NTNU
Norwegian University of
Science and Technology

Automated Analyses of Malicious Code

Kris Mikael Krister

Master of Science in Computer Science
Submission date: June 2009
Supervisor: Torbjørn Skramstad, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Malware (malicious software) is a general term for any malicious program such as a virus, worm, trojan, bot or rootkit. The prevalence of malware is rapidly growing on the Internet and poses an increasing threat to computer systems.

Effective incident handling can be achieved by analysing malware in order to understand its functionality and capacity. This may involve analysing already compromised systems or assessing the level of risk for systems being exposed to such a threat.

However, malware analysis is labour intensive and time consuming, and does not scale well with the ever increasing prevalence of malware. Automating parts of the analysis process can reduce the required amount of human intervention, and save precious time during the analysis.

Techniques involving running malware in a controlled and secure environment whilst monitoring its behaviour is often referred to as dynamic analysis. This project should focus on studying dynamic malware analyses, and automate such a process.

The student is free to automate any analysis process that is regarded dynamic, but it would be preferable if the information gained from the automated analysis is actionable. That is, the information gained is directly useful for handling the particular incident(s) where the malware is involved.

The project may optionally look into integrating the final system with NorCERT's internal system for handling malware samples.

Assignment given: 15. January 2009
Supervisor: Torbjørn Skramstad, IDI

Preface

This report is the result of a problem stated by the Norwegian Computer Emergency Response Team (NorCERT), a division in the Norwegian National Security Authority (NSM). The task was structured and written by Kris-Mikael Krister, attending the Norwegian University of Science and Technology (NTNU), Department of Computer and Information Science (IDI) and was accomplished as a master's thesis in the late spring of 2009. I would like to thank my internal supervisor, Torbjørn Skramstad from IDI NTNU for guidelines and support when writing and structuring the thesis, and my external supervisor Lars Haukli from NorCERT for assistance regarding the thesis' content, and facts about the malware topic.

Oslo, June 2009

Kris-Mikael Krister

Summary

Sophisticated software with malicious intentions (malware) that can easily and aggressively spread to a large set of hosts is located all over the Internet. Such software struggles to avoid malware analysts to continue its malicious actions without interruption. It is difficult for analysts to find the locations of machines infected with unknown and alien malware. Likewise, it is hard to estimate the prevalence of the outbreak of the malware. Currently, the processes are done using resource demanding manual work, or simply rough guessing.

Automating these tasks is one possible way to reduce the necessary resources. This thesis presents an in-depth study of which properties such a system should have. A system design is made based on the findings, and an implementation is carried out as a proof of concept system. The final system runs (malicious) software, and at the same time observes network traffic originating from the software. A signature for intrusion detection systems (IDSes) is generated using data from the observations. When loaded in an IDS, the signature localises hosts that are infected with the same malware type, making network administrators able to find and repair the hosts. The thesis also covers a deep introductory study of the malware problem and possible countermeasures, focusing on a malware analyst's point of view.

Contents

List of Figures v
List of Tables vii
Listings ix

1 Introduction 1
1.1 Terminology and acronyms 1
1.2 Result goals 3
1.3 Methodology 3
1.4 Related work 6
1.5 Document structure 6

2 Background and Rationale 9
2.1 Malware propagation 9
2.2 Techniques for controlling malware 12
2.3 Custom made malware 15
2.4 The development of malware 16
2.5 Threats from malware 22
2.6 In need for more effective countermeasures 28
2.7 Computer emergency response teams (CERTs) 30
2.8 Summary 33

3 Phases of a Malware Analysis 35
3.1 Structure of a malware sample 35
3.2 Surface scanning 37
3.3 Dynamic malware analysis 38
3.4 Static malware analysis 38
3.5 Finalising the analysis 39

4 Dynamic Analyses in Depth 41
4.1 Analysis methods 41
4.2 Information prioritisation 46
4.3 Available complete sandbox-solutions 47
4.4 In need for applications running locally 48
4.5 Available smaller dynamic analytical tools 48
4.6 Summary of described tools and solutions 49
4.7 Problems with dynamic analysis 49

5 A Malware Analysis Scenario 55

5.1 Initial setup and environment overview 55
5.2 Surface scanning 56
5.3 Dynamic analysis 59
5.4 Reflections 61

6 Concretisation of the Task 63

6.1 Related work 63
6.2 Available approaches 65
6.3 Selecting an approach 72
6.4 Selecting products as base 73
6.5 Zero Wine in detail 79
6.6 Summary of selections 81

7 Requirements Specification 83

7.1 High level requirements 83
7.2 Use case analysis 84
7.3 Overall list of requirements 88
7.4 Mapping of requirements 90

8 Design 93

8.1 Modifying Zero Wine 93
8.2 Task automation 94
8.3 Generating IDS signatures 95
8.4 Other design choices 96
8.5 Choice of programming languages 96
8.6 System deployment overview 97
8.7 Design limitations 98

9 Implementation 101

9.1 Implementing vmcom lite 101
9.2 Implementing Zero+One 101
9.3 Code conventions 106
9.4 Testing 108
9.5 Software licenses 112
9.6 Integration with NAAS 112

10 Evaluation 115

10.1 Result goals achieved, system requirements fulfilled 115
10.2 Discussion 119
10.3 Further work 119

11 Thesis Conclusion 123

11.1 Contributions 123
11.2 The future 124

References 137

A User Manual **139**

A.1 Configuring and using Zero+One 139

A.2 Configuring and using vmcom lite 139

B Screen Captures **141**

C Implementation Appendix **145**

C.1 iptables script 145

C.2 vmcom lite source code 146

C.3 Zero+One source code 152

D Testing Appendix **163**

D.1 Suspicious API calls returned 163

D.2 Email correspondence 163

List of Figures

2.1 Increase in the amount of discovered malware during the time span from 1997 to 2008. 10

2.2 Amount of discovered vulnerabilities in software code during the time span from 1997 to 2008. 11

2.3 Attacker in control of three networks of infected hosts. 13

2.4 Control flow alteration in a sample to change its appearance. . . . 19

2.5 Oligomorphic, polymorphic and metamorphic malware mutations. 21

2.6 Attacker controlling an attack on a server through his or her network of infected hosts (botnet). 24

3.1 Graphical representation of the different steps in a malware analysis. 36

4.1 Analysing malware in a separate network isolated from remote communication. 43

4.2 Conditional `if`-clause giving two branches in the program flow. . . 52

6.1 Example of an IDS architecture. 69

6.2 Example of an IPS architecture. 70

6.3 Dependency tree for a unmodified version of `Zero Wine`. 80

6.4 UML Sequence diagram displaying program flow from unmodified `Zero Wine`. 82

7.1 Use case diagram for “Generate `Snort` signature”. 85

7.2 Use case diagram for “Gain information about sample”. 86

8.1 A possible system set up shown as a UML deployment diagram. . . 97

8.2 Sequence diagram showing program flow in the final system. 98

9.1 `NAAS` integration in a simplified UML class diagram. 114

B.1 Screen capture from `vmcom lite`’s help screen. 142

B.2 Screen capture from `Zero+One`’s help screen. 142

B.3 Screen capture from `Wireshark` capturing network traffic. 143

B.4 Screen capture from `PEiD`’s results. 143

List of Tables

1.1 An overview of terms and acronyms frequently used in the thesis. . 4

2.1 Summary of code obfuscation techniques and mutation methods. . 23
2.2 Brief summary of the different sections in the chapter. 33

4.1 An overview over the analysis tools described. 50

5.1 Host names contacted by the sample during the scenario. 60

6.1 Summary of discussed approaches. 72
6.2 List of network based IDS products. 78

7.1 Use case for "Generate Snort signature". 85
7.2 Use case for "Gain information about sample". 87
7.3 Requirements for the implementation part of the thesis. 90
7.4 Mapping of high level requirements to functional and non-functional requirements 91

9.1 API calls filtered out by Zero+One during execution of samples. . . 103
9.2 Summarised test results. 109
9.3 Testing internal system communication. 110
9.4 Testing parsing of API calls. 111
9.5 Testing the system's capability of finding packer technologies. . . . 111
9.6 Testing IDS signature generation. 113

10.1 Result goals, and descriptions of each requirement. 116
10.2 Reaching result goals, and fulfilment of requirements. 118

Listings

2.1 Assembly code showing a simple decryption loop that can be used to decrypt the actual malicious content of a sample. 20

4.1 Ruby code showing the problem of single path executions during dynamic analysis of malware. 51

4.2 C code returning a non-zero value if a virtualized environment is detected. 53

5.1 Aggregated results from an antivirus scan by ten different antivirus products. 57

5.2 String content of the sample used in the analysis scenario. 58

9.1 Example from the `Zero+One` code base. 106

9.2 Example from the `vmcom lite` code base. 107

C.1 Script containing `iptables` chains for filtering out traffic coming from the VM OS. 145

C.2 Complete source code for the `vmcom lite` program. 146

C.3 `Zero+One`'s configuration settings. 152

C.4 `Zero+One`'s library methods. 153

C.5 Entry point for `Zero+One`. 159

C.6 Shell script used by `Zero+One` to execute malware. 160

Chapter 1

Introduction

The amount of malicious software being spread on the Internet increases steadily and rapidly [Krister et al., 2007], and forms a threat to the users of the Internet. To sustain secure computer environments, such software must be evaded. However, the software uses complex and intelligent techniques to replicate and spread, which makes it difficult to completely avoid malware at all times [Schultz, 2004; Szewczyk et al., 2008].

NorCERT[1] handles threats and attacks related to key networks and critical IT infrastructure in Norway. This master thesis covers, in cooperation with NorCERT, a study of the threats that comes with malicious software and discusses possible approaches to overcome the risks and countermeasure the threats. The thesis looks into the possibility of improving the effectiveness of malware analyses by automating one or more of the time consuming analytical processes. Based on knowledge and background material from the document, analytical processes that could benefit from job automation are presented. A prioritisation is made concerning what is most advantageous for a malware analyst, combined with what is most suitable for this thesis and feasible with the time available. A proof of concept software implementation able to automate the highest prioritised process is implemented. The implemented system is released and published with a freely to use open source license, making it possible to use and contribute to for anyone interested.

1.1 Terminology and acronyms

Specific terms used in this thesis are subject to ambiguous interpretation and their meaning are thus explained in this section. Table 1.1a on page 4 can be used as a summary of the section.

The *malware* term is used throughout this document to signify a large group of software containing a number of more specialised groups of software including, but not limited to, *trojan horses*, *logic bombs*, *worms*, *spyware* and *viruses*. Each of these groups have their own definition, but share an important property—they are all software written to compromise the integrity, confidentiality or availability[2] of a victim's data. The literature about malware is wide and sometimes ambiguous,

[1] General information about NorCERT can be found at `http://www.nsm.stat.no`.

[2] The three terms "integrity", "confidentiality" and "availability" together form the basis of "software security". [McGraw, 2006].

making a proper definition of these terms difficult to specify. This thesis does not focus on differences in these groups, but whenever possible instead treat all subgroups of malicious software under the term “malware”, and use “malicious code” and “malicious software” interchangeably.

One of the methods malware uses to spread to other hosts is utilising *software vulnerabilities*, which are weaknesses in a computer system allowing attackers to violate the integrity of that system. An *attacker* is the person that performs the malicious actions. A malware producer is therefore not considered an attacker until he or she uses the malware for malicious gains.

A reported case or an observed event where there is reason to believe that malware and/or cyber crime are involved is called an *incident*.

The *sample* term is used in this thesis to describe a limited quantity (usually one file) that is characterised as a suspicious object, and subject for an *analysis*. During the analysis process, functionality and goals of the sample are meant to be deduced. “Analysis” and “malware analysis” are terms used interchangeably to describe the process. The term *malware analyst*, or the shortened form *analyst*, signifies the person performing the actual analysis. One of the results from an analysis is a score indicating how threatening the analysed sample is. The score is called the *threat level*, and a high value signifies an imminent threat from the sample, forcing the analysts to pay particular attention to the malware. Which values the threat level spans varies, but the span over three levels “High”, “Medium” and “Low” is an example.

The *signature* term is used throughout the thesis with sometimes slightly different meanings. A signature is, in general, a characteristic byte pattern or set of rules to filter out certain events, files or actions from a larger set [Rehman, 2003]. A signature is not used standalone, and requires a host program to have any real usage. Antivirus programs and intrusion detection/prevention systems are heavily based on signatures to localise and eliminate threats. The description of these appliances is deferred until Section 2.6.1 on page 28 and Section 6.2.3 on page 67, respectively.

The software implementation produced in this thesis is referred to as *the final system*, *the implemented system*, *the realised system*, and also *the system* where it is clear that it is a reference to the software implementation produced during this master’s thesis. The terms *implementation* and *realisation* are used interchangeably through the thesis. “`Zero+One`” and “`vmcom lite`” are named subparts, or submodules, of the final system.

Newly introduced non-trivial terms are highlighted in *emphasised text* the first time they are written, with a description. The detail level of each description depends upon the importance of the term concerning the thesis. Command line text and application names are written in a `typewriter font` to make it clear that it is in fact a reference to a program, a command line function or similar. Using such a typography ensure distinctions between software (“`VMware Server`”), and the company behind the software (“VMware”).

Frequently repeated non-trivial acronyms used in this thesis are shown with corresponding meanings in Table 1.1b on page 4. Their meaning are, in general, written in full length. However, their abbreviated versions are used in sections where the words are often repeated. The first time the acronym is presented, the meaning is defined together with the acronym—in addition to in the table. For acronyms found in one single section in the thesis, their meaning are presented only at the relevant sections, and not in the table.

1.2 Result goals

A few concise goals are stated in this section. The goals are each derived from the problem description, and they are labelled for a referring purpose. The goals are derived using the given problem description, after removing information that is unnecessary in regard to a concretised goal. The information that is removed is convenient for explaining and reasoning for the problem description, but not particularly usable when stating a result goal.

RG.01 Study the problems of malware, how the malicious software operates and how it propagates. Discuss available countermeasures to the problems, and reason for why these methods are not sufficient to overcome threats from malware.

RG.02 Describe the structure of a malware analysis. State of the art analytical methods and professional tools used in such an analysis shall be studied.

RG.03 Choose a solution that can be used to reduce the time spent during a manual dynamic analysis phase. The solution shall be possible to realise as an automated process of the dynamic malware analysis phase. The solution should be capable of handling incidents where the malware is involved (the solution should be "actionable").

RG.04 Implement a system able to automate the solution chosen in **RG.03**. The system shall not be fully functional, but merely a proof of concept implementation.

RG.05 Use the implemented solution and look into the possibility of integrating the solution with NorCERT's existing system for handling malware analyses. Describe what is needed to integrate the implemented system.

1.3 Methodology

This document consist of two main parts: (1) a gathering of background material regarding the malware problem, and (2) a realisation of a system able to automate particular tasks in a dynamic malware analysis. The first part is the largest, and is referred to as the preliminary studies of the document. The preliminary studies are based on research documented in articles, conference proceedings and information from web pages, forums and other online documents. All essential references of importance are stated where appropriate. In most cases, published articles and proceedings have been more up to date than books with the same topics, and are for that reason prioritised as references above books. Still, a few references to books are used, but they are used as bibliography only and have not necessarily been read throughout. Where web pages that are subject for modification are used as references, the date accessed is written close to the corresponding reference. Web pages of a static nature is not marked with a last date accessed.

The preliminary studies contain a survey of state-of-the-art tools and best practice work methods used by malware analysts at NorCERT. All tools and every

Term	Comment
Attacker	A person that is performing actions with malicious intentions.
Incident	A violation or imminent threat of violation of computer security policies, acceptable use policies or standard security practices.
Malicious code/software	Signifies a large group of software written to compromise the security of a victim's data.
Malware	Same as above
Malware analyst	Person performing the malware analysis.
Sample	A suspicious file subject for a malware analysis.
Signature	A characteristic byte pattern or rule set to filter out certain events, files or actions.
Software vulnerability	A weakness in software code that may lead to a possible security breach.
Final/realised system	The software implementation produced during this master's thesis.
Threat level	A relative score based on a sample's malicious capabilities.

(a) Terminology

Acronym	Meaning
API	Application programming interface
C&C	Command and control (server)
CERT	Computer emergency response team
(D)DoS	(Distributed) denial of service (attack)
RG.X	Result goal number X
IDS	Intrusion detection system
IPS	Intrusion prevention system
OS	Operating system
P2P	Peer-to-peer
PE	Portable executable
VM	Virtual machine

(b) Acronyms

Table 1.1: An overview of terms and acronyms frequently used in the thesis.

possible work method cannot be covered in this thesis due to time constraints and space limitations, but the *most frequently* used tools and work methods concerning malware analyses are discussed. An analysis scenario is conducted as part of the preliminary studies. The scenario is using a sample of the `Asprox` botnet, which got famous due to its rapid spread, aggressive behaviour [Bradbury, 2008] and frequent appearance in media from early 2008 to early 2009. An `Asprox` sample is chosen due to its representation of malware as complex software, and its ability to clarify the difficulty of performing malware analyses and conclude them with a correct threat level.

The given problem description is stated too wide to make it possible to start a realisation of a system, so it has to be narrowed prior to starting a requirements specification for a system. The problem description is interpreted and narrowed in the last parts of the preliminary studies, where focus areas are defined (Chapter 6). The interpretation of the problem description presents relevant approaches for analysis automation by describing them one by one. Doing so assists ensuring the best approach is selected when implementing a system. The preliminary studies are not affected by the interpretation of the problem description. Additionally, the studies support the selection of focus areas with background material. That is why the interpretation of the problem description is not placed earlier in the thesis.

It is important to note that actual requirements for the implemented system are *not* elicited until Chapter 7, meaning the system architecture and design are not chosen before the requirements specification is complete. However, the problem description interpretation in Chapter 6 chooses the specific *approach* to realise, merely a proof of concept implementation that can be used to generate vital *input* for intrusion detection systems (IDSes). The input is referred to as a *signature*, and is a pattern to filter out certain events [Rehman, 2003]. Using the generated signature, the IDSes are able to localise infected hosts in a potentially large network. When localised, network administrators can take appropriate action to solve the threat. The system is utilising an existing open source sandbox solution to execute malware. The signatures are generated by observing network traffic initiated by the malware. The existing sandbox solution is modified, so familiarising with the existing code base was needed to understand the program logic and flow. The released system is not a fully finished implementation and must undergo quality improvements and proper testing before it can be deployed in a real environment.

The reason for realising a proof of concept system instead of a complete and ready-to-use system is due to this master's thesis focus. The focus stays on state-of-the-art background material, a preliminary study of the malware problem, approaches to countermeasure the problem, and a survey of systems and applications that can be *useful* for a system that can reduce necessary human intervention in an analysis. The actual realisation of the system receives less attention, but still sufficient to document, release and test a working prototype. The thesis suggests relevant further work to the software implementation in Section 10.3 on page 119.

The development process of the system followed a traditional waterfall process [Braude, 2000], and is documented and structured in this thesis accordingly. Diagrams following the UML notation are shown during the development part of the document. The diagrams are shown to make the development process more clear and understandable for the reader, without the need of a vast amount of supplementary text. Utilising best practice standards, such as the UML notation for the diagrams, gives the reader that has software development experience a notation he or she quickly can familiarise with. A brief book about the basics of the UML notation is written by Fowler [2003], but is not necessarily needed as the diagrams shown are kept fairly simple.

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119 [Bradner, 1997]. The key words are found throughout the document, but is particularly important upon defining requirement and result goal levels.

To avoid unnecessary huge listings of code in the document, only example code is shown. Full source code can be found in the appendices for readers with special interest.

Reasoning for conclusions and choices made are given where appropriate, in particular where the problem description is narrowed and concretised.

1.4 Related work

Automating malware analyses are vital to keep up with the increasing prevalence of malware. The problem description is, at this stage in the document, not yet narrowed or concretised. A survey of related work and relevant research papers that suggest improvements to analytical processes are in consequent deferred to Section 6.1 on page 63. The section also assumes some background knowledge on the topic, found in its preceding chapters.

1.5 Document structure

In addition to this introduction, the thesis is structured in the ten following chapters.

Chapter 2 reasons for importance of the problems regarding malware, and discusses malware’s progress, its threats and available countermeasures.

Chapter 3 explains, at a general level, all phases of a malware analysis process.

Chapter 4 studies in detail the analytical phase that is most relevant for this thesis, merely the dynamic analysis phase.

Chapter 5 contains a practical study of a malware analysis, focusing on the dynamic analysis phase that is discussed in the previous chapter.

Chapter 6 narrows the problem description, concretises *what* to achieve with a software implementation, and selects focus areas for the implementation using background material from previous chapters. The preliminary studies of the thesis is then finished, and continues with the actual realisation of the system.

Chapter 7 contains the requirements specification for the system.

Chapter 8 sketches the design of the system based on requirements defined in the previous chapter.

Chapter 9 uses the sketched design and describes the realisation of the system as an implementation chapter. System tests are also found in the same chapter.

Chapter 10 evaluates the work process.

Chapter 11 finalises the thesis with a conclusion.

Chapter 2

Background and Rationale

This chapter contains information about how malware works, how it evolves, the threats from malware and available countermeasures to prevent the dangerous software from compromising systems. This way, the chapter gives a deep study of malware in general, and functions as a necessary knowledge foundation before continuing to the following chapters. At the same time, the chapter indicates that better, more robust and trustworthy solutions are needed in the fight against malware.

2.1 Malware propagation

With an increasing amount of online hosts having more and more computing resources available, taking control over machines can prove valuable for people having malicious intentions [Bradbury, 2006; Grizzard et al., 2007; Li et al., 2008a; Muttik, 2008]. The amount of discovered malware increases aggressively, and has continued with an exponential rate the last years as shown in Figure 2.1 on the following page. Malware spreads more than ever before, and the ongoing use of the Internet makes it easy to earn money and gather intelligence with help from malicious software. This section describes two main techniques malware use for propagation, and thus is able to spread to new hosts.

2.1.1 Utilising weaknesses in software

A goal for malware is often to infect as many hosts as possible, and in general, there are two main methods to achieve this. One of them is utilising weaknesses in software code [Heiser, 2004], and as the amount of flaws and bugs has increased almost every year [Krister et al., 2007], it tends to get easier to accomplish. Figure 2.2 on page 11 shows the amount of vulnerabilities reported annually since 1996, and indicates that software vulnerabilities are not likely to disappear all of a sudden. Several reasons explain the discovered vulnerabilities, but as software grows in size when functionality is added, its level of complexity rises consequently. Complex code is harder to maintain for a software developer and increases the probability of generating faults in the code. The available code to attack also increases as a result of larger software projects, which makes the situation more imminent since only *one* exploitable weakness is sufficient for a successful attack. The data in the

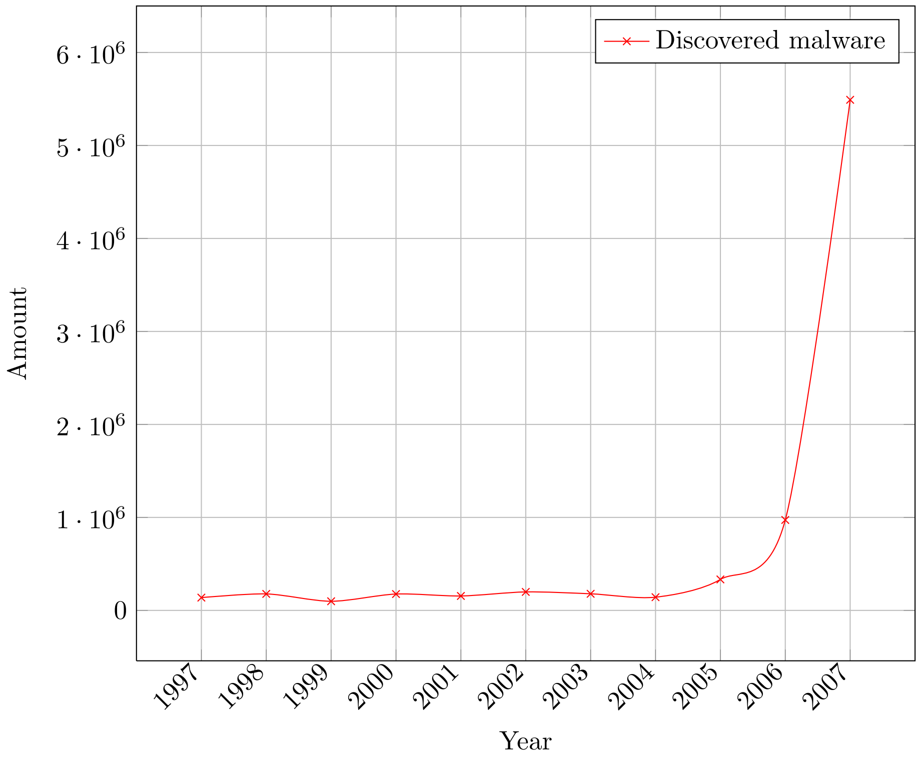


Figure 2.1: Increase in the amount of discovered malware during the time span from 1997 to 2007. The numbers are not cumulative, meaning the counter resets at the start of each year. Discovered variants derived from a unique malware type are all counted. The graph is based on data from `http://www.av-test.org`.

figure is collected from the Common Vulnerability Enumeration (CVE)[1], which is considered the de-facto standard for collecting and storing information about software vulnerabilities [Mann and Christey, 1999]. A first and quick glance of the graph indicates that the increase in amount of discovered flaws continues, but that is not necessarily the case; the two last years show the opposite trend—a *decrease* in reported software vulnerabilities. Software developers are now getting more focused on producing secure code due to the consequences of flawed systems [Wyk and McGraw, 2005], and new operating systems are designed to be more secure and can prevent common methods of exploiting vulnerabilities [Ahmad, 2007] such as trying to overflow a predefined buffer (buffer overflow) [Scambray, 2007]. Together, these reasons hopefully lead to less vulnerabilities in produced software, and to further decrease the amount of (discovered) software weaknesses.

2.1.2 Social engineering

Another method malware uses to spread is by misusing the fact that people are blissfully ignorant about the threats they are facing on the Internet. Users visit

[1] CVE's web site can be found at `http://cve.mitre.org`.

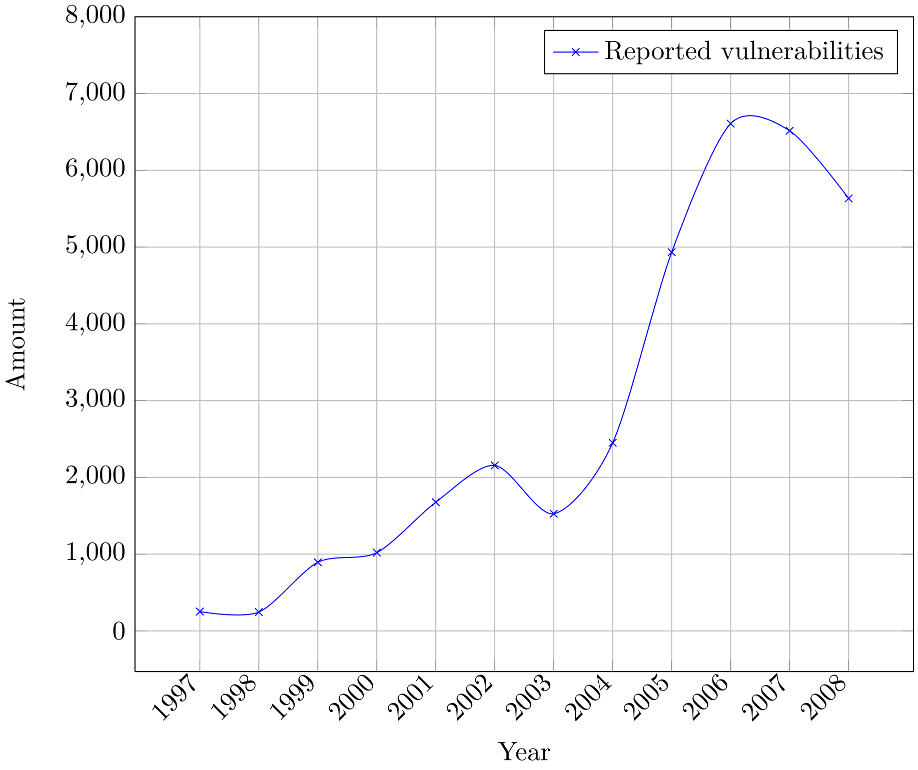


Figure 2.2: Amount of discovered vulnerabilities in software code during the time span from 1998 to 2008. Data is collected from the National Vulnerability Database (`http://www.web.nvd.nist.gov`).

malicious links that install malicious software on their computers - even when they are recommended by the system not to proceed [Dhamija et al., 2006]. An example is malware spreading using instant messaging (IM). An increase of such events have been reported over the last years [Leavitt, 2005] and has proven as an effective way of replication. By utilising the interfaces exposed by the IM clients, malware is able to send messages to all contacts an infected user has on his or her list. To be able to infect more machines, the message usually recommends to click on a link that downloads malicious software [Hindocha and Chien, 2003]. Users often uncritically visit these links, effectively infecting their own machine. Fooling a user into performing an action that may lead to infection is called *social engineering*.

2.1.3 Using a combination of the methods

Regularly, a combination of the two methods is used to infect a host as the IM example shows. The first step is to ensnare a user to visit a link that install malicious programs using social engineering. The installed software then tries to exploit one or more weaknesses in software that is present on the machine under attack (the victim). The vulnerable program(s) are arbitrary programs having one or several open (exploitable) vulnerabilities. If the attack is successful, it is possible that the attacker takes total control over the machine, and the computer is rendered insecure.

2.2 Techniques for controlling malware

With some exceptions, most malware is not instructed to infect particular hosts and is satisfied as long the infection is successful. A large list of hosts can be infected during a short time spam depending on how difficult it is to remove the malware, and how effectively the malware spreads. For certain types of malware, the infected hosts are operated remotely as controlled “robots” (bots). The network of infected hosts operated by an attacker is called a “botnet” and is a platform for distributed malicious computing [Savage, 2005]. The combined amount of CPU and bandwidth from the set of infected hosts can be incredible powerful, and is often used in a profitable business for the attacker in control [Li et al., 2008a].

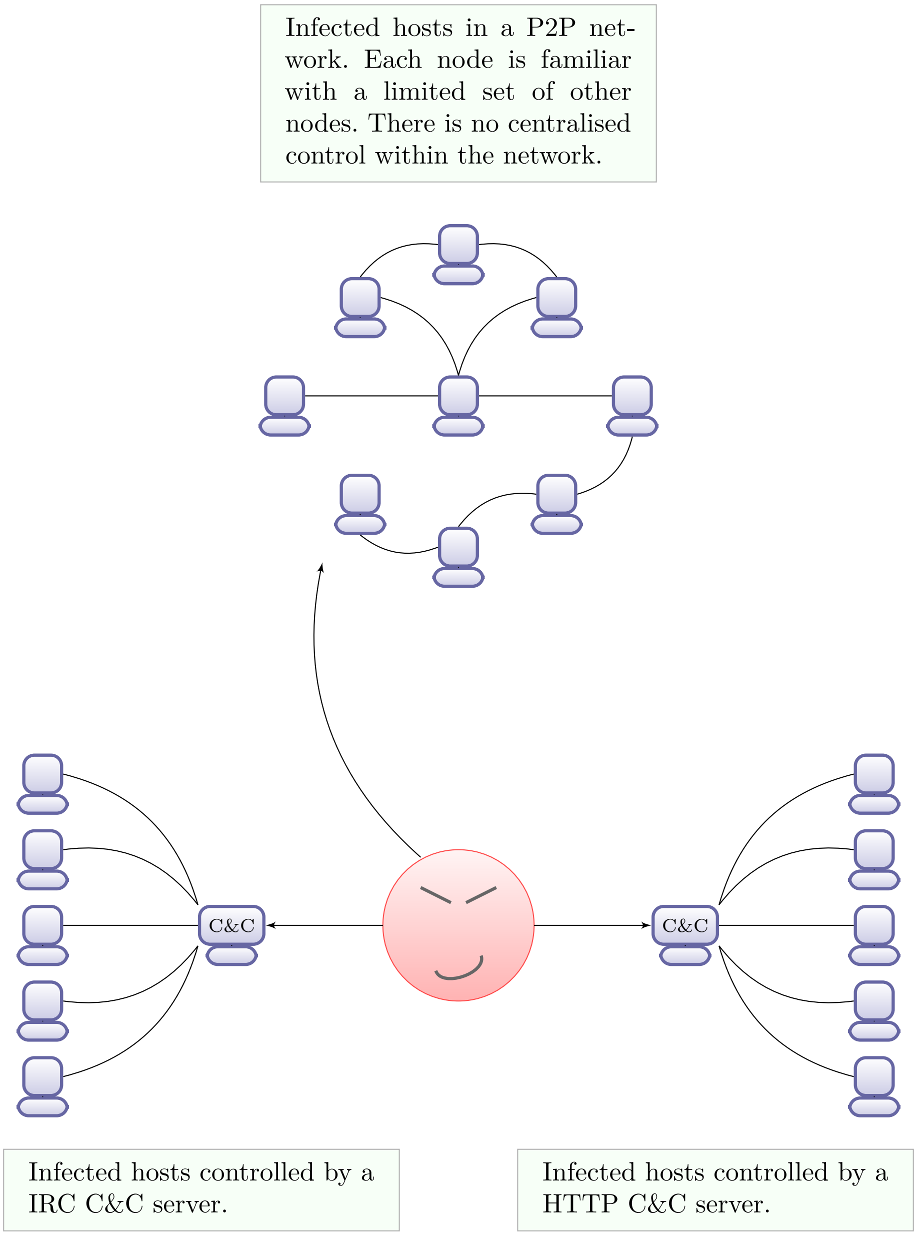
The attacker can prioritise hosts to infect, and powerful hosts with high uptime are the most valuable machines. Consequently, hosts with low downtimes, large network bandwidths and powerful CPUs are higher priority than dial-up hosts online only 30 minutes each day—as they are not worth much for the attacker. A botnet can grow and shrink in size as its population continuously changes when infected hosts are repaired, or new hosts are getting infected [Dagon et al., 2005]. This section presents two approaches used to determine how the malware operates: (1) a remote control technique, having the possibility to dynamically change the malware’s behaviour, and (2) a predetermined and fixed set of tasks.

2.2.1 Remotely controlled malware

Different methods are available to control the infected hosts, and the three most common [Ianelli and Hackworth, 2007] are presented below. The two first methods uses one or more centralised servers (called “command & control” (C&C) hosts), while the latter two are not. A graphical representation of three controlled networks is shown in Figure 2.3 on the facing page.

IRC/chat C&C

The online chat phenomena has existed for a long time, and even if the Internet relay chat (IRC) protocol was introduced early [Oikarinen and Reed, 1993], the technology is still in use. IRC server software is publicly and freely available, and can be used by anyone having a certain technical skill level. The servers allow a large number of concurrent connected users and require only limited hardware resources, which is one of the reasons for its success in the malware circles. The communication in the networks can be plain text, but also encrypted [Stinson and Mitchell, 2007]. Thus, the data a client sends to the IRC server can be encrypted and practically impossible to decrypt without knowing the decryption key. Additionally, the communicating clients can use encrypted sessions on top of this encryption layer using securely transmitted keys over a non-trusted channel with Diffie-Hellman key exchange [Krister, 2007]. The data traffic in the resulting communication channel is impossible to interpret for others than the communicating party. Malware producers utilise the IRC technology extensively [Gryaznov, 2005], where infected hosts automatically join chat rooms and listen for instructions



Infected hosts in a P2P network. Each node is familiar with a limited set of other nodes. There is no centralised control within the network.

Infected hosts controlled by a IRC C&C server.

Infected hosts controlled by a HTTP C&C server.

Figure 2.3: Attacker (red face) in control of three networks of infected hosts. Two of them are operated by a centralised C&C server, while one is a P2P network without any form of centralised control within the network. Arrowed lines show the attacker's communication path, while solid lines show routing paths for each network.

(commands). Using a symmetric encryption algorithm for communication requires that the malware has access to the encryption key somewhere [Grawrock, 2005], and even if data traffic is encrypted, it is possible to analyse communication and pick up commands if the encryption key is found. When the encryption key is stored in the malware binary, a (skilled) analyst are always eventually able to deduce the key given sufficient time and resources [Dube et al., 2008].

An attacker controls his or her network by issuing commands to the bots. A command can be as simple as `attack 1.2.3.4`, with the meaning "all hosts attack the IP address `1.2.3.4`". What an "attack" is varies, and common attack methods are discussed in Section 2.5 on page 22. If a defender identifies a C&C server, the communication between the attacker and the infected hosts can be disrupted, and the botnet is no longer effective. To somewhat cope for this limitation, attackers use DNS host names with small time to live (TTL) values and change the pointer at regular intervals. If a server is taken down, the host name pointer can easily be changed, and another pointer can be introduced instantly. By using a small TTL value, DNS cache servers will soon update their records and be aware of the new address.

Using services to register sub-domains, an attacker can easily change the domain pointer. Such services are called *dynamic DNS*, and are perfectly legal and a useful technology, but are subject to misuse [Heron, 2007b]. Since dynamic DNS services may be anonymous, tracing back to the attacker (the DNS account owner), might be difficult. In addition to disrupting the communication, the collection of infected hosts is revealed once the C&C server is detected. To make tracing the attacker even more complicated, the C&C servers are themselves compromised hosts or hacked servers. To prevent authorities taking down the servers, they are placed in a country without laws that cover cyber crime activities, making prosecutions juridically difficult [Chaikin, 2006].

HTTP/web C&C

A similar approach to the IRC C&C is the HTTP based C&C, where infected hosts look for tasks on a web server [Polychronakis et al., 2008]. The interface used by the attacker can be more easily be custom made using HTML and JavaScript, and give a more pleasant overview over the controlled network(s) than the IRC servers usually do. The two C&C approaches share most of their properties, but use different technologies for communication. When analysing network traffic without any specific unique data or patterns to look for, detecting traffic to and from a HTTP based C&C server is more difficult than on IRC based servers. A significant amount of legitimate HTTP traffic is already present in most networks. So to catch HTTP based C&C traffic, a method to filter out the legitimate traffic is needed unless the analyst know exactly what to look for. HTTP based traffic can more easily venture undetected through network observations. Additionally, it is easier to block IRC traffic in a network, since the consequences of blocking HTTP traffic are usually much higher than IRC traffic that fewer people use [Myers, 2006].

Peer-to-peer networks (P2P)

The centralised architecture in a C&C server provides efficient communication to all infected hosts but it is also a central point of failure. If the C&C server goes offline, the control is temporarily lost until the C&C is replaced [Grizzard et al., 2007]. In a P2P architecture, there is no centralised control over infected hosts, and botnet commands are instead retransmitted through the network so all peers receive the command [Dittrich and Dietrich, 2008]. Each compromised host know of a limited number of other hosts, contrasting IRC C&C where everyone knows of all the others. All nodes are allowed to send commands in a P2P network, but to avoid anyone from controlling the botnet, commands are digitally signed using public-key cryptography [Heron, 2007a]. When signing commands with the attacker's private (secret) key, each hosts can determine if the command comes from the attacker by validating the signature using the corresponding public key. If the validation succeeds, the command is considered legitimate and carried out by the bot. If the validation on the other hand does not succeed, a non-authorised command is detected—possibly sent by an intruder. Depending on how the bots are programmed, they can respond to this event as an intrusion and act accordingly; automatically initiate attacks on the source IP address is one possibility to avoid further unauthorised requests.

Email-controlled networks

P2P and IRC C&C's expose the list of some, or all, compromised hosts with varied detail level, and can be used to localise infected hosts. Using more dispersed networks can make the detection of infected hosts almost impossible. New research proposes a solution where the bots do not know of any of the other bots [Singh et al., 2008]. The malware registers an email account (using such as a free and anonymous online provider) automatically from the infected host, and reports the email address to the attacker. The attacker sends emails to each of the addresses containing botnet commands. The malware periodically polls the email account and checks for new emails. The botnet commands are camouflaged using brute force breakable encryption and steganography[2]. When the malware downloads the emails, it tries to break the encryption and decode the steganography, which requires a significant part of the CPU. Consequently, the approach makes it infeasible for a large email provider to do the same on all incoming emails to find attack emails.

2.2.2 Malware without control mechanisms

Not all malware uses a control mechanism, but controlling the infected hosts is a valuable property for the attacker since he or she can make the malware perform any kind of operation, making the malware highly dynamic. Malware not controlled in this way certainly exists, but is less dynamic. Some malware samples are programmed for static goals, such as log credit card numbers and report to a server and do not need any modifications. In these cases, it is favourable for the attacker to avoid using a control mechanism, as such can assist exposing the attacker or the malware infection itself.

2.3 Custom made malware

Malware can be made custom for a set of users to perform a more concrete set of tasks. The tasks can be anything the attacker wants, and hosts are selected

[2] Steganography is techniques to hide information to avoid detection of hidden messages [Katzenbeisser and Petitcolas, 2000].

by properties including, but not limited to, their location, a connection link to interesting networks, or that they are owned by persons of interest by the attacker. Such malware may or may not be controlled, but the amount of infected hosts might be significantly lower compared to the malware using control mechanisms. One infection might be sufficient. One of the main reasons for creating custom made malware is for information gathering [Bethencourt et al., 2008]. Attackers use malware as tools in espionage to collect documents, files or other secret data located on the infected computer or on one of its connected networks [Magruder and Lewis Jr, 2006]. This also affects encrypted networks, as the computer authenticates and handles the encryption routines as normal, and the malware works as a regular application, located on a higher level in the OSI model [Zimmermann, 1980] than the encryption algorithms. Malware can also capture keystrokes and pick up login credentials sent to applications, servers and networks.

Computers with specific foreign locations are also high value targets for central point of administration (C&C) or used as special routing detours (proxies) to make the life harder for those trying to track down the source of the malware. Proxies are discussed in Section 2.5.5 on page 28.

2.4 The development of malware

Malware has been around since the dawn of personal computers [Harrington, 2005, chapter 8], but the malicious intentions was, compared to now, relatively harmless in malware's early years. The goals were simple during malware's first ten years; the malware deleted files locally out of pure spite, wasted CPU cycles with pointless calculations and infinite loops, and occasionally displayed dialog boxes announcing its infection to tease the user behind the keyboard. In other words, the main goal from the malware was to make itself known to the user [Greiner, 2006]. Malware has since then become much more intelligent, and as opposed to its early years, malware is a key element in a lucrative business where CPU capacity, stolen credit card information and enormous amounts of stolen bandwidth are sold to people with malicious intentions [Bradbury, 2006]. It is now valuable for the malware producer to hide the software instead of exposing itself to the user, and the malware goals have changed from frustrate a user to gain financial profit [Li et al., 2008a]. The evolution of malware follows the steadily increasing amount of Internet nodes [Goudey, 2004], and so does the level of sophisticating techniques malware uses to multiply it self and camouflage its functionality [Katzenbeisser et al., 2005].

