Reverse Engineering Compendium

How to Analyze Applications With Olly Debugger?

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

How to Identify and Bypass Anti-reversing Techniques?

Write Your Own Debugger
Next Generation Sandbox System

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

Technology Leader

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

Cross Platform

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

Quality Support and Consulting

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

The new solution for in-depth malware analysis on Android based systems. Using **Hybrid Code Analysis**, static and dynamic analysis is combined in a clever way.

Powerful Instrumentation Engine

The highly-configurable, generic Instrumentation Engine not only analyzes **System API calls**, but any function matching specified signatures up to parameter level.

Generic Behavior Signatures

Providing an open interface and a solid initial set of generic behavior signatures, application activity is abstracted into well-formatted report data.

Free Services Available Online

All of Joe Security’s Sandbox Systems are available as free web services at [apk-analyzer.net](http://apk-analyzer.net), [file-analyzer.net](http://file-analyzer.net), [url-analyzer.net](http://url-analyzer.net) and [document-analyzer.net](http://document-analyzer.net)
Dear Readers,

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

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

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

Enjoy your time with Hakin9!

Regards,
Ewelina Nazarczuk
Hakin9 Magazine Junior Product Manager and Hakin9 Team
How to Analyze Applications With Olly Debugger?

By Jaromir Horejsi, Malware Analyst at AVAST Software

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.

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

By Shane R. Spencer, Information Technology Professional

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.

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

By Jacek Adam Piasecki, Tester/Programmer

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.

MALWARE REVERSE ENGINEERING

Malware Reverse Engineering

By Bamidele Ajayi, OCP, MCTS, MCITP EA, CISA, CISM

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

Android Reverse Engineering: An Introductory Guide to Malware Analysis

By Vicente Aguilera Diaz, CISA, CISSP, CSSLP, PCI ASV, ITIL Foundation, CEH|I, ECSP|I, OPSA

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

Deep Inside Malicious PDF

By Yehia Mamdouh, Founder and Instructor of Master Metasploit Courses, CEH, CCNA

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

**REVERSE ENGINEERING TECHNIQUES**

**How to Identify and Bypass Anti-reversing Techniques?**  
*By Eoin Ward, Security Analyst – Anti Malware at Microsoft*

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.

**How to Defeat Code Obfuscation While Reverse Engineering?**  
*By Adam Kujawa, Malware Intelligence Analyst at Malwarebytes*

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.

**Reverse Engineering – Shellcodes Techniques**  
*By Eran Goldstein, CEH, CEI, CISO, Security+, MCSA, MCSE Security*

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

**REVERSE IT YOURSELF**

**How to Reverse the Code?**  
*By Raheel Ahmad, Writer – Information Security Analyst & eForensics at Hakin9*

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.

**How to Reverse Engineer dot NET Assemblies**  
*By Soufiane Tahiri, InfoSec Institute Contributor and Computer Security Researcher*

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

Reversing with Stack-Overflow and Exploitation

By Bikash Dash, RHCSA, RHCE, CSSA
The prevalence of security holes in program and protocols, the increasing size and complexity of the internet, and the sensitivity of the information stored throughout have created a target-rich environment for our next generation advisory. The criminal element is applying advance technique to evade the software/tool security. So the Knowledge of Analysis is necessary. And that pin point is called “The Art Of Reverse Engineering”

How to Reverse Engineer?

By Lorenzo Xie, The owner of XetoWare.COM
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?

Write your own Debugger

By Amr Thabet, malware Researcher at Q-CERT, ictQUATAR
Do you want to write your own debugger? … Do you have a new technology and see the already known products like OllyDbg or IDA Pro don’t have this technology? … Do you write plugins in OllyDbg and IDA Pro but you need to convert it into a separate application? … This article is for you. In this article, I’m going to teach you how to write a full functional debugger using the Security Research and Development Framework (SRDF) … how to disassemble instructions, gather Process Information and work with PE Files … and how to set breakpoints and work with your debugger.
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.
How to Analyze Applications With Olly Debugger?

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

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

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.

Figure 4. Setting up memory breakpoint

Figure 5. Setting up hardware breakpoint
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 these 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-

![Figure 7. The first message box in crackme_01.exe](image)

![Figure 8. The second message box in crackme_01.exe](image)
dress 401021, where we set software breakpoint. It is just before the second message box will be displayed. Now, we will press F8 Step Over, toolbar icon and another message is displayed (Figure 8).

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

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

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

![Figure 9. The third message box in crackme_01.exe](image1)

![Figure 10. Dialog for replacing instructions](image2)

![Figure 11. Replacing with NOP instructions](image3)

![Figure 12. Replaced PUSHes and CALL](image4)
How to Analyze Applications With Olly Debugger?

Figure 13. Copying modifications into new executable

Figure 14. Saving modified executable into new file
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).
How to Analyze Applications With Olly Debugger?

Figure 17. Interesting string

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...
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 447597, 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
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...

\[
(c \times e) + (r \times m) + (a \times k) + (c \times c) + (k \times a) + (m \times r) + (e \times c) = \\
= (0x63 \times 0x65) + (0x72 \times 0x6D) + (0x61 \times 0x6B) + (0x63 \times 0x63) + (0x6B \times 0x61) + (0x6D \times 0x72) + (0x65 \times 0x63) = \\
= 0x270F + 0x308A + 0x308A + 0x2649 + 0x288B + 0x308A + 0x270F = 0x12691 = 75409 \text{ (in decimal)}
\]

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.
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
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.

**Prerequisites**

- Install Socat – Multipurpose relay for bidirectional data transfer: [http://www.dest-unreach.org/socat/](http://www.dest-unreach.org/socat/)
- Install TinyCA – Simple graphical program for certification authority management: [http://tinyca.sm-zone.net/](http://tinyca.sm-zone.net/)
- Install LogJam – Client for LiveJournal-based sites: [http://andy-shev.github.com/LogJam/](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 gener-
ated 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.

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 standard 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.
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-OPENSSL:LISTEN:443,verify=0,fork,cafile=www.livejournal.com-key.pem
spencersr@bigboote:~$ sudo socat -vvv OPENSSL:LISTEN:443,verify=0,fork,key=www.livejournal.com-key.pem,
```

Simple enough. Browsing to https://www.livejournal.com with w3m and wget should work success-fully 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+rwx \
/usr/share/ca-certificates/custom/Yoyodyne-cacert.crt
spencersr@bigboote:~$ sudo dpkg-reconfigure -p declining 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

```bash
> 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
> 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-VNS-Id: bill-varn21\r
X-Gateway: bill=swlb10\r
\r
auth_scheme
c0
challenge
c0:1346227200:656:60:xxxxxxxx:xxxxxxxxxxxxxxx
expire_time
1346227916
server_time
1346227856
success
OK
```
Socat instance one:

```bash
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:

```bash
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 127.0.0.1 (Figure 4).

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

**What now?**

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

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

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

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

**Conclusion**

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

Remember to thank your open source hacker friends.

---

**SHANE R. SPENCER**

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

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

www.threat-analytics.com

www.threatintelligence.com

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

Contact Us Now
IDA Pro

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

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

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

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

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

Loading an Executable
When you load an executable, IDA Pro will try to recognize the file’s format and processor architecture. Figure 1 displays the first step in loading an executable into IDA Pro. When loading a file into IDA Pro (such as a PE file with Intel x86 architecture), the program maps the file into memory as if it had been loaded by the operating system loader. To have IDA Pro disassemble the file as a raw binary, choose the Binary File option in the top box. This option can prove useful because malware sometimes appends shellcode, additional data, encryption parameters, and even additional...
executables to legitimate PE files, and this extra data won’t be loaded into memory when the malware is run by Windows or loaded into IDA Pro. In addition, when you are loading a raw binary file containing shellcode, you should choose to load the file as a binary file and disassemble it.

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

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

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

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

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

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

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.

![Figure 3. Text mode of IDA Pro’s disassembly window](image)
**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, double-click the imported function of interest, and then use the cross-reference feature to locate the import call in the code listing.

Listing 1. Navigational links within the disassembly window

