NTNU

Norwegian University of
Science and Technology

PowerScan: A Framework for Dynamic Analysis and Anti-Virus Based Identification of Malware

Thomas Langerud
Jøran Vagnby Lillesand

Master of Science in Communication Technology
Submission date: June 2008
Supervisor: Svein Johan Knapskog, ITEM
Co-supervisor: Christoph Birkeland, NorCERT
Lars Haukeli, NorCERT

Norwegian University of Science and Technology
Department of Telematics

Problem Description

This project will focus on aspects involving the automation of malware analysis. New malware samples, in the form in which they are distributed, typically have a detection rate of 30%, on average, among anti-virus products. By utilizing multiple products, and aggregating the results, one would increase the chances of detecting and identifying the sample at hand, and get more precise results as to what family of malware the sample belongs to. A growing number of malware dictates the need for automation in the analysis process.

This assignment is divided into two phases, and the students should focus on the first phase before moving on to the next.

Phase one will focus on automating surface analysis, in which the samples are scanned by anti-virus products without being run. One way of doing this is to use multiple simulated environments, where different anti-virus engines are installed on separate hosts, and results are aggregated in a centralized host.

If time permits, phase two will focus on automating aspects of dynamic analysis. A first step could build directly on the solution of phase one, by simply running the samples in environments where anti-virus products are already operating. Malware which is not detected by the techniques of phase one, may be detected when run. The next step would be to monitor what actions are being performed by the malware on the system when executed.

Assignment given: 15. January 2008
Supervisor: Svein Johan Knapskog, ITEM

Abstract

This thesis describes the design and implementation of a framework, PowerScan, which provides the ability to combine multiple tools in the analysis of a malware sample. The framework utilizes XML configuration in order to provide extendability so that new tools can be added post compilation without significant effort. The framework deals with three major types of malware analysis:

1. Surface scan of a sample with multiple on-demand anti-virus engines.

2. Execution of malware sample with real-time (on-access) anti-virus engines running in the background.

3. Execution of malware sample with different dynamic analysis solutions running. These tools may monitor the file system, registry, network or other aspects of the operating systems during execution.

The reasoning behind each of these phases are:

1. Using multiple scanners increases the probability that at least one of the vendors has created a detection signature for the given malware.

2. Executing the sample ensures that the malware code sooner or later will be written to disk or memory. This should greatly enhance detection rate for samples obfuscated using packers with encryption or other techniques, as the code at some point must be deobfuscated before execution. Additionally, on-demand scanners might use more advanced (and resource consuming) techniques when monitoring files executed on the system[1]. As for surface scanning, the odds of correctly identifying the malware also increases when using more scanners.

3. Although several good sandbox analysis tools exist, the solution presented here allows the malware analyst to choose which analysis tools to use - and even use different tool for analyzing the same aspect of the execution.

[1] As these files *definitely* will infect the system if not stopped, opposed to the case when the system is merely scanned, where files *may* infect the system if they are executed later (or may already have infected the system).

A thorough description of both design, implementation and testing is given in this report.

In addition to the implementation of the PowerScan framework described above, the theory behind all involved components is presented. This includes description of the Microsoft Windows platform (which is used for executing malware in PowerScan, and the one definitely most targeted by malware at the time of writing), virtualization (which is used in the virtual machines), anti-virus technology, malware hiding techniques and more. Surveys of the usability of different anti-virus engines and dynamic analysis tools in the framework have been conducted and are presented in the appendices, together with a comprehensive user guide.

Preface

The work on this thesis has been carried out at the Norwegian University of Science and Technology during the tenth semester of the authors' studies. The thesis constitutes 30 ECTS per person and the work has been carried out over a period of 20 weeks. Both authors are completing a master's degree in Telematics with specialization in information security under the study program "Communication Technology" at the Norwegian University of Science and Technology, Department of Telematics. Fields of study include communication systems and networks, computer technology, services, systems development and information security.

The issues addressed in this thesis have been suggested by the Norwegian Computer Emergency Response Team (NorCERT), a unit resident with the Norwegian National Security Authority (NSM) under the Department of Defence.

We wish to thank our supervisor Professor Svein Johan Knapskog at the Department of Telematics, Norwegian University of Science and Technology for his much valued assistance and guidance. We would also like to thank the staff at NSM/NorCERT. Special thanks are given to Head of the NorCERT department Dr. Ing. Christophe Birkeland and Senior Engineer Lars Haukli from the Incident Handling team for their feedback on the structure and technical details in the thesis. Thanks are also given to Chief Engineer Einar Oftedal and Senior Engineer Simen Støvland at NorCERT for help during the work.

Trondheim June 10th, 2008,

Jøran Vagnby Lillesand & Thomas Langerud

Contents

List of Figures

Code Listings

Abbreviations, acronyms and definitions

Anti-virus Efforts made to detect and prevent spreading of malicious code. In this definition of the term, malicious code refers to viruses, worms, Trojans and other code written with malicious intent.

API Application Programming Interface. An interface that an operating system, library or service provides to support requests from other computer programs.

API hooking The term hooking represents a fundamental technique of getting control over a particular piece of code execution. This can be used to alter an OS's or third party software's behavior without having access to the source code. The term API hooking then refers to performing hooking on a given API.

Backdoor A malicious subroutine within a program that allows adversaries to bypass security mechanisms without the knowledge of the users/owners/administrators.

CARO Computer Anti-virus Researchers' Organization.

CARO VNC A virus naming convention first adopted by CARO in 1991.

CLI Command Line Interface.

CME Common Malware Enumeration. A malware classification initiative under the non-profit MITRE organization.

DDoS attack Distributed Denial of Service attack.

DLL Dynamic Linked Library. Microsoft's implementation of the shared library concept.

DTD Document Type Definition. Used to describe a class of XML documents. Has been superseded by the XSD standard.

Dynamic analysis In the context of malware analysis, dynamic analysis refers to the technique of executing a suspected malware sample in order to analyze its behavior during execution.

Emulation To make some system duplicate the functionality of another system in such a manner that it appears to behave identical to the original one.

Exploit A rogue code action that takes advantage of a security flaw in a particular system or language.

Guest OS The operating system running on a virtual machine.

GUI Graphical User Interface.

Host OS The operating system running on the computer which hosts a hypervisor and virtual machines.

Hypervisor A hypervisor is a virtualization platform that makes it possible to run different guest operating systems on top of it. A native hypervisor (type 1) runs directly on top of the hardware, while a hosted hypervisor (type 2) needs a host operating system to run on top of.

IAT Import Address Table. A section of the Portable Executable file format, which is used as a lookup table when the executable code is calling an imported API function.

IDT Interrupt Descriptor Table. A data structure in the x86 architecture that describes correct responses to hardware interrupts, software interrupts and processor exceptions.

In the wild A malware sample that is said to be "in the wild" if a malware sample is spreading as a result of normal day-to-day operations on and between the computers of unsuspecting users.

JNA Java Native Access. A library used for accessing native libraries from Java.

JNI Java Native Interface. A programming framework used allowing Java code to call and be called by by native applications.

LDT Local Descriptor Table. A memory table used in the x86 processor architecture.

Malware Malware is short for malicious software. The term is used to refer to any software designed to cause damage to computing unit or network of computing units without the owners consent.

