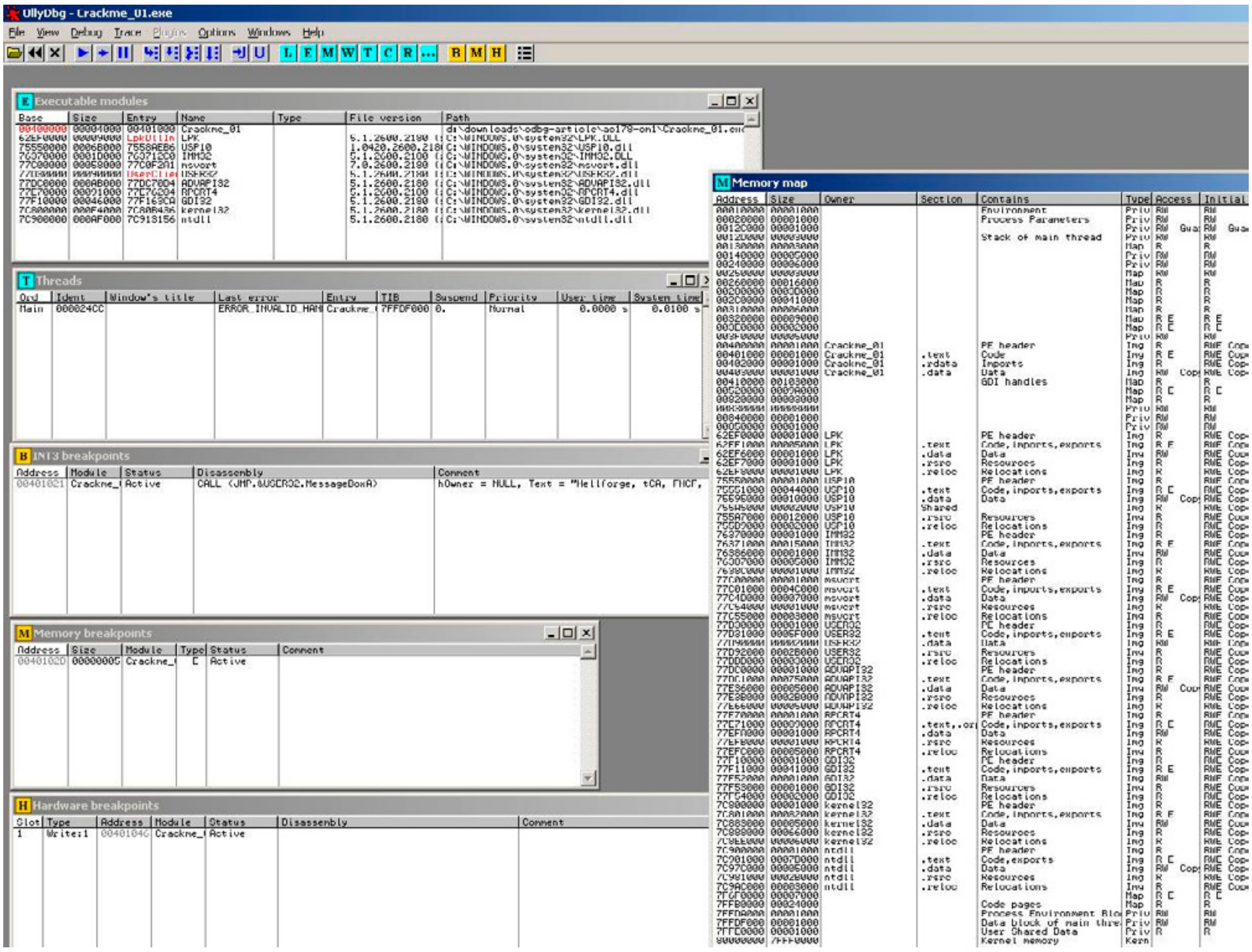
```
Edit memory breakpoint at Crackme_01:.text:0040102D..00401031
Break on:
[ ] Read access
[ ] Write access
[✓] Execution
[ ] Disabled
OK   Cancel
```

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*



```
OllyDbg - Crackme_01.exe

Executable modules
Base     Size     Entry    Name       Type  File version      Path
00400000 00004000 00401000 Crackme_01                         d:\downloads\odbg-article\ao178-onl\Crackme_01.exe
62EF0000 00009000 LpkDllIn LPK              5.1.2600.2180 (   C:\WINDOWS.0\system32\LPK.DLL
75550000 0006B000 7558AEB6 USP10            1.0420.2600.218   C:\WINDOWS.0\system32\USP10.dll
76370000 0001D000 763712C0 IMM32            5.1.2600.2180 (   C:\WINDOWS.0\system32\IMM32.DLL
77C00000 00058000 77C0F2A1 msvcrt           7.0.2600.2180 (   C:\WINDOWS.0\system32\msvcrt.dll
77D30000 00090000 UserClie USER32           5.1.2600.2180 (   C:\WINDOWS.0\system32\USER32.dll
77DC0000 000AB000 77DC70D4 ADVAPI32         5.1.2600.2180 (   C:\WINDOWS.0\system32\ADVAPI32.dll
77E70000 00091000 77E76204 RPCRT4           5.1.2600.2180 (   C:\WINDOWS.0\system32\RPCRT4.dll
77F10000 00046000 77F163CA GDI32            5.1.2600.2180 (   C:\WINDOWS.0\system32\GDI32.dll
7C800000 000F4000 7C80B436 kernel32         5.1.2600.2180 (   C:\WINDOWS.0\system32\kernel32.dll
7C900000 000AF000 7C913156 ntdll            5.1.2600.2180 (   C:\WINDOWS.0\system32\ntdll.dll

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
00401021 Crackme_  Active CALL <JMP.&USER32.MessageBoxA>      hOwner = NULL, Text = "Hellforge, tCA, FHCF,

Memory breakpoints
Address  Size     Module   Type Status Comment
0040102D 00000005 Crackme_  E   Active

Hardware breakpoints
Slot Type    Address  Module   Status Disassembly   Comment
1    Write:1 00401046 Crackme_ Active

Memory map
Address  Size     Owner      Section   Contains                  Type Access   Initial
00010000 00001000                      Environment               Priv RW       RW
00020000 00001000                      Process Parameters        Priv RW       RW
0012C000 00001000                                                Priv RW  Gua: RW  Gua:
0012D000 00003000                      Stack of main thread      Priv RW       RW
00130000 00003000                                                Map  R        R
00140000 00005000                                                Priv RW       RW
00240000 00006000                                                Priv RW       RW
00250000 00003000                                                Map  RW       RW
00260000 00016000                                                Map  R        R
00280000 0003D000                                                Map  R        R
002C0000 00041000                                                Map  R        R
00310000 00006000                                                Map  R        R
00320000 00009000                                                Map  R E      R E
003D0000 00002000                                                Map  R E      R E
003F0000 00005000                                                Priv RW       RW
00400000 00001000 Crackme_01           PE header                 Img  R        RWE Cop
00401000 00001000 Crackme_01 .text     Code                      Img  R E      RWE Cop
00402000 00001000 Crackme_01 .rdata    Imports                   Img  R        RWE Cop
00403000 00001000 Crackme_01 .data     Data                      Img  RW  Cop  RWE Cop
00410000 00103000                      GDI handles               Map  R        R
00520000 0007A000                                                Map  R E      R E
00820000 00003000                                                Map  R        R
00830000 00001000                                                Priv RW       RW
00840000 00001000                                                Priv RW       RW
00850000 00001000                                                Priv RW       RW
62EF0000 00001000 LPK                  PE header                 Img  R        RWE Cop
62EF1000 00005000 LPK        .text     Code,imports,exports      Img  R E      RWE Cop
62EF6000 00001000 LPK        .data     Data                      Img  RW       RWE Cop
62EF7000 00001000 LPK        .rsrc     Resources                 Img  R        RWE Cop
62EF8000 00001000 LPK        .reloc    Relocations               Img  R        RWE Cop
75550000 00001000 USP10                PE header                 Img  R        RWE Cop
75551000 00044000 USP10      .text     Code,imports,exports      Img  R E      RWE Cop
75595000 00010000 USP10      .data     Data                      Img  RW  Cop  RWE Cop
755A5000 00002000 USP10      Shared                              Img  R        RWE Cop
755A7000 00012000 USP10      .rsrc     Resources                 Img  R        RWE Cop
755D9000 00002000 USP10      .reloc    Relocations               Img  R        RWE Cop
76370000 00001000 IMM32                PE header                 Img  R        RWE Cop
76371000 00015000 IMM32      .text     Code,imports,exports      Img  R E      RWE Cop
76386000 00001000 IMM32      .data     Data                      Img  RW       RWE Cop
76387000 00005000 IMM32      .rsrc     Resources                 Img  R        RWE Cop
7638C000 00001000 IMM32      .reloc    Relocations               Img  R        RWE Cop
77C00000 00001000 msvcrt               PE header                 Img  R        RWE Cop
77C01000 0004C000 msvcrt     .text     Code,imports,exports      Img  R E      RWE Cop
77C4D000 00007000 msvcrt     .data     Data                      Img  RW  Cop  RWE Cop
77C54000 00001000 msvcrt     .rsrc     Resources                 Img  R        RWE Cop
77C55000 00003000 msvcrt     .reloc    Relocations               Img  R        RWE Cop
77D30000 00001000 USER32               PE header                 Img  R        RWE Cop
77D31000 0005F000 USER32     .text     Code,imports,exports      Img  R E      RWE Cop
77D90000 00002000 USER32     .data     Data                      Img  RW       RWE Cop
77D92000 0002B000 USER32     .rsrc     Resources                 Img  R        RWE Cop
77DBD000 00003000 USER32     .reloc    Relocations               Img  R        RWE Cop
77DC0000 00001000 ADVAPI32             PE header                 Img  R        RWE Cop
77DC1000 00075000 ADVAPI32   .text     Code,imports,exports      Img  R E      RWE Cop
77E36000 00005000 ADVAPI32   .data     Data                      Img  RW  Cop  RWE Cop
77E3B000 0002B000 ADVAPI32   .rsrc     Resources                 Img  R        RWE Cop
77E66000 00005000 ADVAPI32   .reloc    Relocations               Img  R        RWE Cop
77E70000 00001000 RPCRT4               PE header                 Img  R        RWE Cop
77E71000 0007D000 RPCRT4     .text,.or Code,imports,exports      Img  R E      RWE Cop
77EFA000 00001000 RPCRT4     .data     Data                      Img  RW       RWE Cop
77EFB000 00001000 RPCRT4     .rsrc     Resources                 Img  R        RWE Cop
77EFC000 00005000 RPCRT4     .reloc    Relocations               Img  R        RWE Cop
77F10000 00001000 GDI32                PE header                 Img  R        RWE Cop
77F11000 00041000 GDI32      .text     Code,imports,exports      Img  R E      RWE Cop
77F52000 00001000 GDI32      .data     Data                      Img  RW       RWE Cop
77F53000 00001000 GDI32      .rsrc     Resources                 Img  R        RWE Cop
77F54000 00002000 GDI32      .reloc    Relocations               Img  R        RWE Cop
7C800000 00001000 kernel32             PE header                 Img  R        RWE Cop
7C801000 00082000 kernel32   .text     Code,imports,exports      Img  R E      RWE Cop
7C883000 00005000 kernel32   .data     Data                      Img  RW       RWE Cop
7C888000 00066000 kernel32   .rsrc     Resources                 Img  R        RWE Cop
7C8EE000 00006000 kernel32   .reloc    Relocations               Img  R        RWE Cop
7C900000 00001000 ntdll                PE header                 Img  R        RWE Cop
7C901000 0007D000 ntdll      .text     Code,exports              Img  R E      RWE Cop
7C97E000 00005000 ntdll      .data     Data                      Img  RW  Cop  RWE Cop
7C983000 0002B000 ntdll      .rsrc     Resources                 Img  R        RWE Cop
7C9AE000 00003000 ntdll      .reloc    Relocations               Img  R        RWE Cop
7F6F0000 00007000                                                Map  R E      R E
7FFB0000 00024000                      Code pages                Map  R        R
7FFDA000 00001000                      Process Environment Blo   Priv RW       RW
7FFDF000 00001000                      Data block of main thre   Priv RW       RW
7FFE0000 00001000                      User Shared Data          Priv R        R
80000000 7FFF0000                      Kernel memory             Kern
```

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-

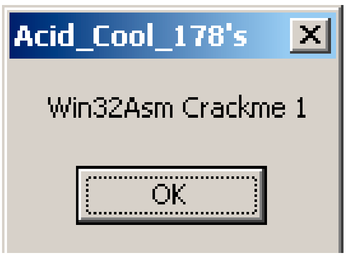

Acid_Cool_178's
Win32Asm Crackme 1

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

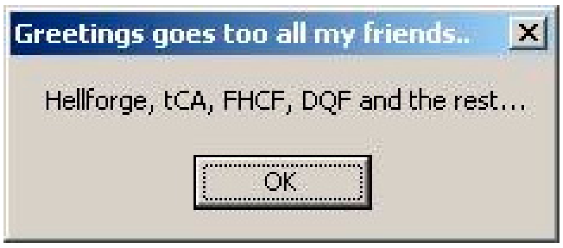

Greetings goes too all my friends..
Hellforge, tCA, FHCF, DQF and the rest...

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

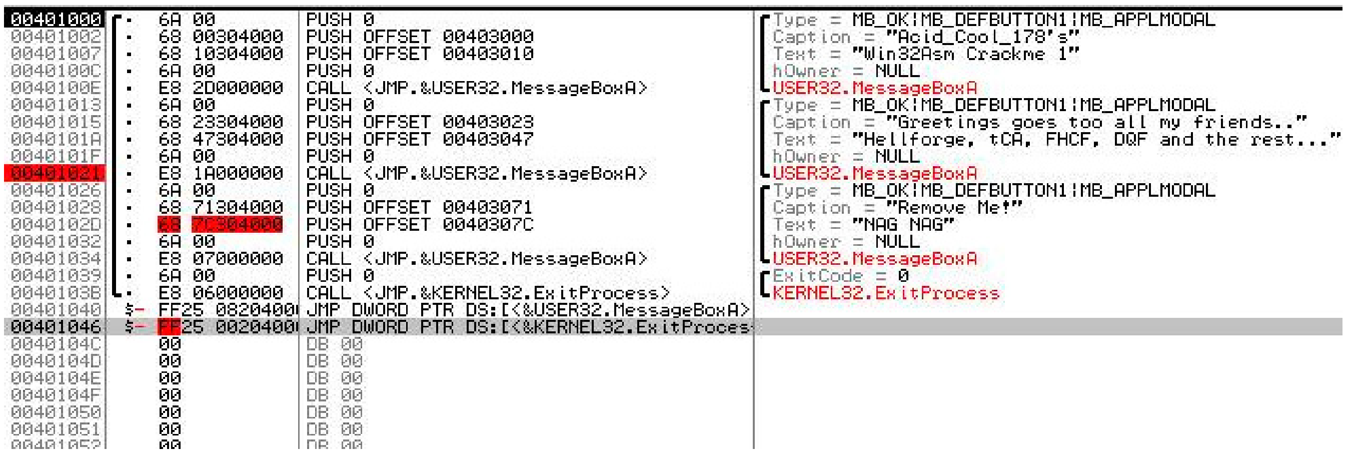
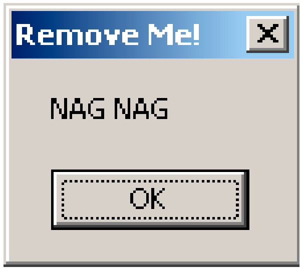

```
00401000  ┌.  6A 00         PUSH 0                                    ┌Type = MB_OK|MB_DEFBUTTON1|MB_APPLMODAL
00401002  │.  68 00304000   PUSH OFFSET 00403000                      │Caption = "Acid_Cool_178's"
00401007  │.  68 10304000   PUSH OFFSET 00403010                      │Text = "Win32Asm 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).

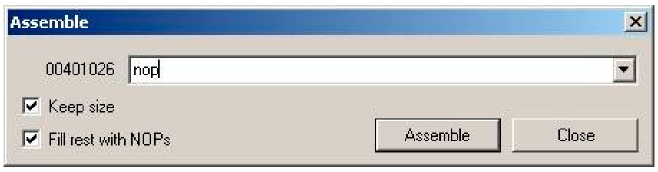
After pressing OK, we stop at 401026. If we press F9 (Run) again, we stop at 40102D, because we set Memory Breakpoint on Execute at this address. We can continue either by pressing


Remove Me!

NAG NAG

OK

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


Assemble

00401026 nop

[✓] Keep size

[✓] Fill rest with NOPs

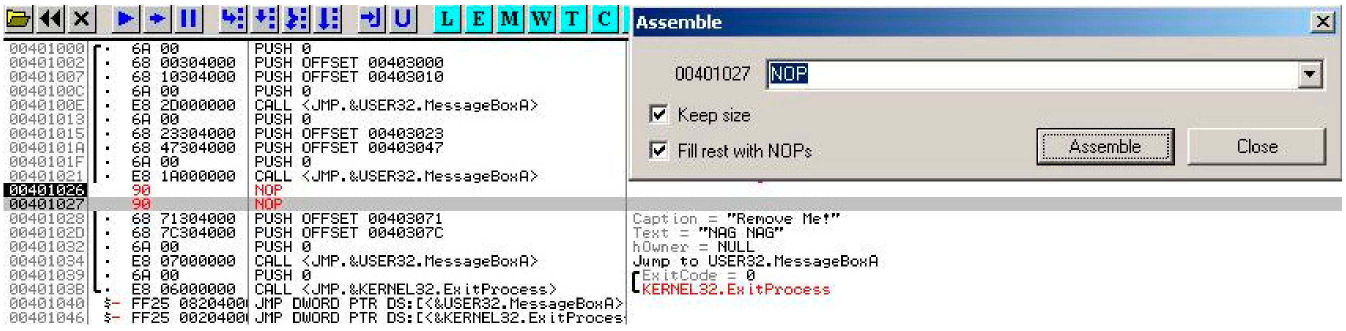
Assemble Close

Figure 10. *Dialog for replacing instructions*

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


```
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*

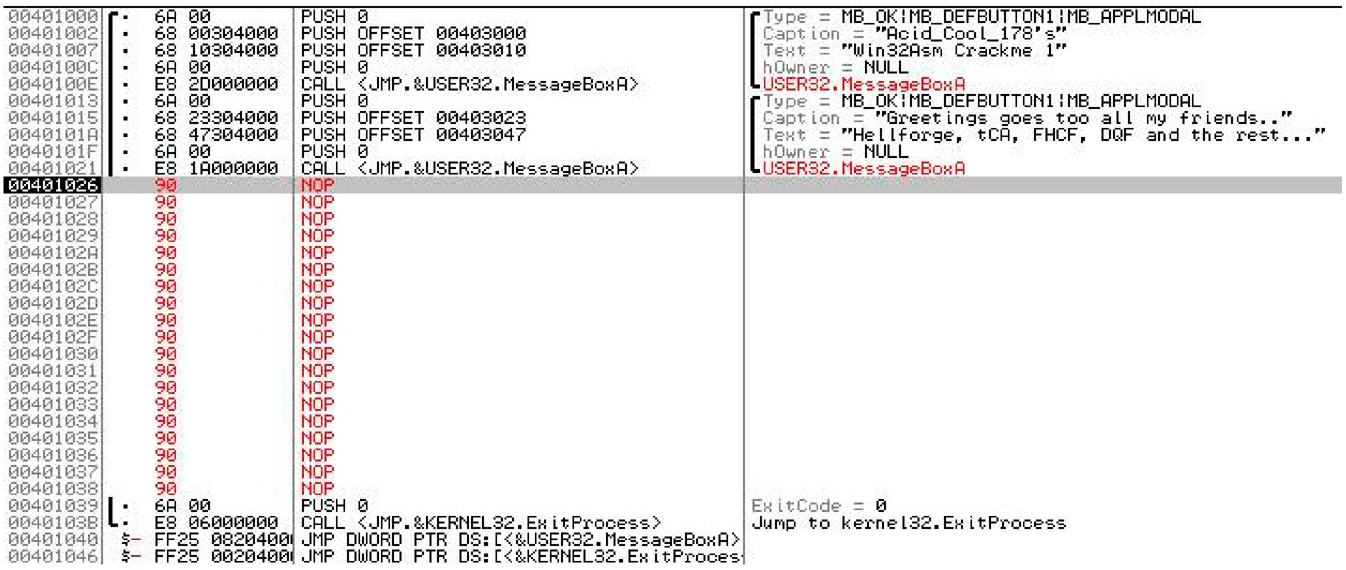
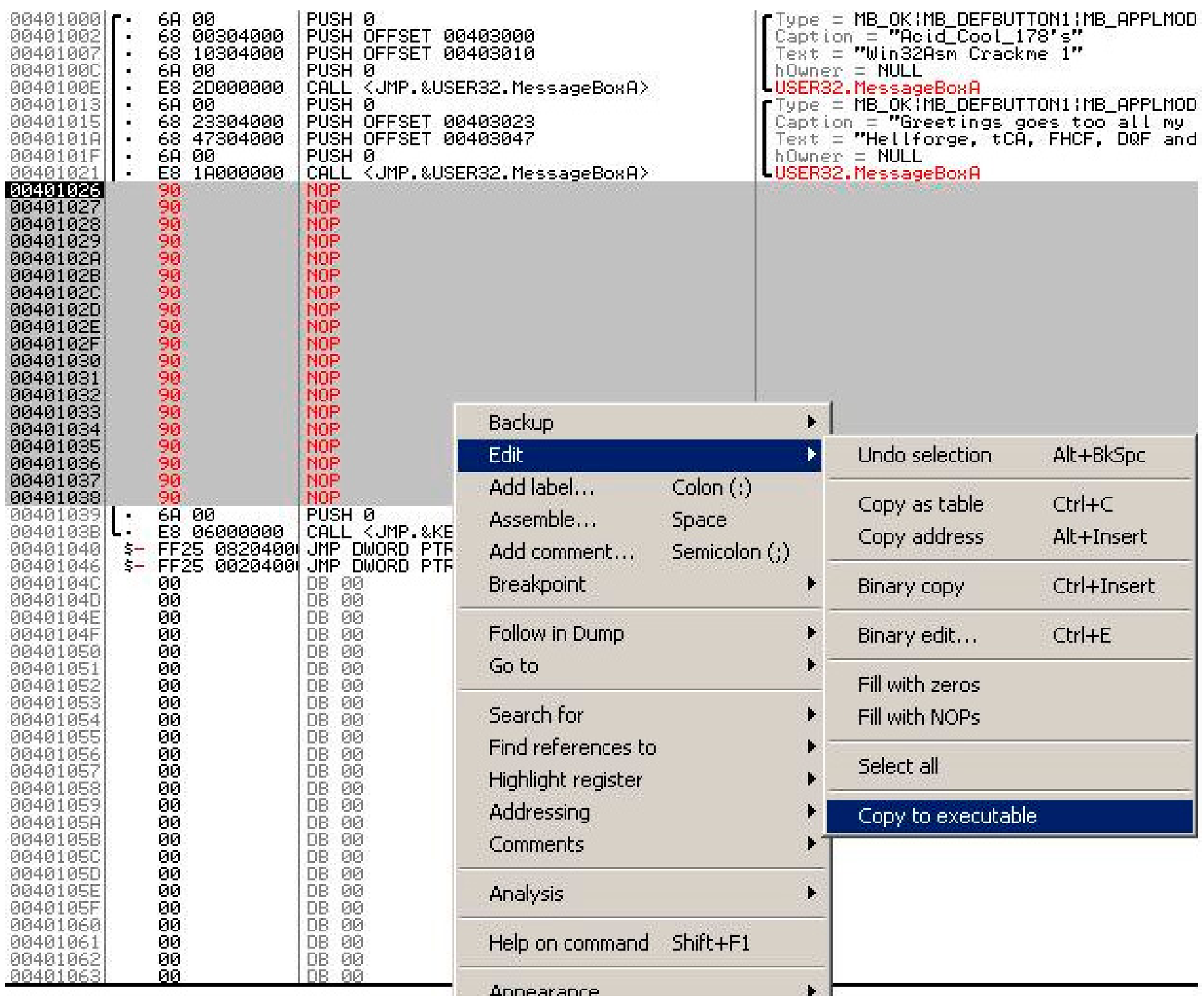
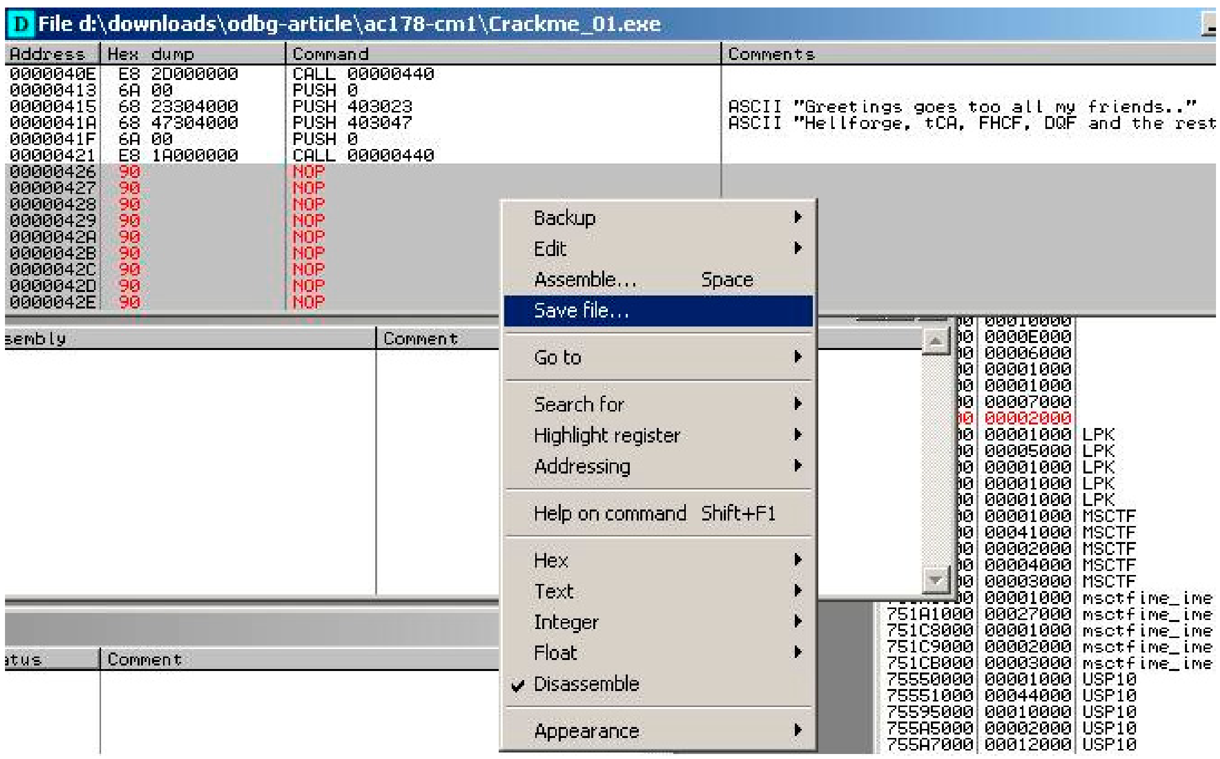

```
00401000  6A 00          PUSH 0                                 Type = MB_OK|MB_DEFBUTTON1|MB_APPLMODAL
00401002  68 00304000    PUSH OFFSET 00403000                   Caption = "Acid_Cool_178's"
00401007  68 10304000    PUSH OFFSET 00403010                   Text = "Win32Asm Crackme 1"
0040100C  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 0820400   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*



D 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 rest
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*

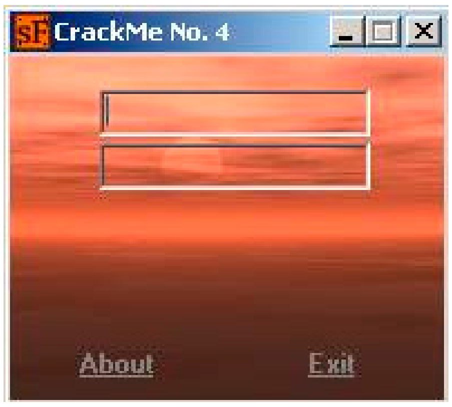


Figure 15. *The second crackme*

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).

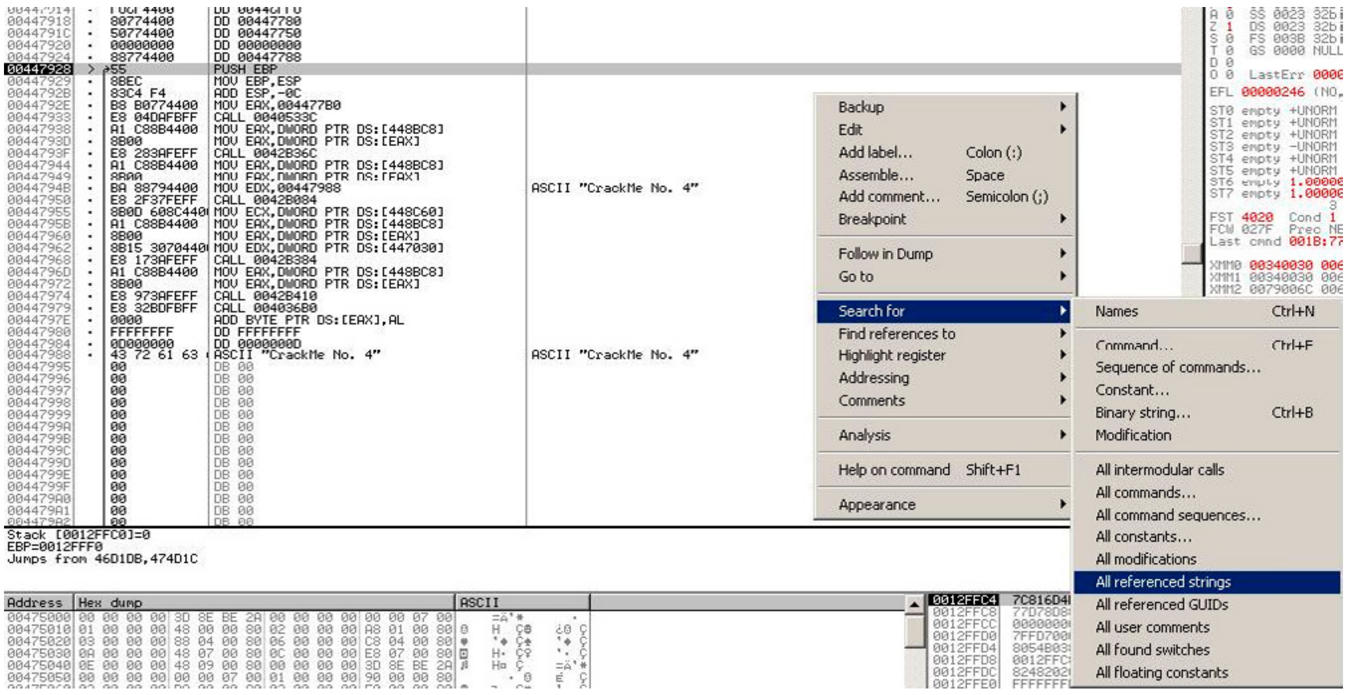
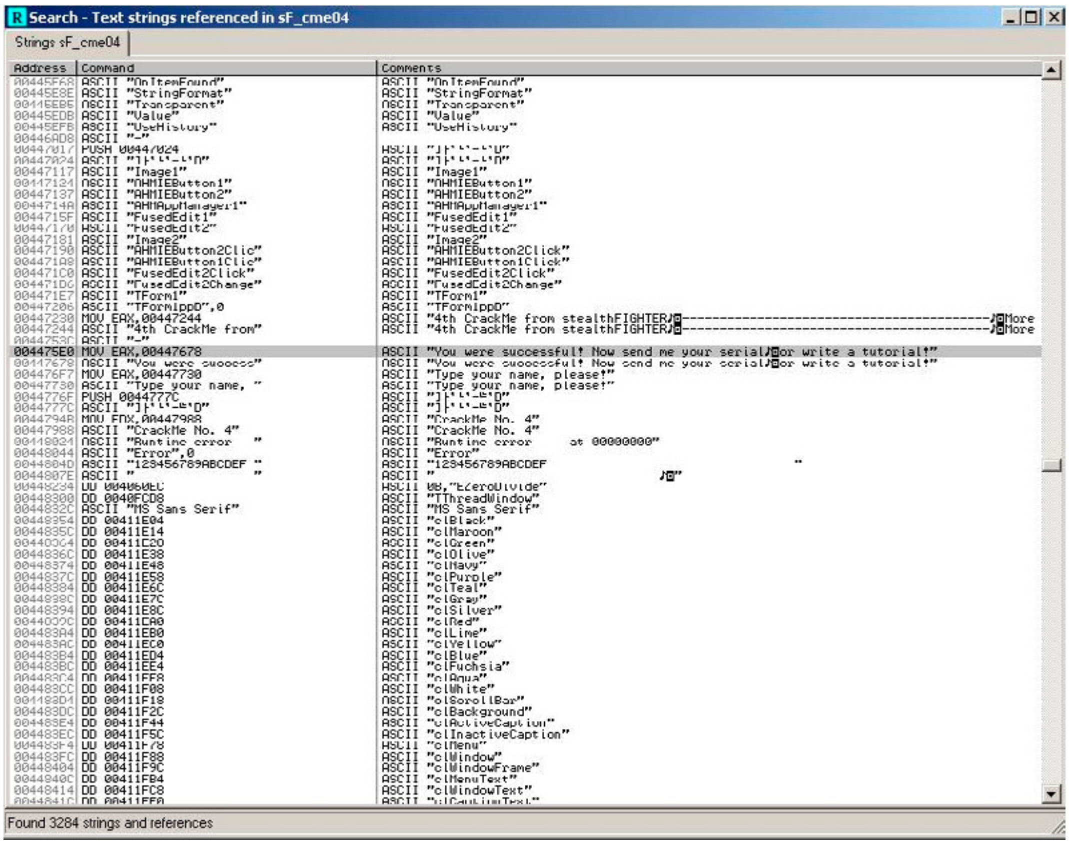