```
00401075  jnz  short loc_40107E
00401077  mov  [ebp+var_18], 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
```

**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.

Figure 4. Navigational buttons

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 references, they are also navigational links. For example, aPrintNumberD can be used to jump the display to where that string is defined in memory.

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

Listing 3. Code cross-references

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Related Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00401000</td>
<td>sub_401000 proc near</td>
<td>CODE XREF: _main+3p</td>
<td></td>
</tr>
<tr>
<td>00401000</td>
<td>push ebp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00401000</td>
<td>mov ebp, esp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00401000</td>
<td>loc_401003:</td>
<td>CODE XREF: 00401020</td>
<td>var_C = dword ptr -0Ch</td>
</tr>
<tr>
<td>00401000</td>
<td>sub_401000+19j</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00401003</td>
<td>mov eax, 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00401008</td>
<td>test eax, eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0040100A</td>
<td>jz short loc_40101B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00401011</td>
<td>call printf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00401016</td>
<td>add esp, 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00401019</td>
<td>jmp short loc_401003</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Listing 4. Data cross-references

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Related Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0040C000</td>
<td>dword_40C000 dd 7P000001h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0040C004</td>
<td>aHostnamePort &lt;Hostname&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0040C010</td>
<td>loc_40104B:</td>
<td>CODE XREF:</td>
<td></td>
</tr>
<tr>
<td>0040104B</td>
<td>sub_401000+3o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0040104B</td>
<td>call sub_401000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Listing 5. Function and stack example

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Related Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00401020</td>
<td>; ===== SUBROUTINE =====</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00401020</td>
<td>move ebp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00401020</td>
<td>; Attributes: ebp-based frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00401020</td>
<td>function proc near</td>
<td>CODE XREF:</td>
<td></td>
</tr>
<tr>
<td>00401020</td>
<td>_main+1Cp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6. Xrefs window
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_408990, 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
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 -0xCh; 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.

Table 2. Function Operand Manipulation

<table>
<thead>
<tr>
<th>Without renamed arguments</th>
<th>With renamed arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>004013C8 mov eax, [ebp+arg_4]</td>
<td>004013C8 mov eax, [ebp+port_str]</td>
</tr>
<tr>
<td>004013CB push eax</td>
<td>004013CB push eax</td>
</tr>
<tr>
<td>004013CC call _atoi</td>
<td>004013CC call _atoi</td>
</tr>
<tr>
<td>004013D1 add esp, 4</td>
<td>004013D1 add esp, 4</td>
</tr>
<tr>
<td>004013D4 mov [ebp+var_598], ax</td>
<td>004013D4 mov [ebp+port], ax</td>
</tr>
<tr>
<td>004013DB movzx ecx, [ebp+var_598]</td>
<td>004013DB movzx ecx, [ebp+port]</td>
</tr>
<tr>
<td>004013E2 test ecx, ecx</td>
<td>004013E2 test ecx, ecx</td>
</tr>
<tr>
<td>004013E4 jnz short loc_4013F8</td>
<td>004013E4 jnz short loc_4013F8</td>
</tr>
<tr>
<td>004013E6 push offset aError</td>
<td>004013E6 push offset aError</td>
</tr>
<tr>
<td>004013EB call printf</td>
<td>004013EB call printf</td>
</tr>
<tr>
<td>004013FD add esp, 4</td>
<td>004013FD add esp, 4</td>
</tr>
<tr>
<td>004013F3 jmp loc_4016FB</td>
<td>004013F3 jmp loc_4016FB</td>
</tr>
<tr>
<td>004013F8 ; --------------------</td>
<td>004013F8 ; ------------------</td>
</tr>
<tr>
<td>004013F8</td>
<td>004013F8</td>
</tr>
<tr>
<td>004013F8 loc_4013F8:</td>
<td>004013F8 loc_4013F8:</td>
</tr>
<tr>
<td>004013F8 movzx edx, [ebp+var_598]</td>
<td>004013F8 movzx edx, [ebp+port]</td>
</tr>
<tr>
<td>004013FF push edx</td>
<td>004013FF push edx</td>
</tr>
<tr>
<td>00401400 call dshtons</td>
<td>00401400 call dshtons</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Before symbolic constants</th>
<th>After symbolic constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov esi, [esp+1Ch+argv]</td>
<td>mov esi, [esp+1Ch+argv]</td>
</tr>
<tr>
<td>mov edx, [esi+4]</td>
<td>mov edx, [esi+4]</td>
</tr>
<tr>
<td>mov edi, ds:CreateFileA</td>
<td>mov edi, ds:CreateFileA</td>
</tr>
<tr>
<td>push 0 ; hTemplateFile</td>
<td>push NULL ; hTemplateFile</td>
</tr>
<tr>
<td>push 80h ; dwFlagsAndAttributes</td>
<td>push FILE _ATTRIBUTE _NORMAL ; dwFlagsAndAttributes</td>
</tr>
<tr>
<td>push 3 ; dwCreationDisposition</td>
<td>push OPEN _EXISTING ; dwCreationDisposition</td>
</tr>
<tr>
<td>push 0 ; lpSecurityAttributes</td>
<td>push NULL ; lpSecurityAttributes</td>
</tr>
<tr>
<td>push 1 ; dwShareMode</td>
<td>push GENERIC _READ ; dwShareMode</td>
</tr>
<tr>
<td>push 00000000h ; dwDesiredAccess</td>
<td>push FILE _SHARE _READ ; dwShareMode</td>
</tr>
<tr>
<td>push edx ; lpFileName</td>
<td>push edx ; lpFileName</td>
</tr>
<tr>
<td>call edi ; CreateFileA</td>
<td>call edi ; CreateFileA</td>
</tr>
</tbody>
</table>

![Figure 10. Function operand manipulation](image)

![Figure 11. Standard symbolic constant window](image)

Table 3. Code Before and After Standard Symbolic Constants
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, 142 in octal, 1100010 in binary, or the character b in ASCII – whatever suits your needs and your situation.

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

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

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

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

---

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

<table>
<thead>
<tr>
<th>File before pressing C</th>
<th>File after pressing C</th>
</tr>
</thead>
<tbody>
<tr>
<td>00008384 db 28h ; (</td>
<td>00008384 db 28h ; (</td>
</tr>
<tr>
<td>00008385 db 0FCh ; n</td>
<td>00008385 db 0FCh ; n</td>
</tr>
<tr>
<td>00008386 db 10h</td>
<td>00008386 db 10h</td>
</tr>
<tr>
<td>00008387 db 90h ; Ė</td>
<td>00008387 nop</td>
</tr>
<tr>
<td>00008388 db 90h ; Ė</td>
<td>00008388 nop</td>
</tr>
<tr>
<td>00008389 db 88h ; i</td>
<td>00008389 mov ebx, eax</td>
</tr>
<tr>
<td>0000838A db 0D8h ; +</td>
<td>0000838B add ebx, 28h ; '('</td>
</tr>
<tr>
<td>0000838B db 83h ; Ą</td>
<td>0000838E add dword ptr [ebx], 1Bh</td>
</tr>
<tr>
<td>0000838C db 0C3h ; +</td>
<td>00008391 mov ebx, [ebx]</td>
</tr>
<tr>
<td>0000838D db 28h ; (</td>
<td>00008393 xor ecx, ecx</td>
</tr>
<tr>
<td>0000838E db 83h ; Ą</td>
<td>00008395 xor 33h ; 3</td>
</tr>
<tr>
<td>0000838F db 3</td>
<td>00008395 xor _8395:</td>
</tr>
<tr>
<td>00008390 db 1Bh</td>
<td>; CODE XREF: seg000:000083A0j</td>
</tr>
<tr>
<td>00008391 db 88h ; i</td>
<td>00008395 xor byte ptr [ebx], 97h</td>
</tr>
<tr>
<td>00008392 db 1Bh</td>
<td>00008398 inc ebx</td>
</tr>
<tr>
<td>00008393 db 33h ; 3</td>
<td>00008399 inc ecx</td>
</tr>
<tr>
<td>00008394 db 0C9h ; +</td>
<td>0000839A cmp ecx, 700h</td>
</tr>
<tr>
<td>00008395 db 80h ; Ć</td>
<td>000083A0 jnz short loc_8395</td>
</tr>
<tr>
<td>00008396 db 33h ; 3</td>
<td>000083A2 retn 7B1Ch</td>
</tr>
<tr>
<td>00008397 db 97h ; Ą</td>
<td>000083A2 ; --------------------------000083A5 db 16h</td>
</tr>
<tr>
<td>00008398 db 43h ; C</td>
<td>000083A6 db 7Bh ; {</td>
</tr>
<tr>
<td>00008399 db 41h ; Ā</td>
<td>000083A7 db 8Fh ; Ā</td>
</tr>
<tr>
<td>0000839A db 81h ; Ė</td>
<td>0000839B db 0F9h ; Ė</td>
</tr>
<tr>
<td>0000839C db 0</td>
<td>0000839D db 7</td>
</tr>
<tr>
<td>0000839E db 0</td>
<td>0000839F db 0</td>
</tr>
<tr>
<td>000083A0 db 75h ; Ė</td>
<td>000083A1 db 0F3h ; Ė</td>
</tr>
<tr>
<td>000083A2 db 0C2h ; Ė</td>
<td>000083A3 db 1Ch</td>
</tr>
<tr>
<td>000083A4 db 7Bh ; Ė</td>
<td>000083A5 db 16h</td>
</tr>
<tr>
<td>000083A6 db 7Bh ; Ė</td>
<td>000083A7 db 8Fh ; Ė</td>
</tr>
</tbody>
</table>
vide an easily remembered name for the programmer, but they are implemented as an integer in the binary. Unfortunately, once the compiler is done with the source code, it is no longer possible to determine whether the source used a symbolic constant or a literal.

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

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

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

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

On the Web
MOVE TOMORROW’S BUSINESS TO THE CLOUD TODAY

YOUR TRUSTED ADVISOR ON CLOUD COMPUTING

MULTI-VENDOR
ANY DEVICE
HYBRID CLOUD
Malware Reverse Engineering

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

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

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

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

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

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

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

- User-accessible registers – The most common division of user-accessible registers is into data registers and address registers.
- Data registers can hold numeric values such as integer and floating-point values, as well as characters, small bit arrays and other data. In some older and low end CPUs, a special data register, known as the accumulator, is used implicitly for many operations.
• Address registers hold addresses and are used by instructions that indirectly access primary memory. Some processors contain registers that may only be used to hold an address or only to hold numeric values (in some cases used as an index register whose value is added as an offset from some address); others allow registers to hold either kind of quantity. A wide variety of possible addressing modes, used to specify the effective address of an operand, exist. The stack pointer is used to manage the run-time stack. Rarely, other data stacks are addressed by dedicated address registers, see stack machine.

• Conditional registers hold truth values often used to determine whether some instruction should or should not be executed.

• General purpose registers (GPRs) can store both data and addresses, i.e., they are combined Data/Address registers and rarely the register file is unified to include floating point as well.

• Floating point registers (FPRs) store floating point numbers in many architectures.

• Constant registers hold read-only values such as zero, one, or pi.

• Vector registers hold data for vector processing done by SIMD instructions (Single Instruction, Multiple Data).

• Special purpose registers (SPRs) hold program state; they usually include the program counter (aka instruction pointer) and status register (aka processor status word). The aforementioned stack pointer is sometimes also included in this group. Embedded microprocessors can also have registers corresponding to specialized hardware elements.

• Instruction registers store the instruction currently being executed. In some architectures, model-specific registers (also called machine-specific registers) store data and settings related to the processor itself. Because their meanings are attached to the design of a specific processor, they cannot be expected to remain standard between processor generations.

• Control and status registers – There are three types: program counter, instruction registers and program status word (PSW).

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

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

Memory Stacks
There are 3 main sections of memory:

• Stack Section – Where the stack is located, stores local variables and function arguments.

• Data Section – Where the heap is located, stores static and dynamic variables.

• Code Section – Where the actual program instructions are located.

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

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

• GNU Debugger

• Intel Debugger

• LLDB

• Microsoft Visual Studio Debugger

• Valgrind

• WinDbg

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

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

- IDA
- OllyDbg

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

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

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

**Background**

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

Once Zeus Trojan infects a machine, it remains idle until the user visits a Web page with a form to fill out. It allows criminals to add fields to forms at the browser level. This means that instead of directing the end user to a counterfeit website, the user would see the legitimate website as it executes. Debuggers allow breakpoints to be set and the assembly code to be edited at runtime.

**Zeus Crimeware Toolkit**

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

**Architecture**

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

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

**Code Analysis**

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

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

Zeus Configuration file includes some commands namely:

- **url_loader**: Update location of the bot
- **url_server**: Command and control server location
- **AdvancedConfigs**: Alternate URL locations for updated configuration files
- **Webfilters**: Web filters specify a list of URLs (with masks) that should be monitored. Any data sent to these URLs such as online banking credentials is then sent to the command and control server. This data is captured on the client prior to SSL. In addition, one can specify to take a screenshot when the left-button of the mouse is clicked, which is useful in recording PIN numbers selected on virtual keyboards.
- **WebDataFilters**: Web data filters specify a list of URLs (with masks) and also string patterns in the data that must be matched. Any data sent to these URLs and match the specified string patterns such as ‘password’ or ‘login’ is then sent to the command and control server. This data is also captured on the client prior to SSL.
- **WebFakes**: Redirect the specified URL to a different URL, which will host a potentially fake version of the page.
- **TANGrabber**: The TAN (Transaction Authentication Number) grabber routine is a specialized routine that allows you to configure match patterns to search for transaction numbers in data posted to online banks. The match patterns include values such as the variable name and length of the TAN.
- **DNSMap**: Entries to be added to the HOSTS file often used to prevent access to security sites or redirect users to fake Web sites.

**References**

- [http://searchsecurity.techtarget.com/definition/Zeus-Trojan-Zbot](http://searchsecurity.techtarget.com/definition/Zeus-Trojan-Zbot)
- [https://github.com/Visgean/Zeus](https://github.com/Visgean/Zeus)
- [http://www.cs.fsu.edu](http://www.cs.fsu.edu)

**Conclusion**

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

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

**Part 2 (Continued in Next Article)**

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

---

**BAMIDELE AJAYI**

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

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

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

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

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

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

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

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

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

typically we will find the following resources: Figure 1.

An interesting resource is the “AndroidManifest.xml” file. In this XML file, all specifications of our application are declared, including Activities, Intents, Hardware, Services, Permissions required by the application [2], etc. Note that this is a binary XML file and cannot be readily read or edited directly by humans.

![Figure 1. Typical Structure of an APK File](image-url)
file, so if you want to read easily its contents you should convert it to a human-readable XML format. The “AXMLPrinter2.jar” tool performs this task:

$ java –jar AXMLPrinter2.jar AndroidManifest.xml > AndroidManifest.xml.txt
$ less AndroidManifest.xml.txt

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

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

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

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

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

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

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

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

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

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

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

Figure 2. Contents of an AndroidManifest.xml File

Figure 3. Viewing the Source Code Decompiled with JD-GUI
Although “JD-GUI” allows us to browse the entire hierarchy of objects in a comfortable manner, we generally find applications where there is a large number of Java classes to analyze, so we need to rely on other tools to facilitate the understanding of the code.

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

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

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

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

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

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

**Malware Identification in the Play Store**

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

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

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

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

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

In this article we focus on the second case: application with malicious code that exploits the user’s trust.
How to Identify Malicious Applications on the Play Store?
A malicious application includes code that performs some action not expected by the user. For example, if a user downloads from the official store an application to change the wallpaper of his device, the user do not expect that this app can read his emails, can make phone calls or send SMS messages to premium accounts, for example.

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

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

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

Package
com.ownskin.diy_01zti0rso7rb

Permissions
android.permission/internet
android.permission/read_phone_state
android.permission/access_network_state
android.permission/write_external_storage

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

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

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

Step 1: Get the APK file of the application
There are multiple ways to obtain an APK:

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

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

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

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

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

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

Step 3: Decompile the Java code

Using a Java decompiler (like “JD-GUI”), we can obtain the Java source code from the .class files. In our case, we will choose a fast track. "JD-GUI" allows us to save the entire application source code in a ZIP file. We’ll keep this file as “com.ownskin.diy_01zti0rso7rb-dex2jar.src.zip”, and unzip it to perform a manual scan.

We note that there are 353 Java source files:

```
$ find /vad/lab/Android/com.ownskin.diy_01zti0rso7rb-dex2jar.src/ -type f | wc -l
353
```

Step 4: Find malicious code in the application

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

Listing 1. Finding Malicious Code in the Application

```
$ cd /vad/lab/Android/com.ownskin.diy_01zti0rso7rb-dex2jar.src/
$ grep -i sms -r *
```

```
com/ownskin/diy_01zti0rso7rb/ht.java:import android.telephony.SmsMessage;
com/ownskin/diy_01zti0rso7rb/ht.java:    SmsMessage[] arrayOfSmsMessage = new
SmsMessage[arrayOfObject.length];
com/ownskin/diy_01zti0rso7rb/ht.java:    arrayOfSmsMessage[0] = SmsMessage.createFromPdu((byte[])arrayOfObject[0]);
com/ownskin/diy_01zti0rso7rb/ht.java:    hs.a(this.a, arrayOfSmsMessage[0].getOriginatingAddress());
com/ownskin/diy_01zti0rso7rb/ht.java:    hs.c(this.a, arrayOfSmsMessage[0].getMessageBody());
com/ownskin/diy_01zti0rso7rb/ht.java:    if (!"SMS_MMS".equalsIgnoreCase(this.U))
com/ownskin/diy_01zti0rso7rb/ht.java:        a(Uri.parse("content://sms"));
com/ownskin/diy_01zti0rso7rb/ht.java:        Uri localUri = Uri.parse("content://sms");
com/ownskin/diy_01zti0rso7rb/ht.java:        this.P.l().registerReceiver(this.ac, new
IntentFilter("android.provider.Telephony.SMS_RECEIVED"));
```
Let’s see if the application actually use this permission (Listing 1).

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

- com/ownskin/diy_01zt0rso7rb/hm.java
- com/ownskin/diy_01zt0rso7rb/hs.java
- com/ownskin/diy_01zt0rso7rb/ht.java

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

- com/ownskin/diy_01zt0rso7rb/hm.java
  
  ```java
  if (!"SMS_MMS".equalsIgnoreCase(this.U))
    break label89;
    a(Uri.parse("content://sms"));
    a(Uri.parse("content://mms"));
  ...
  ```

- com/ownskin/diy_01zt0rso7rb/hs.java
  
  It creates a „localUri“ object of the “Uri” class, calling the “parse” method to be used in the query to the Content Provider that allows to access to the SMS inbox:

  ```java
  public static final Uri a = localUri;
  public static final Uri b = Uri.
  withAppendedPath(localUri, “inbox”);
  ...
  ```

- com/ownskin/diy_01zt0rso7rb/ht.java
  
  This class implements a Broadcast Receiver. This is simply an Android component that allows the registered Receiver to be notified of events produced in the system or in the application itself.