Malware family Malware is usually grouped into families based on similarities of its code. The fact that a new malware is classified into a particular known family conveys that some of the detection and disinfection methods for the other members of that family might be applicable to the new malware.

Malware variant Part of malware naming used to distinguish between different malware programs that belong to the same family.

Malware sample A single file infected with malicious code.

NE New Executable. The predecessor of the Portable Executable file format. Highly outdated, last used in 16-bit Windows versions.

On-demand scanner The part of an anti-virus engine that can be instructed to scan single files or system objects at scheduled times or as initiated by the user.

Packer An executable packer is a tool used to compress an executable file, which combines the compressed data and the unpacker code into one file. In malware context, a packer is often used to avoid the malicious code being detected by signature scanners. The packing process may also be combined with encryption, in which both the unpacking and decryption code must be included in the final file.

PE Portable Executable. A file format used for among other executables and DLLs in 32bit and 64bit versions of the Microsoft Windows operating system.

Platform A computer platform is a hardware architecture and/or software framework which allows other software to run.

Real-time scanner Also known as on-access scanner. The part of an anti-virus engine that provides automatic malware protection by scanning files and system objects as they are being used.

Static analysis In the context of malware analysis, static analysis refers to manual analysis of the malware code in order to understand its full behavior. In most cases, the malware code will be on the assembly level. This is by far the most time consuming analysis technique.

stderr Standard Error, a preconnected output channel between a program and the environment from which it is executed (typically the command prompt or a *nix shell). Used as default output location for error messages.

stdout Standard Out, a preconnected output channel between a program and the environment from which it is executed (typically the command prompt or a *nix shell). Used as default output location for non-error messages.

Surface analysis Refers to scanning the surface of a malware sample, looking for a given signature which identifies the sample. This is the most simple analysis technique and is employed by most anti-virus scanners.

Virtualization To create a virtual version of a computing resource. In this context, it is used to allow several instances of an operating system to run on a single physical computer.

VMM Virtual Machine Monitor. The portion of a Hypervisor responsible for managing a single virtual machine.

XML Extensible Markup Language. A specification to create custom markup languages to represent information in a human readable manner.

XSD W3C XML Schema Definition. XSD is based on XML and is a language used to describe a class of XML documents.

Zero-day/0-day virus A previously unknown virus or malware for which no specific anti-virus signatures are available.

Chapter 1

Introduction

> "The man who moves a mountain begins by carrying away small stones."

- Confucius

1.1 Introduction

Malicious code is a growing problem and reason for concern for every person, business or institution utilizing computer systems. Malicious code attacks can potentially do a lot of damage to computer systems, and in the worst case render business critical systems unavailable. This does not only have short term consequences in terms of loss of revenue during downtime and cost to restore a system to working state, but could also have longer term impacts on reputation which in turn could lead to loss of contracts, impact on stock prices and other indirect consequences. Critical systems do not only include systems directly interfacing customers, but also systems used by employees during conduct of business. As more and more systems are connected to the Internet, adversaries are given ever more possibilities to perform attacks. Disruption of business is only one of the consequences following an attack. Others include loss of confidential business information and sensitive personal information about customers or employees.

A clear trend in the recent years is that malware creation has become a more professionalized business. Up to recently, most malware was written by individuals with nothing or little to gain except status in relatively closed circles. Although these individuals were often knowledgeable enough, many lacked the skill and motivation[1] required to create truly disastrous malware. Now, with the dawn of malware released with political and economical intent combined with the ever growing connectivity of the world wide web, both the motivation and the possibility for creating wide-spreading malware is certainly present. This also means that one can expect that the whole malware creation scene will be more obfuscated than before[2]. There has also been example of both malware and networks of computers controlled by malware (so-called botnets) being put out for sale[3]. Another example of the increasing professionalism of malware creation and cybercrime in general is the reports of North Korea setting up a "hacker school"[4]. Also, several reports indicate that both malware creation[5] and infections are on a steady rise.

[1] Not every underground malware creator is interested of having law enforcement agencies world wide turn their attention towards her or him.

[2] Earlier it was not uncommon for malware source code to be published on public sites.

[3] See for example the article "Malware moves up, becomes commercial" at `http://arstechnica.com/news.ars/post/20060225-6264.html`.

[4] See for example `http://www.wired.com/politics/law/news/2003/06/59043`.

[5] See for example F-Secure's press release " F-Secure Reports Amount of Malware Grew by 100% during 2007" at `http://www.f-secure.com/f-secure/pressroom/news/fs_news_20071204_1_eng.html`.

All in all, there is every reason to expect that malware will continue to be an as big - if not bigger - problem in the future compared to today. Therefore it is increasingly important to come up with new and efficient ways to identify, analyze and classify malware. This thesis - and implementation - attempts to contribute toward making classification and identification of malware simpler and more accurate by providing a framework for effortlessly combining multiple anti-virus engines and analysis tools and techniques.

1.2 Problems to be addressed

When a new piece of malware is detected, it is desirable to analyze the sample as quickly and easily as possible to determine if it is a minor variation of an already known malware, or if it is something new. It is also desirable to analyze the behavior of malware as quickly as possible to be able to construct countermeasures or take mitigating actions. The problem description suggested in cooperation with NorCERT is the following:

> This project will focus on aspects involving the automation of malware analysis. New malware samples, in the form in which they are distributed, typically have a detection rate of 30%, on average, among anti-virus products. By utilizing multiple products, and aggregating the results, one would increase the chances of detecting and identifying the sample at hand, and get more precise results as to what family of malware the sample belongs to. A growing number of malware dictates the need for automation in the analysis process.
>
> This assignment is divided into two phases, and the students should focus on the first phase before moving on to the next.
>
> Phase one will focus on automating surface analysis, in which the samples are scanned by anti-virus products without being run. One way of doing this is to use multiple simulated environments, where different anti-virus engines are installed on separate hosts, and results are aggregated in a centralized host.
>
> If time permits, phase two will focus on automating aspects of dynamic analysis. A first step could build directly on the solution of phase one, by simply running the samples in environments where anti-virus products are already operating. Malware which is not detected by the techniques of phase one, may be detected when run. The next step would be to monitor what actions are being performed by the malware on the system when executed.

As malware is continuously evolving and malware creators still come up with new concepts, the proposed design will function as a framework rather than an actual implementation of a set of tools. While an implementation of a set of tools may quickly go out of fashion as new threats evolve, a framework should (hopefully) be usable in its current form for a longer period of time.

This is also important as it is unlikely that any of the authors will be able to provide support and updates of the implementation after its completion.

1.3 Own contribution

The main contribution given by this thesis is the implementation of an extensible and configurable system able to run various anti-virus engines and analysis tools and aggregate the results from these. Additionally, a thorough discussion of the implementation is given, highlighting all central components of the implementation. A significant number of tools, both anti-virus engines and dynamic analysis tools, have also been investigated for their usability with the solution offered here.

