A Manual To REVERSE ENGINEERING

HOW TO ANALYZE APPLICATIONS WITH OLLY DEBUGGER?

HOW TO IDENTIFY AND BYPASS ANTI-REVERSING TECHNIQUES?

SO CAT AND WIRESHARK FOR PRACTICAL SSL PROTOCOL REVERSE ENGINEERING

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

PLUS

JSCRAMBLER – PROTECT YOUR CODE (REVIEW)

MODERN WEBSITES, WHICH USE WEB 2.0 AND AJAX, OFTEN GENERATE HTML AND JAVASCRIPT CODE ON THE FLY
Cloud & Virtualization
- Cloud & Virtualization Consultancy
- Building Virtualized Infrastructure
- Infrastructure on Public Cloud
- Building Private Cloud
- Cloud Management Setups
- Big Data Setups
- Infrastructure Management and Support

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

Security & Compliance
- Server & Network Security Setups
- Security Testing, Audit and Compliance
- Incident Response
- Managed Security Service
Atola Insight
That’s all you need for data recovery.

Atola Technology offers Atola Insight – the only data recovery device that covers the entire data recovery process: in-depth HDD diagnostics, firmware recovery, HDD duplication, and file recovery. It is like a whole data recovery Lab in one Tool.

This product is the best choice for seasoned professionals as well as start-up data recovery companies.

Emphasized features at a glance:
- Automatic in-depth diagnostic of all hard drive components
- Automatic firmware recovery and ATA password removal
- Very fast imaging of damaged drives
- Imaging by heads
- Case management
- Real time current monitor
- Firmware area backup system
- Serial port and power control
- Write protection switch

Visit atola.com for details
Dear Readers,

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

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

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

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

MERRY CHRISTMAS!

Regards,

Krzysztof Samborski
and Hakin9 Team
How to Analyze Applications With Olly Debugger?
BY JAROMÍR HOREJSI
When you write your own programs and you would like to change or modify some of their functions, you simply open the source code you have, make desired changes, recompile and your work is done. However, you don’t need to have source code to modify function of a program – using specialized tools, you can understand a lot from program binary file, you can add your new functions and features and you can also modify and alter its behavior.

How to Disassemble and Debug Executable Programs on Linux, Windows and Mac OS X?
BY JACEK ADAM PIASECKI
The Interactive Disassembler Professional (IDA Pro) is an extremely powerful disassembler distributed by Hex-Rays. Although IDA Pro is not the only disassembler, it is the disassembler of choice for many malware analysts, reverse engineers, and vulnerability analysts.

How to use Socat and Wireshark for Practical SSL Protocol Reverse Engineering?
BY SHANE R. SPENCER
Secure Socket Layer (SSL) Man-In-the-Middle (MITM) proxies have two very specific purposes. The first is to allow a client with one set of keys to communicate with a service that has a different set of keys without either side knowing about it. This is typically seen as a MITM attack but can be used for productive ends as well.

How to Identify and Bypass Anti-reversing Techniques?
BY EOIN WARD
Learn the anti-reversing techniques used by malware authors to thwart the detection and analysis of their precious malware. Find out about the premier shareware debugging tool Ollydbg and how it can help you bypass these anti-reversing techniques. This article aims to look at anti-reversing techniques used in the wild. These are tricks used by malware authors to stop or impede reverse engineers from analysing their files.

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

JScrambler
Protect Your Code (Review)
BY RAHEEL AHMAD
Modern websites, which use Web 2.0 and AJAX, often generate HTML and JavaScript code on the fly. This means that standard static code analysers cannot fully scan the source code and locate client-side JavaScript issues, since the source code itself does not yet include the entire HTML and JavaScript code.
How to Analyze Applications With Olly Debugger?