  ```java
  ...this.P.l().registerReceiver(this.ac,new
  IntentFilter(“android.provider.
  Telephony.SMS_RECEIVED”));
  ...• com/ownskin/diy_01zt0rso7rb/ht.java
  ```

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

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

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

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

Listing 2. When the User Receives an SMS, the Application Identify its Source and Read the Message
important to recall that we have accepted that the application can perform these actions, because we have accepted the permissions required and the application has informed to us of this situation prior to installation.

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

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

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

References

VICENTE AGUILERA DIAZ

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

Vicente has collaborate in several open-source projects, is a regular speaker at industry conferences and has published several articles and vulnerabilities in specialized media. Vicente has the following certifications: CISA, CISSP, CSSLP, PCI ASV, ITIL Foundation, CEH[, ECSPI[, OPSA and OPST.
Exchange Glances - Look At Each Other's Websites

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

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

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

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

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

In nowadays, people share documents all the time, and most of the attacks are based on client-side attacks and target applications that exist in the user’s employee OS. From one single file, the attacker can compromise a large network. PDF is the most sharing file format due to PDFs can include active content, passed within the enterprise and across networks. In this article, we will make an analysis to catch malicious PDF files.

When we start to check the PDF files that exist in our PC or laptop, we may use an antivirus scanner, but in these days, it seems not good enough to detect malicious PDF that contains a shell code because, as the attacker mostly encrypts it’s counting to bypass the antivirus scanner and in many times targets a zero-day vulnerability that exists in Adobe Acrobat reader or unupdated version, the Figure 1 shows how PDF vulnerabilities are rising every year.

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

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

PDF Header
The first line of PDF shows the PDF format version, the most important line that gives you the basic information of the PDF file, for example, “%PDF-1.4 means that file fourth version.

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

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

The Trailer
The trailer contains links to cross-reference table and always ends up with %%EOF to identify the end.
of a PDF file the trailer enables a user to navigate to the next page by clicking on the link provided.

**Malicious PDF through Metasploit**

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

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

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

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

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

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

**PDFid**

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

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

So we going to use peepdf.

**Peepdf**

Peepdf its python tool very powerful for PDF analysis, the tool provide all necessary components that security researcher need in PDF analysis without using many tools to do that, it support encryption, Object Streams, Shellcode emulation, Javascript Analysis, and for Malicious PDF it shows potential Vulnerabilities, Shows Suspicious Elements, Powerful Interactive Console, PDF Obfuscation (bypassing AVs), Decoding: hexadecimal – ASCII and HEX search (Figure 4).

**Analysis**

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

*choose the LHOST which is our IP address and we can view through typing ifconfig in new terminal

*finally we type exploit to create the PDF file with configuration we created before

---

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

This the default output but we see some interesting things first one we see is the highlighted one object 15 continue JavaScript code and we have also one object 4 continue two executing elements (/AcroForm & /OpenAction) and the last one is /U3D showing to us Known Vulnerability for now we will start to explore this objects by getting an interactive console by typing syntax `/usr/bin/peepdf -i msf.pdf` (Figure 6).
The tree commands shows the logical structure of the file, and starting explore object 4 (/Acro-Form) (Figure 7).

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

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

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

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

**Defend**

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

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

**Conclusion**

We’ve take a tour pdf file format structure and what it counties and we’ve seen how to detect a mali- cious pdf and know where and how can locate sus- picious objects and showing the JavaScript code, an finally know how to defend our network.

YEHIA MAMDOUH EL GHALY  
Certified (CCNA, CEH), Founder and in-structor of Master Metasploit (Course). Trained in (Exploiting Web Applications with Samurai- Application Security- Cyber Crime Investigation). I also have 5 years experience in penetration testing
Reverse Engineering, Malware Analysis, Forensic Analysis, Vulnerability Analysis, Pentest, Hacking, Exploitation & Bug Hunting... www.itsecurity.ma is the most advanced blog in MOROCCO
How to Identify and Bypass Anti-reversing Techniques?

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

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

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

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

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

Tools
For tools I am going to try and limit it to just one; ‘Ollydbg.’ Ollydbg is a debugger just like the debug-
ger 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 Ollydbg 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

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 1. Setting up the UDD directory

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>.

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 ASCII 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

Listing 1. Verification of dropped location

```c
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);
}
```
How to Identify and Bypass Anti-reversing Techniques?

Layer 2: Anti-debugger

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

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

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

```c
BOOL WINAPI IsDebuggerPresent()
```

Listing 3. IsDebuggerPresent ‘if statement’

```c
void Second_challenge()
{
    if( IsDebuggerPresent() )
    {
        printf("Running in a debugger");
        exit ( 0 );
    }
    else
    {
        printf("Not running in a debugger");
    }
}
```
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 GetProcAddress as these two API’s give us the ability to load any API.

Layer 3: GetProcAddress

‘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 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

```c
LPCWSTR get_Kernel32_string() {
    char buffer_Kernel32[9] = {'K', 'e', 'r', 'n', 'e', 'l', '3', '2', 0};
    wchar_t * wchar_string = new wchar_t[strlen(buffer_Kernel32) + 1];
    mbstowcs_s(wchar_string, buffer_Kernel32, _TRUNCATE);
    return wchar_string;
}
```

![Figure 7. Import Table](image7.png)

![Figure 8. Building Kernel32 as a Character Array](image8.png)
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:

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 GetProcAddress 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 lightweight and fast. It is also very easy to break.

The string I wanted to hide was copied 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 ciphertext 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.

---

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

```cpp
LPCSTR get_CreateFileA_string()
{
    char * buffer_CreateFileA = new char[12];
    buffer_CreateFileA[1] = ‘r’; //0x72
    buffer_CreateFileA[4] = ‘l’; //0x6c
    buffer_CreateFileA[6] = ‘i’; //0x69
    buffer_CreateFileA[7] = ‘t’; //0x74
    buffer_CreateFileA[8] = ‘C’; //0x67
    buffer_CreateFileA[9] = ‘e’; //0x65
    buffer_CreateFileA[10] = ‘e’; //0x65
    buffer_CreateFileA[12] = ‘\0’;
    return (LPCSTR)buffer_CreateFileA;
}
```

**Figure 9. Building CreateFileA as a Character Array**
REVERSE ENGINEERING TECHNIQUES

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

```c
unsigned char buffer_SecretCode[24] = {0xae, 0x92, 0x93, 0x89, 0xda, 0x91,
0x89, 0xda, 0x9e, 0x92, 0x9f, 0xda, 0xa9, 0x9f, 0x99, 0x88, 0x9f, 0x8e,
0xda, 0x9b, 0x95, 0x9e};
```

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

Listing 7. Calling CreateFileA dynamically using getProcAddress and LoadLibrary

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

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 GetProcAddress to locate it within the Kernel32 DLL. Next, we dynamically call the CreateFileA 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.

Layer 4: Anti-Virtualisation

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

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

Conclusion

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

EOIN WARD

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

Defeat Code Obfuscation

While Reverse Engineering?

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

If you have been in either of these situations, chances are you were dealing with obfuscated code or a packed binary. Not all is lost however, as getting around these methods of code protection is not impossible. However, all obfuscated code must be de-obfuscated before it can run. Keeping this in mind, it is possible to decrypt, 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...
base full of different packer signatures in order to identify what packing algorithm is in use.

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

**Manual Identification**

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

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

**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-
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.

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

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

**IDA Pro Roadmap**

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

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

**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).
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.

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).
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.
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...
How to Defeat Code Obfuscation While Reverse Engineering?

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.

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
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 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
Reverse Engineering – Shellcodes Techniques

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

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

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

The Various Shellcodes’ Techniques

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

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

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

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

For these reasons a “socket-reuse shellcode” should be used in highly secure systems as it does not create any new connections and therefore is harder to detect and block.
A different type of shellcode is the “download and execute shellcode”. This type of shellcode directs the target to download a certain executable file outside the target machine itself and to locate it locally as well as executing it. A variation of this type of shellcode downloads and loads a library.

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

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

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

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

**Introduction to MSFPAYLOAD Command**

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

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

```
# msfpayload -h
```

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

```
# msfpayload -l
```

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

In this case we chose the “shell_bind_tcp” payload as an example. Prior to the continuum of our action let us change our working directory to the Metasploit framework as so:
Once a payload was selected (in this case the `shell_bind_tcp` payload) there are two switches that are used most often when crafting the payload for the exploit you are creating.

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

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

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

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

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

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

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

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

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

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

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

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

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

```
$ msfpayload windows/exec S
Name: Windows Execute Command
Version: 5773
Platform: ["Windows"]
... Arch: x86
Needs Admin: No
Total size: 113
Provided by: vlad902
```

---

Figure 4. Specifying the Shellcode Options Data

Figure 5. Generating the Shellcode Using Msfpayload
Reverse Engineering – Shellcodes Techniques

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

Description:
Execute an arbitrary command

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

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

```
$ msfpayload windows/exec CMD=calc.exe X > pscalc.exe
```

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

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

```
$ ls -l pscalc.exe
```

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

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

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

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