```
00447918  .  80774400      DD 00447780
0044791C  .  50774400      DD 00447750
00447920  .  00000000      DD 00000000
00447924  .  88774400      DD 00447788
00447928  > ↱55            PUSH EBP
00447929  .  8BEC          MOV EBP,ESP
0044792B  .  83C4 F4       ADD ESP,-0C
0044792E  .  B8 B0774400   MOV EAX,004477B0
00447933  .  E8 04DAFBFF   CALL 0040533C
00447938  .  A1 C88B4400   MOV EAX,DWORD PTR DS:[448BC8]
0044793D  .  8B00          MOV EAX,DWORD PTR DS:[EAX]
0044793F  .  E8 283AFEFF   CALL 0042B36C
00447944  .  A1 C88B4400   MOV EAX,DWORD PTR DS:[448BC8]
00447949  .  8B00          MOV EAX,DWORD PTR DS:[EAX]
0044794B  .  BA 88794400   MOV EDX,00447988                 ASCII "CrackMe No. 4"
00447950  .  E8 2F37FEFF   CALL 0042B084
00447955  .  8B0D 608C440  MOV ECX,DWORD PTR DS:[448C60]
0044795B  .  A1 C88B4400   MOV EAX,DWORD PTR DS:[448BC8]
00447960  .  8B00          MOV EAX,DWORD PTR DS:[EAX]
00447962  .  8B15 3070440  MOV EDX,DWORD PTR DS:[447030]
00447968  .  E8 173AFEFF   CALL 0042B384
0044796D  .  A1 C88B4400   MOV EAX,DWORD PTR DS:[448BC8]
00447972  .  8B00          MOV EAX,DWORD PTR DS:[EAX]
00447974  .  E8 973AFEFF   CALL 0042B410
00447979  .  E8 32BDFBFF   CALL 004036B0
0044797E  .  0000          ADD BYTE PTR DS:[EAX],AL
00447980  .  FFFFFFFF      DD FFFFFFFF
00447984  .  0D000000      DD 0000000D
00447988  .  43 72 61 63   ASCII "CrackMe No. 4"                ASCII "CrackMe No. 4"
00447995     00            DB 00
00447996     00            DB 00
00447997     00            DB 00
00447998     00            DB 00
00447999     00            DB 00
0044799A     00            DB 00
0044799B     00            DB 00
0044799C     00            DB 00
0044799D     00            DB 00
0044799E     00            DB 00
0044799F     00            DB 00
004479A0     00            DB 00
004479A1     00            DB 00
004479A2     00            DB 00
Stack [0012FFC0]=0
EBP=0012FFF0
Jumps from 46D1DB,474D1C

Address   Hex dump                                          ASCII
00475000  00 00 00 00 3D 8E BE 2A 00 00 00 00 00 00 07 00      =Ä'*    •
00475010  01 00 00 00 48 00 00 80 02 00 00 00 A8 01 00 80  ☺   H  Ç☻   ¿☺ Ç
00475020  03 00 00 00 88 04 00 80 06 00 00 00 C8 04 00 80  ♥   ê♦ Ç♠   È♦ Ç
00475030  0A 00 00 00 48 07 00 80 0C 00 00 00 E8 07 00 80  ◙   H• Ç♀   è• Ç
00475040  0E 00 00 00 48 09 00 80 00 00 00 00 3D 8E BE 2A  ♫   H○ Ç    =Ä'*
00475050  00 00 00 00 00 00 07 00 01 00 00 00 90 00 00 80        • ☺   É  Ç

0012FFC4  7C816D4
0012FFC8  77D78D8
0012FFCC  0000000
0012FFD0  7FFD700
0012FFD4  8054B03
0012FFD8  0012FFC
0012FFDC  8248202
0012FFE0  FFFFFFF

A 0  SS 0023 32bi
Z 1  DS 0023 32bi
S 0  FS 003B 32bi
T 0  GS 0000 NULL
D 0
O 0  LastErr 0000
EFL 00000246 (NO,
ST0 empty +UNORM
ST1 empty +UNORM
ST2 empty +UNORM
ST3 empty -UNORM
ST4 empty +UNORM
ST5 empty +UNORM
ST6 empty 1.0000
ST7 empty 1.0000
               3
FST 4020  Cond 1
FCW 027F  Prec NE
Last cmnd 001B:77
XMM0 00340030 006
XMM1 00340030 006
XMM2 0079006C 006
```

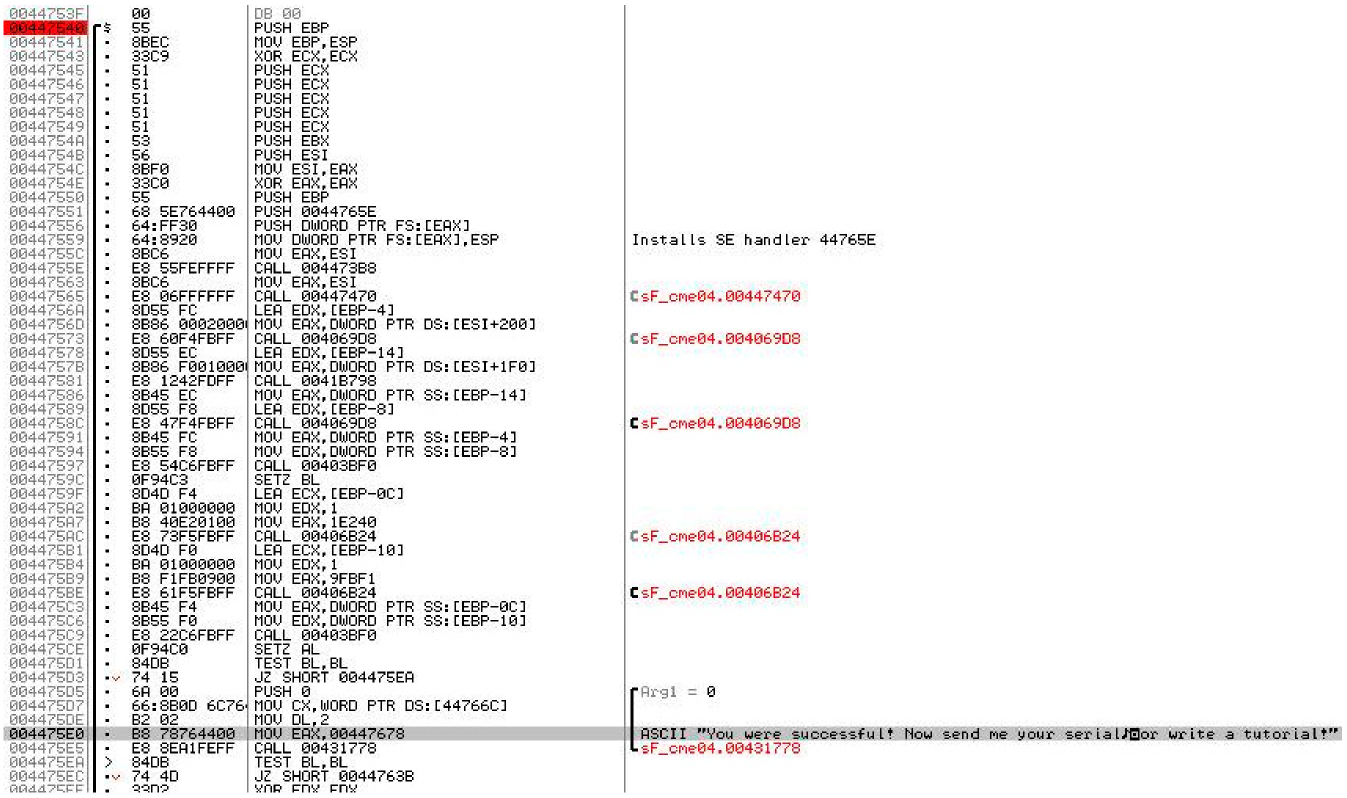
Figure 16. *Displaying all referenced text strings*



```
Address  Command                          Comments
00445F68 ASCII "OnItemFound"              ASCII "OnItemFound"
00445E8E ASCII "StringFormat"             ASCII "StringFormat"
00445EB6 ASCII "Transparent"              ASCII "Transparent"
00445EDB ASCII "Value"                    ASCII "Value"
00445EFB ASCII "UseHistory"               ASCII "UseHistory"
00446AD8 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"
004471A8 ASCII "AHMIEButton1Clic"         ASCII "AHMIEButton1Click"
004471C0 ASCII "FusedEdit2Click"          ASCII "FusedEdit2Click"
004471D6 ASCII "FusedEdit2Change"         ASCII "FusedEdit2Change"
004471E7 ASCII "TForm1"                   ASCII "TForm1"
00447206 ASCII "TForm1ppD",0              ASCII "TForm1ppD"
00447230 MOV EAX,00447244                 ASCII "4th CrackMe from stealthFIGHTER♪◘-----------------------------------------♪◘More
00447244 ASCII "4th CrackMe from"         ASCII "4th CrackMe from stealthFIGHTER♪◘-----------------------------------------♪◘More
0044753C ASCII "-"
004475E0 MOV EAX,00447678                 ASCII "You were successful! Now send me your serial♪◘or write a tutorial!"
00447678 ASCII "You were successf"        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"
0044794B 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"
0044804D ASCII "123456789ABCDEF "         ASCII "123456789ABCDEF                    "
0044807E ASCII "                "         ASCII "                        ♪◘"
00448234 DD 004060EC                      ASCII 0B,"EZeroDivide"
00448300 DD 0040FCD8                      ASCII "TThreadWindow"
0044832C ASCII "MS Sans Serif"            ASCII "MS Sans Serif"
00448354 DD 00411E04                      ASCII "clBlack"
0044835C DD 00411E14                      ASCII "clMaroon"
00448364 DD 00411E20                      ASCII "clGreen"
0044836C DD 00411E38                      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 00411F18                      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*



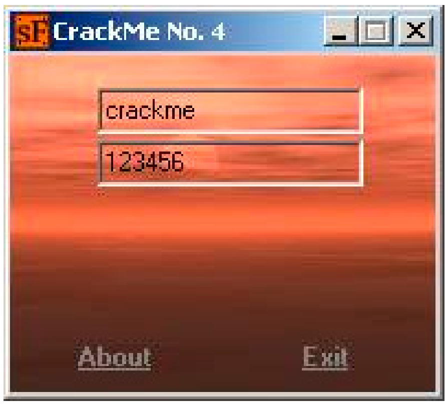
```
0044753F    00              DB 00
00447540  ┌$ 55              PUSH EBP
00447541  │. 8BEC            MOV EBP,ESP
00447543  │. 33C9            XOR ECX,ECX
00447545  │. 51              PUSH ECX
00447546  │. 51              PUSH ECX
00447547  │. 51              PUSH ECX
00447548  │. 51              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                         CsF_cme04.00447470
0044756A  │. 8D55 FC         LEA EDX,[EBP-4]
0044756D  │. 8B86 0002000    MOV EAX,DWORD PTR DS:[ESI+200]
00447573  │. E8 60F4FBFF     CALL 004069D8                         CsF_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                         CsF_cme04.004069D8
00447591  │. 8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]
00447594  │. 8B55 F8         MOV EDX,DWORD PTR SS:[EBP-8]
00447597  │. E8 54C6FBFF     CALL 00403BF0
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                         CsF_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                         CsF_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  │. 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 78764400     MOV EAX,00447678                      │ASCII "You were successful! Now send me your serial♪◘or write a tutorial!"
004475E5  │. E8 8EA1FEFF     CALL 00431778                         └sF_cme04.00431778
004475EA  │> 84DB            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

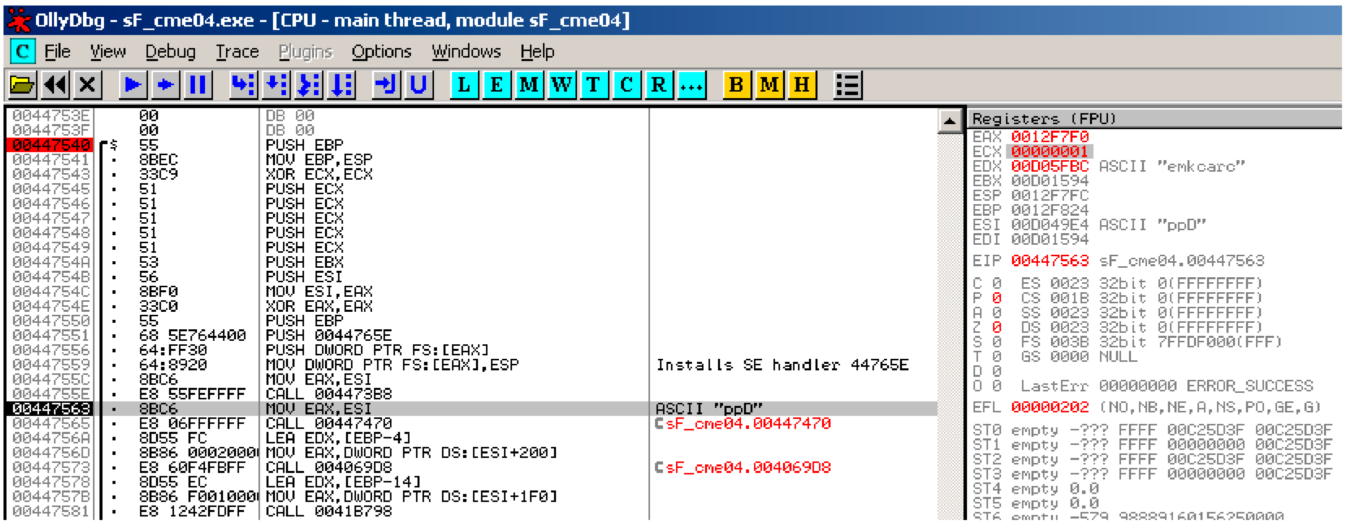

CrackMe No. 4

crackme

123456

About   Exit

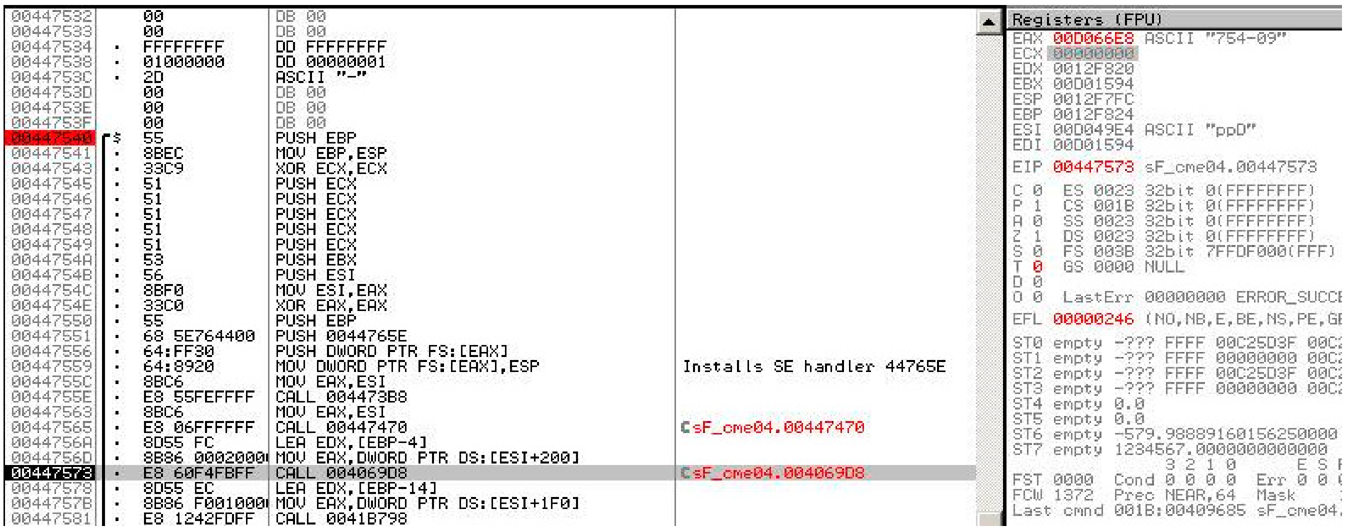
Figure 19. *Crackme window with both textboxes filled up*


OllyDbg - sF_cme04.exe - [CPU - main thread, module sF_cme04]

```
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                           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 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 0012F7F0
ECX 00000001
EDX 00D05FBC 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 -579.98889160156250000
```

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  ┌$  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 "754-09"
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_SUCCE

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

ST0 empty -??? FFFF 00C25D3F 00C2
ST1 empty -??? FFFF 00000000 00C2
ST2 empty -??? FFFF 00C25D3F 00C2
ST3 empty -??? FFFF 00000000 00C2
ST4 empty 0.0
ST5 empty 0.0
ST6 empty -579.98889160156250000
ST7 empty 1234567.0000000000000
               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).

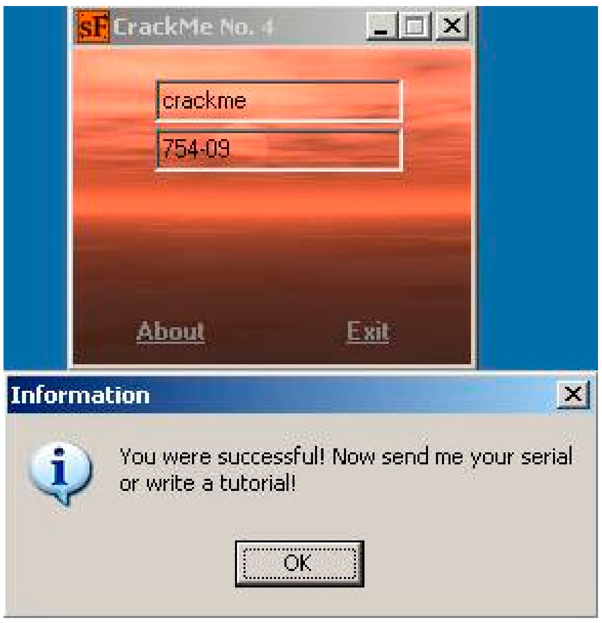
A few lines below – at address 447597, register EAX contains our magic value “754-09”, register EDX contains string “1234567” (which we entered to the second text box). Then at 00447597 a procedure is called and if a zero flag is set during the call of the procedure, then SETZ BL sets BL register to 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.

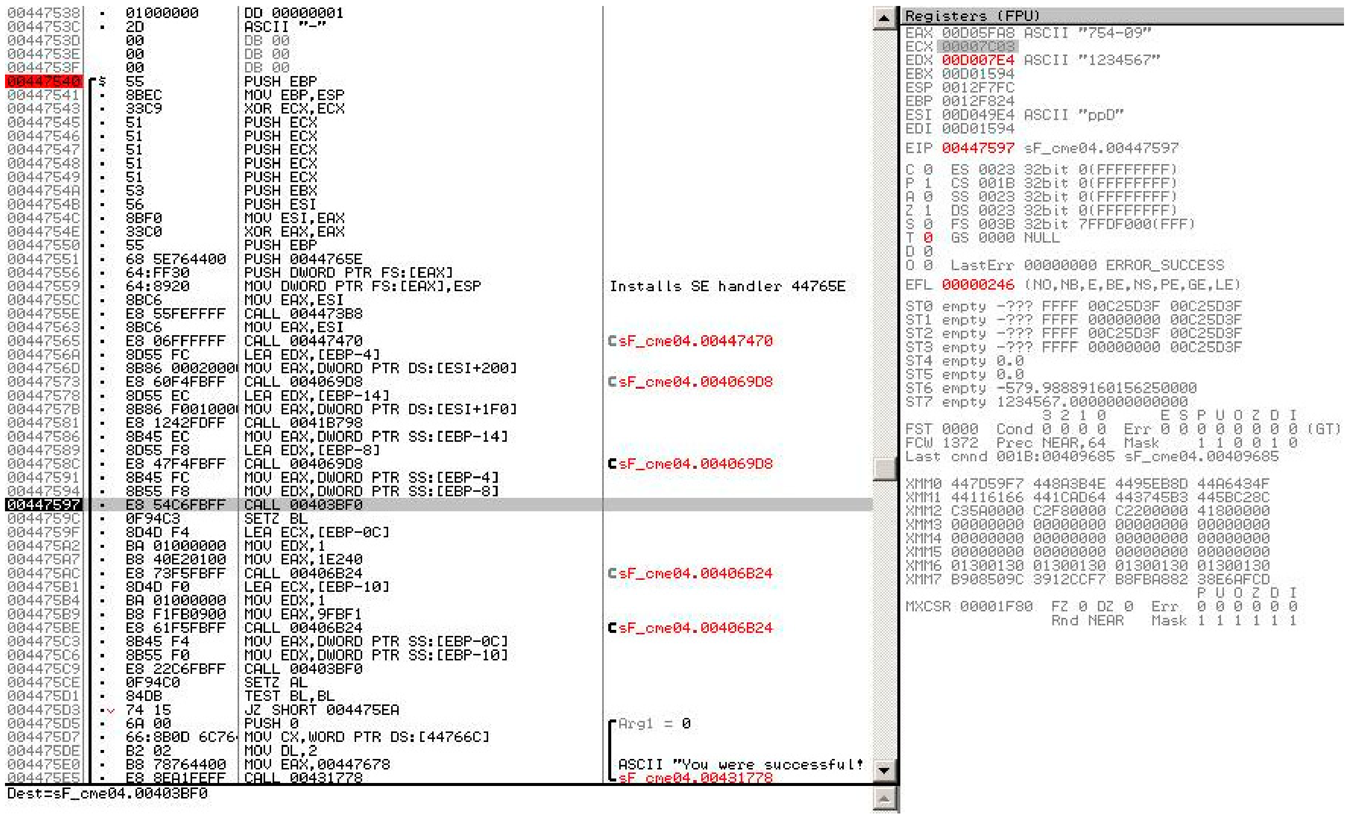
From the aforementioned description, we can expect that instruction CALL at address 447597 is comparison of two strings, which pointers are passed in registers EAX and EDX. You can simply verify it by keeping the first text box with text “crackme” and modifying the second text box to value “754-09”.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

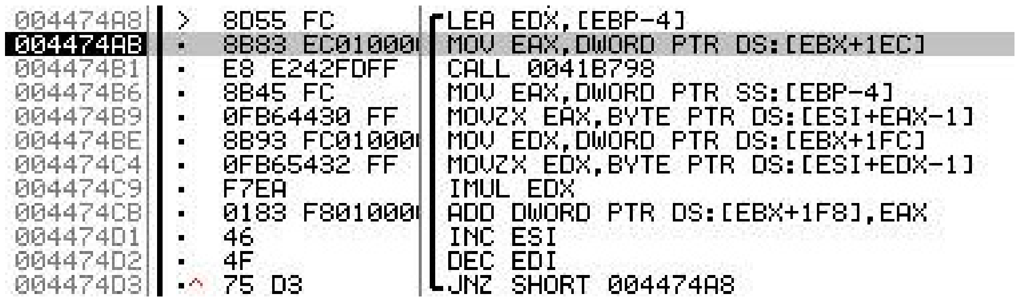
Figure 23. *Correct name/serial combination found*



