fhwedel
UNIVERSITY OF APPLIED SCIENCES

DEPARTMENT OF COMPUTER SCIENCE

Seminar IT Security

Malware

Handed in on:

23 Mai 2019

Handed in by
Amar Bolkan
Tinsdalerweg 125
22880 Wedel
Tel.: (040) 12 34 56 78
E-mail: amar.bolkan@web.de

Advisor:
Prof. Dr. Gerd Beuster
Fachhochschule Wedel
Feldstraße 143
22880 Wedel
Phone: (041 03) 80 48-38
E-mail: gb@fh-wedel.de

Contents

List of Figures III

1 Introduction 1
- 1.1 Prolog . . . 1
- 1.2 Overview . . . 1
- 1.3 Goal . . . 1

2 Types of Analysis 2
- 2.1 Analysis Techniques . . . 2
  - 2.1.1 Static Analysis . . . 3
  - 2.1.2 Dynamic Analysis . . . 3

3 Obfuscation Techniques 5
- 3.1 Polymorphism . . . 5
- 3.2 Metamorphism . . . 5
  - 3.2.1 Metamorphic Engines . . . 5

4 Thwarting Analysis attempts 7
- 4.1 Anti-Disassembly . . . 7
  - 4.1.1 Types of Disassemblers . . . 7
  - 4.1.2 Anti-Disassembly Techniques . . . 9
- 4.2 Packers . . . 11
  - 4.2.1 Overview . . . 11
  - 4.2.2 Workflow . . . 11
  - 4.2.3 The Unpacking Stub . . . 11
  - 4.2.4 Unpacking Methods . . . 11
- 4.3 Antidebugging . . . 13
  - 4.3.1 Debuggers . . . 13
  - 4.3.2 Detecting Debuggers . . . 13

5 Conclusion 14

Bibliography 15

List of Figures

2.1 Screen after the infection of wannacry 2

3.1 Metamorphic virus propagation scheme 6

4.1 Example of an impossible disassembly situation 10
4.2 Illustration of an Packing/Unpacking cycle 12

1 Introduction

1.1 Prolog

From simple viruses in the 90s printing out weird text messages to sophisticated weapons used in international warfare, Malware indeed has come quite a long way. While many of the first recorded attacks weren't of malicious nature and more of an annoyance than a real threat, nowadays a whole black market economy has risen up around the trading of personal data or company secrets. Criminals have access to a plethora of sophisticated tools, which has made it increasingly more difficult if not impossible for Defenders to prevent such attacks.

1.2 Overview

This paper gives a basic introduction into the topic of malware analysis and will give a little insight into what techniques are applied to hide it's presence. The first section tackles some basic ways to extract information from malware samples as well as how to utilize those informations to detect samples of similar type. The second section addresses some popular methods used to hinder Analysts such as Packing, Antidebugging and Antidisassembly.

1.3 Goal

The goal of this paper is to show some of the tricks used by malware in the wild (meaning not yet classified malware) to thwart detection by Antivirus Vendors and impede the analysis process of their captured samples. For this I discuss several examples which should give the reader an easier understanding of the topics at hand.

2
Types of Analysis

What exactly is malware? Malware is an umbrella term referring to software that gets installed and operates against a user's will usually for the benefit of a third party [MBA+13]. Malware can be divided into many categories. Viruses, Worms, Rootkits, Trojans, Ransomware, Spyware just to name a few. Often real world Malware contains components, which might make it part of multiple of the aforementioned categories. An example of a malware, which combined properties of different categories was the wannacry worm, which targeted windows machines in 2017 and encrypted their data, demanding a ransom in cryptocurrency for the decryption key.

Figure 2.1: Screen after the infection of wannacry

Wannacry, after infection, spread by itself over the network. A trait typically seen in malware classified as Worms. On the other hand holding data hostage is a common trait of the ransomware categories. The whole attack lasted for 4 days with an estimated damage of at least 4 Billion Dollar worldwide, infecting companys such as the Deutschebahn and O2.