Avoiding detection is important not only to prevent the victims of infection from repairing their machine, but also from automated *antivirus applications*[3] that are usually more observant than the user and struggle to deny malicious actions. To prevent a third party from learning a program's functionality, its source code can be kept private for no one to see. However, by resorting to *reverse engineering* (reversing) techniques on a compiled binary file, its source code can in fact be partially or completely deduced [Eilam, 2005]. Code obfuscation is a common approach used to avoid successful reversing attempts by complicating the program's source code and binary content. Multitude of techniques are available, and new ones occur from time to time. The obfuscation techniques are usually

[3] Section 2.6.1 on page 28 discusses the usage of antivirus applications.

static, meaning when someone figures out how to detect or countermeasure the technique, it is no longer of any value [Dube et al., 2008]. Malware has proven to use nonstandard techniques and clever, never before seen methods to avoid exposing its real content and prolonging a successful attempt of deducing its (evil) functionality. Reasons for obfuscating malware includes, but are not limited to, the following [Skulason, 1990].

Preventing code analysis Suspicious code instructions are assets to look for during malware analysis. Obfuscating code disguises the actual malware content and hides the instructions from the analysts, which effectively prevents deducing the sample's real functionality.

Prolong the dissection process When the functionality of a malware sample is deduced, it is usually easier to deploy countermeasures, decrease its threat level and eventually eliminate it completely. Obfuscating code complicates and dwell out malware analyses, and thus increases the life span of the malware. Additionally, properly implemented malware can spread on a large scale in only a few hours, so each passed minute without descent available countermeasures is in great favour for the malware producers [Beaucamps, 2007].

Evade detection Changing code makes it appear different from the original version, and generates distinct malware samples. Each different sample is called a *variance* of the original form, and changing appearance makes detection more difficult for applications that use signatures to match files, such as antivirus applications.

Malware writers have taken code obfuscation even one step further, and resorted to *self mutating* programs that can more efficiently evade detection and complicate analyses. The malware changes itself dynamically either during running time, while being dormant in a computer or upon infecting another host. Doing so leads to a malware variant which appears different on the machines it infects. The variances of the malware more easily evade the antivirus applications, as each variance is unrecognisable offspring from the original malware. Malware using such techniques are called *oligomorphic*, *polymorphic* [Nachenberg, 1997] or *metamorphic* [Ször and Ferrie, 2001] malware, each method with an increasing level of complexity. The problem of self mutating malware has been known a long time, but is still truly difficult to countermeasure. How the malware changes is interesting, and the mutating *methods* used rarely differ. Still, it may be very complicated to detect the source of the new variance. Techniques used by malware to create a mutation of itself work mostly on low-level code, and operate on instructions found in the binary file [Walenstein et al., 2007a].

The most common techniques for code obfuscation are described below [Borello and Mé, 2008; Collberg et al., 1997], immediately followed by a description of the three different mutation techniques.

2.4.1 Junk insertion

Obfuscation by *junk insertion* is applied by adding code sequences to a sample without changing its behaviour [Christodorescu et al., 2007]. Using such modifications effectively evade *signature based* detection techniques, as a a new signature must be created to recognise each of the modified malware variants. Antivirus signatures are rule sequences made to recognise particular files, and are frequently used by antivirus products to detect malware. The added code sequences do not change the program's behaviour, and are *empty operations.* An empty operation (or "nop" (no-operation) has no effect to the behaviour of the program, but changes the checksum of the program. If the unnecessary operations are removed prior to matching them with a signature, only one signature is needed for all of the malware's variants. Research shows that it is possible to remove unnecessary instructions on assembly level [Christodorescu et al., 2005], and end up with a sample stripped from all junk insertions. If this method is applied on all files, only one signature is needed to catch all variances of a malware type.

2.4.2 Control flow alteration

Instructions in an executable file can be placed syntactically different, and by using unconditional jumps, the functionality remains indifferent even when the instructions depend upon each other. An example based on a report by Christodorescu et al. [2005] is shown in Figure 2.4 on the facing page. Figure 2.4a shows the original set of instructions, located sequentially in the compiled sample. The instructions are then reordered internally in the file, and "jump" instructions are inserted to point to the next correct instruction. Figure 2.4b shows the modified instruction set.

2.4.3 Code permutation

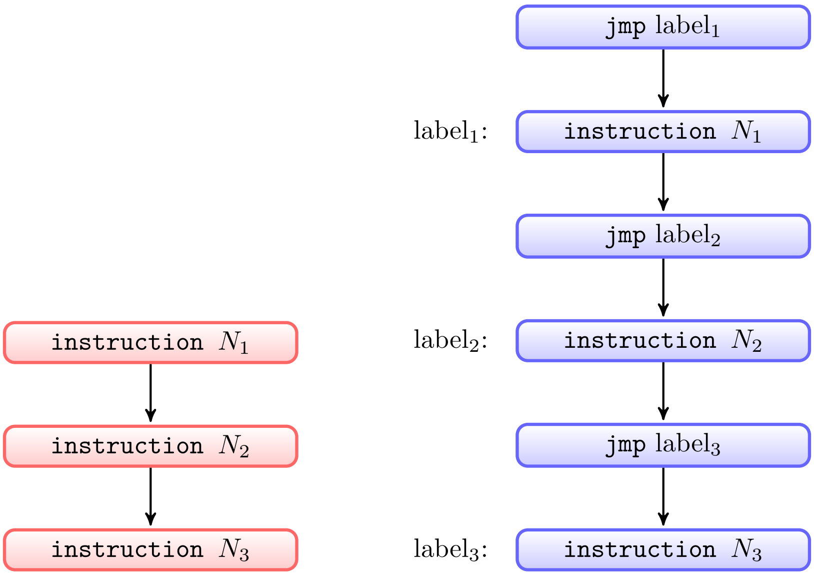
Instructions independent of each other can be arbitrarily placed in the program without altering the functionality. The program keeps its functionality, but appears different for each change. An example partly taken from the literature [Bruschi et al., 2006] goes as follows (the && operator signifies a logical `AND`-operation).

$$\begin{aligned} a &= b \times c \\ d &= b + e \\ f &= b \&\& c \end{aligned}$$

The above statements can be executed whenever wanted due to no mutual dependencies. As long as the statements are all executed, variable a, d and f always bound to the same values whether the statements are executed in the same order. However, if the above statements are the only known part of a larger program, their sequence can only be changed internally, since it is possible that the variable values are used elsewhere in the program and other, unknown, statements depend on them.

2.4.4 Utilise executable packers

An *executable packer* is a program that can transform a binary into a smaller version and thus changing its appearance. The packed binary is in most cases in an unpacked form in memory, but is stored compressed on the persistent storage (disk) [Yan



(a) Original instruction set

(b) Reordered instruction set leading to an alteration in the control flow. The instructions can be placed at arbitrary places in the binary file and "jump" (`jmp`) instructions are inserted to point to the place of the next instruction.

Figure 2.4: Control flow alteration in a sample to change its appearance [Christodorescu et al., 2005].

et al., 2008]. Over 80% of malware use packing techniques, and often apply different techniques recursively to complicate analyses further [Guo et al., 2008]. More than 200 packer families are known and together sum up over 2000 packer variances. If a packed binary is unpacked, the body is easier to detect. However, understanding the different packers requires a significant amount of resources and time. As new packer variances easily can be introduced by an attacker, learning how all packers work is not practically feasible.

It is possible for antivirus applications to emulate a malware execution and wait for the unpacking to happen, but it is hard to ensure this event, as its execution can depend upon the environment, and its running time can be arbitrary long [Zuo et al., 2005].

Martignoni et al. [2007] proposed a design that would detect packed binaries and then execute an antivirus scan the moment the binary has unpacked itself in memory. The design does not guarantee negative side effects, and is its imminent drawback. Guo et al. [2008] published a similar design that does not leave any undesirable side effects, but packer technologies are highly dynamic and evolves prior to the countermeasures, and can evade the unpacking. The designs can assist the detection process, but do not guarantee a correct result.

```
mov esi , ADR_OF_SOURCE        ; move the start address (pointer)
                               ; of the encrypted data into esi
mov edi , ADR_OF_DESTINATION   ; move the start address (pointer)
                               ; of what will become the
                               ; decrypted data
mov bx, SIZE_OF_DATA           ; move the size of the encrypted
                               ; contents into bx

loop:
    lodsd                      ; fetch a byte of the address
                               ; pointed to by esi into al
    xor    al , 0A5H           ; decrypt this byte using a
                               ; simple XOR operation with
                               ; the key "0A5H"
    stosb                      ; store the result into the byte
                               ; pointed to by edi
    dec    bx                  ; are we finished (decrement
                               ; the size by one)
    jnz    again               ; repeat if the size variable (bx)
                               ; is nonzero. upon completion, bx
                               ; has decremented to zero

...                            ; code continues after decryption
                               ; is complete
```

Listing 2.1: Assembly code showing a simple decryption loop that can be used to decrypt the actual malicious content of a sample. The code is based on an example by Skulason [1990].

2.4.5 Oligomorphic mutation

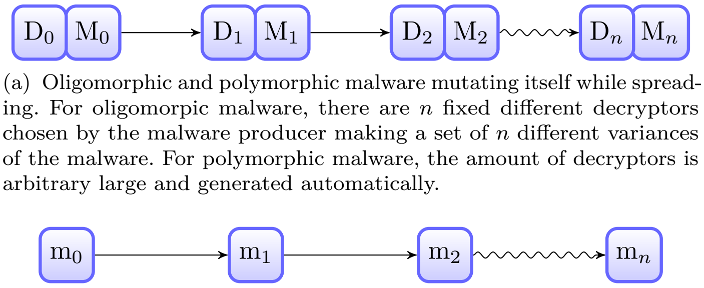
One of the first and simplest methods used to obfuscate malware was to encrypt its content [Konstantinou, 2008]. The malware must be exposed in its original form upon execution, and a decryption key is used to unlock the malware's original content. The algorithm used for decryption is called the *decryptor*, and is in the simplest case static in each copy of the sample. How to find the decryption key and the decryptor are stored in the sample code, and can be found using reverse engineering techniques. When the key and algorithm are found, the sample can be manually decrypted in an analysis.

If the sample is capable of generating a multiple decryptors n, where $n > 1$ but still a *limited* set, it is capable of an *oligomorphic* mutation [Szor, 2005, page 258-260]. The malware is able to generate different variances of itself upon mutation, and n different signatures must be made to detect it using signature based detection techniques.

A simple decryption algorithm (decryptor) is shown as assembly code in Listing 2.1, where the first column contains an assembly operation. What the different operations do are briefly described below.

lodsd (load string doubleword) loads a string into the processor register.

xor is the mathematical `XOR` operation that can be used as a simple encryption mechanism.



(a) Oligomorphic and polymorphic malware mutating itself while spreading. For oligomorpic malware, there are n fixed different decryptors chosen by the malware producer making a set of n different variances of the malware. For polymorphic malware, the amount of decryptors is arbitrary large and generated automatically.

(b) Metamorphic malware mutating itself while spreading using a variety of code transformation techniques. The sample m_0 appears completely different from its mutated variance m_1, but the functionality remains untouched.

Figure 2.5: Oligomorphic, polymorphic and metamorphic malware mutations.

stosb (store string byte) copies the altered data from the processor register.

dec decrement variable by 1

jnz jump if condition is met.

The oligomorphic process is shown in Figure 2.5a where the malware is able to generate a limited amount of decryptors, n. The malware multiplies itself, and to be detected by signature based techniques, its body must either be decrypted, or n different signatures must be made.

2.4.6 Polymorphic mutation

Polymorphic malware can dynamically change the available decryptors to an extensive number, making an large amount of actually used decryption methods [Nachenberg, 1997]. The polymorphic mutation makes it practically impossible to create signatures for all variances. To detect polymorphic malware, the detectors must be able to scan samples while they are in memory, decrypted and in the original constant form [Konstantinou, 2008]. Figure 2.5a applies for polymorphic mutations as well as oligomorphic ones, but a polymorphic mutation is able to use a significantly larger set of nondeterministic decryptors than oligomorphic mutations are.

2.4.7 Metamorphic mutation

Instead of changing decryptors and encryption methods, metamorphic malware mutates its own body, usually upon propagation [Ször and Ferrie, 2001]. In theory, except for the behaviour, variances have not necessarily anything in common with each other, and can be virtually undetectable when using signature based detection [Chouchane and Lakhotia, 2006]. Metamorphic malware changes its appearance without the use of an encryption routine, but at the same time keeps the original functionality. To achieve this, combinations of the different transformations discussed in this section are used. Metamorphic malware can be extensively complex and incredible difficult to detect. In some cases, a metamorphic malware sample can decompile itself, change its body in source code form and recompile to generate a completely new variance of itself [Ször and Ferrie, 2001].

Lakhotia and Mohammed [2004] suggested a *normalisation* of the malware so fewer signatures are needed to detect all variances. Malware normalisation undoes obfuscations applied on a sample, and results in a normalised executable. The approach is able to reduce the number of variances, but depending on the quality of the normalising techniques used, the decrease in variances is not sufficient to be of any practical use. For example, the authors reported 10^{183} variances were reduced to 10^{20} normal forms, which are still too many forms to cover using signatures.

Christodorescu et al. [2005] suggested a similar approach giving better results. The approach normalises effects from the obfuscation techniques code reordering, packing and junk insertion. A significant drawback is long running times for the normalisation processes.

Figure 2.5b on the preceding page shows how metamorphic malware changes during spreading. The malware is not using any encryption, so no decryptor is available. Instead, the malware transforms dynamically and independently of the initial malware sample. Even its core changes during the transformation, and the mutated sample m_1 looks completely different from its predecessor m_0.

2.4.8 Summary of code obfuscation and mutation methods

Table 2.1 on the next page shows a brief summary of the different code obfuscation techniques and mutation methods described in this section. Each applied method and technique effectively evades signature based detection techniques by producing a variance of the malware. Using a combination of the techniques and methods complicates the detection process even further, and makes it practically impossible to use purely signature based detection techniques to detect all variances.

2.5 Threats from malware

In addition to compromising the security in a system, numerous threats comes along with a malware infection. Without a sample analysis, the complete set of possible effects from the malware is seldom known, or at least not guaranteed to be known. In consequence, malware can operate unexpectedly and sudden; as malware is capable of doing anything a computer can do, only the creativity limits the possibilities. This section focuses on the most dangerous threats from malware, which are the threats that indicate a struggle for economic gains [Grizzard et al., 2007].

2.5.1 Denial of Service (DoS)

DoS attacks are operations to prevent a service from operate normally, or overload it in such a level that legitimate users cannot use it. The DoS attacks can be local, denying usage of the computer or software on it, but attacking remote servers are more common. A DoS attack can be any attack, but the common goal for the attacks is to prevent legitimate users from using one or more servers and/or services. An example is the “SYN attack” that floods a victim server with half-open connections. The attack is explained in a handful of papers, but Lau et al. [2000] have published a good and simple example. The attack goes as follows. When a client initiates a

Technique	Description
Junk insertion	Adding empty (nop) code sequences to a file. The added code has no meaning to the behaviour.
Control flow alteration	Introducing additional jumps in the program.
Code permutation	Reordering the location of instructions independent of each other.
Utilise executable packers	Compressing a file into a smaller version. The file content are mostly unreadable without running or manually unpacking it, but it is stored as its original form in memory.
Encryption	Obfuscating file content by using an encryption key. The simplest encryption routine uses the `XOR` operator on file content with a key stored together with the file.

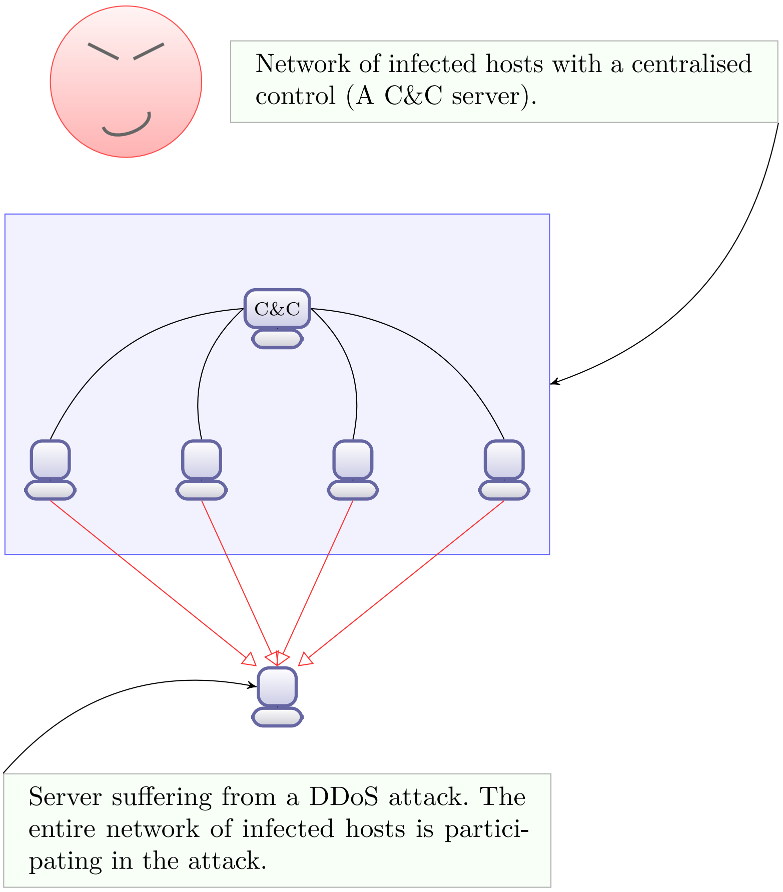
(a) Code obfuscation techniques

Method	Description
Oligomorphic mutation	Same as encryption, but the decryption routine (decryptor) differs.
Polymorphic mutation	Same as oligomorphic, but uses a significantly larger set of decryptors.
Metamorphic mutation	Utilises a multiple of the different techniques explained in this section to obfuscate the code.

(b) Mutation methods

Table 2.1: Summary of code obfuscation techniques and mutation methods.

TCP connection to a server, a “three way handshake” is required. Handshakes apply to all TCP connections, and are used to ensure a *reliable* connection (TCP manages an active connection even if packets in a data stream are lost or duplicated in the network). When the client never transmits the last ACK packet required for the handshake to succeed, the memory used to hold the connection remains half-open and is not freed until a timer expires [Schuba et al., 1997]. If tens of thousands infected hosts request a connection to the server at the same time, the memory and available ports will be exhausted, and the service is soon overloaded. Such flooding attack is one of different methods to overload a service, and distributing the attack over a multiple of sources is called a distributed denial of service (DDoS) attack. As a botnet can vastly outnumber a victim in available bandwidth and computing resources, the victim must try to seize the attack, but a DDoS attack is very hard to prevent due to its distributed nature and its capability of exploiting weaknesses in best practice solutions such as TCP. There are in general no bullet proof defence against a large DDoS attack, but safety measures to avoid the attacks in the first place should be used, and precautions should be taken to limit the consequences of an attack [Mirkovic and Reiher, 2004]. Due to the difficulties of preventing a DDoS attack, the attacker is subject to severe consequences if the case is brought to court [Curran, 2006; Hilley, 2006]. The attack is, however, difficult to trace when



Network of infected hosts with a centralised control (A C&C server).

Server suffering from a DDoS attack. The entire network of infected hosts is participating in the attack.

Figure 2.6: An attacker (red face) controlling an attack on a server through his or her network of infected hosts (botnet). The botnet can be arbitrary large, and combined vastly outnumber the server in resources. The victim will have a hard time preventing the attacker be technical means, and must solve the problem using other approaches.

compromised hosts are used to control it, and consequently places the attacker in a role only indirectly involved in the attack [Lau et al., 2000].

To gain profit using a DDoS attack, the attacker can run a "sample" attack, proving what he or she is capable of, and then demands money to stop further attacks, turning the DDoS attack into an extortion [Bridges, 2008]. The attacks are usually targeted against a few networks, but there are reports of incidents shutting down the IT infrastructure in a whole country—for political reasons [Goth, 2007; Lesk, 2007]. Figure 2.6 displays a DDoS attack over a network graphically.

2.5.2 Spam

Spam is a term that describes unwanted advertisement, often sent in very large quantities. Spam is distributed using a variety of different technologies, but the email service is currently the most popular to use [Gomes et al., 2004]. A way to stop email spam is deny incoming emails sent from hosts known to send spam (blacklisting), and is an approach widely used [Ramachandran et al., 2007]. Additionally, filters based on guessing whether the content is genuine or not are applied to stop distributed spam runs. Millions of spam emails are sent every day, and if they all were sent from a strict set of hosts, it would have been relatively easy to block the hosts using a blacklist-.even if the hosts changed from time to time. Using a large set of compromised hosts that each send their share of spam emails makes managing a blacklist difficult. Moreover, impossible if the amount of hosts is sufficiently large. Spam emails often contain malicious attachments [Barroso, 2007], which could be the same malware that infected the source of the sent email. Due to the enormous amounts of emails sent, the probability that some of the users execute the malware and get infected, is high.

2.5.3 Phishing

Phishing is a process that masquerades a trustworthy entity, and trying to acquire sensitive information such as login credentials or credit card numbers. A common scenario is to construct a replica of a web page, copying the design and content of the real page to fool a user to believe the copied site is the legitimate one. To avoid tracing to the attacker, such web sites are hosted on compromised systems, and any collected information is bounced to the attacker. If the server is taken down, the attacker can simply start a server on another compromised host. Phishing is a technique relying on manipulating people into performing an action (social engineering) to succeed [Bilogorskiy, 2005]. Bank, web auction and social networking sites are prime targets for phishing, and attackers use a variety of different techniques to avoid revealing that their fraudulent site is in fact a fake one [Fogg et al., 2001]. Common methods used for phishing are described below.

Utilising URL semantics

The URL "`www.myBank.com`" will for most persons be acknowledged as safe and trustworthy, especially if "myBank" is the name of their bank. Accessing the URL would render the page hosted at "`myBank.com`", as expected. However, by using sub domains, an attacker can construct URLs similar to "`www.myBank.attacker.com`". When accessing the constructed URL, the host "`attacker.com`" is contacted instead of `myBank.com`. Of course, the legitimate "myBank" has nothing to do with `attacker.com`. "`Attacker.com`" is not the best disguised domain name, but not all persons would notice the irregularity in the URL. A similar method is using different representations of the URL; IP addresses in different notations can disguise the threat using representations the user is not familiar with [Lance, 2006]. If the IP address for "`attacker.com`" is `15.16.17.18`, the host can be represented as shown in Equation (2.1).

$$(256^3 \times 15) + (256^2 \times 16) + (256^1 \times 17) + (256^0 \times 18) = 252711186 \qquad (2.1)$$

Therefore, a different way of typing the URL pointing to the attacker's site is "`http://15.16.17.18`" and "`http://252711186`". Yet another method is using the "at" sign (@) as a denominator between the legitimate site and the fake. Accessing the URL "`http://myBank.com@252711186`" use "myBank.com" as user name to the attacker's site. This method is used legitimately for web sites using HTTP authentication. Most browsers now warn the user when such tricks are detected, but a warning is not necessarily sufficient to stop people from entering the site [Dhamija et al., 2006; Florencio and Herley, 2005]. If the attacker's site has implemented a HTTP authentication mechanism (the web server asks each users to supply login credentials), the warning a web browser displays is similar to the following[4]: "You are about to log in to the site 252711186 with the user name myBank.com". The warning will, in many cases, sound legitimate for users lacking the necessary knowledge to reveal the threat.

Visual deception

Attackers mimic text, images and windows to gain the trust from potential victims. The letter "w" in a URL can be switched with two "v"s [5] , and the"l" letter with an "i". Observing this irregularity is not trivial, even for an experienced user. Assuming an attacker registers a domain `www.trustvvorthybank.com`, points it to his web site which has a complete replica of the real "Trustworthy Bank" web site. An unobservant user with a bank account at the real bank could mistakenly believe the URL is perfectly legitimate, and supply his or her login credentials to the attacker.

Additionally, if multilingual character sets are available, an attacker can use characters in different language sets that resemble each other. Some of the Cyrillic and Latin letters have very similar appearance, which is possible to exploit during a phishing attack. Such an attack is called a *internationalised domain name (IDN) homograph attack* [Gabrilovich and Gontmakher, 2002], and when internationalised universal resource locators (URLs) will be more common on the web, this attack point can be exploited if it is not properly taken care of.

Avoid the pitfalls of amateurism

A company is commonly having well designed web sites. Phishing sites (or emails) with (obvious) typographical errors, broken links or other small glitches are easier revealed fraudulent than the ones with professionally produced content. Therefore, the most successful phishing attempts use exact site/email copies, and every detail is as close as possible to the source itself.

Tailor the user experience

A site is more credible when some elements are custom made for that specific user. Examples of such personal elements are last date accessed, a personal welcome message with the name of the user or specific ads directed to the user. Getting such information is not trivial, but a successful attack makes it easier to gain the users trust.

[4] The exact wording varies in the different browsers.

[5] The attack was recently performed on Twitter's web page, `http://twitter.com`. An attacker registered the domain name `tvvitter.com` (with double "v"s), and used the domain as a phishing site. See `http://www.sophos.com/blogs/gc/g/2009/05/21/beware-tvvitercom-video-live-twitter-phishing-attack/` for more information (accessed 2009-05-22).

Combination of methods

Attackers commonly use a combination of the above methods to succeed, and properly designed phishing sites can fool as many as 90% of the users [Dhamija et al., 2006]. An inexpensive method to fool users to visit web sites is sending emails. The email service can by used to send fake content to a large number of recipients. Email is using the old SMTP protocol that was never designed for security [Piessens and De Win, 2002] and is easy to forge. SMTP is unauthenticated, and sending emails from a fake source address is trivial. The "FROM" field in an email can be manipulated similar to traditional mail. Phishing emails can in that way be sent from an address that appears legitimate, and can contain content identical to what is expected from the corresponding legitimate source. Information can easily be sent to a large number of users, where some of these users will most certainly access the fraudulent site with. Using malware to distribute such spam makes it possible for an attacker to send large quantity of emails having links to the phishing sites. The quality of the emails sent varies significantly, from obvious grammatical errors to sophisticated elaborate documents indistinguishable from official correspondence [Goudey, 2004].

2.5.4 Click fraud

One of the most popular ways to advertise currently on the Internet charges the advertiser by number of "clicks" on the ads. The ads are usually images, flash animations or occasionally text with special crafted HTTP hyperlinks. The ads are located at any web site, and a scenario of concern is *click fraud*. The problem was published with detailed attack scenarios as early as 1999 [Anupam et al., 1999] and is because that clicking on ads are nothing else than clicking on a link on a web site. A browser sends the same HTTP headers when clicking on an ad as any other link, so the click can easily be programmed. If a large amount of compromised hosts is instructed to simulate a click on a specific ad, the action can cost an advertiser a significant amount of money and is difficult to prevent since the sources are distinct and unrelated IP addresses. An attacker benefits from this in the following two cases.

1. The attacker instructs his or her compromised bots to simulate a click on one of his competitors ads, thus generating an expensive and wasteful bill for the competitor.

2. An attacker having an arbitrary web site rents out space on his or her site for advertisement material that is selected by an advertiser. Each click on the rented space costs the advertiser X\$ that he or she must pay the owner of the web page. Since the owner is also the attacker in control, the click traffic can be generated from a set of compromised bots.

Studies show that rates between 12% and 16% of advertisement clicks on the Internet are done by automated scripts, bots and low wage workers earning their living simply by clicking on ads [Jansen, 2006]. A large error percentage is calculated into these numbers, as it is hard to distinguish between a legitimate click and a programmed one since they technically do not differ. So statistical learning mechanisms regarding a legitimate user's navigational behaviour *before* and *after* the click happens is carried out [Immorlica et al., 2005]. For example, thousand instant clicks from a page that normally has 100 visits per day are likely a network of bots clicking on the advertisement and not potential customers. It is difficult to guarantee that such behaviour is from a human or not, so the statistical methods suffer from high error rates and are in the worst cases pure guessing. As soon as the attackers behind the click fraud implements these navigational techniques into their programs that simulate clicks, the statistical methods seize to work, and have to be continually changed to be of any use; an endless race between the attackers and advertisers happens with this approach. Another method suffering from the same problems is client side observation of mouse movement and keyboard events that are normally expected from a legitimate user. However, such movements are possible to simulate in a program as well [KyoungSoo et al., 2006], or easily manipulated on the client side.

Statistics regarding the amount of advertisement clicks that later lead to a sale, can be utilised to filter out automatic clicks [Brooks, 2006]. This is relevant particularly for web shop advertisements, where it is feasible to conclude that if none of the registered clicks lead to a purchase, the click sources are not human. Of course, the method is also subject to high error rates as the expected click/buy ratio depends on too many factors to be correctly calculated in all cases.

A different approach is authenticating clients before the clicks happen, and count the clicks only when performed by authenticated clients [Juels et al., 2007]. The authentication is transparent for the user and utilises generated cryptographic tokens supplied by a third party that validates the clients. Such an approach is feasible, but significantly increases the complexity level of the advertisement design.

2.5.5 Proxies

A *proxy server*, or a *bouncer*, is a daemon receiving incoming connections and forward them to the actual receiver. A proxy server at an infected host can be used to launder connections with malicious payloads through that particular machine and hide the attacker's source [Levy, 2003]. The actual receiver sees the infected host's IP address instead of the attacker's. Recursive usage of proxies is also possible, making tracing back to the original source even more complicated. Proxies are therefore used for malicious gains to prevent the attacker being caught by IP logs.

2.6 In need for more effective countermeasures

Malware is without doubt dangerous, and to guarantee a secure system, malicious software cannot be allowed to enter its environment. This section covers the most common countermeasures to stop malware, and to prevent it from performing malicious actions.

2.6.1 Antivirus

For an end user point of view, the most common countermeasure against the threats from malware is an installed antivirus application actively monitoring the system. The antivirus software tries to withstand malware and can deny the execution of malware if detected. However, antivirus software does not always succeed. There are several issues to be aware of when relying on an antivirus application, notably the fact that the software is *flawed by design* and can rarely, or never, guarantee a correct result [Krister, 2008]. The antivirus programs are based on a mix of techniques such as signature based detection and "heuristic" scanning techniques (explained below) [Kay, 2005; Sanok, 2005]. To be able to detect threats using signature based detection, the antivirus software must be instructed by a byte-pattern signature to localise each threat [Bailey et al., 2007]. One signature must therefore be created for each threat to recognise. Newly discovered threats are in consequence not likely to be detected until the antivirus vendor creates and publishes updates with new signatures. Malware can thus easily evade the signature based detection by morphing a slight variance of itself. Malware mutations are explained in Section 2.4 on page 16, but a new signature is needed for each unique variance, when signatures based detection is used [Christodorescu and Jha, 2004]. If the malware changes rapidly enough, publishing new signatures for each variance of the malware are eventually infeasible. The antivirus industry figured out these issues with signature based approaches, and implemented the *heuristic* detection type that is able to recognise threats based on common suspicious behaviour. The problem with the heuristic scanning is scan results are *never* guaranteed to be correct, and can only be used as an indication of the actual reality [Krister, 2008]. There are chances for both false positives and false negatives, making the actual security undetermined and brittle. Malware analysts know this, and fight a hard match against the dangerous software.

2.6.2 Stay updated

As malware often targets specific versions of applications or operating systems, it is essential for users to stay alert and apply security update patches [Cole et al., 2007]. Many operating systems automatically "push" updates out to the user to avoid forcing the user to manually patch the system, which often implies large delays [Byrne, 2006, section 5]. New security patches repair vulnerabilities, flaws and bugs, but may introduce new vulnerabilities as with any other software—possible even more critical than the repaired vulnerabilities. The longer the applications are known to be vulnerable, the more people will know about the particular flaw and it will be easier to take advantage of it as applications exploiting the weakness are often easily available [Barroso, 2007; Maynor and Mookhey, 2007]. Having a non-patched application to avoid introduce possible new flaws is not the optimal way of running a system, and it is in general better to patch systems even due to the fact that patching can introduce new software vulnerabilities.

2.6.3 Extensive analyses

For malware producers, it is usually not important if their software works on all kinds of computers and systems. The malicious software usually targets a specific application, and simply halts if the application is not found. The malware may even clean up itself to remove any traces. A malware analyst on the other hand must ensure samples run in their expected environment to avoid false conclusions about threat levels. How to ensure the correct environment, which kind of vulnerable software must be installed is difficult, and hours of analysing might be needed to make a survey of such requirements. Due to the large amount of malware, prioritising samples correctly is a central element for the analysts. However, when prioritising a large number of malware samples, the chance of missing important samples is high.

Thus, solutions that are more efficient required to fight the continuously increasing amount of malware. Automating tasks is one of the steps towards a faster analysis, and is the overall main focus in this thesis. Repetitive tasks are especially relevant for automation, but all tasks that have some sort of fixed structure are possible candidates for automation.

2.6.4 Firewalls

Another common proactive defence is an appliance that inspects network traffic and denies certain packets based on rules set by the user. These devices are called *firewalls*, and can prevent malicious software from entering a machine or network. If a host is already infected, a firewall can prevent malware from "signalling home"[6]. Firewall types span from application level software to hardware based appliances. Firewalls can be based on traffic flow attributes such as source of origin and destination address; and also *states* in the communication protocols [Bellovin and Cheswick, 1994]. A firewall can as an example allow all traffic from the address X but deny the rest—unless the traffic was initiated from *within* the network (having state "ESTABLISHED" or "RELATED"). Firewalls can certainly be used to block malware, but the required set of rules to stop all kind of malware is too large to manage a complete and updated set. Many state-based firewalls allows all established connections from the machine or network, but since it is often the user behind the keyboard that initiate actions leading to infections, a firewall suddenly becomes useless. If used correctly, firewalls can prevent attacks or infections, but the way they are normally used does not hinder the user to click on neither a malicious link nor visiting a web site filled with malware.

2.6.5 User awareness

Malware may require human interaction for a successful infection, such as visiting a link or execute a harmful program. Otherwise, the malware at least requires one or more present vulnerabilities to exploit so it can gain control of the system. Using resources on user awareness gives users a chance to stop malware before it enters the system by understanding the threat instead of relying on error prone applications. Users lack, in general, knowledge about the threats from malware, its common ways of infecting a system and simple steps to countermeasure the dangerous software. Training users in these areas prevent infections and lower malware's propagation abilities.

2.7 Computer emergency response teams (CERTs)

A CERT is a group of professionals dedicated to handle computer security issues. A CERT observes networks and handles any threats to their covered IT structure. Most countries, or at least the ones dependent on their IT-infrastructure, have one or more CERTs that cover network traffic in important networks. Some CERTs cover a specific (sub)network only, while others cover a whole country [West-Brown et al., 2003]. The CERTs in the latter group do not necessarily observe all network

[6] Malware tends to communicate with fixed servers (their "home") with the purpose of receiving an updated set of instructions, commands or software.

traffic in a country, but merely traffic to and from important assets. Uninett CERT[7] that covers the networks for universities in Norway is a network CERT, while NorCERT is the Norwegian national CERT observing particular vital networks in Norway.

As computer security is a wide and complex area, it is important for the CERT to determine which services they offer, and to whom. The following services are examples of services provided by CERT's.

Announcements The CERT analyses new security threats and publishes warnings based on their investigation.

Vulnerability handling The CERT follows updates on software vulnerabilities, their available exploits and how to withstand any attacks using them.

Training The CERT educates people by sharing knowledge and giving practical examples.

Risk analysis The CERT analyses software and/or hardware in regard to its security level. That is, how properly it can withstand attacks. Looking for vulnerabilities in the source code and studying design flaws are key elements here.

Watching technology The CERT observes the network and looks for new attack patterns and scenarios. The information is published to other CERT teams and cooperative groups.

Software development The CERT produces new (security) software or improves existing solutions.

Exactly where to limit the coverage depends upon a variety of different practical factors, but it is important the team can understand and familiarise with their assets, determine attack trends on their networks, and stay updated on new threats and how to withstand them [Smith, 1994].

2.7.1 Malware analysis in a CERT

A CERT usually has a group of malware analysts available, maybe even dedicated to work on deducing functionality from malicious code. The observant reader may ask why duplicate the work probably already done by antivirus companies or other persons publishing malware information on the Internet, but that is not the exact case. As discussed in Section 2.4 on page 16, the sophistication level of malware has increased tremendously since the first types of malicious software. The malware analysed in a CERT is sometimes unique, and targeted for a particular person, company or network for a very specific purpose. The reasons include gaining access to specific networks originally meant hidden for the rest of the world, or stealing intelligence and classified documents. Therefore, the information published by antivirus vendors and other individuals can be very limited for some of the samples analysed in a CERT, and often nothing at all.

[7] See `http://cert.uninett.no` for more information about Uninett.

2.7.2 Network monitoring tools

Malware analysed in a CERT is often picked up by a *sensor* monitoring their networks, or other suspicious events targeted directly to persons located on the inside of any of the networks. The sensors are nodes connected to selected entry points in the network actively monitoring all traffic passing by, looking for suspicious events. The amount of concurrent network traffic each CERT can analyse depends on the amount of sensors available and their physical and logical deployment in the network. A national CERT have a multitude of sensors connected to possible large networks, while a smaller one can suffice with only one sensor deployed on the outer rim in the network, covering all hosts inside. Some sensors can be instructed with patterns (signatures) to filter out certain events, which can be used to estimate the size of infection from a newly discovered sample. The sensors are usually a part of a "network intrusion detection system" solution, which is covered in detail in Section 6.2.3 on page 67.

2.7.3 Cooperation

There are more malware producers than malware analysts, and the producers can suddenly get one step ahead of analysts by creating new never-before-seen malware. To be able to cope with this, and quickly deduce the functionality to find the threat level from new malware, cooperation between the different CERT teams, antivirus vendors and other individuals is essential. CERT analysts work towards the same goal, and their fight against malware is usually not driven by profit. For this reason, the teams communicate and seek assistance from each other to improve their proactive and reactive response to security incidents. One of the key cooperation groups are the Forum of Incident Response and Security Teams (FIRST)[8] group having over 200 teams, including NorCERT, as members. In addition to bringing CERT teams together, FIRST organises conferences, provides education and the teams help each other on incidents, both local small ones and large ones affecting the whole world.

2.7.4 Confidentiality

A security incident is not always a public matter, and this is particularly so for malware targeting determined systems. Incidents involving systems that people assume to be safe and trustworthy are also not necessarily information that should leak. If information about infected networks or web sites is published, the reputation of the company in harm can be significantly weakened. People may react as avoiding the companies and their web pages for the fear of infection and doubting their competence, strongly decreasing the revenue for the company.

A CERT utilises a multitude of different systems and appliances to run analyses efficiently, where some of them are third party commercial applications only available online. To use these services, a necessity is to upload samples to the third party and let their systems handle the analysis. Depending on the overall field of responsibility for a CERT, the CERT can be under special juridical terms and must act accordingly. In Norway, NorCERT is subject to specific laws, and using third party online services to analyse samples or material is far from optimal and is illegal by law when the

[8] See FIRST's web page `http://www.first.org` for more information.

material, or the incident connected to the material, is a classified matter. Analysing samples in-house and having local versions of tools and applications are in some cases therefore the only option for a CERT like NorCERT. *Sandboxes*[9] used in malware analyses are often located online, where some sell the services as a local version for an (expensive) cost. Such economic resources are not available in this thesis, prioritising freely licensed software.

2.8 Summary

Due to the chapter's length, this section lists a summary of the chapter's sections, shown in Table 2.2.

Section	Section summary
Propagation	The amount of discovered malware increases aggressively, and malware is able to propagate by utilising a combination of weaknesses in software code and *social engineering* techniques - fooling people into installing the malware on their systems.
Control techniques	Malware can be controlled to do what an attacker wants, using communication methods such as HTTP, IRC, P2P and email. Malware can also be static, with a predetermined set of tasks to perform.
Custom made	Most malware is made to infect as many hosts as possible, but sometimes the hosts to infect are carefully selected and the malware is custom made for the targets.
Development	Malware is now complex pieces of software, as opposed to its initial years where it was mostly made to tease its victims. Malware uses sophisticated methods to avoid detection and be able to stay active as long as possible.
Threats	By controlling a sufficient amount of infected machines, an attacker can have an enormous amount of available processing and bandwidth capacity. This power can be used for malicious gains and it often is. DDoS, spam, phishing, click fraud and proxy usage are the most common attack scenarios.
Countermeasures	Countermeasure malware is mostly applied using error prone antivirus and firewall software. Users often rely blindly on these appliances, but they are not bullet proof.
CERTs	CERTs are teams dedicated to handle computer security issues. They understand the threats from malware, and works continuously to fight malware and research the risks from such software.

Table 2.2: Brief summary of the different sections in the chapter.

[9] A sandbox is software and/or hardware used to execute malware in a safer environment. The sandbox phenomena is studied more in detail in Section 4.1.5 on page 45.