```
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 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 54C6FBFF   CALL 00403BF0
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 22C6FBFF   CALL 00403BF0
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 78764400   MOV EAX,00447678                  ASCII "You were successful!
004475E5  .  E8 8EA1FEFF   CALL 00431778                     sF_cme04.00431778
Dest=sF_cme04.00403BF0
```

```
Registers (FPU)
EAX 00D05FA8 ASCII "754-09"
ECX 00007C03
EDX 00D007E4 ASCII "1234567"
EBX 00D01594
ESP 0012F7FC
EBP 0012F824
ESI 00D049E4 ASCII "ppD"
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 -579.98889160156250000
ST7 empty 1234567.0000000000000
               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 1372  Prec NEAR,64  Mask    1 1 0 0 1 0
Last cmnd 001B:00409685 sF_cme04.00409685

XMM0 447D59F7 448A3B4E 4495EB8D 44A6434F
XMM1 44116166 441CAD64 443745B3 445BC28C
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 B908509C 3912CCF7 B8FBA882 38E6AFCD
                                P U O Z D I
MXCSR 00001F80  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 EC01000    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 FC01000    MOV EDX,DWORD PTR DS:[EBX+1FC]
004474C4  .   0FB65432 FF     MOVZX EDX,BYTE PTR DS:[ESI+EDX-1]
004474C9  .   F7EA            IMUL EDX
004474CB  .   0183 F801000    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 icritical websense 8MAN WEBROOT Cisco Systems SOURCEfire maildistiller

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.

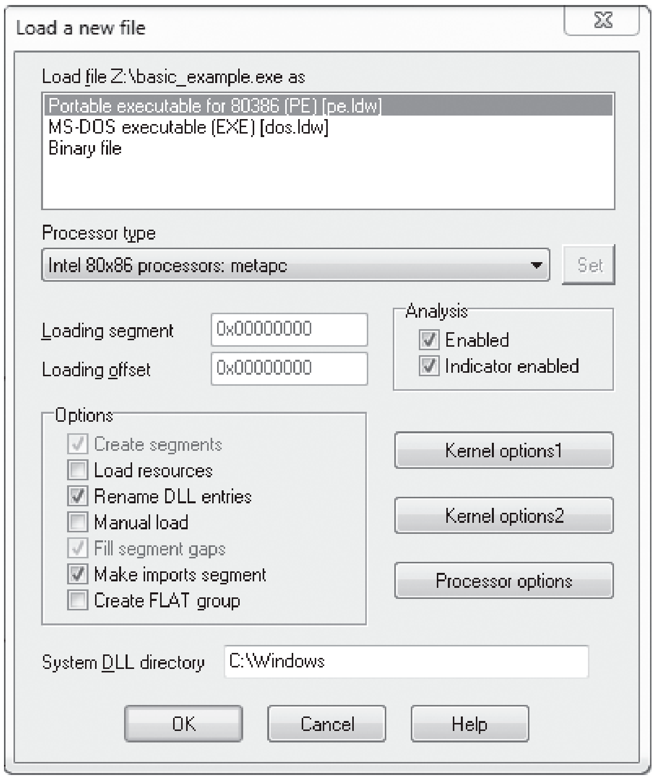
The program is published by Hex-Rays (*http://www.hex-rays.com*), which provides a free version for non-commercial uses that is one version less than the current paid version. It is now version 5.0.

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 exe-


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*

cutables 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.

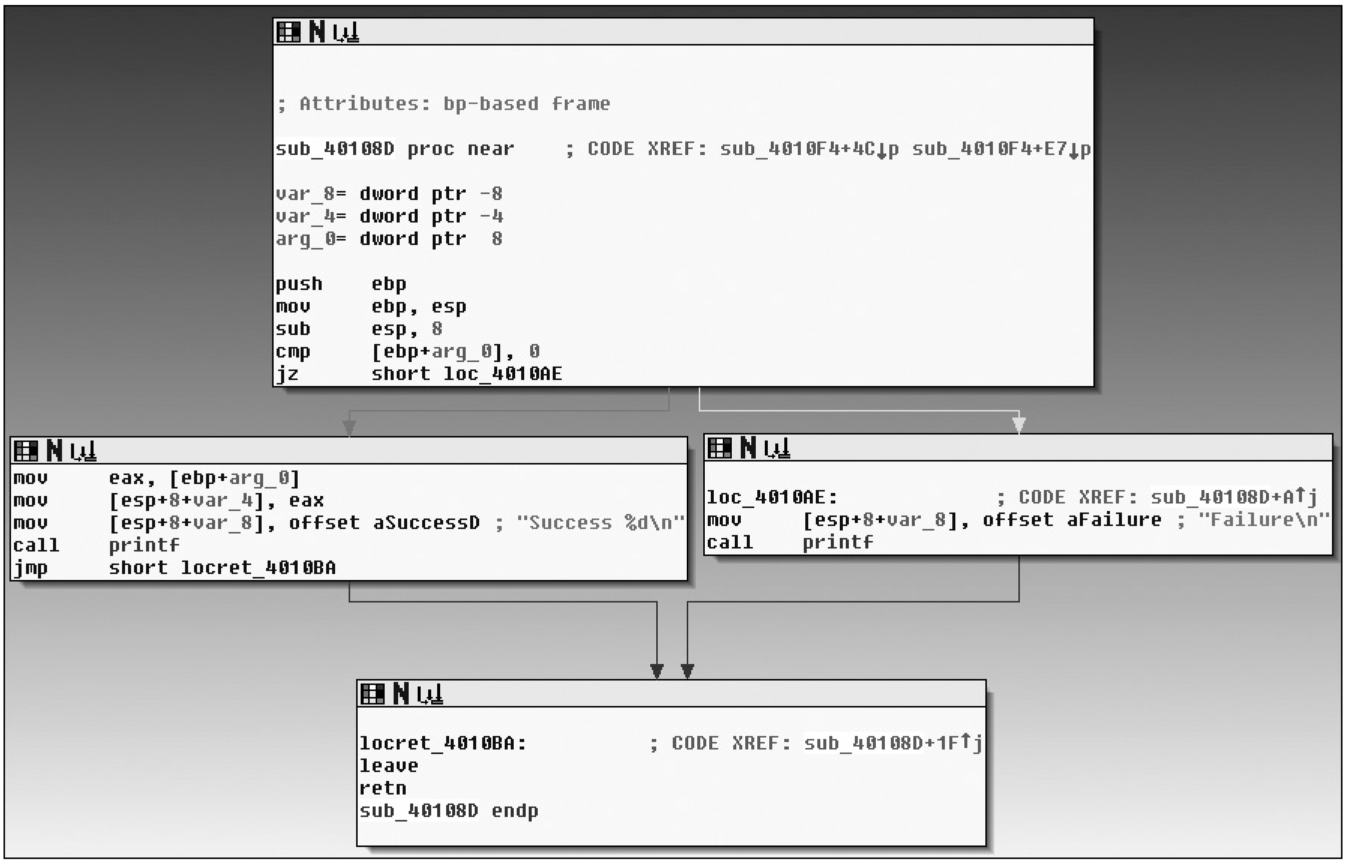
Disassembly Window Modes

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.

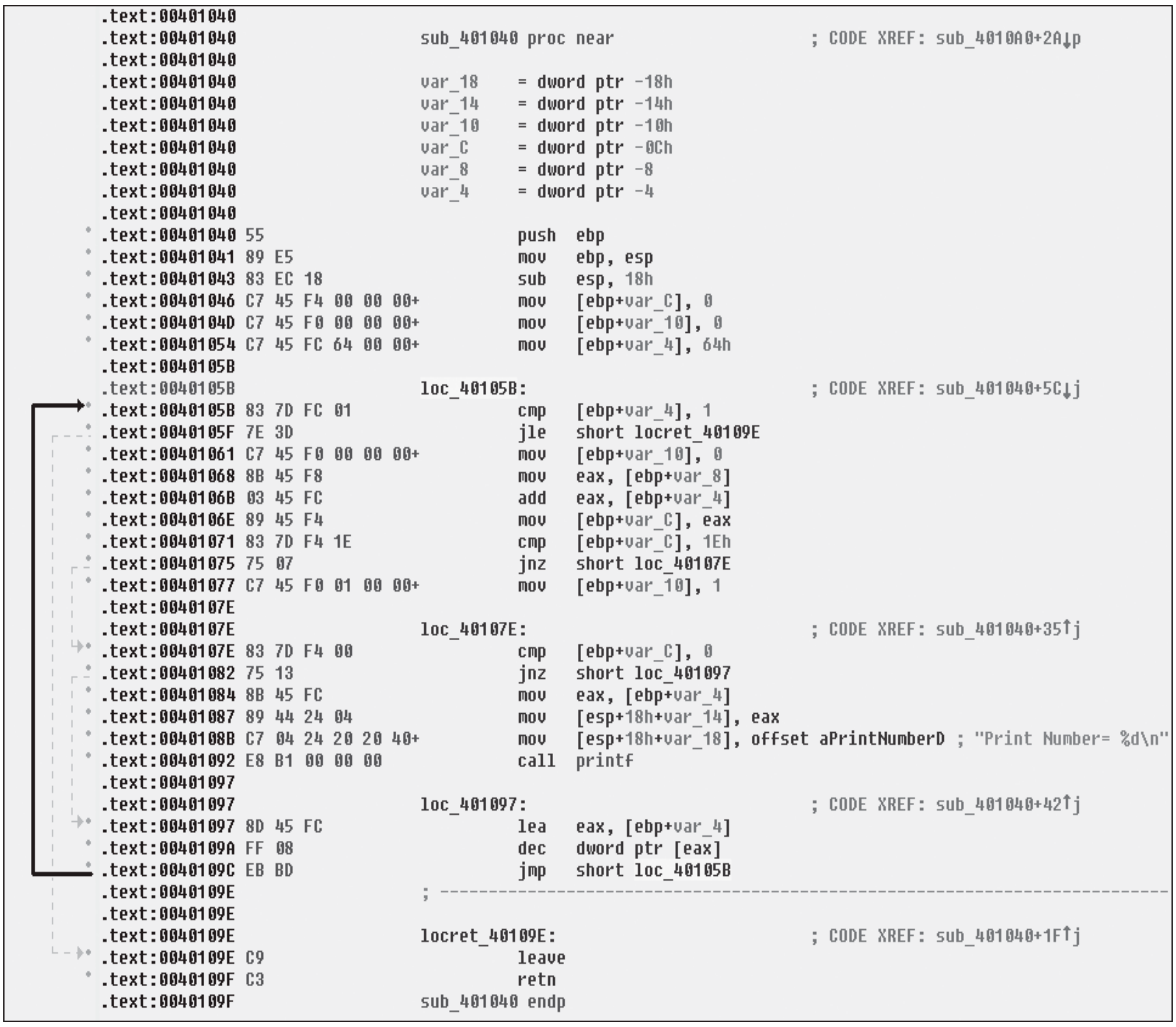
The left portion of the text-mode display is known as the arrows window and shows the program's nonlinear flow. Solid lines mark unconditional jumps, and dashed lines mark conditional jumps. Arrows facing up indicate a loop. The example includes the stack layout for the function and a comment (beginning 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 B1 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.

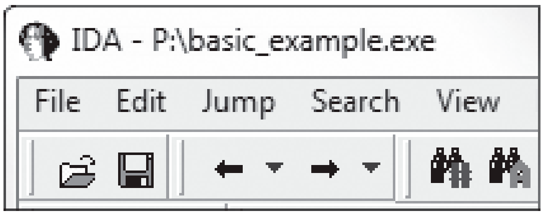


Figure 4. *Navigational buttons*

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
```

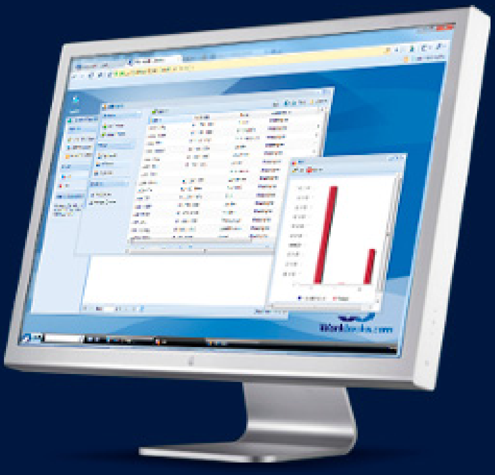
a d v e r t i s e m e n t

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

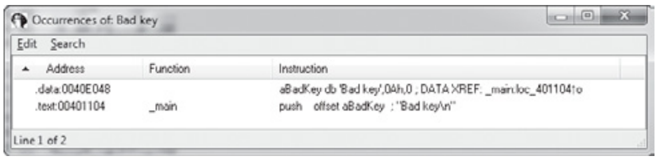
Another way to navigate IDA Pro is to use the links within the disassembly window, such as the links shown in Listing 1. Double-clicking any of these links will display the target location in the disassembly window. The following are the most common types of links:

- *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"
Line 1 of 2
```

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.

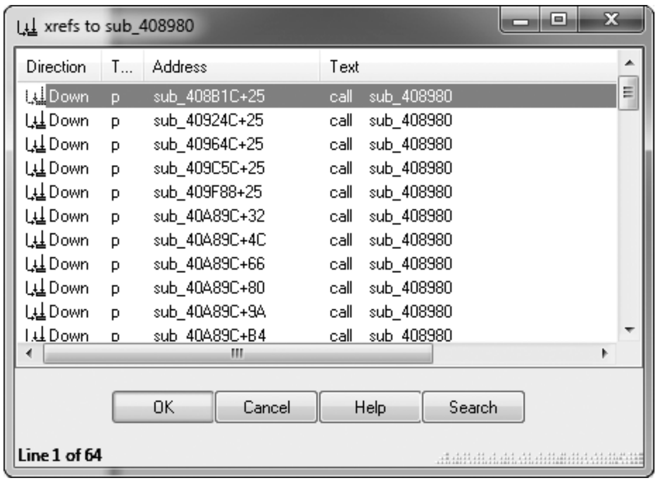
- Choose *Search→Sequence of Bytes* to perform a binary search in the hex view window for a certain byte order. This option can be useful when you're searching for specific data or opcode combinations.

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



Direction	T...	Address	Text
Down	p	sub_408B1C+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

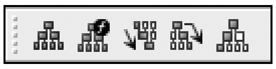


Figure 7. *Graphing button toolbar*

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.

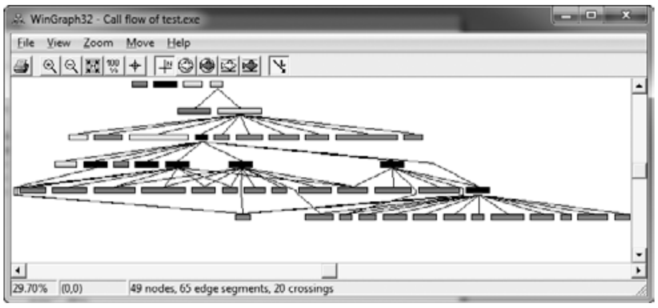

WinGraph32 - Call flow of test.exe
File View Zoom Move Help
29.70% (0,0) 49 nodes, 65 edge segments, 20 crossings

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

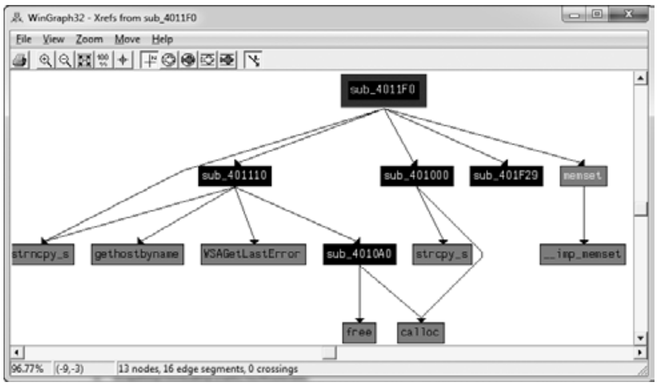


Figure 9. *Cross-reference graph of a single function (sub_4011F0)*

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.

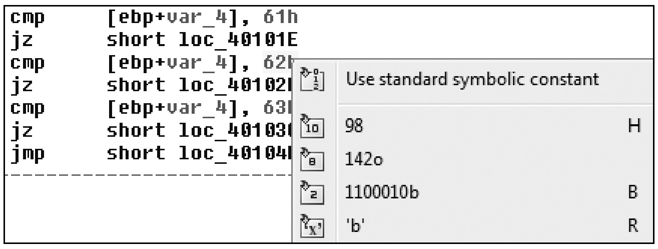
After you've renamed a dummy name to something more meaningful, cross-references will become much easier to parse. For example, if a function `sub_401200` is called many times throughout a program and you rename it to DNSrequest, it will be renamed DNSrequest throughout the program. Imagine how much 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

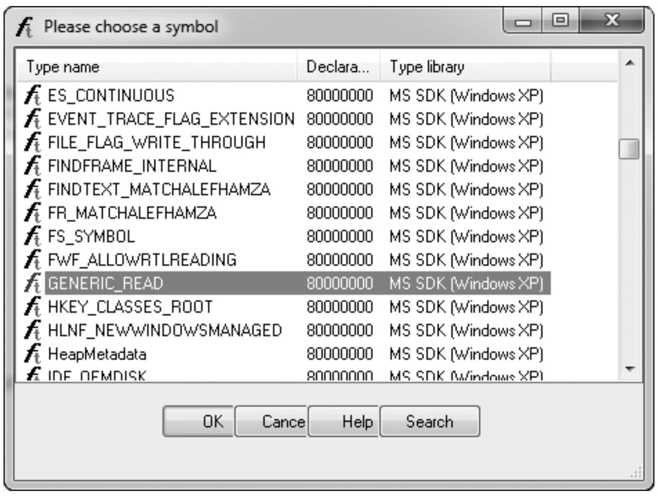
IDA Pro lets you embed comments throughout your disassembly and adds many comments automatically.

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_

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`	

READ in their source code. Named constants provide 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.

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.

On the Web

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

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

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).

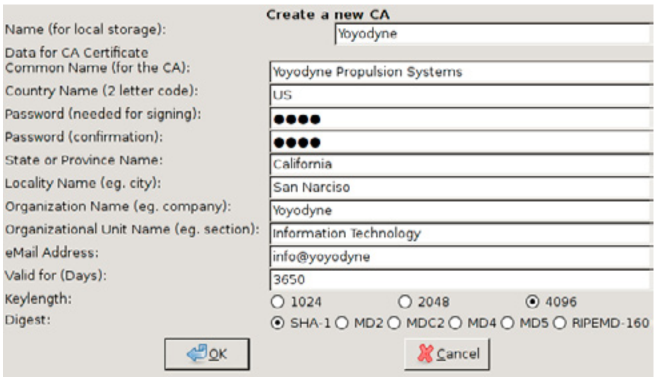
It doesn't matter what you put here if you don't plan on keeping this certificate authority information for very long. The target server at LiveJournal.com will never see the keys you are generating and they will stay completely isolated to your testing environment. Be sure to remember the password since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

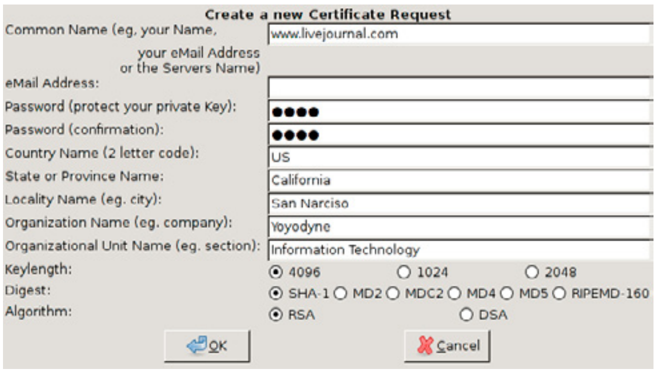
The common name must be *www.livejournal.com*. The password can be anything and we will be removing it when we export the key for use.



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

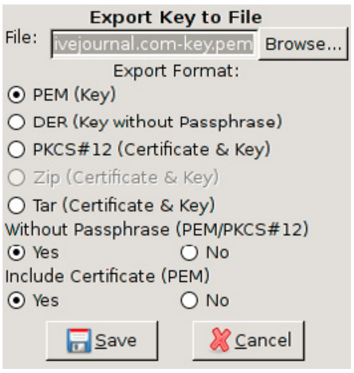
So we have to dig in a bit to understand what SSL Certificate trust database LogJam will be using. Most Linux based GTK and console programs rely on OpenSSL which has it's own certificate authority database that is very easy to add a new certificate to.