As can be seen from the related work presented in section 1.7, parts of what is achieved in the framework presented here has been done already. However, there are some aspects that separate PowerScan from existing solutions:

- PowerScan executes the malware sample with real-time anti-virus solutions running in the background. This assures that the code is executed and that the malicious code of the malware is decrypted (if it is encrypted in the first place) and written either to memory or disk. Additionally, this increases the chance that heuristic detection and other similar techniques are utilized. This will be discussed further later in this report.
- PowerScan is user extensible, meaning that it does not depend on author support for adding support for new tools and scan engines. This also means that when new tools are released in the future, it is possible to add them to the PowerScan framework. It also means that a malware analyst may set up an automated analysis environment based on his or her favorite tools.

These elements will be discussed in detail later in the report.

1.4 Document structure

As this report consists of both a theoretical and practical section, it can read in different ways, depending on the intent of the reader. If one is interested in theory regarding automated malware identification, analysis and classification on the Win 32 platform in general and Windows XP in particular, the theory/background chapter will prove a good starting point. For technical details about the implementation of the PowerScan framework, the implementation chapter should be read and so on.

The main sections of this report are:

Chapter 1 - this chapter - contains some customary elements such as an introduction to the problem domain, placement of this work in relation to others and a description of the used methodology. Additionally, an introduction to central malware concepts/types is given.

Chapter 2 gives an introduction to the different technologies/theory that are relevant to the implementation described here. This includes an introduction to the Win 32 platform, virtualization and problems with its usage in malware analysis, anti-virus technology, malware techniques for avoiding detection and so on. In general, it has been sought to describe any theoretical area directly relevant to the implementation.

Chapter 3 describes the technical details regarding the implementation in this thesis. It is further divided into three main sections, which deals with what was done *before*, *during* and *after* the implementation respectively. The first of these sections deals with planning of the implementation, such as requirement analysis, choice of high-level architecture and programming language and so on. The second section deals with how the implementation was done, and consists of package diagrams, class diagrams and so on. Textual descriptions of all packages and most classes are also given. The third and last section deals with evaluation of the implementation. This includes an analysis of which requirements were met, some testing, trade-offs and other simple analysis. Additionally, known weaknesses of the implementation are described.

Chapter 4 contains a conclusion of the work described here. This includes a summary of strengths and weaknesses of the implementation, suggestions for suitable usages of PowerScan and lessons learned. Additionally, some suggestions for further work are given.

Additionally, the following appendices are included:

Appendix A describes how to set up and use the PowerScan framework. This section can be used without reading the rest of the report.

Appendix B performs a survey of a significant number of anti-virus engines with respect to usage in the PowerScan framework.

Appendix C performs a survey of a significant number of dynamic analysis tools with respect to usage in the PowerScan framework.

Appendix D contains examples of configuration files used with the PowerScan framework.

Appendix E contains functional test documents for PowerScan.

1.5 Methodology

This section describes the methodology used during this thesis.

As the time span for writing this thesis is limited, some trade-offs must be made. The optimal methodology would be to identify the research front, assess the existing solutions and technology, then review all the components that may be used in the implementation before finally starting the the actual implementation work. This is not practically doable in the given time frame, so a more pragmatic approach must be used. After an intensive initial research period, a reasonable overview of the research front should be obtained, state-of-the-art existing tools briefly analyzed and a handful of fundamentally different ways of implementing the system identified[6]. Then, based on this research, some initial choices can be made, so that the design and implementation of the software solution can start immediately. This way, the software development can run in parallel with theoretical investigations of the components involved. This means that not all discovered weaknesses will necessarily be covered in the implementation, but they will still be described in the report.

The software development chosen is an agile-like approach, with focus on building a core system first, and then expanding it iteratively by adding the most important features first. Being an academic work, however, more emphasis is put on documentation than is usually the case in agile development. Design artifacts to be used include high-level architecture overview, pseudo code, package diagrams, class diagrams, sequence charts and test cases. For more information about the usage of documentation, see chapter 3.

[6]Fundamentally different meaning so different that the decision would have to be taken from “day 1” of implementation/design.

1.6 Introduction to malware concepts

This section introduces some common malware terms, and points out features that are specific to different kinds of malware. Instead of listing every single type of malware (where the distinction often lies in what they do - their purpose, such as is the case with dialers, spyware, adware and so on), the focus is on giving an introduction to the different ways malware might work. In this case, "work" refers to the fundamental way the malware functions and propagates, without focusing too much on the finer details of its intent. Here, malware is defined to be software which has malicious intent, meaning that the definition focuses on the intent of the programmer. This means that software which has bugs that can have harmful consequences is *not* considered to be malware under this definition.

Note that in reality, the different categories are mostly pragmatic; actual malware may overlap and display characteristics of several categories. For example, a virus can display worm-like characteristics by attaching itself to outgoing mails, while it still has all the normal characteristics of a virus. Similarly, a Trojan may display rootkit traits when trying to conceal itself from both the system itself and the user of a system. Hence, the following characteristics are just characteristics and not mutual exclusive classifications.

1.6.1 Viruses

The earliest recorded use of the term "computer virus" was by Frederick Cohen in 1984. His definition of a computer virus was a formal mathematical mode, which will not be discussed in any further detail here. This model led to a more informal definition, which is simple and easy to comprehend: "*A virus is a program that is able to infect other programs by modifying them to include a possibly evolved copy of itself*" [1].

The main feature of computer viruses is that they spread from file-to-file or file-to-sector on the machines they infect. Spreading between different computers is typically done by infected files being copied and executed, either with intent of an active attacker or unwittingly by an infected user. Other spreading mechanisms can be by using characteristics of other types of malware, such as network spreading worms. Some computer viruses have what is called a malicious payload, which is code that execute commands on comput-

ers such as deleting or corrupting files, disabling computer security software or performing other malicious deeds [2]. Note that the virus does not necessarily need to have a malicious payload, although in reality, they often have. A virus usually modifies a host file or boot area, often with a complete copy of the malicious code program [3]. This is, however, somewhat changed with the introduction of more advanced techniques such as multi-staged attacks. Multi-staged attacks is a term used to describe viruses that does not carry the entire malicious code in its payload, but instead downloads the required code from a web server or another resource when is has successfully spread to a new victim.

Viruses typically consist of some common components. The first is a replication mechanism that allows the virus to find new potential victims and then replicate itself onto that victim via some transport mechanism. This can for example could be binary executables or office documents containing macros. The second component is some sort of trigger mechanism that determines if and when the malicious code should be run. Trigger events can be date and time, presence of specific files, documents or availability of network access. Trigger events could also be the presence of some other malware, such as a back door, or a known weakness. Similarly, events and objects could also be used to determine that a virus should *not* be triggered, such as the existence of specific protection mechanisms, patches or even the presence of an analysis environments. Finally the virus contains the malicious payload, or some code to acquire the missing parts of the virus from a remote location.

Note that plain viruses are becoming less and less common. Most widespread malware today use some Internet-based vector, such as for example remote exploits or e-mail attachments.

1.6.2 Trojan horses

The term Trojan horse is in the computer security context used to describe malicious code that allows its creators to execute commands on the infected computer [2], often by opening network ports which allows the attacker to control the machine remotely. A Trojan is a piece of code which tries to hide its real purpose from the user. This is vital, as the Trojan often relies on a user executing the program that the malicious code is hidden within. Trojans are, opposed to viruses and worms, non-replicating. A Trojan does not necessarily modify or infect other program files, but may install additional programs [3]. The motivation of hiding the malicious code within an apparently legitimate

program is to trick the user into executing it and to hide its presence. Trojans can be further divided into two subcategories; pure Trojan programs created with the sole purpose of introducing the malicious code and Trojans hidden within other programs. The latter can for example be distributed using open-source applications, as attackers can download the original source code, modify it to install the Trojan, compile it and then publish it as desired [1].