Chapter 3

Phases of a Malware Analysis

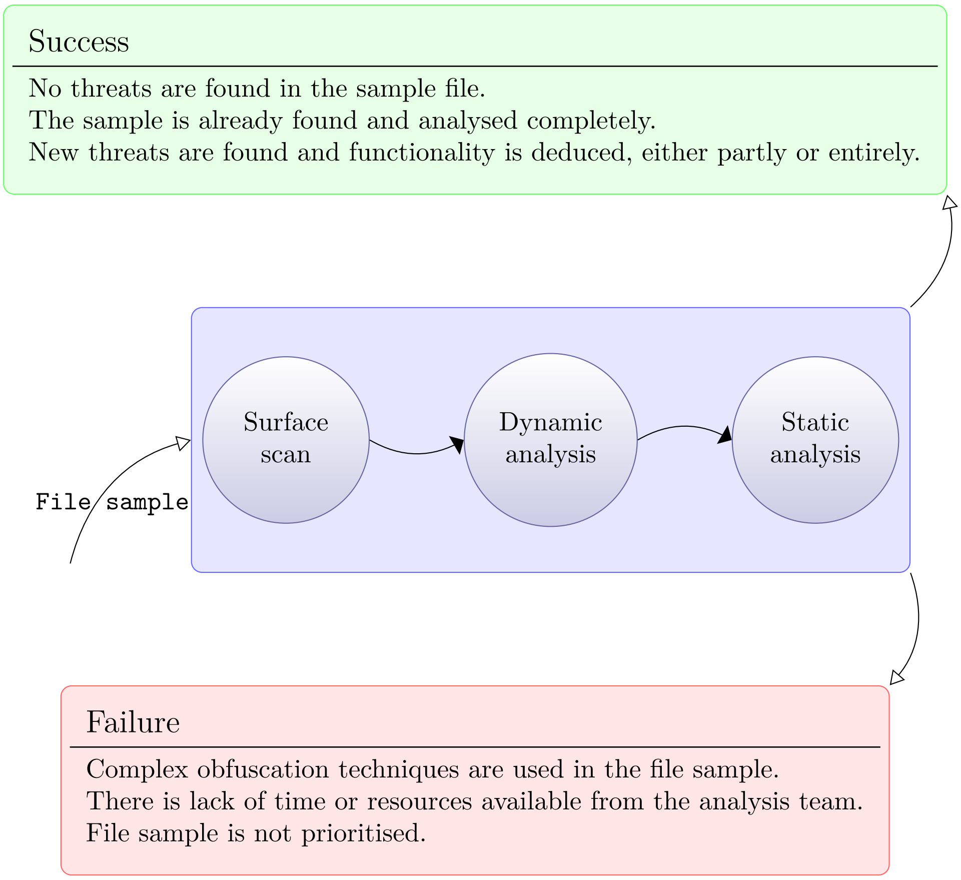
When source code and documentation for a program is unavailable, understanding its complete functions and behaviour is sometimes difficult. Analysts are often set in this position since malware source code is rarely known [Bruschi et al., 2006], and malware producers struggle to obfuscate and hide their code—as seen in the previous chapter. There are two main approaches to solve this problem, called respectively *dynamic* and *static* program analysis. Nevertheless, prior to any of these phases, an analyst usually runs a quick *surface scan* on the sample to parse valuable metadata and information about the file(s). This is done without running the file or going into its inner details. This chapter covers how these three groups of analysing techniques works, but firstly describes the overall structure of a sample file, which is necessary knowledge throughout the thesis. Figure 3.1 on the following page shows a graphical representation of the different phases during the analysis process.

3.1 Structure of a malware sample

Malware, like a legitimate application, is not entirely unique and have similar structure and content [Barr et al., 2008; Walenstein et al., 2007b]. In addition to the use of packers and obfuscation, which is discussed in the previous chapter, low-level operations and overall file structure are often similar for the different malware samples. This section covers how malware files are structured, and why and how usage of API calls is carried out by malware. The file structure can always differ and there is no rule without an exception, but this section covers the file structure commonly seen.

3.1.1 File structure

The form of a malware sample spans from textual macro content to complex binary files, but malware can be in any form - all depending on its targets, functionality and ability to evade detection. Most of the reported malware targets the Windows operating systems [Pegoraro, 2003], and are often in an *executable* binary form. An executable file is meant to “perform indicated tasks” according to Merriam Webster online dictionary, but an executable file is a file you can run standalone from other programs. The Windows operating systems usually mark such files with one of the



Success

No threats are found in the sample file.
The sample is already found and analysed completely.
New threats are found and functionality is deduced, either partly or entirely.

Failure

Complex obfuscation techniques are used in the file sample.
There is lack of time or resources available from the analysis team.
File sample is not prioritised.

Figure 3.1: Graphical representation of the different steps in a malware analysis.

file extensions `.exe` or `.bat`, while UNIX environments mark executables with a special executable parameter.

Executable files are not necessarily in binary form, but malware usually is to more easily hide its functionality. The files must be structured specifically so the operating system is able to recognise their binary form as executables and thus is able to run them. The preferred, or only way, of structuring executable binary files in the Windows operating systems are in the *portable executable (PE)* binary form that was introduced in 1993 [Kath, 1993] together with the `Windows NT 3.1` system. Studying the PE file format is out of the scope for this thesis, and being aware that it exits and used in most executable malware samples are sufficient. The PE structure is documented and discussed in detail by Pietrek, and the reader of special interest is encouraged to read his articles [Pietrek, 1994, 2002].

3.1.2 Usage of APIs

If the malicious intentions are overlooked, malware works in many ways similar to a normal application and performs the same operations a legitimate program does. For an application written for the Windows operating systems, *how* to perform the various operations may differ from the different revisions or release versions. To avoid needing rewriting of all applications with each operating system change or version upgrade, *application programming interfaces (APIs)* are made public for usage by developers, including developers of malware. Using the APIs avoid calling kernel code, which potentially differ in the various operating system revisions. So operations such as creating a file do not need to call kernel code, but the developer can instead call a high level API method that takes care of the details. Malware may or may not use the APIs, but uses them in most cases [Willems et al., 2007]. By *hooking* on an API, a programmer is able to observe the calls to the API before they are actually performed. The programmer can then later on forward the call to the actual receiver so the functionality in the system remains the same. This approach is called API hooking and is an important strategy used by malware [Yin et al., 2008], but also its countermeasures are utilising the method [Sanok, 2005]. Malware hooks on to the Windows APIs to filter each running application's view of whats going on in the system so the system users are not able to see it [Harley and Lee, 2006]. As a result, malicious software can benefit from hooking techniques by maintaining undetected and legitimate applications can hook on to selected APIs to observe calls to them. As an example, creating or modifying files is a common procedure from malware. Hooking on the respective APIs for creating and modifying files gives the possibility to pick up the calls before they are issued, analyse them and do whatever action seems reasonable. Hooking on the API for file modifications gives an analyst the possibility to observe all file system changes applied by the malware, and is one of the methods a *sandbox* environment is utilising. Sandboxes are studied in Section 4.1.5 on page 45, but is a safer way of running malware samples.

3.2 Surface scanning

The surface scanning phase consist mainly of a set of automated tasks performed on a suspect sample file [Wedum, 2008]. Hash sums on the suspect file are often generated and compared to a list of known text strings to ensure the sample is not already known, or maybe even already analysed.

A multitude of commercial antivirus applications are used to scan the suspect file for known malware [Krister, 2008]. The analyst can base a threat level for the sample using information and results gained in the surface scanning. The threat level is used to prioritise the sample concerning its importance, and eventually as background material for the next phases of analysis. Additionally, a few other small programs may also be used during the surface analysis, depending on the file type. For example, if the file is a document in a `Microsoft Office` file format, certain algorithms can be used to scan the file for suspicious content that is not necessarily detected by antivirus software[1].

If the sample is unknown, not already analysed or considered safe by the antivirus applications[2], the analysis process continues. If the sample is already known and analysed, the analysis process is complete and exits successful. If the results from the antivirus scan indicate a harmful file, the antivirus vendors may

[1] The `MOICE` tool can be used to find suspicious content in `Microsoft Office` files. See `http://support.microsoft.com/kb/935865` for more information.

[2] Remember from Section 2.6 on page 28, antivirus results must be considered as possible false negatives.

already have analysed the sample in their laboratories. Their information databases can supply sufficient data about the malware functionality so the analysis process completes successfully. This of course assumes the antivirus scan results are not including false positives, and the vendor's reports are trustworthy. A deeper and more throughout analysis may be needed if the published information cannot be trusted, or is incomplete.

3.3 Dynamic malware analysis

The dynamic analysis phase studies the behaviour of a program while active and running in its normal, or simulated, environment. Applying the dynamic analysis methods on a malicious program require the analyst to closely observe, and possible trap, program activity including file system changes, network communication and suspicious collection of data from the system. The analytical environment may suffer from the consequences by running malware the same way as a normal computer would do. In consequence, it is likely the environment will be compromised if that is the sample's intention. For this reason, it is extremely important the analysis process is under tight control and changes to the environment are reversed as soon as possible to sustain a secure environment.

Malware often tries to communicate with host servers to download new malware or receive further instructions and tasks [Dittrich and Dietrich, 2007]. Therefore, an Internet connection might be unwise as what flows through the communication network is unknown and can cause further harm from the malware. On the other hand, to simulate a *real* infection, the analysis must be run as it would in an actual infected environment to properly observe all effects—including prior and after any suspect network communication. There is no single correct answer of how to perform a dynamic analysis, but if the sample is allowed to run wild, it is essential the analyst ensures he or she can control the execution properly. To manage control, the analyst needs as much information of the software as possible to perform the analysis securely. For well known and widespread samples, the amount of information from analytical sources are potentially high. For new samples and the ones with low spread, few or no details are available.

Most analyses stop at the dynamic analysis, successful or not, due to the difficulty of the next phase which is the static analysis. Nevertheless, some samples require special treatment and are given the resources needed for the last analysis step. Dynamic analysis is the focus area for this thesis, and is studied more in detail in Chapter 4. A practical example of a malware analysis can be found in Chapter 5.

3.4 Static malware analysis

While dynamic analysis has its similarities with proactive research, static analysis is more reactive where the goal is to deduce the functionality of the program by reversing the compilation process. This is, as mentioned in the previous chapter, commonly called "reverse engineering". Applying static analysis methods on a sample forces the analyst to work on byte code and is, compared to high level programs such as Java, Ruby and C++, extremely low-level and complex. However, by using the correct techniques and having sufficient time available, the analyst is eventually able to discover the entire program flow [Dube et al., 2008]. Producers of malicious code write programs that obfuscates itself using sophisticated methods, which does not make the problem any easier [Ször and Ferrie, 2001]. A discussion about code obfuscation can be found in Section 2.4 on page 16.

3.5 Finalising the analysis

Analysing malware is both time consuming and difficult. Compiled programs written by an attacker in a couple of hours may require days of analysing to reverse the process. A combination of both dynamic and static methods are therefore often required to understand the functionality of the program with the use of a *reasonable* amount of resources. As malicious software is getting more intelligent, the analyst is required to think like an attacker and familiarise with obfuscation patterns to conduct a successful analysis. Static analysis does not receive particular attention in this thesis, and the focus stays on the dynamic analysis phase.

The analysis ends as a *success* when the analysis goal is reached, and as a *failure* if the analyst stops the analysis before reaching the goal. An utopian goal is to deduce the complete functionality from a malware sample, but due to the expenses required to do so, the analyses usually succeed when only *partial* functionality is deduced; sufficient data to base a threat level on. If the sample is already analysed successfully, the analysis exits successfully already at the surface analysis phase upon identification of the sample.

A failure during analysis happens frequently if few resources are available, either in amount of analysts or available time and money. Additionally, new previously undiscovered samples may be prioritised above old ones, making analysts abort currently running analyses. A particular example of this is samples camouflaged and obfuscated in such a level that an analysis is not feasible with the available resources, leading to a failed analysis.

Chapter 4

Dynamic Analyses in Depth

The dynamic analysis phase usually initiates just after the surface scan ends. This is previously mentioned, and also reflected in Figure 3.1 on page 36. A dynamic analysis is the process of studying a running sample in its expected environment. The analyst observes every kind of event happening during its running time, and possibly after the execution is complete, if other processes were altered or started. The analyst then observes these processes the same way. The phase is contrasting the surface scan by currently consisting of mostly manual work, using a handful of different tools to assist the process. Most of the tools work at the same level as regular applications do, which is “on top of the operating system” away from hardware and low-level functionality. It is therefore important to note if malware has managed to breach down deep in the operating system kernel and successfully altered the normal flow expected in an application, the dynamic analysis can give erroneous and incorrect information about a sample’s threat level [Hoglund and Butler, 2005]. This malware type is called *rootkit*, and uses API hooking, which is discussed in Section 3.1.2 on page 36. Using a rootkit allows the malware to bypass operating system checks and gain higher privileges operations than other applications do, and therefore interfere with the dynamic analysis. It is possible to monitor kernel level functionality in a dynamic analysis using specialised kernel debugging tools, but is for the extreme cases due to the complexity of these methods. Kernel level debugging is not a topic in this thesis.

The chapter contains a description of commonly used methods to assist a dynamic analysis, and continues with a list of important events to look for during the analysis. The chapter flows by describing assisting solutions and tools available in the dynamic analysis phase, and finishes with weaknesses with the phase.

4.1 Analysis methods

Luckily, the analyst has some tricks up his sleeve, which make the fight against malicious software feasible. This section covers the most commonly used techniques and methods during the dynamic analysis phase. Which of the method(s) to use in the different cases depends on the sample’s threat level, the consequences if the security is broken and which resources that are available. In addition, analytical experience can determine which method(s) to use as some of the methods are more complex to use than others.

4.1.1 Physical deployment

Having total control over the analysis is essential to avoid damages to the analytical environment by the malicious software. If the machines used for analysis were to be infected and controlled by an external attacker, there is a chance for fatal consequences; the analysts reputation would be strongly crippled and valuable information accessible in the analytical environment can be stolen. A dedicated room (a "laboratory") can be used during the analysis, physically sealing both analysts and the dangerous software from the rest of the world.

4.1.2 Separate networks

Another more practical solution than a change in the physical deployment is to separate networks with a set of few observed entry and exit points. Doing so strictly controls the network traffic, avoiding attackers from gaining access and malware escape the exit point(s). Still, the physical architecture and environment remains unchanged.

As malware tends to use the Internet for receiving instructions and sending back information, the malware analysis may be required to use Internet access, and therefore is in need for dedicated controlled and monitored channels for Internet traffic. All response malware expects to receive must be simulated by the analyst to ensure the program flow reflects an actual infected system. A possible method is explained below as a scenario.

Practical example

A malware laboratory consists of two computers where the first of them is running a Windows based installation and the latter a Linux based operating system. The two machines can communicate with each other, but no other machines are connected to the network during analyses. The two machines are isolated from the outside world, but only the Linux machine is aware of this. The Linux based machine acts as a router for the small network, meaning all traffic within the network ventures through the router machine before entering the remote network—which is not available during analyses. During an analysis, all remote communication is dropped at the network boundary, and the traffic halts at the Linux based machine during the routing process. Figure 4.1 on the facing page displays the scenario graphically.

The analyst can observe a running sample's effects from both machines. The sample is executed on the Windows host, since the Windows operating systems are widely used and are prime targets for malware. The Windows host is therefore the *victim* host in this context. The victim host is subject to an observation regarding changes to the registry and file system, while network traffic should be observed at the Linux based router, as one cannot know if the network observation on the victim machine is tampered with by the malware. File system observation is also subject to tampering attempts, but file changes are difficult to observe from outside of the machine, and therefore still performed from the victim machine. In addition to observing network traffic, fake responses to remote communication requests can be given from the router to fool the malware to believe it is actually has access to a remote network. Using this approach, one can fool the malware to send communication requests.

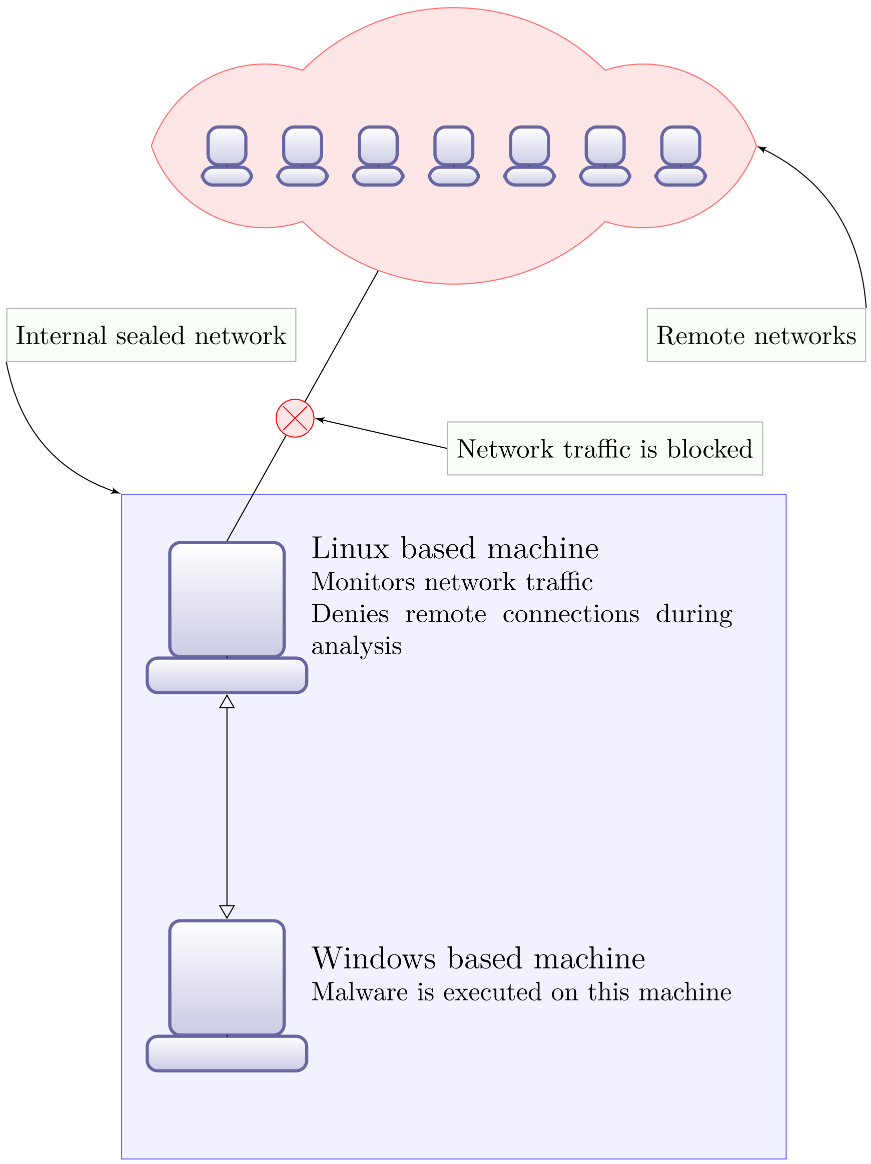

Internal sealed network
Remote networks
Network traffic is blocked
Linux based machine
Monitors network traffic
Denies remote connections during analysis
Windows based machine
Malware is executed on this machine

Figure 4.1: Analysing malware in a separate network isolated from remote communication. All traffic in the internal sealed network must flow through the Linux based machine before entering the remote network. The Linux based machine denies all such traffic, but is able to observe it all.

4.1.3 Virtual environments and software based snapshots

When using a *virtual environment* instead of a normal software environment, the analyst is handled unique properties. An operating system normally expects unique access to physical hardware, so without special modifications it is not possible to use more than one active operating system on a computer at the same time. *Platform virtualization* is a technique that escapes the limitation and allows multiple operating systems to run simultaneously on the same machine. A *virtual machine* consists of a operating system, and has a mapping between the real hardware elements and virtual hardware elements. The operating system having the virtual environment installation (and all installed virtual machines) is called the "*host*" operating system, and each virtual machine's operating system is called a "*guest*" operating system. Platform virtualization during a dynamic analysis is advantageous for the following reasons.

Control network access Dangerous and unknown malware may force an analyst to shut off all remote communication to sustain control over the malicious software. At the same time, malware might deviate from its normal program flow when it is unable to access particular remote host servers, forcing the analyst to enable the connection again. Instead of opening the remote network link, the machine hosting the virtual environment can *act* as the remote machine instead of the actual one. Exactly how is applied is discussed more in detail in the practical example shown in Section 4.1.2 on page 42.

Easy switching of operating systems Malware often targets a vulnerability in a particular program, or sometimes, in a particular operating system. Virtualization makes it possible to more easily change between different versions of operating systems and programs by keeping each one separated without any need of physical restructure of hard drives, which are often the case when using multiple OSes in the same machine.

Snapshots Several of the available platform virtualization solutions offer the possibility to store the current system state, called a "snapshot" of the system. If an analyst is afraid of having his system compromised during an analysis, he or she can simply store a clean and safe state, and later on reverse to the state using a simple procedure in the virtualization software.

The preliminary project for this thesis contains an in depth explanation of the platform virtualization phenomena, and the reader of special interest is encouraged to study the corresponding document for a more detailed discussion [Krister, 2008].

4.1.4 Hardware based snapshots

The virtual environments use software based methods for reverting a system state. This is however not the optimal case in all situations. Hardware based snapshots instead load the default (clean) state into memory at each reboot. Such an approach clears all changes applied in the machine after the snapshot was taken, thus cleaning possible infections (assuming there were no infections at the time the snapshot was taken). Reverting to software based snapshots is usually an on demand procedure initiated by the analyst, but hardware based snapshots make the reversion process *default* on each reboot and makes it unnecessary to manually perform the procedure. Additionally, hardware based snapshots gives the analyst the opportunity of working in a non-virtual environment. Certain malware is able to detect virtual environments, and can change its program flow if such an environment is found. Detecting virtual environments is discussed in Section 4.7 on page 49, but is a solid reason to avoid analytical environments in software based virtual environments.

4.1.5 Utilising sandboxes

Recall that keeping control is a key element during malware analysis. This is particularly so for the dynamic analysis, where malicious code is observed while it is running. A security technique called *sandboxing* is a way of more safely execute programs in a strictly controlled environment [Prevelakis and Spinellis, 2001]. The sandbox operates as a jailed environment around a system, usually with snapshot capabilities. The sandboxes might run periodical reversions to keep clean and safe system states. Sandboxes cover a large spectrum of operations and functionality, but concerning malware they are most commonly used to monitor system (API) calls, particular system events and network traffic. When the sandbox analysis is complete, an aggregated report file showing spotted events is created and returned to the analyst. At least the following three functions can be performed with help from a sandbox analysis [1].

Analysing threats By using sandbox report files, malware analysts can deduce a sample's functionality on a general level. The report files usually contain operations requested by the malware during its execution. Certain patterns are common to see from malicious software, and an experienced analyst may spot these suspicious patterns from a report file.

Construction tools for removal To quickly remove the effects from malware, the reports can be used to build simple programs, or even scripts, to revert the reversible malicious effects. Such tools are often built more quickly than an antivirus update is published. The tools are usually directed to one particular malware type, contrasting the antivirus design that is meant to countermeasure as many possible threats as possible.

Signature generation The sandbox analysis might help to generating signatures for *intrusion detection systems*[2], antivirus applications and other appliances. A signature in correlation with sandboxes are a characteristic byte pattern or rule set to filter out certain events, files or actions [Rehman, 2003].

From the above three functionality areas in a sandbox, the final implemented system is utilising the signature generation, based on an automatic threat analysis. The system is outlined and designed starting from Chapter 6.

A malware analyst can utilise the power from sandboxes to deduce the functionality of a malware sample safely by monitoring network interfaces, disk usage and process spawning—and at the same time be able to trap, analyse or stop critical behaviour. This way of setting custom restrictions on the system, may be

[1] Information about the functionality is based on `Joebox`'s visions, located at the web page `http://www.joebox.org/vision.php` (accessed 2009-05-25). `Joebox` is a sandbox application that is explained in Section 4.3.1 on page 47.

[2] Intrusion detection systems are explained in Section 6.2.3 on page 67.

required prior to running an unknown program to sustain control over the malicious software.

4.2 Information prioritisation

The amount of information and data gained during a dynamic analysis is in most cases huge, and makes correctly prioritising the information a challenging task. However, certain actions by malware are more important than others and should receive particular attention [Moser et al., 2007]. A list of the most commonly observed actions performed by malware is shown below. Events such as these should be closely looked for during an analysis.

Check for Internet access As malware communicating remotely is common [Bailey et al., 2007], a sample checking for Internet access may indicate a suspicious program. This is especially so if the check happens at an early stage of the program flow, or the program immediately exits without any further notice when no Internet access is available.

File system and registry activity Malware often targets weaknesses in a particular version of a program, and tries to exploit these. Whether a program is installed on a Windows machine can in most cases be found in the Windows registry, and the rest directly on the file system. Research shows that over 80% of reported malware changes the file system, and over 70% of them changes the Windows registry [Bayer et al., 2008]. For these reasons, it is important to keep an eye up for file system and registry checks or creations especially if the program halts immediately if not found.

Check for a *mutual exclusion object* (mutex) A mutex is a lock used in programs to ensure one single execution from a method or the entire program in parallel. Malware often checks for mutex objects to guarantee only one instance of itself is active concurrently.

Read from file File accesses can lead to interesting discoveries. Samples reading files containing sensitive information are for example likely to be suspicious.

Read from/write to network Malware is highly dependent on a network connection to either spread or receive further commands from an attacker in command; if a significant part of the sample's instructions are reading from and/or writes to the network, special attention should be given to the event.

Spotting one or more of the actions above during an analysis do not necessarily indicate neither a harmful nor a safe file, as the actions may be perfectly legitimate and are just as commonly used in normal applications as malicious ones. The list should be used only to highlight certain events when the amount of information is overwhelming. Doing so can more easily give the analyst an overall view of the sample, but it is important that no conclusions regarding threat levels should be made solely on counting interesting elements spotted during an analysis.

4.3 Available complete sandbox-solutions

There exist a multitude of sandbox systems, and the most used publicly available ones are each briefly described in this section. Sandboxes are usually expensive software and/or hardware with a license that makes modifying functionality complicated. However, many of the vendors offer free usage to a web based interface for their product where users can execute samples on the vendor's servers and receive the results back. The economic resources for buying a sandbox solution to run locally, is not available in this thesis. The web interfaces are instead used.

4.3.1 Joebox

The sandbox service `Joebox` can be used by anyone with access to the Internet. Sample executable files are uploaded via its web interface, and analysed on one of their servers. `Joebox` uses a real system and not an emulated one, and utilise the fact that it is harder for malware to detect the analytical system. Results are emailed back to the user when ready, and are structured in open formats such as HTML and XML. Due to its parse-friendly format, the XML files make an integration with other tools more simple, but if that is not needed, HTML files are also available. `Joebox` does not supply network traffic monitoring in the result files, and a local installation of the application is not available. All sample files must venture through the web interface, and reports are sent unencrypted over the SMTP protocol.

4.3.2 CWSandbox

`CWSandbox` has similar functionality found in `Joebox`, and, at a general level, they look like clones of each other. There are still a few differences, especially in their different designs. Though the results from `CWSandbox` contains network traffic as opposed to `Joebox`, the different reports are quite similar. Sample uploads must be made through their web site for non-commercial licenses. `CWSandbox` runs in a *simulated system environment* [Willems et al., 2007]. Such an environment utilise virtualization products, contrasting `Joebox` which runs in an actual system.

4.3.3 Anubis

`Anubis` is yet another sandbox project, and as `Joebox`, it spawned from an academic research project. Its predecessor was called `TTAnalyze` and had origin from a masters thesis [Bayer, 2005]. Further developments are now performed by International Secure Systems Lab as the service `Anubis`. Usage of the service requires uploading a sample through the web interface. One of `Anubis`' positive properties is a published script for automatically submit samples to the web interface. An email address must be enclosed so results can be emailed back to the submitter. The `Anubis` service runs in the `QEmu` open source virtual environment.

4.3.4 Zero Wine

An open source research project called `Zero Wine` was released late in 2008. The project has functionality close up to what the commercial sandboxes offers, but is unique of its kind by releasing its source code publicly. Zero Wine ships with a Debian based Linux operating system running in a virtual machine. With a ready-to-use web server and interface that can be used to upload sample files, Zero Wine is a quite interesting project. The uploaded samples are executed on a Linux based host using a *translation layer* able to load programs for Microsoft Windows. This is required since uploaded samples are assumed PE-based files for the Windows operating systems. API calls are monitored, collected and aggregated; and a report is produced and shown to the user. The Zero Wine software is too new to be extensively tested, and it is not very widespread. Due to its publicly available source code it is still a unique project. Zero Wine and its usage of the translation layer “Wine” is studied more in detail in Section 6.5 on page 79.

4.4 In need for applications running locally

While access to a free web upload interface is sufficient for most users, sensitive malware that is targeted against a particular firm or company is not necessarily a software that should be spread further to a third party service. As mentioned in Section 2.7.4 on page 32, software samples may even be classified—making uploading to the Internet illegal by law. In consequence, it is important that any sandbox system used in this thesis has an open source license, to be modifiable and freely available. This property ensures the software can be used locally without a large license fee. Zero Wine is currently the only sandbox solution that complies with this requirement.

4.5 Available smaller dynamic analytical tools

This section covers the most relevant tools used in the dynamic analysis phase that are relevant for this thesis. This is not a complete list of tools used in a dynamic analysis. A more extensive list can be found in the document for this thesis’ preliminary project [Krister, 2008].

4.5.1 Sysinternals’ system information utilities

Sysinternals’ project consists of dozens of administration and diagnostic utilities for the Windows operating system. There are over sixty applications on the project’s list of tools, including coverage for both troubleshooting and diagnostic tools as well as monitoring for internal activity and network traffic. When performing dynamic analyses, some of the tools can help keep track of changes to the file system or additions to the registry. In addition, malware often tries to open connections to foreign hosts, and the Sysinternals tools can give the analyst immediate notice. The most relevant tools for malware analysis from the Sysinternals package is briefly discussed below.

TCPView

Imagine you are working on a computer that is suspected to be infected with malicious software. Malware is often working as a background process and is waiting for remote commands before the actual malicious events happen. Naturally, you are curious of which hosts the computer is sharing information with, or if it is listening for incoming requests. `TCPView` can be used to gain such information, and it is configurable to display the information you need and strip away the unnecessary part. The `TCPView` application does however only list the connections, not the data traffic. In a way, `TCPView` can be described as a graphical user interface to structure and parse information from the `netstat` program shipped with Windows installations.

Process monitor

To be able to spread through other processes, run at each system boot or store non-volatile information, malware often writes and reads to the file system and registry [Bayer et al., 2008; Erdélyi, 2004]. To understand how a particular malware sample works, it is for this reason essential to know *what kind* of data being stored and *where* to locate it. After executing a binary file for dynamic analysis, it may or may not tamper with the computer. `Process monitor` assists the analyst with a list of changes done by the started process. Since this is such a common procedure to expect from malicious programs, `Process monitor` is one of the core utilities in the malware hunting toolkit.

4.5.2 Wireshark

`Wireshark` is a network protocol analysis tool that saw the bright of the day in 1998. The continuous improvements of this freely available and open source software have made it strong, enriched with add-ons and it is now the de-facto standard for sniffing and reading network traffic. With the support for decryption of IPSec, SSL, WEP and Kerberos, even encrypted data can be read as decrypted text using this powerful tool. The main usage of this application regarding malware analysis is its basic data capturing properties. It shows the content of data traffic, compared to `TCPView`, which show only a list of active connections. For this reason, `Wireshark` is a more in-depth tool for analysing network data. Due to its level of detail and lack of connecting network traffic to system processes, it is not used as a replacement for `TCPView` but rather as an addition. A screen capture of `Wireshark` used to capture network traffic is shown in Appendix B.

4.6 Summary of described tools and solutions

Table 4.1 on the next page is shown to give an overview over the tools and sandbox systems discussed in this chapter.

4.7 Problems with dynamic analysis

Dynamic analysis is far from an exact science [Valli and Brand, 2008]. How to perform a successful analysis differs significantly depending on the complexity and behaviour of the malware sample, available resources, and each analyst's experience. There are also design issues with the dynamic analysis phase, making it error prone in some cases. This section covers the most well known weaknesses and problems regarding a general dynamic analysis.

Name	Comment
`Joebox`	Commercial sandbox solution. A free version is available as a web based interface that run analyses remotely.
`CWSandbox`	Commercial sandbox solution. A free version is available as a web based interface that run analyses remotely.
`Anubis`	Online sandbox solution. Only a web based interface is available. An uploading script is made available.
`Zero Wine`	Free sandbox solution. Unique for its open source license, making it modifiable and possible to use locally.
`TCPView`	Windows application that monitors simple data (source and destination) regarding network traffic.
`Process monitor`	Windows application that monitors changes to the file system and registry.
`Wireshark`	Monitors all kinds of details regarding network traffic. It is open source and cross-platform.

Table 4.1: An overview over the analysis tools described.

4.7.1 Observing single path executions

One of the main weaknesses with dynamic analysis comes from a limitation in its design. Dynamic analysis is all about observing programs executing as it would in a normal machine, preferably in an environment equal to an environment expected by the malicious program. The problem using such a black box approach is that it is hard for an analysis to guarantee *all* possible execution paths are covered. Only one or a few single paths are examined in an analysis, using a finite set of input parameters. A malware sample in an analysis may remain dormant until some particular event happens, and if the event happens rare, there is a large chance that the analyst draws incorrect conclusions about the risk from the sample [Moser et al., 2007]. To understand this, consider the code example shown in Listing 4.1 on the facing page. The intended effect of the program is to erase "a set of critical data", but only on the 15th day of each month. Therefore, even if the program runs in an analysis, the harmful effect is not seen until the system clock reaches the 15th day of month. Assuming the system clock is set correctly, a dynamic analysis is in this case only able to examine the harmful effects one day of each month. Conditional executions with malicious intentions are called *trigger-based behaviour*, and can be extremely hard to find during a dynamic analysis if hidden properly [Brumley et al., 2007]. Of course, in this example the program reads from the system clock and the date can easily be changed manually, but the analyst is not necessarily aware of the conditional check. In the general case, trigger-based behaviour can be camouflaged to run only when certain properly selected conditions are met. A static analysis on the other hand does not suffer from this weakness since the entire program flow is eventually deduced if the correct techniques are used.

There are certain methods available to force multiple execution paths. One is by keeping track of conditional branches in the program during a normal execution, and later on revert to each of the conditions with changed parameter value(s) [Moser et al., 2007]. An example can be shown from the code above. Say the current date is the 10th of January, so running the program yields "10" to the `today` variable. The

```
# Method for erasing a victim's valuable data
def erase_all_critical_data
  # Insert harmful code here
end

while true
  # today variable gives a fixnum (Ruby integer) from 1 to 31
  # depending on which current day of month the system clock
  # is currently set to
  today = Time.now.day

  # Call harmful method only on the 15th day of the month
  erase_all_critical_data if today == 15

  sleep 120 # Sleep two minutes to avoid wasting the CPU (
      prevent detection from the user)
end
```

Listing 4.1: Ruby code showing the problem of single path executions during dynamic analysis of malware. Ruby is selected in this example simply because it is easy to read and very expressive.

program continues, but when the conditional "`if`-clause" arise, control is taken from the program and a snapshot of the process is made. After a successful snapshot, control is given back to the program. The conditional clause in the program returns `false`, and the execution loops inside the `while` loop. Since there is *at least* one uncovered path from the conditional clause, the process is reverted to the state at the conditional `if`-clause by using the stored snapshot. Before restarting the execution, the `time` variable is changed in-memory to force the conditional clause to return `true` instead of the earlier value which was `false`. The conditional check changes, and the harmful method executes. The analyst is then able to observe its effect, even when the day is not the 15th day of the month.

The method observes *conditional jumps* from instructions executed by the program, and inverts the conditions until all *branches* in the program are executed. A conditional `if`-clause in a high-level program compiles to a "conditional jump" instruction in low-level code, leading to two or more execution branches. Assuming an `if`-clause checks for either `true` or `false` values, and there are no further branches in the program, all of the program's execution paths are then covered. Figure 4.2 on the next page shows a graphical example of a condition giving two branches from a simple `if`-clause.

4.7.2 Detection of analysis environments

Even if virtual environments are meant to act like a complete replica of a normal computer, research shows that algorithms and methods available to detect the environment exists [Garfinkel et al., 2007; Quist and Smith, 2000]. Malware analysts often work in virtual environments to control the malware more easily. When the malware detects a virtual environment, it may act differently than in a normal system [Haukli, 2007]. The detection methods are commonly called *anti virtual*

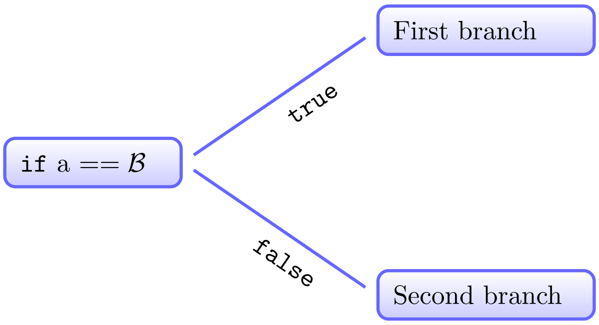


Figure 4.2: Conditional `if`-clause giving two branches in the program flow (the clause can evaluate to `true` or `false`). Both branches contain program logic and the conditional clause result must be tampered with to force the program to visit both branches, and thus cover all execution paths.

machine methods, but is a subgroup of methods able to detect general environments often used by malware analysts. The parent group is called *anti-debugging methods*, and covers VM detection as well as other debugger detection. A list of possible actions by the malware upon detection is shown below.

- Escape the protected environment and try to infect the host system.
- Signal home and report the current IP address or subnet. CERT and other analytical networks can easily be a prioritised target for DDoS attacks if their IP addresses are exposed to the malware producers, as they are their number one enemies.
- Choose to run differently than in a normal system environment, maybe fooling the analyst to falsely conclude a safe threat level on the sample.
- Choose not to run at all.

Avoiding the normal malicious execution if a virtual environment is detected is common [Ferrie, 2006; Raffetseder et al., 2007]. Using anti-debugging methods stealth the real functionality of the malware, and avoids exposing its functionality to the analysts working in protected virtual environments. If the analyst even notices the irregularity in the program, the malware forces him or her to one of the following actions.

- Move the dynamic analysis to a normal operating system. The approach leads to a more difficult process of reversing changes to the system state by the malware.
- Move on to a static analysis, which is potentially much more resource demanding than a dynamic one.

The few lines of code in Listing 4.2 on the facing page is an implementation written in C by the Polish security researcher, Rutkowska. These lines are one of the results from her "Red Pill project" [Rutkowska, 2004], and is one code example able to detect if the operating system is running in a virtual machine or not[3].

[3] There are some limitations, but the listing is just meant as an example to show what kind of methods available for detecting virtual environments.

```c
int swallow_redpill () {
 unsigned char m[2+4], rpill[] = "\x0f\x01\x0d\x00\x00\x00\x00\xc3";
 *((unsigned*)&rpill[3]) = (unsigned)m;
 ((void(*)())&rpill)();
 return (m[5]>0xd0) ? 1 : 0;
}
```

Listing 4.2: C code returning a non-zero value if a virtualized environment is detected. The details of the code is not important, and the importance of the figure is to observe how easy (in lines of code) it can be to detect a virtual machine. Only a few lines of carefully selected code is necessary for a detection [Rutkowska, 2004].

Dynamic analyses might fail if a sample has a Red Pill implementation, or similar, in its core that halts the sample if a virtual environment is detected. Static analysis and on demand virus scanning on the other hand do not study the programs while running, so such anti-debugging methods are not a problem for these phases. Software may try to avoid reverse engineering techniques to protect its own code, and is often applied due to software licenses and anti-piracy [Gagnon et al., 2007]. Except for this, there are few reasons for a legitimate program to look for a debugging environment. Therefore, anti-debugger code stubs are good indications of a malicious sample that alters its normal flow depending on what kind of environment it is running in.

4.7.3 Scalability issues

The large amount of discovered malware, together with the increasing level of sophisticated functionality makes the fight against malware complex and tough. Days or weeks of analysing could be required to deduce the functionality of a program, and it is difficult to estimate the value of an analysis before seeing the actual results. When performing a dynamic analysis manually, the analyst may for that reason be required to prioritise samples above others due to limited time available. Important samples will be skipped due to this fact, and can result in severe consequences depending on the *actual* importance and danger of the sample. Automating more of the tasks during dynamic analyses is a possible method to increase the ratio between analysed and skipped samples.

Chapter 5

A Malware Analysis Scenario

To understand choices made later in this thesis, an analysis scenario of a suspicious sample is given in this chapter. The scenario also helps the reader to familiarise with the structure and execution of a malware analysis by presenting a practical case. Additionally, results found in the scenario are used for measuring tests results of the final system. The test results are located in Section 9.4 on page 108. The scenario does not cover all possible ways of doing an analysis, but is merely an example of one. Task automation is one of the possibilities to reduce time spent in an analysis, so the chapter includes discussions about automation of tasks where appropriate.

The chapter contains a description of the general setup and physical deployment of the analytical environment, and continues with an analysis of the two first phases in a malware analysis: the surface scanning and the dynamic analysis. The last phase, which is the static analysis, is *not* the focus for this thesis, and consequently excluded from the scenario and chapter.

5.1 Initial setup and environment overview

The system setup is similar to the practical example shown in Figure 4.1 on page 43, using a `Windows XP` based "victim" machine, and an `Ubuntu Linux` based router. The malware is executed on the victim machine, hence the name. Events such as registry changes and file modification are observed locally on the same victim machine. Network traffic is observed at both machines, but remote communication requests from the victim machine are trapped at the router, making the victim machine unable to contact other machines.

The victim machine is a virtual machine located in the same computer as the router, making administration and setup easier than introducing an additional physical machine. A virtual machine offers valuable properties as seen in Section 4.1.3 on page 44, and makes it easy to revert to a clean state using a stored system snapshot. Virtualization software by VMware is used for this task reasoned for by the same causes this thesis' preliminary project did [Krister, 2008]; (1) the software must support a job automation interface, and (2) the software should be stable and properly tested.

A network interface having full Internet access is enabled on the router. The virtual machine (victim) has no access to this interface, but instead a separate interface that is set up on the router dedicated for the virtual machine. This network interface uses a fixed IP subnet and is created for unique use in the scenario. By using router software, the interfaces can be connected so certain network traffic from the malware network interface is able to pass through out on the Internet.

The sample to be used in the scenario is obtained from the web page `http://www.offensivecomputing.net`, which has a large database of discovered malware. An infected sample spread by the well known `Asprox` botnet, a successful botnet that utilises SQL injection vulnerabilities [Provos et al., 2009], is fetched from their web page and used as a subject for analysis in the scenario. The reason for choosing an `Asprox` sample is its representation of malware as complex software, and its ability to clarify the difficulty of concluding a malware analyses a correct threat level.

5.2 Surface scanning

The surface scanning phase consists mostly of automated tasks [Wedum, 2008], but the scenario ventures through the phase as they were done manually.

5.2.1 Generating and comparing hash sums

First, hash sums of the samples are made. The programs `md5sum`, `sha256sum` and `sha512sum` are used to make unique hash sums on the sample. The most prominent visible difference from them is the output string length (hash), which are respectively 128, 256 and 512 bytes. In addition, their level of security varies, but that is not important in this context. The output is shown below.

```
$ md5sum asprox.exe
1311f650aa1209a3ec962b6a9a38fc98 asprox.exe

$ sha256sum asprox.exe
9885504c9d21193735adbb1f8c9cb53e
9c044d0fb0f9449df12bbe537820838a asprox.exe

$ sha512sum asprox.exe
0f0caf930fea5fb5d7f3f2750e38d740
c29bf205934e6cce5accbb50b9cda683
9202b5b7bde8cdf87259ec91afe125c9
6041e87a3f0ee64826cb191872fd654e asprox.exe
```

If any of the hash sums are found in an previously completed analysis, the sample is already analysed[1], and the process is complete. The task is trivial to automate and analysts can easily arrange automatic hash sum generation on uploaded samples in their analysis systems.

