University of Applied Science
Bonn-Rhein-Sieg

Department of Computer Science

Sankt Augustin, Germany

Malware Analysis

Tools and techniques

March 4, 2018

Author:
Jan Arends
jan.arends@mailbox.org

Supervisor:
Prof. Dr.-Ing. Kerstin Lemke-Rust

Declaration of Authorship

Contents

List of Figures

List of Abbreviations

API Application Programming Interface

C and C Command and Control Server

DNS Domain Name System

PCAP Packet Capture

IRC Internet Relay Chat

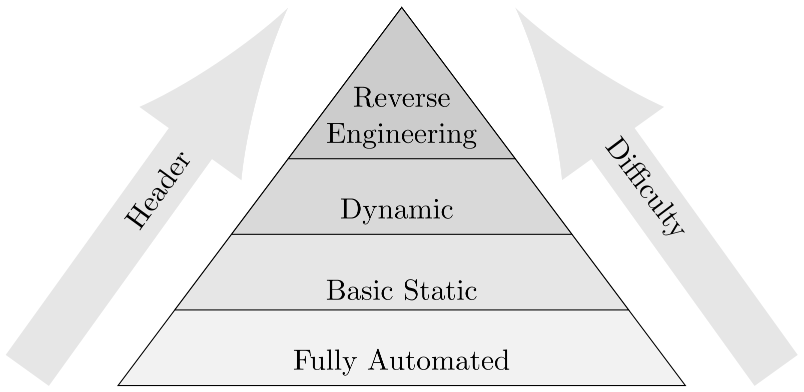


Figure 1: Categories of Malware Analysis

1 Introduction

The number of software developed for malicious purpose is still growing. Computer criminals are no longer some script kiddies. They've become very organized and professional and are highly motivated to gain deep knowledge to fulfill their goals. The more complex and intelligent a malware gets the more time consuming is the process of analyzing it. Luckily many tools out there has been adapted to handle such complexity. Some of them will be introduced shorty later.

The focus of this paper is the brief introduction of basic techniques and entry points to analyze malware. An overview of all associated categories is shown in Figure 1. Beginning at the bottom this is the typical order an investigator would go through. The higher the entry in the pyramid, the more knowledge and practice is needed to stay efficiently. The *fully automated* approach is not covered here, since it obviously uses the listed techniques. The term *reverse engineering* is such a massive topic which is why it's stated separately. Technically it belongs to both of the main categories of this paper where it will be covered: The static- and the dynamic analysis. They'll be discussed successively. Associated anti techniques are mentions rarely to stay focused. Also, examples are limited to Windows and Linux.

Besides, the reader should be aware of the differences between malware analysis and malware detection. Due to the fact that software might also be analyzed for detection of maliciousness the techniques of the two terms can overlap but the basic idea differs. However, just the analyzing techniques of software which is already considered as malicious is covered here.

1.1 Incident Response

Usually the process of analyzing starts as response to a security incidents e.g. within an organization. To respond to and recover from an incident appropriately the nature and impact of the incident have to be explored. Following questions [1] about the malware should be answered in the end of the analyze process.

- What type of malware is it?
- What is the intended purpose and how does it accomplish it?
- What is the functionality and capability?
- What affect does it have on the system?
- How does is spread?

An analyst can't possibly understand every detail of the malware. The focus should be on the key features [2, p.5].

1.2 Lab environment

Analysis of malware means to work with potential damaging code. Therefor it requires a safe and secure lab environment to overcome the risk of damage. To ensure that the code is contained and unable to connect to any production system or the internet the program file should be placed on an isolated or "sandboxed" system with a simulated network. Otherwise it can affect the production system which could end in a lot of trouble. [1]. It might be worth to take a look at the common Linux distribution REMnux: "A Linux Toolkit for Reverse-Engineering and Analyzing Malware" [3].