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

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

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

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

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

What is Olly Debugger?
Olly Debugger (we will call it OllyDbg) is a 32-bit debugger for analyzing portable executable (PE) files for Microsoft Windows. (There are many different types of computer files. PE files are standard executable .EXE files, DLL libraries, SCR screensavers, etc... When you open the file in any editor, you notice two signatures – MZ in the beginning and PE a bit further. At address 0x3c you will see the offset of PE signature. In our example value on address 0x3c is 0xB0, therefore on address 0xB0 you will see PE signature). See Figure 1 for screenshot.

![Figure 1. PE file format](image-url)
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.
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 6. Software, memory and hardware breakpoints](image-url)
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 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 \texttt{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).
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 4475D1, you can see instruction TEST BL, BL followed by JZ 4475EA (you can see it in Figure 22 too). If BL equals 0, TEST BL, BL (which corresponds to logical function BL & BL) sets zero flag to 1 (0 & 0 = 0, result is zero, therefore zero flag = TRUE = 1) and JZ jumps to 4475EA, therefore no message is displayed.

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

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

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

Figure 22. Comparison procedure

Figure 23. Correct name/serial combination found
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 (00447481: CALL 0041B798)
- gets pointer to the text of the first text box (00447486: MOV EAX,DWORD PTR SS:[EBP-4])
- reads (ESI-1)-th character from the beginning of the string to EAX (00447490: 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 (004474C3: 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…

\[
\begin{align*}
(c \times e) + (r \times m) + (a \times k) + (c \times c) + \\
(k \times a) + (m \times r) + (e \times c) = \\
(0 \times 63 \times 0 \times 65) + (0 \times 72 \times 0 \times 6D) + (0 \times 61 \times 0 \times 6B) + (0 \times 63 \times 0 \times 67) + (0 \times 6D \times 0 \times 72) + (0 \times 65 \times 0 \times 63)
\end{align*}
\]

\[
= 0x270F + 0x308A + 0x288B + 0x2649 + 0x288B + 0x308A + 0x270F = 0x12691 = 75409 \text{ (in decimal)}
\]

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 is a computer virus re-searcher 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/encrypted, 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
IDA Pro
How to Disassemble and Debug Executable Programs on Linux, Windows and Mac OS X?