[1] Basing results on up to three three distinct hash generation algorithms makes *hash collisions* very unlikely to happen [Biham and Chen, 2004]. A collision occurs when a hash algorithm outputs the same hash for two distinct files.

5.2.2 Antivirus coverage testing

Assuming the hash sums are not already known, the analysis process continues. The opinion antivirus vendors give the sample is at this time valuable. That is, if the vendors consider the sample harmful or not. Antivirus checks are performed using a multiple of antivirus products from different vendors, since the opinion from one vendor is not necessarily the correct one. The implemented result from this thesis' preliminary project was a program made to automate antivirus scans by a multiple of antivirus programs [Krister, 2008]. The program is called `vmcom`, and can initialise concurrent antivirus scanning from different products using virtual machines. `vmcom` is used in this scenario using ten different antivirus products to scan the sample automatically. Aggregated results are shown as comma separated columns in Listing 5.1. Column two displays any threat found by each product, where "0" means no threat found.

```
$ ./vmcom.pl --scan /tmp/asprox.exe
PRODUCT,                  THREAT FOUND,INFO,ERROR
Avira AntiVir,            0,0,0
AVG Free,                 -1,Error,1
Panda Engine 1.4.3,       0,0,0
ClamWin,                  0,0,0
BitDefender Free,         0,0,0
Kaspersky Anti-Virus,     0,0,0
F-Secure AntiVirus,       -1,Error,1
ESET NOD32,               probably a variant of Win32/Agent.NEQ
    trojan,0,0
Trend Micro,              Mal_Asprox,0,0
Avast Professional,       Win32:Agent-GPS [Trj],0,0
McAfee,                   0,0,0
```

Listing 5.1: Aggregated results from an antivirus scan by ten different antivirus products.

The results show that using a multiple of antivirus products when scanning are essential for the check to be of any worth. Only three of total ten antivirus products are considering the sample harmful (detection coverage = $\frac{3}{10}$), even if the sample had existed for some time—and should be picked up by the various antivirus signature databases. However, not all of the virus definition databases were up to date with the latest signatures during the execution, and two of the products refused to scan the sample due to expired software licenses (leading to two "error"s in the results). Handling these issues would probably increase the detection coverage, but it is not vital in this scenario. It would be however, during an actual analysis where all antivirus software must be up to date to allow analysts using leading edge opinions from the different vendors.

One of `vmcom`'s key requirements was to allow an easy integration to existing analysis systems, so automating the antivirus coverage testing is already a straightforward task.

5.2.3 Find string data

Files contain readable text (strings) even if they are in binary form [Schultz et al., 2001]. While the binary data is most interesting during the static analysis, readable text is still relevant during the surface scanning. Valuable information is gained by listing string data such as API calls, XML data and even IP addresses. Analysing structure of API calls can help the analyst to deduce the overall functionality of the sample. XML can contain interesting values, words or structure, and IP addresses can be used to signify which host(s) the malware tries establishing a connection to. API calls are used to deduce functionality done locally, which can be file system modification, observing processes or any other functionality related to the local operating system.

With assistance from the free program `GNU strings`, shipped with most Linux distributions, locating text data in files is a trivial procedure. However, on the current malware sample, the output from `GNU strings` is mostly unreadable for the human eye—even when the data is plain text. The output is shown in Listing 5.2, where the first running of `GNU strings` counts the amount of lines found (using the program `GNU wc` (word count)), while the second lists the actual output. Comments added are started with a hash sign (#).

```
$ strings /tmp/asprox.exe | wc -l
802

$ strings /tmp/asprox.exe
This program cannot be run in DOS mode.
Rich
TVtI
.text
‘.rdata
@.data
.rsrc
jH:L
$X A
h(!A
$P!A
$x!A
$D‘‘A
$t’’A
$t#A
Y_^[
# Around 750 lines of more gibberish are removed here

# API calls follows
memcpy
free
malloc
fclose
fwrite
fopen
strncmp
memmove
strlen
# End of API calls
```

Listing 5.2: String content of the sample used in the analysis scenario. The listing is shown to display the few readable strings that are present in the sample.

Only a few API calls are possible to parse from the sample file. The reason for lacking intelligible strings may suggest the sample is packed or otherwise obfuscated [Lyda and Hamrock, 2007]. In the current case, the few API calls shown in clear text are most certainly methods used to decompress and/or decrypt the content during running time. Automating execution of `GNU strings` can be easily accomplished, and is available in sandbox solutions discussed in Section 4.1.5 on page 45.

5.2.4 Packer information

If the sample is packed and/or encrypted, the analyst benefits from this knowledge since it definitely should raise a warning signal. `PEiD` is a program used to detect packer technologies commonly used by PE-based malware. Using `PEiD` on the sample shows "`Microsoft Visual C++`" is used to pack the sample. If necessary, the analyst can use this information in a static analysis to unpack the sample and locate the actual instructions. A recursive packing or encryption can be applied below the first level of compression, to further complicate the de-obfuscation process. A screen capture of `PEiD`'s results is shown in Appendix B.

5.3 Dynamic analysis

It is important to be aware of the danger of running an `Asprox` sample, and the possible consequences to the system and network if it is not handled properly. This is especially true during the dynamic analysis, where the sample's effects to the system and the environment are observed while it is running.

The victim machine is now used for the first time in the scenario. A clean install of a `Windows XP` operating system is installed as a virtual machine. The scenario does not describe how to install the OS, and the documentation from VMware should be used to study the installation details if necessary.

Generally, to be as protected from vulnerability exploits as possible, all security patches should be installed. However, for the current scenario, it is *preferable* to be unprotected. This allows malware that targets a specific vulnerability to discover the weak point. It is still one important issue to consider concerning patching or not, namely the single path execution problem discussed in Section 4.7 on page 49. Applying security patches change the operating system environment, and malware terminating its execution when the environment is not as it expects is common. Of course, this also applies the other way around. The malware may just as possible target a specific *patched* version, as new program updates can introduce new vulnerabilities [Lippmann et al., 2002, section 4]. The system is kept non-patched as the gain from patching the system is unknown, and it is a larger probability a non-patched system is vulnerable than an updated one [Arbaugh et al., 2000].

Prior to running the sample, key applications are installed on the operating system. Two of the programs that were explained in Section 4.5 on page 48, `ProcessExplorer` and `TCPView`, are installed and set to monitor respectively running processes, and network events on the victim machine. `Wireshark` is installed on the router and set to capture traffic going through the dedicated network interface.

Many Internet services rely on the DNS protocol to work, so such traffic is accepted through the network and forwarded on to the Internet. Additionally, it is

Host name	Maps to IP address
`203-174-83-75.rev.ne.com.sg`	203.174.83.75
`ha-42.web.de`	217.72.195.42
`www.web.de`	217.72.195.42
`www.yahoo.com`	69.147.76.15
`ns.uk2.net`	Expired

Table 5.1: Host names contacted by the sample during the scenario. Their IP address mappings at the time of writing (2009-05-26) are shown in the second column. `ns.uk2.net` has an expired mapping.

possible the malware halts if an Internet connection is not found if the malware requires remote communication. Therefore, if the malware requests a particular service remotely, more traffic should be allowed to pass the router barrier. Several solutions can be used for the purpose, but the `iptables` IP filtering software is selected since it ships with most Linux distributions, is open source, and has all the functionality needed. `iptables` is based on command line functions so an executable batch script is created. See Appendix C for the script used in the scenario.

When all programs are set up, the sample is executed. Immediate action is seen on all observation points, and a summary is shown below.

5.3.1 Connection attempts

Several attempts of establishing connection to hosts are made, but the `iptables` script is currently denying the attempts out on the Internet. The malware initialise the connection using fixed host names, and by observing their respective domain suffixes, they seem to belong to hosts in Singapore (`.sg`) and Germany (`.de`). The malware also attempts to contact the Yahoo web server. The host names in Table 5.1 are looked up using standard DNS queries, which *are* allowed by the `iptables` script. Their IP address matches at the time of writing are shown in the right column in the table. A screen capture of `Wireshark` while actively monitoring the virtual machine's network interface is shown in Appendix B.

5.3.2 Processes

The sample created a system service named `aspimgr.exe`, and inserted it into the operating system together with the other running operating system services. The service listened on port 80, which is usually a port used for web traffic. Attempting to connect to this service was tried using raw TCP connections with help from `netcat` and telnet. However, the service did not reply on the attempts, and it is possible the new system service waits for a particular TCP sequence, a concrete source IP address or another hidden event that is hard to deduce from a dynamic analysis.

5.3.3 File system changes

During the observation of network traffic, the sample file had already done a handful of API calls and file system operations. `Process Monitor` picked up 2082 events spanning from opening files, creating new files and deleting files. The sample scanned for FTP login credentials in common files created by the `CuteFTP` programs, and was the most obvious suspicious operation observed. The executed malware sample deleted itself by creating and running a batch script after it successfully created the `aspimgr.exe` service. Monitoring the new system service process, around additional 1000 file system events were found, including ongoing process profiling, and reading and writing a binary file called `s32.txt` which was written to the main Windows operating system folder. After a couple of minutes after the process' creation, it started looking for other files with similar names as `s32.txt`, but none of these files existed. A theory is by using the server listening on port 80, an attacker could create the files searched by the service and populate them with any (malicious) instructions. However, that is just a speculation and a possibility, and due to the binary file format hard to find out in a dynamic analysis.

5.3.4 Changing routing

Manipulating a particular file[2] on the victim machine makes it possible to fool a DNS query and spoof the corresponding IP address. Inserting the router's IP address for the hosts earlier shown, makes the malware send traffic to the router instead of the host's actual IP addresses.

To be able to fetch the data, and respond to the connections, a server listening on the particular port must be set up. Again, the program `netcat` is capable of exactly this, and is used for the purpose. The malware issued several connection requests, having invalid data that `netcat` was unable to parse properly. It is possible that the sample sent malformed network packets, but analysing the traffic further is out of the scope for this thesis.

5.4 Reflections

By using a practical example, the scenario has shown that analytical techniques can assist deducing the functionality and behaviour from a (malicious) sample. Surface scanning is a quick process that indicates valuable information for the rest of the analysis, such as feedback from antivirus software. The surface scanning also ensures the sample is not previously analysed.

The dynamic analysis phase is more complex and non-deterministic. The environment where the sample runs is monitored for suspicious activity, but it is vital the analyst remains in control the entire execution period. How the phase is structured, and which tasks to apply varies on how the analysed sample operates, and can differ significantly from time to time. The scenario shows the analysed sample's search for login credentials in the file system, and its connection attempts to a set of host names. However, this is not necessarily the sample's main goals. It is possible the analysis missed one or more important triggers. These triggers might have led to a different program flow than the observed execution. If the sample is utilising anti-debugging checks, vital triggers are likely to be missed in a dynamic analysis.

[2] The file `c:\windows\System32\drivers\etc\hosts` is a flat file containing host-to-IP addresses. Thus it acts as a very small static DNS database. DNS queries from the local machine search this file before asking remote DNS servers, and if a match is found, a host-to-IP mapping is assumed to be correct [Hare and Siyan, 1996, chapter 1].

It is difficult to guarantee a complete deducing of the sample's functionality based on information from a dynamic analysis phase, but for many cases, a dynamic analysis is valuable and sufficient. Some cases must continue the analysis with a static analysis, where the binary code is reverse engineered, making it possible to guarantee to observe the entire program flow if sufficient resources are available.

Chapter 6

Concretisation of the Task

This chapter contains an interpretation of the problem description and narrows it down to make it more specific. An in-depth study of which properties a system that can reduce necessary human intervention in a malware analysis is carried out. The chapter decides, in particular and at a general level, *what* the system does, and why this is chosen. More concretised requirements for the system are found in Chapter 7.

The chapter first presents related work in the field of automatic malware analysis, and flows by describing a set of possible problems to solve with a new system implementation. A selection of these approaches is then made based on a prioritisation. To be able to finalise the selected approach using the limited time available, existing software solutions must be utilised; the chapter therefore follows with a survey of software that can assist the realisation of the system, and presents which of these that are in use—and why. The chapter flows by presenting a detailed description of `Zero Wine`, which is a central part of the final system. The chapter ends with a brief summary of the selections made in the chapter.

6.1 Related work

Automating malware analyses are vital to keep up with the prevalence, and increase of complexity in malware. This section covers techniques and published research about automation of analyses, where focus is kept on processes linked to the dynamic analysis phase.

Bohne [2008] suggested an automatic way of uncompressing files packed using packing technologies. A prototype called "`Pandora's Bochs`" was developed that is able to monitor running, initially compressed, processes and automatically save the decompressed version from memory as soon as it is unpacked.

Brumley et al. [2007] published research that shows how to detect conditional execution flow in malware automatically, and how to find inputs that manipulates the triggers. Conditionally executions, also called "trigger-based behaviour", is explained in Section 4.7 on page 49. A prototype called "`Minesweeper`" was created, and works on binary programs. It is able to manipulating the triggers automatically, forcing malicious behaviour from these binary samples.

Another approach is proposed by Yin et al. [2007]. The authors designed the "`Panorama`" system, which is able to detect malicious behaviour in running programs based on what they call "sharing of fundamental characteristics". The characteristics are patterns commonly seen in the observed programs. The approach is similar to the heuristic scanning techniques found in antivirus products.

Singh et al. [2004] published similar research, and proposed an automated approach for detecting *previously unknown* malware. That is, automating the generation of signatures when new suspicious software is detected. The authors base the design upon two key behavioural characteristics: (1) a common exploit sequence found in malicious software, and (2) unique sources that generate infections, together with destinations being targeted.

Another approach is suggested by Li et al. [2008b]. The authors developed the system "`AGIS`", which is able to detect new malware infections by monitoring sample's behaviour in a controlled environment. Upon detection, the system is able to generate a signature automatically. The signature generation is similar to an antivirus signature, which is usually created in a manual process.

An completely different angle was suggested by Kim and Karp [2004]. The authors developed a prototype called "`Autograph`" able to automatically generate intrusion detection (IDS) signatures for the `Bro` IDS. The explanation of intrusion detection systems is deferred until Section 6.2.3 on page 67, but IDSes cover several hosts concurrently and look for suspicious activity in network external data or internal traffic for each host. The generation is based upon TCP transport content, and the signatures are generated by analysing payloads in the traffic content—looking for specific suspicious network flow. The generated signatures are heavily based on *payload* found in malware, and not the data traffic's source or destination. The observation part of the approach is similar to what is seen in antivirus products. However, while an antivirus product usually observes *one* host, `Autograph` uses IDSes to observe an entire *network*.

Another version of the prototype, "`Polygraph`", is using the same design but with an improved the signature generation for polymorphic malware [Newsome et al., 2005]. A polymorphic technique is a way of mutating malware, and is explained in Section 2.4.6 on page 21. The `Polygraph` application collects specific substrings from the polymorphic malware that is *known to be* alike in the different variances of the malware, and gives fewer false negatives than its predecessor `Autograph`.

Kreibich and Crowcroft [2004] describe "`Honeycomb`", a system that gathers suspicious traffic sent to hosts that are assumed to receive *no* legitimate traffic. All traffic to such a system, called a *Honeypot*, is considered suspicious [Spitzner, 2003]. Signatures are automatically generated based on pattern matching techniques and similarities in the different network communication protocols from the observed traffic data.

Hsu et al. [2006] have designed a framework to remove malware from a system while preserving complete system integrity; no undesirable side effects are made by removing the malware. The framework utilises system log files to reverse (malicious) actions applied by malware.

Sandboxes are software using techniques to restrict executions of applications in a controlled environment [Prevelakis and Spinellis, 2001]. There exists a multiple of sandbox solutions, some of them already supporting automated execution and analysing of samples with extensive report generation. The sandbox phenomena is a frequently used method in dynamic analyses, but a discussion is not repeated as Section 4.1.5 on page 45 is dedicated to the topic.

6.2 Available approaches

According to **RG.04** (the fourth result goal), the implemented system shall automate one (or more) dynamic analysis tasks. However, *how*, or even *which* task(s) are currently not decided. Therefore, prior to drafting a design for a new system, an interpretation of the problem description must be made; the interpretation is presented in this section. The relevant part regarding automation is quoted from the problem description and shown below.

> *The student is free to automate any analysis process that is regarded dynamic, but it would be preferable if the information gained from the automated analysis is* actionable*. That is, the information gained is directly useful for handling the particular incident(s) where the malware is involved.*

The last half of the above quote can be ambiguously interpreted, so a concrete interpretation is made. Representatives from NorCERT defined “actionable information” as data that effectively can be used to handle *incidents* regarding malware. An incident is a violation or imminent threat of violation of computer security policies, acceptable use policies or standard security practices [Scarfone and Mell, 2007].

An incident is usually a reported case or an observed event where there is reason to believe malware, compromised hosts and/or cyber crime is involved. Incidents can be malicious, but does not need to be. A person can as an example, accidentally type the wrong host address and attempt to connect to a different system than originally expected without authorisation. Such an action is not malicious, but is still considered an incident.

One additional important part of the problem description is about NoCERT’s malware analysis environments, and is quoted below.

> *The project may optionally look into integrating the final system with NorCERT’s internal system for handling malware samples.*

Available approaches are prioritised having integration with NorCERT’s internal system in mind. However, *look into* is emphasised, and the integration with the internal system is to be delivered as a draft, and *not* implemented.

Three approaches, having a varied level of actionable output, are presented below. The approaches shown are having the following three characteristics: (1) the approach is believed to be useful in a dynamic malware analysis, (2) is feasible with the given time limitations, and (3) does not prevent the fulfilment of stated result goals. It is believed to be other possible approaches, but the shown approaches are considered a sufficient amount of options. Not everything from all of these approaches can be implemented due to time limitations. For that reason, a prioritisation of the approaches is performed in Section 6.3 on page 72.

6.2.1 Further development of sandbox solutions

Following Section 4.1.5 on page 45, one of the tasks subject to automation is implementing a sandbox solution able to run locally. Realising a new standalone sandbox solution is feasible, but it is already done several times before and the sandbox concept has existed a long time. However, since the list of publicly available sandboxes is far from extensive, a new sandbox with an open source license can give the sandbox community a non-commercial competitor. This is, however, not sufficient in this case, and producing a new sandbox solution is not chosen for the following reasons.

- A handful of available sandbox solutions already exists, including at least one publicly available solution (`Zero Wine`) licensed under the General Public License (GPL)[1].
- The existing sandbox solutions available are complex pieces of software developed by dedicated large teams with a large amount of resources available. The possibility of any breakthrough or massive improvements from this thesis is thus slim.

It is however possible to *improve* one or more existing sandbox solutions available, and at the same time look into an integration with NorCERT's internal system for handling malware samples. Such an approach avoids starting a sandbox development from scratch, and instead utilises work and best practice methods from existing sandbox systems. The improvements can be given to NorCERT in source code so any requested changed could be applied whenever needed.

6.2.2 Antivirus coverage testing during dynamic analysis

Further improvements on this thesis' preliminary project is possible. That is, further developing on the implementation delivered at the end of the project (`vmcom`) [Krister, 2008]. Malware today is often camouflaged, packed and/or encrypted, as discussed in Chapter 2. This shell of "protection" is frequently modified by manipulating internal encryption keys, packing algorithms or similar procedures, effectively evading detection. Antivirus applications are heavily based on signature based scanning. This means, prior to be able to recognise threats, a set of signatures must be loaded into the antivirus application for each threat to recognise. For that reason, an antivirus application might falsely consider a camouflaged malicious sample safe since it is currently not having signatures for the camouflaged variant of the malware.

A compressed executable unpacks, *in most cases*, its content before its real functionality is exposed to the system. Consequently, at some point during execution of a sample, its content is stored as de-obfuscated form in memory [Christodorescu et al., 2006]. The de-obfuscated sample is more easily detected by an antivirus application than obfuscated versions since it is in the original form of the malware [Royal et al., 2006]. A possible approach is therefore an automated antivirus scan on unpacked files and all de-obfuscated content as soon as they become available in an analysis. Additionally, all newly introduced files whether they are downloaded by the malware or simply spawned from the original sample can be checked automatically. By utilising the `vmcom` system that is already implemented in the thesis' preliminary project, an arbitrary amount of antivirus products can be automatically instructed to scan these files. The results could be aggregated and a report created to get the opinion from a multiple of antivirus products. However, there is one issue to consider shown below.

[1] See `http://www.gnu.org/licenses/gpl.html` for more information.

- Analysts are working in their own environments, using their own virtual machines, hardware snapshots, tools and operating systems. Implementing a solution able to properly observe all file creations and similar low level functionality could be difficult without assuming a particular type of analytical environment.

6.2.3 Generate signatures for network intrusion detection systems

Intrusion detection systems (IDSes) are software and/or hardware used to detect unwanted activity by monitoring and analysing events in a computer system or networks [Scarfone and Mell, 2007]. An IDS is usually able to record and store all information gathered in the process. Additionally, most IDSes collect and aggregate the potential large amount of information and produce clean reports periodically based on the collection. An IDS can cover a large set of hosts and can use traffic sensors to cover thousands of hosts concurrently. An IDS can also be more fine grained, focusing on one single host and its corresponding applications and running processes. However, all IDSes observe suspicious behaviour.

IDSes are limited to probing an asset and analyse the data, and do, by definition, nothing to directly *prevent* any suspicious behaviour. An IDS consists of a *sensor*, which is observing the network traffic and triggers alerts based on the observations. The sensor can be a regular computer with IDS-software, but just as well be a device with specialised hardware. A normal computer is usually the cheapest, while special devices can claim better performance. Different methods for observing traffic are available, where the most prominent are described below [Sanders, 2007; Tanenbaum, 2002].

Using hub-based networks

A *hub* is a repeating device that connects machines together in a network. Incoming traffic to the hub is sent to *all* connected machines, and since it is in most cases one single machine that expects the traffic data, a large quantity of unnecessary traffic is generated. Even if hubs suffer from this bandwidth problem, it is a positive property when observing traffic; it is easy to plug in the IDS in an available port on the hub. The IDS can then observe traffic to and from all other hosts connected to the hub. However, due to hubs' bandwidth problem, the devices are now rarely used.

Using switch-based networks

Switches are similar to hubs, but keep an internal table over connected machines to ensure traffic are sent to *one*, and only one, machine (the receiver). The table maps machines' IP addresses and *media access control (MAC)* addresses. The IP address scheme is used as addressing in the network layer from the OSI-model [Zimmermann, 1980], but the layer below, the data link layer, uses an addressing scheme called MAC. A MAC address is a 48 bit long address uniquely given to hardware interfaces on a network [Tanenbaum, 2002]. The switch keeps track over machines by mapping the addresses in the two layers and ensures traffic is not wastefully duplicated in the network. For that reason, observing traffic through a switch is more difficult than in a hub, but some switches can be configured to have a *spanning* port that all traffic is mirrored to. The spanning port is a particular port on the switch that receives traffic to and from all other ports, just as in a hub. If the spanning port does not have sufficient bandwidth to cover the combined traffic in the switch, the switch may be dropping packets if its capacity is reached.

ARP cache poisoning

The mapping between IP and MAC addresses internally in a switch is carried out using a protocol called ARP. A computer can receive another computer's traffic by forging the other machine's MAC address and publish the false information repeatedly to the switch [Welch and Lathrop, 2003]. The switch stores the mapping in an expiring cache, but the mapping remains in an incorrect state even when the data is expired, since the attacker repeatedly publishes false ARP data. This is called *ARP cache poisoning*, and is an attack commonly used in man-in-the-middle attacks [Gu and Hunt, 2005]. If a computer runs a cache poisoning attack on the switch with a forged MAC identical to the router in the network, all network traffic going to the router ventures through the attacker's machine, and can be observed. Forwarding the traffic to the router makes it hard to detect the irregularity. This is a man-in-the-middle attack, but is in this case used for a legitimate purpose in a network.

Using a router

Routers are similar to switches and the technology is located in the same OSI model layer as switches. A router is however, a more advanced device for forwarding network packets, used to connect different logical subnets in a network. Everyone with administrator privileges in a router can easily monitor all the traffic venturing through the device, and ready-to-use applications structuring and listing traffic flow are already available for free download.

Similar systems as IDSes with the additional capability to stop suspicious events are called *intrusion prevention systems* (IPSes). An IPS is an IDS blended with a firewall [Zhang et al., 2004], but the prevention capability of an IPS force constraints upon its physical deployment. It is not sufficient to simply tap the network traffic as with an IDS, but the flow in the network must venture *through* the IPS host so certain traffic can be dropped. Deploying the IPS system in (or as) a router device is sufficient to be able to drop the unwanted traffic. See Figure 6.1 on the facing page and Figure 6.2 on page 70 for an example of *network based*[2] IDS and IPS architecture, respectively. The figures show a firewall *outside* of the IDS/IPS sensor, enabling a *course grained* filtering. Coarse-grained filtering drops traffic that is obvious unwanted, allowing the IDS/IPS sensor to better utilise its CPU and memory capacity. Firewalls use CPU and memory at a very low rate compared to an IDS/IPS. The figures also show a management server connected to the IDS/IPS device. This server is used as a dedicated entry point to the IDS/IPS device to prevent non-authorised users from gaining entrance.

Signatures are used the same way for intrusion detection whether the traffic is stopped. Therefore, the physical deployment of the systems are not important

[2] There are several types of IDSes, but the network type is most relevant for this thesis. Read more about the different types of IDSes in Section 6.4.1 on page 73.

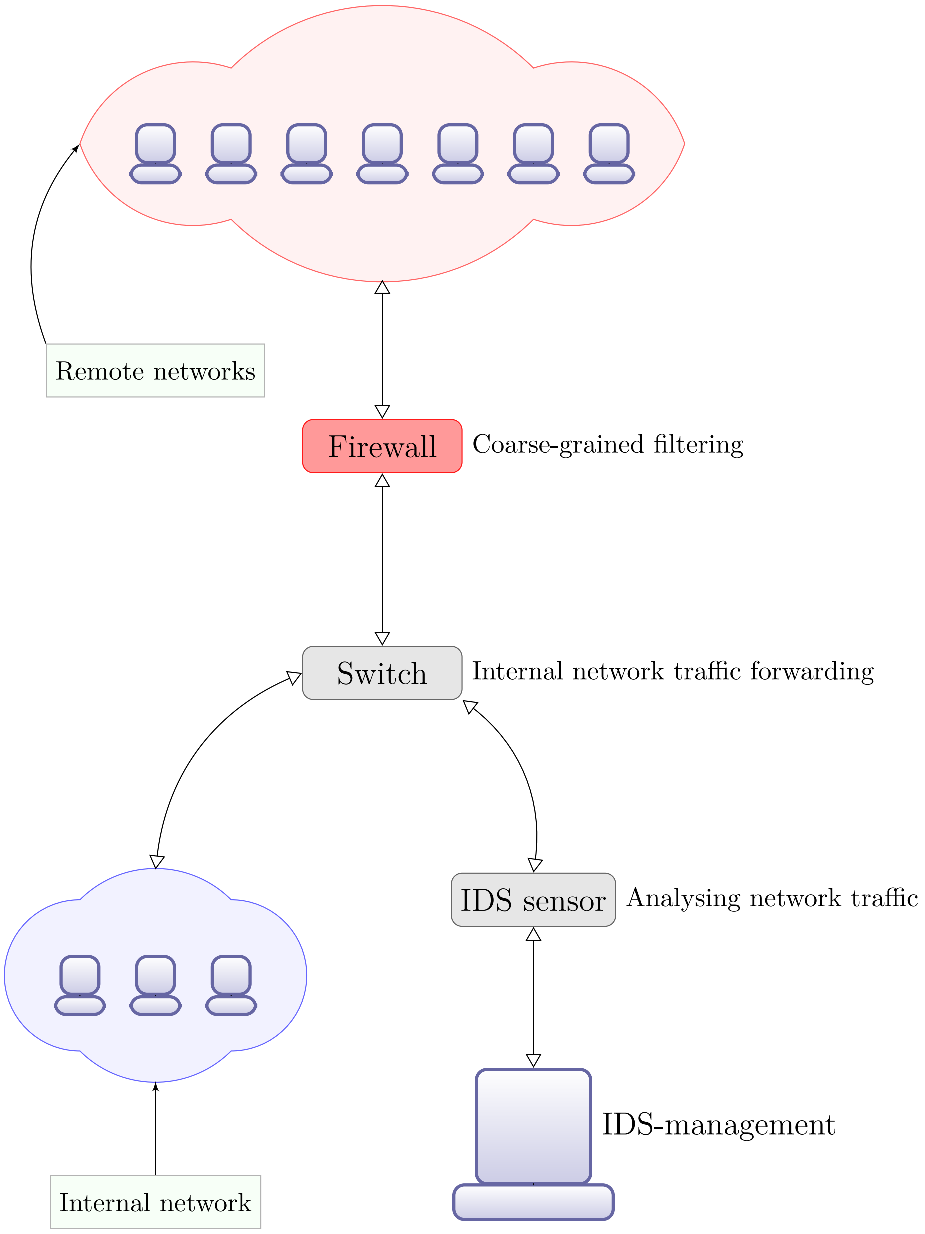


Figure 6.1: Example of an IDS architecture. The IDS sensor is connected to a *spanning* port on the switch. All traffic to the switch passes through the spanning port. The IDS is then able to analyse traffic to and from all sources. The IDS observes and analyses the traffic, but does nothing to actually stop it. The IDS management is a dedicated server to manage the IDS sensor.

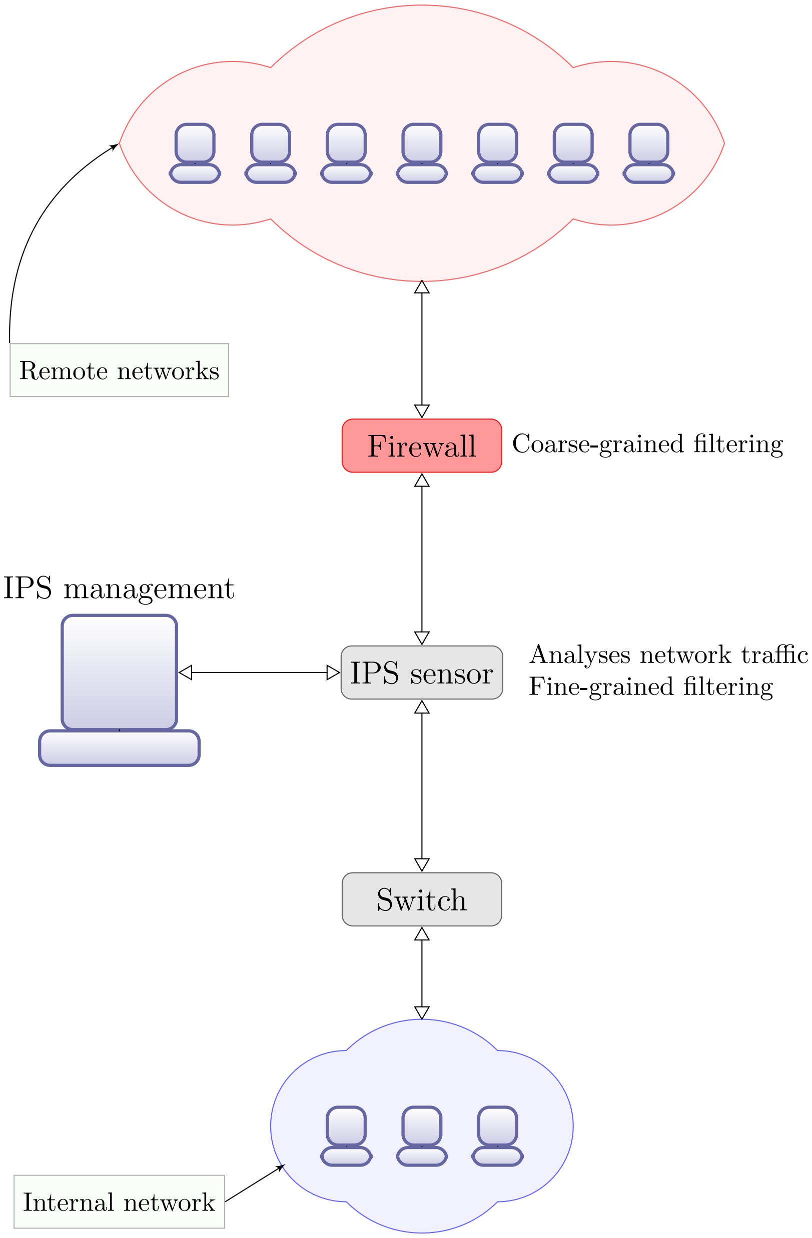


Figure 6.2: Example of an IPS architecture. The IPS sensor is deployed as a router so all remote traffic must pass through it before entering the internal network. This allows the device to deny any non-wanted traffic entrance to the internal network. The IPS management is a dedicated server to manage the IPS sensor.

when generating signatures, as a signature is meant primarily to *detect* traffic. For this reason, this thesis does not distinguish between the two different IDS/IPS approaches. An IDS solution suffices for the final system, so “IDSes” is the term used to refer to the appliances observing traffic and detecting suspicious behaviour in a network.

Pattern based signatures are used to filter out requested information from observed network data. Filters can be set to fields such as IP addresses, or specific protocol types. As an example, the pattern “find all network traffic between the dates 2009-01-01 and 2009-01-31 from source IP address `200.200.200.0` with destination port 53 using the UDP protocol” might be structured as an IDS signature. The syntax used for signatures varies for the different IDS solutions, but the idea is nevertheless the same. The signatures can thus be used as an advanced search mechanism on a huge quantity of data.

Utilising IDS signatures during dynamic analyses

As discussed in Chapter 2, malware is often dependent on Internet access for its malicious actions, either it is to signal home or multiply itself. Malware can spread to a large set of hosts, and the software struggles to avoid malware analysts to continue its malicious actions without interruption. To be able to repair or disable hosts that are infected, these hosts must be identified and localised. However, as seen in previous chapters, malware tries to camouflage its existence to remain undetected; and it is difficult for analysts to localise machines that are infected. This is particularly true for unknown and alien malware, as their functionality and behavioural patterns are not necessarily known. Analysts are for that reason not aware of *exactly what what to look for*, and estimate the prevalence of the outbreak of the malware is difficult. In the worst case, an entire network of hosts are infected without anyone being aware of the threat.

The functionality in a malware type tends to remain the same, even if the malware might frequently mutate its appearance. By utilising the power from existing IDS solutions, an analyst can observe an entire network of hosts. Combining this power, with the fact that malware rarely changes its *functionality*, one can gather vital characteristics from a malware sample’s network communication behaviour, and create an IDS signature. By using the generated signature in an IDS, analysts can find hosts that have the same (or very similar) communication patterns.

Consider a new malicious sample is executed in a sandbox environment during a dynamic analysis. During the execution, the sample tries to initialise remote communication with a host using an IP address or a DNS host name. The sandbox catches the remote communication attempt, and key information about the connection is filtered out. Using this information, a signature is created to find similar events in any network observed by an IDS. In an IDS-observed network, the scale of the infection can then easily be found using the generated signature to filter out machines matching the same connection patterns. However, the process can be automated, which reduces the necessary human intervention.

CERT teams often use IDSes in their infrastructure, covering network spanning large geographic areas, and machines connected to vital networks. For CERT teams, a signature able to recognise possible infected hosts can help tracing down infected clients and eventually the source of infection to properly mitigate threats exposed by the malware.

Further development of sandbox solutions
Creating a new, or improving an already existing sandbox solution is feasible, but not an optimal approach. Instead, integrating best practice methods from existing sandbox solutions into NorCERT's malware analysis environments is a more suitable approach for this thesis.

Antivirus coverage testing during dynamic analysis
Forcing automatic virus checks on new files that appear during an analysis is a positive functionality for an analyst, and the work from the preliminary project for this thesis can be used as foundation for the approach. However, due to large differences in the analysts environments, this is not an optimal approach and is not selected as a problem to solve.

Automating signature creation for IDSes
IDSes play an important role for analysts, especially in CERT teams. The IDSes are monitoring large Internet segments that have large numbers of hosts. Working with NorCERT, which has numerous IDS sensors monitoring traffic across Norway, makes the signature creation an excellent choice as analysts more easily can get an overview over potential malware outbreaks with the generated signatures.

Table 6.1: Summary of discussed approaches.

6.3 Selecting an approach

The previous section presented three possible approaches, but due to the time constraints in this thesis, it is not feasible to implement everything from all of the approaches. The approaches and their corresponding weaknesses and strengths are already discussed, but Table 6.1 is shown as a summary. A prioritisation of the approaches is made, based on the following characteristics.

- The approach is believed to reduce time spent during dynamic malware analyses,
- it is believed to be feasible with the time available,
- it is actionable,
- it is innovative, and
- it can be integrated into NorCERT's internal analysis system.

The first two approaches presented have issues that are presented together with the corresponding approach. The first approach is not particularly innovative, and the time required for realising a usable sandbox application is significantly more than is available during this master's thesis. The second approach is innovative and actionable. However, it is likely the approach is not very usable as it must assume too much of an analyst's environment. Analyst's environments are subject to frequent changes.

The last approach, automating signature creation for intrusion detection systems is, due to the mentioned reasons, of overall best current interest. Additionally, the first approach is *partly implemented* together with the signature generation. However, an entirely new sandbox solution is *not* realised. Instead, selected functionality from existing sandbox solutions is integrated with the system, making it unnecessary for an analyst to run two distinct sandbox solutions when analysing a sample. The last approach, and parts from the first, is therefore taken further to a requirements elicitation of a new system. Combined, the approaches base a system that is "actionable", by loading the generated signature in an IDS, and can reduce time spent in a dynamic analysis. Also, the combined solution can be integrated into NorCERT's internal analysis system (`NAAS`) by avoiding a graphical user interface, and utilising available interfaces to `NAAS`. A discussion about how the integration with `NAAS` is feasible is deferred to 9.6 on page 112. The approach is innovative by the way signatures are generated. However, these design details are not decided until Chapter 8, and a discussion about the final system's uniqueness and innovative aspects is deferred to Section 10.2 on page 119.

Representatives from NorCERT have approved the suggestion of the approach.

6.4 Selecting products as base

This section covers systems that can be of use in an architecture for automatic generation of signatures. Implementing the entire signature generation system from scratch is neither necessary nor practically possible with the available time, so existing software and/or hardware are utilised. Below is a list of products evaluated and selected in this section. A particular software and/or hardware solution is chosen for each of the entries.

1. An IDS product must be selected. However, prior to this, one of three *types* of IDS solutions must be chosen, as they differ significantly in their setup and use. Section 6.4.1 describes and chooses an IDS type to use.
2. An IDS product is used to decide the *syntax* of the generated signatures. This is to decrease the amount of manual work required upon a successful signature generation. By generating the signatures with a predetermined syntax, the signature can be used by the selected IDS product without any necessary modifications. Section 6.4.2 on the next page lists IDS products and chooses one of them to use.
3. To securely run a sample, an environment able to execute malware and safely restore a clean environment is needed. A sandbox solution fits for the requirement, but the sandbox should be open source and freely available—which eliminates most of the available options. Section 6.4.4 on page 77 chooses a sandbox solution to use.

6.4.1 IDS Type

The different IDS types available are differentiated primarily by which level they observe events. The three most prominent types [Scarfone and Mell, 2007] are each briefly described below.

Host-based IDSes monitor a single host for suspicious activity. A "host" can be a desktop computer, server, router or any other device where log files, processes, files and/or changes in the configuration are present and can be observed for changes.

Network behaviour analysis (NBA) IDSes monitor and examine network traffic to identify threats generating *unusual* traffic flow. NBA technologies primarily detects large deviations from the normally excepted behaviour, and the configuration is updated mostly in a automatic dynamic matter. Observing the network during a time period where there are no infections or attacks, might be used as a basis for a normal state in the network. Each deviation from this normal state might trigger an alarm. NBA IDSes offer strong detection capabilities for certain threats such as distributed denial of service attacks (DDoS), but their signature based detection capabilities are very limited.

Network based IDSes monitor and analyse network traffic in a network segment to identify suspicious activity. Network based IDSes are similar to NBA IDSes, but they are more focused on single events. Additionally, using signatures in network based IDSes are easier to accomplish than in NBA IDSes. NBA IDSes usually cover a larger set of hosts concurrently than network based IDSes do, and therefore work on a more overall and course grained level than network based IDSes do.

NorCERT's infrastructure is utilising IDSes that monitor a large set of hosts, and host-based IDSes are for this reason not particularly useful. Since NBA IDSes by design have limited signature based search functionality, it is not chosen as the IDS type either. In consequence, the syntax in the signatures is structured for a network based IDS product.

6.4.2 IDS Product

How the signature files are structured depends on the actual IDS product used; there are no standardised way of writing IDS signatures [Kreibich and Crowcroft, 2004], although there have been attempts of deploying a standard using XML [Cansian et al., 2002]. No details regarding NorCERT's IDS system(s) are publicly available, and their implemented IDS product(s) remains unknown. For this reason, exactly which product(s) NorCERT uses are not taken into account when choosing a particular product. The final implemented system depends on the selected IDS product by its specified syntax rules. However, the system is made to allow switching output syntax to suit another IDS product relatively easy. Therefore, the signature generator can be changed to reflect another IDS product without too much effort. The selection criteria for a product are not that strict, and are based on the following properties. Each criteria is labelled for a referring purpose, and named as "Product Selection Criterion" (PSC) More concrete and specific requirements are elicited in Chapter 7.

PSC.01 - Open source Using open source software makes it possible to change or add functionality wherever needed. By supporting add-ons to be plugged into the core, adding functionality to an open source software piece is also relatively easy [Lerner and Tirole, 2002]. Open source software is usually free of cost, which is highly preferable for software in this thesis. The open source communities are strong, and assistance and guides are often available by the use of forums, web sites and live chat rooms. Additionally, encouraging use of open source software causes a competition for the commercial solutions which are often widely used in a monopolised market.

PSC.02 - Deployable without any hardware elements Some IDSes require specialised hardware elements in their deployment, and buying such (expensive) hardware elements is not an option. Thus, IDS products requiring hardware elements other than one, or possibly a couple of working computers with a network interface must be avoided.

PSC.03 - Network based From the conclusion in Section 6.4.1 on page 73, the chosen IDS product should be network based.