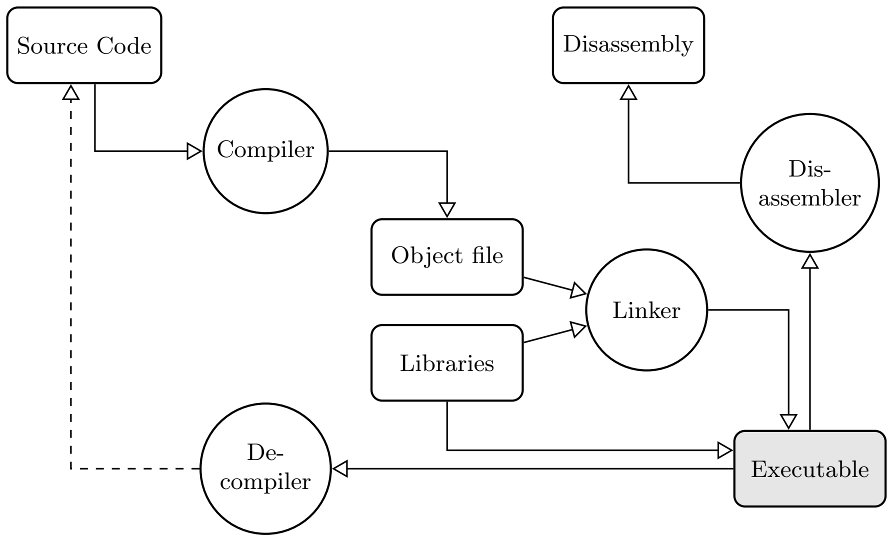


Figure 2: Steps to consider during static analysis

2 Static Analysis

Let's begin with static analysis. It's the process of analyzing an executable binary file without actually executing it. It usually the first step in studying malware and necessary to gain enough information about the file to categorize it and discuss further steps and techniques [1].

But before getting into deeper explanations about analyzing an executable it is worth to recall the way such file is created usually. Figure 2 gives an overview of files and components involved.

While compiling the source code into a binary executable some information (e.g., the size of data structures or variables) gets lost. [4]. An executable file itself can still offer various hints about the way it is created. This might be important for analysis or later reverse engineering. Tools and techniques which extract Metadata or embedded information make this possible an will be discussed in this section.

Because the code in the executable is based on the targeted machine language it meant to be processed by a machine and isn't meant to be human readable. This makes the direct analysis of this code impossible. However tools and techniques exits to recreate readily code from the binary exists, e.g. disassembler or decompilerand. They will be discussed later.

Platform	Format
Windows	Portable Execution
Linux	Executable and Linkable Format
MacOS	Mach-O

Table 1: Common File Formats

Due to the time consuming process of analyzing the reverse engineering code, the analysis process sometimes reduces the applicable static methods to those that retrieve the information from the binary representation of the malware [4]. At least if the analyst has a short and limited time frame.

2.1 File Identification

2.1.1 File Formats

It's relevant to identify the file format it is to be analyze to perform a appropriate analysis. Therefore file signatures are used. A file signature is a unique sequence of identifying bytes written to a files' header. The signatures differ from file type [1, p.394]. To inspect the file signature either a file identification tool is used or the binary is reviewed manually using a hexadecimal viewer or editor. Table 1 shows common file formats for the major platforms the malware analyst has to deal with.

2.1.2 Fingerprint

To produce a unique hash value of the malicious file is a common technique to start with [2, p.8]. Therefor the algorithms MD5 or SHA-1 are used. The hash acts like a fingerprint and is used to identify the file within the community. In addition current antivirus software are base on such fingerprints, at least partially. Hence it's also a good step to run the file through multiple antivirus programs. Websites such as `www.virustotal.com` offers to scan the file using multiple antivirus engines. In the end the investigator gets indications about weather the file was being analyzed or captured before [2, p.10].

2.1.3 Obfuscation

Obfuscation is a technique against the analyzing process, also called a anti technique. The code is converted to a new different version without affect to the functionality. Hence e.g. fingerprints are different. The goal is to make the analyzing process more difficult. A simple example of a obfuscation technique is the adding of some ineffective instructions to the program,