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

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

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

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

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

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

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

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

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

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

In graph mode, the color and direction of the arrows help show the program’s flow during analysis. The arrow’s color tells you whether the path is...
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 (B3EC18) 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-url)
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_10], 1
0040107E loc_40107E: ; CODE XREF:
sub_401040+35j
0040107E cmp [ebp+var_C], 0
00401082 jnz short loc_401097
00401084 mov eax, [ebp+var_4]
00401087 mov [esp+18h+var_14], eax
0040108B mov [esp+18h+var_18], offset aPrintNumberD ; "Print Number= %d\n"
00401092 call printf
00401097 call sub_4010A0
```

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

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

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

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

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

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

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

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

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

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

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

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

The static analysis of strings can often be used as a starting point for your analysis. If you see an...
interesting string, use IDA Pro’s cross-reference feature to see exactly where and how that string is used within the code.

**Analyzing Functions**

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

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

**Using Graphing Options**

IDA Pro supports five graphing options, accessible from the buttons on the toolbar shown in Figure 7.
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 \texttt{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 \texttt{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 \texttt{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 \texttt{arg\_4} to \texttt{port\_str} and \texttt{var\_598} to \texttt{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...
across the disassembly window whenever there is a cross-reference to the address in which you added the comment, press the semicolon (;) key.

**Formatting Operands**

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

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

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

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

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

**Using Named Constants**

Malware authors (and programmers in general) often use named constants such as GENERIC_

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

<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 ; Ò</td>
<td>00008387 nop</td>
</tr>
<tr>
<td>00008388 db 90h ; Ò</td>
<td>00008388 nop</td>
</tr>
<tr>
<td>00008389 db 88h ; ́</td>
<td>00008389 mov ebx, eax</td>
</tr>
<tr>
<td>0000838A db 0Dh ; +</td>
<td>0000838B add ebx, 28h ; '('</td>
</tr>
<tr>
<td>0000838B db 83h ; å</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 ; å</td>
<td>00008395</td>
</tr>
<tr>
<td>0000838F db 3</td>
<td>00008395 loc_8395:    ; CODE XREF: seg000:000083A0j</td>
</tr>
<tr>
<td>00008390 db 1Bh</td>
<td>00008395 xor byte ptr [ebx], 97h</td>
</tr>
<tr>
<td>00008391 db 88h ; Í</td>
<td>00008398 inc ebx</td>
</tr>
<tr>
<td>00008392 db 1Bh</td>
<td>00008399 inc ecx</td>
</tr>
<tr>
<td>00008393 db 33h ; 3</td>
<td>0000839A cmp ecx, 700h</td>
</tr>
<tr>
<td>00008394 db 0C9h ; +</td>
<td>000083A0 jnz short loc_8395</td>
</tr>
<tr>
<td>00008395 db 80h ; Ì</td>
<td>000083A2 retn 7B1Ch</td>
</tr>
<tr>
<td>00008396 db 33h ; 3</td>
<td>000083A2 ; -----------------------------000083A5 db 16h</td>
</tr>
<tr>
<td>00008397 db 97h ; ò</td>
<td>000083A6 db 7Bh ; {</td>
</tr>
<tr>
<td>00008398 db 43h ; Ì</td>
<td>000083A7 db 8Fh ; Å</td>
</tr>
<tr>
<td>00008399 db 41h ; À</td>
<td></td>
</tr>
<tr>
<td>0000839A db 81h ; ò</td>
<td></td>
</tr>
<tr>
<td>0000839B db 0F9h ; -</td>
<td></td>
</tr>
<tr>
<td>0000839C db 0</td>
<td></td>
</tr>
<tr>
<td>0000839D db 7</td>
<td></td>
</tr>
<tr>
<td>0000839E db 0</td>
<td></td>
</tr>
<tr>
<td>0000839F db 0</td>
<td></td>
</tr>
<tr>
<td>000083A0 db 75h ; u</td>
<td></td>
</tr>
<tr>
<td>000083A1 db 0F3h ; =</td>
<td></td>
</tr>
<tr>
<td>000083A2 db 0C2h ; -</td>
<td></td>
</tr>
<tr>
<td>000083A3 db 1Ch</td>
<td></td>
</tr>
<tr>
<td>000083A4 db 7Bh ; {</td>
<td></td>
</tr>
<tr>
<td>000083A5 db 16h</td>
<td></td>
</tr>
<tr>
<td>000083A6 db 7Bh ; {</td>
<td></td>
</tr>
<tr>
<td>000083A7 db 8Fh ; Å</td>
<td></td>
</tr>
</tbody>
</table>
READ in their source code. Named constants provide an easily remembered name for the programmer, but they are implemented as an integer in the binary. Unfortunately, once the compiler is done with the source code, it is no longer possible to determine whether the source used a symbolic constant or a literal.

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

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

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

On the Web

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
MOVE TOMORROW’S BUSINESS TO THE CLOUD TODAY

YOUR TRUSTED ADVISOR ON CLOUD COMPUTING

MULTI-VENDOR
ANY DEVICE
HYBRID CLOUD
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/
• Install Wireshark – Network traffic analyzer: http://www.wireshark.org/
• Install OpenSSL – Secure Socket Layer (SSL) binary and related cryptographic tools: http://www.openssl.org/
• Install TinyCA – Simple graphical program for certification authority management: http://tinyca.sm-zone.net/
• Install LogJam – Client for LiveJournal-based sites: http://andy-shev.github.com/LogJam/

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

An SSL CA signs SSL certificates and is nothing more than a set of certificates files that can be used by tools like OpenSSL to sign newly 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-LISTEN:443,verify=0,fork,key=www.livejournal.com-key.pem,cacert=www.livejournal.com-key.pem cafile=Yoyodyne-ca-cert.pem OPENSSL:208.93.0.128:443,verify=0,fork
```
Simple enough. Browsing to https://www.livejournal.com with w3m and wget should work sucessfully now and a stream of random encrypted information will be printed by socat.