PSC.04 - Widespread and well known To more easily get assistance if any problems arise, a well known product with a solid user community is preferable. This can indicate of how thoroughly tested the product is, and also how well it is to detect intrusions.

The amount of available IDS products spans a large list, making a product selection non-trivial. However, due to criterion **PSC.03**, only network based products are relevant; all host based and network behaviour analysis IDSes are excluded. Table 6.2 on page 78 lists a set of network based IDS products, and even if **PSC.03** already excluded many products, the amount of products is still significant in size. Most of the products are commercially licensed and too expensive for this thesis, sorting out pretty much the rest the products. Additionally, the products in the list require specialised hardware equipment from the respective vendor, making them even more expensive. Few products are open source software, making most of the products fail concerning criteria **PSC.01** and **PSC.02**. Products `Bro` by developer Vern Paxson from Lawrence Berkeley National Laboratory, and `Snort` by the firm Sourcefire are the two most suitable products regarding the first three selection criteria and are taken further for a final selection.

`Snort` and `Bro` are similar systems. They operate on a normal computer system that has a network interface, so their environment does not need any necessary specialised hardware elements. Both products were designed to be extensible systems, and to be used in high speed network environments [Paxson, 1998; Roesch, 1999]. Their essential functionality is capturing network packets and do content matching based on signatures to detect network intrusions. While `Snort` is primarily signature based and looks for specific traffic content, `Bro` analyses network traffic at a higher level of abstraction. `Bro` records the network history and tries to understand the context of the traffic. The signatures used in `Bro` have the potential to be more advanced than `Snort` signatures are. `Snort` signatures are usually a one-liner, contrasting the long `Bro` signatures. Developers of `Bro` have gone as far as writing a separate scripting language to use when writing signatures. Sourcefire, the team behind `Snort`, asserts their flagship product to be the de-facto standard in intrusion detection systems according to their web site[3]. Millions of downloads are registered, and Sourcefire has received a multiple of high value awards and is ranked among the top quality systems available [Rehman, 2003]. `Bro` that started out as a research project, is not that widespread. This has not necessarily anything to do

[3] See `http://www.sourcefire.com/company/` for more information about Sourcefire.

with its quality, but rather a handful of other factors. `Snort` is GPL licensed for users wanting to be on their own, but Sourcefire also offers network security services for the commercial sector. The `Bro` community supplies code, documentation and assistance, but the product has still strong academic roots with non-profit ideas. This could be one of many reasons `Bro` is less widespread than Sourcefire's `Snort`, which team has more resources available and is financially driven.

`Bro` has without doubt more extensive capabilities when creating signatures, but the automatically generated signatures in the final system are not utilising such advanced functionality. To generate a signature, only standard fields (IP address and port numbers) are needed. This is chosen to make the system independent to the malware's content; the malware can mutate and change its data payloads, but still be detected by the same IDS signature. Not using data content as basis for the signatures also avoids unnecessary complexity in the final system. `Snort`'s simplicity therefore surpasses the advanced signature methods available in the `Bro` scripting language, and is sufficient for the final system. Both products are considered to fulfil **PSC.04**. However, due to `Snort`'s dedicated signature based design and its simplicity of writing new signatures [Roesch, 1999] compared to `Bro`, it is chosen as the IDS product to determine the syntax for automatically generated signatures.

6.4.3 Structuring Snort signatures

`Snort` uses a simple and powerful descriptive language for its signatures. They are usually not exceeding one "readable line" of length, but will in some cases expand to multiple lines for fine grained filters. The "header" part of the signature contains the following fields[4].

- Transmission protocol: "tcp", "udp", "icmp" or "ip". The first three signifies one particular protocol, and the latter all of them.
- Source and destination address: "any", one particular IP address, or a network of IP addresses using a *netmask* as routing prefix. As an example, "`10.10.10.10`" signifies one specific IP, while "`10.10.10.0/255.255.255.0`" (or a shorter notation "`10.10.10.0/24`") means a subnetwork with the netmask `255.255.255.0`. Using the bitwise "`AND`" operation on an IP address and a netmask, results in the network *destination* address. So a packet having the source IP address "`10.10.10.65`", its network destination is as shown in Equation (6.1). Note the conversion of the IP addresses to binary form to more easily apply the `AND` operation.

$$\begin{aligned} &00001010.00001010.00001010.01000001\\ \text{AND }&11111111.11111111.11111111.00000000\\ &=00001010.00001010.00001010.00000000\\ &=10.10.10.0 \end{aligned} \tag{6.1}$$

[4] How to write `Snort` signatures can be studied more in detail with the online help documents found at `http://www.snort.org/docs/writing_rules/chap2.html` (accessed 2009-04-22).

All IP addresses from the IP address “10.10.10.1” to “10.10.10.255” belongs to the particular network shown in Equation (6.1), and will therefore potentially trigger alerts in signatures specifying source address as “10.10.10.0/24”.

- Source and destination port: can be a specific port, a range using “:” as a separator or all ports using “any”. An example of the first 80 ports is “1:80”, or just “:80”.
- Action, that is the job to perform if the signature is triggered. The relevant options are “log” and “alert”, where the first one writes to a log file and the latter also signals an alert to the IDS.

In addition to the signature header, Snort gives the possibility of tuning signatures using additional *options*. The options make it possible to match data traffic content with the signature, check if the packet has a particular TCP sequence number, and more. The example shown below logs all traffic to the 10.10.10.0/24 subnet having a source port above 1024, destination port 55 and TCP flags set to SYN+FIN. Upon triggering the rule, “SYN-FIN packet” is written in the log file in addition to the current time, source and destination data. The rule is useless, and is just stated as a simple example for this section.

```
log any 1024: -> 10.10.10.0/24 55 (flags: SF; msg: "SYN-FIN
    packet";)
```

6.4.4 Sandbox product

Prior to selecting a particular sandbox solution, a set of selection criteria are determined. The criteria are shown below, each labelled as “sandbox selection criteria X” (**SSC.X**) for a referring purpose.

SSC.01 - Sample execution The sandbox shall be able to run samples files, and it must be possible to fetch any results from the execution.

SSC.02 - Clean up The environment where the sandbox runs in shall be cleaned up after executing the sample, preferably with least amount of effort possible.

SSC.03 - Observing network traffic The sandbox shall either support observing network traffic on its own, or be able to route the traffic through an interface that can. The latter is preferred, as malware can cause irregularities in the sandbox and deny service to, or interfere with the local network observer.

SSC.04 - Open source The sandbox should have an open source license so any necessary change more easily can be carried out.

Most of the sandbox solutions, including the ones discussed in Section 4.1.5 on page 45, cover both **SSC.01** and **SSC.02**. Some of them also cover **SSC.03**. However, Zero Wine is the only known solution that covers **SSC.04**. An open source license is vital for this thesis, as modifying the code base is required to develop the final system. Zero Wine does not, however, cover **SSC.02** and **SSC.03**,

Product name	**PSC.01**	**PSC.02**	Note
`Attack Mitigator`	⊗	⊗	Commercial product
`Bro`	√	√	Open source software. Free of charge.
`Cisco IPS`	⊗	⊗	Commercial product
`Cyclops`	⊗	√	Commercial product
`DefensePro`	⊗	⊗	Commercial product
`Dragon`	⊗	⊗	Commercial product
`Juniper Networks IDP`	⊗	⊗	Commercial product
`IntruShield`	⊗	⊗	Commercial product
`iPolicy`	⊗	⊗	Commercial product
`IPS-1` (Former `Sentivist`)	⊗	⊗	Commercial product
`Proventia`	⊗	⊗	Commercial product
`SecureNet`	⊗	⊗	Commercial product
`Snort`	√	√	Open source software. Free of charge.
`Sourcefire`	⊗	⊗	Commercial version of `Snort` with more functionality. The same company has developed both systems.
`StoneGate`	⊗	⊗	Commercial product
`Strata Guard`	⊗	√	Commercial product
`UnityOne`	⊗	⊗	Commercial product

Table 6.2: List of network based IDS products. Column two and three list fulfilment of **PSC.01** and **PSC.02**, respectively. Each item in the list fulfils **PSC.03**. The list is based on data from National Institute of Standards and Technology [Scarfone and Mell, 2007].

but as Zero Wine is an open source system, these limitations are possible to get around using little effort. How to overcome these limitations are discussed in Section 8.3 on page 95. Consequently, without further discussions, Zero Wine is selected as the sandbox solution, and is studied in detail in the following section.

6.5 Zero Wine in detail

Zero Wine is previously mentioned in Section 4.3.4 on page 47, but is studied more in detail in this section. Zero Wine is a research product made to analyse malware in a vaguely protected sandbox environment. Zero Wine runs on a normal Linux based host and depends upon a translation layer called Wine to run samples. The translation layer makes it possible to execute Windows programs through Linux (and similar) operating systems. Wine is therefore capable of running PE-structured files, and malware is usually structured in this form. The PE structure is mentioned in Section 3.1.1 on page 35.

Since Wine behaves as a Windows operating system, it operates as a *light* version of a virtual environment. More powerful virtual machine solutions use dedicated techniques to isolate operations inside the virtual environment, while Wine is not designed for execution of malware. Therefore, Wine's protection level cannot be matched with the commercial sandbox solutions discussed in Section 4.3 on page 47.

Wine is not meant as a operating system replica, but merely to execute software written for Windows operating systems in Linux based systems. Wine is able to do so by simulating the Windows core such as dynamic link library (DLL) files, registry and file system. It is modularly designed, meaning changes in the Windows core (usually DLL modifications or additions) can be added to Wine when needed.

Zero Wine depends on Wine for sample executions, but is an appliance written purely in Python. It ships with both a preconfigured environment ready without any necessary configuration, and as modifiable source code if that is needed. The preconfigured release of Zero Wine uses a Debian based image, making it possible to launch Zero Wine as a *QEmu* based virtual machine to utilise the application directly without any configurations. QEmu is an open source machine emulator that Zero Wine utilises to allow users to quickly try out the functionality. Zero Wine is shipped with a simple web server that is automatically started from the QEmu image, and a web based interface that communicates with the scripts handling the analyses. The web interface supports one main interaction from the user, which is uploading of sample files. After receiving a sample, Zero Wine initiates its magic by running the sample under Wine with debugging functionality enabled. While the sample runs in the background, API calls are parsed from the debug output and is returned to Zero Wine.

Using GNU strings, non-binary content in the sample file is collected and returned. Using a third party Python module, Zero Wine is also able to scan for file headers and information about used packers to compress the file. Lastly, Zero Wine looks for suspicious anti-debugging methods, as explained in Section 4.7.2 on page 51, and is able to dump a running binary if needed. Dumping a running binary enables the analyst to fetch the sample from memory, which is often in an unpacked and unencrypted form.

What is especially interesting with the Zero Wine project is its simplicity, but yet powerful functionality. It is in version "0.0.0.2" at the time of writing

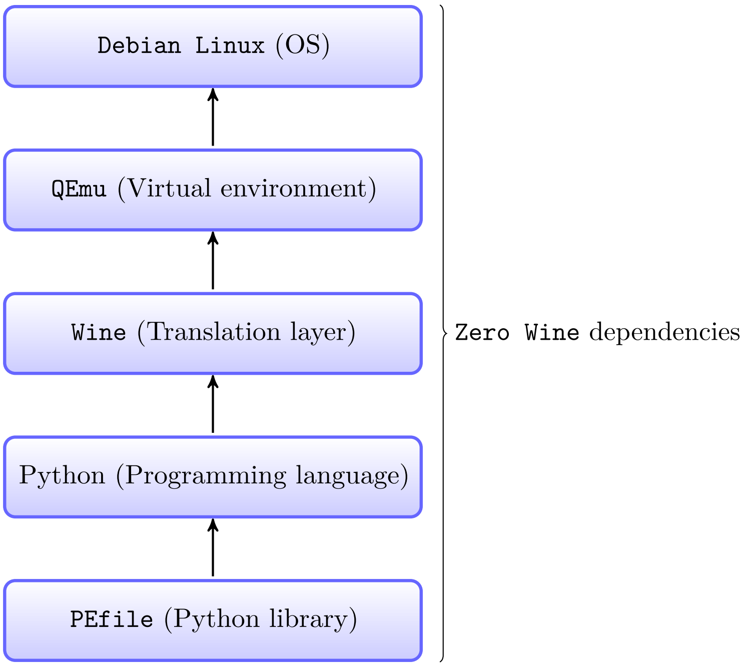


Figure 6.3: Dependency tree for a unmodified version of Zero Wine. The software depends on PEfile, which again depends on Python. To be able to run samples, it depends on Wine and so on.

and is currently far from a full-blown sandbox capable of what the commercial sandbox variants are. Zero Wine is still a good example of what it is possible to achieve using relatively simple additions to existing software solutions. Zero Wine is released under the GPL license, making it an excellent choice for further studies and improvements since the source code is modifiable and redistributable. The software depends on a handful of different tools, applications and solutions to work, a property that at the same time is positive and negative. On the good side, using already existing solutions decrease the necessary code base, and abstract away tasks to other solutions or tools. On the other hand, having many different dependencies make the deployment more cumbersome, and make it more difficult for a user to quickly start using Zero Wine. The developer has, however, solved this problem by the ready-to-use QEmu image. The dependency tree for Zero Wine is shown in Figure 6.3.

6.5.1 Program flow and structure of Zero Wine's automated analysis

Zero Wine's program flow can be deduced by studying the source code shipped with the software. Starting from the initial step, just after downloading Zero Wine, a script launching a QEmu virtual machine with a Debian based Linux operating system must be executed. The script contains one line only, merely the arguments QEmu requires to start. They are not important in this context, and are not explained further. After the QEmu image is started, a virtual QEmu machine is running in the background. A web server is automatically started from the virtual machine, and serves a simple HTML upload form. The form forwards uploaded files to the Python script "`upload.py`", which handles the analysis of the file. The analysis is using two other library files, namely "`libmalware.py`" and "`libutils.py`". These library files contain the main analysis logic. The scripts depend on a third party Python script able to parse information from PE based files, named `PEfile`, which also ships with `Zero Wine`. After the analysis is complete, the Python scripts returns results back to the web server, which presents them to the user. A simplified sequence diagram displaying the program flow is shown in Figure 6.4 on the following page, but one of the library files are omitted as it contains methods to modify the appearance of the HTML web page only. The flow shown in the figure assumes a file has been uploaded using the HTML upload forms. Method calls required to understand the basic program flow are shown, but the rest is skipped.

6.6 Summary of selections

This section briefly summaries the conclusions made in this chapter. The selected approaches are marked in *emphasised* text.

The selected main approach solved by the final system realisation is to *automatically generate signatures for intrusion detection systems (IDSes).* The signatures are created by observing a sample's network traffic while it is running. The malware samples are executed in the open source sandbox solution *`Zero Wine`*, and signatures are generated according to the *`Snort`* syntax. `Snort` is an open source *network based* IDS, meant to observe network traffic to and from a multitude of hosts in a network.

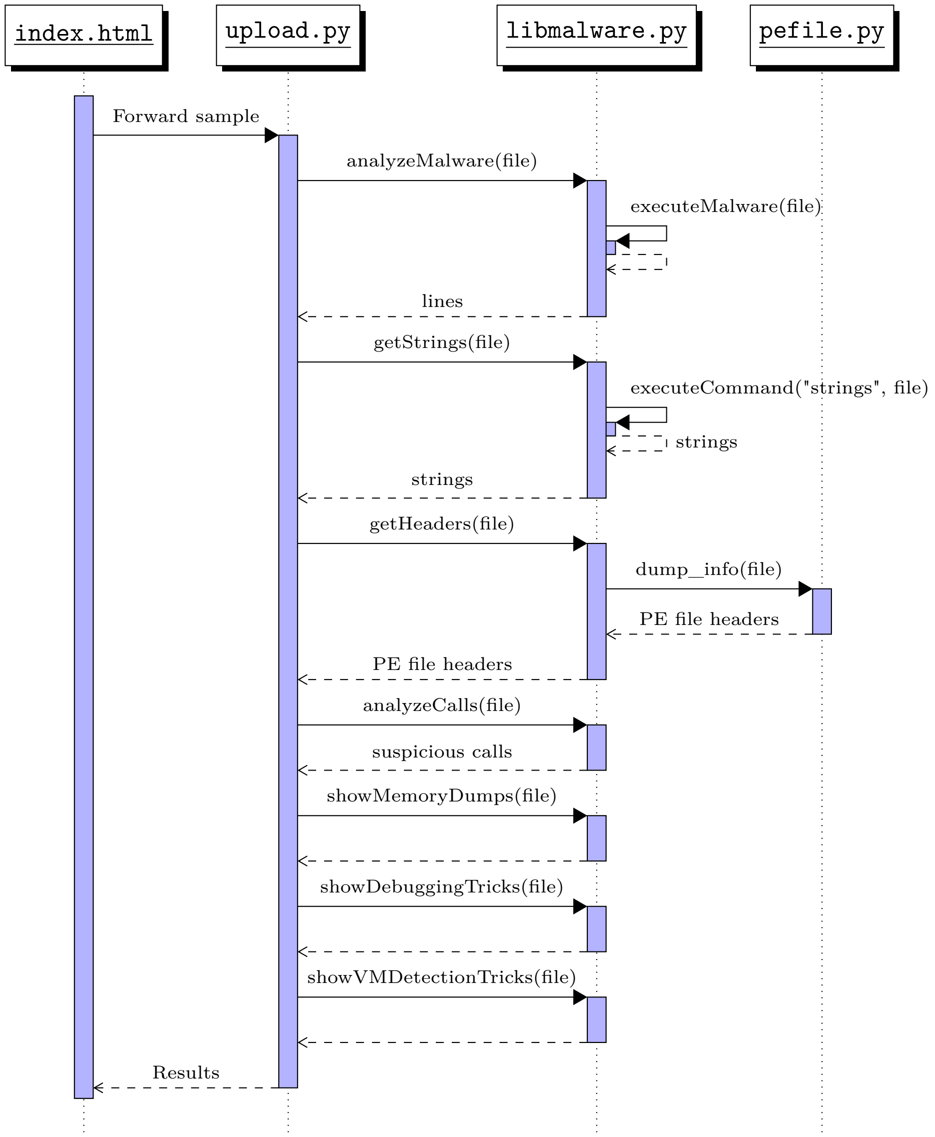


Figure 6.4: UML Sequence diagram displaying program flow from unmodified `Zero Wine`.

Chapter 7

Requirements Specification

This chapter focuses on elicitation, analysis and specification of the requirements for the implementation part of this thesis. A top-down approach to the requirements analysis is presented. This approach refers to a development process utilising a set of selected existing software solutions. The overall main goal is to offer new important functionality when using these solutions.

Within the requirements specification phase it is quite important to describe the application's qualities that has to be assured, rather than describing how such qualities will be designed or implemented ("what" versus "how"). Functionalities that the system has to supply are defined in this part of the thesis. This is done without indicating a specific architecture or a particular algorithm to adopt in the implemented solution. Software requirements analysis is one of the key elements of the development work flow as the other stages are based upon it, such as software design.

The chapter starts by deciding a set of high level requirements based on conclusions and selections found in previous chapters, and from the problem description. Next, the chapter presents a use case analysis that eventually leads to a collection of formal requirements for the system. The chapter ends with a mapping of the requirements to the high level requirements.

7.1 High level requirements

Before analysing requirements by textual use cases and corresponding diagrams, an initial list of more general requirements is defined and shown below. The entries in the list are each named for referring purposes. Their detail level is low, but more concrete requirements is elicited in Section 7.3 on page 88. These high-level requirements are derived from the previous chapter, partly combined with the result goals from Section 1.2 on page 3. Remember the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in RFC 2119 [Bradner, 1997].

HR.01 The system shall generate signatures for the `Snort` intrusion detection system based on a sample running in a dynamic analysis.

HR.02 Once the system is set up and configured, the only manual effort required by an analyst should be to submit the sample file to the system. The rest of the tasks should be automated.

HR.03 The system must be *secure*. Executed malware shall not be able to escape the system, and the environment shall be cleaned for all possible infections after an execution.

HR.04 The software should be kept open source and freely available for other people to use and modify.

HR.05 The software shall be documented so possible contributors more easily can familiarise with the code.

7.2 Use case analysis

The purpose of this section is to elicit important *use cases* [Fowler, 2003]. Use cases are used to capture functional requirements of the final system. The use cases are described in a textual form underlining the involved actors, the goal for the use case, its priority, entry conditions, and basic flow of events. The goal describes what the actor wants to achieve with the current use case. Entry conditions are used to imply requirements for the use case to happen. Use cases are specified only if the condition is not considered trivial. Use cases assists in at least three areas, each listed below.

- Use cases can assist to the process of deducing an initial set of requirements.
- Creating and analysing use cases can help determining overseen requirements.
- The simple notation assists the communication with supervisors and contact persons at NorCERT.

The following two use cases are each supplied with a use case diagram to graphically display the textual representation. Together with the high level requirements, they are considered as the starting point for the functional and non-functional requirements specification. Investigating the use cases and their relationships make it possible to more easily understand which are the most important services and functionalities the system shall supply. Implementing the described operations, moreover, is the best way to fulfil the expressed requirements.

7.2.1 Use case: Generate Snort signature

An analysis is performed using a sample file. The goal of the analysis is to create an IDS signature, which can be used in `Snort` intrusion detection systems. Utilising the generated signature makes the analyst able to get an overview over the clients that are infected with the particular sample. Additionally, hosts infected with the sample's mutated variants are also found. The use case is shown in Table 7.1 with a corresponding use case diagram shown in Figure 7.1.

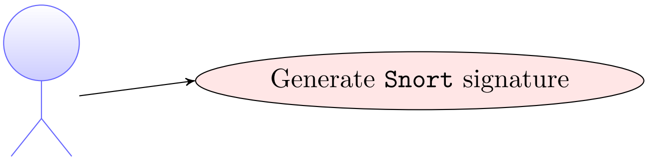


Figure 7.1: Use case diagram for "Generate `Snort` signature".

Name	Generate `Snort` signature
Actor	Analyst
Goal	The actor wants to automatically generate a signature that can be used in the `Snort` IDS. The signature looks for infected clients of a distinct malware type.
Priority	High
Entry conditions	(a) A `Snort` IDS is installed, and is actively covering a set of nodes, (b) a sample file is available, and (c) a command line shell is available (through terminal, secure shell (SSH) or similar).
Flow of events	1. The actor starts the command line script with a sample file and waits while the dynamic analysis is running. 2. The system presents a `Snort` signature file created by observing network traffic that originates from the sample.

Table 7.1: Use case for "Generate `Snort` signature".

7.2.2 Gain information about sample

An analysis is performed using a sample file. The actor's intentions are to automate the following tasks and return the results back. The corresponding use case is shown in Table 7.2 on the next page with a use case diagram shown in Figure 7.2.

- Let the system parse the sample file for any packer technologies. This assists the actor in the following two areas.
  - The actor uses the information as an indication of whether the sample is malicious or not, as malware often use packers to obfuscate and camouflage code.
  - The actor saves the information for a later phase, and then utilises the knowledge to more effectively run the analysis.
- Let the system fetch *suspicious* API calls made to the system. The API calls are used to get an overall view of how the sample operates, and whats its functions are. The types of calls considered "suspicious" are presented in Section 9.2.1 on page 102.
- Let the system fetch string data from the sample file. Valuable information is often found as plain text in binary files, and getting important information in an early analysis phase is cheaper than it is in the following phases.

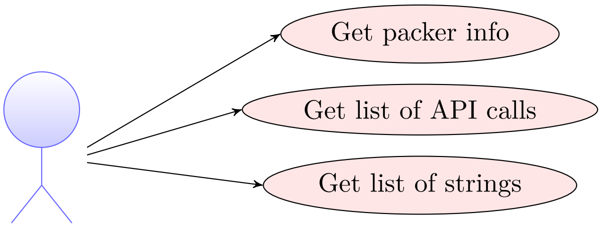


Figure 7.2: Use case diagram for "Gain information about sample".

Name	Gain information from sample
Actor	Analyst
Goal	The actor wants to get vital information about a sample: (1) packer technologies used to pack the sample, (2) important API calls made by the sample, (3) and its string data content.
Priority	Medium
Entry conditions	A sample file is available.
Flow of events	1. The actor starts the command line script with a sample file and waits while the dynamic analysis is running. The actor specifically supplies certain selected arguments to enable string scanning together with API call and information about used packer technologies. 2. Important API calls made by the sample, found packer technologies used in the sample and string content of the sample file is returned to the actor.

Table 7.2: Use case for “Gain information about sample”.

7.3 Overall list of requirements

This section contains the list of requirements derived from the use case analysis combined with the high level requirements found previously in the chapter. Each requirement has a brief description and a measurable property. Also, a priority that expresses the requirement's importance with respect to the others is included. The priorities are one of "High", "Medium", or "Low". A comment is included with each requirement to reason for why the corresponding requirement is priorities as it is.

All of the requirements are shown below in Table 7.3 below, and are classified in one of the two following groups.

The F group contains the *functional* requirements for the system, showing its behaviour and results. This group concretises what the system is supposed to do.

The NF group contains the *non-functional* requirements, showing how the system is supposed to be. Contrasting the functional requirements, the NF group does not list specific behaviour.

ID	F.01
Name	Parse out API calls made by the sample
Description	The system should execute (run) samples and monitor API calls made while they are running.
Measurement	Start the system using a sample file with a known set of API calls as input. Compare the list of API calls with the result from the system.
Priority	Medium
Comment	API calls may give an analyst clear indications of what a sample does, and can help deducing its functionality and threat level. Automatically parse suspicious API calls can definitely help reducing time spent during dynamic analyses, making the requirement important. The idea is already applied in other solutions—sandboxes in particular. However, if the API call parsing is implemented in the final system, the probability of running two distinct analyses decreases. For these reasons, the requirement is prioritised with a "Medium" level.
ID	F.02
Name	Automatically generate a `Snort` signature
Description	The system shall generate signatures to be used in the `Snort` IDS based on network traffic during execution of samples.
Measurement	Start the system using a sample file that contacts various hosts and compare the signature output with the hosts.
Priority	High
Comment	The IDS signature generation is the main goal of the system, and is believed to fulfil all characteristics shown in Section 6.3 on page 72. The requirement is, in consequence, of top priority.

Table 7.3: (continued)

ID	F.03
Name	Restore clean states after sample execution
Description	The system shall be *secure* and shall be automatically reverted to a clean state after each executed sample.
Measurement	Create a sample that changes a specified token or setting. Start the system using this sample and ensure the corresponding token/setting has changed back to the original form after the execution of the sample.
Priority	High
Comment	Each malicious sample analysed may infect the analytical environment. Using snapshots is a method to easily ensure the system is returned to a clean state. The requirement is prioritised high, since it is essential the system remains secure and avoids permanent infections between the analyses.
ID	F.04
Name	Fetch list of strings in sample
Description	The system may optionally fetch text data (strings) in samples.
Measurement	Create a sample containing a specific list of string data and start the system using this sample and ensure the results contains the expected strings.
Priority	Low
Comment	Text data may disclose information about a sample's behaviour, and in that way help deducing its functionality. The requirement helps assuring analysts not necessarily need two distinct solutions. However, finding text data is related to the surface scanning, and is possible to find using existing software. The requirement is therefore prioritised low.
ID	F.05
Name	Fetch information about executable packer from sample
Description	The system may optionally parse samples for executable packer technologies used to compress it.
Measurement	Start the system using a packed sample. Compare results from the system and the packer technology (or technologies) used.
Priority	Low
Comment	A sample compressed with an executable packer raises a warning signal. If which used packer type is found, the analyst may be able to manually unpack the sample and disclose its real content. However, the requirement covers techniques used in surface scanning and static analyses, and for that reason prioritised low.
ID	NF.01
Name	Open source
Description	The system should be released in open source form to allow the public to utilise and improve the code.

Table 7.3: (continued)

Measurement	Not applicable
Priority	Medium
Comment	Using an open source license makes it easier to improve software after its release by contributing to the code. Since the implementation is a proof of concept system, it is preferable that *anyone* is able to study the code and add functionality as needed. The requirement is prioritised with “Medium” importance.
ID	NF.02
Name	Modifiable system
Description	The system shall be sufficiently documented so modifications can be made without familiarise with the complete code base.
Measurement	Not applicable
Priority	High
Comment	A proof of concept system is assumed to be further developed, and familiarising with the code base is then needed. By documenting code, a developer not familiar with the code will more easily be able to understand what the software does. The requirement is therefore highly prioritised.
ID	NF.03
Name	Possible to follow `Zero Wine` upgrades
Description	It is recommended that it is manageable to upgrade code strongly related to `Zero Wine`. That is, when new releases of `Zero Wine` are made public, the new functionality should be possible to integrate in the system.
Measurement	Not applicable
Priority	Low
Comment	New `Zero Wine` upgrades may add useful functionality. However, the rest of the requirements are more important, prioritising this requirement with “low” importance.

Table 7.3: Requirements for the implementation part of the thesis.

7.4 Mapping of requirements

The high level requirements defined in Section 7.1 on page 83 are in this section mapped to the non-functional and functional requirements specified in the previous section. Table 7.4 on the facing page can be used as an overview.

Requirement **HR.01** maps to **F.02**. **HR.02** is covered mainly by **F.02**, but also **F.01** and **F.03** are partly covering the high level requirement due to its design details. The design specifications are described in the following chapter.

HR.03 maps directly to **F.03**, and **HR.04** directly to **NF.01**. The last high level requirement, **HR.05** is covered by non-functional requirements, **NF.02** and

High level requirement	Maps to
HR.01	**F.02**
HR.02	**F.02**, and partly **F.01** and **F.03**
HR.03	**F.03**
HR.04	**NF.01**
HR.05	**NF.02** and **NF.01**

Table 7.4: Mapping of high level requirements (**HR**) to functional **F** and non-functional (**NF**) requirements.

NF.01. Two additional functional requirements and one non-functional requirement remains with no mapping to the high level requirements. The relevant requirements are **F.04**, **F.05** and **NF.03** which all include useful functionalities for an analyst, but they are prioritised lower due to the higher importance of other requirements.

Chapter 8

Design

Other than describing the system in terms of components and relations among them, this part of the thesis registers all the significant decisions about the design and the motivations that led to these decisions. It is vital to cover requirements defined in the previous chapter by the design proposed in this chapter, so no design choices conflicting the requirements are made. Doing so ensures the design does not prevent fulfilment of the requirements.

8.1 Modifying Zero Wine

The system utilises existing work from the `Zero Wine` code base, but modifies the software to be a standalone system part. The web interface that is originally shipped with `Zero Wine` is removed, since such an interface is meant primarily for human interaction—and has no use in an automated system. Instead, a command line interface is implemented in the final system. Such a design choice allows running the system with, or without, an external interface; making it easier to switch between automatic and manual control. The approach enables an easy integration with other arbitrary software solutions. This property allows the system to be integrated with NorCERT's internal system for handling malware samples.

`Zero Wine` does not support a snapshot capability, so to be able to revert to a clean state after each sample execution, a virtual machine is used as environment during execution of samples. `Zero Wine` ships with a `QEmu` image with snapshot capabilities, but `QEmu` lacks a *programming interface.* A programming interface is needed to communicate with the virtual machine easily. To communicate with a `QEmu` virtual machine, additional software must be used or implemented specifically for the purpose. `QEmu` is for this reason replaced with `VMware Server`. The preliminary project for this thesis showed `VMware Server` is able to automate virtual machine tasks using its programming interface, `VIX`. Consequently, `VIX` is not extensively described in this document, but knowing it is an interface to `VMware Server` is sufficient knowledge.

The system is split up in **two** main parts, one located on the virtual machine *guest* operating system and one on the *host* operating system where the virtual machine software (`VMware Server`) is running. The modified `Zero Wine` part is deployed in the *guest* environment, while the host contains a script to control the virtual machine using `VIX`. Results are aggregated and structured from the guest

OS. Using this approach together with VMware Server makes it possible to reuse existing virtual machine communication code from the preliminary project. That is the reasons the system is split, and the two parts (or modules) are as follows.

1. "Zero+One" is the modified Zero Wine code that handles sample execution and analyses events during the execution. The main task Zero+One does is generating IDS signatures, but also do other tasks common in an analysis. These tasks are described in Section 8.4 on page 96. Most of the analytical functionality from the original Zero Wine project is kept, but some unnecessary methods are removed and others added. The execution of malware in Wine does *by no means* guarantee a safe execution, and a cleanup most be done after each analysis to ensure a safe environment. Therefore, the modified system uses an operating system in a virtual machine environment that supports restoring clean system states. VMware Server has this functionality, and is already selected to be the virtual machine system. The first part (Zero) of the module's name derives from Zero Wine, and the additions to Zero Wine's original functionality are the reasons for the last part of the name (+One).

2. "vmcom lite", is a script that communicates with virtual machines. The full version, vmcom, was developed during this thesis' preliminary project. vmcom lite is a modified version of this program, without the extensive focus on antivirus scanning. Instead, it is a general version that can be used run arbitrary programs from a virtual machine. vmcom lite is the entry point of the system connecting to and executing the Zero+One program from the virtual machine. After Zero+One has completed its execution, vmcom lite receives aggregated and computed data results back. vmcom lite presents the results as-is (text based data), but interfaces using vmcom lite can present this data in any form.

To avoid complex and tangled control structures between the different parts, the vmcom lite part shall be stripped from analytical program logic, and let Zero+One do the analytical work.

8.2 Task automation

VMware Server supports task automation using a software component called "VIX" to communicate with the virtual machines. VIX is a high level API accessible from a variety of programming languages. The API is partially supported by VMware, and was extensively used in the preliminary project for this thesis [Krister, 2008]. The following tasks are automated on the virtual machine by vmcom lite, and are all supported using VIX.

- Copy the sample file to the virtual machine.
- Start the analysis of the sample from the virtual machine using the modified Zero Wine software (Zero+One).
- Return the analysis results from the virtual machine.
- Revert the virtual machine's system state to a previously stored and clean snapshot, reversing any change done by the sample.

8.3 Generating IDS signatures

By analysing network traffic generated by the sample while it is running, `Snort` signatures can be generated based on the observed traffic data—so an application monitoring the network device used by the samples is needed. The application should be configurable, meaning: (1) it should be easy to modify the application's *filter* mechanisms so the correct network traffic events are processed, and (2) as many events as possible that are considered uninteresting for the signature should be dropped. Traffic filtering is the process of sorting out interesting traffic and events. There are different approaches available to capture network traffic and filter out particular events, but the following network capture product selection criteria (NCPSC) are emphasised when selecting a solution.

NCPSC.01 - Modifiable The selected approach shall be easily *modified*, during development or in a later stage. What modifiable means in this context is *what* to filter out must be possible to change without an extensible amount of effort.

NCPSC.02 - Easy to set up The selected network capturing approach should not require too much effort before it is ready for use.

Since all network traffic from the guest virtual machine passes through the host operating system, it is possible to place the network listener in the host, as well as in the guest operating system. Available approaches are as follows.

Implement a new listener A new network listener can be programmed for either the Perl part that utilises `VIX`, or the modified Python `Zero Wine` code that performs the signature generation. Network listening is not directly analytical work, but is closest to `Zero+One`'s functionality. Therefore, the most relevant choice is to place it together with `Zero+One`. A problem using such an approach is due to operating systems' privileges levels, administrator privileges are needed to observe network traffic. Running malware with high privileges can be damaging to the environment as the malware can do anything an administrator of the system is allowed to do.

Use an existing software module It is possible to try localising an already existing open source network listener project and integrate it into the implemented. This is similar to the above approach with the same drawbacks, except a less necessity of custom made implementation.

Utilise third party external solutions A multitude of network listener applications running independently and standalone, exists. A negative side of this approach is yet another dependency is added to the system. However, the approach gives the possibility to keep the malware execution code free from any network listening code, and simply instead parse results from the third party application. A third party application can safely run with administrator privileges as it does not even need to know about the malware execution—just log observed network traffic.

The approach of best current interest is using a third party application running with administrator privileges, simply because it is the safest of the three above options. The network monitoring application Wireshark is already discussed in Section 4.5.2 on page 49 and proves to be more than sufficient for this case, except for its graphical user interface (GUI) that is meant for human (non-automatic) interaction. However, Wireshark has a command line version named tshark which has the same functionality as the GUI version. tshark is more simple to use for automatic interactions, and is for that reason also in use in the final system. It supports a large variety of command line parameters to modify its filters, making it modifiable. No further selection is needed, as Wireshark/tshark supplies the needed functionality and the **NCPSC** selection criteria. The network observer that monitors data traffic is placed together with Zero+One in the guest operating system. The reason for this is to keep vmcom lite free from other functionality than virtual machine communication, and have all analytical functionality in Zero+One.

8.4 Other design choices

Even though generating an IDS signature is the main focus for the final system, three additional features are also mentioned in the requirements specification. The features are found in other software solutions, but are still relevant for this system to avoid the need for additional sandbox software when performing an IDS signature generation. The features are shown below, with a brief description regarding their corresponding design. The features are prioritised medium and low, so they are not on the same level of importance as the signature generation is. With the help from Zero Wine's code base, they are all relatively easily implemented and is still implemented in the final system.

Fetch API calls Wine is configured to output API calls made by programs executing in its environment, so suspicious calls can be filtered out while an analysis is running. Results are returned as plain text to the analyst. The text is *not* parsed in any way, simply filtered out and returned. "Interesting" calls are API calls that can be suspiciously abused, often seen in relation to malware. This includes file accesses and manipulation, remote connection attempts, common anti-debugging techniques and Windows registry changes.

Fetch list of strings Text data in the sample is fetched using the UNIX program *GNU strings*.

Get packer information Packers that are used to compress the sample are found using packer signature files (available online).

8.5 Choice of programming languages

There are two main modules of the system to consider when selecting programming languages, namely the VIX communication program vmcom lite, and the modified version of Zero Wine, Zero+One. Zero Wine is written completely in Python, and porting its entire code base is not an option due to time limitations. Thus, Python is chosen as the programming language to use when modifying Zero Wine into Zero+One.

vmcom lite can be written in any language supported by the VIX API, which is C, Perl, Python, Java and COM based languages. If Python was selected also

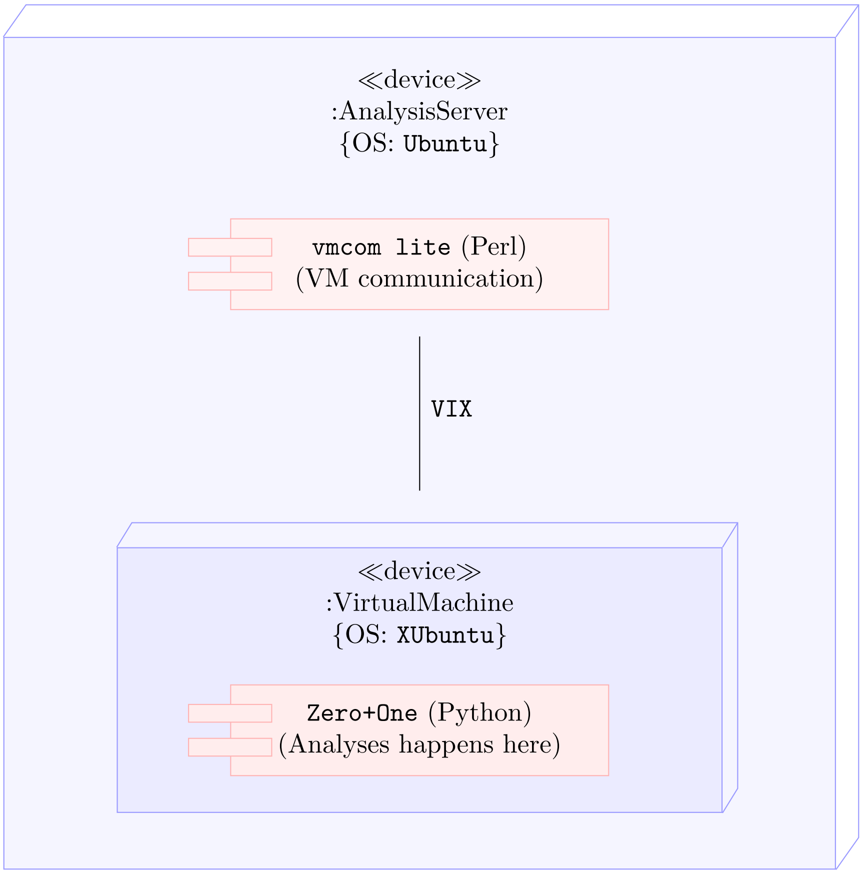


Figure 8.1: A possible system set up shown as a UML deployment diagram.

for this module, the same programming language would have been used for the complete system. However, Python is not supported by VMware and requires a third party wrapper to work. Using the wrapper adds another dependency. Additionally, Perl was used to communicate with the virtual machines in this thesis' preliminary project [Krister, 2008] so reuse of the Perl code is more suitable, and saves time. For these reasons, Perl is selected for the virtual machine communicator (`vmcom lite`) and Python for the modified `Zero Wine` code (`Zero+One`). Two good books for learning these languages are written by Schwartz et al. [2008] and Hetland [2005] for respectively Perl and Python.

8.6 System deployment overview

Figure 8.1 shows a UML deployment diagram of a possible system configuration. The figure is using *one* machine only, but if it is necessary to seal the analysis away, the `Zero+One` element can be placed in a separate physical machine. It is preferable the virtual machine's operating system is stripped from unnecessary background programs to avoid consuming the system's resources. `XUbuntu` is a

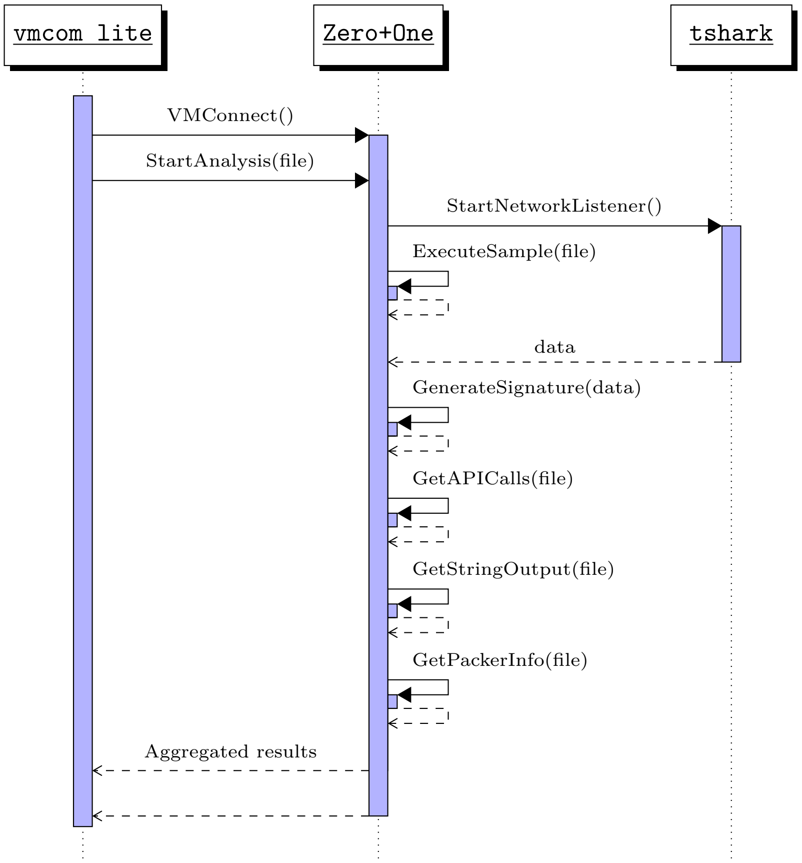


Figure 8.2: Sequence diagram showing program flow in the final system including virtual machine communication (`vmcom lite`), modified `Zero Wine` code (`Zero+One`) and network capturing using `tshark`.

lightweight version of `Ubuntu` and is, in this deployment example, selected for the virtual machine's operating system.