1.6.3 Worms

Worms are self-contained self-spreading malicious programs. A worm uses its own program code to spread, and does (usually) not require any user interaction. A worm might attach itself to a piece of outgoing email or use a file transfer command between trusted systems. One way to discriminate between worms and Trojans is that Trojans try to masquerade as an innocent piece of program code whilst worms try to act invisibly in the system. While a Trojan often attempts to trick the user into executing the file containing the infection, a worm will often attempt to get in the "back door," by exploiting some bug or flaw in installed software. Worms do not, as opposed to viruses, require a host program or document to infect, but is self-contained. As some worms also employ file infection techniques, it is evident that the distinction between viruses and worms is not always clear. Worms can be thought of as a special subclass of viruses with main focus on spreading over networks [1].

1.6.4 Rootkits

A rootkit is a set of programs or code that allows a permanent, undetectable presence on a computer. The main task of the rootkit is most often to provide some unauthorized user access to perform operations as the root user (or equivalent). Most of the tricks and techniques employed by a rootkit are designed to hide code and data on a system. Rootkits comes in two major variants; user mode rootkits that do not employ kernel modifications but instead rely upon user-level services and kernel rootkits that employ modifications to the OS kernel itself [4]. Kernel rootkits have a better chance of concealing their presence, as user mode rootkits often can be detected by kernel mode defense mechanisms [1]. A rootkit generally does not reproduce itself automatically, but rather seeks to hide itself on the computer where it is installed. This is a distinction from more traditional forms of malware, such as viruses and worms, which often/always actively seek to reproduce

themselves. In the real world, rootkits are usually combined with other types of malware, as a technique used to hide these once a system is infected.

1.6.5 Bots

The term bot, as used to characterize malware, comes from the word robot. What is characteristic about bot malware is that it infects a computer, and does nothing without being given orders. Typically, a bot malware may, once it has been installed on the target, establish a connection to a web server, FTP server or IRC server and then wait for orders by the bot master. Bots obeying the same bot master are characterized as a botnet. Botnets are often used for malicious deeds such as coordinated DDoS attacks, sending spam or hosting phishing web sites. Botnets can consist of thousands of computers, and capacity on botnets have been reported to be for rent on underground markets.

1.7 Related work

This section describes software with similar functionality to that offered by PowerScan. In general, these other solutions are more specialized and less extendable. Typically, each solution presented below aims a solving a small subset within malware identification and classification. In the cases where the solutions are extendable, the authors are responsible for adding the extra functionality, making the users dependent on the authors for providing future updates.

1.7.1 OPSWAT Metascan

Metascan[7] is a solution made by OPSWAT that functions as a common front end for multiple anti virus engines. The program is able to scan files, archives and data streams. The output of the program is a list of results from the various AV engine with the suggested classification. The program also includes functionality to trigger the update of signature files for all engines simultaneously. The program includes queuing functionality so that multiple files can be scanned in sequence.

The current release of Metascan includes the following anti-virus engines by default:

- Norman Scan Engine
- MicroWorld scanning engine
- Eset scanning engine
- ClamAV
- eTrust Engine
- VirusBuster EDK
- F-Secure Anti-Virus Client Security

The Metascan program does not have the ability to perform any more advanced analysis than pure surface scan.

[7]Metascan, File Scanning API for Symantec/Norton, McAfee, Trend Micro, AVG - `http://www.opswat.com/metascan.shtml`.

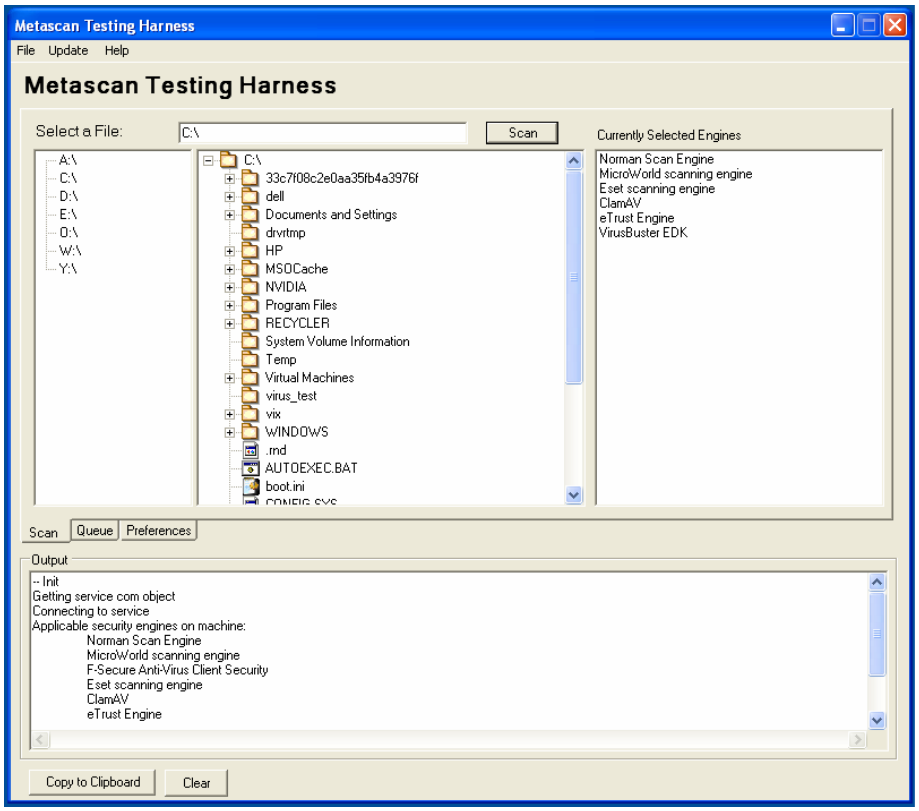


Figure 1.1: Metascan application GUI

In addition to the GUI front end, shown in figure 1.1, Metacan has an API that allows for automation and integration with other applications. The libraries are written in C++.

1.7.2 Hispasec Sistemas VirusTotal

Hipsasec Sistemas VirusTotal[8] is a free online solution which performs on-demand scan of submitted file samples with an impressive number of anti-virus engines; at the time of writing 32 different scanners are used for submitted samples.

VirusTotal can be invoked using e-mail or a web-based interface, shown in figure 1.2. The result is then presented as a list of the results reported by the different engines, shown in figure 1.3.

VirusTotal does not offer any advanced capabilities other than simple surface scan with the included engines; no dynamic analysis or sandboxed execution of the files is performed.

1.7.3 Norman Sandbox Malware Analyzer

Norman Sandbox Malware Analyzer[9] is a commercial product aimed at simplifying dynamic analysis of malware in executable files. The product reports back on

- Assumed malware category.
- Changes to the file system.
- Changes to registry.
- Network service usage, URLs and IRC servers contacted.
- Compression and executable type of the analyzed file.