Listing 1. Install Yoyodyne CA certificate

```
spencersr@bigboote:~$ sudo mkdir /usr/share/ca-certificates/custom
spencersr@bigboote:~$ sudo cp Yoyodyne-ca-cert.pem /usr/share/ca-certificates/custom/Yoyodyne-ca-cert.crt
spencersr@bigboote:~$ sudo chmod +rw /usr/share/ca-certificates/custom/Yoyodyne-ca-cert.crt
spencersr@bigboote:~$ sudo dpkg-reconfigure -p none ca-certificates configuration

Trust new certificates from certificate authorities? 1
(Enter the items you want to select, separated by spaces.)
Certificates to activate: 2
```
```
1. cacert.org/cacert.org.crt
2. custom/Yoyodyne-ca-cert.crt
3. debian.org/ca.crt
150. mozilla/XRamp_Global_CA_Root.crt
151. spi-inc.org/spi-ca-2003.crt
152. spi-inc.org/spi-ca-cert-2008.crt
```
```
Updating certificates in /etc/ssl/certs... 1 added, 0 removed; done.
Adding debian/Yoyodyne-ca-cert.pem done.
```
Chaining two socat instances together with an unencrypted session in the middle.
So far so good! Now we need to have socat connecting to another socat using standard TCP4 protocol in order to view the unencrypted data. This works by having one socat instance listening on port 443 (HTTPS) and then forwarding to another socat on port 8080 (HTTP) which then forwards on to port 443 (HTTPS) of the real www.livejournal.com.

Listing 2. Socat terminal

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

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

Socat instance two:

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

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

Hurray! You should be dancing at this point.

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

Using Wireshark to capture and view the unencrypted stream.

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

Since by default Wireshark captures all traffic we should set up a capture filter that only listens for packets on port 8080 of host 127.0.2.1 (Figure 4). Once LogJam is run packet will start streaming in while Wireshark is recording (Figure 5).

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.

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

Figure 5. Wireshark with captured unencrypted packets

SHANE R. SPENCER

Shane R. Spencer is based out of Anchorage Alaska and has over 10 years of system administration and programming experience. Many of his projects are Python based and interface with external services that provide no usable API and communicate over HTTPS only.
StaffCop Standard allows you to monitor all activities on company computers and prevent the unauthorized distribution of sensitive corporate information.

StaffCop will help you:

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

You need StaffCop to:

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

Who needs StaffCop:

CEO/CTO
Corporate Security Manager
HR Manager
System Administrator

More Information, Demo Versions, Videos and Technical Guides -

www.STAFFFCOP.com

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

---

**Figure 17. API Call found in OllyDbg**

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

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

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

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**

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
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 their 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](image)

![Figure 2. Find referenced strings](image)
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);
}
```

Figure 3. Strings as seen in Hiew32

Figure 4. Layer 1 Directory Detection, Assemble view
equal too) to insure you are directed to the “Well done first layer passed”. This technique of manipulating the jump can be used throughout the binary to jump to your chosen branch.

**Layer 2: Anti-debugger**

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

The assembler code is available in Figure 6. It calls the `IsDebuggerPresent` API and based on its response jumps to the “Not running in a debugger” print or continues on to the print 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

```c
Listing 2. IsDebuggerPresent API

BOOL WINAPI IsDebuggerPresent(

Listing 3. IsDebuggerPresent ‘if statement’

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

Figure 6. IsDebuggerPresent ‘if’ statement as see from Ollydbg
REACHING THE CODE

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 wildcard. 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

LPCWSTR get_Kernel32_string()
{
  char buffer_Kernel32[9];
  buffer_Kernel32[0] = 'K';
  buffer_Kernel32[1] = 'e';
  buffer_Kernel32[2] = 'r';
  buffer_Kernel32[3] = 'n';
  buffer_Kernel32[4] = 'e';
  buffer_Kernel32[5] = 'l';
  buffer_Kernel32[6] = '3';
  buffer_Kernel32[7] = '2';
  buffer_Kernel32[8] = '\0';

  //The following is code to convert the char buffer into a LPCWSTR
  size_t newsize = strlen(buffer_Kernel32) + 1;
  wchar_t * wcstring = new wchar_t[newsize];
  size_t convertedChars = 0;
  mbstowcs_s(&convertedChars, wcstring, newsize, buffer_Kernel32, _TRUNCATE);
  return wcstring;
}
code it will be slightly more difficult. First is a method where you push values into an array and then convert the array to a string, see Listing 4.