Figure 8.2 displays the program flow in the system, and gives an overview using a UML sequence diagram. The method names in the diagram are partly taken from `Zero Wine`'s existing code base, but is in use also in the final system.

8.7 Design limitations

There are negative aspects with the system design that are exposed during developing, and while using and administrating the final system. This section discusses known problems with the proposed design.

8.7.1 Dependencies

Using `Zero Wine` as base for the execution of samples leads to a large number of software dependencies. This might make system administration tedious when new versions of the software dependencies are released, and these new versions introduce changes to code taken for granted by the modified Python code. However, the software dependencies are relatively easy to install, and significantly decrease the necessary amount of own implementation to get the analysis working. Therefore, no attempts are made to reduce the amount of dependencies.

8.7.2 Follow Zero Wine upgrades

At the release of new revisions of `Zero Wine`, it is preferable to simply overwrite some code base, for example one or more files—and nothing more. Since the `Zero+One` part of the system uses modified methods found in `Zero Wine`, new versions of `Zero Wine` must be *manually* merged with the system. However, method names and content together with the overall program structure are kept as close to the original `Zero Wine` project as possible to simplify upcoming upgrade processes.

8.7.3 Detection of Wine

As with most new open source software, `Zero Wine` is not perfect. The source code is not documented, and there are in some cases hard to figure out what the code means to do. Luckily, the number of code lines is at a manageable level. Another more complex and serious flaw comes from `Zero Wine`'s design. The software utilises `Wine` as a "virtual machine" when executing the samples, but a program running in `Wine` can easily detect the environment—so a malware sample might detect `Wine` if it wants to. Two methods to detect the environment are shown below.

- Check the size of critical system files in the Windows operating system folder. The size in `Wine` is much smaller than a normal Windows operating system installation.
- `Wine` adds values to its internal Windows registry. There is a good indication of a present `Wine` environment if one of the following values are found.
  - `HKEY_CURRENT_USER\Software\Wine`
  - `HKEY_LOCAL_MACHINE\Software\Wine`

  Other registry checks are also most likely possible, as the registry used by `Wine` is minimal in size compared to a real Windows operating system.

`Wine` was never meant to safely run malware, so an additional security layer must be applied upon `Zero Wine` to safely execute samples. By using an already existing virtual machine solution, it is possible to automate a cleanup after each sample execution by reverting to a clean snapshot.

8.7.4 Multiple tools in an analysis

It is preferable that an analyst is not required to use multiple distinct sandbox solutions when performing a malware analysis. Using one single sandbox solution able to do perform all necessary tasks reduce time spent of managing different software solutions. The design does not include all tasks used in sandboxes, but a set of common functionality in addition to the unique signature generation.

Analysts might be forced to run an additional sandbox analysis if analytical tasks that are not included in the design are needed. However, the design is not trying to present a complete sandbox system, and the goal in this thesis is not to create a new sandbox. The design does not prevent additions of new functionality, making it possible to implement these in further work.

Chapter 9

Implementation

This chapter contains a description of how the system is realised, and explains the two main parts of the system; (1) vmcom lite, and (2) Zero+One. Additionally, a description of the code conventions used in the implementation is found in the chapter, together with code examples. The chapter flows by testing the final system, and continues with a selection of a software license to apply for the published system. The chapter continues with a presentation of limitations the system have, and finalises by looking into an integration with NorCERT's internal system for handling malware samples (NAAS).

This chapter explains how the system works, but the gory details of the system can be studied using the full source code listings, and its in-line documentation, in Appendix C.

9.1 Implementing vmcom lite

As explained in the previous chapter, the modified Zero Wine system runs in a virtual machine to ensure the environment is kept secure and free for infections. vmcom lite takes care of the communication with the virtual machine. This means the entry point of the system is vmcom lite, and vmcom lite does in particular forward the analysis asset (a sample) and starts the analysis by initiating Zero+One within the virtual machine. vmcom lite then waits until Zero+One has finished the analysis, and downloads the results from a text file located in the virtual machine. vmcom lite is a stripped version of the vmcom system implemented in this thesis preliminary project, and is documented in detail in the corresponding document [Krister, 2008].

9.2 Implementing Zero+One

Zero+One is the modified version of Zero Wine, but is closely linked to the tshark network observer so IDS signatures can be generated based on network traffic originating from malware samples. This sections covers how the Python code base of Zero+One and the network observer, tshark, works.

9.2.1 Python code base

Even if the Zero+One architectural design differs from Zero Wine's, existing code from Zero Wine is heavily utilised. Some of the functionality needed in Zero+One is already implemented in Zero Wine, and the corresponding code has been kept unchanged whenever possible, including method names and content. Doing so makes it easier for people already familiar with Zero Wine to more quickly understand Zero+One. Further improvements to Zero Wine can then also more easily be integrated with Zero+One. Web interface code is *not* separated from analytical code in Zero Wine, and was removed since it is unused in Zero+One. Zero+One performs a task depending on which startup argument given to the program. Zero+One can perform the following different tasks, with the argument shown in parenthesis.

- Parse suspicious API calls (`-e`, or `--execute`)
- Find packer technologies that are used to compress a sample (`-p`, or `--packers`)
- Fetch list of strings (`-s`, or `--strings`)
- Generate an IDS signature (`-e`, or `--execute`)
- Show a brief help screen (`-h`, or `--help`)

The "`-e`" argument is used to parse suspicious API calls *and* generate an IDS signature. Therefore, results from both API calls and the signature generation are returned when using the "`-e`" argument. This is a design choice to avoid running the sample twice if the analyst forgets to supply two distinct arguments. The different tasks are explained in this section.

Parsing API calls

By enabling Wine's debug functionality, one can parse out API calls generated by executing programs. A program is calling a large number of of API's during its execution, but some calls are more suspicious and consequently more important to look for than others. Table 9.1 on the next page shows the list of API calls filtered out by Zero+One, and a brief comment about why they are considered suspicious. There are a few additional API calls filtered out by Zero+One, but the entries in the table are the most important. For a more detailed description for the entries in the table, see Section 4.2 on page 46.

Finding which type of executable packers that are used to compress a sample

Zero Wine uses the Python library PEfile, which supports scanning a PE-based file for used packer technologies. PEfile uses file signatures to localise the packers, and the functionality is easily implemented in Zero+One. A list with 1832 signatures are shipped with the final system[1]. A drawback using this approach is when a sample is recursively packed, only the *first* layer of compression is found.

[1] Packer signatures are available from `http://www.peid.info/BobSoft/Downloads.html` (accessed 2009-04-20).

API call	Comment
`CreateMutex`	Mutex objects are created to ensure one, and only one, instance of a program is running concurrently.
`CopyFile`	Malware is frequently utilising the file system and registry.
`CreateFile`	Same as above.
`RegCreate`	Same as above.
`RegSet`	Same as above.
`CreateProcess`	Malware is frequently manipulating processes.
`getbyhostname`	Malware is often dependent on the Internet. "`getbyhostname`" performs DNS queries.

Table 9.1: API calls filtered out by `Zero+One` during execution of samples. See Section 4.2 on page 46 for a more detailed discussion about *why* the functionality from the API calls are suspicious.

Fetch list of strings

To fetch a list of strings from a sample, `Zero+One` simply runs the program "`GNU strings`", which is available in most Linux distributions, and returns the result.

Signature generation

An entirely new functionality in `Zero+One` is the IDS signature generation, which is based on activity observed while a sample is running. When a sample has finished its execution, data from a network analysis is gathered, aggregated, and parsed to generate *one* IDS signature which can be used in the `Snort` IDS system. The signature can give analysts overview over potential infections in a network. The final system executes the sample *once*, meaning the signature generation process has data for one single execution only, using a limited set of execution paths. This approach can lead to a signature that is subject to generate false positives and/or negatives. Therefore, based on a "granularity level" (explained in the next section) the analyst can create signatures covering a more fine– (or coarse–) grained network.

9.2.2 IDS signature quality

Section 6.4.3 on page 76 explains how a `Snort` signature is structured, but it is more than one single method to generate such a signature. This sections presents approaches to generating a signature and their corresponding problems and benefits. The section also reasons for and explains how the signature generation works in the final system.

To localise hosts in a network infected with particular malware with help from IDS signatures, the signature must consist of rules matching observed network data against *fixed* network data expected to see from the particular malware type. For example, if a sample always connects to a specific foreign IP address on a particular port, a signature can be generated based on the knowledge of the IP address and port. A perfect IDS signature finds *all* infected hosts the signature is meant for, and only those. In practice, this is truly difficult to achieve; there may be too few similarities in network traffic generated by malware to guarantee the traffic originates from an infected host, or the IDS signature is too narrow, allowing infected hosts to dodge the signature.

The final system is balancing these two issues, and is basing the signature generation purely on IP source and destination addresses and TCP/UDP source and destination ports. Traffic data content is *not* processed and ignored by the signatures. To localise infections in a network based on the generated signature, the system assumes the malware contacts the same hosts (or they belong to the same subnetwork if a more coarse-grained signature generation is performed). Observed network traffic during executions of samples are aggregated and collected after the sample has finished running (or is killed when a time limit is reached). The IP addresses and TCP/UDP ports found are then combined, and only unique values are filtered out, avoiding redundant and unnecessary data in the generated signatures.

Using such an approach makes the solution independent to malware's content. A malware sample can mutate and change, but as long as the malware communicates with the same hosts, the IDS signature is able to find the infections of a malware type and its variants. IDS systems are subject to consume large amounts of resources when matching traffic with signatures [Lee et al., 2002]. The approach used in the final system does not match the content of the data, leading to a significant decrease in the necessary processing power in the IDSes.

The signature generation can be set to a more coarse-grained output returning subnetworks of hosts instead of IP addresses. This is practical when samples are contacting machines located in the same networks. Consider a malware sample contacting the IP addresses `14.10.10.1`–`14.10.10.255` in somewhat random sequences. Instead of listing each of these IP addresses in the generated signature, an IDS-*granularity level* can be used when generating the signature. A granularity level is a number from 1 to 32, and indicates how broad the IDS signature output is. Granularity level "32" returns an IDS signature including all observed IP addresses (netmask of all bits set to "1"), while granularity level "24" returns IP addresses using a netmask having the 24 (of total 32) most significant bits set to "1". For the above example using granularity example "24", the subnetwork `14.10.10.0/24` is generated which covers IP addresses including and between `14.10.10.1` and `14.10.10.255`. See Section 6.4.3 on page 76 for more information, or use books by Tanenbaum and Stevens for a detailed overview of how IP subnets work [Stevens, 1994; Tanenbaum, 2002].

There are negative sides with the approach as well. The generated signature is subject to not find infected hosts when the malware is not contacting the same IP addresses as it did at the time the signature was generated. Different IP addresses can be contacted due to trigger based behaviour in the malware, or the malware has been modified in some way, and contacts different addresses. This problem is somewhat coped with by using host names instead of IP addresses, and is discussed more in detail in Section 10.3.2 on page 120.

If the signature has a granularity level too high, or malware is contacting random hosts, there is a large chance of introducing false positives by adding hosts that are obviously legitimate. For example, a malware sample is looking up "`www.google.com`" does not mean that the domain is necessarily used for malicious actions—but rather a simple check for Internet access. This limitation is discussed more in detail in Section 10.3.3 on page 121.

9.2.3 tshark

The `tshark` application is launched within the virtual machine operating system, and is constantly running as a background process to collect network traffic. `tshark` collects information from the IP protocol as well as TCP and UDP headers and content. The information includes the following relevant data.

- IP source and destination
- Transport protocol ports for IP source and destination
- Traffic content, both the textual and binary content

To generate an IDS signature, the final system filters out necessary elements from network traffic and ensures the output is structured so the signature generation process can be performed automatically.

Due to the possible complexity of traffic data and the difficult process of automatically mapping it to malicious actions, signatures generated from the final system is using the first two entries from the above list and excludes traffic content entirely. Further work suggests to include traffic data, but signature generation based on possible large quantities of highly dynamic and non-deterministic data content requires a project of its own. See Section 10.3.4 on page 121 for more information about the issue.

The command line to start `tshark` is shown below, logging the network interface "`eth0`". `tshark` supports a *capturing filter* to limit the amount of logged fields. However, in this case is *all* traffic data observed logged, including header and content. The log file's path is `/var/log/tshark`.

```
sudo tshark −i eth0 –w /var/log/tshark
```

Using a ***tshark** display filter* gives the possibility to filter out particular fields and discard the unnecessary. The below command line outputs IP addresses for source and destination, transport protocol ports and TCP flags, and is sufficient for the current implemented signature generation algorithm. If a TCP flag is absent in a packet, no data is printed in the output. Otherwise, boolean true (typed as "1") is written. Fields are separated with a semicolon (";"). For more information about the application, see `tshark`'s manual page found at `http://www.wireshark.org/docs/man-pages/tshark.html`.

```
tshark −r /var/log/tshark –T fields –E separator=';' \
−e ip.src \
−e tcp.srcport \
−e ip.dst \
−e tcp.dstport \
−e tcp.flags.ack \
−e tcp.flags.syn \
−e tcp.flags.fin \
−e tcp.flags.rst
```

```
"""
Finds the highest frequently used port in an array.  Returns the port as a
string.
"""
def findHighestFrequentlyUsedPort(ports):
    """Dictionary used to find frequencies"""
    portFrequentDict = {}

    """Simply return 'any' if 0 is found in the list"""
    if 0 in ports:
        return 'any'
    """Iterate the ports and place their count in the dictionary"""
    for port in ports:
        if str(port) not in portFrequentDict:
            portFrequentDict[str(port)] = 1
        else:
            portFrequentDict[str(port)] += 1

    """Find the port with the highest frequency and return it"""
    highestPortFreq = -1
    for k, v in portFrequentDict.iteritems():
        if v > highestPortFreq:
            highestPortFreq = k

    return highestPortFreq
```

Listing 9.1: Example from the `Zero+One` code base. The method shown finds the highest frequently used port in a list of ports found during network traffic observations.

9.3 Code conventions

The application is assumed to undergo changes by other software developers after its release, and it is therefore important to adhere to particular known coding conventions to make code maintenance easier. This section shows examples from the implemented system, but the entire source code is found in Appendix C.

9.3.1 Zero+One and its Python code base

Following `Zero Wine`'s code conventions are important to more easily allow an integration with `Zero Wine`'s new upcoming versions. However, the official Python coding conventions are used wherever reasonable[2]. `Zero Wine`'s original code base does not strictly follow the conventions, especially method names and mixing of tab/space indentations deviate from the rules. The method names are written in the "mixedCase" convention since `Zero Wine` has adopted this particular style, even if it is not recommended. The mixedCase convention starts each word in a function name in uppercase, except the first word that is in lowercase.

The mixing of tab/space indentations is assumed to be a fault by the `Zero`

[2] A Python code style guide can be found at `http://www.python.org/dev/peps/pep-0008/`.

```
# Copies a file from host operating system to guest operating system
# Requires: connection to a virtual machine (connect_to_vm)
#           an open virtual machine handle (open_vm)
#           a powered on virtual machine (power_on_vm)
#           vmware tools loaded (log_in_vm)
#           a user logged into the virtual machine guest OS (log_in_vm)
sub copy_file_to_guest {
   my($err, $vm_handle, $subject_file_location) = @_;

    $err = VMCopyFileFromHostToGuest($vm_handle,
        $subject_file_location,  # src name
        "/tmp/uploaded_file", # dest name
        0, # options
        VIX_INVALID_HANDLE); # propertyListHandle

    abort("VMCopyFileFromHostToGuest() failed", $err) if $err != VIX_OK;
}
```

Listing 9.2: Example from the `vmcom lite` code base. The subroutine copies an arbitrary file from the host OS to the guest OS using the `VIX` API.

`Wine` developer, and is simply fixed in `Zero+One` that uses four spaces for each indentation, and no tabs. `Zero Wine` is not documented directly in the code, but `Zero+One` is released with in-line code documentation following the Python docstrings conventions[3].

Code example

An example taken from the `Zero+One` code base is shown in Listing 9.1 on the facing page. The Python method is utilised by the IDS signature generation, and finds the highest frequently used port in a list of ports found by `tshark` during network traffic observations. The `tshark` logs are parsed prior to running the Python method.

9.3.2 vmcom lite and its Perl code base

The perlstyle conventions[4] were followed when developing the `VIX` communication in the preliminary project, and is for that reason used also for `vmcom lite`. The perlstyle differs from the Python conventions, and acts, in some cases, as a direct contrast. For example, Perl suggest to vertically align up corresponding code, while Python suggests the opposite. The conventions are used for the respective languages anyway, to adhere to a standardised way of coding—even when the conventions are in conflict with each other.

[3] The docstrings conventions can be found at `http://www.python.org/dev/peps/pep-0257/`.

[4] The perlstyle conventions can be found at `http://www.perlmonks.org/?node=perlstyle`.

Code example

An example taken from the `vmcom lite` code base is shown in Listing 9.2 on the previous page. The listing shows a subroutine for copying a file from the host operating system to the guest operating system using the `VIX` API. Prior to the file copy subroutine can be called, the VM must be opened in the `vmcom lite` program, be powered on and have a user logged in. The subroutine acts on behalf of the logged in user when performing the file copy operation.

9.4 Testing

This section contains documentation about the system tests' structure, how each test is performed, and the corresponding results. Testing is done in order to assure the system is functional and in according to requirements elicited in Chapter 7.

9.4.1 Test levels

There are a multitude of *levels* during software testing, and according to the V-model [Watkins, 2001, chapter 4], each of these levels link to one or more phases of the development process. Such a link means the development phase gives requirements to the test level. The V-model states the design phase in a development process leads to requirements for tests dealing with small units in the implemented code, and how these operate together [Copeland, 2004, chapter 1]. These units are usually what a programmer knows as "methods". The tests are known as *unit* and *integration* tests respectively, and are often written as automated test cases.

System testing is also derived from the design phase, but at a more general level. System tests compare design specifications against the actual realised system, and can be automated.

Usability testing and *acceptance testing* are also common test levels during software testing. They derive from the requirements specification (which is usually made prior to the design phase) and is meant to determine whether a system satisfies the elicited requirements. These tests is a manual process including the project owner, and are in consequence *not* automated.

As the implemented system is a proof of concept realisation, focus is held on *integration testing* and *system testing*, where each test specifies submodules of the system, and a successful test run indicates a submodule functioning as it should according to the design specifications.

9.4.2 Test method

"Black box" and "white box" testing are two different ways of testing a system. Black box testing does not consider any internal behaviour of the system, but assure the output of the system is as expected using a well-defined set of input. A mixture of white and black bow testing is performed when testing the final implemented system, but focus is held on testing the *functionality* of the system, not the inner details of the code.

Test name	Result
Testing communication between the system parts (`vmcom lite` and `Zero+One`)	**PASSED**
Testing parsing of API calls	**PASSED**
Testing the system's capability of finding packer technologies	**PASSED**
Testing IDS signature generation	**PASSED**

Table 9.2: Summarised test results.

9.4.3 Test plan

There are four tests, and each test runs separately. If small, easily repairable, bugs are discovered during testing, they will be fixed and the test is immediately restarted. Tests are based on executing submodules of the system, and comparing actual results with a set of expected results. The test still *passes* if the actual results, after potential repairs, are equal to the expected results. If the test still do not pass after repairs are applied, the test eventually *fails*.

The test includes the following five attributes. (1) the command line used to run the test, (2) what the expected results are, (3) which requirement or requirements are affected by the test, (4) a comment to the test run, and (5) entry conditions (not for the third test)

Table 9.2 can be used as an overview of the tests, and a summary of the final results.

Testing internal system communication

The purpose of the first test is to *communicate* with the virtual machine using `vmcom lite`. The test assumes the virtual machine is available and properly configured in `vmcom lite`. The test starts by initiating the analytical parameter -a FILE and supplies the password (shown as the scrambled text `*PASSWORD*`). During the test, attention is given to the virtual machine system to see if the specified file is uploaded at the configured location. Likewise, a simulated result file is created, which should be copied to the host system by `vmcom lite` during the test. It is important to ensure these files are not present before the analysis has started. The virtual machine state is reverted to a clean state, so the uploaded file should be unavailable (removed by the reversion process) after the file has finished running. The test run is shown in Table 9.3 on the next page.

Test name	Testing internal system communication
System module	`vmcom lite`
Entry conditions	(a) A virtual machine OS is set up in `vmcom lite`'s configuration part, referred to as "the VM OS" from now on, and (b) the simulated result file is not present neither in the VM OS nor in the host OS.
Test case	Run "`./vmcomlite.pl -a DUMMYFILE -p *PASSWORD*`" from the host OS.
Expected Results	(a) The dummy file (an arbitrary file) shall be uploaded to the VM OS, (b) the simulated result file shall be created in the VM OS, and (c) the simulated result file shall be removed from the VM OS after the analysis is complete.
Requirements tested	**F.03**
Result	***PASSED***
Comments	
Minor bugs were fixed in `vmcom lite`'s source code before the test ran successfully.	

Table 9.3: Testing internal system communication.

Testing parsing of API calls

From now on is the focus held on `Zero+One` and further tests cover functionality found in `Zero+One`. A real analysis and sample execution would use `vmcom lite` to communicate with and initiate requests to `Zero+One`, but since the previous test passed, communication between `vmcom lite` and the virtual machine is assumed to work.

`Zero+One` supports parsing of suspicious API call during execution of samples. This test executes the file used in the malware scenario in Chapter 5, and studies the results. The results are *not* compared to an expected list of suspicious API calls. Instead, the test is considered successful if API calls regarding the sample's network communication are found during the test. The test run is shown in Table 9.4 on the facing page.

Testing the system's capability of finding packer technologies

The analysis scenario successfully concluded which kind of packer technology was used to compress the sample. This test runs `Zero+One` and supplies the "find packer" argument. The result is compared to the analysis scenario result. The test run is shown in Table 9.5 on the next page.

Test name	Testing parsing of API calls
System module	`Zero+One`
Entry conditions	`tshark` is set to log network traffic, logging data to `/var/log/tshark` in the VM OS.
Test case	Run "`./kmanalyse.py -f asprox.exe -e`" from the VM OS.
Expected Results	API calls shall be returned, where some are concerning the sample's network communication found in the malware analysis scenario.
Requirements tested	**F.01**
Result	***PASSED***
Comments	
The test returned 283 suspicious API calls, where eight calls are related to network communication. The network communication is the same as what was found in the analysis scenario. These eight calls look up one IP address and three host names using the DNS query API. The output of these eight API calls is shown in Appendix D.	

Table 9.4: Testing parsing of API calls.

Test name	Testing the system's capability of finding packer technologies
System module	`Zero+One`
Entry conditions	None
Test case	Run "`./kmanalyse.py -f asprox.exe -p`" from the VM OS.
Expected Results	"`Microsoft Visual C++`" shall be found, the same result as found in the malware analysis scenario from Chapter 5.
Requirements tested	**F.05**
Result	***PASSED***, see comments.
Comments	
The test returned "`Armadillo v1.71`", which is not the same result found in the malware analysis scenario. "`Armadillo`" is a different packer type than "`Microsoft Visual C++`", but the two different packer signatures *are similar*, meaning a sample can match both of them. This information was not easy to find, but email correspondence with the contact point at the web page `offensivecomputing.net` indicating these results can be found in Appendix D. For this reason, the test passes—even due to the found packer name differs from the expected.	

Table 9.5: Testing the system's capability of finding packer technologies.

Testing IDS signature generation

The main purpose of this test is to ensure the system is able to generate an IDS signature. Additionally, generating an IDS signature requires the additional `tshark` component to work, so the test also covers this functionality.

The sample used in the malware analysis scenario in Chapter 5 is used for the test asset. From the scenario, a set of distinct hosts and IP addresses were found. These addresses are the expected content of the generated signature, and the test pass if they are found.

The `Snort` IDS is a system already tested thorough. Therefore, as long as the generated IDS signature follows the `Snort` signature syntax, the signature will not be tested by loading it into an IDS. The syntax is described in Section 6.4.3 on page 76. The reason for this selection is the amount of time required to establish an IDS with sufficient network coverage is extensive, and is considered unnecessary due to `Snort`'s proven functionality. The test run is shown in Table 9.6 on the next page.

9.5 Software licenses

In order to choose a license on the developed software, it is important to note that existing source code from `Zero Wine` has been used in the final system. `Zero Wine` is released under the Public License (GPL), which ensures free use of the software, and also *modifications* of it. Therefore, the final system, which is a modified version of `Zero Wine`, is released entirely under the GPL license. Using a GPL license forces the source code in the final system to be public, including further modifications of it. The GPL license can be studied at `http://www.gnu.org/licenses/gpl.html`.

9.6 Integration with NAAS

NorCERT's analysis repository, `NAAS`, is a web based system used for storing information about samples and analyses. `NAAS` is the internal system for handling malware samples, mentioned in the problem description. It is a collaborative tool and is used to share the information across the analysis team. The software is partly implemented by NorCERT's software developers, and standalone programs will, in time, be possible to use through the regular `NAAS` interface. An interface connecting `NAAS` to third party program is now being designed, and at its completion there will be possible to hook standalone programs onto NAAS, and run them as components in the `NAAS` system.

The standalone applications use command line interfaces for simplicity, are able to run without `NAAS`, and use `NAAS` as a graphical user interface component only. To integrate a standalone application, which in this example is the system for automatically generating IDS signatures, only a few settings in a Python class is needed. The Python class is one part of the `NAAS` design chosen by developers from NorCERT. To understand the link, consider Figure 9.1 on page 114 that shows how third party applications can be connected through `NAAS`, using a simplified UML class diagram. The figure is stripped from details, and shown to give an overview only. It displays the different operations (called "tools"), collected in an array instantiated by an "AnalysisClient" located together with the main NAAS

Test name	Testing IDS signature generation
System module	`Zero+One`
Entry conditions	`tshark` is running on the VM OS and logs network traffic to `/var/log/tshark`.
Test case	Run "`./kmanalyse.py -f asprox.exe -e`" from the VM OS.
Expected Results	An IDS signature to use in the `Snort` IDS shall be generated. The content of the signature shall match some, or all of the addresses found in the malware analysis scenario.
Requirements tested	**F.05**
Result	***PASSED***
Comments	

The test ran for 200 seconds before the system generated the following signature. "`log [$HOME_NET] any -> [58.65.233.17/32, $HOME_NET, 217.72.195.42/32, 87.248.113.14/32] 80`", which is following the `Snort` IDS signature syntax. When active in the `Snort` IDS, this signature look for connections *from* the home network on any port *to* a set of IP addresses observed by the network monitor during the sample execution. The IP address `217.72.195.42` maps to *two* of the five hosts found during the malware analysis scenario (results from the scenario are shown in Table 5.1 on page 60). These two hosts are "`ha-42.web.de`" and "`www.web.de`". The three remaining hosts are missing from the signature, but two new IP addresses are shown instead. One of the missing hosts has an expired IP mapping, meaning no data traffic can be sent to it, and thus not seen by the network observer and neither in the generated signature. The IP address "`87.248.113.14`" maps to the host name "`f1.us.www.vip.ird.yahoo.com`", owned by Yahoo. One of the missing IP addresses from the generated signature is the "`www.yahoo.com`" pointer, so there is probably a connection between the two different addresses. The second new IP address is "`58.65.233.17`", which originates in Hong Kong. These results indicate the sample *varies* its contact points from time to time, but the test is nevertheless successful as the generated signature strongly resemble the results from the analysis scenario.

Table 9.6: Testing IDS signature generation.

application. To generate an IDS signature, simply run the "generateSignatureTool" Python class from `NAAS` that takes care of the execution details. After the tool's execution, the AnalysisClient receives the results. The IDS signature is in this case the result. Upon receiving the signature, the AnalysisClient delivers it to the `NAAS` system for representation.

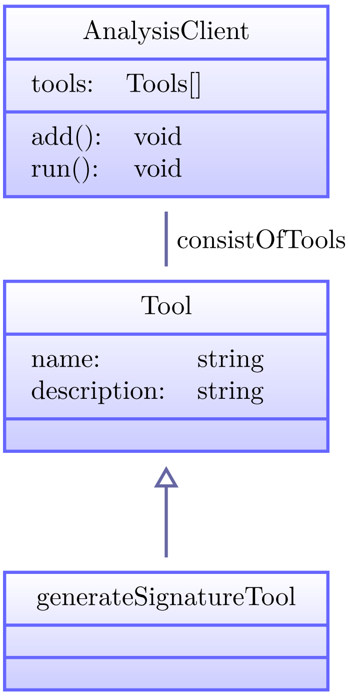


Figure 9.1: `NAAS` integration in a simplified UML class diagram.

Chapter 10

Evaluation

This chapter contains an evaluation of achievements from the work described in this thesis. The chapter starts by listing whether the stated result goals are reached, and elicited requirements fulfilled. The chapter finishes by comparing the thesis to related work, and describes limitations in the system that is possible to improve in further work.

10.1 Result goals achieved, system requirements fulfilled

This section contains a discussion about whether each result goal is reached, and to which extent the requirements are fulfilled. Result goals are stated in Section 1.2, and the full requirements elicitation is documented in Section 7.3. To make it unnecessary to turn back to these sections when reading, Table 10.1 on the following page repeats each result goal, and the description of each requirement. A summary is shown in Table 10.2 on page 118.

10.1.1 Result goals

The problems of malware have been mentioned throughout the thesis, but particularly Chapter 2 is dedicated to discussions about threats from malware and available countermeasures. The chapter contains research and documentation sufficient for reaching the first result goal (**RG.01**).

A malware analysis scenario has been performed in this thesis, focusing on the dynamic analysis phase and excluding static analysis. Since the problem description explicitly emphasises the dynamic analysis phase, **RG.02** is considered reached by the analysis scenario documented in Chapter 5.

Chapter 6 contains a survey of possible tasks to automate, thus tasks that reduce time spent during dynamic analyses. The content of two of the tasks are combined into requirements for a new system able to automate these tasks. **RG.04** is reached by implementing the final system, and its corresponding documentation.

The implemented IDS signature generation can be used to get an overview over infected hosts in a network. The signature generation process is in consequence directly useful to handle incidents where malware is involved, and with the content of Chapter 6, **RG.03** is therefore reached.

Label	Goal
RG.01	Study the problems of malware, how the malicious software operates and how it propagates. Discuss available countermeasures to the problems, and reason for why these methods are not sufficient to overcome threats from malware.
RG.02	Describe the structure of a malware analysis. State of the art analytical methods and professional tools used in such an analysis shall be studied.
RG.03	Choose a solution that can be used to reduce the time spent during a manual dynamic analysis phase. The solution shall be possible to realise as an automated process of the dynamic malware analysis phase. The solution should be capable of handling incidents where the malware is involved (the solution should be “actionable”).
RG.04	Implement a system able to automate the solution chosen in **RG.03**. The system shall not be fully functional, but merely a proof of concept implementation.
RG.05	Use the implemented solution and look into the possibility of integrating the solution with NorCERT’s existing system for handling malware analyses. Describe what is needed to integrate the implemented system.

(a) List of result goals

Label	Description
F.01	The system should execute (run) samples and monitor API calls made while they are running.
F.02	The system shall generate signatures to be used in the `Snort` IDS based on network traffic during execution of samples.
F.03	The system shall be *secure* and shall be automatically reverted to a clean state after each executed sample.
F.04	The system may optionally fetch text data (strings) in samples.
F.05	The system may optionally parse samples for executable packer technologies used to compress it.
NF.01	The system should be released in open source form to allow the public to utilise and improve the code.
NF.02	The system shall be sufficiently documented so modifications can be made without familiarise with the complete code base.
NF.03	It is recommended that it is manageable to upgrade code strongly related to `Zero Wine`. That is, when new releases of `Zero Wine` are made public, the new functionality should be possible to integrate in the system.

(b) Description of requirements

Table 10.1: To make it unnecessary to turn back to Section 1.2 (where result goals are stated), and Section 7.3 (where requirements are elicited), key information about result goals and requirements is repeated.

Third party tool integrations in NorCERT's analytical system, NAAS is under development by NorCERT's software developers. By using NAAS' current Python interfaces, a sketch of how to integrate the final system with NAAS is documented in Section 9.6 on page 112. Due to the incompleteness of the integration interface, and the emphasis on "*look into an integration*" from the problem description, delivery of the sketched design is considered sufficient to reach **RG.05**.

10.1.2 Functional requirements

The final system executes samples using Wine, which is set to filter out suspicious API calls when a sample is running. The system presents the results to the user, thus satisfying requirement **F.01**.

An IDS signature is generated during the execution of a sample. It is structured to find infected hosts in a network based on observed network traffic generated by a sample during its execution. As a result, requirement **F.02** is fulfilled.

By utilising functionality from the vmcom program previously implemented during this thesis' preliminary project together with platform virtualization by VMware, the final system is automatically reverted to a clean state after each sample execution. Such an approach is made possible by the use of virtualization software, and ensures a secure and non-infected environment. Requirement **F.03** is therefore fulfilled.

Text content of binary files are parsed with help from the program GNU strings, thus fulfilling requirement **F.04**.

Utilising functionality from PEfile and an open packer signature database, the final system is able to localise 1832 different packer technologies. However, only *one* layer of compression is found with the final system, and recursively packed layers are not exposed. **F.05** is for that reason only partly reached, but the requirement is given a low prioritisation, so an incomplete fulfilment of the requirement is not critical. It may be possible to manually unpack the sample and rerun the check, but this procedure is not implemented in the final system.

10.1.3 Non-functional requirements

The final system is released under the GPL license that ensures the system and further modifications remain open source and freely available. Requirement **NF.01** is thus fulfilled.

Best practice documentation conventions are used to document the final system in the code. Additionally, Chapter 9 is dedicated to explain the structure and functionality of the system, thus satisfying **NF.02**.

Method names and the content of methods are kept unchanged wherever possible. In some occasions, where Zero Wine originally had tangled its web interface together with the analytical functionality, the corresponding methods were changed to sustain a readable code structure. Additionally, some methods from Zero Wine, unused by the final system, are completely removed. This choice makes requirement **NF.03** partly fulfilled, but it is a low prioritisation requirement so a partial fulfilment is not critical.

Result goal	Reached?
RG.01	**Yes**, by Chapter 2 that studies threats and countermeasures concerning malware.
RG.02	**Yes**, by the malware analysis scenario in Chapter 5.
RG.03	**Yes**, by work documented in Chapter 6, where focus areas for the final system are chosen.
RG.04	**Yes**, by the final system.
RG.05	**Yes**, by the sketched integration design found in Section 9.6 on page 112.

(a) Reaching result goals

Requirement	According to measurements?
F.01	**Yes**, by utilising functionality from `Zero Wine`.
F.02	**Yes**, by modifying `Zero Wine` and adding new functionality during execution of samples.
F.03	**Yes**, by automatically revert to a clean state after each sample execution.
F.04	**Yes**, by utilising functionality from the `GNU strings` application.
F.05	**Partly**. The system is able to localise 1832 different packer technologies, but only *one* layer of compression and not layers below if packing is applied recursively.
NF.01	**Yes**, the final system is released under the GPL license.
NF.02	**Yes**, the final system is documented according to best practice conventions.
NF.03	**Partly**. Method names and content are kept wherever possible, but some changed are made to sustain readability and follow code conventions.

(b) Fulfilment of requirements

Table 10.2: Reaching result goals, and fulfilment of requirements.

10.2 Discussion

The information presented in this thesis provides a rich picture of the negative consequences by an increasing prevalence of malware on the Internet. The thesis has clarified malware as advanced pieces of software, and deducing malware's functionality is a complex process that requires much manual work and human intervention. Automating tasks performed in analyses is a way to decrease the necessary amount of resources to conduct an analysis.

This thesis has showed it is possible to automate common analytical tasks, and has finished the implementation of a proof of concept system. The system performs various analytical tasks, but particular focus has been kept on running malware, observing its network usage, and generating signatures for intrusion detection systems (IDSes) in an automated manner. Using the generated signatures, one can observe a network monitored by the `Snort` IDS for data traffic that matches the signature.

A host is considered infected if the signature matches traffic to or from the host. The probability of a false positive is based on how coarsely grained the granularity level of signature is, but the granularity level is a modifiable parameter. By utilising the generated signature, one is able to find compromised hosts in a network, and eventually be able to repair them or disable them to prevent them causing further harm.

The implemented system is unique of its kind by the way signatures are generated, and on which data basis that is used to derive them. The final system is focusing on traffic source and destination, contrasting the `Polygraph` system that was suggested by Newsome et al. [2005]. `Polygraph`, and its predecessor `Autograph`, focus on *content* in the observed data traffic—not source and destination addresses and ports. `Zero+One` is also easy to integrate with analytical systems.

10.3 Further work

The realised system is automated, leading to a more swift and easier analytical process than manual analyses. The system requires only minor interaction with an analyst, and scales better than a normal, often manually performed, dynamic analysis process. Still, and not surprisingly, there are room for improvements. The final system is a proof of concept implementation and not a fully functional realisation, so further work and improvements were always expected.

This section discusses various areas in the final system that can be improved and suggests solutions to the issues. The suggested solutions are feasible for future work.

10.3.1 General issues with dynamic analyses

The final system is strictly connected to the dynamic analysis phase and therefore suffers from some of the same general problems in the phase. Key issues from the dynamic analysis phase are discussed in Section 4.7 on page 49, so they are not repeated.

The problem of observing single path executions are the most prominent weaknesses from dynamic analyses and this is also reflected in the final system.

The input sample is executed and analysed *once* when requesting an IDS signature or API calls listing from the system. No attempts are made to force multiple execution paths during the execution, but results are instead gathered from one single execution of the sample. Without knowing the structure of the sample, an analyst is never guaranteed to generate a signature that covers all possible network traffic generated by a sample. This is simply because the system does not guarantee to visit all possible execution paths in the executable file. Looking for packer information and string parsing are not affected by the issue, as these processes are not required to execute the sample.

Is is possible to force the system to visit all conditional branches, but the techniques to do so are too complex for the scope of this thesis and saved for future work.

As explained in Section 8.7.3 on page 99, it is also straightforward to detect the analytical environment used in the system. Samples can therefore easily halt its own execution if `Wine` is found, using simple checks to locate the environment. Forcing multiple execution paths in the signature generation would have resolved the issue by manipulating the results returned by the conditional check.

10.3.2 Using host names instead of IP addresses

Fast flux networks are a set of compromised computers and constantly changing DNS records that points to them. A domain name can have thousands of IP addresses assigned to it concurrently [Holz et al., 2008]. The domain name on the other hand points to *one* single IP address at a given time, but the pointer changes as often as every three minutes to a new IP address [Riden, 2007]. Fast flux networks are used to ensure high service availability by having a service redundantly spread over a large number of (compromised) machines. If one machine is disabled, the host name pointer changes to another machine's IP address, which is online and available. Additionally, fast flux networks effectively prevent an authority to take down the service due to its highly distributed architecture. Signatures generated by the final system is based on *IP addresses*, leading to a potential enormous number of relevant IP addresses when analysing a sample that utilises fast flux networks. In the worst case, each infected machine contacts distinct IP addresses. The generated signature will then consist of a set of unique IP addresses unique for *that particular infection*. A way to solve this problem is to store the *host name* (which is constant) when generating signatures. The host name should be used in the signature instead of its current IP address pointer, which is changed almost instantly after the signature is generated. All IP addresses pointed to by the host name must of course be fetched prior to using the signature in an IDS. Special *passive DNS databases* that keeps track over host names DNS history exists, and are handy for cases like this [Weimer, 2005]. These DNS databases listen to all kinds of DNS queries and continuously store results returned by the DNS name servers. Eventually, the databases have history of a domain name's history and usage. The web site `http://www.bfk.de/bfk_dnslogger.html` is one of the services available to show the pointer history of a host name. However, automated queries to the services are not allowed without explicit agreements with the respective owners.

10.3.3 Adding a whitelist

The signature generation is currently not able to distinguish between obviously legitimate IP addresses from more suspicious addresses, leading to legitimate IP addresses showing up in the generated IDS signature. When the signature is used, the analysts may falsely conclude clean hosts are infected. It is possible to reduce the amount of such false positives by manually checking the generated IDS signature for known legitimate IP addresses or their host names. It is also possible to create a *whitelist*, which consists of trusted host names that should not be added to generated signatures.

The same problem applies to generated signatures having too high granularity levels. One IP addresses in a subnetwork might be used for suspicious actions, but the rest of the IP addresses in the network might be perfectly legitimate. In consequence, too low granularity levels should be used with caution when generating signatures.

10.3.4 Using additional metadata during signature generation

Signatures are generated solely on observed addresses and ports in network traffic. By abstracting away the necessity of processing *data content* in network traffic, the system is unique compared to previous work. There are still implications with such an approach, notably the fact that false negatives and positives can easily be introduced when using the generated signature.

Data content matching can be used to strengthen the generated signatures, and decrease the probability of detecting false positives and negatives when using the signature to find infected hosts. The data content should still be prioritised lower in the signature than addresses and ports are, but used as *assistance* in signatures to reduce false positives and negatives.

10.3.5 Using statistical filter algorithms

The selection of *which* addresses and ports to use in the generated signature is now straightforward; all IP addresses are processed, and the highest frequently used ports are chosen. Vital ports may easily be missed, making this approach error prone and subject to false negatives. The correct port value(s) can be deduced using statistical algorithms that observe similarities in *multiple executions* of the same sample. These algorithms should be capable of filtering out the fixed port(s), or a range of changing ports that are used by the malware and increase the quality level of generated IDS signatures.