```
$ ./pscalk.exe
```

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

ERAN GOLDSTEIN

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

Figure 2. IDA in Flow

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. Fol-
How to reverse the Code?

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

Figure 4. High Level Reversing Methodology

Figure 5. Identification of core driver module of fingerprint reader from System Manager
serve as a good starting point to our High Level understanding of device and the system operation. Typically, the next step includes examination of system interaction with the underlying operating system. Again, a number of tools exists for this purpose – well known tools such as Sysinternal tools, regmon, filemon and process explorer, provide great deal of possibility for exploring process interaction with registry, file system and the other processes respectively. Here, knowledge about DLL Mapping is the essential, which I highlighted in the beginning refer 003 – DLL Mapping.

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

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

Phase 2: Identifying Relevant Component Functions
This is the analysis of components identified in the previous phase to dig out function level information from the components. We will again need help of various tools for this. Here, we are interested in identifying named and exported functions and the virtual memory addresses for specified DLL files. DLL Export View can be used as presented in Figure 7.

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

Note
Keep documenting what you have so far obtained.

Phase 3: High Level Functional Analysis
This is nothing but the high level analysis of the function code that you should be able to obtain in

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

Figure 7. DLL Export Viewer to Identify Functions
How to Reverse the Code?

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 CISP, CEH, CEI, MCP, MCT, CRISC, and CobIT Foundation. Raheel is a certified instructor for ethical hacking boot camps.
How to Reverse Engineer dot net Assemblies

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Understanding MSIL**

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

Before going ahead, it’s wise to say that CLR executes the IL code allowing this way making operations and manipulating data, CLR does not handle directly the memory, it uses instead a stack, which is an abstract data structure which works accord-
**Table 1. Non-exhaustive IL instruction list**

<table>
<thead>
<tr>
<th>IL Instruction</th>
<th>Function</th>
<th>Byte representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>And</td>
<td>Computes the bitwise AND of two values and pushes the result onto the evaluation stack.</td>
<td>5F</td>
</tr>
<tr>
<td>Beq</td>
<td>Transfers control to a target instruction if two values are equal.</td>
<td>3B</td>
</tr>
<tr>
<td>Beq.s</td>
<td>Transfers control to a target instruction (short form) if two values are equal.</td>
<td>2E</td>
</tr>
<tr>
<td>Bge</td>
<td>Transfers control to a target instruction if the first value is greater than or equal to the second value.</td>
<td>3C</td>
</tr>
<tr>
<td>Bge.s</td>
<td>Transfers control to a target instruction (short form) if the first value is greater than or equal to the second value.</td>
<td>2F</td>
</tr>
<tr>
<td>Bge.Un</td>
<td>Transfers control to a target instruction if the first value is greater than the second value, when comparing unsigned integer values or unordered float values.</td>
<td>41</td>
</tr>
<tr>
<td>Bge.Un.s</td>
<td>Transfers control to a target instruction (short form) if the first value is greater than the second value, when comparing unsigned integer values or unordered float values.</td>
<td>34</td>
</tr>
<tr>
<td>Bgt</td>
<td>Transfers control to a target instruction if the first value is greater than the second value.</td>
<td>3D</td>
</tr>
<tr>
<td>Bgt.s</td>
<td>Transfers control to a target instruction (short form) if the first value is greater than the second value.</td>
<td>30</td>
</tr>
<tr>
<td>Bgt.Un</td>
<td>Transfers control to a target instruction if the first value is greater than the second value, when comparing unsigned integer values or unordered float values.</td>
<td>42</td>
</tr>
<tr>
<td>Bgt.Un.s</td>
<td>Transfers control to a target instruction (short form) if the first value is greater than the second value, when comparing unsigned integer values or unordered float values.</td>
<td>35</td>
</tr>
<tr>
<td>Ble</td>
<td>Transfers control to a target instruction if the first value is less than or equal to the second value.</td>
<td>3E</td>
</tr>
<tr>
<td>Ble.s</td>
<td>Transfers control to a target instruction (short form) if the first value is less than or equal to the second value.</td>
<td>31</td>
</tr>
<tr>
<td>Ble.Un</td>
<td>Transfers control to a target instruction if the first value is less than or equal to the second value, when comparing unsigned integer values or unordered float values.</td>
<td>43</td>
</tr>
<tr>
<td>Ble.Un.s</td>
<td>Transfers control to a target instruction (short form) if the first value is less than or equal to the second value, when comparing unsigned integer values or unordered float values.</td>
<td>36</td>
</tr>
<tr>
<td>Blt</td>
<td>Transfers control to a target instruction if the first value is less than the second value.</td>
<td>3F</td>
</tr>
<tr>
<td>Blt.s</td>
<td>Transfers control to a target instruction (short form) if the first value is less than the second value.</td>
<td>32</td>
</tr>
<tr>
<td>Blt.Un</td>
<td>Transfers control to a target instruction if the first value is less than the second value, when comparing unsigned integer values or unordered float values.</td>
<td>44</td>
</tr>
<tr>
<td>Blt.Un.s</td>
<td>Transfers control to a target instruction (short form) if the first value is less than the second value, when comparing unsigned integer values or unordered float values.</td>
<td>37</td>
</tr>
<tr>
<td>Bne.Un</td>
<td>Transfers control to a target instruction when two unsigned integer values or unordered float values are not equal.</td>
<td>40</td>
</tr>
<tr>
<td>Bne.Un.s</td>
<td>Transfers control to a target instruction (short form) when two unsigned integer values or unordered float values are not equal.</td>
<td>33</td>
</tr>
<tr>
<td>Br</td>
<td>Unconditionally transfers control to a target instruction.</td>
<td>38</td>
</tr>
<tr>
<td>Brfalse</td>
<td>Transfers control to a target instruction if value is false, a null reference (Nothing in Visual Basic), or zero.</td>
<td>39</td>
</tr>
<tr>
<td>Brfalse.s</td>
<td>Transfers control to a target instruction if value is false, a null reference, or zero.</td>
<td>2C</td>
</tr>
<tr>
<td>Brtrue</td>
<td>Transfers control to a target instruction if value is true, not null, or non-zero.</td>
<td>3A</td>
</tr>
<tr>
<td>Brtrue.s</td>
<td>Transfers control to a target instruction (short form) if value is true, not null, or non-zero.</td>
<td>2D</td>
</tr>
<tr>
<td>Br.s</td>
<td>Unconditionally transfers control to a target instruction (short form).</td>
<td>2B</td>
</tr>
<tr>
<td>Call</td>
<td>Calls the method indicated by the passed method descriptor.</td>
<td>28</td>
</tr>
<tr>
<td>Clt</td>
<td>Compares two values. If the first value is less than the second, the integer value 1 (int32) is pushed onto the evaluation stack; otherwise 0 (int32) is pushed onto the evaluation stack.</td>
<td>FE 04</td>
</tr>
<tr>
<td>Clt.Un</td>
<td>Compares the unsigned or unordered values value1 and value2. If value1 is less than value2, then the integer value 1 (int32) is pushed onto the evaluation stack; otherwise 0 (int32) is pushed onto the evaluation stack.</td>
<td>FE 03</td>
</tr>
<tr>
<td>Jmp</td>
<td>Exits current method and jumps to specified method.</td>
<td>27</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Opcode</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Ldarg</td>
<td>Loads an argument (referenced by a specified index value) onto the stack.</td>
<td>FE 09</td>
</tr>
<tr>
<td>Ldarga</td>
<td>Load an argument address onto the evaluation stack.</td>
<td>FE 0A</td>
</tr>
<tr>
<td>Ldarga.s</td>
<td>Load an argument address, in short form, onto the evaluation stack.</td>
<td>0F</td>
</tr>
<tr>
<td>Ldarg.0</td>
<td>Loads the argument at index 0 onto the evaluation stack.</td>
<td>02</td>
</tr>
<tr>
<td>Ldarg.1</td>
<td>Loads the argument at index 1 onto the evaluation stack.</td>
<td>03</td>
</tr>
<tr>
<td>Ldarg.2</td>
<td>Loads the argument at index 2 onto the evaluation stack.</td>
<td>04</td>
</tr>
<tr>
<td>Ldarg.3</td>
<td>Loads the argument at index 3 onto the evaluation stack.</td>
<td>05</td>
</tr>
<tr>
<td>Ldarg.s</td>
<td>Loads the argument (referenced by a specified short form index) onto the evaluation stack.</td>
<td>0E</td>
</tr>
<tr>
<td>Ldc.I4</td>
<td>Pushes a supplied value of type int32 onto the evaluation stack as an int32.</td>
<td>20</td>
</tr>
<tr>
<td>Ldc.I4.0</td>
<td>Pushes the integer value of 0 onto the evaluation stack as an int32.</td>
<td>16</td>
</tr>
<tr>
<td>Ldc.I4.1</td>
<td>Pushes the integer value of 1 onto the evaluation stack as an int32.</td>
<td>17</td>
</tr>
<tr>
<td>Ldc.I4.2</td>
<td>Pushes the integer value of -1 onto the evaluation stack as an int32.</td>
<td>15</td>
</tr>
<tr>
<td>Ldc.I4.s</td>
<td>Pushes the supplied int8 value onto the evaluation stack as an int32, short form.</td>
<td>1F</td>
</tr>
<tr>
<td>Ldstr</td>
<td>Pushes a new object reference to a string literal stored in the metadata.</td>
<td>72</td>
</tr>
<tr>
<td>Leave</td>
<td>Exits a protected region of code, unconditionally transferring control to a specific target instruction.</td>
<td>DD</td>
</tr>
<tr>
<td>Leave.s</td>
<td>Exits a protected region of code, unconditionally transferring control to a target instruction (short form).</td>
<td>DE</td>
</tr>
<tr>
<td>Mul</td>
<td>Multiplies two values and pushes the result on the evaluation stack.</td>
<td>5A</td>
</tr>
<tr>
<td>Mul.Ovf</td>
<td>Multiplies two integer values, performs an overflow check, and pushes the result onto the evaluation stack.</td>
<td>D8</td>
</tr>
<tr>
<td>Mul.Ovf.Un</td>
<td>Multiplies two unsigned integer values, performs an overflow check, and pushes the result onto the evaluation stack.</td>
<td>D9</td>
</tr>
<tr>
<td>Neg</td>
<td>Negates a value and pushes the result onto the evaluation stack.</td>
<td>65</td>
</tr>
<tr>
<td>Newobj</td>
<td>Creates a new object or a new instance of a value type, pushing an object reference (type O) onto the evaluation stack.</td>
<td>73</td>
</tr>
<tr>
<td>Not</td>
<td>Computes the bitwise complement of the integer value on top of the stack and pushes the result onto the evaluation stack as the same type.</td>
<td>66</td>
</tr>
<tr>
<td>Or</td>
<td>Compute the bitwise complement of the two integer values on top of the stack and pushes the result onto the evaluation stack.</td>
<td>60</td>
</tr>
<tr>
<td>Pop</td>
<td>Removes the value currently on top of the evaluation stack.</td>
<td>26</td>
</tr>
<tr>
<td>Rem</td>
<td>Divides two values and pushes the remainder onto the evaluation stack.</td>
<td>5D</td>
</tr>
<tr>
<td>Rem.Un</td>
<td>Divides two unsigned values and pushes the remainder onto the evaluation stack.</td>
<td>5E</td>
</tr>
<tr>
<td>Ret</td>
<td>Returns from the current method, pushing a return value (if present) from the caller’s evaluation stack onto the caller’s evaluation stack.</td>
<td>2A</td>
</tr>
<tr>
<td>Rethrow</td>
<td>Re throws the current exception.</td>
<td>FE 1A</td>
</tr>
<tr>
<td>Stind.I1</td>
<td>Stores a value of type int8 at a supplied address.</td>
<td>52</td>
</tr>
<tr>
<td>Stind.I2</td>
<td>Stores a value of type int16 at a supplied address.</td>
<td>53</td>
</tr>
<tr>
<td>Stind.I4</td>
<td>Stores a value of type int32 at a supplied address.</td>
<td>54</td>
</tr>
<tr>
<td>Stloc</td>
<td>Pops the current value from the top of the evaluation stack and stores it in a the local variable list at a specified index.</td>
<td>FE 0E</td>
</tr>
<tr>
<td>Sub</td>
<td>Subtracts one value from another and pushes the result onto the evaluation stack.</td>
<td>59</td>
</tr>
<tr>
<td>Sub.Ovf</td>
<td>Subtracts one integer value from another, performs an overflow check, and pushes the result onto the evaluation stack.</td>
<td>DA</td>
</tr>
<tr>
<td>Sub.Ovf.Un</td>
<td>Subtracts one unsigned integer value from another, performs an overflow check, and pushes the result onto the evaluation stack.</td>
<td>DB</td>
</tr>
<tr>
<td>Switch</td>
<td>Implements a jump table.</td>
<td>45</td>
</tr>
<tr>
<td>Throw</td>
<td>Throws the exception object currently on the evaluation stack.</td>
<td>7A</td>
</tr>
<tr>
<td>Xor</td>
<td>Computes the bitwise XOR of the top two values on the evaluation stack, pushing the result onto the evaluation stack.</td>
<td>61</td>
</tr>
</tbody>
</table>
REVERSE IT YOURSELF

ing to the “last in first out” basis, we can do two important things when talking about the stack: pushing and pulling data, by pushing data or items into the stack, any already present items just go further down in this stack, by pulling data or items from the stack, all present items move upward toward the beginning of it. We can handle only the topmost element of the stack.

Every IL instruction has its specific byte representation, I’ll try to introduce you a non exhaustive list of most important IL instructions, their functions and the actual bytes representation, and you are not supposed to learn them but use this list as a kind of reference: Table 1.

What this Means to a Reverse Engineer?
Nowadays there are plenty of tools that can “reflect” the source code of a dot NET compiled executable; a good and really widely used one is “Reflector” with which you can browse classes, decompile and analyze dot NET programs and components, it allows browsing and searching CIL instructions, resources and XML documentation stored in a dot NET assembly. But this is not the only tool we will need when reversing dot NET applications and we will need more than one article to cover all of them.

What Will you Learn From this First Article?
This first essay will show you how to deal with Reflector to reverse a simple practice oriented crack

![Crack Me's main form](image)

![Reflector's main window](image)

![Crack Me loaded on Reflector](image)

![You Can Expend the Target by Clicking the “+” Sign](image)

![Keep on Developing Tree and See What is Inside of this Crack Me](image)

![We Can See Actual Code Just by Clicking on the Method’s Name the Way We Get This](image)
How to reverse engineer dot net Assemblies

me I did the basic way, so I tried to simulate in this Crack Me a “real” software protection with disabled button, disabled feature and license check protection (Figure 2).

So basically we have to enable the first button having “Enable Me” caption, by clicking it we will get the “Save as…” button enabled which will let us simulate a file saving feature, we will see where the license check protection is trigged later in this article.

Open up Reflector, at this point we can configure Reflector via the language’s drop down box in the main toolbar, and select whatever language you may be familiar with, I’ll choose Visual Basic but the decision is up to you of course (Figure 3).

Load this Crack Me up into it (File > Open menu) and look anything that would be interest us. Technically, the crack me is analyzed and placed in a tree structure, we will develop nodes that interest us: Figure 4.

You can expend the target by clicking the “+” sign: Figure 5.

Keep on developing tree and see what is inside of this Crack Me: Figure 6.

Now we can see that our Crack Me is composed by References, Code and Resources.

- Code: this part contains the interesting things and everything we will need at this point is inside of HiddenNAME _ dotNET _ Reversing (which is a Namespace)
- References: is similar to “imports”, “includes” used in other PE files.
- Resources: for now this is irrelevant to us, but it this is similar to ones in other windows programs.

By expanding the code node we will see the following tree: Figure 8.

We can already clearly see some interesting methods with their original names which is great, we have only one form in this practice so let’s see what Form1_Load(object, EventArgs): void has to say, we can see actual code just by clicking on the method’s name the way we get this: Figure 7.

If you have any coding background you can guess with ease that “this.btnEnableMe.Enabled = false;” is responsible of disabling the component “btnEnableMe” which is in our case a button. At this point it’s important to see the IL and the byte representation of the code we are seeing, let’s switch to IL view and see: Listing 1. In the code above we can see some IL instruction worth of being explained (in the order they appear):

- ldarg.0 Pushes the value 0 to the method onto the evaluation stack.
- callvirt Calls the method get() associated with the object btnEnableMe.
- ldc.i4.0 Pushes 0 onto the stack as 32bits integer.
- callvirt Calls the method set() associated with the object btnEnableMe.

This says that the stack got the value 0 before calling the method set Enabled(bool), 0 is in general associated to “False” when programming, we will have to change this 0 to 1 in order to pass “True” as parameter to the method set Enabled(bool); the IL instruction that pushes 1 onto the stack is ldc.i4.1.

In a section above we knew that byte representation is important in order to know the exact location

<table>
<thead>
<tr>
<th>IL Instruction</th>
<th>Function</th>
<th>Byte representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ldc.i4.0</td>
<td>Pushes the integer value of 0 onto the evaluation stack as an int32.</td>
<td>16</td>
</tr>
<tr>
<td>Ldc.i4.1</td>
<td>Pushes the integer value of 1 onto the evaluation stack as an int32.</td>
<td>17</td>
</tr>
<tr>
<td>Callvirt</td>
<td>Call a method associated with an object.</td>
<td>6F</td>
</tr>
<tr>
<td>Ldarg.0</td>
<td>Load argument 0 onto the stack.</td>
<td>02</td>
</tr>
</tbody>
</table>

Figure 8. Crack Me’s nodes expanded
of the IL instruction to change and by what changing it, so by referring to the IL byte representation reference we have: Table 2.

We have to make a big sequence of bytes to search the IL instruction we want to change; we have to translate ldc.i4.0, callvirt, ldarg.0 and callvirt to their respective byte representation and make a byte search in a hexadecimal editor.


Things are getting more complicated and we will use some extra tools, I'm calling ILDasm! This tool is provided with dot NET Framework SDK, if you have installed Microsoft Visual Studio, you can find it in Microsoft Windows SDK folder, in my

Figure 9. ILDASM

Listing 1. IL code