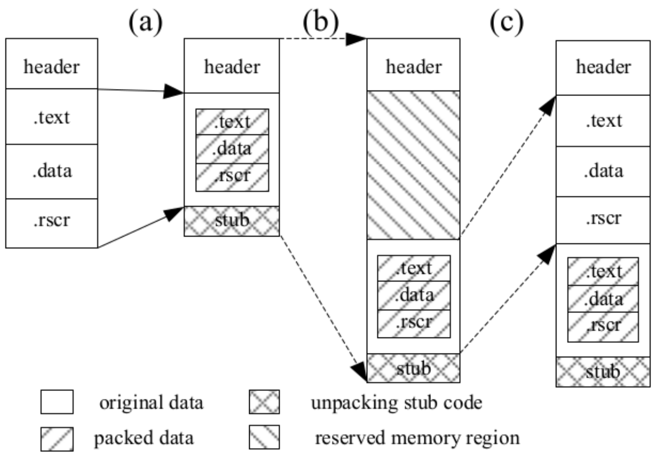


Figure 3: Packing and unpacking process

namely junk code insertion [5].

With regard to section 3 another technique is worth mention. To go a step further malware authors obfuscates the entire file with a program called *packer* illustrated in Figure 3 from [6]. A packer typically encrypts the malware as illustrated in Figure 3 (a). When loaded into memory like in Figure 3 (b) it unpacks itself as shown in Figure 3 (c).

Clearly, it's important to analyze obfuscation techniques like this to efficiently analyze malware in the static process. The detection is possible because although the encrypted part is unique and hence the signature is not well known, the decryptor remains constant from generation to generation [5]. This can be used as an indicator. To address the shortcoming of detection and static analysis, malware authors uses oligomorphic and polymorphic techniques which generates different decryptors [5]. These techniques are also addressable, but since it's loaded in memory it's rather a step in the dynamic analysis process. Note: A high entropy value properly means compression or/and encryption is being used.

2.2 Embedded Artifact Extraction

The file identification process gives first clues and hints about the malicious file. It can then be a guide to choose the next steps which may must be taken before going further, e.g. talk to other analysts or take care of obfuscation techniques. Otherwise a great number of other potentially important facts can be gathered from the file / binary code itself which will be discussed now.

An executable can either be statically linked using a linker or dynamically

Linux	- GNU C Library manual - The Open Group index of functions - Linux man-pages
Windows	- Windows API reference - Microsoft DLL Help Database - TechNet Library

Table 2: Web references

linked during runtime. Dynamic linking relies on invoking shared libraries and functions which must be load into the host systems memory, to successfully execute. Hence these libraries are refereed to as *dependencies.* Conversely, statically linked executable already contain all functions to successfully execute, therefore don't have that kind of dependencies [1, p.384]. In general all evidences should be inspected separately to have a clear file context and to promote organization in the analysis process. Anyway redundant information gathering across different analysis entry points is a good thing.

During the inspection of embedded entities its handy to have reference web sites available for quick perusal [1, p. 316,409]. The following table show typical references for the discussed platforms.

2.2.1 Strings

Usually the code of a malicious file has sequences of characters embedded, namely Strings. They're represented in either ASCII or Unicode format and easy to extract. The tool *Strings* is often the first choice to do so and available for all common platforms. Although *Strings* has a couple of options to choose from, false- positives will properly happen and must be identified manually. Since they don't represent legitimate text they're obvious to identify [2, p.12].

The following list shows some examples of what the strings might represent [2] [7]:

- Calls to functions, shared libraries and APIs
- Error messages and comments
- Network information
- IRC Channels and C&C Server
- Directory and file names
- Compiler and version

After interpreting the strings with the appropriate documentation a first conclusion can be done. The basic functionality and potential behavior might be identified. The analyst might already get a feeling about the functionality. In the next steps the analyst will properly face the same values, but to double check artifacts is a good thing.
The fact, that malware authors often plant strings in their code to throw digital investigators off track [1, p.316] makes this techniques harder. To plant strings is a fairly common technique and is mostly just recognizable through experience.

Overall it's still a good way to get an overview of what is in the file. Unless it's not obfuscated in some manner. In that case, a noticeable small amount of strings would be appear and further investigations about the packing should be done.

2.2.2 Dependencies

Most malware is dynamically linked hand hence have dependencies. To know about them is a common and important technique and will come across later again.
To discover them by identifying shared libraries within the string search is a good starting point. Nevertheless additional research has to be done to further explore the binary dependencies. Once the dependencies are disclosed they can be looked up in the proper documentation. Thus the behavior can be explored.