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: ivejournal.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-
                  cacert.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
                  configuration
-----------------------------
  ...
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
```

ISAF

The **Most Comprehensive** Exhibition of the Fastest Growing Sectors of recent years in the **Center of Eurasia**

ISAF IT SECURITY
INFORMATION, DATA AND NETWORK SECURITY EXHIBITION

ISAF SAFETY&HEALTH
OCCUPATIONAL SAFETY AND HEALTH EXHIBITION

ISAF SMART HOUSES
SMART HOUSES AND BUILDING AUTOMATION EXHIBITION

ISAF SECURITY
16th INTERNATIONAL SECURITY AND RFID EXHIBITION

ISAF FIRE
16th INTERNATIONAL FIRE, EMERGENCY RESCUE EXHIBITION

www.isaffuari.com

SEPTEMBER 20th - 23rd, 2012
IFM ISTANBUL EXPO CENTER (IDTM)

KOSGEB

MARMARA TANITIM FUARCILIK
T. +90 212 503 32 32 | marmara@marmarafuar.com.tr
www.marmarafuar.com.tr

THIS EXHIBITION IS ORGANIZED WITH THE PERMISSIONS OF T.O.B.B. IN ACCORDANCE WITH THE LAW NUMBER 5174.

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 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).

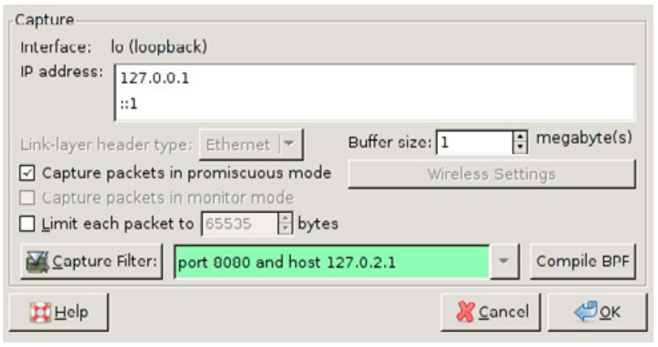
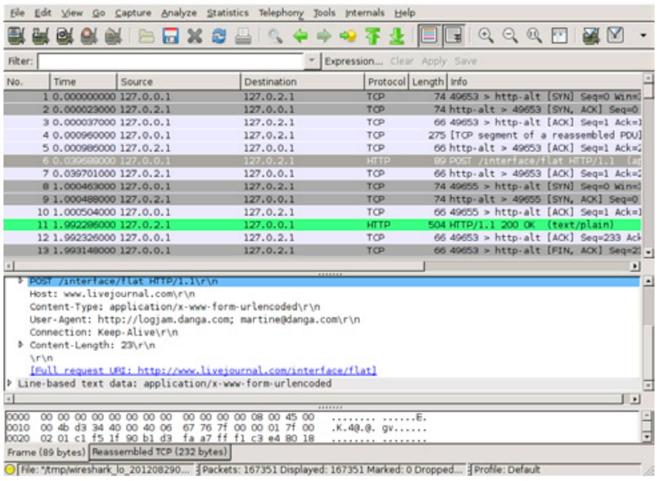

Capture
Interface: lo (loopback)
IP address: 127.0.0.1
::1
Link-layer header type: Ethernet
Buffer size: 1 megabyte(s)
☑ Capture packets in promiscuous mode
☐ Capture packets in monitor mode
☐ Limit each packet to 65535 bytes
Capture Filter: port 8080 and host 127.0.2.1

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


File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save

```
No.  Time        Source     Destination Protocol Length Info
   1 0.000000000 127.0.0.1  127.0.2.1   TCP      74 49653 > http-alt [SYN] Seq=0 Win=
   2 0.000023000 127.0.2.1  127.0.0.1   TCP      74 http-alt > 49653 [SYN, ACK] Seq=0
   3 0.000037000 127.0.0.1  127.0.2.1   TCP      66 49653 > http-alt [ACK] Seq=1 Ack=1
   4 0.000960000 127.0.0.1  127.0.2.1   TCP     275 [TCP segment of a reassembled PDU]
   5 0.000986000 127.0.2.1  127.0.0.1   TCP      66 http-alt > 49653 [ACK] Seq=1 Ack=
   6 0.039688000 127.0.0.1  127.0.2.1   HTTP     89 POST /interface/flat HTTP/1.1  (a
   7 0.039701000 127.0.2.1  127.0.0.1   TCP      66 http-alt > 49653 [ACK] Seq=1 Ack=
   8 1.000463000 127.0.0.1  127.0.2.1   TCP      74 49655 > http-alt [SYN] Seq=0 Win=
   9 1.000488000 127.0.2.1  127.0.0.1   TCP      74 http-alt > 49655 [SYN, ACK] Seq=0
  10 1.000504000 127.0.0.1  127.0.2.1   TCP      66 49655 > http-alt [ACK] Seq=1 Ack=1
  11 1.992286000 127.0.2.1  127.0.0.1   HTTP    504 HTTP/1.1 200 OK  (text/plain)
  12 1.992326000 127.0.0.1  127.0.2.1   TCP      66 49653 > http-alt [ACK] Seq=233 Ack
  13 1.993148000 127.0.0.1  127.0.2.1   TCP      66 49653 > http-alt [FIN, ACK] Seq=2

  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: http://www.livejournal.com/interface/flat]
Line-based text data: application/x-www-form-urlencoded

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E.
0010  00 4b d3 34 40 00 40 06  67 76 7f 00 00 01 7f 00   .K.4@.@. gv......
0020  02 01 c1 f5 1f 90 b1 d3  fa a7 ff f1 c3 e4 80 18   ........ ........
Frame (89 bytes) Reassembled TCP (232 bytes)
File: "/tmp/wireshark_lo_201208290...  Packets: 167351 Displayed: 167351 Marked: 0 Dropped...  Profile: Default
```

Figure 5. *Wireshark with captured unencrypted packets*

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.

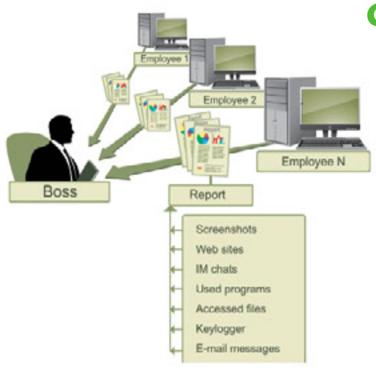
Remember to thank your open source hacker friends.

SHANE R. SPENCER

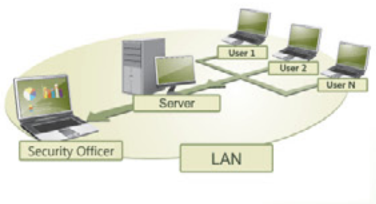
Shane R. Spencer is based out of Anchorage Alaska and has over 10 years of system administration and programming experience. Many of his projects are Python based and interface with external services that provide no usable API and communicate over HTTPS only.

STAFFCOP

PC monitoring, Corporate Security and Data Loss Prevention Software



StaffCop Standard allows you to monitor all activities on company computers and prevent the unauthorized distribution of sensitive corporate information.



StaffCop will help you:

To locate possible data loss channels and prevent loss
To gain insight into how your employees spend their work time
To increase company and departmentals efficiency

You need StaffCop to:

Gather work time efficiency statistics
Easily control your employees in real-time mode
Improve discipline and motivation of your employees

Main Features of StaffCop:

Screenshot recording
Application monitoring
E-mail monitoring
Web site monitoring
Chats/IM activity recording
USB device monitoring
Clipboard monitoring
Social Networks Monitoring
Search Term Tracking
File and Folder tracking
Keystroke recording
System Event Monitoring
Whitelists and Blacklists
PC activities reporting
Stealth installation/monitoring
Strong security
Alert notifications
Remote Install / Uninstall

Who needs StaffCop:

CEO/CTO
Corporate Security Manager
HR Manager
System Administrator

More Information, Demo Versions, Videos and Technical Guides -

www.STAFFCOP.com

Phone: +1-707 -7098405
Skype: staffcop.com
Email: sales@staffcop.com, paul@atompark.com

Microsoft CERTIFIED Partner

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, de-obfuscate 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

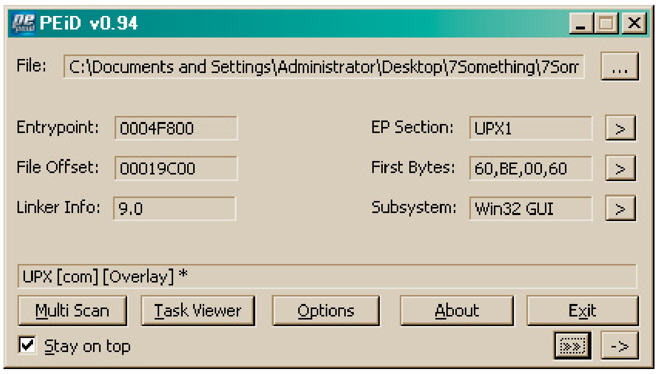
Packer algorithms are employed in order to distort the code of a compiled binary. A packing application takes the algorithm, runs the data of the binary through it, and attaches a decryption routine to the binary. The resulting file is a distorted version of the original and, if fed into a disassembler 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] *

Figure 1. *PEID Interface*

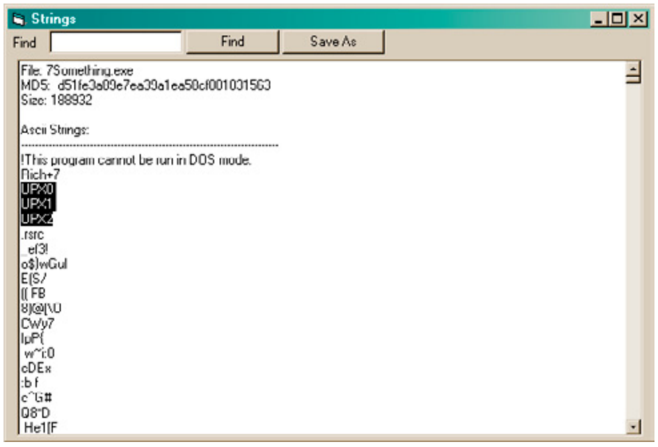
base full of different packer signatures in order to identify what packing algorithm is in use.

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

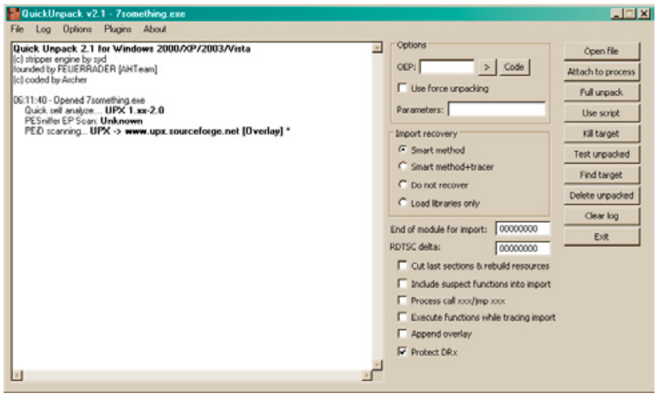
If you do not have access to PEID or it does not recognize the packer employed, you might have some luck by examining certain features of the binary, looking for anything that might reveal the packer. In some cases that is incredibly easy, for example figure 2 shows the file strings associated with a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



```
File: 7Something.exe
MD5: d51fe3a09e7ea39a1ea50cf001031563
Size: 188932

Ascii Strings:
------------------------------------------------------------
!This program cannot be run in DOS mode.
Rich+7
UPX0
UPX1
UPX2
.rsrc
 _ef3!
o$)wGul
E(S/
(( FB
8)@[\U
CWy7
IpP(
 w^i:0
cDEx
:b f
c^G#
Q8"D
 He1[F
```

Figure 2. *UPX File Strings*



```
Quick Unpack 2.1 for Windows 2000/XP/2003/Vista
(c) stripper engine by syd
founded by FEUERRADER [AHTeam]
(c) coded by Archer

06:11:40 - Opened 7something.exe
    Quick self analyze... UPX 1.xx-2.0
    PESniffer EP Scan: Unknown
    PEiD scanning... UPX -> www.upx.sourceforge.net [Overlay] *
```

Import recovery: [✓ Smart method]; End of module for import: 00000000; RDTSC delta: 00000000; [✓ Protect DRx]

Figure 3. *QUnpack Interface*

best left unknown and you might not be able to unpack it in any easy way.

Custom Packer

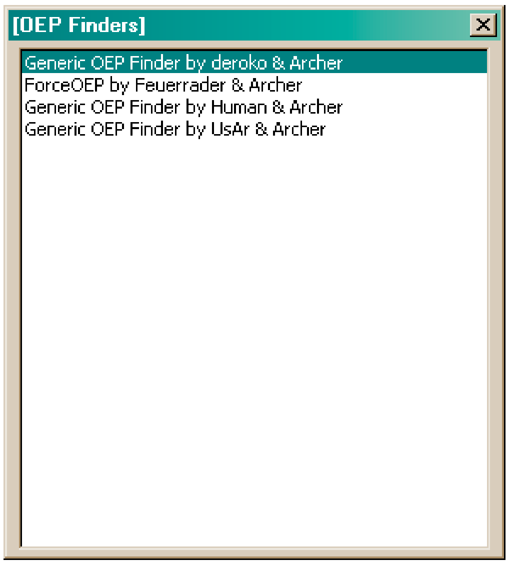
While there are plenty of publicly known packers out there and many of them are used by both legitimate software and malware organizations, it does not mean they are the only ones used. Cybercrime organizations will create their own "custom packer algorithm" which they can quickly modify in order to 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.

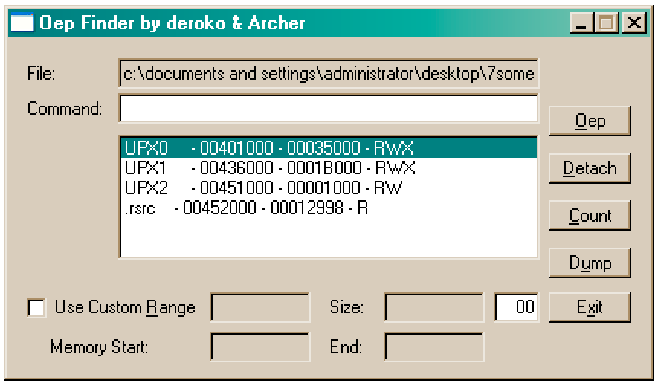

```
Oep Finder by deroko & Archer
File:     c:\documents and settings\administrator\desktop\7some
Command:
UPX0   - 00401000 - 00035000 - RWX
UPX1   - 00436000 - 0001B000 - RWX
UPX2   - 00451000 - 00001000 - RW
.rsrc  - 00452000 - 00012998 - R
Use Custom Range      Size:        00
Memory Start:         End:
```

Figure 5. *OEP Finder Interface*

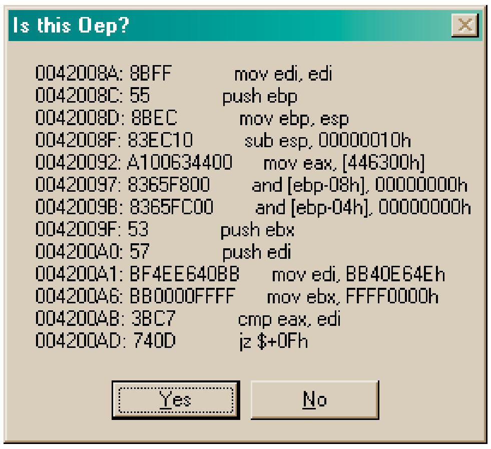

```
Is this Oep?
0042008A: 8BFF          mov edi, edi
0042008C: 55            push ebp
0042008D: 8BEC          mov ebp, esp
0042008F: 83EC10        sub esp, 00000010h
00420092: A100634400    mov eax, [446300h]
00420097: 8365F800      and [ebp-08h], 00000000h
0042009B: 8365FC00      and [ebp-04h], 00000000h
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*

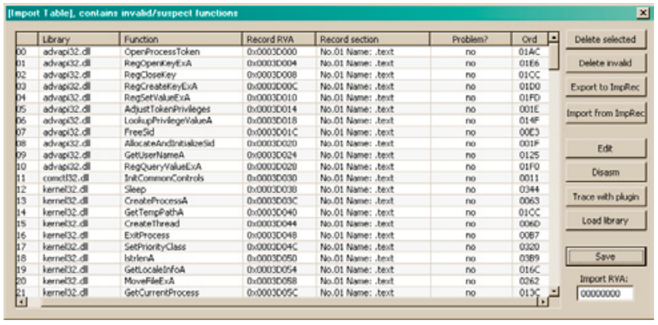
With the OEP located, our next step is to click on the "Full Unpack" button on the right side of the QUnpack interface. The unpacker will analyze the binary and attempt to retrieve the import table. Keep in mind that this might not happen with other packers or a binary using a custom packer; lucky for 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

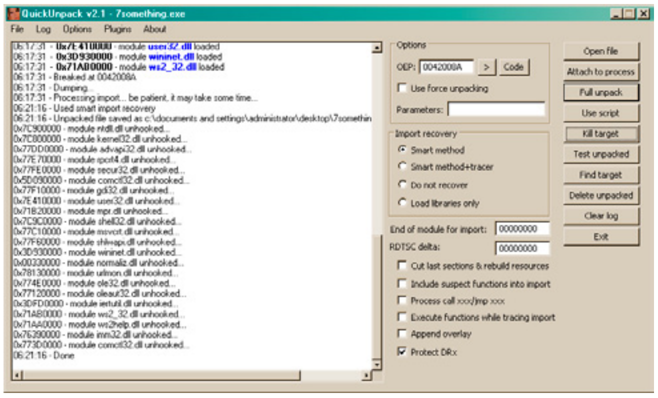
Automated unpacking is the most efficient way of revealing the true code of a packed binary. How-


[Import Table], contains invalid/suspect functions

	Library	Function	Record RVA	Record section	Problem?	Ord
00	advapi32.dll	OpenProcessToken	0x0003D000	No.01 Name: .text	no	01AC
01	advapi32.dll	RegOpenKeyExA	0x0003D004	No.01 Name: .text	no	01E6
02	advapi32.dll	RegCloseKey	0x0003D008	No.01 Name: .text	no	01CC
03	advapi32.dll	RegCreateKeyExA	0x0003D00C	No.01 Name: .text	no	01D0
04	advapi32.dll	RegSetValueExA	0x0003D010	No.01 Name: .text	no	01FD
05	advapi32.dll	AdjustTokenPrivileges	0x0003D014	No.01 Name: .text	no	001E
06	advapi32.dll	LookupPrivilegeValueA	0x0003D018	No.01 Name: .text	no	014F
07	advapi32.dll	FreeSid	0x0003D01C	No.01 Name: .text	no	00E3
08	advapi32.dll	AllocateAndInitializeSid	0x0003D020	No.01 Name: .text	no	001F
09	advapi32.dll	GetUserNameA	0x0003D024	No.01 Name: .text	no	0125
10	advapi32.dll	RegQueryValueExA	0x0003D028	No.01 Name: .text	no	01F0
11	comctl32.dll	InitCommonControls	0x0003D030	No.01 Name: .text	no	0011
12	kernel32.dll	Sleep	0x0003D038	No.01 Name: .text	no	0344
13	kernel32.dll	CreateProcessA	0x0003D03C	No.01 Name: .text	no	0063
14	kernel32.dll	GetTempPathA	0x0003D040	No.01 Name: .text	no	01CC
15	kernel32.dll	CreateThread	0x0003D044	No.01 Name: .text	no	006D
16	kernel32.dll	ExitProcess	0x0003D048	No.01 Name: .text	no	0087
17	kernel32.dll	SetPriorityClass	0x0003D04C	No.01 Name: .text	no	0320
18	kernel32.dll	lstrlenA	0x0003D050	No.01 Name: .text	no	0389
19	kernel32.dll	GetLocaleInfoA	0x0003D054	No.01 Name: .text	no	016C
20	kernel32.dll	MoveFileExA	0x0003D058	No.01 Name: .text	no	0262
21	kernel32.dll	GetCurrentProcess	0x0003D05C	No.01 Name: .text	no	013C

Import RVA: 00000000

Figure 7. *QUnpack Import Table Output*


```
QuickUnpack v2.1 - 7something.exe
06:17:31 - 0x7E410000 - module user32.dll loaded
06:17:31 - 0x3D930000 - module wininet.dll loaded
06:17:31 - 0x71AB0000 - module ws2_32.dll loaded
06:17:31 - Breaked at 0042008A
06:17:31 - Dumping...
06:17:31 - Processing import... be patient, it may take some time...
06:21:16 - Used smart import recovery
06:21:16 - Unpacked file saved as c:\documents and settings\administrator\desktop\7somethin
0x7C900000 - module ntdll.dll unhooked...
0x7C800000 - module kernel32.dll unhooked...
0x77DD0000 - module advapi32.dll unhooked...
0x77E70000 - module rpcrt4.dll unhooked...
0x77FE0000 - module secur32.dll unhooked...
0x5D090000 - module comctl32.dll unhooked...
0x77F10000 - module gdi32.dll unhooked...
0x7E410000 - module user32.dll unhooked...
0x71B20000 - module mpr.dll unhooked...
0x7C9C0000 - module shell32.dll unhooked...
0x77C10000 - module msvcrt.dll unhooked...
0x77F60000 - module shlwapi.dll unhooked...
0x3D930000 - module wininet.dll unhooked...
0x00330000 - module normaliz.dll unhooked...
0x78130000 - module urlmon.dll unhooked...
0x774E0000 - module ole32.dll unhooked...
0x77120000 - module oleaut32.dll unhooked...
0x3DFD0000 - module iertutil.dll unhooked...
0x71AB0000 - module ws2_32.dll unhooked...
0x71AA0000 - module ws2help.dll unhooked...
0x76390000 - module imm32.dll unhooked...
0x773D0000 - module comctl32.dll unhooked...
06:21:16 - Done
```

Options
OEP: 0042008A
☐ Use force unpacking
Parameters:

Import recovery
● Smart method
○ Smart method+tracer
○ Do not recover
○ Load libraries only

End of module for import: 00000000
RDTSC delta: 00000000
☐ Cut last sections & rebuild resources
☐ Include suspect functions into import
☐ Process call xxx/jmp xxx
☐ Execute functions while tracing import
☐ Append overlay
☑ Protect DRx

Figure 8. *QUnpack unpacked operations output*

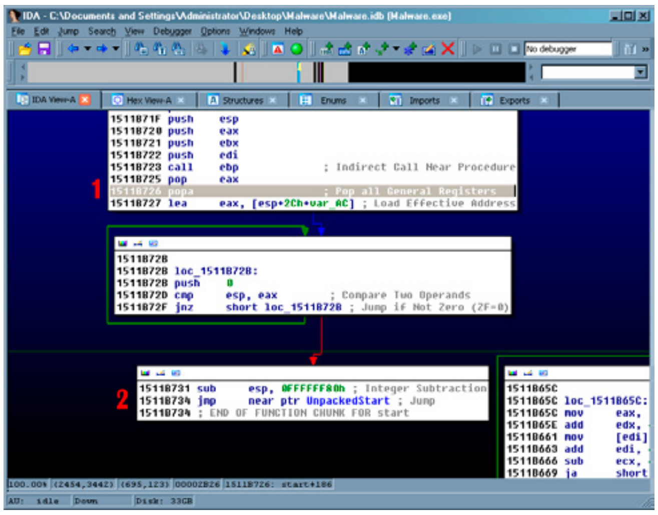
ever, there may be some instances when using an unpacker might not work, in which case you will need to unpack the binary manually. You might find yourself in this situation if you are working on a binary that is packed with a custom algorithm or if dealing with a modified known packer, resulting in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


IDA - C:\Documents and Settings\Administrator\Desktop\Malware\Malware.idb (Malware.exe)
File Edit Jump Search View Debugger Options Windows Help
No debugger
IDA View-A
Hex View-A
Structures
Enums
Imports
Exports
1
2

```
1511871F push    esp
15118720 push    eax
15118721 push    ebx
15118722 push    edi
15118723 call    ebp                     ; Indirect Call Near Procedure
15118725 pop     eax
15118726 popa                            ; Pop all General Registers
15118727 lea     eax, [esp+2Ch+var_AC] ; Load Effective Address

1511872B
1511872B loc_1511872B:
1511872B push    0
1511872D cmp     esp, eax          ; Compare Two Operands
1511872F jnz     short loc_1511872B ; Jump if Not Zero (ZF=0)

15118731 sub     esp, 0FFFFFF80h ; Integer Subtraction
15118734 jmp     near ptr UnpackedStart ; Jump
15118734 ; END OF FUNCTION CHUNK FOR start

1511865C
1511865C loc_1511865C:
1511865C mov     eax,
1511865E add     edx,
15118661 mov     [edi]
15118663 add     edi,
15118666 sub     ecx,
15118669 ja      short

100.00% (2454,3442) (695,123) 00002B26 15118726: start+186
AU:  idle   Down      Disk: 33GB
```

Figure 9. *Unpacking algorithm exit JMP call*

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).

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).

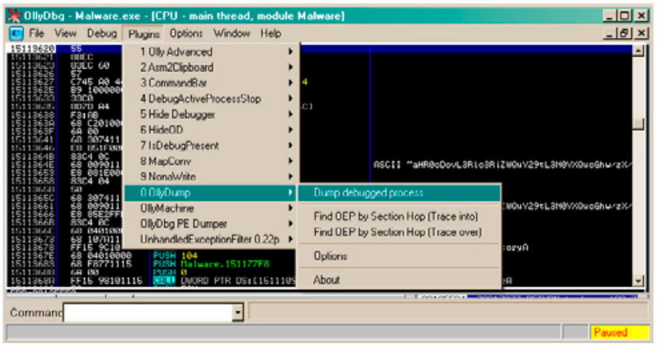
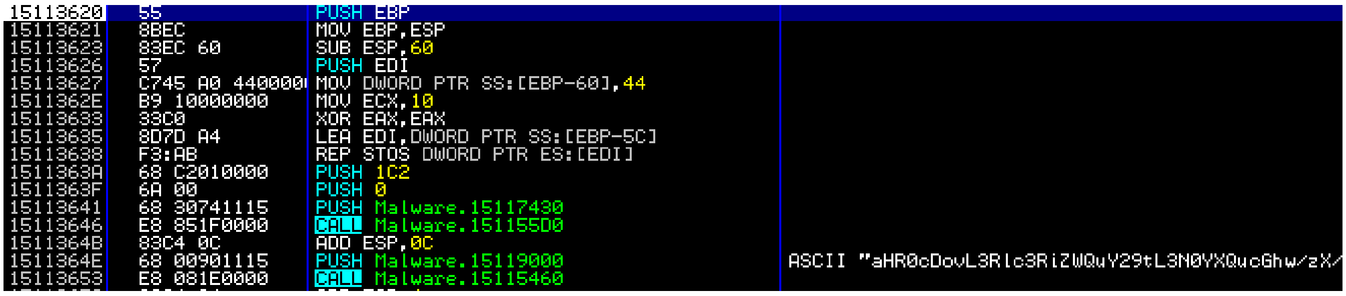


Figure 11. *OllyDump menu navigation*

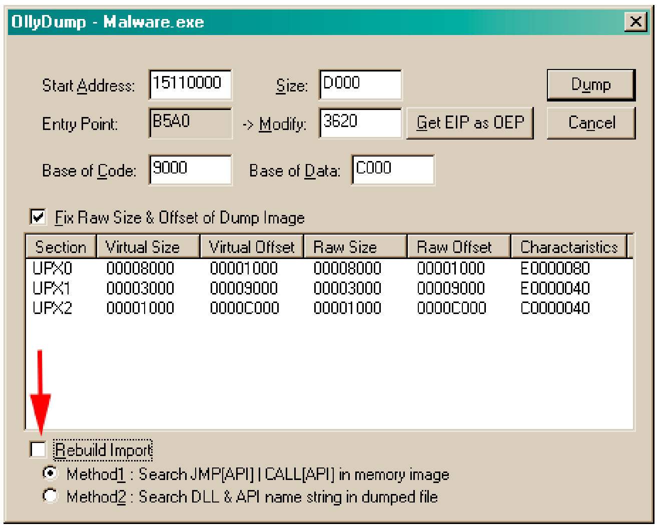


```
15113620  55             PUSH EBP
15113621  8BEC           MOV EBP,ESP
15113623  83EC 60        SUB ESP,60
15113626  57             PUSH EDI
15113627  C745 A0 440000 MOV DWORD PTR SS:[EBP-60],44
1511362E  B9 10000000    MOV ECX,10
15113633  33C0           XOR EAX,EAX
15113635  8D7D A4        LEA 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*

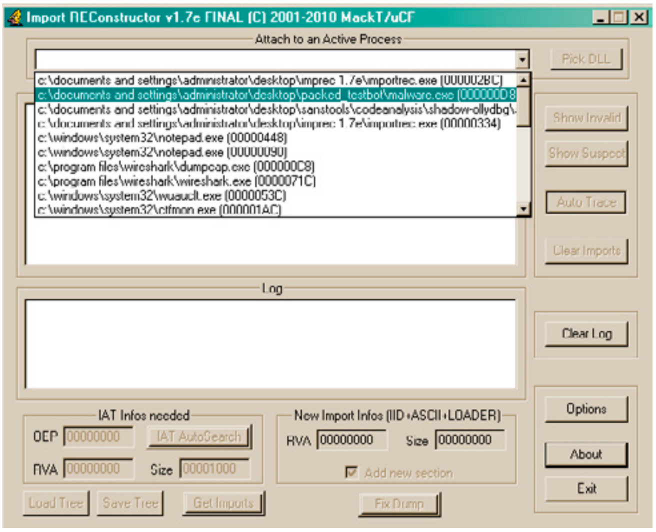
ImpREC

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).

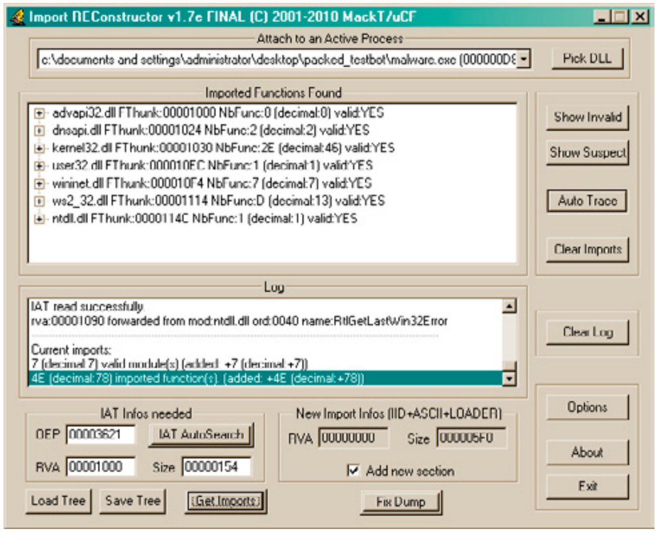

Import REConstructor v1.7e FINAL (C) 2001-2010 MackT/uCF

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 (000000D8
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 (00000448)
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 [✓] Add new section

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 (000000D8

Imported Functions Found

```
+ advapi32.dll FThunk:00001000 NbFunc:0 (decimal:0) 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.