[8]VirusTotal - Free Online Virus and Malware scan - `http://www.virustotal.com/`.

[9]Sandbox Malware Analyzer - `http://www.norman.com/microsites/malwareanalyzer/Products/analyzer`.

Figure 1.2: Hispasec Sistemas VirusTotal web-based interface.

File **eicar.com** received on **04.02.2008 12:51:24 (CET)**
Current status: **finished**
Result: **32/32 (100.00%)**

Compact Print results

Antivirus	Version	Last Update	Result
Authentium	-	-	EICAR_Test_File
F-Prot4	-	-	EICAR_Test_File
Avast	-	-	EICAR Test-NOT virus!!
AntiVir	-	-	Eicar-Test-Signature
AVG	-	-	EICAR_Test
CAT-QuickHeal	-	-	EICAR Test File
BitDefender	-	-	EICAR-Test-File (not a virus)
eSafe	-	-	EICAR Test File
DrWeb	-	-	EICAR Test File (NOT a Virus!)
Fortinet	-	-	EICAR_TEST_FILE
Ewido	-	-	Not-A-Virus.Test.Eicar
eTrust-Vet	-	-	the EICAR test string
NOD32v2	-	-	Eicar test file
McAfee	-	-	EICAR test file
Microsoft	-	-	Virus:DOS/EICAR_Test_File
Panda	-	-	EICAR-AV-TEST-FILE
Sunbelt	-	-	EICAR (v)
Norman	-	-	EICAR_Test_file_not_a_virus!
Ikarus	-	-	EICAR-ANTIVIRUS-TESTFILE
TheHacker	-	-	EICAR_Test_File
Prevx1	-	-	EICAR_Test
VirusBuster	-	-	EICAR_test_file
Sophos	-	-	EICAR-AV-Test
F-Secure	-	-	EICAR_Test_file_not_a_virus!
AhnLab-V3	-	-	EICAR_Test_File
FileAdvisor	-	-	High threat detected
ClamAV	-	-	Eicar-Test-Signature

Figure 1.3: Hispasec Sistemas VirusTotal web-based interface.

The application can also create a log of API usage and give the analyst access to the altered files from the Norman Sandbox Malware Analyzer virtual hard drive.

The file that is to be analyzed is executed within a confined environment which emulates a full computer with hardware access, network functionality, file system and registry. However, no instructions from within the sandbox are actually executed in the host system CPU.

Norman Sandbox Malware Analyzer also comes with a Pro version. This application include the same sandbox environment as described above, but also includes tools do perform real-time inspection of Win32 PE formatted executable files and examine instructions at arbitrary memory addresses, created threads and their status and the content of memory areas. It addition, the Pro version include the ability to set breakpoints in the memory to halt execution, and a Live Internet Communicator module that enables analysis of content retrieved from the Internet.

1.7.4 CWSandbox

[5] presents a sandbox solution, CWSandbox, for the analysis of Win32 programs. CWSandbox employs dynamic analysis using API hooking and DLL injection to hide its presence from the malware. It monitors system calls to be able to report on what operations the malware is performing in the operating environment, such as

- File accesses and changes.
- Changes to the Windows registry.
- Loaded DLLs.
- Virtual memory addresses accessed.
- Created processes.
- Network traffic; both destination and contents.
- Accesses to kernel services and device drivers.

The CWSandbox application consists of an executable that runs the sandbox environment and a DLL file that is hooked into the target program. The DLL is responsible for examining the API call parameters, calls the original function and examines the return value. The DLL then reports back to the sandbox using inter-process communication. The sandbox application generates the XML formatted result report.

1.7.5 TTAnalyze and Anubis

TTAnalyze, presented in [6], is another tool made to perform dynamic analysis of malware. It is constructed to analyze Win32 PE formatted executables, and is able to monitor both Windows WIN32 API calls and native kernel calls (these are explained later in this thesis). TTAnalyze focuses on being undetectable for the malware that is being analyzed, in order to prevent the sample from modifying its behavior during analysis. Some of the techniques utilized for this are usage of emulation software, Qemu, instead of a virtual machine, registry and context switch monitoring instead of API hooking and so on.

Support for the actual TTAnalyze implementation has now been discontinued, to be replaced by its successor project Anubis[10], which appears to be based on the same principles. Anubis claims to support the following support on their webpage[11]:

- Analysis of Registry Activities.
- Analysis of File Activities.
- Analysis of Process Acitivites.
- Analysis of Windows Services Activities.
- Analysis of Network Activities.
- Native API Aware Analysis.
- Unobtrusive analysis.
- Complete View of the PC System.

[10] Anubis: Analyzing unknown binaries - `http://analysis.seclab.tuwien.ac.at/`.

[11] Source: `http://analysis.seclab.tuwien.ac.at/features.php`. The page also contains a comparison of Anubis, Norman Sandbox and CWSandbox.

Chapter 2

Background and theory

> "Victorious warriors win first and then go to war, while defeated warriors go to war first and then seek to win."

- Sun-Tzu

2.1 Microsoft Windows architecture

The by far most widespread OS today is the 32-bit Windows platform[1]. For this reason, most malware is directed at this architecture. To be able to analyze what actions a given malware sample performs, it is useful to have some knowledge about the platform on which it is running. This section gives an introduction to the Windows 32-bit architecture, including the most important APIs, which are essential to understand in order to analyze malware behavior and interaction with the OS. This introduction to the APIs aims to give an understanding of the principles used by dynamic analysis tools on the Win32 platform, where an important technique is hooking of the relevant OS APIs. The concept of hooking is introduced in section 2.10. The material presented here is mostly gathered from the Microsoft Developer Network[2].

Generally, an Application Programming Interface (API) is a means for an operating system, library or other component to expose its services to other computer programs. The *Win32 API* gives applications the possibility to execute services offered by the operative system, through a collection of system calls made available to user mode applications. The core of the Windows 32-bit architecture is made up of a number of DLL files, each offering a set of services. The set of DLL files called the Win32 API makes up the core system which is the commonly used (and intended) interface toward the kernel. These DLL files include[3]:

kernel32.dll handles processes, threads and file systems.

user32.dll contains most of the user interface functionality.

gdi32.dll contains most of the functionality for drawing graphics.

advapi32.dll contains registry and security related functions, service management and system start/stop/restart.

wininet.dll contains functionality for offering network and Internet related services and can be used to for example managing FTP and HTTP sessions.

[1]See for example w3schools' statistics at `http://www.w3schools.com/browsers/browsers_os.asp`.

[2]Especially from web pages starting at Win32 and COM Development - `http://msdn.microsoft.com/en-us/library/aa139672.aspx`.

[3]From Microsoft TechNet: Windows Architecture - `http://www.microsoft.com/technet/archive/ntwrkstn/evaluate/featfunc/winarch.mspx`.