2.1 Analysis Techniques

The field of malware analysis can be divided into static and dynamic analysis. Each of them utilizing their own tools and methodologies to elicit information out of often very convoluted and well protected pieces of malware. The use of each method largely depend on the goal and context of the analysis. Sometimes malware applies techniques specifically designed to thwart one single method. This puts it into the hands of the professionel to determine through careful observation which methods are the most applicable.

Other times it is not possible to utilize the perfect method because of resource constraints. This is often the case in Antivirus Software, which is required to be fast and efficient as well as being accurate, which is often not possible to achieve.

The information gained from Malware analysis can be used to aquire a deeper understanding of already identified malware. It also helps in developing methods for automatic detection of malware in the wild.

2.1.1 Static Analysis

Static Analysis describes the way of analysis that does not run the malware. This type of strategy is fast and not very ressource intensive, which makes it quite applicable for real time protection inside AV Programs [SH18]. This type of protection is made possible by storing a huge amount of fingerprints generated by static Analysis inside a Database. These fingerprints are produced by putting the files through a hash Function such as SHA256. This produces a concise and mostly unique value. To determine whether or not a new file is malicious it just needs to be Hashed, with the same Algorithm used for the samples inside the database, and finally compared. If it doesnt match any of the entries it could be considered non malicious. The flaws of this approach are the following. The database could be incomplete, leading to new and unregconized Malware passing as harmless. Malware also applies many techniques to obfuscate itself. This will be discussed further in the chapter Obfuscation. But what this means is that malware can change the appearance of its source code each time it infects a new system or file. In general Static Analysis can be divided into Basic and Advanced methods [AH12].

Basic

In Basic static Analysis information is extracted without executing or looking at the actual instructions of the file itself. This technique is often very straightforward and can extract a certain amount of useful information about the functionality and behavior. Often a first step includes running the file through several AV Programs. This can be done on websites such as virustotal.com, giving a first hint at whether or not the sample is of malicious nature. Another popular technique is to create a Hash of the file and simply google the hash, which can also yield useful information and shorten the time spend analysing. Sharing these kinds of fingerprints is also very useful as these are often the basis of heuristic malware detection. Besides the aformentioned methods there are also programs such as strings, which return all the Strings inside a binary. This can already give an insight into potentialy malicious behavior. For example if the file contains the name of a known compromised domain.

Advanced

While basic static Analysis can be seen as a quick and dirty way to get a somewhat rough understanding of the binaries functionality it is often required to dig a little further. This is where methods such as reverse engineering are applied. Reverse engineering uses a disassembler to return the binary file into a state of human readable form. This method requires sophisticated knowledge of assembly and concepts of the targeted operating system and can give the analyst a full insight into what the loaded file actually does. The Chapter Antianalysis highlights some of the tactics applied by malware to complicate this process.

2.1.2 Dynamic Analysis

Besides static Analysis there is also dynamic Analysis, where the file is actually executed. While this requires a few steps of preparation, it is actually a very good way of determining if a file is

actually malicious or not. This method can also be found inside Antivirus Software in the form of sandboxes, which allow the runnig of the potential malware without harming the actual host system. This is quite computing intensive as it requires a full virtualization of the process as well as chosen components of the host machine. On the other hand this can produce much better results in terms of fingerprinting the sample as well as predicting if its behavior can be classified as malicious. Compared to static methods.

Basic

Requires the setting up of some kind of virtual environment, to safely run the potential malware. These are, as previously mentioned, called Sandboxes which run the file and record the changes made to the virtual system. These logs can then be looked into. This can result in an accurate fingerprint of the executable [AH12]. This comes as aformentioned with certain drawbacks such as the high computation costs especially compared to static analysis. Another non trivial problem is to decide which services and parts of the host system to emulate, as some malware might require certain aspects to be present to function as it was intended to do [Fer07]. There are also ways for malware to detect whether or not it is running inside a virtual environment.

Advanced

Applies debuggers to examine the internal state of the malware, at runtime. Gives dynamic insight into the program during its execution and even allows the analyst to change the flow of the executed program by assigning new values to variables. This can lead to a complete understanding of the examined sample.