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

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

As you can see, the values are not pushed in order. If you look at the code you can see ‘real-FitCeeA!’ 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 light weight and fast. It is also very easy to break.

The string I wanted to hide was copied it into a buffer. I ran the code once and it created the ciphertext. I placed this ciphertext into the original buffer so the next time I ran it would create the plaintext. I have only used a byte wise encryption, malware may use longer keys. The C++ code to build the buffer containing the 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 continue. 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[2] = 'e'; //0x65
    buffer_CreateFileA[3] = 'a'; //0x61
    buffer_CreateFileA[4] = 'l'; //0x6c
    buffer_CreateFileA[5] = 'F'; //0x46
    buffer_CreateFileA[6] = 'i'; //0x69
    buffer_CreateFileA[7] = 't'; //0x74
    buffer_CreateFileA[8] = 'C'; //0x61
    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

Figure 10. The assembler code (Listing 6).
**Listing 6. Secret Code Buffer, (ciphertext) Xored with 0xFA to produce plaintext**

```c
unsigned char buffer_SecretCode[24] = {0xae, 0x92, 0x93, 0x89, 0xda, 0x93,
0x89, 0xda, 0x8e, 0x92, 0x9f, 0xda, 0xa3, 0x9f, 0x99, 0x88, 0x9f, 0x8e,
0xda, 0x99, 0x95, 0x9e, 0x9f};
```

```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*) (LPCSTR, DWORD, DWORD, LPSECURITY_ATTRIBUTES, DWORD, DWORD, HANDLE)
GETADAPTORSFUNC fpGetProcAddress;

fpGetProcAddress = (GETADAPTORSFUNC)GetProcAddress(hDLL, PROC);
```

```c
//Dynamically call CreateFileA
hFile = fpGetProcAddress(fname, GENERIC_READ, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
```

```c
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 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 to an EXE file.

The source code, the human readable and understandable file that is created by the programmer himself is firstly compiled in to an object file with readable symbols, meaning that it is still understandable by a normal human.

The compiler then transforms the object file in to an executable, the format which all of your windows programs is compiled in, rendering the binary code symbol-less, in other words: unreadable.
The source code of a simple ‘Hello World’ application

For example, if you make a simple application in C++, you need to write a source file first, something like ‘MyApp.c’. When you are done, you want to make an executable file out of your code, so you compile it. During the compilation, the file ‘MyApp.c’ is translated into object and then binary code, making it extremely hard to humanly interpret and almost impossible to uncompile or decompile back to the original file; ‘MyApp.c’.

Programmers rely on this idea for security of their application. The harder it is to decompile their application and reverse the actions of a compiler, the more secure their code. However, when there’s a way in, you can be sure that there is one out.

Editing Code AKA Debugging

Although the compiled code is unreadable, there are, however, programs that can translate it into a semi-readable state. These programs are called debuggers. Debuggers are programs that read those binary codes that the program has been compiled to and convert them into easier to understand terms. Those terms make up an extremely low level programming language known as Assembly. If you thought learning C++ was a headache then wait till you try out assembly. Though complex as it may be, assembly code is what all applications are written in when compiled. It is extremely low level meaning. It takes approximately 10 lines of assembly to compensate for one line of C++. For that reason, assembly code is not a preferred language among software developers.

Now knowing the connection between your program, assembly and the debugger, we can move on to the next topic: the debugging.

Debugging is the process of removing bugs or errors from a program

A debugger, is a program that does what its name implies, it removes bugs. To do that, it allows users to edit the assembly of a program, changing its structure and function. For example, if I had an annoying bug where a program always counts 0s as 1s, I can create a fix myself with a debugger by simply loading my program and then editing the section of assembly where the program confuses 0s with 1s. Then I can release the fix online for all the users of that program.

Assembly Code

Before you can debug anything, you need a fair bit of knowledge on assembly, not enough to code programs, but enough to understand how programs
are coded in assembly. You can access this great tutorial here: http://www.cs.virginia.edu/~evans/cs216/guides/x86.html.

**Tools of the Trade**

OK, so you know a bit of assembly and you have a program to reverse engineer, let’s get a debugger. Nowadays, there are a lot of debuggers available so choosing the right one can be confusing.