Tools to generate a list of all dependencies a file has are available for all common platforms. In a UNIX/ Linux system the CLI tool *ldd* is often the first choice. For each dependency, ldd displays the location of the matching object and the (hexadecimal) address at which it is loaded [8].

The most popular tool for Windows is called Dependency Walker. It provides both a command line interface (CLI) and a Graphical user interface (GUI) which offers the investigator a granular perspective trough a complete module dependency tree diagram. It lists all required files, all functions that are actually called and all those which can be imported from the file [9].

Knowing the dependencies allows a good guess about what the program does and is therefore one of the most useful pieces of information that can be gathered about an executable during the basic static analysis [2, p.15].

2.2.3 Symbols

Symbols are like identifiers for program variables and functions. They're used for interpreting or debugging software. They're typically stored in a *symbol table* and like the file dependencies easy to access. Required that the author didn't discarded the symbol information from the file, which is fairly possible. An investigator may gain insight into the program's capabilities from symbols.

- Function name and addresses
- Variable names, addresses, data types and scopes
- Structure and class definitions

Similarly, if a hostile program is compiled in debug mode it will provide additional information such as source code and debugging lines [1, p.290,421].

A structured output of symbol information on Linux offers *eu-nm* utility as part of the *elfutils* suite. In case the binary is striped the symbol table can be further explored by using the *eu-readelf* utility. To display the symbols present in a PE on Windows the program DUMPBIN with a specific argument can be used.

2.2.4 Metadata

Also an executable file has metadata resides in it and can provide valuable insights of the file. Metadata in the context of an executable file can contain information about the origin, ownership, and history of the file rather then technical information [1, p.330]. They can be gathered from various parts of the file structure. During the creation of an executable like shown in Figure 2, "including the high-level language in which the program was written, the type and version of the compiler and linker used to compile the code, and the date and time of compilation", as well as timestamps, location and previous file names [1, p.330].

2.3 Analyzing reversed engineered code

Machine code is way too difficult for a human to comprehend. Therefor security analysts often have to resort to methods to reverse engineering the code. This could be done either by using a disassembler or a decompiler which will be introduces next. The usage is often considered as advanced malware analysis because it's very difficult and time-consuming [10].

2.3.1 Disassembly code

As mentioned earlier, malware samples are mostly just available in binary form at the machine code level. However, it's possible to recover a low-level language code from machine code reliably and consistently [2, p.67]. Remember, a low-level language is a human-readable version of a computer architecture's instruction set [2, p.67] The assembly language is the most common of its kind. The code can be generated from a binary using a disassembler.

It's theoretically a perfect chance to analyze the malicious behavior. But with regards to the high-level language the malware in written in, the instructions has a one-to-many relationship to assembly code. Meaning that one high-level instruction consists of several low-level instructions [11, p.1]. Hence the assembly code is way larger and overall harder to read and understand then the original high level representation.

Here's an example from [2, p.116]. Listing 1 shows a for-loop written in C and Listing 2 shows the disassembled version of that structure:

```c
int i;
for(i=0; i<100; i++) {
        printf("i equals %d\n", i);
}
```