```
.method private
    instance void Form1_Load ( 
        object sender,
        class [mscorlib]System.EventArgs e 
    ) cil managed
    { 
        // Method begins at RVA 0x1b44c 
        // Code size 29 (0x1d) 
        .maxstack 2 
        .locals init ( 
        )
        IL_0000: ldarg.0
    IL_0002: ldarg.0
    IL_0004: ldarg.0
    IL_0006: ldc.i4.0
    IL_0008: ldarg.0
    IL_000a: ldc.i4.0
    IL_000c: ret 
    } // end of method MainForm::Form1_Load
```
system ILDasm is located at C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\Bin (Figure 9). ILDasm can be easily an alternative tool to Reflector or ILSpy except the fact of having a bit less user friendly interface and no high level code translation feature. Anyway, once located open it and load our Crack Me into it (File -> Open) and expand trees as following: Figure 10.

ILDasm does not show byte representation by default, to show IL corresponding bytes you have to select View -> Show Bytes: Figure 11. Then double click on our concerned method (Form1_Load…) to get the IL code and corresponding bytes: Figure 12.

We have more information about IL instructions and their Bytes representations now, in order to use this amount of new information, you have to know that after “|” the low order byte of the number is stored in the PE file at the lowest address, and the high order byte is stored at the highest address, this order is called “Little Endian”.

What Does this Mean?
When looking inside Form1_Load( ) method using ILDasm, we have this:

IL_0006: /* 16 |
IL_0007: /* 6F | (0A) 000040
IL_000c: /* 02 |
IL_000d: /* 6F | (06) 000022

These Bytes are stored in the file this way: 166F4000000A026F22000006.

Back to Our Target
This sequence of bytes is quite good for making a byte search in a hexadecimal editor, in a real situation study; we may face an annoying problem which is finding more than one occurrence of our sequence. In this situation, instead of searching for bytes sequence we search for (or to better say “go to”) an offset which can be calculated.

An offset, also called relative address, is used to get to a specific absolute address. We have to calculate an offset where the instruction we want to change is located, referring to Figure 1, ILDasm

![Figure 10. Target loaded on ILDASM](image)

![Figure 11. Show bytes on ILDASM](image)

![Figure 12. ILDasm IL + bytes representations encoded Form1_Load() method](image)
and ILSpy indicate the Relative Virtual Address (RVA) at the line // Method begins at RVA 0x1b44c and in order to translate this to an offset or file location, we have to determine the layout of our target to see different sections and different offsets / sizes, we can use PEID or any other PE Tool, but I prefer to introduce you a tool that comes with Microsoft Visual C++ to view PE sections called “dumpbin” (If you do not have it, please refer to links on “References” section).

Dumpbin is a command line utility, so via the command line type “dumpbin -headers target_name.exe” (Figure 13).

By scrolling down we find interesting information:

SECTION HEADER #1
.text name
1C024 virtual size
2000 virtual address
1C200 size of raw data
400 file pointer to raw data
0 file pointer to relocation table
0 file pointer to line numbers
0 number of relocations
0 number of line numbers
60000020 flags
Code
Execute Read

Notice that the method Form1_Load() begins at RVA 0x1b44c (refer to Figure 1) and here the text section has a virtual size of 0x1c024 with a virtual address indicated as 0x2000 so our method must be within this section, the section containing our method starts from 0x400 in the main executable file, using these addresses and sizes we can calculate the offset of our method this way:

(Method RVA – Section Virtual Address) + File pointer to raw data; all values are in hexadecimal so using the Windows’s calculator or any other calculator that support hexadecimal operations we get: (1B44C – 2000) + 400 = 1984C (Figure 14).

So 0x1984C is the offset of the start of our method in our main executable, using any hexadecimal editor we can go directly to this location and what we
want change is few bytes after this offset considering the method header.

Going back to the sequence of bytes we got a bit ago 166F4000000A026F22000006 and going to the offset calculated before we get: Figure 15.

We want to change idc.i4.0 which is equal to 16 by idc.i4.1 which is equal to 17, let’s make this change and see what it reproduces (before doing any byte changes think always to make a backup of the original file) (Figure 16).

And yes our first problem is solved; we still have “Unregistered Crack Me” caption and still not tested “Save as…” button. Once we click on the button “Enable Me” we get the second one enabled which is supposed to be the main program feature. By giving it a try something bad happened: Figure 17.

Before saving, the program checks for a license, if not found it disables everything and aborts the saving process.

Protecting a program depends always on developer’s way of thinking, there is as mush ways to protect software as mush ways to break them. We can nevertheless store protections in “types” or “kinds” of protections, among this, there is what we call “license check” protections. Depending on how developer imagined how the protection must behave and how the license must be checked, the protection’s difficulty changes.

Let’s see again inside our target: Figure 18.

The method `btn_EnableMe_Click()` is trigged when we press the button “Enable Me” we saw this, `btn_About_Click()` is for showing the message box when cliquing on “About” button, then we still have two methods: `btn_EnableMe_Click()` and `checkLicence()` which seems to be interesting.

Let’s go inside the method `btn_EnableMe_Click()` and see what it has to tell: Figure 19.

By clicking on the button save, instead of saving directly, the Crack Me checks the “registration stat” of the program, this may be a kind of “extra protection”, which means, the main feature which is “saving file” is protected against “forced clicks”; The Crack Me checks if it is correctly registered before saving even if the “Save as…” button is enabled when the button “Enable Me” is clicked, well click on `checkRegStat()` to see its content: Figure 20.

Here is clear that there is a Boolean variable that changes, which is `isRegistered` and till now we made no changes regarding this. So if `isRegistered` is false (if (!this.isRegistered)…) the Crack Me makes a call to the `checkLicense()` method, we can see how `isRegistered` is initialized by clicking on `.ctor()` method: Figure 21.

`ctor()` is the default constructor where any member variables are initialized to their default values. Let’s go back and see what the method `checkLicense()` does exactly: Figure 22.

This is for sure a simple simulation of software “license check” protection, the Crack Me checks for the presence of a “lic.dat” file in the same directory of the application startup path, in other words, the Crack Me verifies if there is any “lic.dat” file in the same directory as the main executable file.
REVERSE IT YOURSELF

Listing 2. checkLicence() IL code

.method public instance void checkLicence() cil managed
{
    .maxstack 3
    .locals init (    [0] string str,

L_0005: ldstr "\\lic.dat"
L_000a: call string [mscorlib]System.String::Concat(string, string)
L_000f: stloc.0
L_0010: ldloc.0
L_0011: call bool [mscorlib]System.IO.File::Exists(string)
L_0016: brtrue.s L_006b
L_0018: ldstr "license file missing. Cannot save file."
L_001d: ldc.i4.s 0x10
L_001f: ldstr "License not found"
L_0029: pop
L_002a: ldarg.0
L_002b: ldc.i4.0
L_002c: stfld bool CrackMe2_HidenName_dotNET_Reversing.MainForm::isRegistered
L_0031: ldarg.0
L_0041: ldarg.0
L_0047: ldstr "Unregistered Crack Me"
L_0051: ldarg.0
L_0057: ldc.i4.0
L_005d: ldarg.0
How to reverse engineer dot net Assemblies

Well, technically at this point, we can figure out many solutions to make our program run fully, if we remove the call to the `checkLicense()` method, we will remove the same way the main feature which is saving, since it is done only once the checking is done (Figure 2).

If we force the `isRegistered` variable taking the value `True` by changing its initialization (Figure 3), we will lose the call to `checkLicense()` method that itself calls the main feature ("saving") as its only called if `isRegistered` is equal to false as seen here (refer to Figure 2):

```csharp
public void checkRegStat()
{
    this.LblStat.ForeColor = Color.Green;
    this.LblStat.Text = «Saving...»;
    if (!this.isRegistered)
    {
        this.checkLicence();
    }
}
```

We can alter the branch statement (if... else... endif, Figure 4) the way we can save only if the license file is *not found*.

We saw how to perform byte patching the "classical" way using offsets and hexadecimal editor, I’ll introduce you an easy way which is less technical and can save us considered time.

We will switch again to *Reflector* (please refer to previous parts of this series for further information), this tool can be extended using plug-ins, we will use *Reflexil*, a Reflector add-In that will allow us editing and manipulating IL code then saving the modifications to disk. After downloading Reflexil you need to install it; Open Reflector and go to Tools -> Add-ins (in some versions View -> Add-ins), a window will appear click on “Add...” and select “Reflexil.Reflector.dll”; Once you are done you can see your plug-in added to the Add-ins window which you can close.

Well basically we want to modify the Crack Me a way we get "File saved!", Switch the view to see IL code representation of this C# code: Listing 2.

I marked interesting instructions that need some explanations, so basically we have this:

```csharp
.method public instance void checkLicence() cil managed
    {
        .maxstack 3
        // (...)
        L_0011: call bool [mscorlib]System.IO.File::Exists(string)
        L_0016: brtrue.s L_006b
        L_0018: ldstr "license file missing. Cannot save file."
        (...)
        L_0069: br.s
        L_006b: ldarg.0
        (...)
        L_0081: ldstr «File saved !»
        (...)
    }
```

By referring to our IL instructions reference we have: Table 3.

The Crack Me makes a Boolean test regarding the license file presence (Figure 4), if file *found*...
it returns True, which means brtrue.s will jump to the line L_006b and the Crack Me will load “File saved!” string, otherwise it will go to the unconditional transfer control br.s that will transfer control to the instruction ret to get out from the whole method.

Table 3. IL Instructions

<table>
<thead>
<tr>
<th>IL Instruction</th>
<th>Function</th>
<th>Byte representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call</td>
<td>Calls the method indicated by the passed method descriptor.</td>
<td>28</td>
</tr>
<tr>
<td>Brtrue.s</td>
<td>Transfers control to a target instruction (short form) if value is true, not null, or non-zero.</td>
<td>2D</td>
</tr>
<tr>
<td>Br.s</td>
<td>Unconditionally transfers control to a target instruction (short form).</td>
<td>2B</td>
</tr>
<tr>
<td>Ret</td>
<td>Returns from the current method, pushing a return value (if present) from the caller’s evaluation stack onto the caller’s evaluation stack.</td>
<td>2A</td>
</tr>
</tbody>
</table>

Remember, we want our Crack Me to check for license file absence the way it returns True if file not found so it loads “File saved!” string. Let’s get back to reflector, now we have found the section of code we want to change (Figure 5), here comes the role of our add-in Reflexil, on the menu go to Tool -> Reflexil v1.x; This way you can get Reflexil panel under the source code or IL code shown by Reflector: Figure 23.

This is the IL code instruction panel of Reflexil as you can see, there are two ways you can make changes using this add-in but I’ll introduce for now only one, we will see how to edit instructions using IL code.

After analyzing the IL code above we know that we have to change the “if not found” by “if found” which means changing brtrue.s (Table 1) by its opposite, by returning to the IL code reference we find, brfalse.s: Branch to target if value is zero (false),
short form. This said, on Reflexil’s panel; find out where is the line we want to change: Figure 24.

Right click on the selected line -> Edit…, now you get a window that looks like: Figure 25.

Remove “brtrue.s” and type the new instruction “brfalse.s” then click “Update”, you see your modification done. To save “physically” this change, right click on the root of the disassembled Crack Me select Reflexilv1.x then Save as… (Figure 26).

This way we have a modified copy of our Crack Me, we have the “Enable Me” button enabled, by clicking on it we enable “Save as…” button and by clicking on this last we get our “File Saved!” message: Figure 27.

This article is at his end, it takes more time with more complex algorithms and protections but if you are able to get the IL code and can read it clearly you will with no doubt be able to bypass software protection.

SOUFIANE TAHIRI

Soufiane Tahiri is also an InfoSec Institute contributor, and computer security researcher from Morocco, specializing in reverse code engineering and software security. He is also founder of www.itsecurity.ma and practiced reversing for more several years. Dynamic and very involved, Soufiane is ready to catch any serious opportunity to be part of a workgroup. Contact Soufiane in whatever way works for you: Email: soufianetahiri@gmail.com Twitter: https://twitter.com/17s3cuni7y LinkedIn: http://ma.linkedin.com/in/souflanetahiri.

References
- Reflexil – http://sourceforge.net/projects/reflexil/
- Crack ME #2 – http://www.mediafire.com/?42vml4f1c6jy097

Figure 27. All problems are solved!
Reversing with Stack-Overflow and Exploitation

The theater of the Information security professional has changed drastically in the world of computing or digital World. So we are going to find the root. The keynote for secure the business is complete analysis of internal Business.

The prevalence of security holes in program and protocols, the increasing size and complexity of the internet, and the sensitivity of the information stored throughout have created a target-rich environment for our next generation advisory. The criminal element is applying advance technique to evade the software/tool security. So the Knowledge of Analysis is necessary. And that pin point is called “The Art Of Reverse Engineering”.

What is Reverse Engineering
Reverse engineering is the process of taking a compiled binary and attempting to recreate (or simply understand) the original way the program works. A programmer initially writes a program, usually in a high-level language such as C++ or Visual Basic (or God forbid, Delphi). Because the computer does not inherently speak these languages, the code that the programmer wrote is assembled into a more machine specific format, one to which a computer does speak. This code is called, originally enough, machine language. This code is not very human friendly, and often times requires a great deal of brain power to figure out exactly what the programmer had in mind.

Why Should you Know

- Military or commercial espionage. Learning about an enemy’s or competitor’s latest research by stealing or capturing a prototype and dismantling it. It may result in development of similar product.
- Improve documentation shortcomings. Reverse engineering can be done when documentation of a system for its design, production, operation or maintenance have shortcomings and original designers are not available to improve it. RE of software can provide the most current documentation necessary for understanding the most current state of a software system.
- Software Modernization. RE is generally needed in order to understand the ‘as is’ state of existing or legacy software in order to properly estimate the effort required to migrate system knowledge into a ‘to be’ state. Much of this may be driven by changing functional, compliance or security requirements.
- Product Security Analysis. To examine how a product works, what are specifications of its components, estimate costs and identify potential patent infringement.
- Bug fixing. To fix (or sometimes to enhance) legacy software which is no longer supported by its creators.
- Creation of unlicensed/unapproved duplicates.
- Academic/learning purposes. RE for learning purposes may be understand the key issues of an unsuccessful design and subsequently improve the design.
- Competitive technical intelligence. Understand what your competitor is actually doing, versus what they say they are doing.

What Should you Know?

The Stack: The stack is a piece of the process memory, a data structure that works LIFO (Last
2nd International Symposium in Grey-Hat Hacking
Submission deadline: June 30, 2013

Vulnerability Discovery, & Exploitation
Reverse Engineering & Obfuscation
Malware Creation, Analysis & Prevention
Embedded Systems Security
Hardware Vulnerabilities
Web Application Security

Network Exfiltration
Applied Cryptography & Cryptanalysis
Intrusion Detection & Prevention
Security & Privacy in Cloud, P2P
Penetration Testing
Disclosure & Ethics
Digital Forensics

GreHack
2nd edition
November 15, 2013
Grenoble, France

Program committee

(Mario Heiderich (Ruhr U. Bochum, Germany)
Pascal Lafourcade (VERIMAG, France)
Cédric Lauradoux (INRIA, France)
Pascal Malterre (CEA-DAM, France)
Laurent Mounier (VERIMAG, France)
Marie-Laure Potet (VERIMAG, France)
Paul Rascagneres (Malware.Lu, Luxembourg)
Sanjay Rawat (India)
Raphaël Rigo (ANSSI, France)
Nicolas Ruff (EADS Innovation Works, France)
Steven Seeley (Immunity, US)
Peter Van Eeckhoutte (CMU, US)
Manuel Egele (IF-UJF, France)
Philippe Elbaz-Vincent (ESIEA, France)
Eric Filioi (Thailand)
The Grugq

Don Alloun (Intel, Israel)
Ruo Ando (Kudelski Sec., Switz.)
Jean-Philippe Aumasson (Google, US)
Elie Bursztein (CEA-DAM, France)
Fabrice Desclaix (UCSB, US)
Adam Doupe (LIG, France)
Fabien Duchène (Veracode, US)
Chris Eng (Corelan, Belgium)
Peter Van Eeckhoutte (CMU, US)
Manuel Egele (IF-UJF, France)
Philippe Elbaz-Vincent (ESIEA, France)
Eric Filioi (Thailand)
The Grugq

www.grehack.org
REVERSE IT YOURSELF

in first out). A stack gets allocated by the OS, for each thread (when the thread is created). When the thread ends, the stack is cleared as well. The size of the stack is defined when it gets created and doesn’t change. Combined with LIFO and the fact that it does not require complex management structures/mechanisms to get managed, the stack is pretty fast, but limited in size.

LIFO means that the most recent placed data (result of a PUSH instruction) is the first one that will be removed from the stack again. (by a POP instruction).

Each and every software has predefined subroutine or sub function that is called dynamically in the program, means.

When a function/subroutine is entered, a stack frame is created. This frame keeps the parameters of the parent procedure together and is used to pass arguments to the subrouting. The current location of the stack can be accessed via the stack pointer (ESP), the current base of the function is contained in the base pointer (EBP) (or frame pointer).

The CPU’s general purpose registers (Intel, x86) are:

• EAX: accumulator: used for performing calculations, and to store return values from function calls. Basic operations such as add, subtract, compare use this general-purpose register.
• EBX: base (does not have anything to do with base pointer). It has no general purpose and can be used to store data.
• ECX: counter: used for iterations. ECX counts downward.
• EDX: data: this is an extension of the EAX register. It allows for more complex calculations (multiply, divide) by allowing extra data to be stored to facilitate those calculations.
• ESP: stack pointer
• EBP: base pointer
• ESI: source index: holds location of input data
• EDI: destination index: points to location of where result of data operation is stored
• EIP: instruction pointer

So The Espinosa tools are used for complete go through or analytic of software which are listed below.

What kinds of tools are used?
There are many different kinds of tools used in reversing. Many are specific to the types of protection that must be overcome to reverse a binary. There are also several that just make the reverser’s life easier. And then some are what I consider the ‘staple’ items- the ones you use regularly. For the most part, the tools fit into a couple categories:

Disassemblers
Disassemblers attempt to take the machine language codes in the binary and display them in a friendlier format. They also extrapolate data such as function calls, passed variables and text strings. This makes the executable look more like human-readable code as opposed to a bunch of numbers strung together. There are many disassemblers out there, some of them specializing in certain things (such as binaries written in Delphi). Mostly it comes down to the one your most comfortable with. I invariably find myself working with IDA.

Debuggers
Debuggers are the bread and butter for reverse engineers. They first analyze the binary, much like a disassembler Debuggers then allow the reverser to step through the code, running one line at a time and investigating the results. This is invaluable to discover how a program works. Finally, some debuggers allow certain instructions in the code to be changed and then run again with these changes in place. Examples of debuggers are Windbg, Immunity Debugger and Ollydbg. I almost uses Immunity debugger and ollydbg.

REAL ATTACK
Before start this we are using the following vulnerability which have stack based overflow and we will reversely analyze that file and will exploit for our cause.

• Vulnerability item-RM To MP3 Converter
• BOX-Windows xp SP2/SP3 (I m using sp3)
• Tool: Ollydbg, Immunity Debugger
• Backtrack Machine/Machine with metasploit installed

First of all create a python script with predefined written data into buffer and create a .m3u file. Open this file in rm to mp3 converter.so the file/software will crash due to stack overflow. In the image I loaded a script with 30,000 bytes of data into mp3 file which will get crash on the 2nd image or buffer overflow causes. This is the program (Figure 1).

#!/usr/bin/python
filename = ’30000.m3u’
buffer = “\x41” * 30000
file = open(filename,'w')
print"Done!"
file.close()

So the below diagram is the crash file of rm to mp3 (Figure 2).

**The Debugger**

In order to see the state of the stack (and value of registers such as the instruction pointer, stack pointer etc.), we need to hook up a debugger to the application, so we can see what happens at the time the application runs (and especially when it dies).

There are many debuggers available for this purpose. The two debuggers I use most often are ollydbg, and Immunity’s Debugger (Figure 3 and Figure 4).

This GUI shows the same information, but in a more...errr... graphical way. In the upper left corner, you have the CPU view, which shows assembly instructions and their opcodes (the window is empty because EIP currently points at 41414141 and that's not a valid address). In the upper right windows, you can see the registers. In the lower left corner, you see the memory dump of 00446000 in this case. In the lower right corner, you can see the contents of the stack (so the contents of memory at the location where ESP points at).

Anyways, in both cases, we can see that the instruction pointer contains 41414141, which is the hexadecimal representation for AAAA. And the position is called "offset" value.

**Checking The EIP Position**

- From the result we know that the ESP and EIP register is overwritten.
- We don’t know where the ESP and EIP register overwritten, so we make the structured string using pattern_create.rb to know the location the register overwritten.

Backtrack has the solution like metasploit.so we will use