3
Obfuscation Techniques

3.1 Polymorphism

One of the first techniques used by the earliest malware was simple encryption. In which the malware was shipped with a decryptor and an encrypted body. Each time the malware propagated it encrypted its body with a new key [RMI12]. This technique has one major flaw. While the body of the malware might be unrecognizable due to being encrypted, the decryptor itself is static. Meaning it can be easily detected by AV Programs. Polymorphism takes care of this problem, by changing the decryptor as well. So called Mutation Engines utilize common obfuscation techniques such as instruction replacement, register reassignment or insertion of dead code, just to name a few [RMI12].

This effectivly creates countless versions of decryptors, thwarting pattern based detection. This is where dynamic analysis comes into play. Even if an attacker can sneak his malware into a targeted system, at some point it has to be unpacked and run. The only problem left is that the body of the malware which contains the actual malicious code isnt changed at all, which can lead to it being detected after being decrypted for example inside a sandbox. Which leads to the next stage of obfuscation.

3.2 Metamorphism

This type of malware changes its body entirely from generation to generation. While this is hard to implement, it represents a powerful tool to thwart detection by any AV Vendor. As is effectivly changes the entire appearance of the malware while keeping its core functionality intact.

3.2.1 Metamorphic Engines

A metamorphic engine is the core of each metamorphic malware. The aformentioned changing of the whole malware is managed by it.

It includes the following components:

1. Disassembler
2. Code analyzer
3. Code transformer
4. Assembler

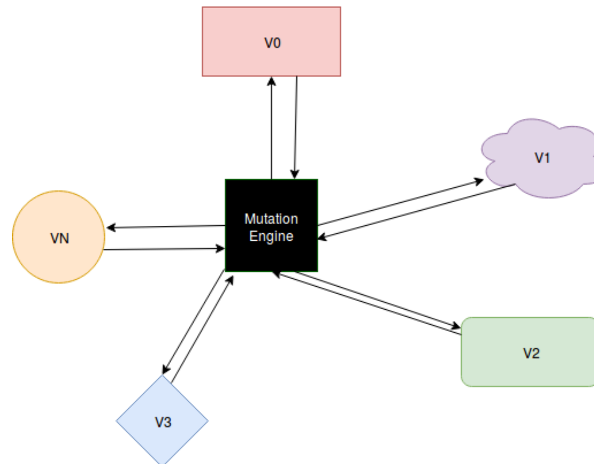

Figure 3.1: Metamorphic virus propagation scheme

The disassembler and code analyzer both prepare all the information needed for the code transformer to work. This includes recreating the assembly code from the binary as well as providing meta data to not break the functionality of the program once the transformer starts its work. The assembler component turns the new assembly code back into binary machine code [RMI12].

Register Swapping

A simple technique, which leads to registers being swapped inside the code. For example POP, ECX might be exchanged with POP, EDX. Both of which just take whatever is on top of the stack and place it into the given register.

Garbage Code insertion

Insertion of instructions which either never get executed or dont have any effect on the flow of the program itself. For example:

```
ADD EAX, 1
SUB EAX, 1
```

Subroutine permutation

Reordering parts of the malware body. This could be done by splitting the code into subroutines which then could be mixed up and connected with unconditional jumps in between them. With ***n*** different subroutines a morphing engine can produce ***n factorial*** different structures.

Substituting Instructions

In this case the mutation engine replaces instructions with some of their equivalents. For example:

```
MOV  EAX, EBX     <=>     PUSH EBX
                          POP  EAX
```

[Sri12]

4 Thwarting Analysis attempts

This chapter looks at some of the techniques applied by attackers to throw off and complicate the manual as well as the automated analysis of their malware. The following examples are given in x86 Assembly, because it allows attackers to hide their malicious code. This comes down to two facts. One is the variable-length encoding which trades simplicity for speed and brevity. The other is the permission to mix data and code inside a section. This can cause problems for the disassembler as it often leads to sections being interpreted as instructions that would never be executed at runtime [WZH+11].