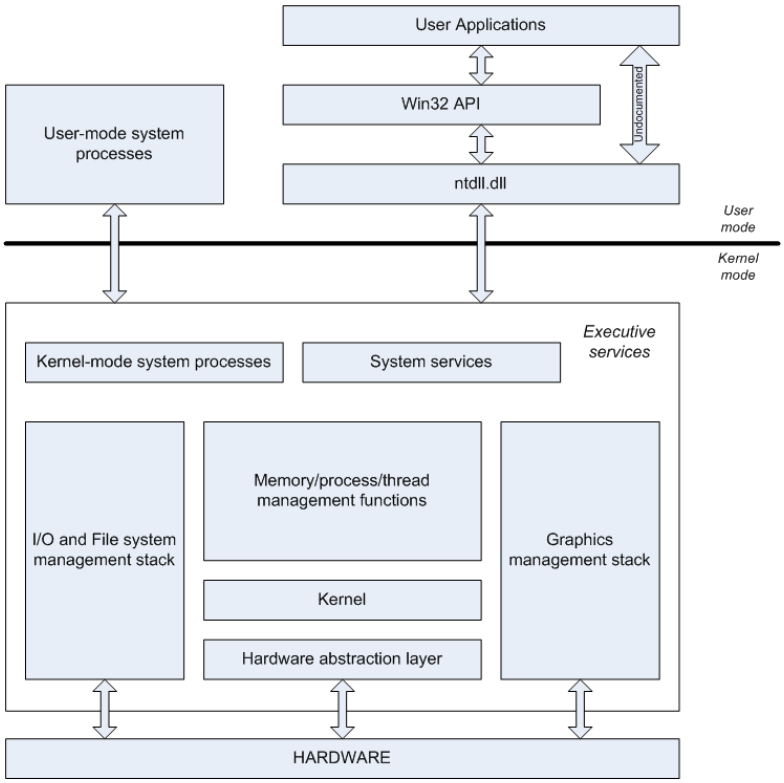


Figure 2.1: Microsoft Windows architecture overview, adapted from [7].

These libraries offer services by making calls to the kernel library *ntdll.dll*. It is also possible to perform direct calls to *ntdll.dll* itself, although it is largely undocumented[4]. The lack of documentation makes the *ntdll.dll* library especially interesting for malware authors, as these undocumented calls may be exploited to avoid detection and perform other malicious operations. This bypassing of the intended API and an overview of the Microsoft Windows architecture is shown in figure 2.1.

Other important files include *comdlg32.dll* (Common Dialog Box lib), *comctl32.dll* (Common Control lib) and *shlwapi.dll* (Windows shell).

By monitoring functions exposed by these APIs, it is possible to analyze interactions between an application and the operating system. The following sections look into some important system function calls that may be used to influence and use the registry, file system, processes and network interface re-

[4]At least from an official point of view - in practice, many of the functions offered have been attempted documented by the community.

spectively. It is worth noticing that the API is not identical across Windows NT, 2000 and XP. Functions might have the same name, but could possibly return other values or require different arguments [3]. Some operating systems support the Win32 API without implementing all the functionality. An example of this is the Windows CE OS, developed for use on PDAs, which supports both the Win32 API and the PE file format, but still lacks implementation of a significant number of system calls. For this reason, some malware will only function on certain Win32 OS versions. In malware naming conventions, this is indicated by prefixing the names of malware by Win95, WinNT and so on. The Win32 API is (at least) found the Windows 95, 98, ME, NT, 2000, XP, 2003, CE and Vista versions. As 64 bit architectures are introduced, there is also a new API collection called Win64 entering the field. This should however not introduce any major changes with respect to malware analysis.

The following sections present several critical parts of the Win32 API and for each of them a number of especially relevant function calls which should be monitored when analyzing malware. For more information about techniques for monitoring function calls, refer to the introduction on API hooking in section 2.10. Lastly, an introduction to windows file formats is given.

2.1.1 The registry

The Windows registry is a database native to Windows used by the OS and other applications to store configuration data[5], as an alternative to using individual .ini configuration files. This opportunity is used by almost every Windows application [3]. The Windows 32 registry is organized as 5 trees located under a common "My Computer" element, as shown in figure 2.2. Each node in a tree is called a key, and a key might have sub-keys and data entries associated with it. A *hive* is a group of keys and data entries, starting at a predefined node. A hive is stored in its own file, located in the `%SYSTEMROOT%\system32\config` folder, except the files for the hive *HKEY_CURRENT_USER* which is located in `%SYSTEMROOT%\Profile\$¬ username`. Among the functions using the registry is the Windows startup feature; applications wishing to be executed at every boot need to write their path into one of several possible keys. Another central feature using the registry is file type association, which decides which application is to be

[5]Microsoft Developer Network: Registry (Windows) - `http://msdn2.microsoft.com/en-us/library/ms724871(VS.85).aspx`.

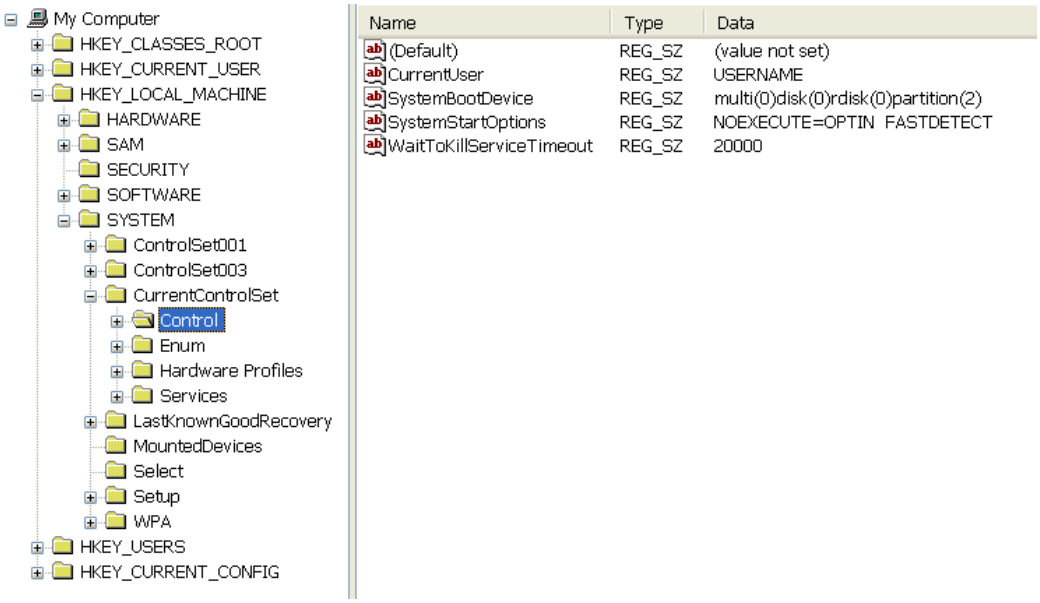


Figure 2.2: Screenshot of the Windows registry, as presented in the Microsoft Registry Editor application.

launched for files with a given extension. These are typical features which malware may exploit in order become resident on an infected machine and modify its behavior.