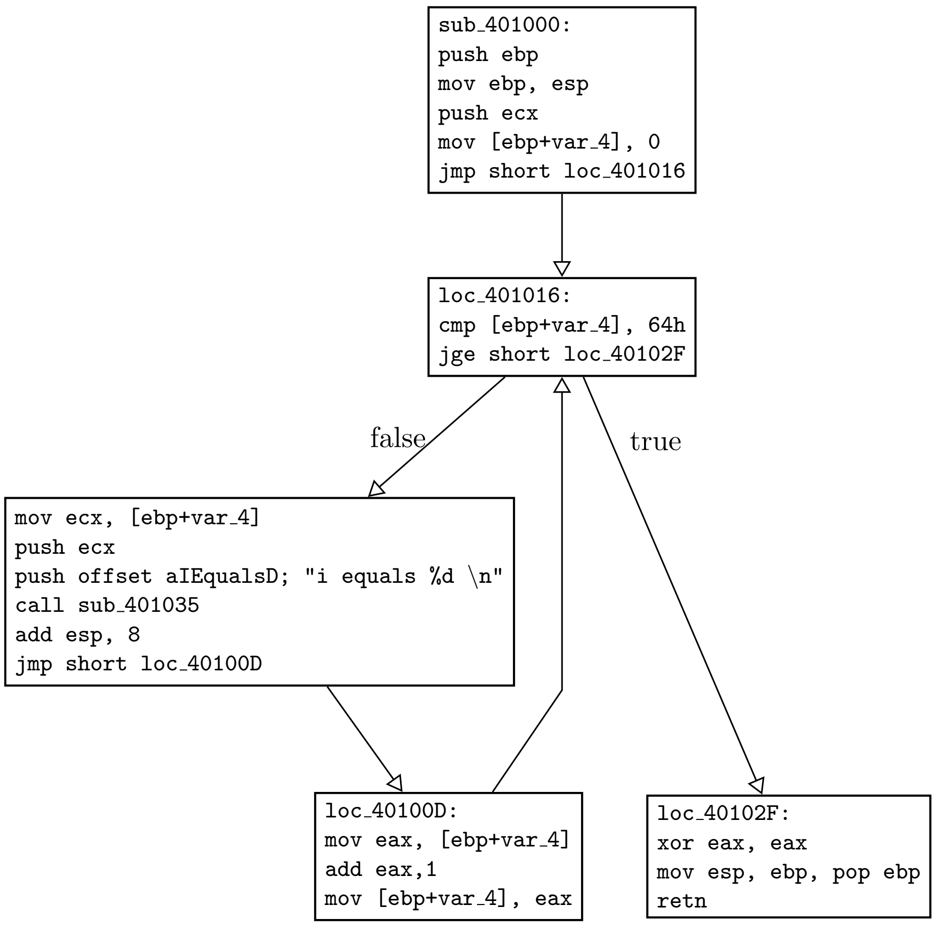
Listing 1: For-loop written in C

```
00401004                 mov     [ebp+var_4], 0
0040100B                 jmp     short loc_401016
0040100D loc_40100D:
0040100D                 mov     eax, [ebp+var_4]
00401010                 add     eax, 1
00401013                 mov     [ebp+var_4], eax
00401016 loc_401016:
00401016                 cmp     [ebp+var_4], 64h
0040101A                 jge     short loc_40102F
0040101C                 mov     ecx, [ebp+var_4]
0040101F                 push    ecx
00401020                 push    offset aID ; "i equals %d\n"
00401025                 call    printf
0040102A                 add     esp, 8
0040102D                 jmp     short loc_40100D
```

Listing 2: Assembly code for the for loop example above

To not evaluate every individual assembly instruction it is worth analyzing instructions as groups [2, p.109]. Techniques exists to recognize all common high-level constructs such as loops and conditional statements. They can be visualized with control flow graph to obtain a high-level picture of the code functionality.

Figure 4 shows such graph for the for-loop example above. Although the explanation of assembly code is out of the scope of this paper, it's obvious how this visualization can speed up the process.



```
sub_401000:
push ebp
mov ebp, esp
push ecx
mov [ebp+var_4], 0
jmp short loc_401016
```

```
loc_401016:
cmp [ebp+var_4], 64h
jge short loc_40102F
```

```
mov ecx, [ebp+var_4]
push ecx
push offset aIEqualsD; "i equals %d \n"
call sub_401035
add esp, 8
jmp short loc_40100D
```

```
loc_40100D:
mov eax, [ebp+var_4]
add eax,1
mov [ebp+var_4], eax
```

```
loc_40102F:
xor eax, eax
mov esp, ebp, pop ebp
retn
```

Figure 4: Graph view for the for loop example above

The state of the art software in this case is IDA from Hex-Rays. It offers a disassembler for a large number of processors, code graphing and a hole lot of other features in addition.

2.3.2 Decompiled code

To speed up the reverse engineering process *decompilers* are very important tools. They enable to reason about the high-level form of code instead of its low-level assembly form [10]. In addition, while high-level abstractions (eg. if-else-statements or loops) are getting lost during compilation a decompiler tries to recover these abstractions to analyze the code more easily [12].