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

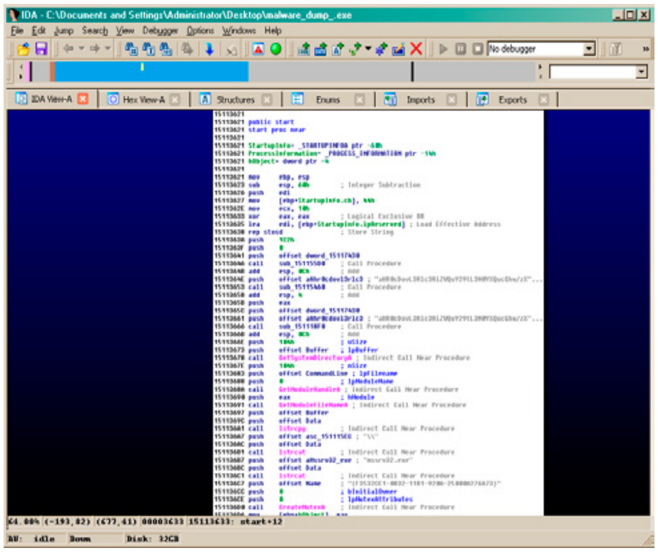
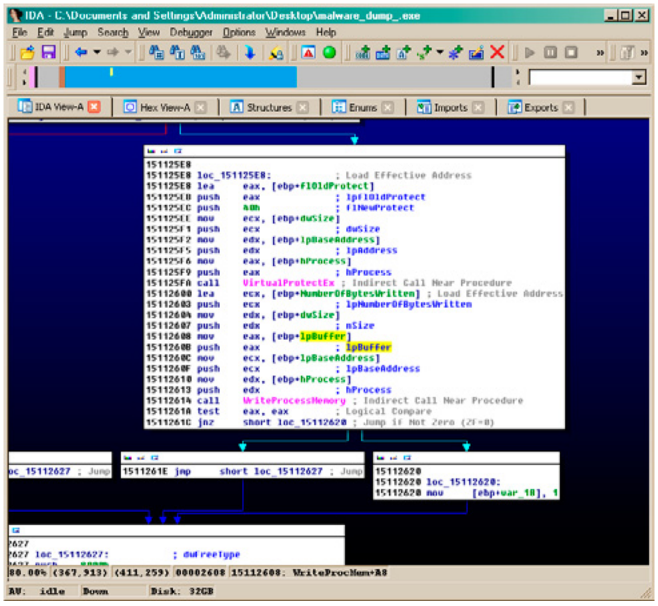

IDA - C:\Documents and Settings\Administrator\Desktop\malware_dump_.exe
File Edit Jump Search View Debugger Options Windows Help
No debugger
IDA View-A
Hex View-A
Structures
Enums
Imports
Exports

Figure 15. *Unpacked binary loaded in IDA Pro*


IDA - C:\Documents and Settings\Administrator\Desktop\malware_dump_.exe
File Edit Jump Search View Debugger Options Windows Help
IDA View-A
Hex View-A
Structures
Enums
Imports
Exports

```
151125E8
151125E8 loc_151125E8:                ; Load Effective Address
151125E8 lea     eax, [ebp+flOldProtect]
151125EB push    eax                  ; lpflOldProtect
151125EC push    40h                  ; flNewProtect
151125EE mov     ecx, [ebp+dwSize]
151125F1 push    ecx                  ; dwSize
151125F2 mov     edx, [ebp+lpBaseAddress]
151125F5 push    edx                  ; lpAddress
151125F6 mov     eax, [ebp+hProcess]
151125F9 push    eax                  ; hProcess
151125FA call    VirtualProtectEx ; Indirect Call Near Procedure
15112600 lea     ecx, [ebp+NumberOfBytesWritten] ; Load Effective Address
15112603 push    ecx                  ; lpNumberOfBytesWritten
15112604 mov     edx, [ebp+dwSize]
15112607 push    edx                  ; nSize
15112608 mov     eax, [ebp+lpBuffer]
1511260B push    eax                  ; lpBuffer
1511260C mov     ecx, [ebp+lpBaseAddress]
1511260F push    ecx                  ; lpBaseAddress
15112610 mov     edx, [ebp+hProcess]
15112613 push    edx                  ; hProcess
15112614 call    WriteProcessMemory ; Indirect Call Near Procedure
1511261A test    eax, eax             ; Logical Compare
1511261C jnz     short loc_15112620 ; Jump if Not Zero (ZF=0)

1511261E jmp     short loc_15112627 ; Jump

15112620
15112620 loc_15112620:
15112620 mov     [ebp+var_18], 1

15112627 loc_15112627:           ; dwFreeType
```

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

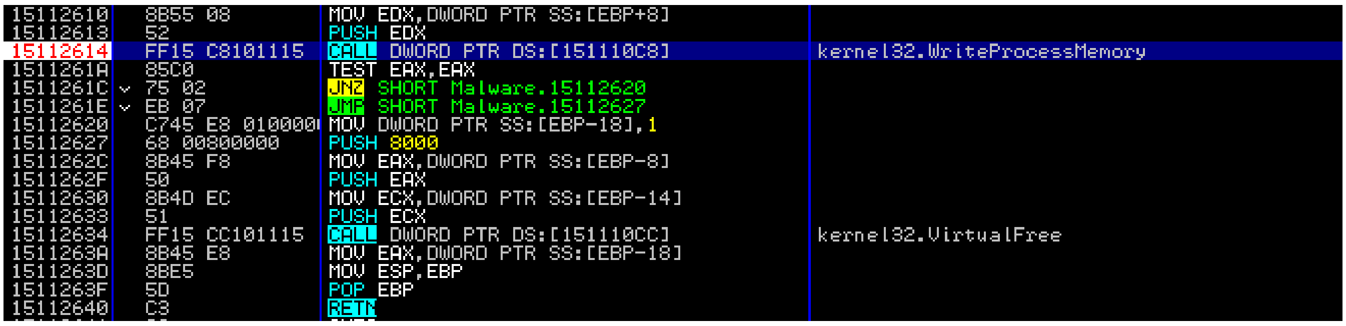
and the possibility of getting lost in the code by parsing through with only a debugger is slim to none. In addition, you can be prepared for the creation of dynamic memory and keep track of what variables are being referenced or what data is being copied. I find that when attempting to extract previously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

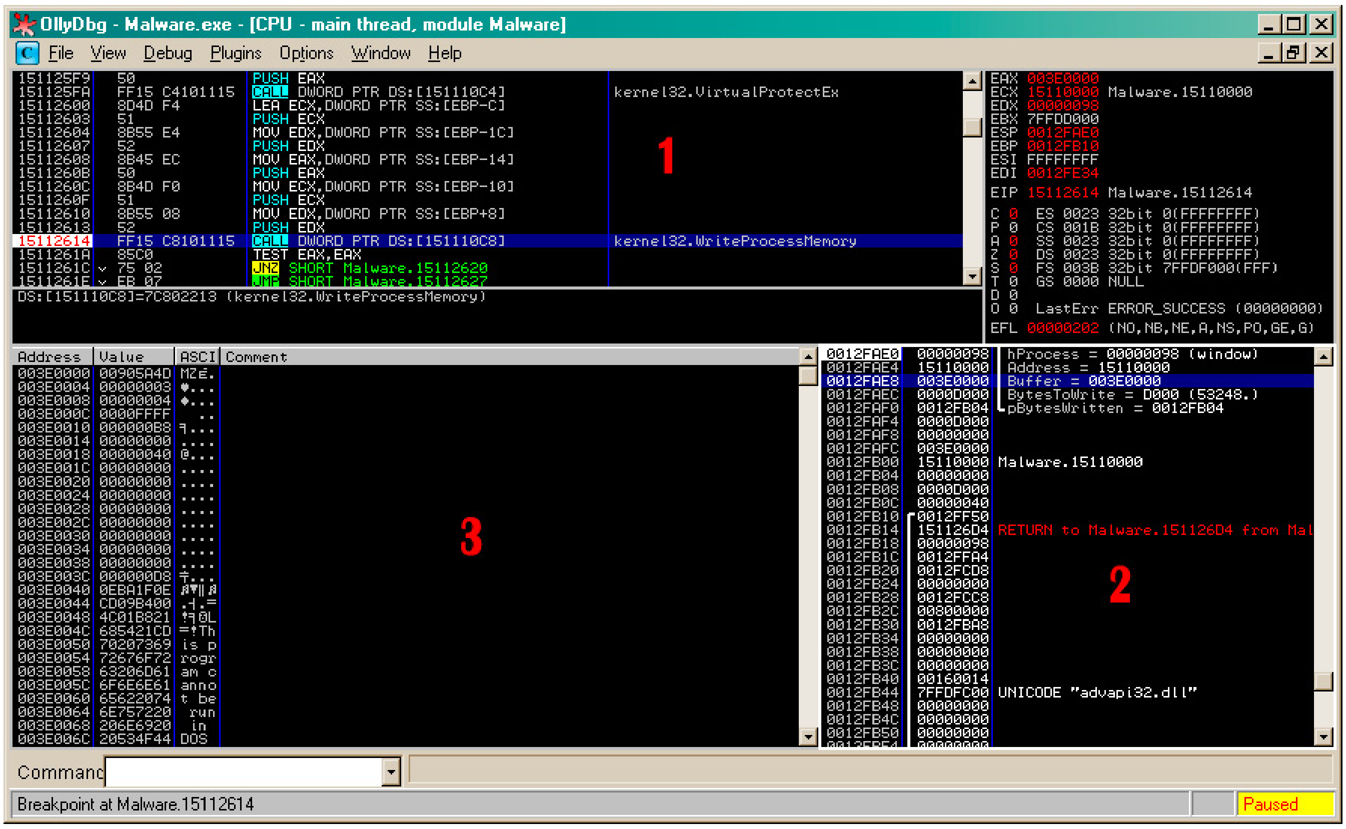
Extracting the Code

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 ⌄ 75 02            JNZ SHORT Malware.15112620
1511261E ⌄ EB 07            JMP SHORT Malware.15112627
15112620   C745 E8 010000   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 ⌄ EB 07            JMP SHORT Malware.15112627
DS:[151110C8]=7C802213 (kernel32.WriteProcessMemory)
```

```
EAX 003E0000
ECX 15110000 Malware.15110000
EDX 00000098
EBX 7FFDD000
ESP 0012FAE0
EBP 0012FB10
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    ASCI 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  0012FB04  └pBytesWritten = 0012FB04
0012FAF4  0000D000
0012FAF8  00000000
0012FAFC  003E0000
0012FB00  15110000  Malware.15110000
0012FB04  00000000
0012FB08  0000D000
0012FB0C  00000040
0012FB10 ┌0012FF50
0012FB14 │151126D4  RETURN to Malware.151126D4 from Mal
0012FB18 │00000098
0012FB1C │0012FFA4
0012FB20 │0012FCD8
0012FB24 │00000000
0012FB28 │0012FCC8
0012FB2C │00800000
0012FB30 │0012FBA8
0012FB34 │00000000
0012FB38 │00000000
0012FB3C │00000000
0012FB40 │00160014
0012FB44 │7FFDFC00  UNICODE "advapi32.dll"
0012FB48 │00000000
0012FB4C │00000000
0012FB50 │00000000
```

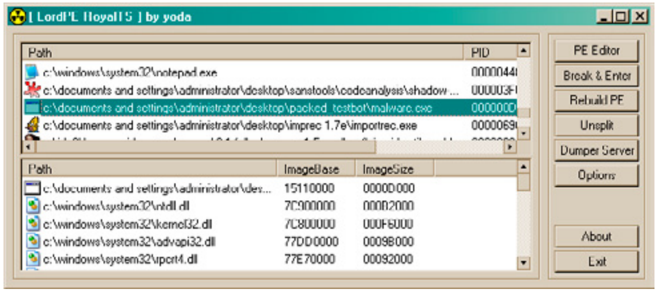
Breakpoint at Malware.15112614

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.

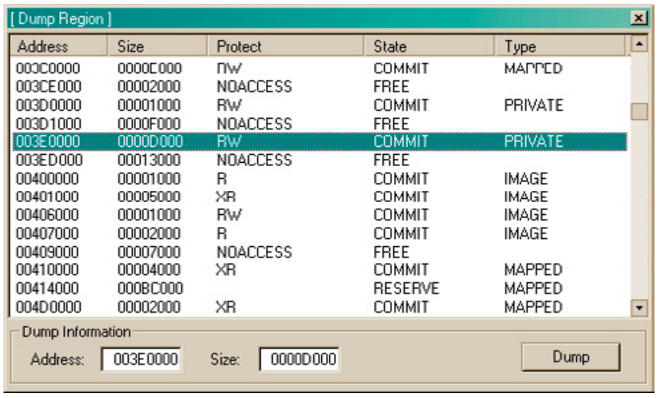
Using the above information, we can successfully extract the obfuscated code in one of two ways, using an application to extract the data and extracting it manually.



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_testbot\malware.exe	000000D
c:\documents and settings\administrator\desktop\imprec 1.7e\importrec.exe	0000069

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.

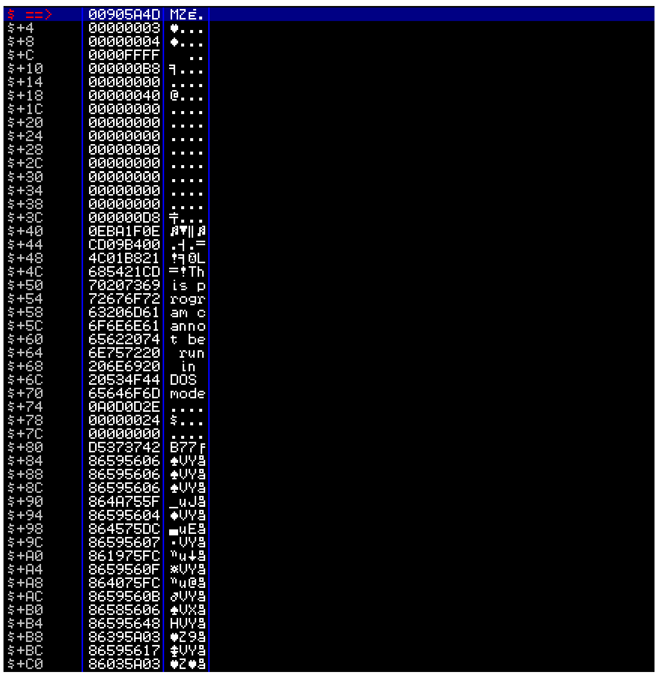
When we find our process, we right click it and select the "Dump Region" option. Using the dump region interface, we scroll through all of the memory regions belonging to the file and find the one that correlates to the buffer memory address we observed previously.

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

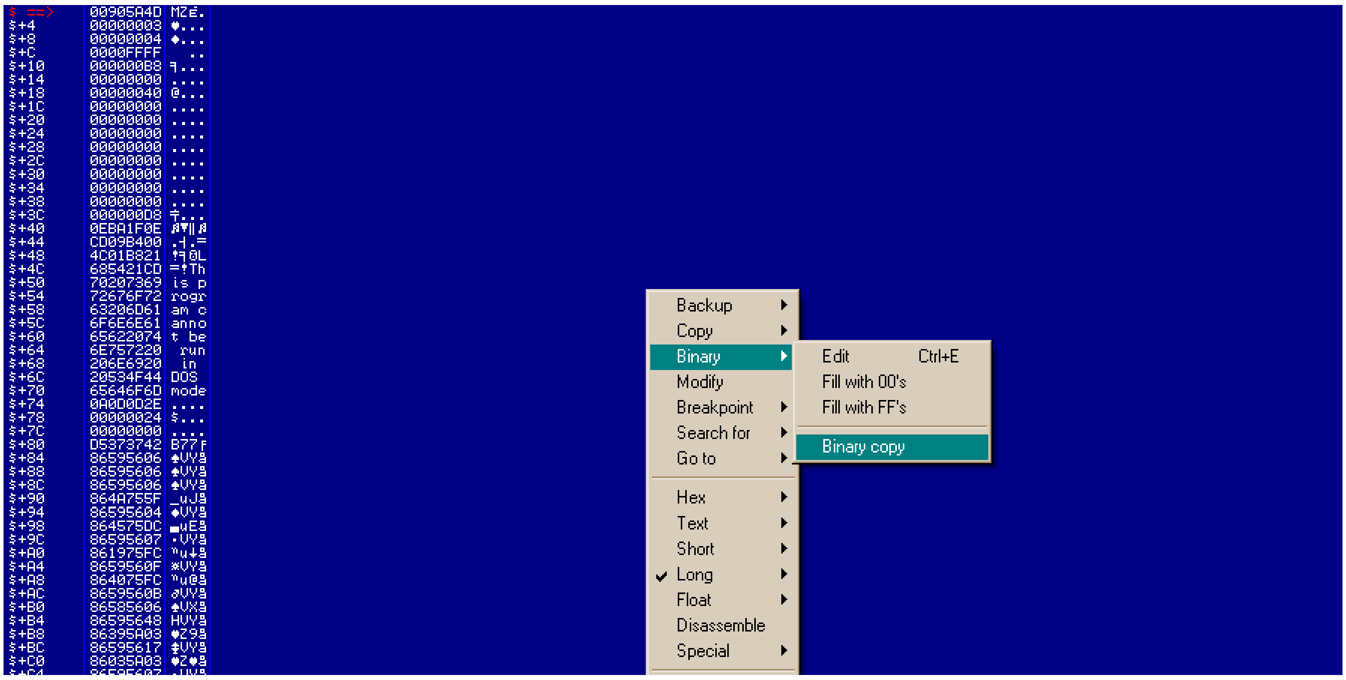
While rare, there might be an occasion when you cannot use LordPE to extract code from memory. This might be due to memory locked by the binary using it. In any case, there is a way around this problem and it is as simple as 'cut and paste'.

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     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     D5373742 B77╒
$+84     86595606 ♠VYå
$+88     86595606 ♠VYå
$+8C     86595606 ♠VYå
$+90     864A755F _uJå
$+94     86595604 ♦VYå
$+98     864575DC ▄uEå
$+9C     86595607 •VYå
$+A0     861975FC ⁿu↓å
$+A4     8659560F ☼VYå
$+A8     864075FC ⁿu@å
$+AC     8659560B ♂VYå
$+B0     86585606 ♠VXå
$+B4     86595648 HVYå
$+B8     86395A03 ♥Z9å
$+BC     86595617 ↨VYå
$+C0     86035A03 ♥Z♥å
```

Figure 21. *OllyDbg dump window using address offsets*



```
$ ==>    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     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     D5373742 B77╒
$+84     86595606 ♠VYå
$+88     86595606 ♠VYå
$+8C     86595606 ♠VYå
$+90     864A755F _uJå
$+94     86595604 ♦VYå
$+98     864575DC ▄uEå
$+9C     86595607 •VYå
$+A0     861975FC ⁿu↓å
$+A4     8659560F ☼VYå
$+A8     864075FC ⁿu@å
$+AC     8659560B ♂VYå
$+B0     86585606 ♠VXå
$+B4     86595648 HVYå
$+B8     86395A03 ♥Z9å
$+BC     86595617 ↨VYå
$+C0     86035A03 ♥Z♥å
```

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

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

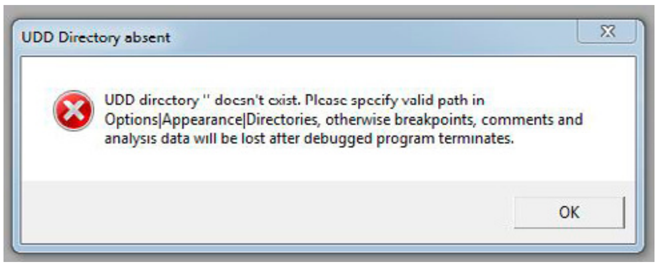
I used Visual Studio 2010 to compile the "reverse me" sample, if you do not have it installed on your analysis machine you will require the following DLLs to run the binary: *http://www.microsoft.com/en-us/download/details.aspx?id=5555*.

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 ☺.

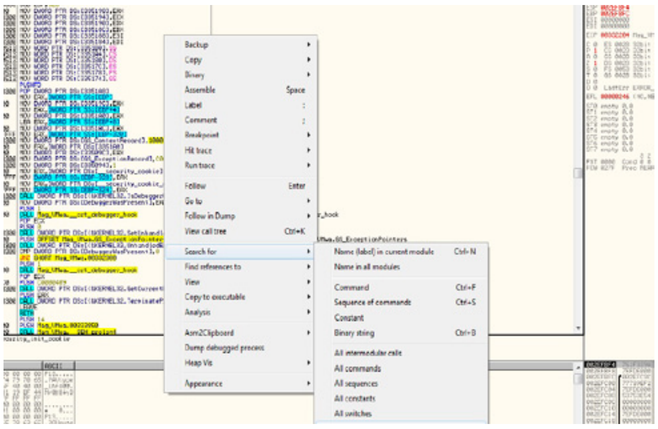


Figure 2. *Find referenced strings*

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
                  directory
   _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);
}
```

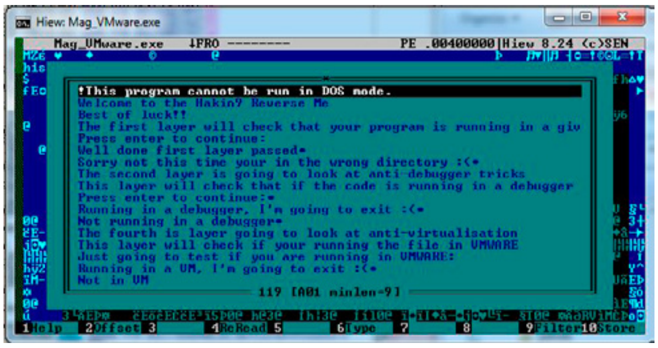
So, this basic trick can be effective against simple dynamic analysis. We will see later how to obfuscate strings which would make this technique even harder to detect by hiding the word "Temp."

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.

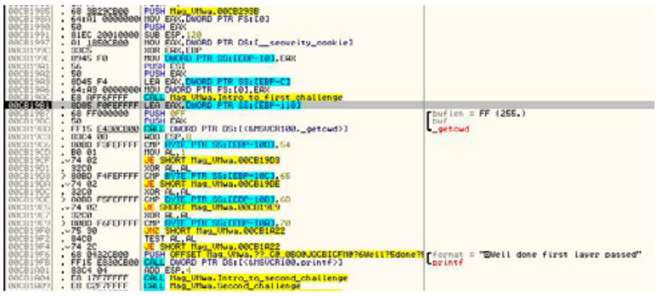
The corresponding assembler code as produced by Ollydbg is in Figure 4. As this may be your first time seeing assembler we will try and walk you through the code. The first point to identify is the call to `_getcwd`, this will get the current working directory. The next few lines compare the values in 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


Hiew: Mag_VMware.exe
Mag_VMware.exe ↓FRO -------- PE .00400000 Hiew 8.24 (c)SEN
1Help 2Offset 3 4ReRead 5 6Type 7 8 9Filter10Store

```
!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-virtualisation
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
                         119 [A01 minlen-9]
```

Figure 3. *Strings as seen in Hiew32*



```
00CB1985  . 68 3B29CB00     PUSH Mag_VMwa.00CB293B
00CB198A  . 64:A1 00000000  MOV EAX,DWORD PTR FS:[0]
00CB1990  . 50              PUSH EAX
00CB1991  . 81EC 20010000   SUB ESP,120
00CB1997  . A1 1850CB00     MOV EAX,DWORD PTR DS:[__security_cookie]
00CB199C  . 33C5            XOR EAX,EBP
00CB199E  . 8945 F0         MOV DWORD PTR SS:[EBP-10],EAX
00CB19A1  . 56              PUSH ESI
00CB19A2  . 50              PUSH EAX
00CB19A3  . 8D45 F4         LEA EAX,DWORD PTR SS:[EBP-C]
00CB19A6  . 64:A3 00000000  MOV DWORD PTR FS:[0],EAX
00CB19AC  . E8 AFF6FFFF     CALL Mag_VMwa.Intro_to_first_challenge
00CB19B1  . 8D85 F0FEFFFF   LEA EAX,DWORD PTR SS:[EBP-110]
00CB19B7  . 68 FF000000     PUSH 0FF                                   [buflen = FF (255.)
00CB19BC  . 50              PUSH EAX                                   |buf
00CB19BD  . FF15 E430CB00   CALL DWORD PTR DS:[<&MSVCR100._getcwd>]    [_getcwd
00CB19C3  . 83C4 08         ADD ESP,8
00CB19C6  . 80BD F0FEFFFF   CMP BYTE PTR SS:[EBP-110],54
00CB19CD  . B0 01           MOV AL,1
00CB19CF  .~74 02           JE SHORT Mag_VMwa.00CB19D3
00CB19D1  . 32C0            XOR AL,AL
00CB19D3  > 80BD F4FEFFFF   CMP BYTE PTR SS:[EBP-10C],65
00CB19DA  .~74 02           JE SHORT Mag_VMwa.00CB19DE
00CB19DC  . 32C0            XOR AL,AL
00CB19DE  > 80BD F5FEFFFF   CMP BYTE PTR SS:[EBP-10B],6D
00CB19E5  .~74 02           JE SHORT Mag_VMwa.00CB19E9
00CB19E7  . 32C0            XOR AL,AL
00CB19E9  > 80BD F6FEFFFF   CMP BYTE PTR SS:[EBP-10A],70
00CB19F0  .~75 30           JNZ SHORT Mag_VMwa.00CB1A22
00CB19F2  . 84C0            TEST AL,AL
00CB19F4  .~74 2C           JE SHORT Mag_VMwa.00CB1A22
00CB19F6  . 68 0432CB00     PUSH OFFSET Mag_VMwa.??_C@_0BO@JOCBICFH@?6Well?5done?$  [format = "Well done first layer passed"
00CB19FB  . FF15 E830CB00   CALL DWORD PTR DS:[<&MSVCR100.printf>]     [printf
00CB1A01  . 83C4 04         ADD ESP,4
00CB1A04  . E8 17F7FFFF     CALL Mag_VMwa.Intro_to_second_challenge
00CB1A09  . E8 C5F7FFFF     CALL Mag_VMwa.Second_challenge
```

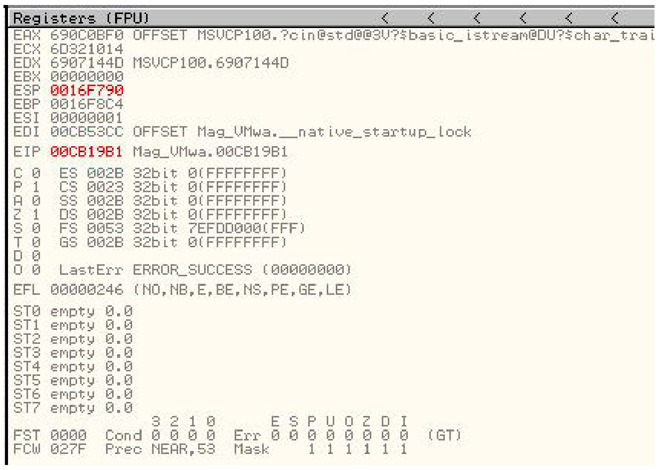
Figure 4. *Layer 1 Directory Detection, Assemble view*

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



```
Registers (FPU)                  <    <    <    <    <    <
EAX 690C0BF0 OFFSET MSVCP100.?cin@std@@3V?$basic_istream@DU?$char_trai
ECX 6D321014
EDX 6907144D MSVCP100.6907144D
EBX 00000000
ESP 0016F790
EBP 0016F8C4
ESI 00000001
EDI 00CB53CC OFFSET Mag_VMwa.__native_startup_lock
EIP 00CB19B1 Mag_VMwa.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*

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.