The Win32 hives[6] can be seen in the following table:

Hive	*Contents*
`HKEY_CURRENT_CONFIG`	Info about current HW profile
`HKEY_CURRENT_USER`	Env. variables, system and application preferences.
`HKEY_LOCAL_MACHINE\SAM`	Security Access Manager. Virtual hive that contains security information, user permissions and passwords.
`HKEY_LOCAL_MACHINE\SECURITY`	Includes SAM as a subkey. Dedicated to the security of the computer.
`HKEY_LOCAL_MACHINE\SOFTWARE`	Contains keys written by applications installed on the system.
`HKEY_LOCAL_MACHINE\SYSTEM`	Information about system hardware drivers and services.
`HKEY_USERS\.DEFAULT`	Default configuration for new users
`HKEY_LOCAL_MACHINE\HARDWARE`	Information about drivers and other system properties related to hardware.
`HKEY_CLASSES\ROOT`	Contains among other things the file extension associations. Combined view of two sources, namely `HKEY_LOCAL_MACHINE\Software\¬ Classes` and `HKEY_CURRENT_USER\Software\Classes`.

The following are some Win32 API function calls that can be used to manipulate the registry, and therefore would be interesting to monitor for a malware analyst:

[6]Microsoft Developer Network: Predefined Keys (Windows) - `http://msdn2.microsoft.com/en-us/library/ms724836(VS.85).aspx`.

From the *advapi32.dll*:

RegOpenKeyEx opens the specified registry key.

RegOpenKeyTransacted opens the specified registry key and associates it with a transaction.

RegCreateKeyEx creates the specified registry key. If the key already exists, the function opens it.

RegCreateKeyTransacted creates the specified registry key and associates it with a transaction. If the key already exists, the function opens it.

RegSaveKey saves the specified key and all of its subkeys and values to a new file, in the standard format.

RegSaveKeyEx saves the specified key and all of its subkeys and values to a registry file, in the specified format.

RegLoadKey creates a subkey under `HKEY_USERS` or `HKEY_LOCAL_MACHINE` and loads the data from the specified registry hive into that subkey.

RegCloseKey closes a handle to the specified registry key.

RegDeleteKey deletes a subkey and its values.

RegDeleteTree deletes the subkeys and values of the specified key recursively.

RegDeleteKeyEx deletes a subkey and its values from the specified platform-specific view of the registry.

2.1.2 The file system

The file system provides applications access to the permanent storage of the computer. Windows XP, Vista, Server 2003 and 2000, which are the most relevant versions today, support the FAT16, FAT32 and NTFS file systems. As FAT16 and FAT32 has a maximum volume size of 4GB and 32GB respectively[7], NTFS is usually the preferred file system. File systems supported by Windows contains the following logical entities; volumes, partitions, directories and files. A volume is the highest entity in the hierarchy, and contains one or more partitions. A partition contains a file system, which is a collection of directories and files[8]. A directory is a logical entity which is a collection of other directories and files, while a file is a collection of data belonging together. Most malware attempts to manipulate the file system for purposes such as hiding itself, distribution/infection and much more.

[7]The theoretical limit of FAT32 is actually 2TB, but Windows XP only allows formatting of drives up to 32GB.

[8]Microsoft Developer Network: File Systems (Windows) - `http://msdn2.microsoft.com/en-us/library/aa364407(VS.85).aspx`.

Directory management

A directory is a logical collection of directories and files. Directories can be manipulated through the Win32 API using the following functions:

From *kernel32.dll*:

CreateDirectory creates a new directory. Implemented as ANSI version (CreateDirectoryW) UTF-8 version (CreateDirectoryA).

CreateDirectoryEx creates a new directory with the attributes of a specified template directory. Implemented as ANSI version (CreateDirectoryExW) and UTF-8 version (CreateDirectoryExA).

RemoveDirectory deletes an existing empty directory. Implemented as ANSI version (RemoveDirectoryW) and UTF-8 version (RemoveDirectoryA).

File management

kernel32.dll defines the following file operations:

From the *kernel32.dll*:

FindFirstFile searches a directory for a file or subdirectory that matches the indicated file name.

CreateFile creates or opens a file, file stream, directory, physical disk, volume, console buffer, tape drive, communications resource, mailslot, or named pipe. Implemented as ANSI version (CreateFileA) and UTF-8 version (CreateFileW).

DeleteFile deletes an existing file. Implemented as ANSI version (DeleteFileA) and UTF-8 version (DeleteFileW).

OpenFile creates, opens, reopens, or deletes a file. Note: Only use this function with 16-bit versions of Windows. For newer applications, use the CreateFile function.

ReOpenFile reopen an already open file using new access privileges, flags or sharing mode.

ReadFile reads data from a file or I/O device, starting at the position that the file pointer indicates. Can be used for both synchronous and asynchronous operations.

ReadFileEx reads data from a file or I/O device asynchronously. Lets the calling application perform other actions during a file read operation.

CopyFile copies an existing file to a new file. Implemented as ANSI version (CopyFileA) and UTF-8 version (CopyFileW).

CopyFileEx same as CopyFile, but asynchronous.

MoveFile moves an existing file or a directory, including its children. Implemented as ANSI version (MoveFileA) and UTF-8 version (MoveFileW).

MoveFileEx moves an existing file or directory, including its children, with various move options.

WriteFile writes data to the specified file or I/O device at the position specified by the file pointer. This function is designed for both synchronous and asynchronous operation.

WriteFileEx writes data to the specified file. Reports its completion status asynchronously, calling a specified callback routine when writing is completed or canceled and the calling thread is in an alertable wait state.

2.1.3 Processes

A running instance of an executable program is referred to as a *process*. A process consist of one or more *threads*, which is an atomic unit when it comes to processor time allocation. All threads that run in the context of a given process share the same address space, security context and environment variables [8]. A process executing in user mode, also known as unprivileged mode, is restricted from making certain system calls without making a call to functions running under strict control in privileged mode. Some operating systems have three levels of privilege; kernel mode, system mode and user mode. Windows only has two of these, kernel and user[9]. The thread operations are essential to understand the behavior of more complex malware, which may use threading and/or try to interact with the address space of other processes.

These are some of the critical system calls when it comes to process handling that should be considered monitored when analyzing malware:

From the *kernel32.dll*:

CreateProcess creates a new process and its primary thread. The new process runs in the security context of the calling process. Implemented as ANSI version (CreateProcessA) and UTF-8 version (CreateProcessW).

CreateProcessAsUser Creates a new process and its primary thread. The new process runs in the security context of the user represented by the specified token. Implemented as ANSI version (CreateProcessAsUserA) and UTF-8 version (CreateProcessAsUserW).

OpenProcess opens an existing local process object.

CreateRemoteThread creates a thread that runs in the virtual address space of another process.

CreateThread creates a thread to execute within the virtual address space of the calling process.

ExitProcess ends the calling process and all its threads with an exit code.

ExitThread ends the calling thread.

TerminateProcess terminates the specified process and all of its threads without giving an exit code.

[9]Microsoft TechNet: Windows Architecture - `http://www.microsoft.com/technet/archive/ntwrkstn/evaluate/featfunc/winarch.mspx`.

2.1.4 Networking

The file I/O functions[10] provide the basic interface for opening and closing a communication resource handle and for performing read and write operations[11]. This means that when a process wishes to communicate through a communication device, it can perform a call to *CreateFile* specifying COM1 or LPT1 or another valid device name, and then write to the returned handle. The process can use the *DeviceIoControl*-call to send control codes to a device. For more information about the functions that are identical for network and file operations refer to the file management discussion in section 2.1.2. Several types of malware perform operations against the local network and/or the Internet in order to infect other computers, receive updated malware code or interact with its creators. In order to analyze this behavior, it is essential to monitor the calls from the malware to the network API.