Chapter 11

Thesis Conclusion

The dangers from malicious software become imminent by its increasing level of complexity and prevalence on the Internet. The key to countermeasure malware is to understand how it operates, but deducing its functionality is a time consuming and difficult task. Malware exposes a serious threat to computer systems, and efficient solutions to prevent the software from performing malicious actions are needed more than ever before.

Malware is highly dependent on the Internet to perform its malicious duty, spanning from simple checks to determine if an Internet connection is available, to participation in enormous distributed attacks. When malware spreads to other hosts, it replicates or creates a variance of its original form. The new sample may look different, but usually behaves the exact same way on hosts it successfully infects.

11.1 Contributions

This thesis has presented a thorough preliminary process of studying malware—from trivial programs that bug the user, to sophisticated and complex pieces of software. Malware producers use large amounts of resources to create software that is efficient and dangerous. They struggle to implement programs that evade detection and can avoid exposing their own malicious code. The thesis has shown how state-of-the-art programming techniques, in both a theoretical and practical way, are utilised in such software to make deducing its functionality difficult. Likewise, the thesis presents countermeasures to malware, and how to avert computer systems from being compromised—both for malware analysts, and end-users. The thesis points out a clear indication that more effective solutions are needed to fight malicious code when the prevalence of malware and its sophistication level continue to increase.

Automating resource demanding malware analyses is one of the steps to reduce the necessary human intervention, and thus save analysts for precious time. However, malware analyses are subject to significant variations depending on the observed behaviour of a sample, meaning one analysis can be performed completely different from another. These variations make a complete automation of an analysis infeasible, but constructing tools to automate *common processes and tasks* in an analysis, is the first step to save valuable time for analysts. The thesis presents an in-depth study of which properties such a system should have, and why it is so. A prioritisation of available approaches is made to select the correct choices properly, and these choices are reasoned for throughout the thesis. Based on the findings, a system is outlined and designed. The system automatically creates a signature for an intrusion detection system (IDS) by executing malware and observing its network activity. The system is realised as a proof of concept implementation that, in a secure manner, controls the execution of the malware and generates the IDS signature. The signature uses a syntax equal to the rule specifications defined by the `Snort` IDS, and can thus be easily loaded in `Snort` IDSes. The system is can be integrated into existing solutions for handling malware analyses using minor necessary effort. The generated signature is useful to identify infected hosts in an arbitrary large network by watching the network activity, and allows analysts to take appropriate action for hosts identified as being infected. Otherwise, since malware tends to camouflage its existence, hosts may be infected, under control by an attacker, and used for malicious activity—without anyone being aware of the incident. Additionally, functionality commonly seen in sandboxes is included in the system. This choice is made to show that the system is capable of performing multiple tasks, allowing analysts to use only one analytical tool in an analysis.

The results goals defined in the thesis are reached, and all medium– and high-prioritised requirements are fulfilled. The system works as expected and has the potential to significantly reduce the necessary human intervention when locating infected hosts in an IDS covered network. However, the system is a proof of concept implementation, and is subject to improvements and further work. The system can be modified to suit any network monitored by an IDS, and is a strong contribution when estimating the scale of an infection. The system assists the process of identifying infected hosts, which gives the opportunity to immediately disable or repair the hosts to prevent the malware from doing further harm. Being an open source solution allows anyone that sees potential in the software to contribute to the code. In time, the system may grow to a complete dynamic analysis tool and a free full value sandbox system that is a competitor for commercial sandbox solutions and malware analysis tools.

11.2 The future

Producers of malware are no longer computer savvy teenagers, but rather organised groups of experts having vast amount of resources. Malware's goals are no longer to tease the user, but is a profitable business using sophisticated and state-of-the-art techniques. The fight against malicious software continues, and current trends indicate it will be increasingly difficult to countermeasure the threats exposed by the software. Automation of analyses are one way to more swiftly deduce malware's functionality to be able to produce countermeasures, but the analyses will eventually take too much time as the prevalence of malware in time becomes overwhelming. New approaches to decrease the potential value of an attack are necessary to make the cost of an attack higher than the value gained if the attack is successful. By forcing producers of malware to spend more resources on the malware than it is able to generate as profit, the malware market would significantly reduce, and eventually vanish or change beyond recognition. The techniques to achieve such an utopia are far from obvious, but approaching the problem from multiple angles are most certainly necessary. Software must be developed with explicit requirements regarding its security, more resources must be dedicated to produce code that is free for bugs and designs that have no flaws. Users must be trained to be able to withstand obvious intrusion attempts and to understand the threats from malware and consequences by a security breach. Appliances dedicated to protect hosts from malware should be unnecessary, and kept as assistance in defence in depth strategies only. When such appliances are used, they should be efficient, solid and trustworthy to *guarantee* the results they supply are correct. Approaches such as these could effectively reduce, or in time eliminate, the available attack points in computer systems, and thus reduce the possible consequences from malware. The process of eliminating all malware is lengthy, but each approach, solution and method reducing the necessary human intervention when fighting malware is one step closer to utopia.

References

David Ahmad. The contemporary software security landscape. *IEEE Security and Privacy*, 5(3):75–77, 2007. ISSN 1540-7993. doi: http://doi.ieeecomputersociety.org/10.1109/MSP.2007.73.

V. Anupam, A. Mayer, K. Nissim, B. Pinkas, and M.K. Reiter. On the security of pay-per-click and other Web advertising schemes. *Computer Networks-the International Journal of Computer and Telecommunications Networkin*, 31(11): 1091–1100, 1999.

W.A. Arbaugh, W.L. Fithen, and J. McHugh. Windows of vulnerability: a case study analysis. *Computer*, 33(12):52–59, Dec 2000. ISSN 0018-9162. doi: 10.1109/2.889093.

M. Bailey, J. Oberheide, J. Andersen, Z.M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. *Lecture Notes in Computer Science*, 4637, 2007.

S.J. Barr, S.J. Cardman, and D.M. Martin. A boosting ensemble for the recognition of code sharing in malware. *Journal in Computer Virology*, 4(4):335–345, 2008.

D. Barroso. Botnets–The Silent Threat. *ENISA position paper, November*, 2007.

U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. A View on Current Malware Behaviors. Technical report, Technical University Vienna, 2008.

Ulrich Bayer. TTAnalyze: A Tool for Analyzing Malware. Master's thesis, Technical University of Vienna, 2005.

P. Beaucamps. Advanced polymorphic techniques. *International Journal of Computer Science*, 2(3):194–205, 2007.

Steven M. Bellovin and William R. Cheswick. Network firewalls. *Communications Magazine, IEEE*, 34:50–57, 1994.

J. Bethencourt, D. Song, and B. Waters. Analysis-Resistant Malware. In *15th Annual Network & Distributed System Security Symposium*, pages 10–13, 2008.

Eli Biham and Rafi Chen. Near-Collisions of SHA-0. *Lecture Notes in Computer Science*, 3152:290–305, 2004.

Nick Bilogorskiy. Click Me: Social Engineering in Malware, 2005. Online at `http://www.fortiguardcenter.com/resources/click.me.pdf`. Last date accessed 2009-03-08.

L. Bohne. *Pandora's Bochs: Automatic Unpacking of Malware*. PhD thesis, University of Mannheim, 2008.

J.M. Borello and L. Mé. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3):211–220, 2008.

D. Bradbury. Batten down the hatches. *Infosecurity*, 5(6):26–29, 2008.

Danny Bradbury. The metamorphosis of malware writers. *Computers & Security*, 25:89–90, 2006.

S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119 (Best Current Practice), March 1997. URL `http://www.ietf.org/rfc/rfc2119.txt`.

Eric Braude. *Software Engineering: An Object-Oriented Perspective*. Wiley, 2000.

Lloyd Bridges. The changing face of malware. *Network Security*, 2008(1): 17–20, 2008. ISSN 1353-4858. doi: DOI:10.1016/S1353-4858(08)70010-2. URL `http://www.sciencedirect.com/science/article/B6VJG-4RNS8H7-B/2/bf940bf4a4110e9a934925f4d28be3b5`.

N. Brooks. Repeat search behavior: Implications for advertisers. *Bulletin of the American Society for Information Science and Technology*, 32(2), 2006.

D. Brumley, C. Hartwig, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Towrads automatically identifying trigger-based behavior in malware. *Botnet Detection*, 36:65–88, 2007.

D. Bruschi, L. Martignoni, and M. Monga. Using code normalization for fighting self-mutating malware. In *Proceedings of International Symposium on Secure Software Engineering*, 2006.

Paul Byrne. Application firewalls in a defence-in-depth design. *Network Security*, 2006(9):9–11, 2006. ISSN 1353-4858. doi: DOI:10.1016/S1353-4858(06)70422-6. URL `http://www.sciencedirect.com/science/article/B6VJG-4KWCG6Y-6/2/a814796dc70a8c1892ac3622bc3a5f2b`.

A.M. Cansian, M. Souza, and A.R.A. Silva. Developing an Attack Signature Standard. In *Proceedings of SAM'02-The 2002 International Conference on Security and Management*, 2002.

D. Chaikin. Network investigations of cyber attacks: the limits of digital evidence. *Crime, Law and Social Change*, 46(4):239–256, 2006.

M.R. Chouchane and A. Lakhotia. Using engine signature to detect metamorphic malware. In *Proceedings of the 4th ACM workshop on Recurring malcode*, pages 73–78. ACM New York, NY, USA, 2006.

M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, H. Veith, et al. Malware normalization. Technical report, Technical Report 1539, University of Wisconsin, Madison, Wisconsin, USA, 2005.

M. Christodorescu, S. Jha, J. Kinder, S. Katzenbeisser, and H. Veith. Software transformations to improve malware detection. *Journal in Computer Virology*, 3 (4):253–265, 2007.

Mihai Christodorescu and Somesh Jha. Testing malware detectors. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 34–44, 2004.

Mihai Christodorescu, Somesh Jha, Sanjit Seshia, Dawn Song, and Randal Bryant. Semantics-Aware Malware Detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, 2006.

Alma Cole, Michael Mellor, and Daniel Noyes. Botnets: The Rise of the Machines. In *Proceedings on the 6th Annual Security Conference*, 2007.

C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, University of Auckland, 1997.

L. Copeland. *Practitioners guide to software test design, A*. Artech House Publishers, 2004.

J.F. Curran. Internet Crime Victimization: Sentencing. *Mississippi Law Journal*, 76:909–922, 2006.

D. Dagon, G. Gu, C. Zou, J. Grizzard, S. Dwivedi, W. Lee, and R. Lipton. A taxonomy of botnets, 2005. Unpublished paper online at `http://www.math.tulane.edu/~tcsem/botnets/ndss_botax.pdf`. Last date accessed 2009-03-12.

Rachna Dhamija, J. D. Tygar, and Marti Hearst. Why phishing works. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 581–590, New York, NY, USA, 2006. ACM. ISBN 1-59593-372-7. doi: http://doi.acm.org/10.1145/1124772.1124861.

D. Dittrich and S. Dietrich. New directions in peer-to-peer malware. *Sarnoff Symposium, 2008 IEEE*, pages 1–5, April 2008. doi: 10.1109/SARNOF.2008.4520101.

David Dittrich and Sven Dietrich. Command and control structures in malware. *LOGIN*, 32, 2007.

T.E. Dube, B.D. Birrer, R.A. Raines, R.O. Baldwin, B.E. Mullins, R.W. Bennington, and C.E. Reuter. Hindering reverse engineering: Thinking outside the box. *Security & Privacy, IEEE*, 6(2):58–65, March-April 2008. ISSN 1540-7993. doi: 10.1109/MSP.2008.33.

Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley, 2005.

Gergely Erdélyi. Hide'n'seek? anatomy of stealth malware. *Virus Bulletin Conference*, 2004.

P. Ferrie. Attacks on virtual machine emulators. *Symantec Security Response*, 2006.

D. Florencio and C. Herley. Stopping a phishing attack, even when the victims ignore warnings. *Microsoft Research*, 2005.

B. J. Fogg, Jonathan Marshall, Othman Laraki, Alex Osipovich, Chris Varma, Nicholas Fang, Jyoti Paul, Akshay Rangnekar, John Shon, Preeti Swani, and Marissa Treinen. What makes web sites credible?: a report on a large quantitative study. In *CHI '01: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 61–68, New York, NY, USA, 2001. ACM. ISBN 1-58113-327-8. doi: http://doi.acm.org/10.1145/365024.365037.

Marin Fowler. *UML Distilled*. Addison-Wesley, 3 edition, 2003.

E. Gabrilovich and A. Gontmakher. The homograph attack. *Communications of the ACM*, 45(2), 2002.

MN Gagnon, S. Taylor, and AK Ghosh. Software Protection through Anti-Debugging. *IEEE Security & Privacy*, 5(3):82–84, 2007.

Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is not transparency: VMM detection myths and realities. In *HotOS 2007*, 2007.

Luiz Henrique Gomes, Cristiano Cazita, Jussara M. Almeida, Virgílio Almeida, and Wagner Meira, Jr. Characterizing a spam traffic. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 356–369, New York, NY, USA, 2004. ACM. ISBN 1-58113-821-0. doi: http://doi.acm.org/10.1145/1028788.1028837.

Greg Goth. The politics of ddos attacks. *Distributed Systems Online, IEEE*, 8(8), August 2007. ISSN 1541-4922. doi: 10.1109/MDSO.2007.50.

Heather Goudey. Watch the Money-Go-Round, Watch the Malware-Go-Round. In *Proceedings from the Virus Bulletin Conference 2004*, 2004.

D. Grawrock. The Intel Safer Computing Initiative, 2005. Online at `http://www.intel.com/intelpress/chapter-secc_ch05.pdf` Last date accessed 2009-05-10.

Lynn Greiner. The new face of malware. *netWorker*, 10(4):11–13, 2006. ISSN 1091-3556. doi: http://doi.acm.org/10.1145/1186564.1186572.

J.B. Grizzard, V. Sharma, C. Nunnery, B. Kang, and D. Dagon. Peer-to-peer botnets: Overview and case study. In *Proceedings of the First USENIX Workshop on Hot Topics in Understanding Botnets*, 2007.

D. Gryaznov. Malware in Popular Networks. In *Proceedings of the 15 th virus bulletin international conference*, 2005.

X. Gu and R. Hunt. Wireless LAN attacks and vulnerabilities. *the proceeding of IASTED Networks and Communication Systems*, pages 18–20, 2005.

F. Guo, P. Ferrie, and T.C. Chiueh. A Study of the Packer Problem and Its Solutions. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 98–115. Springer, 2008.

C. Hare and K. Siyan. *Internet firewalls and network security.* New Riders, 1996.

David Harley and Andrew Lee. The Root of All Evil? - Rootkits Revealed. White paper, ESET, 2006. Online at `http://www.eset.hk/softdown/manual/Whitepaper-Rootkit_Root_Of_All_Evil.pdf` Last date accessed 2009-04-12.

J.L. Harrington. *Network security: a practical approach.* Morgan Kaufmann, 2005.

Lars Haukli. Analysing Malicious Code: Dynamic Techniques. Master's thesis, Norwegian University of Science and Technology, 2007.

Jay G. Heiser. Understanding today's malware. Technical report, TruSecure Ltd, 2004.

S. Heron. Botnet command and control techniques. *Network Security*, 2007(4): 13–16, 2007a.

S. Heron. Working the botnet: how dynamic DNS is revitalising the zombie army. *Network Security*, 2007(1):9–11, 2007b.

Magnus Lie Hetland. *Beginning Python, Novice to Professional.* APress, 2005.

S. Hilley. Five years for Californian botmaster. *Network Security*, 2006(6), 2006.

Neal Hindocha and Eric Chien. Malicious threats and vulnerabilities in instant messaging. White paper, Symantec Security Response, 2003. Online at `http://www.internetnortonsecurity.com/avcenter/reference/malicious.threats.instant.messaging.pdf` Last date accessed 2009-02-06.

G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel.* Addison-Wesley Professional, 2005.

T. Holz, C. Gorecki, K. Rieck, and F.C. Freiling. Measuring and Detecting Fast-Flux Service Networks. In *Proceedings of the Network & Distributed System Security Symposium*, 2008.

F. Hsu, Hao Chen, T. Ristenpart, J. Li, and Zhendong Su. Back to the future: A framework for automatic malware removal and system repair. In *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pages 257–268, Dec. 2006. doi: 10.1109/ACSAC.2006.16.

Nicholas Ianelli and Aaron Hackworth. Botnets as a Vehicle for Online Crime. *The International Journal of FORENSIC COMPUTER SCIENCE*, 1:19–39, 2007.

Nicole Immorlica, Kamal Jain, Mohammad Mahdian, and Kunal Talwar. Click Fraud Resistant Methods for Learning Click-Through Rates. In *Internet and Network Economics*, pages 34–45. Springer Berlin / Heidelberg, 2005.

B.J. Jansen. Adversarial information retrieval aspects of sponsored search. In *the 2 nd International Workshop on Adversarial Information Retrieval on the Web (AIRWeb)*, 2006.

Ari Juels, Sid Stamm, and Markus Jakobsson. Combating Click Fraud via Premium Clicks. In *Proceedings from the 16th USENIX Security Symposium*, pages 17–26, 2007.

R. Kath. The Portable Executable file format from top to bottom. *MSDN Library, Microsoft Corporation*, 1993.

S. Katzenbeisser and F.A. Petitcolas. *Information hiding techniques for steganography and digital watermarking.* Artech House, Inc. Norwood, MA, USA, 2000.

S. Katzenbeisser, C.schallhart, and H. Veith. Malware engineering. Institut fur Informatik, Technische Universitat Munchen, 2005.

James Kay. Low volume viruses: new tools for criminals. *Network Security*, 2005 (6):16–18, 2005. ISSN 1353-4858. doi: DOI:10.1016/S1353-4858(05)70249-X. URL `http://www.sciencedirect.com/science/article/B6VJG-4GHB78C-9/2/c2d1f2bacc016147652ffef8135cf0d4`.

H.A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of 13th USENIX Security Symposium*, pages 271–286, 2004.

E Konstantinou. Metamorphic Virus: Analysis and Detection. Technical report, Royal Holloway, University of London, 2008.

Christian Kreibich and Jon Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *SIGCOMM Comput. Commun. Rev.*, 34(1):51–56, 2004. ISSN 0146-4833. doi: http://doi.acm.org/10.1145/972374.972384.

Kris-Mikael Krister. Diffie-Hellman Key-Exchange ...in detail., 2007. Online at `http://folk.ntnu.no/krismika/dh_in_detail.pdf`. Last date accessed 2009-03-17.

Kris-Mikael Krister. Automated testing of antivirus coverage. Preliminary project for this master thesis, 2008. Online at `http://folk.ntnu.no/krismika/automated_detection_of_ac_c_final_report.pdf` Last date accessed 2009-02-06.

Kris-Mikael Krister, Egil Trygve Baadshaug, Daniele Spampinato, Eilev Hagen, Ketil Velle, and Eirik Reksten. SeaMonster - a security modeling software, 2007. Online at `http://folk.ntnu.no/krismika/seamonster/seamonster_final_report-web.pdf` Last date accessed 2009-02-06.

Park KyoungSoo, Pai Vivek, Lee Kang-Won, and Calo Seraphin. Securing Web Service by Automatic Robot Detection. In *Proceedings from the USENIX Technical Conference*, 2006.

A. Lakhotia and M. Mohammed. Imposing order on program statements to assist anti-virus scanners. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 161–170, 2004.

James Lance. *Phishing Exposed.* Syngress, 2006.

F. Lau, S.H. Rubin, M.H. Smith, and L. Trajkovic. Distributed denial of service attacks. *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, 3:2275–2280, 2000. ISSN 1062-922X. doi: 10.1109/ICSMC.2000.886455.

Neal Leavitt. Instant messaging: a new target for hackers. *Computer*, 38:20–23, 2005.

W. Lee, J.B.D. Cabrera, A. Thomas, N. Balwalli, S. Saluja, and Y. Zhang. Performance adaptation in real-time intrusion detection systems. *Lecture notes in computer science*, pages 252–273, 2002.

J. Lerner and J. Tirole. Some simple economics of open source. *Journal of Industrial Economics*, pages 197–234, 2002.

M. Lesk. The new front line: Estonia under cyberassault. *Security & Privacy, IEEE*, 5(4):76–79, July 2007. ISSN 1540-7993. doi: 10.1109/MSP.2007.98.

E. Levy. The making of a spam zombie army. dissecting the sobig worms. *Security & Privacy, IEEE*, 1(4):58–59, July-Aug. 2003. ISSN 1540-7993. doi: 10.1109/MSECP.2003.1219071.

Zhen Li, Qi Liao, and Aaron Striegel. Botnet Economics: Uncertainty Matters, 2008a. Presented at the Workshop on the Economics of Information Security (WEIS) 2008. Online at `http://weis2008.econinfosec.org/papers/Liao.pdf`. Last date accessed 2009-03-06.

Zhuowei Li, XiaoFeng Wang, Zhenkai Liang, and M.K. Reiter. Agis: Towards automatic generation of infection signatures. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 237–246, June 2008b. doi: 10.1109/DSN.2008.4630092.

R. Lippmann, S. Webster, and D. Stetson. The effect of identifying vulnerabilities and patching software on the utility of network intrusion detection. *Lecture notes in computer science*, pages 307–326, 2002.

R. Lyda and J. Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE SECURITY AND PRIVACY MAGAZINE*, 5(2), 2007.

S. Magruder and S.X. Lewis Jr. ESPIONAGE VIA MALWARE. *Academy of Information and Management Sciences*, 10(1), 2006.

D.E. Mann and S.M. Christey. Towards a common enumeration of vulnerabilities. *The MITRE Corporation*, 4(5), 1999.

L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2007.

D. Maynor and KK Mookhey. *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research.* Syngress Press, 2007.

Gary McGraw. *Software Security: Building Security In.* Addison-Wesley, 2006.

Jelena Mirkovic and Peter Reiher. A taxonomy of DDoS attack and DDoS defense mechanisms. *SIGCOMM Comput. Commun. Rev.*, 34(2):39–53, 2004. ISSN 0146-4833. doi: http://doi.acm.org/10.1145/997150.997156.

A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

I. Muttik. Fuel Malware. *Sage January*, 2008.

L. Myers. Aim for bot coordination. *Virus Research*, 2006.

Carey Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, 40(1):46–51, 1997. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/242857.242869.

J. Newsome, B. Karp, and D. Song. Polygraph: automatically generating signatures for polymorphic worms. In *Security and Privacy, 2005 IEEE Symposium on*, pages 226–241, May 2005. doi: 10.1109/SP.2005.15.

J. Oikarinen and D. Reed. Internet Relay Chat Protocol. RFC 1459 (Experimental), May 1993. URL `http://www.ietf.org/rfc/rfc1459.txt`. Updated by RFCs 2810, 2811, 2812, 2813.

Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th conference on USENIX Security Symposium, 1998-Volume 7*. USENIX Association Berkeley, CA, USA, 1998.

R. Pegoraro. Microsoft Windows: Insecure by Design. *The Washington Post Sunday, August 24, Page F07*, 2003.

F. Piessens and B. De Win. DEVELOPING SECURE SOFTWARE Asurvey and classification of common software vul-nerabilities. *Integrity, Internal Control and Security in Information Systems: Connecting Governance and Technology*, 2002.

M. Pietrek. Peering Inside the PE: A Tour of the Win32 (R) Portable Executable File Format. *MICROSOFT SYSTEMS JOURNAL-US EDITION-*, 3, 1994.

M. Pietrek. Inside Windows-An In-Depth Look into the Win32 Portable Executable File Format. *MSDN Magazine*, 17(2), 2002.

M. Polychronakis, P. Mavrommatis, and N. Provos. Ghost turns zombie: exploring the life cycle of web-based malware. In *Proceedings of the 1st USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET)*, 2008.

V. Prevelakis and D. Spinellis. Sandboxing applications. In *Proceedings of the USENIX Technical Annual Conference, Freenix Track*, pages 119–126, 2001.

Niels Provos, Moheeb Abu Rajab, and Panayiotis Mavrommatis. Cybercrime 2.0: when the cloud turns dark. *Commun. ACM*, 52(4):42–47, 2009. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1498765.1498782.

Danny Quist and Val Smith. Detecting the presence of virtual machines, 2000. Online at `http://www.offensivecomputing.net/files/active/0/vm.pdf` Last date accessed 2009-02-06.

T. Raffetseder, C. Kruegel, and E. Kirda. Detecting system emulators. *LECTURE NOTES IN COMPUTER SCIENCE*, 4779, 2007.

Anirudh Ramachandran, Nick Feamster, and Santosh Vempala. Filtering spam with behavioral blacklisting. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 342–351, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2. doi: http://doi.acm.org/10.1145/1315245.1315288.

Rafeeq Ur Rehman. *Intrusion Detection Systems with Snort*, volume 31. Prentice Hall PTR, 2003.

Jamie Riden. Know Your Enemy: Fast-Flux Service Networks, 2007. Online at `http://www.honeynet.org/papers/ff` Last date accessed 2009-05-05.

Martin Roesch. Snort - Lightweight Intrusion Detection for Networks. *LISA XIII*, 1999.

Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, 2006.

Joanna Rutkowska. Red pill... or how to detect VMM using (almost) one CPU instruction, 11 2004. Online at `http://invisiblethings.org/papers/redpill.html`, Last date accessed 2009-02-06.

Chris Sanders. *Practical packet analysis.* No Starch Press, San Francisco, CA, USA, 2007. ISBN 9781593271497.

Daniel J. Sanok, Jr. An analysis of how antivirus methodologies are utilized in protecting computers from malicious code. In *InfoSecCD '05: Proceedings of the 2nd annual conference on Information security curriculum development*, pages 142–144, New York, NY, USA, 2005. ACM. ISBN 1-59593-261-5. doi: http://doi.acm.org/10.1145/1107622.1107655.

S. Savage. Large-scale worm detection. In *ARO-DHS Special Workshop on Malware Detection*, 2005.

Joel Scambray. *Hacking Exposed Windows: Microsoft Windows Security Secrets and Solutions.* McGraw-Hill Osborne Media, 2007.

Karen Scarfone and Peter Mell. *Guide to Intrusion Detection and Prevention Systems.* National Institute of Standards and Technology, 2007.

C.L. Schuba, I.V. Krsul, M.G. Kuhn, E.H. Spafford, A. Sundaram, and D. Zamboni. Analysis of a denial of service attack on TCP. *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 208–223, May 1997. doi: 10.1109/SECPRI.1997.601338.

Eugene Schultz. Worms and viruses: are we losing control? *Computers & Security*, 23(3):179–180, 2004. ISSN 0167-4048. doi: DOI:10.1016/j.cose.2004.04.001. URL `http://www.sciencedirect.com/science/article/B6V8G-4C837GF-1/2/c323b7f32018d5d4abcc8fe3e77a920c`.

M.G. Schultz, E. Eskin, F. Zadok, and S.J. Stolfo. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 38–49, 2001. doi: 10.1109/SECPRI.2001.924286.

Randal L. Schwartz, Tom Phoenix, and Brian D. Foy. *Learning Perl*. O'Reilly Media, 5 edition, 2008.

K. Singh, A. Srivastava, J. Giffin, and W. Lee. Evaluating Email's Feasibility for Botnet Command and Control. In *IEEE International Conference on Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008*, pages 376–385, 2008.

S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of OSDI*, pages 45–60, 2004.

F. Skulason. Virus encryption techniques. *Virus Bulletin*, pages 13–16, 1990.

Danny Smith. Forming an incident response team. In *Proceedings of the FIRST Annual Conference*. Australian CERT, 1994.

L. Spitzner. *Honeypots: tracking hackers*. Addison-Wesley Professional, 2003.

W.R. Stevens. *TCP/IP illustrated*. Addison-Wesley Professional, 1994.

E. Stinson and J.C. Mitchell. Characterizing Bots' Remote Control Behavior. *Lecture Notes in Computer Science*, 4579, 2007.

P. Szewczyk, M. Brand, and W.A. Perth. Malware Detection and Removal: An examination of personal anti-virus software. In *Proceedings of the 6th Australian Digital Conference*, 2008.

P. Szor. *The art of computer virus research and defense*. Addison-Wesley Professional, 2005.

Péter Ször and Peter Ferrie. Hunting for metamorphic. In *Proceedings of the Virus Bulletin Conference 2001*, 2001.

A. Tanenbaum. *Computer networks*. Prentice Hall Professional Technical Reference, 2002.

C. Valli and M. Brand. The Malware Analysis Body of Knowledge (MABOK). In *Proceedings of The 6th Digital Forensics Conference*, pages 70–77, 2008.

A. Walenstein, R. Mathur, M.R. Chouchane, and A. Lakhotia. The design space of metamorphic malware. In *Proceedings of 2nd International Conference on Information Warfare and Security: ICIW 2007*. Academic Conferences Limited, 2007a.

A. Walenstein, M. Venable, M. Hayes, C. Thompson, and A. Lakhotia. Exploiting Similarity Between Variants to Defeat Malware. *Proceedings of BlackHat Briefings DC 2007*, 2007b.

J. Watkins. *Testing IT: an off-the-shelf software testing process*. Cambridge University Press, 2001.

Petter Langeland Wedum. Malware Analysis; A Systematic Approach. Master's thesis, Norwegian University of Science and Technology, 2008.

F. Weimer. Passive dns replication. In *17th Annual FIRST Conference on Computer Security Incident Handling (FIRST'05)*, 2005.

D. Welch and S. Lathrop. Wireless security threat taxonomy. In *Information Assurance Workshop, 2003. IEEE Systems, Man and Cybernetics Society*, pages 76–83, June 2003. doi: 10.1109/SMCSIA.2003.1232404.

Moira West-Brown, Don Stikvoort, Klaus-Peter Kossakowski, Georgia Killcrece, Robin Ruefle, and Mark Zajicek. *Handbook for Computer Security Incident Response Teams (CSIRTs)*, 2 edition, 2003.

C. Willems, T. Holz, and F. Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *Security & Privacy, IEEE*, 5(2):32–39, March-April 2007. ISSN 1540-7993. doi: 10.1109/MSP.2007.45.

Kenneth Wyk and Gary McGraw. Bridging the gap between software development and information security. *IEEE Security and Privacy*, 3(5):75–79, 2005. ISSN 1540-7993. doi: http://doi.ieeecomputersociety.org/10.1109/MSP.2005.118.

Wei Yan, Zheng Zhang, and N. Ansari. Revealing packed malware. *Security & Privacy, IEEE*, 6(5):65–69, Sept.-Oct. 2008. ISSN 1540-7993. doi: 10.1109/MSP.2008.126.

H. Yin, Z. Liang, and D. Song. HookFinder: Identifying and understanding malware hooking behaviors. In *15th Annual Network and Distributed System Security Symposium (NDSS'08)*, 2008.

Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2. doi: http://doi.acm.org/10.1145/1315245.1315261.

Xinyou Zhang, Chengzhong Li, and Wenbin Zheng. Intrusion prevention system design. In *Computer and Information Technology, 2004. CIT '04. The Fourth International Conference on*, pages 386–390, Sept. 2004. doi: 10.1109/CIT.2004.1357226.

H. Zimmermann. OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection. *Communications, IEEE Transactions on*, 28(4):425–432, Apr 1980. ISSN 0090-6778.

Z. Zuo, Q. Zhu, and M. Zhou. On the time complexity of computer viruses. *Information Theory, IEEE Transactions on*, 51(8):2962–2966, Aug. 2005. ISSN 0018-9448. doi: 10.1109/TIT.2005.851780.

Appendix A

User Manual

This appendix contains the user manual for the final system. The appendix does not contain specific details about installation of any dependencies, but focuses on usage of `vmcom lite` and `Zero+One`. The final system is located at `https://sourceforge.net/projects/zeroplusone/`.

A.1 Configuring and using Zero+One

Prior to using `Zero+One`, some configuration settings must be edited. They are located in the file "`config.py`". Each setting is self explanatory.

`Zero+One` is a set of Python scripts controlled by command line arguments. The program is able to perform the following tasks.

- Check for known packer signatures in FILE
  (`--packers --file /folder/file`)
- Scan FILE for strings (ASCII data)
  (`--strings --file /folder/file`)
- Execute FILE and get a list of suspicious API calls and an IDS signature with granularity level 16 on IP addresses.
  (`--execute --file /folder/file --granularity 16`)

The network monitoring tool `tshark` must be logging network data to the file `/var/log/tshark` before starting an analysis.

A.2 Configuring and using vmcom lite

`Zero+One` should be run in an virtual environment to avoid permanent infections when running malware. `vmcom lite` ensures the samples are safely executed in a virtual machine. `Zero+One` must be deployed on a `VMware Server` based operating system where the `VIX` API is installed on the virtual machine host for this to work.

Some settings must be edited prior to running `vmcom lite`. They are found in the configuration part of the source code.

`vmcom lite` is a Perl script controlled by command line arguments. It is able to perform the following tasks.

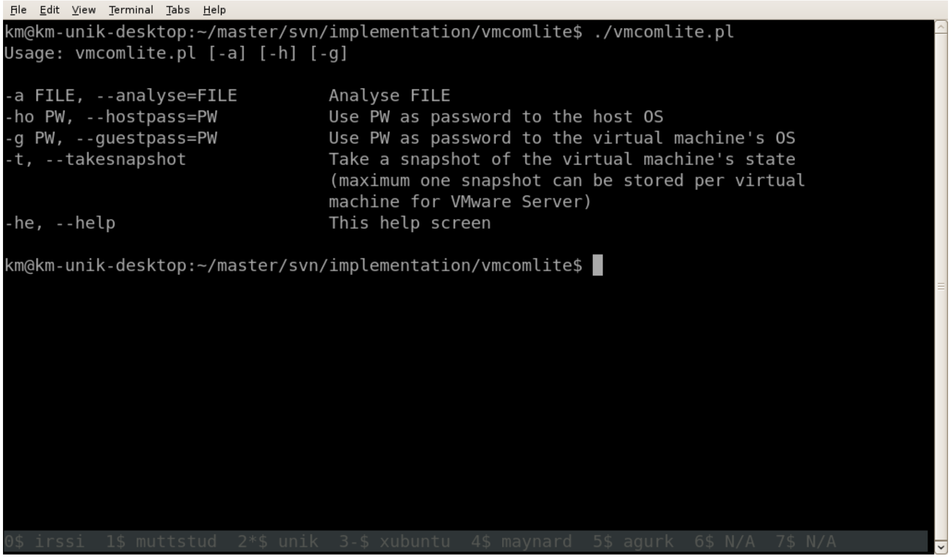
- `vmcom lite` requires a clean snapshot of a system prior to running `Zero+One`.
  `--takesnapshot`
- Run an analysis using `Zero+One`.
  `--analyse --password PASSWORD`, where PASSWORD is the guest operating system password.

Appendix B

Screen Captures

This appendix contains the following screen captures.

- Figure B.1 on the following page shows `vmcom lite`'s help screen and functionality that is accessible using command line arguments.
- Figure B.2 on the next page shows `Zero+One`'s help screen and functionality that is accessible using command line arguments.
- Figure B.3 on page 143 shows `Wireshark`, the graphical version of `tshark`, while observing network traffic.
- Figure B.4 on page 143 shows `PEiD` that has found a packer technology used in a sample.



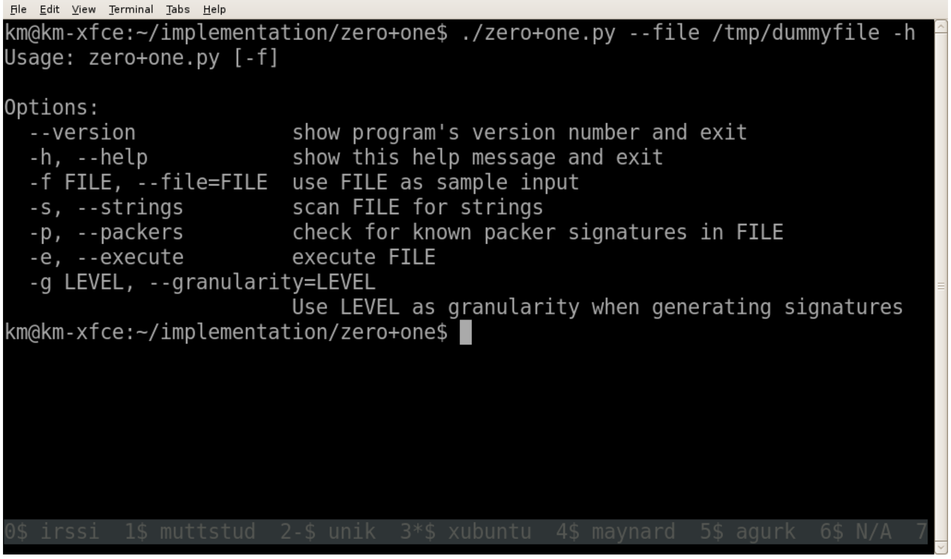
```
km@km-unik-desktop:~/master/svn/implementation/vmcomlite$ ./vmcomlite.pl
Usage: vmcomlite.pl [-a] [-h] [-g]

-a FILE, --analyse=FILE         Analyse FILE
-ho PW, --hostpass=PW           Use PW as password to the host OS
-g PW, --guestpass=PW           Use PW as password to the virtual machine's OS
-t, --takesnapshot              Take a snapshot of the virtual machine's state
                                (maximum one snapshot can be stored per virtual
                                machine for VMware Server)
-he, --help                     This help screen

km@km-unik-desktop:~/master/svn/implementation/vmcomlite$ 

0$ irssi  1$ muttstud  2*$ unik  3-$ xubuntu  4$ maynard  5$ agurk  6$ N/A  7$ N/A
```

Figure B.1: Screen capture from `vmcom lite`'s help screen.