4.1 Anti-Disassembly

As mentioned in the previous chapter disassemblers are a very effective way to extract information about the purpose and functionality of malware samples. While polymorphic as well as metamorphic malware constantly changes its binarycode it is still possible to analyze them using modern disassemblers such as IDA Pro. As previously mentioned the opportunity to break these tools comes from the fact that all of them make certain assumptions and have certain limitations. For example a disassembler only represents a byte as part of one instruction. Which means that if the disassembler is tricked into disassembling at the wrong offset, it would produce false disassembly [AH12].

4.1.1 Types of Disassemblers

In the following will be two basic approaches to disassembling discussed. The linear and the flow oriented Disassembler. In addition two methods on how to break both of those approaches will be highlighted.

Linear Disassemblers

Linear Disassemblers just iterate over the binary code and disassemble block by block. The size of each block is determined by the current op code that needs to be disassembled. While this is quite easily implemented it isnt really robust in terms of accurately disassembling code that has been specifically modified to thwart such attempts.

In general the instruction section of a binary contains code as well as data that when interpreted in context of its instruction makes sense. On the other hand if these data sections are interpreted on their own then this can lead to them being falsely resolved into instructions.

This is possible because the x86 Architecture allows the mixing of code and data. Which makes it easier to hide malicious code.

```
// Code fragment from the bastard disassembler at http://bastard.
   sourceforge.net/
// this code fragment makes use of the disassembly library libdisasm
// implementing a crude linear disassembler
char BUFFER[BUF_SIZE]; // contains the data to be disassembled
int position = 0;

while (position < BUF_SIZE) {
   x86_insn_t  insn;
   //disassembles one instruction
   int size = x86_disasm(buf, BUF_SIZE, 0, position, &insn);
if (size != 0){
    char disassembly_line[1024];
    printf("%s\n",disassembly_line);
    position += size;
} else {
   // invalid instruction
   position++;
}
x86_cleanup();
```

Problem

As previously mentioned the linear disassembler causes problems from time to time, even on non malicious samples. As it doesnt take the context of the current instruction being interpreted into consideration. This can lead to data being falsely interpreted as further instructions. Illustrated in the following example.

```
jmp ds:off_401050[eax*4]; switch hump
; switch cases omitted
xor eax,eax
pop esi
retn
;-----------------------------------
off_401050  dd offset_ loc_401020
            dd offset_ loc_401027
            dd offset_ loc_40102E
            dd offset_ loc_401035
```

Following the retn instruction, there are the pointer values of the switch case. Starting with the value 401020, which in memory will appear as 20 10 40 00 in hex. These 4 values make up 16 bytes of data, but can also be interpreted as instructions. Which the linear disassembler will do, producing wrong assembly code [AH12].

Flow-oriented Disassembly

This is the more advanced technique of reversing executables back into assembly. This time the disassembler actually looks at the instructions and for example notes if a conditional branch is encountered. From that it constructs a list of locations that need to be disassembled individually. This gives the disassembler the opportunity to decide which control path to follow first. The disassembler can even omitt code sections after an unconditional jump as they will never be reached.

```
test eax,eax
jz   short loc_1A
push Failed_string
call printf
jmp short loc_1D
;---------------------
Failed_string: db 'Failed',0
;---------------------
loc_1A:
        xor eax,eax
loc_1D:
        retn
```

The previous code is an example where the linear disassembler would fail as it would interpret the bytes after the jmp instruction and produce false assembly. The flow oriented disassembler, as previously mentioned, would omitt this portion of the code and only disassemble the parts that are actually reachable [AH12].

4.1.2 Anti-Disassembly Techniques

But how do attackers actually mislead disassemblers? In the following there will be some of the most popular techniques discussed.

Jump Instructions with the same targets

A popular technique is to put two conditional jump instructions next to each other. Both of them pointing to the same location.

```
jz loc_512
jnz loc_512
```

This sample is the equivalent of an unconditional jump. But the disassembler doesnt recognize it as such. Which leads to it disassembling the code after the second jump instruction. This could in turn be a section that never gets executed and contains data that could falsely be interpreted as instructions which would lead to wrong disassembly.

```
xor eax,eax
jz loc_4011C4+1
```

Another iteration of the same example would be this supposed conditional jump. This could effectively be replaced by a single jmp instruction. Because xor eax,eax would set the zero flag beforehand which would always trigger the jz instruction. While this is obvious for the human eye, this also has the potential to misguide the decompiler.