Although decompilers has been studied over many years and significant advances have been made most decompilers produce very complex and unreadable code. Therefor analysts frequently go back to analyzing the assembly code [12].

In in 2016 and 2017 major improvements has been done.
For instants the DREAM Decompiler [10] and the extended Version DREAM^{++} [12].
Also the retargetable Decompiler *RetDec* [13], which was developed by AVG and published by Avast after the bought the company in December 2017, looks promising.

3 Dynamic Analysis

Dynamic analysis is the process of analyzing the malicious software while it's actually executing. It's usually done right after the static analysis. It can overcome some of the anti techniques which might have limited the previous steps like obfuscation. But also the basic behavioral analysis do have their limitations, because not all code paths may execute when a piece of malware is run [2, p.40]. In addition, also here do anti techniques like monitoring-, virtual machine- or memory capturing detection exist, but as mentioned at the beginning, this is out of the scope of this paper. At this point the more advanced dynamic analysis, which will be at the end of this section, helps out.

3.1 Malware characteristics

For the later monitoring section a recap of the basic actions might be helpful. In [14] the malware characteristics are good summarized. The authors defined basis behavior operation sets which are categorized as followed:

- File Actions $= \{Create, Delete, Read, Write, Rename\}$
- Process Actions $= \{Create, Terninate\}$
- Network Actions $= \{TCP, UDP, IP\} \oplus \{Listen, Connect, Send, Recv\}$
- Registry Actions $= \{OpenKey, CloseKey, CreateKey\}$
  $\wedge \{SetValue, DeleteValue, QueryValue\}$

These are basic actions also a non malicious software would do. But by combining these operations and studying the order and relationships between them it becomes more meaningful and significant in the malware analysis process. Hence those characteristics are often merged to a signature for official identification and detection as well.

In [14] they defined some of those compositions which an investigator of malware is looking for in the dynamic analysis process. Such behavior can be discovered by monitoring the system in different ways.

3.2 Monitoring

3.2.1 Process and threads

A process is a program in execution. Hence it seems to be an obvious step in the dynamic analysis process. To explore all the processes related to the malware gives important insights to the functionality. In particular the investigator wants to know which processes are running, created or terminated while the malware is executing. This gives great indications of the functionality. The investigator might see for example if the malware is killing an anti-virus process or which programs the malware loads into memory.

In addition a bunch of other interesting information associated with each process can be displayed and analyzed. For example all shared libraries the malware actually loaded, open files and handles or on which port the process is listening.

As tool examples there is the 'Process Monitor' for Windows. It's also a GUI tool which is used to monitor activities related to the next subsections, like registry, file system or networking. In case of the process monitoring if also offers a nice timeline for a fast understanding of the processes order of appearance. Even the shortest appearance of a process will be visible there. The analysis on a Linux system involves tools like `htop`, an interactive process viewer for all kind of information about processes.

A common anti technique which is worth mention is called 'process replacement'. It involves running a process on the system and overwriting its memory space with a malicious executable to provide the malware with the same privileges as the process it is replacing [2, p.49]. To overcome this anti technique the memory image has to be analyzed. This will be discussed later in section 3.3.

3.2.2 System or API calls

Functions that form a coherent set of functionality, such as manipulating files or communicating over the network, are often grouped into an API [4, p.7]. They are usually provided by the operating system to perform common tasks. The term Windows API refers to a set of APIs that provide access to different functional categories, such as networking, security, system services, and management [4, p.7].

For the analyzing point of view, system calls are similar. They're also provided by the operating system as a special well-defined API which separate user-mode from kernel-mode. To interact with the system and the environment the malware has to invoke system calls since only the kernel-mode has direct access it. Typical examples are calls for file modifications or registry changes but also network functionality is provided by the operating system via system calls.

Not just the calls itself are pretty useful, also the tracking of parameters and function return values enables insights in a dynamic analysis approach. It enables to identify the the correlation of individual function calls that operate on the same object [4, p.9].

Overall the identification of all these calls and corresponding parameters it's a great chance to get further knowledge of the behavior. Mainly because APIs and system calls are excellent documented for all platforms, see Table 2.