**Below is the list of debuggers that work for any Windows application. Those include:**

- OllyDbg
- SoftIce
- Microsoft Visual Studio Debugger
- AQTime
- GDB
- AQT

In addition, there is over a hundred different debuggers, all made for different platforms and languages. But since we are debugging under windows, this is not relevant. You can though, simply Wikipedia the word ‘Debugger’ to find a long list of debuggers.

**Reverse Engineering Example**

In this demonstration we will use a free and widely used debugger: OllyDbg. You can get it from their official website: http://www.ollydbg.de/.

After downloading the debugger, unzip and open it. Load your application that you want to debug by clicking ‘Open’ on the main toolbar.

In this demonstration, we will debug a superficial program that simulates the licensing features in a real program. Let’s call it HackMe.EXE. Basically HackME.EXE asks for a serial key and name and returns the message ‘Valid Key’ if the key and name match, and ‘Invalid Key’ if they do not. Your purpose is to either find a valid serial key or a way to bypass this process and skip to the point where you can enter any key, and get a ‘Valid Key’ message.

This is a classic example of RCE and to attack such a problem is fairly easy if you have the right tools. OllyDbg is an excellent choice as it works for all windows compiled executables, has a lot of use functions such as setting breakpoints, finding string references, etc. Because of that we will use OllyDbg as our debugger in our demonstration.

**Step 1**

Open the program ‘HackME.EXE’ in OllyDbg by clicking ‘Open’ and choosing the file.

**Step 2**

Right click on the window where you see a lot of assembly code, and then select ‘Find All Referenced Strings.”

**Step 3**

You should be taken to a window where all the strings in the HackMe.EXE is listed. We want to see all its strings because we know for a fact that the messages ‘Valid Key’ and ‘Invalid Key’ is embedded somewhere in the application. If we can find its location, the corresponding code that generates these messages will also be there.

**Step 4**

Search. Search through all the strings listed until you find the text ‘Invalid Key’. You should find it, if not, then you will have to read the section **defensive mechanisms.**

**Step 5**

Double click on the text ‘Invalid Key.’ It should take you to the disassembly where the actual text is located.

**Step 6**

Now here’s the tricky part. Look at the assembly above where the text is located. If you have done your homework and researched a bit on assembly you will know what to look for. If you don’t, then I will briefly fill you in. In order to determine if the key is valid or not the program needs to actually compare the key and name. This is where we, as REers, do our thing. In windows assembly, the commands JZ, JNZ stand for operators that compare values and if they are true then they will jump to a section of the code.

Because the program we are debugging is comparing your name and serial key, we needed to find the section of the assembly that shows the ‘Invalid Key’ message, as done so in steps 1 to 5. Now that we have located this section, we are going to search for the JNZ or JZ operator replace it with themselves. For example if the program uses JZ to evaluate whether the key is valid or not, we replace it with JNZ and vice versa.

With that being said, look up from the point where you found the text ‘Invalid Key’ search for the commands JZ and JNZ; you only need to find one of them as there is only one anyway.

When you find the command, double click on it on the debugger to edit and do the following:

- If the command is JZ then change it to JNZ
- If the command is JNZ change it to JZ
Now run the program again by clicking ‘Run’ on the toolbar.

**Step 7**

Enter any serial number and name and you should get the message ‘Valid Key.’

Congrats! You have just reverse engineered an application. Seems easy huh? Are application really that easy to modify?

**Defensive Mechanisms**

Reverse engineering a small and unprotected application is extremely easy, but applications today are complex and protected as software piracy is extremely popular.

Since the uprise of reverse engineering, software companies have used packers to encrypt or scramble their code, giving crackers a hard time when they attempt to debug it.

For example, a program that is encrypted and scrambled would be impossible to debug unless the hacker can retrieve the original executable. This process seems secure right? **Wrong.** For every executable packer out there, there is always an unpacker. A hacker can simply search up the packer and then download the unpacker from illegal software piracy websites. The scrambled executable can then be unscrambled and debugged. If you are a software developer, your best bet is to find an uncommon executable packer to secure your files.