Impossible Disassembly

The previous examples all had some rogue byte strategically inserted into a point inside the program. These rogue bytes could easily be forgone by turning them into NOPs. But what if these inserted bytes are actually part of the program that gets executed at runtime? This results impossible disassembly which is only partially true because the code could be disassembled which would require a completely different representation of the code

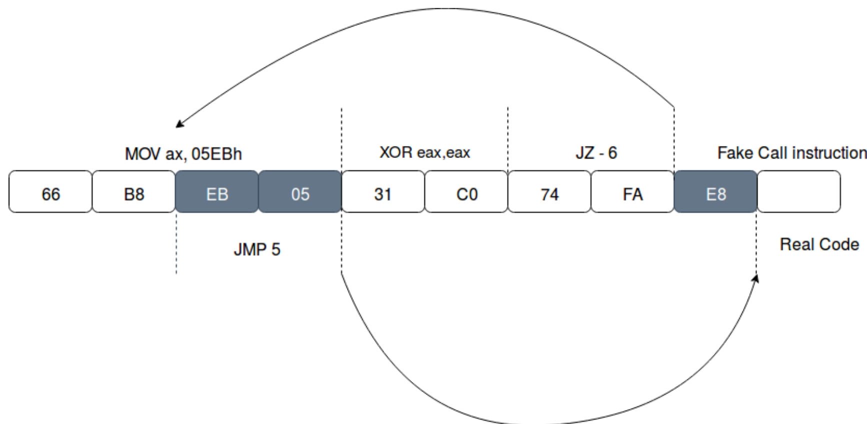

Figure 4.1: Example of an impossible disassembly situation

The figure 3.1 would trick the disassembler into exploring the path after the JZ -6 instruction. As a result it would falsely interpret the byte E8 as the opcode of a starting Call instruction. This would effectivly hide the next 5 bytes after that. Which in turn would produce incorrect disassembly.

Abusing Return Pointer

Normally the retn instruction is used to return the control back to the calling stackframe. But this doesnt always have to be the case. As retn is only a combination of pop and jmp.

```
004011C0   sub_4011C0
004011C0   var_4 = byte ptr -4
004011C0   call $+5
004011C5   add [esp + 4 + var_4], 5
004011C9   retn
004011C9   endp ; sp-analysis Failed
;---------------------------
004011CA   push ebp
004011CB   mov ebp,esp
...
```

This little code fragment shows how the retn statement can be used to confuse the disassembler by being abused to jump to the actual body of the function, instead of returning from the actual function itself. Outline of the body depicted at the memory adress 0040011CA. In this example the call function simply jumps to the position infront of itself, resulting in the adress of the next instruction being pushed on to the stack. The following **add** instruction then increases this adress by 5. Which makes the esp point to the value 0040011CA. This is the code of the actual function. The retn statement pops the value off the stack using it as the location of its **jmp** instruction. The disassembler now interprets this as the first function ending and another one beginning resulting again in false disassembly [AH12].

4.2 Packers

The following section will give a little insight into what packers are. In context of the windows file format PE

4.2.1 Overview

Packers are programs, such as upx, used to compress executables. The basic workflow of a packer is taking an executable and producing another one. This new executable could be compressed, encrypted or equipped with anti-disassembly properties. This new executable will unpack itself at runtime. Responsible for this is the Unpackingstub containing a stub engine.

In general there are several types of packers. Compressors, just like the name suggests, shrink the size of the original executable. Most of the time those dont apply any obfuscation techniques. There are also so called Crypters which encrypt and obfuscate the file. Protectors often combine the functionality of the two previously mentioned types, which makes them very attractive to use for attackers [YZA08].

4.2.2 Workflow

Assuming the target is a PE (Portable Exectuable). During the process of packing, the packer has to keep track of several things to preserve the functionality of the original executable. First it parses the whole internal structure of the PE file. It reorders the headers, import tables and export tables. During the packing process the packer can also apply such things as mutation engines to further armour the code and make it harder to Reverse Engineer.