Layer 3 Obfuscation of strings and hiding APIs

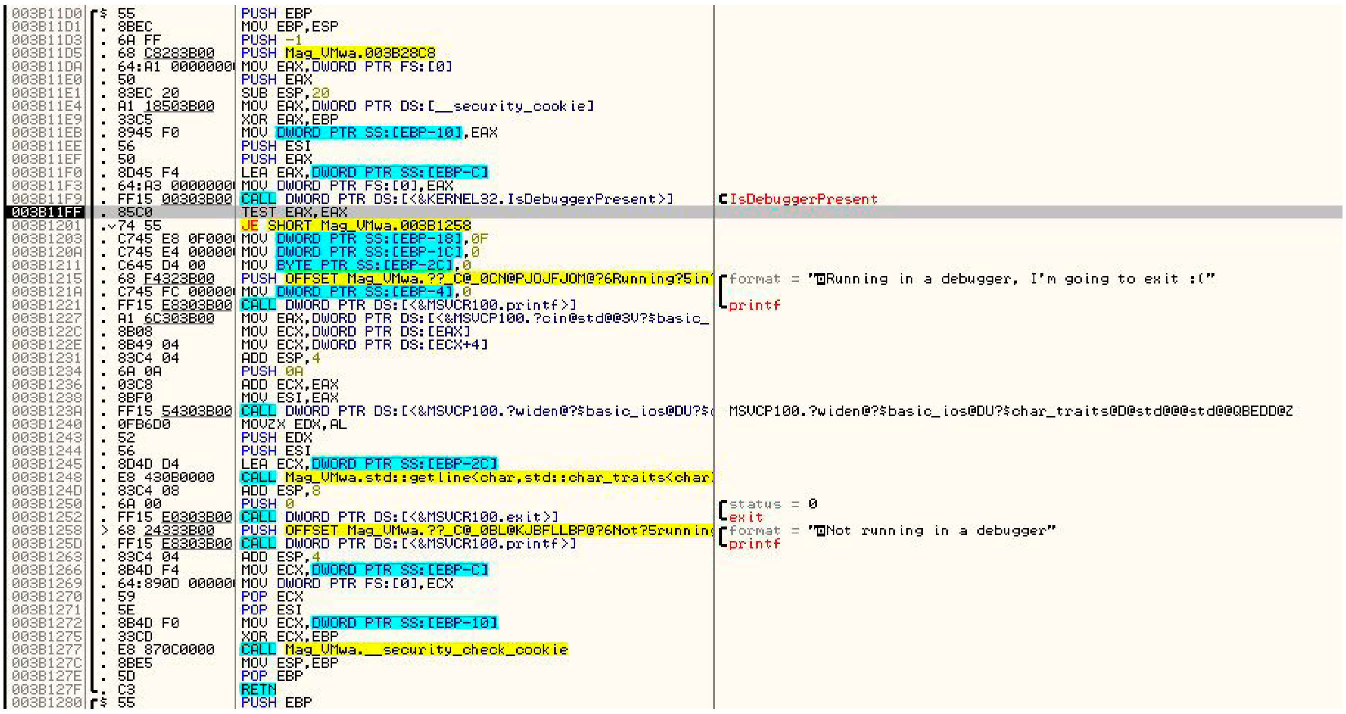
I am going to take these two topics together as they are intrinsically linked. Windows executable files

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”);

   }
}
```



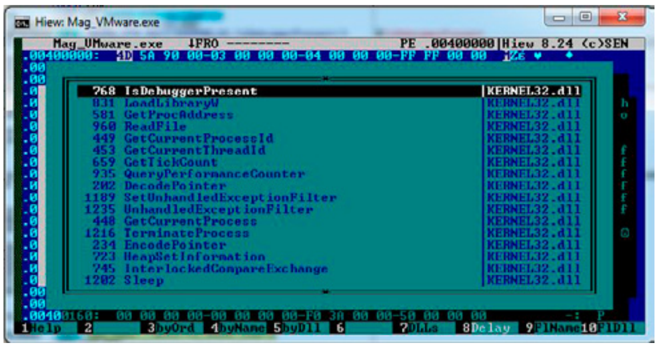
```
003B11D0 ┌$ 55              PUSH EBP
003B11D1 │. 8BEC            MOV EBP,ESP
003B11D3 │. 6A FF           PUSH -1
003B11D5 │. 68 C8283B00     PUSH Mag_VMwa.003B28C8
003B11DA │. 64:A1 0000000   MOV EAX,DWORD PTR FS:[0]
003B11E0 │. 50              PUSH EAX
003B11E1 │. 83EC 20         SUB ESP,20
003B11E4 │. A1 18503B00     MOV EAX,DWORD PTR DS:[__security_cookie]
003B11E9 │. 33C5            XOR EAX,EBP
003B11EB │. 8945 F0         MOV DWORD PTR SS:[EBP-10],EAX
003B11EE │. 56              PUSH ESI
003B11EF │. 50              PUSH EAX
003B11F0 │. 8D45 F4         LEA EAX,DWORD PTR SS:[EBP-C]
003B11F3 │. 64:A3 0000000   MOV DWORD PTR FS:[0],EAX
003B11F9 │. FF15 00303B00   CALL DWORD PTR DS:[<&KERNEL32.IsDebuggerPresent>]        [IsDebuggerPresent
003B11FF │. 85C0            TEST EAX,EAX
003B1201 │.v74 55           JE SHORT Mag_VMwa.003B1258
003B1203 │. C745 E8 0F000   MOV DWORD PTR SS:[EBP-18],0F
003B120A │. C745 E4 00000   MOV DWORD PTR SS:[EBP-1C],0
003B1211 │. C645 D4 00      MOV BYTE PTR SS:[EBP-2C],0
003B1215 │. 68 F4323B00     PUSH OFFSET Mag_VMwa.??_C@_0CN@PJOJFJOM@?6Running?5in  ┌format = "□Running in a debugger, I'm going to exit :("
003B121A │. C745 FC 00000   MOV DWORD PTR SS:[EBP-4],0                            │
003B1221 │. FF15 E8303B00   CALL DWORD PTR DS:[<&MSVCR100.printf>]                └printf
003B1227 │. A1 6C303B00     MOV EAX,DWORD PTR DS:[<&MSVCP100.?cin@std@@3V?$basic_
003B122C │. 8B08            MOV ECX,DWORD PTR DS:[EAX]
003B122E │. 8B49 04         MOV ECX,DWORD PTR DS:[ECX+4]
003B1231 │. 83C4 04         ADD ESP,4
003B1234 │. 6A 0A           PUSH 0A
003B1236 │. 03C8            ADD ECX,EAX
003B1238 │. 8BF0            MOV ESI,EAX
003B123A │. FF15 54303B00   CALL DWORD PTR DS:[<&MSVCP100.?widen@?$basic_ios@DU?$   MSVCP100.?widen@?$basic_ios@DU?$char_traits@D@std@@@std@@QBEDD@Z
003B1240 │. 0FB6D0          MOVZX EDX,AL
003B1243 │. 52              PUSH EDX
003B1244 │. 56              PUSH ESI
003B1245 │. 8D4D D4         LEA ECX,DWORD PTR SS:[EBP-2C]
003B1248 │. E8 430B0000     CALL Mag_VMwa.std::getline<char,std::char_traits<char>
003B124D │. 83C4 08         ADD ESP,8
003B1250 │. 6A 00           PUSH 0                                                 ┌status = 0
003B1252 │. FF15 E0303B00   CALL DWORD PTR DS:[<&MSVCR100.exit>]                   └exit
003B1258 │> 68 24333B00     PUSH OFFSET Mag_VMwa.??_C@_0BL@KJBFLLBP@?6Not?5runnin   ┌format = "□Not running in a debugger"
003B125D │. FF15 E8303B00   CALL DWORD PTR DS:[<&MSVCR100.printf>]                 └printf
003B1263 │. 83C4 04         ADD ESP,4
003B1266 │. 8B4D F4         MOV ECX,DWORD PTR SS:[EBP-C]
003B1269 │. 64:890D 00000   MOV DWORD PTR FS:[0],ECX
003B1270 │. 59              POP ECX
003B1271 │. 5E              POP ESI
003B1272 │. 8B4D F0         MOV ECX,DWORD PTR SS:[EBP-10]
003B1275 │. 33CD            XOR ECX,EBP
003B1277 │. E8 870C0000     CALL Mag_VMwa.__security_check_cookie
003B127C │. 8BE5            MOV ESP,EBP
003B127E │. 5D              POP EBP
003B127F └. C3              RETN
003B1280 ┌$ 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.

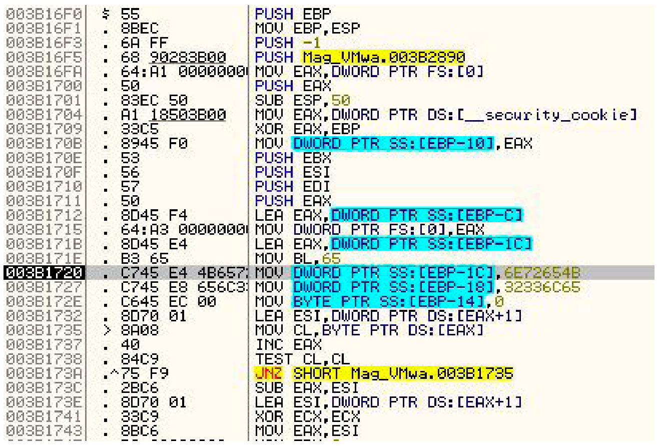
Layer 3:GetProcAdress

'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



```
  Mag_VMware.exe    ↓FRO --------              PE .00400000|Hiew 8.24 (c)SEN
.00400000:  4D 5A 90 00-03 00 00 00-04 00 00 00-FF FF 00 00  MZE
   768 IsDebuggerPresent                          |KERNEL32.dll
   831 LoadLibraryW                               |KERNEL32.dll
   581 GetProcAddress                             |KERNEL32.dll
   960 ReadFile                                   |KERNEL32.dll
   449 GetCurrentProcessId                        |KERNEL32.dll
   453 GetCurrentThreadId                         |KERNEL32.dll
   659 GetTickCount                               |KERNEL32.dll
   935 QueryPerformanceCounter                    |KERNEL32.dll
   202 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
  1202 Sleep                                      |KERNEL32.dll
.00400160:  00 00 00 00-00 00 00 00-F0 3A 00 00-50 00 00 00
1Help 2 3byOrd 4byName 5byDll 6 7DLLs 8Delay 9FlName10FlDll
```

Figure 7. *Import Table*



```
003B16F0  $ 55             PUSH EBP
003B16F1  . 8BEC           MOV EBP,ESP
003B16F3  . 6A FF          PUSH -1
003B16F5  . 68 90283B00    PUSH Mag_VMwa.003B2890
003B16FA  . 64:A1 0000000( MOV EAX,DWORD PTR FS:[0]
003B1700  . 50             PUSH EAX
003B1701  . 83EC 50        SUB ESP,50
003B1704  . A1 18503B00    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 0000000( MOV DWORD PTR FS:[0],EAX
003B171B  . 8D45 E4        LEA EAX,DWORD PTR SS:[EBP-1C]
003B171E  . B3 65          MOV BL,65
003B1720  . C745 E4 4B657: MOV DWORD PTR SS:[EBP-1C],6E72654B
003B1727  . C745 E8 656C3: 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,
                 newsize, 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;
}
```

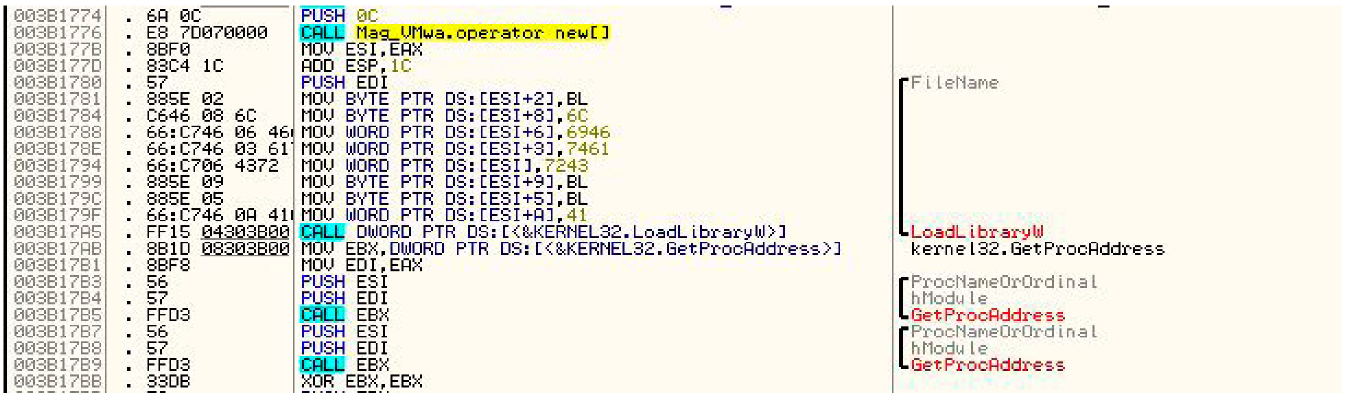
The block of 'mov' instructions builds the string. As you can see, it is much harder to pull out *CreateFileA* from this code. It is a very simple and effective obfuscation technique. The API name is built on the ESI register and then passed to *GetProcAddress*. So, a good option is to put a breakpoint 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  . 885E 05          MOV BYTE PTR DS:[ESI+5],BL                                     │
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
```

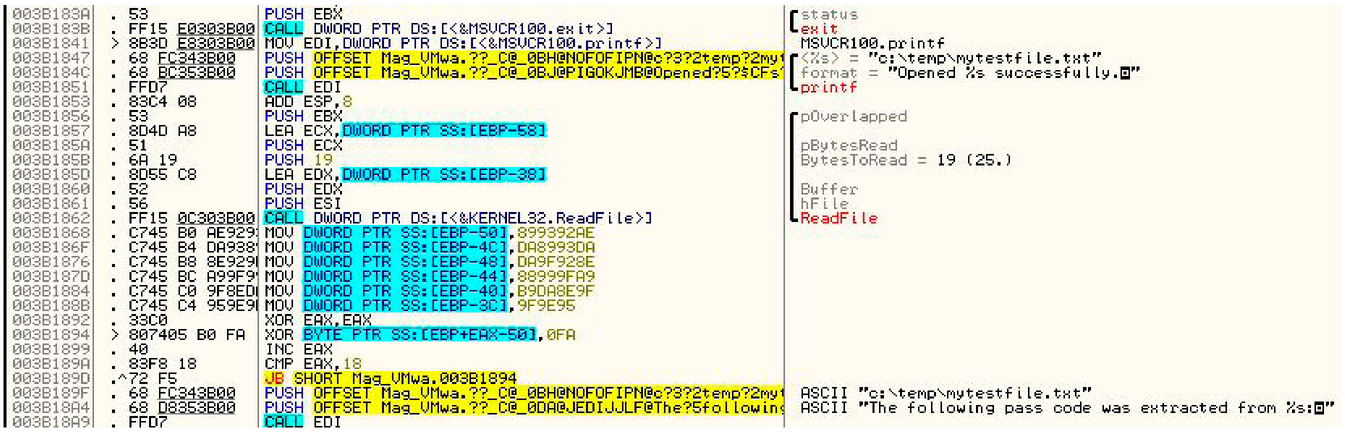
Figure 9. *Building CreateFileA as a Character Array*

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@NOFOFIPN@c?3?2temp?2myt <%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 AE929: MOV DWORD PTR SS:[EBP-50],899392AE
003B186F  . C745 B4 DA938' MOV DWORD PTR SS:[EBP-4C],DA8993DA
003B1876  . C745 B8 8E929| MOV DWORD PTR SS:[EBP-48],DA9F928E
003B187D  . C745 BC A99F9' MOV DWORD PTR SS:[EBP-44],88999FA9
003B1884  . C745 C0 9F8ED| MOV DWORD PTR SS:[EBP-40],B9DA8E9F
003B188B  . C745 C4 959E9| 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@NOFOFIPN@c?3?2temp?2myt ASCII "c:\temp\mytestfile.txt"
003B18A4  . 68 D8353B00    PUSH OFFSET Mag_VMwa.??_C@_0DA@JEDIJJLF@The?5followin  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
                  running 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 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

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 Ace VideoConverter.com. He also works with several other software companies and specialises in windows software development. You can contact him directly at Lorenzo@xetoware.com.

MONITOR STRONY

Innovative e-services for websites monitoring

SEOmonitor
SEO website monitor

SPEEDmonitor
website loading speed monitor

CONTENTmonitor
website content language correctness monitor

www.monitorstrony.pl

FUNDS FOR INNOVATIONS

INNOVATIVE ECONOMY
NATIONAL COHESION STRATEGY

EUROPEAN UNION
EUROPEAN REGIONAL
DEVELOPMENT FUND

PROJECT CO-FINANCED BY THE EUROPEAN REGIONAL DEVELOPMENT FUND UNDER THE OPERATIONAL PROGRAMME INNOVATIVE ECONOMY

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.

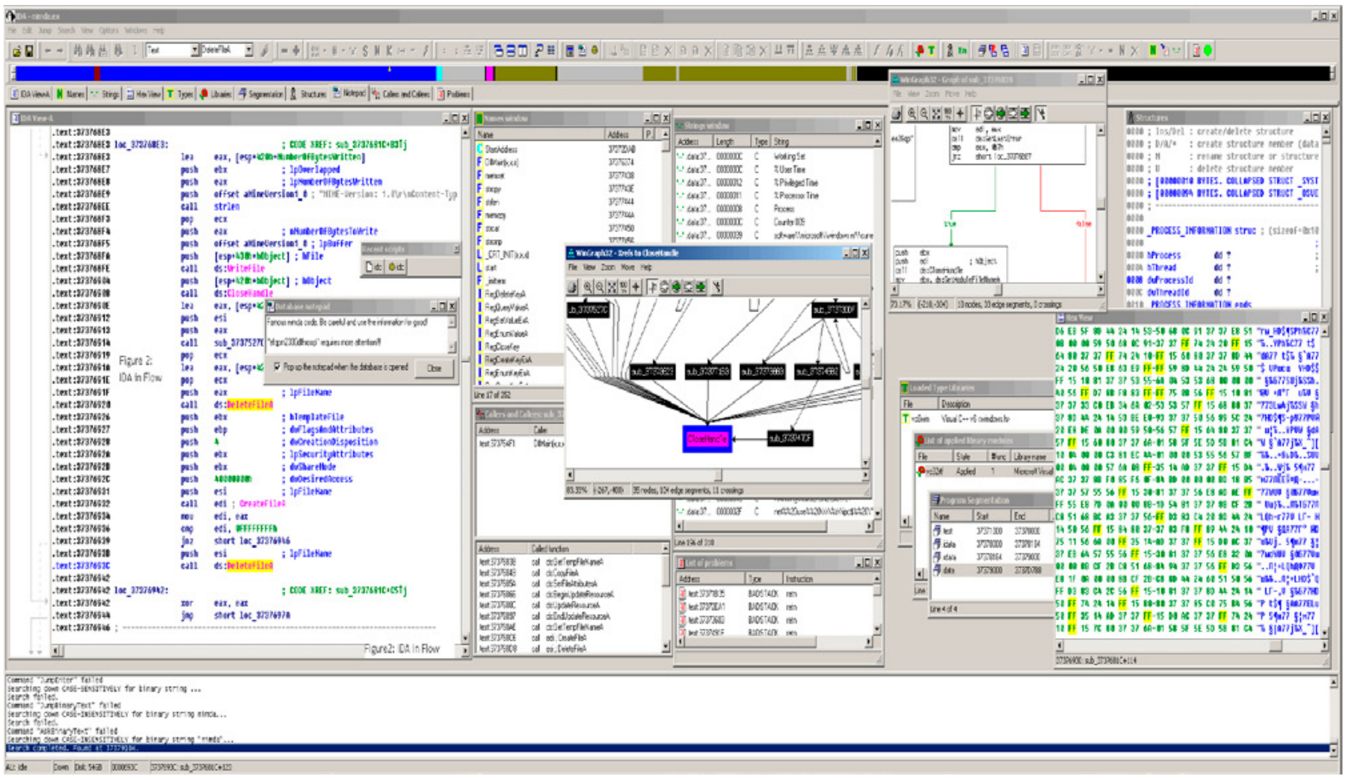
Pre-requisite in Software Reverse Engineering

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*

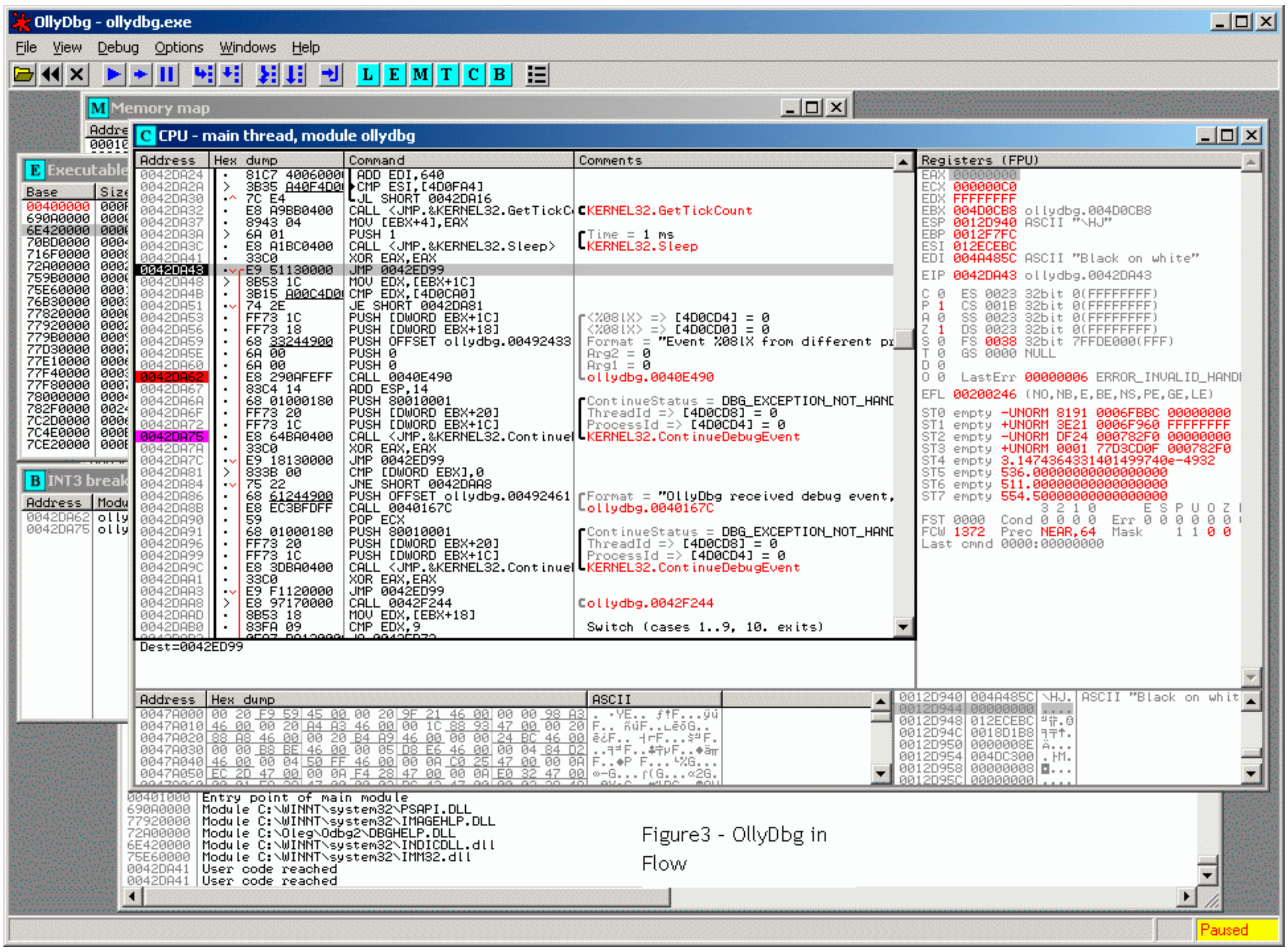


```
.text:373768E3
.text:373768E3 loc_373768E3:                          ; CODE XREF: sub_3737681C+B3↑j
.text:373768E3            lea     eax, [esp+430h+NumberOfBytesWritten]
.text:373768E7            push    ebx             ; lpOverlapped
.text:373768E8            push    eax             ; lpNumberOfBytesWritten
.text:373768E9            push    offset aMimeVersion1_0 ; "MIME-Version: 1.0\r\nContent-Typ
.text:373768EE            call    strlen
.text:373768F3            pop     ecx
.text:373768F4            push    eax             ; nNumberOfBytesToWrite
.text:373768F5            push    offset aMimeVersion1_0 ; lpBuffer
.text:373768FA            push    [esp+430h+hObject] ; hFile
.text:373768FE            call    ds:WriteFile
.text:37376904            push    [esp+420h+hObject] ; hObject
.text:37376908            call    ds:CloseHandle
.text:3737690E            lea     eax, [esp+4C
.text:37376912            push    esi
.text:37376913            push    eax
.text:37376914            call    sub_3737527C
.text:37376919            pop     ecx
.text:3737691A            lea     eax, [esp+4C
.text:3737691E            pop     ecx
.text:3737691F            push    eax             ; lpFileName
.text:37376920            call    ds:DeleteFileA
.text:37376926            push    ebx             ; hTemplateFile
.text:37376927            push    ebp             ; dwFlagsAndAttributes
.text:37376928            push    4               ; dwCreationDisposition
.text:3737692A            push    ebx             ; lpSecurityAttributes
.text:3737692B            push    ebx             ; dwShareMode
.text:3737692C            push    40000000h       ; dwDesiredAccess
.text:37376931            push    esi             ; lpFileName
.text:37376932            call    edi ; CreateFileA
.text:37376934            mov     edi, eax
.text:37376936            cmp     edi, 0FFFFFFFFh
.text:37376939            jnz     short loc_37376946
.text:3737693B            push    esi             ; lpFileName
.text:3737693C            call    ds:DeleteFileA
.text:37376942
.text:37376942 loc_37376942:                          ; CODE XREF: sub_3737681C+C5↑j
.text:37376942            xor     eax, eax
.text:37376944            jmp     short loc_37376970
.text:37376946 ; ---------------------------------------------------------------
```

Database notepad: Famous nimda code. Be careful and use the information for good!