3.2.3 File system and Registry

Also important are the file system or registry interactions which are related to the malware processes. Since the technique is similar they are discussed here together. Even though they could be identified by investigating API or system calls, it's always best practice to double check, due to the possibility to miss important information in previous steps.

Since malware often adds entries to the registry for persistence or configuration data which e.g. allows the malware to run automatically when the computer boots [2, p.172], is a prefect place to look on. Equivalently are files the malware uses to store data, like for arrived data from a remote server. To identify both of these interactions making and comparing *snapshots* is the key here.

On Windows there is an easy to use tool calls *Regshot* to take two snapshots. An analyzer may want to take a snapshot before executing the malware and the second right after it has finished. Which key and values were added or modified in the registry during execution can be seen in the results Regshot is giving.

3.2.4 Network

To operate as desired modern malware often requires some form of Internet access. Additional components or configuration data might be downloaded [4] via C&C or IRC server. To figure out all those activities the network traffic is has to be explored and the the protocol has to be reverse-engineered.

As already mention at the beginning of the paper, a simulated network is the precondition. Commonly used network services like DNS, mail, IRC, and file servers has to be simulated, and all traffic has to be redirected to these services. If the network is simulating properly the malware will exhibit its malicious behavior and the analysis is completely self contained [4].

Normally this process is using PCAPs which is an API for recording network traffic between to points in time. It's used by e.g. *Wireshark, tcpdump or Bro*, which are all commonly used tools in network analysis. Hence network communication can already be explored during the API or system call monitoring process.

In [15] they stated the importance of network behavior analysis due to the limitations of static and other dynamic methods. These limitations are based on attempt to evade detection or analysis of the malware which the authors usually have. With regards to the data the malware needs to download to conduct its attack, it is difficult to evade network behavior based methods.

3.3 Memory forensic

The powerful analysis of memory enables a deep look into the systems behavior. Every action the operating system or an program is taking results in specific modifications to the computer's memory or RAM. The action performed can often persist a long time after it was taken [16, p.XVII]. Additionally, it "provides unprecedented visibility into the runtime state of the system which processes were running, open network connections, and recently executed com- mands" [16, p.XVII]. Besides, critical data often exists exclusively in memory, like encryption keys hence unencrypted messages and content.

Although the inspection of the mentioned entry points can yield compelling evidence, it is often the contents of RAM that enables the full reconstruction of events. It provides the necessary puzzle pieces for determining what happened before, during, and after an infection by malware or an intrusion by advanced threat actors [16, p.XVII].

To archive all the benefits two steps are required:

1. Memory acquisition
2. Memory analysis

Aacquisition is the process of copying the contents of volatile memory to non-volatile storage [16, p.69]. It can be done by capturing, dumping or sampling the memory. With regards to the time of acquisition multiple approaches exist, e.g. terminating-based interval-based or trigger-based [17]. As usual this happens a safe and controlled environment and stored as an memory image file for further investigations on another system. Since there's no possibility to use system APIs when parsing a memory image offline it's possible to also get a look in unallocated data or data which was hidden by the malware [18].

In the analysis process itself the list of entry points is incredibly long. Almost everything can be found in the memory image. E.g. processes, network, file system or registry artifacts. Even disk artifacts because all file actions leave traces in memory. Organizing the data with timelines based on the temporal relationships between digital artifacts is a traditionally leveraged technique [16, p.537].

The *Volatility* Framework is a common collection of tool which makes the extraction of all those digital artifacts possible. It provides the investigator with a single, cohesive and open source framework for all needs.

3.4 Debugging

"A debugger is a piece of software or hardware used to test or examine the execution of another program" [2]. It is usually used for developing or troubleshooting software since it gives insight into what a program is doing while it is executing.

Most software developers are familiar with high level language debuggers but because no high level code is available during malware analysis, the use of assembly level debuggers is necessary. These operate similarly. The user can step through the instructions, set breakpoints to stop on specific lines of the assembly code, and examine memory locations [2, p.168].

A debugger has normally two ways to debug a program. Either the user starts the program with the debugger or the user attaches the debugger to a program that is already running. The second way is useful to debug a process that is affected by the analyzed malware [2, p.169].