**The windows executable format is more vulnerable to debugging and modification than Mac or Linux binaries**

Just packers and encrpyts 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 in to a windows executable. However as VB compiles to a high level, interpreted code, as opposed to C++’s native code, it can be easily dissembled. A hacker can simply use a program such as VB Decompiler or VB Reformer and obtain almost every single source file you wrote.

Though it seems that any windows program is vulnerable to modification and tampering, as long as you compile that program with a native language such as C++ or C, your app should be relatively safe from decompilation.

**Reverse Engineering Online**

Today, there are teams dedicated to REing software, forums dedicated to teaching users the process and websites dedicated to spreading the reverse engineered app. A simple search on Google on something like ‘How to crack’ or ‘How to hack’ will lead you to over a million tutorials on the subject. There are teams, such as CORE which stands for “Challenge Of Reverse Engineering”, there are unnamed websites that allow hackers to upload their work, but why. Why does one reverse engineer?

The answer is simple. It is because software isn’t free. In the world of commercial software, you have to buy a license to use it. You have to subscribe by paying a certain amount every month to use it. You have to register your software to use it.

It would be fine if software were like cars. They can’t be copied or pasted. They can’t be uploaded on to software piracy dedicated websites. That can’t be loaded into debuggers. There is only one car for every person.

However, that’s software’s weak point. Software can be modified, debugged, copied and distributed. Software isn’t real, it’s virtual, and hackers recognized this as early as when the first version of Windows was released.

Reverse engineering software eliminates the requirement of users purchasing a valid license, and in return saves them time and money. Though illegal as it may be, it is human nature to find the cheapest and easiest way to obtain something they want.

**Reverse Engineering in History**

A famous example of reverse-engineering involves San Jose-based Phoenix Technologies Ltd., which in the mid-1980s wanted to produce a BIOS for PCs that would be compatible with the IBM PC’s proprietary BIOS. (A BIOS is a program stored in firmware that’s run when a PC starts up).

To protect against charges of having simply (and illegally) copied IBM’s BIOS, Phoenix reverse-engineered it in a way that was smart but indirect. First, a team of engineers studied the IBM BIOS – about 8KB of code – and described everything it did as completely as possible without using or referencing any actual code. Then Phoenix brought in a second team of programmers who had no prior knowledge of the IBM BIOS and had never seen its code. Working only from the first team’s functional specifications, the second team wrote a new BIOS that operated as specified.

The resulting Phoenix BIOS was different from the IBM code, but for all intents and purposes, it operated identically. Using the clean-room approach, even if some sections of code did happen to be identical, there was no copyright infringement. Phoenix began selling its BIOS to companies that then used it to create the first IBM-compatible PCs.

**Conclusion**

In conclusion, reading this article should have granted you with some more insight in the topic of reverse engineering. You should have learnt how reverse engineering works, how reverse engineering is accomplished and, most importantly, how reverse engineering is used. If you want more information on RE or RCE, you can visit the webpages listed below:

- www.en.wikipedia.org/wiki/Reverse_engineering
- www.searchcio-midmarket.techtarget.com/definition/reverse-engineering
- www.youtube.com/watch?v=vGBFEDslWhQ
- www.securitytube.net/video/1363

---

**LORENZO XIE**

Lorenzo Xie is the owner of XetoWare.com and Ace VideoConverter.com. He also works with several other software companies and specialises in windows software development. You can contact him directly at Lorenzo@xetoware.com.
INNOVATIVE e-services for websites monitoring

SEO monitor
SEO website monitor

SPEED monitor
website loading speed monitor

CONTENT monitor
website content language correctness monitor

www.monitorstrony.pl
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 functionality of the software code by applying the hit and miss rule.
006 – Identifying files used.
007 – Identify variables used in the code, this is very important.

Figure 1. Fundamental Requirements
Figure 2. IDA in Flow

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 powerful 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-
lowing this approach the researcher is free to explore and take any further steps as desired.

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

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