4.2.3 The Unpacking Stub

In general when a program, in this case the PE, gets invoked it has an original entrypoint (OEP). From this point on the operating system will always start the execution of the file. In case of the packed PE the OEP gets replaced by the unpacking stup which contains a so called stub engine. It's engine has three main tasks. Unpack the executable into memory, resolve all the imports of the original executable and transfer execution back to the OEP of the original PE. This results in the original in our case a malicious program only being visible in RAM which effectivly doesnt leave any footprints on the targeted system itself. Excluding the functionality of the unpacked malware itself.

4.2.4 Unpacking Methods

There are different ways for malware analysts and hacker as well as AV Vendors to approache unpacking a suspicious file.

Manual unpacking

As the name suggests this methods consists of the analyst using a debugger to manualy run through the executable and extract for example the compression or encryption algorithms. This takes time and a deep understanding of assembly. Which doesnt make it really viable for example.

Static Unpacking

In static unpacking AV Vendors use tools such as Heaventools' UPX to automatically unpack the present file. This can be a very efficient way of unpacking files but this method runs into problems when malware authors apply custom packers.

Generic Unpacking

Generic unpackers utilize sandboxes or emulators to run the code until it is completely unpacked and then dump the processe's memory into a file. Heuristics are often used to determine the exact point in time. An example could be to wait for an intersectional long jump and check the ESP(Stack Pointer). Which might have drastically changed compared to the previous one. Due to the unpacking stub recovering the orginal state of the registers. Only determining the jump to the OEP isnt often enough as sophisticated packers use different techniques to unpack their content. For example some packers unpack their code on demand. This means that the OEP is reached before even most of the program is actually back into its original state. A malware author could also pack his program multiple times which complicates the problem even further [UPBSB15].

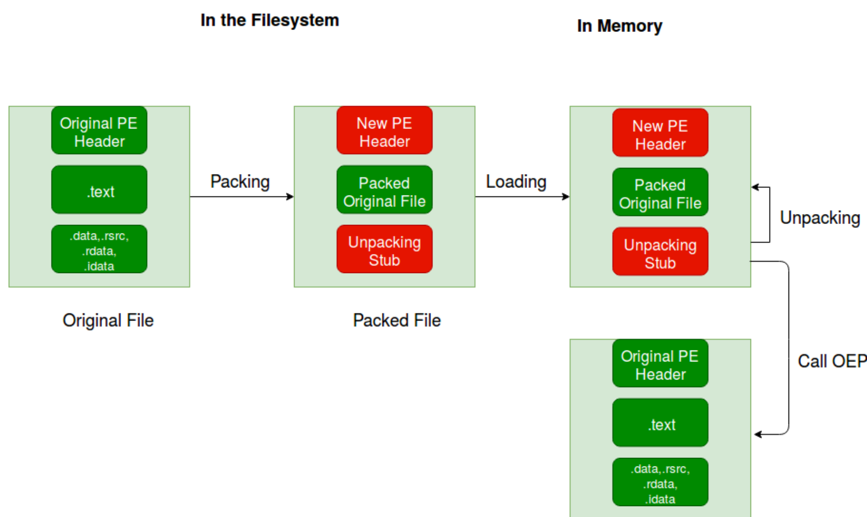

Figure 4.2: Illustration of an Packing/Unpacking cycle

4.3 Antidebugging

A malware equipped with anti-debugging properties can detect wether or not they are being executed by a debugger

4.3.1 Debuggers

Debuggers are pieces of software or hardware which are used to observe a running process. And enable someone for example a developer to examine and even stop the execution of a targeted process. Debuggers are as described in the chapter Basics a greate tool for the experienced analyst to extract important information about a malicious program. They not only allow a detailed insight into every state of a register but can also change the flow of the whole execution by reassigning values at certain points in time. This makes it a mighty tool for the experienced analyst and a big problem for malware authors trying to keep their secrets hidden. Which makes the development of anti-debugging techniques no surprise. To stop the execution of a process the debugger inserts INT instructions. These instruction are used to generate a software Interrupt. In particular the INT3 instruction.