```
root@dmitry-TravelMate-5730:/opt/metasploit3/msf3/tools# ./pattern_create.rb 30000
```

we will got a generation and we will again create m3u file and run to the rm to mp3 converter to see the result (Figure 5).

Again Creating a m3u file with the following generation to check EIP Location and we have to open
in rm to mp3 converter (Figure 6 and Figure 7). So we will get a value which is nearer between 5792 to 26072. see the picture below. so in that location EIP Value is written. EIP sits between 25000 and 30000.

For that reason I have taken 30000 byte of data to see what happens to the data or program. see the picture below you will understand (Figure 8).

In the above screen I used two command to check the EIP AND ESP Location and fortunately I have not get any value for 2nd option and I got 1st value 5792 for command, because I have taken the beyond bytes of data.

### Finding JMP ESP And Memory Location

Before try to exploit we should know the exact memory location, JMP, ESP Location so that our exploit will work perfectly.

Ollydbg: go to view-executable modules and search for Shell 32 modules and right click on shell32, view JMP ESP Command and location.

Same procedure will be applied for Immunity Debugger. For More Information See the Figure 9 Analysis in Immunity Debugger see Figure 10. Analysis in Ollydbg.
Creating Our Own Exolit and Let Die The Application

As we know creating and building exploit there is great contribution towards Metasploit Built-in Payload generator and encoders. so we will use one of them for our Development of exploit.

So we will use Encoder: x86/shikata_ga_nai which is a good encoders for generating the payload which can be available in just writing msfconsole-show payloads-use payload(in this case bind_tcp)-show encoder-generate encoder

And we will use a program namely calculator in windows machine to boom the application. For That we have to run a perl script behind it and open in rm to mp3 converter (Figure 11).

So we will add the encoder to our final exploit to run calculator on “rm to mp3 converter” to get buffer overflow.

And Exactly we add the location of memory as well as EIP ESP Location into exploit of our code to get into buffer.

Again Create Vulnerable .m3u file and run in “rm to mp3 converter” to see the calculator and to analyze in debugger either we have to open in immunity debugger or ollydbg debugger and analyze location where EIP AND ESP Overwritten (Figure 12 and Figure 13).

Application Boom to Calculator Application.

You can create the .m3u file and reverse connect to your shell some tool like nmap.netcat etc...

BIKASH DASH

Bikash Dash over 3 years of experience it security, malware analysis, Reverse engineering, Firewall security, Trojan Analysis, PE Auditor, Assembly Programming Cyber crime analyst, threat management, Honeypot analysis, Speaker.

Current Position: Ethical Hacker At Innobuzz Knowledge solution

Contact-Bikash Dash
Web: www.whitehatsecurity.in
Email: bikash.nit.12@gmail.com
How to Reverse Engineer?

If you are a programmer, software developer, or just tech savvy, then you should have heard about reverse engineering and know both its good and evil side. Just in case, here is a brief introduction for those who don’t know what it is.

In this article, we are going to talk about RCE, also known as reverse code engineering. Reverse code engineering is the process where the code and function of a program is modified, or may you prefer: reengineered without the original source code. For example, if a software programmer has created a program with a bug, does not release a fix, then an experienced end user can reverse engineer the application and fix the bug for everyone using the program. Sounds helpful doesn’t it?

That’s because we only touched the tip of the iceberg; the road of reverse engineering is a long one and the end leads to somewhere dark and illegal. Why you wonder? Because, by that logic, computer users can modify the code of any program, alter licensing features of a commercial product and remove critical features to their own liking. For example, a software such as Photoshop that requires you to buy a serial key to register and use it, can be reverse engineered to either extract a valid key or just to remove the whole serial system altogether. This is illegal and these people who reverse engineer applications illegally, known as crackers or hackers, have encountered legal issues since the first software was released. Teams also dedicate themselves to this activity, but to this present day, most have been arrested or have ‘voluntarily shutdown’.

So how exactly does one reverse engineer? What tool do you need to do so? Read on because we are getting there!

Reverse Engineering

Reverse engineering has drawn a lot of attention to itself in the past few years, especially when hacked programs are released to the general public, and spread across websites that dedicate themselves to distributing them. Though it is mainly used for sinister purposes, reverse engineering can also be used for good, such as removing bugs, fixing crashes and so on. The next paragraph will give you the brief on how programs (EXE files) are created.

The process of making a program is quite straightforward. 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 into 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 into 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
How to reverse engineer?

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 program-
ming 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 into a Windows executable. However, as VB compiles to high level, interpreted code, as opposed to C++’s native code, it can be easily disassembled. 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 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.searchcio-midmarket.techtarget.com/definition/reverse-engineering](http://www.searchcio-midmarket.techtarget.com/definition/reverse-engineering)
- [www.youtube.com/watch?v=vGBFEDsIWhQ](http://www.youtube.com/watch?v=vGBFEDsIWhQ)
- [www.securitytube.net/video/1363](http://www.securitytube.net/video/1363)

**LORENZO XIE**

Lorenzo Xie is the owner of XetoWare.com and AceVideoConverter.com. He also works with several other software companies and specializes in Windows software development. You can contact him directly at Lorenzo@xetoware.com.
A Cyber criminal can target and breach your organization's perimeter in less than a second from anywhere in the world ...

Are You Prepared?

ANRC delivers advanced cyber security training, consulting, and development services that provide our customers with peace of mind in an often confusing cyber security environment. ANRC's advanced security training program utilizes an intensive hands-on laboratory method of training taught by subject matter experts to provide Information Security professionals with the knowledge and skills necessary to defend against today's cyber-attacks and tomorrow's emerging threats.

ANRC's consulting and development services leverage team member knowledge and experience gained in the trenches while securing critical networks in the U.S. Department of Defense and large U.S. corporations. ANRC tailors these services to deliver computer security solutions specific to the needs of the customer's operational environment. Our approach emphasizes a close relationship with our clients as an integral part of our service. We believe we're all in the security battle together, and we view our customers as key members of our team in the fight.

TRAINING :: CONSULTING :: SOLUTIONS  www.anrc-services.com
Write your own Debugger

Do you want to write your own debugger? … Do you have a new technology and see the already known products like OllyDbg or IDA Pro don’t have this technology? … Do you write plugins in OllyDbg and IDA Pro but you need to convert it into a separate application? … This article is for you.

In this article, I’m going to teach you how to write a full functional debugger using the Security Research and Development Framework (SRDF) … how to disassemble instructions, gather Process Information and work with PE Files … and how to set breakpoints and work with your debugger.

Why Debugging?
Debugging is usually used to detect application bugs and traces its execution … and also, it’s used in reverse engineering and analyzing application when you don’t have the source code of this application.

Reverse engineering is used mainly for detecting vulnerabilities, analyzing malware or cracking applications. We will not discuss in this article how to use the debugger for these goals … but we will describe how to write your debugger using SRDF… and how you can implement your ideas based on it.

Security Research and Development Framework
This is a free open source Development Framework created to support writing security tools and malware analysis tools. And to convert the security researches and ideas from the theoretical approach to the practical implementation.

This development framework created mainly to support the malware field to create malware analysis tools and anti-virus tools easily without reinventing the wheel and inspire the innovative minds to write their researches on this field and implement them using SRDF.

In User-Mode part, SRDF gives you many helpful tools … and they are:
- Assembler and Disassembler
- x86 Emulator
- Debugger
- PE Analyzer
- Process Analyzer (Loaded DLLs, Memory Maps … etc)
- MD5, SSDeep and Wildlist Scanner (YARA)
- API Hooker and Process Injection
- Backend Database, XML Serializer
- And many more

In the Kernel-Mode part, it tries to make it easy to write your own filter device driver (not with WDF and callbacks) and gives an easy, object oriented (as much as we can) development framework with these features:
- Object-oriented and easy to use development framework
- Easy IRP dispatching mechanism
- SSDT Hooker
- Layered Devices Filtering
- TDI Firewall
- File and Registry Manager
- Kernel Mode easy to use internet sockets
- Filesystem Filter

Still the Kernel-Mode in progress and many features will be added in the near future.

Gather Information About Process
If you decided to debug a running application or you start an application for debugging. You need
to gather information about this process that you want to debug like:

- Allocated Memory Regions inside the process
- The Application place in its memory and the size of the application in memory
- Loaded DLLs inside the application's memory
- Read a specific place in memory
- Also, if you need to attach to a process already running ... you will also need to know the Process Filename and the commandline of this application

**Begin the Process Analysis**

To gather the information about a process in the memory, you should create an object of cProcess class given the ProcessId of the process that you need to analyze.

```cpp
cProcess myProc(792);
```

If you only have the process name and don’t have the process id, you can get the process Id from the ProcessScanner in SRDF like this:

```cpp
cProcessScanner ProcScan;
```

And then get the hash of process names and Ids from ProcessList field inside the cProcessScanner Class ... and this item is an object of cHash class.

- cHash class is a class created to represent a hash from key and value ... the relation between them are one-to-many ... so each key could have many values.

- In our case, the key is the process name and the value is the process id. You could see more than one process with the same name running on your system. To get the first ProcessId for a process “Explorer.exe” for example ... you will do this:

```cpp
ProcScan.ProcessList["explorer.exe"]
```

This will return a cString value includes the ProcessId of the process. To convert it into integer, you will use atoi() function ... like this:

```cpp
atoi(ProcScan.ProcessList["explorer.exe"])""
```

**Getting Allocated Memory**

To get the allocated memory regions, there’s a list of memory regions named MemoryMap the type of this item is cList.

- cList is a class created to represent a list of buffers with fixed size or array of a specific struct. It has a function named GetNumberOfItems and this function gets the number of items inside the list. In the following code, we will see how to get the list of Memory Regions using cList Functions (Listing 1).

```cpp
struct MEMORY_MAP
{
  DWORD Address;
  DWORD Size;
  DWORD Protection;
};
```

In the previous code, we loops on the items of MemoryMap List and we get every memory region’s address and size.

**Getting the Application Information**

To get the application place in memory ... you will simply get the Imagebase and SizeOfImage fields inside cProcess class like this:

- As you see, we get the most important information about the process and its place in memory (Imagebase) and the size of it in memory (SizeOfImage).