Beside a debugger has a couple of functions necessary to know for the analysis process. First the *single-stepping*. It's not recommended to step through every single instruction in the entire program. To be selective about the code to analysis is more efficient. Furthermore to decide whether to *step-over* a function call or to *step-in* the function has to be reasonable. To step-over wrong function could end up missing important functionality while stepping into a function significantly increase the amount of instructions needed to be analyzed. Last but not least the pausing function using breakpoints. These allow the user to examine a program's state [2, p.169f.]. A common x86 debugger for malware analysis is the easy to use and plug-in rich *OllyDbg*.

4 Conclusion

4.1 Summery

A lot of techniques, tools and entry points for the process of analyzing malware are now introduced. To perform basic static analysis it's possible to gain a certain amount of insight into the functionalities, using a suite of relatively simple tools [2, p.26]. To become really successful in the end of the process more advanced techniques, like usage of a debugger, is needed. A working knowledge of assembly and the disassembly process is the key here. This can just be gained by practice [2, p.85].

The next step is to run the malware during dynamic analysis where the findings from the basic static analysis can be confirmed or confuted. As for the static approach also the dynmaic is divided into basic and advanced levels. The basic analysis is mainly to monitor the systems interactions. For more and detailed information of the runtime state of the system investigations of the computers memory can be done. The use of a debugger might be easier and is also a very common techniques to analyze malware.

4.2 Prospect for the future

Overall it will always be a race between malware authors and analysts. Anti techniques will get even smarter, but detection of those as well. However the growing research of this topic will show where the analysis process will go. The fully automated approach might be satisfying for all kinds of malware in the near future. The artificial intelligence, deep learning and neural networks could assist to reach this goal.

References

[1] C.H. Malin, E. Casey, and J.M. Aquilina. *Malware Forensics: Investigating and Analyzing Malicious Code.* Elsevier Science, 2008.

[2] M. Sikorski and A. Honig. *Practical Malware Analysis: A Hands-On Guide to Dissecting Malicious Software.* No Starch Press, 2012.

[3] *REMux.* www.remnux.org/.

[4] E. Kirda M. Egele, T. Scholte and C. Kruegel. *A Survey on Automated Dynamic Malware-Analysis Techniques and Tools.* ACM Computing Surveys, Vol. 44. ss, 2012.

[5] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *International Conference on Broadband, Wireless Computing, Communication and Applications*, 2010.

[6] G. Yuxin, L. Zexin, and L. Yuqing. Survey on malware anti analysis. In *Fifth International Conference on Intelligent Control and Information Processing*, 2014.

[7] D. Distler. *Malware Analysis: An Introduction.* SANS Institute, 2007.

[8] M. Kerrisk. *Linux Manual Page: ldd(1).* https://www.kernel.org/doc/man-pages/, September 2017.

[9] *Dependency Walker: Using Dependency Walker for General Information about Modules.* www.dependencywalker.com/help/html/overview_3.htm.

[10] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations. In *??*, 2015.

[11] James T. Streib. *Guide to Assembly Language - A Concise Introduction.* Springer, 2011.

[12] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *2016 IEEE Symposium on Security and Privacy*, 2016.

[13] Avast Software. *Retargetable Decompiler RetDec.* https://retdec.com/, December 2017.

[14] P. Ren, Liu W., Sun D., Wu J., and Liu K. Analysis and forensics for behavior characteristics of malware in internet. In *14th Annual Conference on Privacy, Security and Trust (PST)*, 2016.

[15] T. Shibahara, T. Yagi, M. Akiyama, D. Chiba, and Takeshi Yada. Efficient dynamic malware analysis based on network behavior using deep learning. In *2016 IEEE Global Communications Conference (GLOBECOM)*, 2016.

[16] M. Hale Ligh, A. Case, J. Levy, and A. Walters. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory.* John Wiley & Sons, Inc., 2014.

[17] T. Teller and A. Hayon. Enhancing automated malware analysis machines with memory analysis. BlackHat, 2014.

[18] T. Haruyama and H. Suzuki. *One byte Modification for Breaking Memory Forensic Analysis.* `https://media.blackhat.com/bh-eu-12/Haruyama/bh-eu-12-Haruyama-Memory_Forensic-Slides.pdf`, 2012.