4.3.2 Detecting Debuggers

Malware often uses anti debugging techniques to slow down the analyst. This may manifest in changing for example the flow of the execution. The following examples will look at some of the popular techniques used inside a windows environment.

Utilizing the Windows API

A very easy way of determining the presence of a debugger is to utilize functions from the Windows API. For example functions such as IsDebuggerPresent. This function searches the PEB (Process Environment Block) for the Field BeingDebugged. This is for obvious reasons not a very robust way to thwart debuggers, nevertheless it is a possibility.

Manually Checking PEB

Another way of retrieving the desired information is to manually check the PEB itself. While this requires a little bit more knowledge about the windows internals it makes omitting a direct call to the Windows API worth it. But how is this done? During runtime a process can access the PEB via the fs register. The code for extracting this information would look like this.

```
mov eax, dword ptr fs:[30h]
mov ebx, byte ptr [eax + 2]
```

Other methods might use the fact that the execution time of a program is highly warped inside a debugger. This could be done by taking two time stamps using the rdtsc instruction. This instruction returns the number of ticks since the last system reboot. If the difference between the two timestamps exceeds a certain amount then might hint at a debugger being attached to the program.

5 Conclusion

As the arms race between attacker and defender continues, more and more techniques are created to either detect or hide malicious software. This paper gave a short introduction into common mechanisms used by malware authors. But there are so much more interesting ways which were developed during this decades long arms race. Methods in which malware cryptographically linked itself to an infected host got proposed. By using encryption keys compiled from values unique to the infected host system itself. Other methods utilize costum bytecode, representing an arbitrary machine language which gets translated with the help of an attached interpeter [CS13].

As seen in the previous chapters malware applies a vast amount of techniques and tricks to mutate its appearance which makes manual as well as automatic detection really diffcult. Building robust detection methods to catch these types of malware isnt easy. It is an ongoing field of research. One promising approach seems to be machine learning. Taking information gathered by the previously established analysis techniques and building models which are able to detect and even match previously never seen samples [SH18]. This might bring an opportunity for effective and scalable methods involving real time protection of user systems.

Bibliography

[AH12] Michael Sikorski Andrew Honig. *Practical Malware Analysis, The Hand-On Guide to Dissecting Malicious Software.* No starch Press, Inc., 245 8th Street San Fransisco, CA 94103, 2012.

[CS13] Wenke Lee Chengyu Song, Paul Royal. The red book a roadmap for systems security research. Roadmap, Georgia Institute of Technology, 2013.

[Fer07] Peter Ferrie. Attacks on virtual machine emulators. 2007.

[MBA+13] Evangelos Markatos, Davide Balzarotti, Elias Athanasopoulos, Lorenzo Cavallaro, Federico Maggi, Michalis Polychronakis, Asia Slowinska, Iason Polakis, Magnus Almgren, Herbert Bos, Sotiris Ioannidis, Christian Platzer, Philippas Tsigas, Stefano Zanero, Dennis Andriesse, Martina Lindorfer, Farnaz Moradi, Zlatogor Minchev, Simin Nadjm-Tehrani, and Christian Rossow. *The Red Book. A Roadmap for System Security Research (Resumé).* 09 2013.

[RMI12] Babak Bashari Rad, Maslin Masrom, and Suhaimi Ibrahim. Camouflage in malware: from encryption to metamorphism. 2012.

[SH18] Alireza Souri and Rahil Hosseini. A state-of-the-art survey of malware detection approaches using data mining techniques. *Human-centric Computing and Information Sciences*, 8:1–22, 2018.

[Sri12] Sudarshan Madenur Sridhara. Metamorphic worm that carries its own morphing engine. Report, San Jose State University, 2012.

[UPBSB15] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo Garcia Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *IEEE Symposium on Security and Privacy*, pages 659–673. IEEE Computer Society, 2015.

[WZH+11] Richard Wartell, Yan Zhou, Kevin Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating code from data in x86 binaries. volume 6913, pages 522–536, 09 2011.

[YZA08] Wei Yan, Zheng Zhang, and Nirwan Ansari. Revealing packed malware. *IEEE Security & Privacy*, 6, 2008.