```cpp
Listing 1. How to Get the List of Memory Regions Using cList Functions

```cpp
for(int i=0; i<(int)(myProc->MemoryMap.GetNumberOfItems()) ;i++)
{
  cout<<"Memory Address "<< ((MEMORY_MAP*)myProc->MemoryMap.GetItem(i))->Address;
  cout << " Size: " <<hex<<((MEMORY_MAP*)myProc->MemoryMap.GetItem(i))->Size <<endl;
}
```

```cpp
Listing 2. cProcess Class

```cpp
cout<<"Process: "<< myProc->processName<<endl;
cout<<"Process Parent ID: "<< myProc->ParentID <<endl;
cout<< "Process Command Line: "<< myProc->CommandLine << endl;
cout<<"Process PEB:\t"<< myProc->ppeb<<endl;
cout<<"Process ImageBase:\t"<<hex<<(myProc->ImageBase)<<endl;
cout<<"Process SizeOfImageBase:\t"<<dec<<(myProc->SizeOfImage)" bytes"<<endl;
```

```cpp
```
Loaded DLLs and Modules
The loaded Modules is a cList inside cProcess class with name modulesList and it represents an array of struct MODULE_INFO and it’s like this: Listing 3.

To get the loaded DLLs inside the process, this code represents how to get the loaded DLLs: Listing 4.

Read, Write and Execute on the Process
To read a place on the memory of this process, the cProcess class gives you a function named Read(...) which allocates a space into your memory and then reads the specific place in the memory of this process and copies it into your memory (the new allocated place in your memory).

DWORD Read(DWORD startAddress,DWORD size)

For writing to the process, you have another function name Write and it’s like this:

DWORD Write (DWORD startAddressToWrite ,DWORD buffer ,DWORD sizeToWrite)

This function takes the place that you would to write in, the buffer in your process that contains the data you want to write and the size of the buffer.

If the startAddressToWrite is null ... Write() function will allocate a place in memory to write on and return the pointer to this place.

Listing 3. "MODULE_INFO"
struct MODULE_INFO
{
    DWORD moduleImageBase;
    DWORD moduleSizeOfImage;
    cString* moduleName;
    cString* modulePath;
};

Listing 4. How to Get the Loaded DLLs
for (int i=0 ; i<(int)( myProc->modulesList. GetNumberOfItems()) ;i++)
{
    cout<<"Module \\
        ImageBase:  \\
    \

To only allocate a space inside the process you can use Allocate() function to allocate memory inside the process and it’s like that:

Allocate(DWORD preferredAddress,DWORD size)

You have also the option to execute a code inside this process by creating a new thread inside the process or inject a DLL inside the process using these functions

DWORD DllInject(cString DLLFilename)
DWORD CreateThread (DWORD addressToFunction , DWORD addressToParameter)

And these functions return the ThreadId for the newly created thread.

Debugging an Application
To write a successful debugger, you need to include these features in your debugger:

1. Could Attach to a running process or open an EXE file and debug it
2. Could gather the register values and modify them
3. Could Set Int3 Breakpoints on specific addresses
4. Could Set Hardware Breakpoints (on Read, Write or Execute)
5. Could Set Memory Breakpoints (on Read, Write or Execute on a specific pages in memory)
6. Could pause the application while running
7. Could handle events like exceptions, loading or unloading dlls or creating or terminating a thread.

In this part, we will describe how to do all of these things easily using SRDF’s Debugger Library.

Open Exe File and Debug … or Attach to a process
To Open an EXE File and Debug it:

cDebugger* Debugger = new cDebugger(“C:\upx01.exe”);

Or with command line:

cDebugger* Debugger = new cDebugger(“C:\upx01.exe”,“xxxx”);

if the file opened successfully, you will see Is-Found variable inside cDebugger class set to TRUE. If any problems happened (file not found
Write your own Debugger

or anything) you will see it equal FALSE. Always check this field before going further.

If you want to debug a running process ... you will
create a cProcess class with the ProcessId you
want and then attach the debugger to it:

cDebugger* Debugger = new cDebugger(myProc);

to begin running the application ... you will use
function Run() like this:

Debugger->Run();

Or you can only run one instruction using function
Step() like this:

Debugger->Step();

This function returns one of these outputs (until
now, could be expanded):

1. DBG_STATUS_STEP
2. DBG_STATUS_HARDWARE_BP
3. DBG_STATUS_MEM_BREAKPOINT
4. DBG_STATUS_BREAKPOINT
5. DBG_STATUS_EXITPROCESS
6. DBG_STATUS_ERROR
7. DBG_STATUS_INTERNAL_ERROR

If it returns DBG_STATUS_ERROR, you can check the
ExceptionCode Field and the debug_event Field to
go more information.

Getting and Modifying the Registers:
To get the registers from the debugger ... you have
all the registers inside the cDebugger class like:

• Reg[0 → 7]
• Eip
• EFlags
• DebugStatus → DR7 for Hardware Breakpoints

To update them, you can modify these variables
and then use function UpdateRegisters() after the
modifications to take effect.

Setting int3 Breakpoint
The main Debuggers' breakpoint is the instruc-
tion “int3” which converted into byte 0xCC in bi-
nary (or native) form. The debuggers write int3
byte at the beginning of the instruction that they
need to break into it. After that, when the execu-
tion reaches this instruction, the application stops and return to the debugger with exception: STATUS_BREAKPOINT.

To set an Int3 breakpoint, the debugger has a
function named SetBreakpoint(...) like this:

Debugger->SetBreakpoint(0x004064AF);

You can set a UserData For the breakpoint
like this:

DBG_BREAKPOINT* Breakpoint = GetBreakpoint(DWORD
Address);

And the breakpoint struct is like this:

struct DBG_BREAKPOINT
{
    DWORD Address;
    DWORD UserData;
    BYTE OriginalByte;
    BOOL IsActive;
    WORD wReserved;
};

So, you can set a UserData for yourself ... like
pointer to another struct or something and set it
for every breakpoint.

When the debugger's Run() function returns
“DBG_STATUS_BREAKPOINT” you can get the
breakpoint struct DBG_BREAKPOINT by the Eip and get
the UserData from inside ... and manipulate your
information about this breakpoint.

Also, you can get the last breakpoint by using a
Variable in cDebugger Class named LastBreakpoint
like this:

cout << "LastBp: " << Debugger->LastBreakpoint << "\n";

To Deactivate the breakpoint, you can use func-
tion RemoveBreakpoint(...) like this:

Debugger->RemoveBreakpoint(0x004064AF);

Setting Hardware Breakpoints
Hardware breakpoints are breakpoints based on
debg registers in the CPU. These breakpoints
could stop on accessing or writing to a place
in memory or it could stop on execution on an
address. And you have only 4 available break-
points only. You must remove one if you need to
add more.

These breakpoints don't modify the binary of the
application to set a breakpoint as they don't add
int3 byte to the address to stop on it. So they could
be used to set a breakpoint on packed code to
break while unpacked.
To set a hardware breakpoint to a place in the memory (for access, write or execute) you can set it like this:

```cpp
Debugger->SetHardwareBreakpoint(0x00401000, DBG_BP_TYPE_WRITE, DBG_BP_SIZE_2);
Debugger->SetHardwareBreakpoint(0x00401000, DBG_BP_TYPE_CODE, DBG_BP_SIZE_4);
Debugger->SetHardwareBreakpoint(0x00401000, DBG_BP_TYPE_READWRITE, DBG_BP_SIZE_1);
```

For code only, use DBG_BP_SIZE_1 for it. But the others, you can use size equal to 1 byte, 2 bytes or 4 bytes.

This function returns false if you don't have a spare place for your breakpoint. So, you will have to remove a breakpoint for that.

To remove this breakpoint, you will use the function `RemoveHardwareBreakpoint(...)` like this:

```cpp
Debugger->RemoveHardwareBreakpoint(0x004064AF);
```

### Setting Memory Breakpoints

Memory breakpoints are breakpoints rarely to see. They are not exactly in OllyDbg or IDA Pro but they are good breakpoints. It's similar to OllyBone.

These breakpoints are based on memory protections. They set read/write place in memory to read only if you set a breakpoint on write. Or set a place in memory to no access if you set a read/write breakpoint and so on.

This type of breakpoints has no limits but it set a breakpoint on a memory page with size 0x1000 bytes. So, it's not always accurate. And you have only the breakpoint on Access and the Breakpoint on write.

To set a breakpoint you will do like this:

```cpp
Debugger->SetMemoryBreakpoint(0x00401000, 0x2000, DBG_BP_TYPE_WRITE);
```

When the `Run()` function returns DBG_STATUS_MEM_BREAKPOINT so a Memory Breakpoint is triggered. You can get the accessed memory place (exactly) using `cDebugger` class variable: `LastMemoryBreakpoint`.

You can also set a UserData like Int3 breakpoints by using `GetMemoryBreakpoint(...)` with any pointer inside the memory that you set the breakpoint on it (from Address to (Address + Size)). And it returns a pointer to struct which describe the memory breakpoint and you can add your user data in it (Listing 5).

You can see the real memory protection inside and you can set your user data inside the breakpoint.

To remove a breakpoint, you can use `RemoveMemoryBreakpoint(Address)` to remove the breakpoint.

### Pausing the Application

To pause the application while running, you need to create another thread before executing `Run()` function. This thread will call to `Pause()` function to pause the application. This function will call to `SuspendThread` to suspend the debugged thread inside the debuggee process (The process that you are debugging).

To resume again, you should call to `Resume()` and then call to `Run()` again.

You can also terminate the debuggee process by calling `Terminate()` function. Or, if you need to exit the debugger and makes the debuggee process continues, you can use `Exit()` function to detach the debugger.

### Handle Events

To handle the debugger events (Loading new DLL, Unload new DLL, Creation of a new Thread and so on), you have 5 functions to get notified with these events and they are:

1. `DLLLoadedNotifyRoutine`
2. `DLLUnloadedNotifyRoutine`
3. `ThreadCreatedNotifyRoutine`
4. `ThreadExitNotifyRoutine`
5. `ProcessExitNotifyRoutine`

You will need to inherit from `cDebugger` Class and override these functions to get notified on them.

To get information about the Event, you can information from `debug_event` variable (see Figure 1).
Write your own Debugger

DOS MZ header
DOS stub
PE header
Section table
Section 1
Section 2
Section ...
Section n

Figure 1. PE File Format

We will go through the PE Headers (EXE Headers) and how you could get information from it and from cPEFile class in SRDF (the PE Parser).

The EXE File begins with “MZ” characters and the DOS Header (named MZ Header). This DOS Header is for a DOS Application at the beginning of the EXE File.

This DOS Application is created to say “it’s not a win32 application” if it runs on DOS.

The MZ Header contains an offset (from the beginning of the File) to the beginning of the PE Header. The PE Header is the Real header of the Win32 Application.

PE Header

Signature: PE,0,0
File Header
Optional Header
Data Directory

Figure 2. PE Header

It begins with Signature “PE” and 2 null bytes and then 2 Headers: File Header and Optional Header.

To get the PE Header in the Debugger, the cPEFile class includes the pointer to it (in a Memory Mapped File of the Process Application File) like this:

cPEFile* PEFile = new cPEFile(argv[1]);
image_header* PEHeader = PEFile->PEHeader;

The File Header contains the number of section (will be described) and contains the CPU architecture and model number that this application should run into ... like Intel x86 32-Bits and so on. Also, it includes the size of Optional Header (the Next Header) and includes The Characteristics of the Application (EXE File or DLL).

The Optional Header contains the Important Information about the PE as you see in the Table 1.

Table 1. The Optional Header Contains the Important Information about the PE

<table>
<thead>
<tr>
<th>Field</th>
<th>Meanings</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddressOfEntryPoint</td>
<td>The Beginning of the Execution</td>
</tr>
<tr>
<td>ImageBase</td>
<td>The Start of the PE File in Memory (default)</td>
</tr>
<tr>
<td>SectionAlignment</td>
<td>Section Alignment in Memory while mapping</td>
</tr>
<tr>
<td>FileAlignment</td>
<td>Section Alignment in Harddisk (~ one sector)</td>
</tr>
<tr>
<td>MajorSubsystemVersion</td>
<td>The win32 subsystem version</td>
</tr>
<tr>
<td>MinorSubsystemVersion</td>
<td></td>
</tr>
<tr>
<td>SizeOfImage</td>
<td>The Size of the PE File in Memory</td>
</tr>
<tr>
<td>SizeOfHeaders</td>
<td>Sum of All Header sizes</td>
</tr>
<tr>
<td>Subsystem</td>
<td>GUI, Console, driver or others</td>
</tr>
<tr>
<td>DataDirectory</td>
<td>Array of pointers to important Headers</td>
</tr>
</tbody>
</table>

To get this Information from the cPEFile class in SRDF ... you have the following variables inside the class: Listing 6.

DataDirectory are an Array of pointers to other Headers (optional Headers ... could be found or could the pointer be null) and the size of the Header. It Includes:

- Import Table: importing APIs from DLLs
- Export Table: exporting APIs to another Apps
- Resource Table: for icons, images and others
- Relocables Table: for relocating the PE File (loading it in a different place ... different from Imagebase)

Listing 6. Following Variables Inside the Class