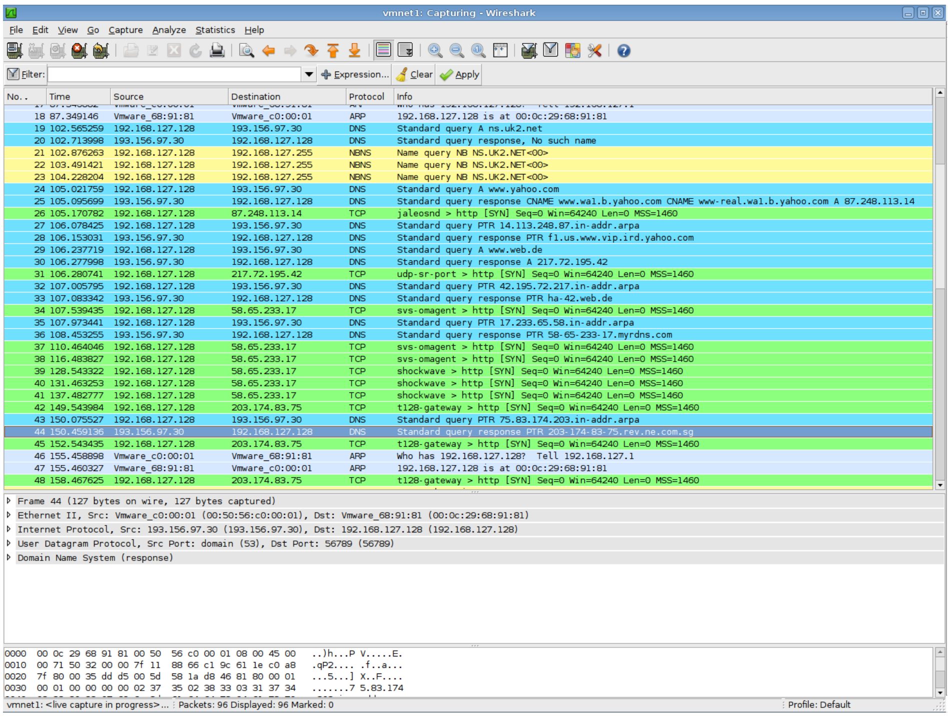


```
km@km-xfce:~/implementation/zero+one$ ./zero+one.py --file /tmp/dummyfile -h
Usage: zero+one.py [-f]

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  use FILE as sample input
  -s, --strings         scan FILE for strings
  -p, --packers         check for known packer signatures in FILE
  -e, --execute         execute FILE
  -g LEVEL, --granularity=LEVEL
                        Use LEVEL as granularity when generating signatures
km@km-xfce:~/implementation/zero+one$ 

0$ irssi  1$ muttstud  2-$ unik  3*$ xubuntu  4$ maynard  5$ agurk  6$ N/A  7
```

Figure B.2: Screen capture from `Zero+One`'s help screen.


vmnet1: Capturing - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression...
Clear
Apply

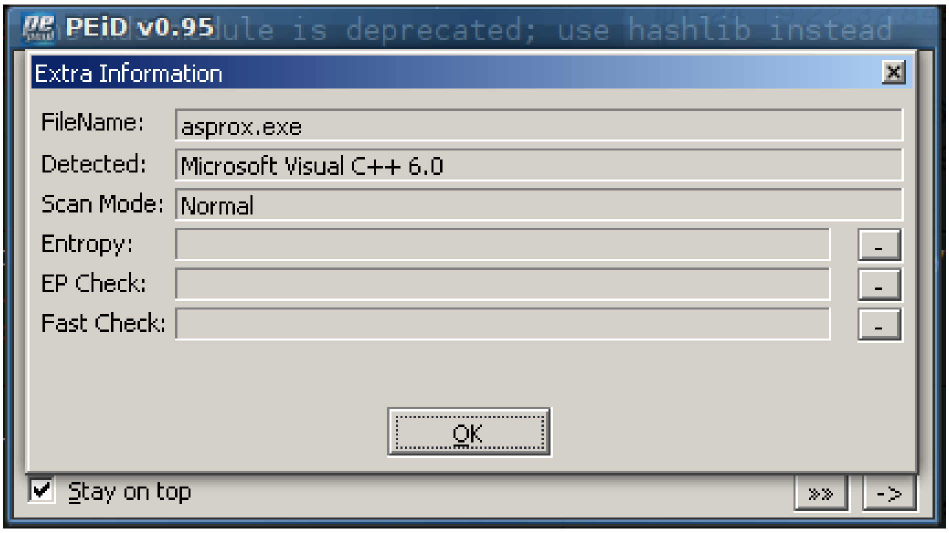
No.	Time	Source	Destination	Protocol	Info
18	87.349146	Vmware_68:91:81	Vmware_c0:00:01	ARP	192.168.127.128 is at 00:0c:29:68:91:81
19	102.565259	192.168.127.128	193.156.97.30	DNS	Standard query A ns.uk2.net
20	102.713998	193.156.97.30	192.168.127.128	DNS	Standard query response, No such name
21	102.876263	192.168.127.128	192.168.127.255	NBNS	Name query NB NS.UK2.NET<00>
22	103.491421	192.168.127.128	192.168.127.255	NBNS	Name query NB NS.UK2.NET<00>
23	104.228204	192.168.127.128	192.168.127.255	NBNS	Name query NB NS.UK2.NET<00>
24	105.021759	192.168.127.128	193.156.97.30	DNS	Standard query A www.yahoo.com
25	105.095699	193.156.97.30	192.168.127.128	DNS	Standard query response CNAME www.wa1.b.yahoo.com CNAME www-real.wa1.b.yahoo.com A 87.248.113.14
26	105.170782	192.168.127.128	87.248.113.14	TCP	jaleosnd > http [SYN] Seq=0 Win=64240 Len=0 MSS=1460
27	106.078425	192.168.127.128	193.156.97.30	DNS	Standard query PTR 14.113.248.87.in-addr.arpa
28	106.153031	193.156.97.30	192.168.127.128	DNS	Standard query response PTR f1.us.www.vip.ird.yahoo.com
29	106.237719	192.168.127.128	193.156.97.30	DNS	Standard query A www.web.de
30	106.277998	193.156.97.30	192.168.127.128	DNS	Standard query response A 217.72.195.42
31	106.280741	192.168.127.128	217.72.195.42	TCP	udp-sr-port > http [SYN] Seq=0 Win=64240 Len=0 MSS=1460
32	107.005795	192.168.127.128	193.156.97.30	DNS	Standard query PTR 42.195.72.217.in-addr.arpa
33	107.083342	193.156.97.30	192.168.127.128	DNS	Standard query response PTR ha-42.web.de
34	107.539435	192.168.127.128	58.65.233.17	TCP	svs-omagent > http [SYN] Seq=0 Win=64240 Len=0 MSS=1460
35	107.973441	192.168.127.128	193.156.97.30	DNS	Standard query PTR 17.233.65.58.in-addr.arpa
36	108.453255	193.156.97.30	192.168.127.128	DNS	Standard query response PTR 58-65-233-17.myrdns.com
37	110.464046	192.168.127.128	58.65.233.17	TCP	svs-omagent > http [SYN] Seq=0 Win=64240 Len=0 MSS=1460
38	116.483827	192.168.127.128	58.65.233.17	TCP	svs-omagent > http [SYN] Seq=0 Win=64240 Len=0 MSS=1460
39	128.543322	192.168.127.128	58.65.233.17	TCP	shockwave > http [SYN] Seq=0 Win=64240 Len=0 MSS=1460
40	131.463253	192.168.127.128	58.65.233.17	TCP	shockwave > http [SYN] Seq=0 Win=64240 Len=0 MSS=1460
41	137.482777	192.168.127.128	58.65.233.17	TCP	shockwave > http [SYN] Seq=0 Win=64240 Len=0 MSS=1460
42	149.543984	192.168.127.128	203.174.83.75	TCP	t128-gateway > http [SYN] Seq=0 Win=64240 Len=0 MSS=1460
43	150.075527	192.168.127.128	193.156.97.30	DNS	Standard query PTR 75.83.174.203.in-addr.arpa
44	150.459136	193.156.97.30	192.168.127.128	DNS	Standard query response PTR 203-174-83-75.rev.ne.com.sg
45	152.543435	192.168.127.128	203.174.83.75	TCP	t128-gateway > http [SYN] Seq=0 Win=64240 Len=0 MSS=1460
46	155.458898	Vmware_c0:00:01	Vmware_68:91:81	ARP	Who has 192.168.127.128? Tell 192.168.127.1
47	155.460327	Vmware_68:91:81	Vmware_c0:00:01	ARP	192.168.127.128 is at 00:0c:29:68:91:81
48	158.467625	192.168.127.128	203.174.83.75	TCP	t128-gateway > http [SYN] Seq=0 Win=64240 Len=0 MSS=1460

```
▷ Frame 44 (127 bytes on wire, 127 bytes captured)
▷ Ethernet II, Src: Vmware_c0:00:01 (00:50:56:c0:00:01), Dst: Vmware_68:91:81 (00:0c:29:68:91:81)
▷ Internet Protocol, Src: 193.156.97.30 (193.156.97.30), Dst: 192.168.127.128 (192.168.127.128)
▷ User Datagram Protocol, Src Port: domain (53), Dst Port: 56789 (56789)
▷ Domain Name System (response)
```

```
0000  00 0c 29 68 91 81 00 50  56 c0 00 01 08 00 45 00   ..)h...P V.....E.
0010  00 71 50 32 00 00 7f 11  88 66 c1 9c 61 1e c0 a8   .qP2.... .f..a...
0020  7f 80 00 35 dd d5 00 5d  58 1a d8 46 81 80 00 01   ...5...] X..F....
0030  00 01 00 00 00 00 02 37  35 02 38 33 03 31 37 34   .......7 5.83.174
```

vmnet1: <live capture in progress>... Packets: 96 Displayed: 96 Marked: 0 Profile: Default

Figure B.3: Screen capture from `Wireshark` capturing network traffic during the malware analysis scenario in Chapter 5. The screen capture is too small to show any details, but is meant to give an overview over the interface only.



FileName: asprox.exe

Detected: Microsoft Visual C++ 6.0

Scan Mode: Normal

Entropy:

EP Check:

Fast Check:

☑ Stay on top

Figure B.4: Screen capture from `PEiD`'s results when scanning for utilised packer technologies on the sample from the malware analysis scenario in Chapter 5. `PEiD` detects the packer used as a `Microsoft Visual C++` packer.

Appendix C

Implementation Appendix

This appendix contains the script used to control the network in Chapter 5, and complete source code listings of the final system.

C.1 iptables script

This section contains the `iptables` script used in the malware analysis scenario from Chapter 5. The script is deployed on the host operating system, and ensures a strictly controlled set of network traffic is allowed out on the Internet. The script shown enables all DNS traffic (UDP traffic on port 53) and SSH traffic (TCP port 22) so it is possible to communicate remotely and secure with the machine. The script is shown below in Listing C.1.

```bash
#!/bin/bash

# Following commands will "flush" the previous iptables-rules
iptables -F
iptables -X
iptables -t nat -F
iptables -t nat -X

# Variable holding the IP-subnet for the malware network
VM_IP=192.168.127.0/24

# Variable holding the remote IP-address
PUB_IP=193.156.97.172

# Enable packet forwarding
echo "1" > /proc/sys/net/ipv4/ip_forward

# Always enable DNS requests
iptables -t nat -A POSTROUTING -s $VM_IP -p udp --dport 53 -o eth0 -j SNAT --to-source $PUB_IP

# Always enable SSH traffic
iptables -t nat -A POSTROUTING -s $VM_IP -p tcp --dport 22 -o eth0 -j SNAT --to-source $PUB_IP

```

```
# Check if we want to enable the rest of the network traffic.
# Commented value disables, otherwise enabled
#iptables -t nat -A POSTROUTING -s $VM_IP -o eth0 -j SNAT --to-source $PUB_IP
```

Listing C.1: Script containing `iptables` chains for filtering out traffic coming from the VM OS.

C.2 vmcom lite source code

`vmcom lite` is the entry point of the system. It is used to initiate commands on the virtual machine operating system, and is ensuring a cleanup after a successful execution. See Listing C.2 below for the complete Perl source code for `vmcom lite`. The program can be downloaded as a regular file from `https://sourceforge.net/projects/zeroplusone/`.

```
#!/usr/bin/perl

# Run in strict mode
use strict;

# Supply the user with warnings
use warnings;

# We need case-switch functionality in the parser
use Switch;

# We also need command line switch parsing
use Getopt::Long;

# Output can be colored
use Term::ANSIColor;

# Import the VIX API
use VMware::Vix::Simple;
use VMware::Vix::API::Constants;

# ##########################################################
#
# CONFIGURATION PART
#

my $host_name = 'https://127.0.0.1:8333/sdk';
my $host_port = 0;
my $host_username = 'km';
my $host_password = ''; # can be fetched from command line arguments
my $COMMAND = '~/zero+one/zero+one.py -f /tmp/uploaded_file -e'; # analyse command to run
my $vmx_location = '[mybook] linux_1/linux_1.vmx';
my $guest_os_username = 'km';
my $guest_os_password = ''; # can be fetched from command line arguments
my $result_file_source = '/tmp/results.txt'; # Source of result file from analysis (on guest OS)
my $result_file_destination  = '/tmp/results.txt'; # Destination of result file from analysis (on host OS)

# Do not wait more than TIMEOUT for each VM operation.
```

```perl
use constant TIMEOUT => 220;

#
# END CONFIGURATION PART
#
# ############################################################


# ############################################################
#
# MAIN PROGRAM
#

# Display a help screen if:
# a) user have supplemented the argument --help
# b) no arguments are specified

print_help() unless ($ARGV[0]);

# This is the Getopts switch hash, configuring the legal command line
# switches. The switches "only" and "scan" requires an argument each,
# respectively a string and an integer.
my %switches=();
GetOptions(
    "analyse=s"=>\$switches{analyse},
    "takesnapshot!"=>\$switches{takesnapshot},
    "hostpass=s"=>\$switches{h_password},
    "guestpass=s"=>\$switches{g_password},
    "help!"=>\$switches{help}
);

print_help() if $switches{help};

# Overwrite the default pw if another one is supplied
$host_password = $switches{h_password} if $switches{h_password};
$guest_os_password = $switches{g_password} if $switches{g_password};

# Fetch the subject file path
my $subject_file_location = $switches{analyse} if $switches{analyse};

# If there are any arguments left after getopts has parsed the
# arguments, some of them was not accepted by getopts. Inform the user
# accordingly.
foreach (@ARGV) {
    if (defined($switches{$_})) {
        print "$_ = $switches{$_}\n";
    }
    else {
        print "$_";
    }
}

# Prints some help.
sub print_help {
    print("Usage: vmcomlite.pl [-a] [-h] [-g]\n\n");
    print("-a FILE, --analyse=FILE\t\tAnalyse FILE\n");
    print("-ho PW, --hostpass=PW\t\tUse PW as password to the host OS\n");
    print("-g PW, --guestpass=PW\t\tUse PW as password to the virtual machine's OS\n");
    print("-t, --takesnapshot\t\tTake a snapshot of the virtual
        machine's state\n");
    print("\t\t\t\t(maximum one snapshot can be stored per virtual\n");
    print("\t\t\t\tmachine for VMware Server)\n");
    print("-he, --help\t\t\tThis help screen\n");
    die "\n";
}

# Aborts the current running operation and reports back to the user.
sub abort {
    my($text, $err) = @_;
    print "Error received, $text, $err ".GetErrorText($err);
    die $text, $err, GetErrorText($err), "\n";
}

# Initiate error and handles
my $err = VIX_OK;
my $host_handle = VIX_INVALID_HANDLE;
my $vm_handle = VIX_INVALID_HANDLE;
my $snapshot_handle = VIX_INVALID_HANDLE;

if ($switches{takesnapshot} or $switches{analyse}) {
    print("Initiating\n");
    $host_handle = connect_to_vm($err, $host_handle, $host_name, $host_port, $host_username, $host_password);
    print("Connected\n");
    ($host_handle, $vm_handle) = open_vm($err, $host_handle, $vm_handle, $vmx_location);
    print("vm opened\n");
    power_on_vm($err, $vm_handle);
    print("vm powered on\n");

    if ($switches{analyse}) {
        print("waiting for vm");
        wait_for_vm($err, $vm_handle);
        print("logging in\n");
        log_in_vm($err, $vm_handle, $guest_os_username, $guest_os_password);
        print("logged in\n");
        copy_file_to_guest($err, $vm_handle, $subject_file_location);
        print("copied file\n");
        run_vm_program($err, $vm_handle, $COMMAND);
        print("program ran\n");
        copy_file_to_host($err, $vm_handle, $result_file_source, $result_file_destination);
        print("copied file back\n");
        delete_result_file_in_guest($err, $vm_handle, $result_file_source);
        print("deleted file\n");
        print("reverting snapshot\n");
        reverte_snapshot($err, $vm_handle, $snapshot_handle);
    }
    elsif ($switches{takesnapshot}) {
        print("taking snapshot\n");
        take_snapshot($err, $vm_handle, $snapshot_handle, 0);
    }
}
else {
    print_help();
}


```

```perl
# Connects to virtual machine.
sub connect_to_vm {
    my($err,
        $host_handle,
        $host_name,
        $host_port,
        $host_username,
        $host_password) = @_;

    ($err, $host_handle) = HostConnect(VIX_API_VERSION,
        VIX_SERVICEPROVIDER_VMWARE_VI_SERVER,
        $host_name,
        $host_port,
        $host_username,
        $host_password,
        0, # options
        VIX_INVALID_HANDLE); # propertyListHandle

    abort("HostConnect() failed", $err) if $err != VIX_OK;

    return $host_handle;
}

# Opens a virtual machine handle
# Requires: connection to a virtual machine (connect_to_vm)
sub open_vm {
    my($err, $host_handle, $vm_handle, $vmx_location) = @_;

    ($err, $vm_handle) = VMOpen($host_handle, $vmx_location);

    abort("VMOpen() failed", $err) if $err != VIX_OK;

    return ($host_handle, $vm_handle);
}

# Power on a virtual machine handle
# Opens a virtual machine handle
# Requires: connection to a virtual machine (connect_to_vm)
#           an open virtual machine handle (open_vm)
sub power_on_vm {
    my($err, $vm_handle) = @_;

    $err = VMPowerOn($vm_handle,
        0, # powerOnOptions
        VIX_INVALID_HANDLE);  # propertyListHandle

    abort("VMPowerOn() failed", $err) if $err != VIX_OK;
}

# Wait until vmware tools is loaded in the virtual machine
# Requires: connection to a virtual machine (connect_to_vm)
#           an open virtual machine handle (open_vm)
#           a powered on virtual machine (power_on_vm)
sub wait_for_vm {
    my($err, $vm_handle) = @_;

    $err = VMWaitForToolsInGuest($vm_handle,
        TIMEOUT); # timeoutInSeconds

    abort("VMWaitForToolsInGuest() failed", $err) if $err != VIX_OK;
}

```

```perl
# Log into virtual machine guest operating system
# Requires: connection to a virtual machine (connect_to_vm)
#           an open virtual machine handle (open_vm)
#           a powered on virtual machine (power_on_vm)
#           vmware tools loaded (log_in_vm)
sub log_in_vm {
   my($err, $vm_handle, $guest_os_username, $guest_os_password) = @_;

    $err = VMLoginInGuest($vm_handle,
       $guest_os_username, # userName
       $guest_os_password, # password
       0); # options
    # vm suspend
    abort("VMLoginInGuest() failed", $err) if $err != VIX_OK;
}

# Copies a file from host operating system to guest operating system
# Requires: connection to a virtual machine (connect_to_vm)
#           an open virtual machine handle (open_vm)
#           a powered on virtual machine (power_on_vm)
#           vmware tools loaded (log_in_vm)
#           a user logged into the virtual machine guest OS (log_in_vm)
sub copy_file_to_guest {
   my($err, $vm_handle, $subject_file_location) = @_;

    $err = VMCopyFileFromHostToGuest($vm_handle,
       $subject_file_location,  # src name
       "/tmp/uploaded_file", # dest name
       0, # options
       VIX_INVALID_HANDLE); # propertyListHandle

    abort("VMCopyFileFromHostToGuest() failed", $err) if $err != VIX_OK;
}

# Delete the result file in the guest
# Requires: connection to a virtual machine (connect_to_vm)
#           an open virtual machine handle (open_vm)
#           a powered on virtual machine (power_on_vm)
#           vmware tools loaded (log_in_vm)
#           a user logged into the virtual machine guest OS (log_in_vm)
sub delete_result_file_in_guest {
   my($err, $vm_handle, $subject_file_location) = @_;
    $err = VMDeleteFileInGuest($vm_handle, "/tmp/results.txt");
}

# Run an arbitrary program on the guest operating system with the logged
# in users privilegies
# Requires: connection to a virtual machine (connect_to_vm)
#           an open virtual machine handle (open_vm)
#           a powered on virtual machine (power_on_vm)
#           vmware tools loaded (log_in_vm)
#           a user logged into the virtual machine guest OS (log_in_vm)
sub run_vm_program {
   my($err, $vm_handle, $command) = @_;
    print("Command er $command");

    $err =  VMRunProgramInGuest($vm_handle,
        $command,
        "/tmp/results.txt",
        0, # options
        VIX_INVALID_HANDLE);

    abort("VMRunProgramInGuest() failed", $err) if $err != VIX_OK;
}

# Copy a file from the guest operating system to the host
# Requires: connection to a virtual machine (connect_to_vm)
#           an open virtual machine handle (open_vm)
#           a powered on virtual machine (power_on_vm)
#           vmware tools loaded (log_in_vm)
#           a user logged into the virtual machine guest OS (log_in_vm)
sub copy_file_to_host {
    my($err, $vm_handle, $result_file_source, $result_file_destination) = @_;

    $err = VMCopyFileFromGuestToHost($vm_handle,
        $result_file_source,  # src name
        $result_file_destination, # dest name
        0, # options
        VIX_INVALID_HANDLE); # propertyListHandle

    abort("VMCopyFileFromGuestToHost() failed", $err) if $err != VIX_OK;
}

# Store the system state of a virtual machine
# Requires: connection to a virtual machine (connect_to_vm)
#           an open virtual machine handle (open_vm)
sub take_snapshot {
    my($err, $vm_handle, $snapshot_handle, $options) = @_;

    ($err, $snapshot_handle) = VMCreateSnapshot($vm_handle,
        undef, # name
        undef, # description
        $options, # options
        VIX_INVALID_HANDLE);

    abort("VMCreateSnapshot() failed", $err) if $err != VIX_OK;
}

# Revert to a previosly taken system state of a virtual machine
# Requires: connection to a virtual machine (connect_to_vm)
#           an open virtual machine handle (open_vm)
#           previosly stored snapshot (take_snapshot)
sub revert_snapshot {
    my($err, $vm_handle, $snapshot_handle) = @_;

    ($err, $snapshot_handle) = VMGetRootSnapshot($vm_handle,
        0); # index
    abort("VMGetRootSnapshot() failed", $err) if $err != VIX_OK;

    $err = VMRevertToSnapshot($vm_handle,
        $snapshot_handle,
        0,                          # options
        VIX_INVALID_HANDLE);        # property handle
    abort("VMRevertToSnapshot() failed", $err) if $err != VIX_OK;
}
```

Listing C.2: Complete source code for the `vmcom lite` program.

C.3 Zero+One source code

`Zero+One` are Python scripts running from the virtual machine operating system. They are performing the executions and analyses of samples. This sections covers the different scripts that combined is the `Zero+One` Python part of the final system.

C.3.1 Configuration

Configuration settings are separated from the other scripts to avoid tangling the functionality and configuration settings. Listing C.3 is shown below, and contains the configuration settings used during the implementation and testing phases from this thesis. Each configuration setting is self explanatory.

```
TIMEOUT = 200
MEMORY = True
MALWARE_FOLDER="/tmp/"
LAUNCHER_PATH="xvfb-run -a /path/to/malware_launcher.sh"
CONFIG_X11_DISPLAY = ":1"
TSHARK_LOGFILE = "/var/log/tshark"
HOME_NETWORK_ADDRESS = "192.168.127.128"
PE_SIGNATURE_PATH = '/path/to/packer_signatures.txt'
INTERESTING_CALLS = ["CreateMutex", "CopyFile", "CreateFile.*WRITE", "NtasdfCreateFile", "call shell32", "advapi32.RegOpenKey",
    "KERNEL32.CreateProcess", "shdocvw", "gethostbyname", "ws2_32.bind", "ws2_32.listen", "ws2_32.htons",
    "advapi32.RegCreate", "advapi32.RegSet", "http://",
    "^([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])",
    # Common debugger detection techniques
    "OutputDebugString",
    "FindWindow", # For OllyDbg, i.e.
    "IsDebuggerPresent"
    ]

JUNK_CALLS = ['trace:file:CreateFileW L"C:\\\\\\\\windows\\\\\\\\win.ini" GENERIC_READ FILE_SHARE_READ FILE_SHARE_WRITE  creation 3 attributes 0x80',
    'L"Software\\\\\\\\Microsoft\\\\\\\\Windows\\\\\\\\CurrentVersion\\\\\\\\ThemeManager', 'RegSetValueEx.*L"Cache"', 'RegSetValueEx.*L"History"',
    'RegSetValueEx.*L"Cookies"', 'L"Software\\\\\\\\Microsoft\\\\\\\\Windows\\\\\\\\CurrentVersion\\\\\\\\Explorer\\\\\\\\User Shell Folders"',
    'L"Software\\\\\\\\Microsoft\\\\\\\\Windows\\\\\\\\CurrentVersion\\\\\\\\Explorer\\\\\\\\Shell Folders"',
    'L"System\\\\\\\\CurrentControlSet\\\\\\\\Control\\\\\\\\NetworkProvider\\\\\\\\Order"'
    'L"Software\\\\\\\\Microsoft\\\\\\\\Windows NT\\\\\\\\CurrentVersion\\\\\\\\ProfileList"'
    'Software\\\\\\\\Wine', 'L"System\\\\\\\\CurrentControlSet\\\\\\\\Control\\\\\\\\Keyboard Layouts',
    'L"winedbg.exe"', '"RPCSSMasterMutex0x', 'c:!windows!',
    'L"System\\\\\\\\CurrentControlSet\\\\\\\\Control\\\\\\\\NetworkProvider\\\\\\\\Order',
    'L"Software\\\\\\\\Microsoft\\\\\\\\Windows NT\\\\\\\\CurrentVersion\\\\\\\\ProfileList',
    # The following SHOULD be removed
    'L"DejaVu', 'Font', 'L"__WINE_FONT_MUTEX__"', '() retval', '(TrueType)',
    'Wine', 'WindowMetrics', 'desktop.ini', 'Colors",00000000', '{9D20AAE8-0625-44B0-9CA7-71889C2254D9}',
    'Control Panel\\\\\\\\Colors'
    ]
```

Listing C.3: `Zero+One`'s configuration settings.

C.3.2 Libraries

The libraries are a collection of methods that performs the main functionality of `Zero+One`. The code found in the Python class shown below in Listing C.4 are partly taken from `Zero Wine` and implemented from scratch.

```
import os
import sys
import md5
import re # regular expression library
import linecache # to be able to read from the tshark logfile efficiently
import string

from config import * # Get all of the config data

"""
Execute a shell command. Command is the program to run including path
(will use PATH) returns everything the program writes to stdout as an
array.
"""
def execute_command(command):
    p = os.popen(command)
    return p.readlines()

"""
Fetches strings from the input file located at the path filename (or
current folder). Returns found strings in array form.
"""
def getStrings(filename):
    return execute_command("strings -n 4 %s" % filename)

"""
Returns PE signatures.
"""
def getSignatureForPe(pe):
    try:
        import peutils
        signatures = peutils.SignatureDatabase(PE_SIGNATURE_PATH)
        return signatures.match_all(pe)
    except:
        return None

"""
Todo
```

```
"""
def getHeaders(filename):
    try:
        import pefile
        pe = pefile.PE(filename)

        signatureInfo = getSignatureForPe(pe)
        peInfo = pe.dump_info()

        if signatureInfo:
            msg = "------------Signature------------\n\n"
            for match in signatureInfo:
                msg += "%s\n" % str(match[0])
            msg += "\n\n"
            msg += "".join(peInfo)
        else:
            msg = "".join(peInfo)

        return msg
    except:
        return "Error getting headers: %s" % str(sys.exc_info()[1])

def timeout_command(command, timeout):
    import subprocess, datetime, os, time, signal
    cmd = command.split(" ")
    start = datetime.datetime.now()
    process = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE)

    while process.poll() is None:
        time.sleep(0.2)
        now = datetime.datetime.now()
        if (now - start).seconds> timeout:
            print "Killing process %d" % process.pid
            os.kill(process.pid, signal.SIGKILL)
            os.waitpid(-1, os.WNOHANG)
            return None
        return process.stdout.readlines()
        return process.stdout.read()

def executeMalware(malware, folder, timeout, memory):
    os.environ["DISPLAY"] = CONFIG_X11_DISPLAY
    if memory:
        num = 1
    else:
        num = 0

    return timeout_command("%s %s %d" % (LAUNCHER_PATH, malware, timeout), timeout)

"""
Runs a malware sample. Supply path argument for malwareFile (or assume
current path), a timeout in seconds for the malware (we do not want to
run infinitely) and enable memory dumping (true or false). Memory
duming is NOT active this version, but keeped to keep consistency with
new Zero Wine versions.
"""
def analyzeMalware(malwareFile, timeout, memory):
    data = malwareFile.read()
    folderMd5 = md5.md5(data).hexdigest()
    folderName = MALWARE_FOLDER + folderMd5

    if os.path.exists(folderName) == 1:
        print "Folder already exists! File was previously analyzed?"
    else:
        os.mkdir(folderName)
        fileName = folderName + os.sep + os.path.basename(malwareFile.name)
        try:
            f = file(fileName, "wb")
            f.write(data)
            f.close()
        except:
            print "Error saving file (%s), exiting" % str(sys.exc_info()[1])
            sys.exit(0)

        try:
            buf = executeMalware(fileName, folderName + os.sep + "dump", timeout, memory)
        except:
            print "Error running malware!"
            print str(sys.exc_info()[1])
            sys.exit(0)

        if "".join(buf).find("wine: Unhandled page fault") > -1:
            print "One or more spawned processes crashed while running!"

        return buf, folderMd5, fileName

"""
Returns bool depending on wether or not the WINE debug line is junk
(true) or not (false).
"""
def junkCall(line):
    for junk in JUNK_CALLS:
        if re.search(junk, line):
            return True
    return False

"""
Filters out interesting calls from input array lines and return a new
array.
"""
def analyzeCalls(lines):
    prevLines = []
    for line in lines:
        for mcall in INTERESTING_CALLS:
            if re.search(mcall, line):
                if line not in prevLines and not junkCall(line):
                    prevLines.append(line)
    return prevLines


"""
Generates a signature for the Snort IDS based on network
traffic data located in the "lines" argument.
Each line should be structured as the following:
source_ip;port;dest_ip;port;ack;syn;fin;rst
Example:
```

```
84.208.124.161;3389;193.156.97.172;42657;1;1;;
193.156.97.172;42657;84.208.124.161;3389;;1;;

The granularity flag specifies how large the subnetworks in the
signature will be.  Granularity signifies the coverage for the *foreign*
destination only, and the $HOME_NET remains constant.  The granularity
level is assumed to be an integer from 1 - 32, or else it will default
to 32.

Examples:
32 - Returned signature uses a /32 CIDR notation mask (The exact same IP addresses - default)
24 - Returned signature uses a /24 CIDR notation mask.      255 IP-addresses covered (Class C subnet)
16 - Returned signature uses a /16 CIDR notation mask.   65535 IP-addresses covered (Class B subnet)
8 - Returned signature uses a /8 CIDR notation mask. 16777215 IP-addresses covered (Class A subnet)
"""
def generateSignature(lines, granularity):

    sourceIps = []  # List to include all source ip addresses found.

    destIps = []  # Similar with this one, including all destination ip
                    # addresses found
    sourcePorts = []
    destPorts = []

    for line in lines:
        substrings    = line.split(";")
        if len(substrings) != 8:
            print "Wrong syntax: "
        else:
            sourceIp      = substrings[0];
            sourcePort    = substrings[1];
            destIp        = substrings[2];
            destPort      = substrings[3];
            flagAck       = substrings[4];
            flagSyn       = substrings[5];
            flagFin       = substrings[6];
            flagRst       = substrings[7];

            """
            Create lists from the IP addresses
            """
            destIpAsList   = cleanList(re.split(r'(\d+)', destIp))
            sourceIpAsList = cleanList(re.split(r'(\d+)', sourceIp))

            """
            In case granularity is set to an invalid value
            """
            if not isinstance(granularity, int):
                try:
                    granularity = int(granularity)

                except ValueError:
                    granularity = 32 # Set the default value

                if granularity < 1 or granularity > 32:
                    granularity = 32 # Set the default value

            """
            Use the IP-list and transform the IP's into larger sets
                based on
            the granularity level set.
            """
            if len(destIpAsList) == 4:
                destIp    = ipToCidr(destIpAsList, granularity)
            else:
                print "Wrong length of destIp"
                print destIpAsList
            if len(sourceIpAsList) == 4:
                sourceIp = ipToCidr(sourceIpAsList, granularity)
            else:
                print "Wrong length of sourceIp"
                print sourceIpAsList

            """
            We do not want our own IP-address in the Snort signature.
                Replace
            it with Snort's $HOME_NET variable
            """
            if destIp.partition('/')[0] == HOME_NETWORK_ADDRESS:
                """String Snort recognises as the IP as the internal
                    home network
                variable."""
                destIp = "$HOME_NET"

            destIps.append(destIp)

            if sourceIp.partition('/')[0] == HOME_NETWORK_ADDRESS:
                sourceIp = "$HOME_NET"

            sourceIps.append(sourceIp)

            try:
                """We do not want to add the port if its is a
                    randomly generated source
                port upon initiation of the connection."""
                if flagSyn == 1 and flagAck == 0:
                    """Append a 0-int signifying an 'any' port"""
                    sourceIps.append(0)
                sourcePorts.append(int(sourcePort))
                destPorts.append(int(destPort))
            except ValueError:
                """Probably an UDP port"""
                sourceIps.append(0)

    """Create the signature and return it."""
    if len(sourceIps) > 1:
        sourceIps = ' [' + ','.join(set(sourceIps))[0:] + '] '
    else:
        sourceIps = ''.join(set(sourceIps))

    if len(destIps) > 1:
        destIps = ' [' + ','.join(set(destIps))[0:] + '] '
    else:
        destIps = ''.join(set(destIps))

    sourcePortRange = findHighestFrequentlyUsedPort(sourcePorts)
    destPortRange = findHighestFrequentlyUsedPort(destPorts)

    signature = "log" + sourceIps + sourcePortRange + " -> " + destIps + destPortRange

    return signature


"""
Finds the highest frequently used port in an array. Returns the port as a
string.
"""
def findHighestFrequentlyUsedPort(ports):
    """Dictionary used to find frequencies"""
    portFrequentDict = {}

    """Simply return 'any' if 0 is found in the list"""
    if 0 in ports:
        return 'any'
    """Iterate the ports and place their count in the dictionary"""
    for port in ports:
        if str(port) not in portFrequentDict:
            portFrequentDict[str(port)] = 1
        else:
            portFrequentDict[str(port)] += 1

    """Find the port with the highest frequency and return it"""
    highestPortFreq = -1
    for k, v in portFrequentDict.iteritems():
        if v > highestPortFreq:
            highestPortFreq = k

    return highestPortFreq

"""
Removes whitespace and dots (.) from the list/set and returns the
altered list
"""
def cleanList(list):
    for block in list:
        if block == " " or block == "." or block == "":
            list.remove(block)

    return list

"""
Changes an IP address in list format ( [ X, Y, Z, W ] ) to CIDR
format depending on the granularity specified. Returns the changed list.
"""
def ipToCidr(ipAsList, granularity):
    if len(ipAsList) != 4:
        print "Feil lengde pa liste:"
        print ipAsList
    else:
        if granularity <= 8:
            ipAsList[1] = "0";
        if granularity <= 16:
            ipAsList[2] = "0";
        if granularity <= 24:
            ipAsList[3] = "0";

    return string.join(ipAsList, '.') + '/' + str(granularity)
```

Listing C.4: Zero+One's library methods.

C.3.3 Entry point source

Zero+One's starting point is separated from the rest of the program since it contains the user interface to operate rest of Zero+One. Separating the entry point in a file makes it easy to switch out the user interface and keep the functionality if that is needed. The separation of files also improve the code's readability. Entry point source code is shown below in Listing C.5.

```
#!/usr/bin/python

import sys
"""Imports pefile also in this file to sustain compatability with new
zero wine version so it is not necessary to modify the packer
    generation
stub"""
import pefile

from config import *
from libraries import *

from optparse import OptionParser

parser = OptionParser(usage="%prog [-f]", version="%prog 1.0")
parser.add_option("-f", "--file", dest="filename",
    help="use FILE as sample input", metavar="FILE",
    type="string")

parser.add_option("-s", "--strings",
    help="scan FILE for strings", dest="strings",
    action="store_true")

parser.add_option("-p", "--packers",
    help="check for known packer signatures in FILE", dest="packers",
    action="store_true")

parser.add_option("-e", "--execute",
    help="execute FILE", dest="execute",
    action="store_true")

parser.add_option("-g", "--granularity",
    help="Use LEVEL as granularity when generating signatures",
    dest="granularity", metavar="LEVEL",
    type="string")

(options, args) = parser.parse_args()

"""Fetch the item to analyse."""
item = options.filename

"""Exit if no file argument is supplied."""
if not item:
    print "Missing FILE argument (Supply a file using \"--file PATH
        \")"
    sys.exit(0)
```

```

"""Try to open the file."""
file = open(item, 'rb')
if file:
    if options.strings:
        print getStrings(item)

    if options.packers:
        print getSignatureForPe(pefile.PE(item))

    if options.execute:
        #logfile = open(TSHARK_LOGFILE, 'r')
        #logfile.readlines() # Place file pointer to the end of the file

        """Executes the malware using malware_launcher.sh"""
        msg, strMd5, filename = analyzeMalware(file, TIMEOUT, MEMORY)

        """Not all of the lines are interesting. Start from the first
        instance of 'Starting process'"""
        idx = -1
        for line in msg:
            idx += 1
            if line.find("Starting process") > -1:
                break

        """Analyse all calls starting from index 'idx' in the msg array."""
        print analyzeCalls(msg[idx:])

        """Fetch data from the tshark log and generate a signature from
        it."""
        print generateSignature(execute_command("tshark -r /var/log/tshark -T fields -E separator=';' -e ip.src -e tcp.srcport -e ip.dst -e tcp.dstport -e tcp.flags.ack -e tcp.flags.syn -e tcp.flags.fin -e tcp.flags.rst"), options.granularity)
        #logfile.close()

    file.close()

else:
    print "Invalid file"
```

Listing C.5: Entry point for `Zero+One`.

C.3.4 Malware launcher

To execute samples from `Zero+One`, a shell script is used. The scripts runs a sample using `Wine`, but first sets the debug flags to allow parsing of API calls. Results are returned to `Zero+One` for processing. The script's source is shown below in Listing C.6.

```
#!/bin/bash

# Warn the user if no file is specified
if [ $# -lt 1 ]; then
        echo "Usage $0 malware.exe timeout"
        exit 1
```

```
fi


# Set wine debug flags
export WINEDEBUG=+relay,+dll,+loaddll,+file,+network,+wininet,+
    winsock,+ole,+msgbox,+ntdll

# Set default timeout to 5 seconds if nothing is specified
if [ $# -gt 1 ]; then
        timeout=$2
else
        timeout=5
fi

# Run wine and parse out important information
wine $1 2>&1 | egrep -v 'CriticalSec|SysLevel|Tls.etVal|window proc|
    CallWindowProc|ntdll.Rtl|Heap|user32.IsDialogMessage|KERNEL32.
    CreateEvent|KERNEL32.ResetEvent|KERNEL32.
    DisableThreadLibraryCalls|user32.GetMessage|user32.SendMessage|
    Ret  dialog proc|Call dialog proc|gdi32.SetBkMode|user32.
    DispatchMessage|user32.DefWindowProc|user32.BeginPaint|user32.
    EndPaint|gdi32.Delete|gdi32.Create|user32.CharNext|KERNEL32.
    lstrcpy|gdi32|KERNEL32.lstr|KERNEL32.Local|KERNEL32.Multibyte|
    user32.GetWindowLong|:Ret   |WindowLongA\(|KERNEL32.Global|
    KERNEL32.TlsAlloc|KERNEL32.CloseHandle|KERNEL32.
    WaitForMultipleObjects|user32.GetSysColor|user32.LoadCursor|
    user32.Register|user32.Char|KERNEL32.GetTickCount|KERNEL32.
    WriteFile\(00|KERNEL32.ReadFile\(00|Call KERNEL32.GetProcAddress
    \(|user32.InvalidateRect|rpcrt4.RpcString|KERNEL32.MultiByte|
    KERNEL32.WideChar|ole32.CoTaskMem|user32.GetDlgItem\(00|user32.
    RedrawWindow|user32.UpdateWindow|user32.PeekMessage|imm32.
    ImmProcessKey|user32.GetKeyboard|user32.PostMessage|Call KERNEL32
    .___wine|Call user32.LoadString|KERNEL32.SetLastError|KERNEL32.
    GetLastError|KERNEL32.FlsGetValue|Call KERNEL32._lread\(00|Call
    KERNEL32._llseek\(00|KERNEL32.SetFilePointer|rpcrt4.Ndr|rpcrt4.
    I_RpcGet|rpcrt4.NDR|rpcrt4.Ndr|ntdll.mem|msvcrt.malloc|ntdll.atoi
    |msvcrt.calloc|ntdll.str|ntdll._str|msvcrtl.realloc|msvcrt.free|
    winealsa.drv|msacm32.drv|Call KERNEL32.WaitFor|Call KERNEL32.
    ReleaseMutex|Call KERNEL32._lwrite\(00|^warn:' &

# Wait ...
sleep $timeout

# Ok, timeout is over. Kill the program if it still runs.
# However, we do not want to kill the Zero+One python script.
ps -edf | grep ^$USER | grep exe | grep -v python | grep -v grep |
    awk '{ print $2 }' | xargs kill -9 && echo "Killed, timeout"
```

Listing C.6: Shell script used by Zero+One to execute malware.

Appendix D

Testing Appendix

This appendix contains material from the testing phase of the system implementation.

D.1 Suspicious API calls returned

During the API call parsing test in Section 9.4.3 on page 110, a significant amount of suspicious calls were found. A counting program found 283 suspicious calls. Below are the network communication API calls shown, which are the same host/IP addresses found in the malware analysis scenario in Chapter 5.

```
001e:Call ws2_32.gethostbyname(7e2534cc "58.65.233.17")
trace:winsock:WS_gethostbyname "58.65.233.17"
001d:Call ws2_32.gethostbyname(7e37c64c "ns.uk2.net")
trace:winsock:WS_gethostbyname "ns.uk2.net" ret (nil)
001d:Call ws2_32.gethostbyname(7e37c754 "www.yahoo.com")
trace:winsock:WS_gethostbyname "www.yahoo.com"
001d:Call ws2_32.gethostbyname(7e37c85c "www.web.de")
trace:winsock:WS_gethostbyname "www.web.de"
```

D.2 Email correspondence

`http://www.offensivecomputing.net` is the web page where the sample used in the malware analysis scenario was downloaded from. During the scenario, the packer signature “`Microsoft Visual C++`” was found using the program `PEiD`. This is documented in Section 5.2.4 on page 59. During the system testing in Section 9.4.3 on page 110, a different packer type was found, namely “`Armadillo v1.71`”. This section contains emails sent to and received from `info@offensivecomputing.net` to underline the possibility of gaining multiple distinct “correct” results when scanning a sample for packer signatures. The received emails are made anonymous to prevent exposing the sender’s name. The name of the responder is replaced with “X Y”.

Initial email sent

The first email sent was meant to find out *why* `PEiD` and `Zero+One` resulted in two different packer signatures on the same sample. On the `http://offensivecomputing.net` web page, the sample has three packer signatures listed so a good place to start asking is the web page's contact point.

```
From: Kris-Mikael Krister <krismika@stud.ntnu.no>
Subject: Questions about Packer signatures in malware search
To: info@offensivecomputing.net
Date: Thu, 21 May 2009 11:06:36 +0200
Greetings,

I have two quick questions about the shown packer signatures when
performing a malware search on the offensivecomputing web page.

- A sample may show a multiple of packer signatures, does this
  mean that the sample is recursively packed?

- What does the numbers in the square brackets to the right of
  the signature name mean?  For example, the sample with md5
  1311f650aa1209a3ec962b6a9a38fc98 has three signatures listed,
  as follows:

 Microsoft Visual C++ [169,677]
 Armadillo v1.71 [197,789]
 Microsoft Visual C++ v5.0/v6.0 (MFC) [142,569]

Thank you

--
Kind regards,
Kris-Mikael Krister
krismika@stud.ntnu.no
```

First response received

The response received indicated the trigger of three different packer signatures when scanning the sample, but it is not clear which of the three actually was used to pack the sample.

```
From: X Y <XY@XY.com>
Subject: Re: Questions about Packer signatures in malware search
To: krismika@stud.ntnu.no
Date: Thu, 21 May 2009 09:22:52 -0600

That just means it matched all three signatures. In order to figure
out what the packer is you will need to manually reverse it. It's
possible that it's recursively packed but the check we do wouldn't
pick up on that.
```

Second email sent

The first response did not answer to what the square numbered brackets right of the signature name signifies, so a new email was sent.

```
From: Kris-Mikael Krister <krismika@stud.ntnu.no>
Subject: Re: Questions about Packer signatures in malware search
To: X Y <XY@XY.com>
Date: Thu, 21 May 2009 23:58:44 +0200

Thank you for your swift reply.

On Thu, May 21, 2009 at 09:22:52AM -0600, X Y wrote:
> That just means it matched all three signatures.
> [..]

Ok, I see. I guess that means the three packer signatures are
similar or even equivalent?

What about the numbers in the square brackets to the right of the
signature name?

For example, the sample with md5 1311f650aa1209a3ec962b6a9a38fc98
has the following three signatures listed on your web page, each
with a six digit number in square brackets.

Microsoft Visual C++ [169,677]
Armadillo v1.71 [197,789]
Microsoft Visual C++ v5.0/v6.0 (MFC) [142,569]

What do the numbers indicate?

Thanks again for your time and valuable help.

--
Kind regards,
Kris-Mikael Krister
krismika@stud.ntnu.no
```

Second response received

The last response and reply to the second sent email concluded the numbers in the square brackets are for internal use by administrators of the `offensivecomputing.net` server. Exactly which packer used is not derivable without initiating manual reverse engineering, but note the sender's opinion about the packer. The sender believes the packer type to be `Armadillo`, which is also the results from `Zero+One`.

```
From: X Y <XY@XY.com>
Subject: Re: Questions about Packer signatures in malware search
To: krismika@stud.ntnu.no
```

Date: Thu, 21 May 2009 19:10:06 -0600

Those numbers are the rule number in our file that matched. All a similar match means is that the signature is probably bad and is triggering on multiple items. For the one you posted, it seems like it is packed with Armadillo.