```
Command "JumpEnter" failed
Searching down CASE-SENSITIVELY for binary string ...
Search failed.
Command "JumpBinaryText" failed
Searching down CASE-INSENSITIVELY for binary string nimda...
Search failed.
Command "AskBinaryText" failed
Searching down CASE-INSENSITIVELY for binary string "nimda"...
Search completed. Found at 3737910A.
```

Figure 2. *IDA in Flow*



```
Address   Hex dump          Command                                  Comments
0042DA24  81C7 40060000     ADD EDI,640
0042DA2A  3B35 A40F4D00     CMP ESI,[4D0FA4]
0042DA30  7C E4             JL SHORT 0042DA16
0042DA32  E8 A9BB0400       CALL <JMP.&KERNEL32.GetTickC             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 A00C4D00     CMP EDX,[4D0CA0]
0042DA51  74 2E             JE SHORT 0042DA81
0042DA53  FF73 1C           PUSH [DWORD EBX+1C]                      <%08lX> => [4D0CD4] = 0
0042DA56  FF73 18           PUSH [DWORD EBX+18]                      <%08lX> => [4D0CD0] = 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 290AFEFF       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 64BA0400       CALL <JMP.&KERNEL32.ContinueI            KERNEL32.ContinueDebugEvent
0042DA7A  33C0              XOR EAX,EAX
0042DA7C  E9 18130000       JMP 0042ED99
0042DA81  833B 00           CMP [DWORD EBX],0
0042DA84  75 22             JNE SHORT 0042DAA8
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.ContinueI            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)
EAX 00000000
ECX 000000C0
EDX FFFFFFFF
EBX 004D0CB8 ollydbg.004D0CB8
ESP 0012D940 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_HANDL

EFL 00200246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty -UNORM 8191 0006FBBC 00000000
ST1 empty +UNORM 3E21 0006F960 FFFFFFFF
ST2 empty -UNORM DF24 000782F0 00000000
ST3 empty +UNORM 0001 77D3CD0F 000782F0
ST4 empty 3.1474364331401499740e-4932
ST5 empty 536.00000000000000000
ST6 empty 511.00000000000000000
ST7 empty 554.50000000000000000
               3 2 1 0      E S P U O Z
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0
FCW 1372  Prec NEAR,64  Mask    1 1 0 0
Last cmnd 0000:00000000
```

```
Address   Hex dump                                         ASCII
0047A000  00 20 F9 59 45 00 00 20 9F 21 46 00 00 00 98 A3  . ·YE.. ƒ!F...ÿú
0047A010  46 00 00 20 A4 A3 46 00 00 1C 88 93 47 00 00 20  F.. ñúF..∟êôG..
0047A020  88 A8 46 00 00 20 B4 A9 46 00 00 00 24 BC 46 00  ê¿F.. ┤⌐F...$╝F.
0047A030  00 00 B8 BE 46 00 00 05 D8 E6 46 00 00 04 84 D2  ..╕╛F..♣╪µF..♦ä╥
0047A040  46 00 00 04 50 FF 46 00 00 0A C0 25 47 00 00 0A  F..♦P F...└%G...
0047A050  EC 2D 47 00 00 0A F4 28 47 00 00 0A E0 32 47 00  ∞-G...⌠(G...α2G.
```

```
0012D940  004A485C  \HJ.  ASCII "Black on whit
0012D944  00000000  ....
0012D948  012ECEBC  ╝╬.☺
0012D94C  0018D1B8  ╕╤↑.
0012D950  0000008E  Ä...
0012D954  004DC300  .├M.
0012D958  00000008  ◘...
0012D95C  00000000  ....
```

```
00401000  Entry point of main module
690A0000  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
75E60000  Module C:\WINNT\system32\IMM32.dll
0042DA41  User code reached
0042DA41  User code reached
```

OllyDbg - ollydbg.exe
File View Debug Options Windows Help
L E M T C B
Memory map
CPU - main thread, module ollydbg
Executable
INT3 break
Figure3 - OllyDbg in Flow
Paused

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

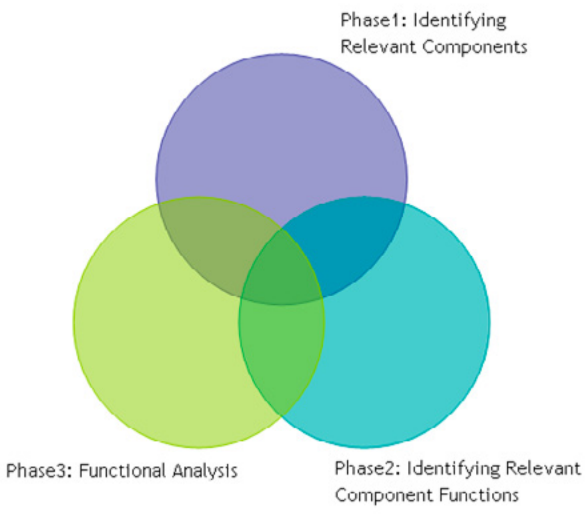
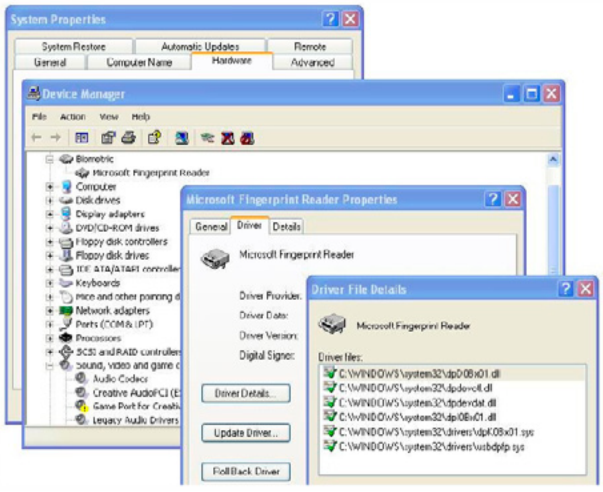

Phase1: Identifying Relevant Components
Phase3: Functional Analysis
Phase2: Identifying Relevant Component Functions

Figure 4. *High Level Reversing Methodology*



C:\WINDOWS\system32\dpD08x01.dll
C:\WINDOWS\system32\dpdevctl.dll
C:\WINDOWS\system32\dpdevdat.dll
C:\WINDOWS\system32\dpI08x01.dll
C:\WINDOWS\system32\drivers\dpK08x01.sys
C:\WINDOWS\system32\drivers\usbdpfp.sys

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.

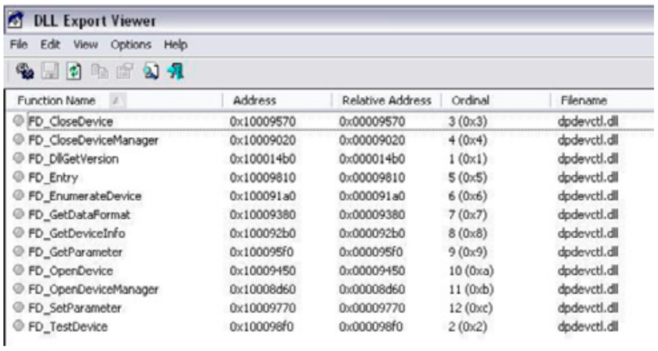
IDA Pro can also be used to dig out this level of information. As you can see, the names of the functions, their addresses in memory and the files they are coded in. We can further reverse the function to get the actual code, but I am limiting this Phase to this level. *You should try your luck after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

Note

Keep documenting what you have so far obtained.

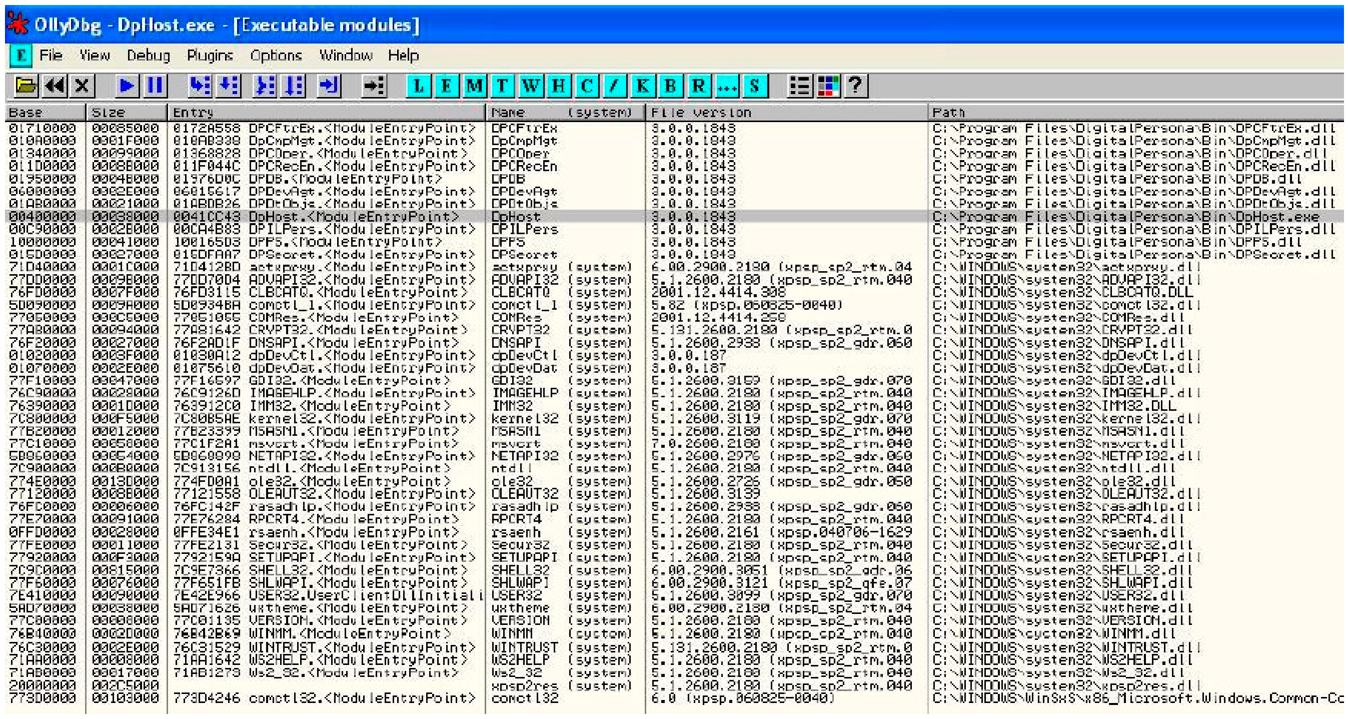
Phase 3: High Level Functional Analysis

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



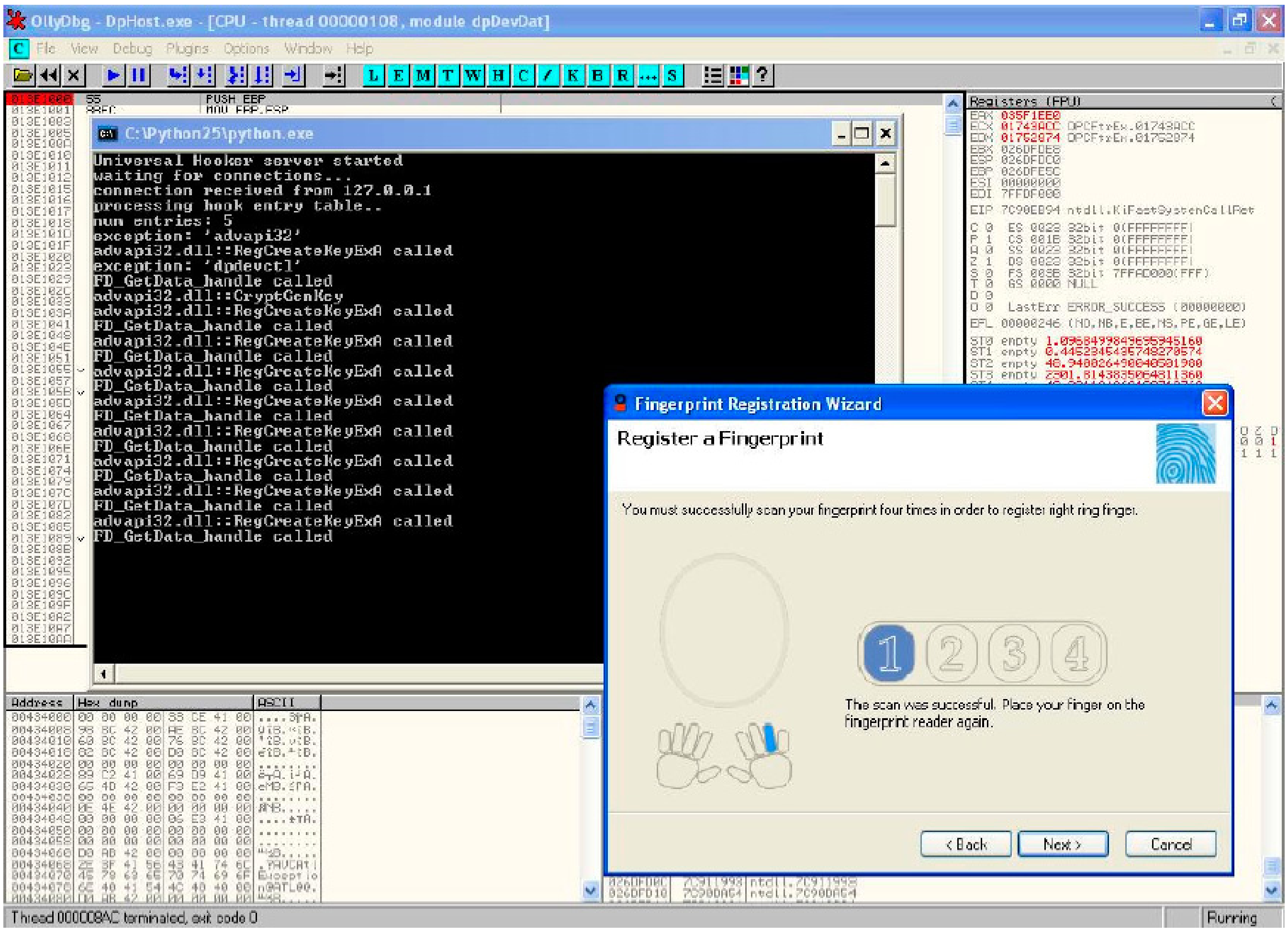
Function Name	Address	Relative Address	Ordinal	Filename
FD_CloseDevice	0x10009570	0x00009570	3 (0x3)	dpdevctl.dll
FD_CloseDeviceManager	0x10009020	0x00009020	4 (0x4)	dpdevctl.dll
FD_DllGetVersion	0x100014b0	0x000014b0	1 (0x1)	dpdevctl.dll
FD_Entry	0x10009810	0x00009810	5 (0x5)	dpdevctl.dll
FD_EnumerateDevice	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]
E 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	00085000	0172A558 DPCFtrEx.<ModuleEntryPoint>	DPCFtrEx	3.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPCFtrEx.dll
010A0000	0001F000	010AB330 DpCmpMgt.<ModuleEntryPoint>	DpCmpMgt	3.0.0.1843	C:\Program Files\DigitalPersona\Bin\DpCmpMgt.dll
01340000	00099000	01368828 DPCOper.<ModuleEntryPoint>	DPCOper	3.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPCOper.dll
011D0000	0008B000	011F044C DPCRecEn.<ModuleEntryPoint>	DPCRecEn	3.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPCRecEn.dll
01950000	0004B000	01976D0C DPDB.<ModuleEntryPoint>	DPDB	3.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDB.dll
06000000	0002E000	06015617 DPDevAgt.<ModuleEntryPoint>	DPDevAgt	3.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevAgt.dll
01AB0000	00021000	01ABDB26 DPDtObjs.<ModuleEntryPoint>	DPDtObjs	3.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDtObjs.dll
00400000	00038000	0041CC43 DpHost.<ModuleEntryPoint>	DpHost	3.0.0.1843	C:\Program Files\DigitalPersona\Bin\DpHost.exe
00C90000	0002B000	00CA4B83 DPILPers.<ModuleEntryPoint>	DPILPers	3.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPILPers.dll
10000000	00041000	10016503 DPFS.<ModuleEntryPoint>	DPFS	3.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPFS.dll
015D0000	00027000	015DFAA7 DPSecret.<ModuleEntryPoint>	DPSecret	3.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPSecret.dll
71D40000	0001C000	71D412BD 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	76FD3115 CLBCATQ.<ModuleEntryPoint>	CLBCATQ (system)	2001.12.4414.308	C:\WINDOWS\system32\CLBCATQ.DLL
5D090000	0009A000	5D09346A comctl_1.<ModuleEntryPoint>	comctl_1 (system)	5.82 (xpsp.060825-0040)	C:\WINDOWS\system32\comctl32.dll
77050000	000C5000	77051055 COMRes.<ModuleEntryPoint>	COMRes (system)	2001.12.4414.258	C:\WINDOWS\system32\COMRes.dll
77A80000	00094000	77A81642 CRYPT32.<ModuleEntryPoint>	CRYPT32 (system)	5.131.2600.2180 (xpsp_sp2_rtm.0	C:\WINDOWS\system32\CRYPT32.dll
76F20000	00027000	76F2AD1F DNSAPI.<ModuleEntryPoint>	DNSAPI (system)	5.1.2600.2938 (xpsp_sp2_gdr.060	C:\WINDOWS\system32\DNSAPI.dll
01020000	0003F000	01030A12 dpDevCtl.<ModuleEntryPoint>	dpDevCtl (system)	3.0.0.187	C:\WINDOWS\system32\dpDevCtl.dll
01070000	0002E000	01075610 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	76C9126D IMAGEHLP.<ModuleEntryPoint>	IMAGEHLP (system)	5.1.2600.2180 (xpsp_sp2_rtm.040	C:\WINDOWS\system32\IMAGEHLP.dll
76390000	0001D000	763912C0 IMM32.<ModuleEntryPoint>	IMM32 (system)	5.1.2600.2180 (xpsp_sp2_rtm.040	C:\WINDOWS\system32\IMM32.DLL
7C800000	000F5000	7C80B5AE kernel32.<ModuleEntryPoint>	kernel32 (system)	5.1.2600.3119 (xpsp_sp2_gdr.070	C:\WINDOWS\system32\kernel32.dll
77B20000	00012000	77B23399 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	5B860098 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	77E76284 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	7792159A 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_qfe.07	C:\WINDOWS\system32\SHLWAPI.dll
7E410000	00090000	7E42E966 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	71AB1279 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-Cc

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



OllyDbg - DpHost.exe - [CPU - thread 00000108, module dpDevDat]
File View Debug Plugins Options Window Help
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
```

```
Registers (FPU)
EAX 035F1EE0
ECX 01743ACC DPCFtrEx.01743ACC
EDX 01752874 DPCFtrEx.01752874
EBX 026DFDE8
ESP 026DFDC0
EBP 026DFE5C
ESI 00000000
EDI 7FFDF000
EIP 7C90EB94 ntdll.KiFastSystemCallRet
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 7FFAD000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty 1.0968499849695945160
ST1 empty 0.4452345435749270574
ST2 empty 48.940026490040501900
ST3 empty 2301.8143835064811360
```

```
Address  Hex dump                 ASCII
00434000 00 00 00 00 33 CE 41 00  ....3‡A.
00434008 98 BC 42 00 AE BC 42 00  ÿ╝B.«╝B.
00434010 60 BC 42 00 76 BC 42 00  `╝B.v╝B.
00434018 82 BC 42 00 D0 BC 42 00  é╝B.╨╝B.
00434020 00 00 00 00 00 00 00 00  ........
00434028 89 C2 41 00 69 D9 41 00  ë┬A.i┘A.
00434030 65 4D 42 00 F3 E2 41 00  eMB.≤ΓA.
00434038 00 00 00 00 00 00 00 00  ........
00434040 0E 4E 42 00 00 00 00 00  ♫NB.....
00434048 00 00 00 00 06 E3 41 00  ....♠πA.
00434050 00 00 00 00 00 00 00 00  ........
00434058 00 00 00 00 00 00 00 00  ........
00434060 D0 AB 42 00 00 00 00 00  ╨½B.....
00434068 2E 3F 41 56 43 41 74 6C  .?AVCAtl
00434070 45 78 63 65 70 74 69 6F  Exceptio
00434078 6E 40 41 54 4C 40 40 00  n@ATL@@.
00434080 D0 AB 42 00 00 00 00 00  ╨½B.....
```

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 000008AC 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.

jscrambler
protect your code

Modern websites, which use Web 2.0 and AJAX, often generate HTML and JavaScript code on the fly. This means that standard static code analyzers cannot fully scan the source code and locate client-side JavaScript issues, since the source code itself does not yet include the entire HTML and JavaScript code.

We used a sample group of 675 websites, including all 500 of the Fortune 500 companies, plus 175 handpicked websites including IT security companies, web application security companies, social networking sites and other popular websites. "*Each application was tested for two main client-side JavaScript issues: DOM-based Cross-site scripting, and open redirects, a vulnerability which allows a malicious attacker to force the victim's browser to automatically redirect to a site he/she owns, and which can be used for Phishing purposes. Our research found that of the 675 websites analyzed, 98 (14.5 percent) were infested with DOM-based Cross site scripting and open redirects* (Figure 1).[1]

1 ftp://public.dhe.ibm.com/common/ssi/ecm/en/raw14252usen/RAW14252USEN.PDF

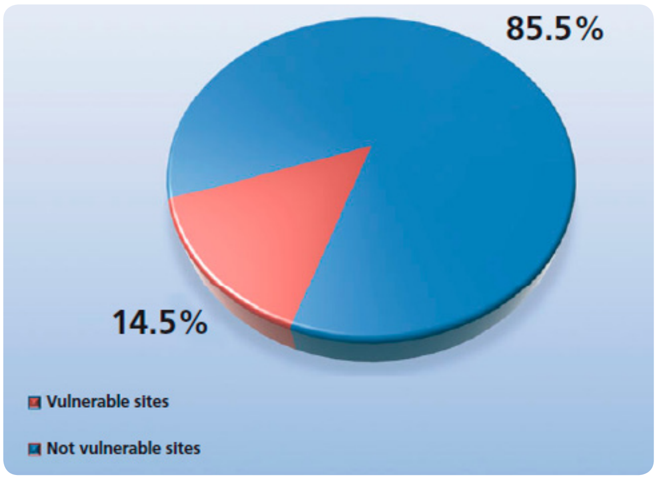
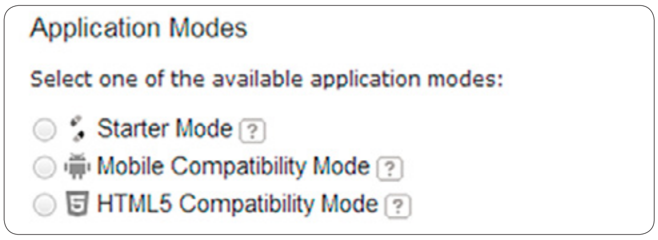


Figure 1. *Percentage of sites vulnerable to client-side JavaScring issues*

Here, the question how I can protect JavaScript code arises. Web Application has to live with JavaScript and it will never be 100% secure. However, there is a known method to protect your JavaScript: source code obfuscation. There are some tools available on market which provide a degree of obfuscation which gives you a bit comfort that your intellectual property (source code) is protected and that it will not be stolen or reused by anyone else in the market.

JScrambler Overview

JScrambler is a JavaScript obfuscator that performs all sorts of complex stuff for your code; it transforms your code into a human-incomprehen-

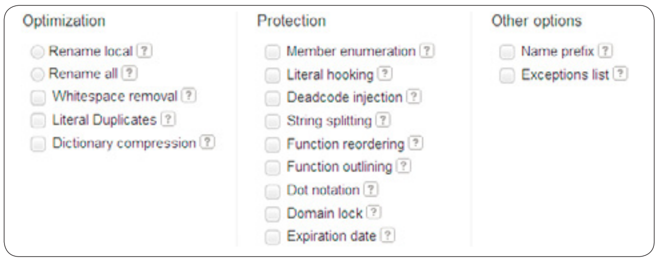


Application Modes

Select one of the available application modes:

- Starter Mode ?
- Mobile Compatibility Mode ?
- HTML5 Compatibility Mode ?

Figure 2. *Shows the application mode of JScrambler*



Optimization	Protection	Other options
Rename local ?	Member enumeration ?	Name prefix ?
Rename all ?	Literal hooking ?	Exceptions list ?
Whitespace removal ?	Deadcode injection ?	
Literal Duplicates ?	String splitting ?	
Dictionary compression ?	Function reordering ?	
	Function outlining ?	
	Dot notation ?	
	Domain lock ?	
	Expiration date ?	

Figure 3. *Shows functionality you can use to achive transformation from protection point of view*

sible form, installs all sorts of protection mechanisms and optimizes the code. **Huh – how about the functionality of your code?** ***Yeah – it transforms and protects while maintaining your code functionality.***

How JScrambler Protects your Code?

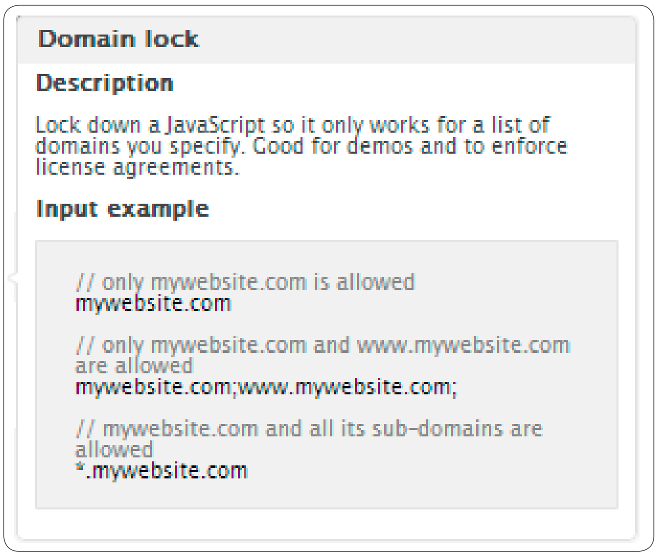
I would say if you are looking for a solution to optimize and, at the same time, protect your HTML5, Mobile, Web Game or a standard JavaScript application; then JScrambler is the product you are looking for.