```cpp
bool FileLoaded;
image_header* PEHeader;
DWORD Magic;
DWORD Subsystem;
DWORD Imagebase;
DWORD SizeOfImage;
DWORD Entrypoint;
DWORD FileAlignment;
DWORD SectionAlignment;
WORD DataDirectories;
short nSections;
```
We include the parser of Import Table … as it includes an Array of All Imported DLLs and APIs like this: Listing 7. After the Headers, there are the section headers. The application File is divided into section: section for code, section for data, section for resources (images and icons), section for import table and so on.

Sections are expandable … so you could see its size in the Harddisk (or the file) is smaller than what is in the memory (while loaded as a process) … so the next section place will be different from the Harddisk and the memory.

The address of the section relative to the beginning of the file in memory while loaded as a process is named RVA (Relative virtual address) … and the address of the section relative to the beginning of the file in the Harddisk is named Offset or PointerToRawData (Table 2 and Listing 8).

Table 2. The Information that the Section Header Gives

<table>
<thead>
<tr>
<th>Field</th>
<th>Meanings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>The Section Name</td>
</tr>
<tr>
<td>VirtualAddress</td>
<td>The RVA address of the section</td>
</tr>
<tr>
<td>VirtualSize</td>
<td>The size of Section (in Memory)</td>
</tr>
<tr>
<td>SizeOfRawData</td>
<td>The Size of Section (in Harddisk)</td>
</tr>
<tr>
<td>PointerToRawData</td>
<td>The pointer to the beginning of file (Harddisk)</td>
</tr>
<tr>
<td>Characteristics</td>
<td>Memory Protections (Execute, Read, Write)</td>
</tr>
</tbody>
</table>

The Real Address is the address to the beginning of this section in the Memory Mapped File. Or in other word, in the Opened File.

To convert RVA to Offset or Offset to RVA … you can use these functions:

Listing 8. You Can Manipulate the Section in cPEFile Class Like This

cout << PEFile->nSections << "\n"; for (int i=0;i< PEFile->nSections;i++)
{
    cout << PEFile->Section[i].SectionName << "\n";
    cout << PEFile->Section[i].VirtualAddress << "\n";
    cout << PEFile->Section[i].VirtualSize << "\n";
    cout << PEFile->Section[i].PointerToRawData << "\n";
    cout << PEFile->Section[i].SizeOfRawData << "\n";
    cout << PEFile->Section[i].RealAddr << "\n";
}
DWORD RVAToOffset(DWORD RVA);
DWORD OffsetToRVA(DWORD RawOffset);

The Disassembler
To understand how to work with assemblers and disassemblers … you should understand the shape of the instructions and so on.
That’s the x86 instruction Format: Figure 3.

• The Prefixes are reserved bytes used to describe something in the Instruction like for example:
  • 0xF0: Lock Prefix … and it’s used for synchronization
  • 0xF2/0xF3: Repne/Rep … the repeat instruction for string operations
  • 0x66: Operand Override … for 16 bits operands like: mov ax,4556
  • 0x67: Address Override … used for 16-bits ModRM … could be ignored
  • 0x64: Segment Override For FS … like: mov eax, FS:[18]

• Opcodes:
  • Opcode encodes information about
    • operation type,
    • operands,
    • size of each operand, including the size of an immediate operand
  • Like Add RM/R, Reg (8 bits) → Opcode: 0x00
  • Opcode Could be 1 byte,2 or 3 bytes
  • Opcode could use the “Reg” in ModRM as an opcode extension … and this named “Opcode Groups”
  • Modrm: Describes the Operands (Destination and Source). And it describes if the destination or the source is register, memory address (ex: dword ptr [eax+ 1000]) or immediate (number).
  • SIB: extension for Modrm … used for scaling in memory address like: dword ptr [eax*4 + ecx + 50]
  • Displacement: The value inside the brackets […] … like dword ptr [eax+0x1000], so the displacement is 0x1000 … and it could be one byte, 2 bytes or 4 bytes
  • Immediate: It’s value of the source or destination if any of them is a number like (move ax,1000) … so the immediate is 1000

That’s the x86 instruction Format in brief … you can find more details in Intel Reference Manual.
To use PokasAsm class in SRDF for assembling and disassembling … you will create a new class and use it like this:
The Output: Listing 10.

Listing 9. Create a New Class and Use It Like This

CPokasAsm* Asm = new CPokasAsm();
DWORD InsLength;
char* buff;
buff = Asm->Assemble("mov eax,dword ptr [ecx+ 00401000h]",InsLength);
cout << "The Length: " << InsLength << "\n";
cout << "Assembling mov eax,dword ptr [ecx+ 00401000h]\n";
for (DWORD i = 0;i < InsLength; i++)
{
    cout << (int*)buff[i] << " ";
}
cout << "\n";
cout << "Disassembling the same Instruction Again\n";
cout << Asm->Disassemble(buff,InsLength) << " ... and the instruction length : " << InsLength << "\n";

Listing 10. The Output

The Length: 6
Assembling mov eax,dword ptr [ecx+ 00401000h]
FFFFFFF8B FFFFFFF8 00000000 00000010 00000040 00000000
Disassembling the same Instruction Again
mov eax ,dword ptr [ecx + 401000h] ... and the instruction length : 6

Listing 11. “DISASM_INSTRUCTION”

struct DISASM_INSTRUCTION
{
    hde32sexport hde;
    int entry;
    string* opcode;
    int ndest;
    int nsrc;
    int other;
    struct
    {
        int length;
        int items[3];
        int flags[3];
    } modrm;
    int (*emu_func)(Thread&,DISASM_INSTRUCTION*);
    int flags;
};
Also, we add an effective way to retrieve the instruction information. We created a disassemble function that returns a struct describes the instruction DISASM_INSTRUCTION and it looks like: Listing 11.

The Disassemble Function looks like:

```c
DISASM_INSTRUCTION* Disassemble(char* Buffer,DISASM_INSTRUCTION* ins);
```

It takes the Address of the buffer to disassemble and the buffer that the function will return the struct inside.

Let’s explain this structure:

1. hde: it’s a struct created by Hacker Disassembler Engine and describes the opcode ... The important Fields are:
2. len: The length of the instruction
3. opcode: the opcode byte ... if the opcode is 2 bytes so see also opcode2
4. Flags: This is the flags and it has some important flags like F_MODRM and F_ERROR_ XXXX (XXXX means anything here)
5. Entry: unused
6. Opcode: the opcode string ... with class “string” not “cString”
7. Other: used for mul to save the imm ... other than that ... it’s unused
8. Modrm: it’s a structure describes what’s inside the RM (if there’s) like “[eax*2 + ecx + 6]” for example ... and it looks like:
9. Length: the number of items inside ... like “[eax+ 2000]” contains 2 items
10. Flags[3]: this describes each item in the RM and its maximum is 3 ... it’s flags is:
11. RM_REG: the item is a register like “[eax ...”
12. RM_MUL2: this register is multiplied by 2
13. RM_MUL4: by 4
14. RM_MUL8: by 8
15. RM_DISP: it’s a displacement like [0x0401000 + ...]
16. RM_DISP8: comes with RM_DISP ... and it means that the displacement is 8-bits
17. RM_DISP16: the displacement is 16 bits
18. RM_DISP32: the displacement is 32-bits
19. RM_ADDR16: this means that ... the modrm is in 16-bits Addressing Mode
20. Items[3]: this gives the value of the item in the modrm ... like if the item is a register ... so it contains the number of this register (ex: ecx → item = 1) and if the item is a displacement ... so it contains the displacement value like [0x0401000] and so on.
21. emu_func: unused
22. Flags: this flags describes the instruction ... some describes the instruction shape, some describes destination and some describes the source ... let’s see
23. Instruction Shape: there are some flags describe the instruction like:
24. NO_SRCDEST: this instruction doesn’t have source or destination like “nop”
25. SRC_NOSRC: this instruction has only destination like “push dest”
26. INS_UNDEFINED: this instruction is undefined in the disassembler ... but you still can get the length of it from hde.len
27. OP_FPU: this instruction is an FPU instruction
28. FPU_NULL: means this instruction doesn’t have any destination or source
29. FPU_DEST_ONLY: this means that this instruction has only a destination
30. FPU_SRCDEST: this means that this instruction has a source and destination
31. FPU_BITS32: the FPU instruction is in 32-bits
32. FPU_BITS16: means that the FPU Instruction is in 16-bits
33. FPU_MODRM: means that the instruction contains the ModRM byte
34. Destination Shape:
35. DEST_REG: means that the destination is a register
36. DEST_RM: means that the destination is an RM like “word ptr [xxxx]”
37. DEST_IMM: the destination is an immediate (only with enter instruction”
38. DEST_BITS32: the destination is 32-bits
39. DEST_BITS16: the destination is 16-bits
40. DEST_BITS8: the destination is 8-bits
41. FPU_DEST_ST: means that the destination is “ST0” in FPU only instructions
42. FPU_DEST_STI: means that the destination is “STx” like “ST1”
43. FPU_DEST_RM: means that the destination is RM
44. Source Shape: similar to destination ... read the description in Destination flags above
45. SRC_REG
46. SRC_RM
47. SRC_IMM
48. SRC_BITS32
49. SRC_BITS16
50. SRC_BITS8
51. FPU_SRC_ST
52. FPU_SRC_STI
53. ndest: this includes the value of the destination related to its type ... if it’s a register ... so it
Write your own Debugger

54. \texttt{src}: this includes the value of the source related to the type ... see the \texttt{ndest} above.

That’s simply the disassembler. We discussed all the items of our debugger. We discussed the Process Analyzer, the Debugger, the PE Parser and the Disassembler. We now should put all together.

Put All Together
To write a good debugger and simple also, we decided to create an interactive console application (like msfconsole in Metasploit) which takes commands like run or bp (to set a breakpoint) and so on.

To create an interactive console application, we will use \texttt{cConsoleApp} class to create our Console App. We will inherit a class from it and begin the modification of its commands (Listing 12).

And the Code: Listing 13.
As you see in the previous code, we implemented 3 functions (virtual functions) and they are:

1. \texttt{SetCustomSettings}: this function is used for modifying the setting for your application ... like modify the intro for the application, include a log file, include a registry entry for the application or to include a database for the application to save data ... as you can see, it’s used to write the intro.

2. \texttt{Run}: this function is called to run the application. You should call to \texttt{StartConsole} to begin the interactive console.

3. \texttt{Exit}: this function is called when the user write “quit” command to the console.

The \texttt{cConsoleApp} implements 2 commands for you “quit” and “help”. Quit exit the application and help show the command list with their description.
To add a new command you should call to this function:

\begin{Verbatim}
AddCommand(char* Name,char* Description,char* Format, DWORD nArgs,PCmdFunc CommandFunc)
\end{Verbatim}

The command Func is the function which will be called when the user inputs this command ... and it should be with this format:

\begin{Verbatim}
void CmdFunc(cConsoleApp* App,int argc,char* argv[])
\end{Verbatim}

it’s similar to the main function added to it the App class. The argv is the list of the arguments for this function and the argc is the number of arguments (always equal to \texttt{nArgs} that you enter in add commands ... could be ignored as it’s reserved).
To use \texttt{AddCommand} ... you can use it like this:

\begin{Verbatim}
AddCommand("dump","Dump a place in memory in Listing 12. Use \texttt{cConsoleApp} Class to Create our Console App
\begin{Verbatim}
class cDebuggerApp : public cConsoleApp
{
 public:
 cDebuggerApp(cString AppName);
 ~cDebuggerApp();
 virtual void SetCustomSettings();
 virtual int Run();
 virtual int Exit();
};
\end{Verbatim}
\begin{Verbatim}
Listing 13. The Code
cDebuggerApp::cDebuggerApp(cString AppName) :
cConsoleApp(AppName)
{
}
cDebuggerApp::~cDebuggerApp()
{
 {{(cApp*)this}}->cApp();
}

void cDebuggerApp::SetCustomSettings()
{
 //Modify the intro of the application
 Intro = "\n "
 "***********************************\n **          Win32 Debugger       **
 ************************************\n ";
}

int cDebuggerApp::Run()
{
 //write your code here for run
 StartConsole();
 return 0;
}

int cDebuggerApp::Exit()
{
 //write your code here for exit
 return 0;
}
\end{Verbatim}
hex","dump [address] [size]",2,&DumpFunc);

The DumpFunc is like that:

```c
void DumpFunc(cConsoleApp* App, int argc, char* argv[])
{
    ((cDebuggerApp*)App)->Dump(argc, argv);
}
```

As it calls to Dump function in the cDebuggerApp class which inherited from cConsoleApp class.

We added these commands for the application: Listing 14.

For Run Function: Listing 15.

As you can see, we make the application start the console while the user enters a valid filename, otherwise, return error and close the application.

We will not describe all commands but commands that are the hard to implement (Listing 16).

This function at the beginning converts the arguments from string (as the user entered) to a hexadecimal value. And then, it reads in the debuggee process the memory that you need to disassemble. As you can see, we added 16 bytes to be sure that all instructions will be disassembled correctly even if one of them exceed the limits of the buffer.

Then, we begin looping on the disassembling process and increment the address by the length of each instruction until we reach the limited size.

**Listing 14. AddCommand**

```c
AddCommand("step","one Step through code","step",0,&StepFunc);
AddCommand("run","Run the application until the first breakpoint","run",0,&RunFunc);
AddCommand("regs","Show Registers","regs",0,&RegsFunc);
AddCommand("bp","Set an Int3 Breakpoint","bp [address]",1,&BpFunc);
AddCommand("hardbp","Set a Hardware Breakpoint","hardbp [address] [size (1,2,4)] [type .. 0 = access .. 1 = write .. 2 = execute]",3,&HardbpFunc);
AddCommand("membp","Set Memory Breakpoint","membp [address] [size] [type .. 0 = access .. 1 = write]",3,&MembpFunc);
AddCommand("dump","Dump a place in memory in hex","dump [address] [size]",2,&DumpFunc);
AddCommand("disasm","Disassemble a place in memory","disasm [address] [size]",2,&DisasmFunc);
AddCommand("string","Print string at a specific address","string [address] [max size]",2,&StringFunc);
AddCommand("removebp","Remove an Int3 Breakpoint","removebp [address]",1,&RemovebpFunc);
AddCommand("removehardbp","Remove a Hardware Breakpoint","removehardbp [address]",1,&RemovehardbpFunc);
AddCommand("removemembp","Remove Memory Breakpoint","removemembp [address]",1,&RemovemembpFunc);
```

**Listing 15. For Run Function**

```c
int cDebuggerApp::Run()
{
    Debugger = new cDebugger(Request.GetValue("default"));
    Asm = new CPokasAsm();
    if (Debugger->IsDebugging)
    {
        Debugger->Run();
        Prefix = Debugger->DebuggeeProcess->processName;
        if (Debugger->IsDebugging)StartConsole();
    }
    else
    {
        cout << Intro << "\n\n";
        cout << "Error: File not Found";
    }
    return 0;
}
The main function will call to some functions to start the application and run it: Listing 17.

**Conclusion**

In this article we described how to write a debugger using SRDF ... and how easy to use SRDF. And we described how to analyze a PE File and how disassembling an instruction works.

**AMR THABET**

I’m a Malware Researcher with 5+ years experience in reversing malware and researching and I’m now a Malware Researcher in Q-CERT. I’m the Author of many open-source tools like Pokas Emulator and Security Research and Development Framework (SRDF). I was a Speaker in Cairo Security Camp 2010 and University of Sydney. I wrote numerous articles in malware and programming in Hakin9 Magazine, SecurityKaizen Magazine and Code Project.

---

**Listing 16. Commands H to Implement**

```cpp
void cDebuggerApp::Disassemble(int argc, char* argv[])
{
  DWORD Address = 0;
  DWORD Size = 0;
  sscanf(argv[0], "%x", &Address);
  sscanf(argv[1], "%x", &Size);
  DWORD Buffer = Debugger->DebuggeeProcess-
                 >Read(Address,Size+16);
  DWORD InsLength = 0;

  for (DWORD InsBuff = Buffer; InsBuff < Buffer+
       Size ; InsBuff+=InsLength)
  {
    cout << (int*)Address << ": " << Asm-
            >Disassemble((char*)
            InsBuff,InsLength) << "\n";
    Address+=InsLength;
  }
}
```

**Listing 17. Start the Application and Run It**

```cpp
int _tmain(int argc, char* argv[])
{
  cDebuggerApp* Debugger = new
cDebuggerApp("Win32Debugger");
  Debugger->SetCustomSettings();
  Debugger->Initialize(argc, argv);
  Debugger->Run();
  return 0;
} 
```