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

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

Phase 1: Identify Relevant Components
This first phase demands the investigation of the core component of the target; in this case it is Microsoft Fingerprint Reader. A number of methods can be applied for identifying core components of Microsoft Fingerprint Reader at this level. The noticeable start point for us would be to include the device drivers that are used, in Windows case the operating system itself provides much information on the device drivers and their system location, it’s only the matter of knowing it as shown in Figure 5.

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

Table 1. Identifying possible system functions from filenames alone

<table>
<thead>
<tr>
<th>System Component / Filename</th>
<th>Likely Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPHost.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>
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 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 from OllyDbg tool is presented in Figure 8.

**Next Steps**

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

**Conclusion**

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

**References**

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

---

**RAHEEL AHMAD**

Raheel Ahmad, CISSP, is an Information Security Consultant with around 10 years of experience in security and forensic investigations while working for Big4 Audit Firms and Consulting companies. He holds several security certifications as CISSP, CEH, CEI, MCP, MCT, CRISC, and Co-bIT Foundation. Raheel is a certified instructor for ethical hacking boot camps.
Modern websites, which use Web 2.0 and AJAX, often generate HTML and JavaScript code on the fly. This means that standard static code analyzers cannot fully scan the source code and locate client-side JavaScript issues, since the source code itself does not yet include the entire HTML and JavaScript code.

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

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

JScrambler Overview

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

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

Figure 2. Shows the application mode of JScrambler

Figure 3. Shows functionality you can use to achieve transformation from protection point of view
sible form, installs all sorts of protection mechanisms and optimizes the code. Huh – how about the functionality of your code? Yeah – it transforms and protects while maintaining your code functionality.

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

Figure 2 shows the application modes available in JScrambler.

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

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

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

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

You can use JScrambler to hide known calls to the browser DOM objects, or HTML5-specific elements like Canvas. Figures 6 and 7 show an obfuscated HTML5 Canvas example. You can find the code available at http://webfensive.com/canvas/.
There’s also the possibility of adding an exclusion attribute to script tags to make JScrambler ignore code which you don’t want it to touch.

Example:

```html
<script src="foo.js" jscrambler="ignore"></script>
```

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

**Conclusion**

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

**RAHEEL AHMAD**

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

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

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

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

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

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

Information Security Management
Improve every aspect of your information security!

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

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

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

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

IMF Academy
info@imfacademy.com
Tel: +31 (0)40 246 02 20
Fax: +31 (0)40 246 00 17
PLEASE SEE WWW.UAT.EDU/FASTFACTS FOR THE LATEST INFORMATION ABOUT DEGREE PROGRAM PERFORMANCE, PLACEMENT AND COSTS.

LEARN:
Advancing Computer Science
Artificial Life Programming
Digital Media
Digital Video
Enterprise Software Development
Game Art and Animation
Game Design
Game Programming
Human-Computer Interaction
Network Engineering

Network Security
Open Source Technologies
Robotics and Embedded Systems
Serious Game and Simulation
Strategic Technology Development
Technology Forensics
Technology Product Design
Technology Studies
Virtual Modeling and Design
Web and Social Media Technologies

www.uat.edu > 877.UAT.GEEK

IT’S IN YOUR PULSE.

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

GEEKED AT BIRTH.