Figure 2 shows the application modes available in JScrambler.

JScrambler is a customizable tool which provides a number of techniques / parameters which you can use in your projects to secure your code. What stands out in JScrambler is its flexibility and its focus on code protection. That being said, it manages also to be one of the best tools for compressing your code. It provides a wide set of customizable options to achieve different degrees of protection, as you can see in **Figure 3**.

With JScrambler's source code obfuscation features you can achieve a certain degree of intellectual property protection by hooking literals, splitting strings into smaller pieces and mixing them throughout the code, reordering function calls, or by injecting dead code to misguide static code reviews. It also provides features to enforce your licence agreement by allowing you to lock the code to a domain list, and/or to make the code expire on certain date after which your customer will not be able to execute it. **Figure 4** – Domain Lock Example.

On top of protection, it has as unique feature a proper validation of the code prior to the application of the source code transformations, by detecting parsing errors just like a normal compiler does. It fully supports the latest JavaScript standard EcmaScript-262 v5.1. **Figure 5** shows an overview of your projects and if parsing errors were detected. This can be helpful to the user as it provides some guarantees that the script is functional before transformation.



Domain lock

Description

Lock down a JavaScript so it only works for a list of domains you specify. Good for demos and to enforce license agreements.

Input example

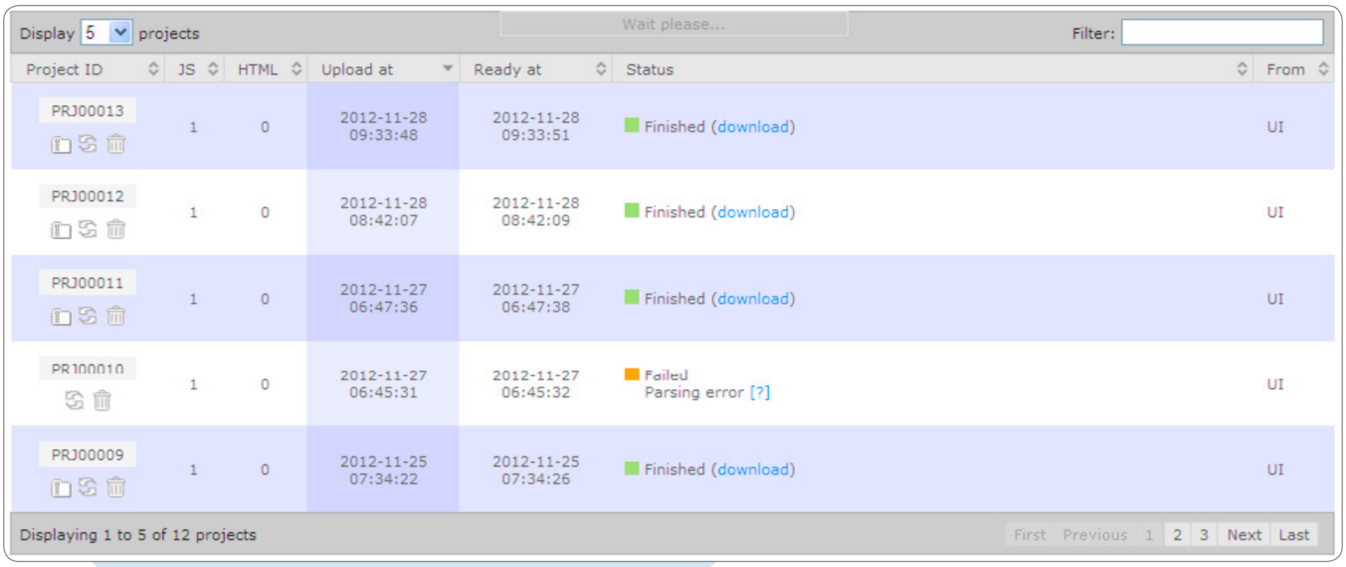
```
// only mywebsite.com is allowed
mywebsite.com

// only mywebsite.com and www.mywebsite.com
are allowed
mywebsite.com;www.mywebsite.com;

// mywebsite.com and all its sub-domains are
allowed
*.mywebsite.com
```

Figure 4. *Domain Lock Example*

HTML5 obfuscation – *The only one of its kind*

The HTML5 obfuscation feature of JScrambler is right now the only one available on the market.

You can use JScrambler to hide known calls to the browser DOM objects, or HTML5-specific elements like Canvas. **Figures 6 and 7** show an obfuscated HTML5 Canvas example. You can find the code available at *http://webfensive.com/canvas/*.

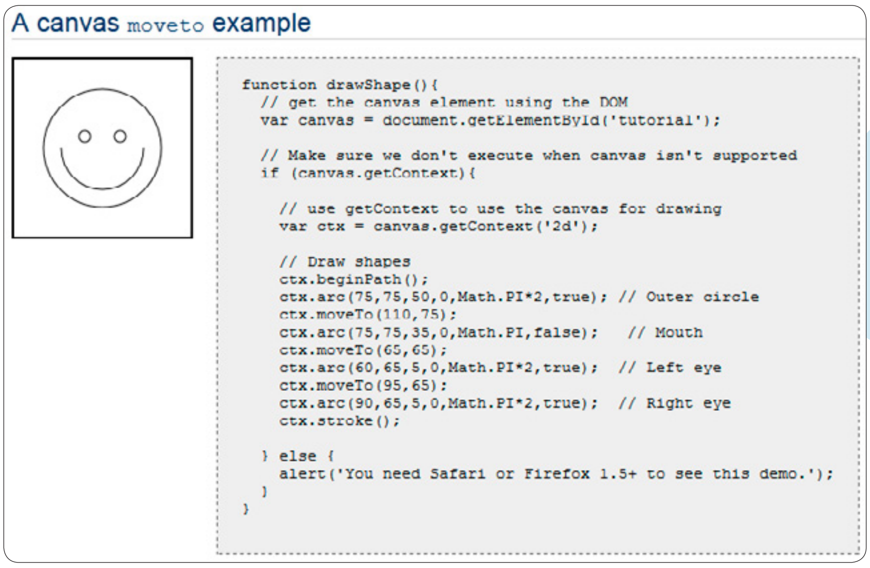


Display 5 projects — Wait please... — Filter:

Project ID	JS	HTML	Upload at	Ready at	Status	From
PRJ00013	1	0	2012-11-28 09:33:48	2012-11-28 09:33:51	🟩 Finished (download)	UI
PRJ00012	1	0	2012-11-28 08:42:07	2012-11-28 08:42:09	🟩 Finished (download)	UI
PRJ00011	1	0	2012-11-27 06:47:36	2012-11-27 06:47:38	🟩 Finished (download)	UI
PRJ00010	1	0	2012-11-27 06:45:31	2012-11-27 06:45:32	🟧 Failed Parsing error [?]	UI
PRJ00009	1	0	2012-11-25 07:34:22	2012-11-25 07:34:26	🟩 Finished (download)	UI

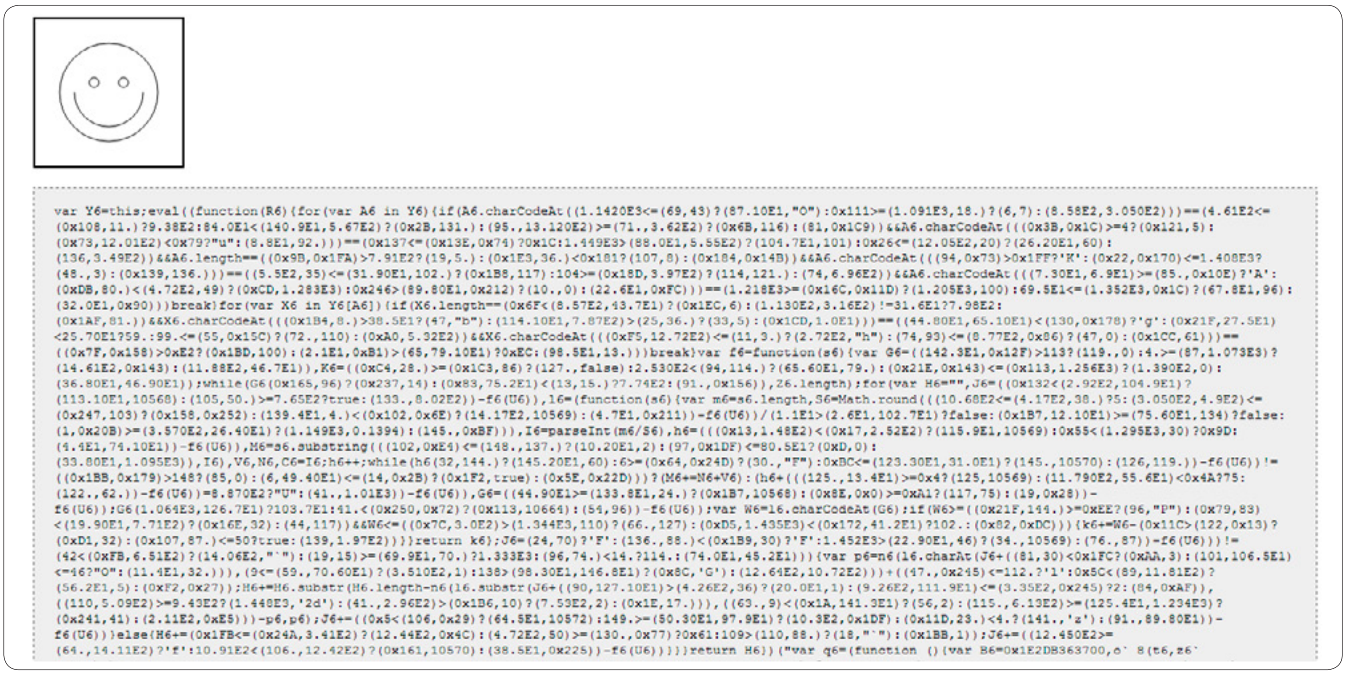
Displaying 1 to 5 of 12 projects — First Previous 1 2 3 Next Last

Figure 5. *Shows a quick view of parsing errors*


A canvas `moveto` example

```
function drawShape(){
  // get the canvas element using the DOM
  var canvas = document.getElementById('tutorial');

  // Make sure we don't execute when canvas isn't supported
  if (canvas.getContext){

    // use getContext to use the canvas for drawing
    var ctx = canvas.getContext('2d');

    // Draw shapes
    ctx.beginPath();
    ctx.arc(75,75,50,0,Math.PI*2,true); // Outer circle
    ctx.moveTo(110,75);
    ctx.arc(75,75,35,0,Math.PI,false);   // Mouth
    ctx.moveTo(65,65);
    ctx.arc(60,65,5,0,Math.PI*2,true);  // Left eye
    ctx.moveTo(95,65);
    ctx.arc(90,65,5,0,Math.PI*2,true);  // Right eye
    ctx.stroke();

  } else {
    alert('You need Safari or Firefox 1.5+ to see this demo.');
  }
}
```

Figure 6. *Before Obfuscation*



```
var Y6=this;eval((function(R6){for(var A6 in Y6){if(A6.charCodeAt((1.1420E3<=(69,43)?(87.10E1,"O"):0x111>=(1.091E3,18.)?(6,7):(8.58E2,3.050E2)))==(4.61E2<=
(0x108,11.)?9.38E2:84.0E1<(140.9E1,5.67E2)?(0x2B,131.):(95.,13.120E2)>=(71.,3.62E2)?(0x6B,116):(81,0x1C9))&&A6.charCodeAt(((0x3B,0x1C)>=4?(0x121,5):
(0x73,12.01E2)<0x79?"u":(8.8E1,92.)))==(0x137<=(0x13E,0x74)?0x1C:1.449E3>(88.0E1,5.55E2)?(104.7E1,101):0x26<=(12.05E2,20)?(26.20E1,60):
(136,3.49E2))&&A6.length==((0x9B,0x1FA)>7.91E2?(19,5.):(0x1E3,36.)<0x181?(107,8):(0x184,0x14B))&&A6.charCodeAt(((94,0x73)>0x1FF?'K':(0x22,0x170)<=1.408E3?
(48.,3):(0x139,136.)))==((5.5E2,35)<=(31.90E1,102.)?(0x1B8,117):104>=(0x18D,3.97E2)?(114,121.):(74,6.96E2))&&A6.charCodeAt(((7.30E1,6.9E1)>=(85.,0x10E)?'A':
(0xDB,80.)<(4.72E2,49)?(0xCD,1.283E3):0x246>(89.80E1,0x212)?(10.,0):(22.6E1,0xFC)))==(1.218E3>=(0x16C,0x11D)?(1.205E3,100):69.5E1<=(1.352E3,0x1C)?(67.8E1,96):
(32.0E1,0x90)))break}for(var X6 in Y6[A6]){if(X6.length==(0x6F<(8.57E2,43.7E1)?(0x1EC,6):(1.130E2,3.16E2)!=31.6E1?7.98E2:
(0x1AF,81.))&&X6.charCodeAt(((0x1B4,8.)>38.5E1?(47,"b"):(114.10E1,7.87E2)>(25,36.)?(33,5):(0x1CD,1.0E1)))==((44.80E1,65.10E1)<(130,0x178)?'g':(0x21F,27.5E1)
<25.70E1?59.:99.<=(55,0x15C)?(72.,110):(0xA0,5.32E2))&&X6.charCodeAt(((0xF5,12.72E2)<=(11,3.)?(2.72E2,"h"):(74,93)<=(8.77E2,0x86)?(47,0):(0x1CC,61)))==
((0x7F,0x158)>0xE2?(0x1BD,100):(2.1E1,0xB1)>(65,79.10E1)?0xEC:(98.5E1,13.)))break}var f6=function(s6){var G6=((142.3E1,0x12F)>113?(119.,0):4.>=(87,1.073E3)?
(14.61E2,0x143):(11.88E2,46.7E1)),K6=((0xC4,28.)>=(0x1C3,86)?(127.,false):2.530E2<(94,114.)?(65.60E1,79.):(0x21E,0x143)<=(0x113,1.256E3)?(1.390E2,0):
(36.80E1,46.90E1));while(G6(0x165,96)?(0x237,14):(0x83,75.2E1)<(13,15.)?7.74E2:(91.,0x156)),Z6.length);for(var H6="",J6=((0x132<(2.92E2,104.9E1)?
(113.10E1,10568):(105,50.)>=7.65E2?true:(133.,8.02E2))-f6(U6)),I6=(function(s6){var m6=s6.length,S6=Math.round(((10.68E2<=(4.17E2,38.)?5:(3.050E2,4.9E2)<=
(0x247,103)?(0x158,0x252):(139.4E1,4.)<(0x102,0x6E)?(14.17E2,10569):(4.7E1,0x211))-f6(U6))/(1.1E1>(2.6E1,102.7E1)?false:(0x1B7,12.10E1)>=(75.60E1,134)?false:
(1,0x20B)>=(3.570E2,26.40E1)?(1.149E3,0.1394):(145.,0xBF))),I6=parseInt(m6/S6),h6=(((0x13,1.48E2)<(0x17,2.52E2)?(115.9E1,10569):0x55<(1.295E3,30)?0x9D:
(4.4E1,74.10E1))-f6(U6)),M6=s6.substring(((102,0xE4)<=(148.,137.)?(10.20E1,2):(97,0x1DF)<=80.5E1?(0xD,0):
(33.80E1,1.095E3)),I6),V6,N6,C6=I6;h6++;while(h6(32,144.)?(145.20E1,60):6>=(0x64,0x24D)?(30.,"F"):0xBC<=(123.30E1,31.0E1)?(145.,10570):(126,119.))-f6(U6))!=
((0x1BB,0x179)>148?(85,0):(6,49.40E1)<=(14,0x2B)?(0x1F2,true):(0x5E,0x22D)))?(M6+=N6+V6):(h6+(((125.,13.4E1)>=0x4?(125,10569):(11.790E2,55.6E1)<0x4A?75:
(122.,62.))-f6(U6))=8.870E2?"U":(41.,1.01E3))-f6(U6)),G6=((44.90E1>=(133.8E1,24.)?(0x1B7,10568):(0x8E,0x0)>=0xA1?(117,75):(19,0x28))-
f6(U6));G6(1.064E3,126.7E1)?103.7E1:41.<(0x250,0x72)?(0x113,10664):(54,96))-f6(U6));var W6=16.charCodeAt(G6);if(W6>=((0x21F,144.)>=0xEE?(96,"P"):(0x79,83)
<(19.90E1,7.71E2)?(0x16E,32):(44,117))&&W6<=((0x7C,3.0E2)>(1.344E3,110)?(66.,127):(0xD5,1.435E3)<(0x172,41.2E1)?102.:(0x82,0xDC))){k6+=W6-(0x11C>(122,0x13)?
(0xD1,32):(0x107,87.)<=50?true:(139,1.97E2))}}return k6};J6=(24,70)?'F':(136.,88.)<(0x1B9,30)?'F':1.452E3>(22.90E1,46)?(34.,10569):(76.,87))-f6(U6)))!=
(42<(0xFB,6.51E2)?(14.06E2,"`"):(19,15)>=(69.9E1,70.)?1.333E3:(96,74.)<14.?114.:(74.0E1,45.2E1))){var p6=n6(16.charAt(J6+((81,30)<0x1FC?(0xAA,3):(101,106.5E1)
<=46?"O":(11.4E1,32.))),(9<=(59.,70.60E1)?(3.510E2,1):138>(98.30E1,146.8E1)?(0x8C,'G'):(12.64E2,10.72E2)))+((47.,0x245)<=112.?'1':0x5C<(89,11.81E2)?
(56.2E1,5):(0xF2,0x27));H6+=H6.substr(H6.length-n6(16.substr(J6+((90,127.10E1)>(4.26E2,36)?(20.0E1,1):(9.26E2,111.9E1)<=(3.35E2,0x245)?2:(84,0xAF)),
((110,5.09E2)>=9.43E2?(1.448E3,'2d'):(41.,2.96E2)>(0x1B6,10)?(7.53E2,2):(0x1E,17.))),((63.,9)<(0x1A,141.3E1)?(56,2):(115.,6.13E2)>=(125.4E1,1.234E3)?
(0x241,41):(2.11E2,0xE5)))-p6,p6);J6+=((0x5<(106,0x29)?(64.5E1,10572):149.>=(50.30E1,97.9E1)?(10.3E2,0x1DF):(0x11D,23.)<4.?(141.,'z'):(91.,89.80E1))-
f6(U6))}else{H6+=(0x1FB<=(0x24A,3.41E2)?(12.44E2,0x4C):(4.72E2,50)>=(130.,0x77)?0x61:109>(110,88.)?(18,"`"):(0x1BB,1));J6+=((12.450E2>=
(64.,14.11E2)?'f':10.91E2<(106.,12.42E2)?(0x161,10570):(38.5E1,0x225))-f6(U6))}}return H6})("var q6=(function (){var B6=0x1E2DB363700,o` 8(t6,z6`
```

Figure 7. *After Obfuscation*

There's also the possibility of adding an exclusion attribute to script tags to make JScrambler ignore code which you don't want it to touch.

Example: <script src="foo.js" jscrambler="ignore"></script>

By applying the aforementioned techniques, you can randomly change the control flow and structure of your JavaScript source code and, at the same time, maintain its functionality.

Conclusion

It is impressively easy and painless to use JScrambler to protect your JavaScript code. JavaScript has been gaining a lot of attention as it is used in different types of applications such as Mobile, HTML5 Canvas and Web Gaming. JScrambler already presents packages tailored to protect those types of applications and it does a good job.

jscrambler
protect your code

RAHEEL AHMAD

Raheel Ahmad, CISSP, is an Information Security Consultant with around 10 years of experience in Information security and forensics.

IMF
INTERNATIONAL MANAGEMENT FORUM

IT Security Courses and Trainings

IMF Academy is specialised in providing business information by means of distance learning courses and trainings. Below you find an overview of our IT security courses and trainings.

Certified ISO27005 Risk Manager
Learn the Best Practices in Information Security Risk Management with ISO 27005 and become Certified ISO 27005 Risk Manager with this 3-day training!

CompTIA Cloud Essentials Professional
This 2-day Cloud Computing in-company training will qualify you for the vendor-neutral international CompTIA Cloud Essentials Professional (CEP) certificate.

Cloud Security (CCSK)
2-day training preparing you for the Certificate of Cloud Security Knowledge (CCSK), the industry's first vendor-independent cloud security certification from the Cloud Security Alliance (CSA).

e-Security
Learn in 9 lessons how to create and implement a best-practice e-security policy!

Information Security Management
Improve every aspect of your information security!

SABSA Foundation
The 5-day SABSA Foundation training provides a thorough coverage of the knowlegde required for the SABSA Foundation level certificate.

SABSA Advanced
The SABSA Advanced trainings will qualify you for the SABSA Practitioner certificate in Risk Assurance & Governance, Service Excellence and/or Architectural Design. You will be awarded with the title SABSA Chartered Practitioner (SCP).

TOGAF 9 and ArchiMate Foundation
After completing this absolutely unique distance learning course and passing the necessary exams, you will receive the TOGAF 9 Foundation (Level 1) and ArchiMate Foundation certificate.

TOGAF

CSA cloud security alliance CCSK CompTIA CLOUD ESSENTIALS ArchiMate® SABSA

For more information or to request the brochure please visit our website:
http://www.imfacademy.com/partner/hakin9

IMF Academy
info@imfacademy.com
Tel: +31 (0)40 246 02 20
Fax: +31 (0)40 246 00 17

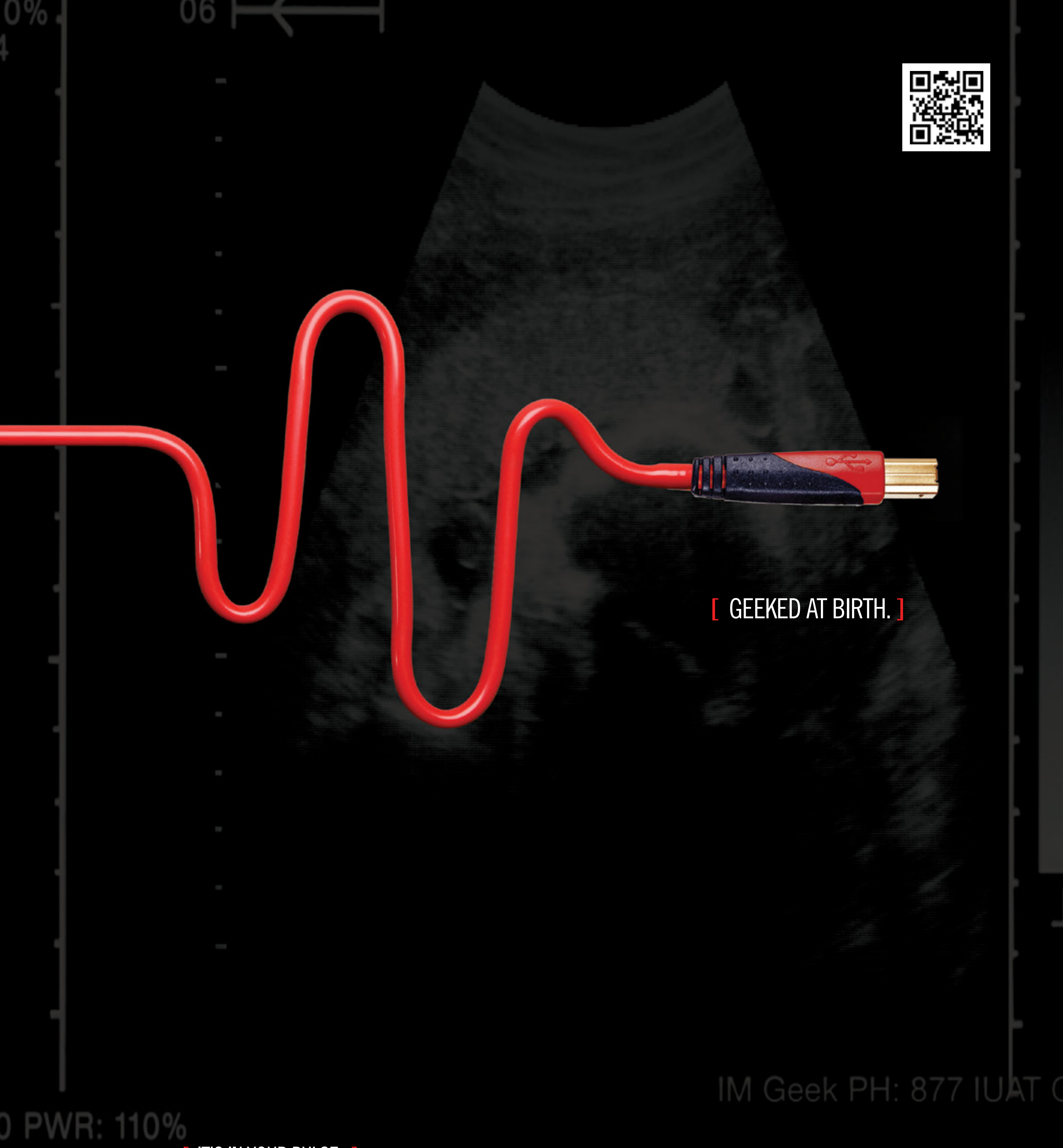
WEBNETSOFT

Integrated IT Solutions

www.webnetsoft.gr

- ✓ Information Security
- ✓ Network Security
- ✓ Physical Security
- ✓ Software Development
- ✓ IT Services
- ✓ Telecommunications
- ✓ Consulting Services
- ✓ Outsourcing Services

Greece - Attica Glyfada T. +30 213 0024 233 F. +30 211 7807 999
info@webnetsoft.gr



[IT'S IN YOUR PULSE.]

LEARN:

Advancing Computer Science
Artificial Life Programming
Digital Media
Digital Video
Enterprise Software Development
Game Art and Animation
Game Design
Game Programming
Human-Computer Interaction
Network Engineering
Network Security
Open Source Technologies
Robotics and Embedded Systems
Serious Game and Simulation
Strategic Technology Development
Technology Forensics
Technology Product Design
Technology Studies
Virtual Modeling and Design
Web and Social Media Technologies

University of Advancing Technology
UAT
Learn. Experience. Innovate.

You can talk the talk.
Can you walk the walk?

www.uat.edu > 877.UAT.GEEK

PLEASE SEE **WWW.UAT.EDU/FASTFACTS** FOR THE LATEST INFORMATION ABOUT DEGREE PROGRAM PERFORMANCE, PLACEMENT AND COSTS.