The following system calls should be monitored:

From *kernel32.dll*:

CreateFile same as for file management.

ReadFile same as for file management.

ReadFileEx same as for file management.

WriteFile same as for file management.

WriteFileEx same as for file management.

DeviceIoControl sends a control code directly to a specified device driver, causing the device to perform the corresponding operation.

In addition the low level I/O device communication API functions listed above, the more high-level *WinInet API* contains many functions for HTTP and FTP networking (as these are simpler to use, they are more likely to be used by malware).

The following system calls should be monitored:

From *wininet.dll*:

InternetConnect opens an File Transfer Protocol (FTP), Gopher, or HTTP session for a given site. Takes port number and host name or IP address, username and password as parameter. Implemented as ANSI (InternetConnectA) or UTF-8 (InternetConnectW).

[10] CreateFile, CloseHandle, ReadFile, ReadFileEx, WriteFile and WriteFileEx.

[11] Microsoft Developer Network: About Communication Resources (Windows) - `http://msdn2.microsoft.com/en-us/library/aa363140(VS.85).aspx`.

InternetOpenUrl opens a resource specified by a complete FTP, Gopher, or HTTP URL. Implemented as ANSI and UTF-8.

FtpFindFirstFile searches the specified directory of the given FTP session. Implemented as ANSI and UTF-8.

FtpOpenFile initiates access to a remote file on an FTP server for reading or writing. Implemented as ANSI and UTF-8.

FtpGetFile retrieves a file from the FTP server and stores it under the specified file name, creating a new local file in the process. Implemented as ANSI and UTF-8.

FtpPutFile stores a file on the FTP server. Implemented as ANSI and UTF-8.

InternetWriteFile writes data to an open Internet file.

InternetReadFile reads data from a handle opened by the InternetOpenUrl, FtpOpenFile, GopherOpenFile, or HttpOpenRequest function.

HttpOpenRequest creates an HTTP request handle. HttpOpenRequest takes an HTTP session handle created by InternetConnect and an HTTP verb, object name, version string, referrer, accept types, flags, and context value. The HTTP verb is a string to be used in the request. Common HTTP verbs used in requests include GET, PUT, and POST.

InternetSetOption sets various internet options, including username and password, retries and timeouts. Implemented as ANSI and UTF-8.

2.1.5 Windows file formats

Executable and DLL files for the Windows 32-bit system are usually in the Portable Executable (PE) file format [9]. This file format is also used by system components, device drivers, screen savers and ActiveX controllers. For the 64 bit architectures, the file format is called PE+. The PE file format has a data structure that contains information intended for the Windows bootloader.

Older versions of Windows (16 bit) used the New Executable (NE) file format. A NE file contains an old MZ DOS EXE header for compatibility, followed by a new NE header [1]. An important difference between the NE and PE file formats in this context is that the 16 bit version requires that functions calling an external DLL must be relocated in memory. This is a time-consuming operation that the system loader must perform before the code is executed. When executing PE files, this relocation is not necessary because the PE file has a data structure called an Import Address Table (IAT) which is used to hold the addresses where the external functions are located. IAT is a feature that is often exploited by malware, for example by modifying the IAT in such a way that it transfers execution control to malicious code [1]. The header of the PE files contains, among other, fields describing:

- For which CPU architecture the file is intended.
- The size of the application given in number of sections.
- Characteristics, for example whether the file is an executable or a library.
- Offset of entry point, which is where the executable code begins.
- A checksum of the file; in executables this is often set to "0" (not used), while in a DLL or a driver it is set to the actual checksum.

After the header, the files have a table that describes the different sections. Sections in PE files are used to discriminate between code, data, global data, debug information and so on. These sections are important in this context, as malware often modify existing or add new sections containing the malware code. Two important sections are the ones called ".idata" and ".edata", which contain the import and export tables respectively. The first one contains a list of imported APIs[12], the latter all API functions made available to other executables. The section table also has a field called characteristics, which holds labels for the sections describing if they are readable, writable, executable and whether they contains data, code or other information.

[12]In other words, this is the IAT described earlier.

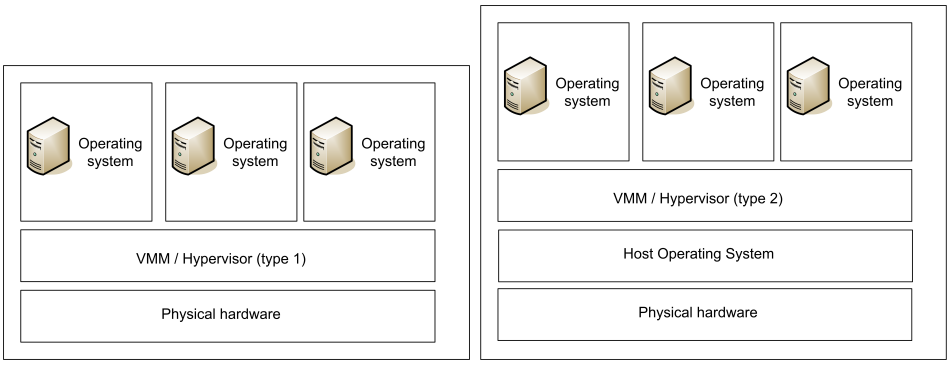
2.2 Virtualization

The main reason for using virtualized environments today is that it offers significant advantages when it comes to maintenance, performance and security when operating multiple services on one physical machine. For example, it offers the possibility of hosting both e-mail and several web servers on one physical machine without a security breach in one affecting the others. Similarly, virtualization can also be used to set up a secure environment in which it is possible to handle malicious code without any significant risk of the code spreading. In the implementation described in this thesis, VMware Server is used for this purpose.

Virtualization is a relatively complex area, which covers several technologies in both software and hardware. Oxford English Dictionary defines virtualization as "*to create a virtual version of (a computing resource or facility)*". Another more concrete definition is given in [10], stating that "*virtualization is a framework or methodology of dividing the resources of a computer into multiple execution environments, by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service, and many others*".

Virtualization can be implemented in several ways and on several system layers. The different layers describe which resources are virtualized and thus what abstraction is offered to the code running in the virtualized environment. It is possible to provide virtualization of hardware, OSs, applications and high-level runtime environments, such as the Java virtual machine. One of the most important features of virtualized systems is partitioning, which means for example that several operating systems can co-exist on a single physical system without interfering or even being aware of each other. Another important advantage is the possibility to create a pool of computing resources that can be re-partitioned when needed.

Hardware level virtualization, which is what is used in PowerScan and thus will be discussed here, is about presenting a virtual replica of hardware resources to several operating systems. To achieve virtualization on the hardware level, is necessary to insert a virtualization layer between the operating system and the physical hardware. This virtualization layer is called a hypervisor. The hypervisor typically contains some base functionality, such as scheduling and support for hardware access. It also has containers that



(a) Hypervisor type 1: Running hypervisor directly on top of hardware.

(b) Hypervisor type 2: Running hypervisor on top of host operating system.

Figure 2.3: Different techniques for implementing virtualization, inspired by a figure in [11].

guest OSs[13] can run within. Such a container is often referred to as a Virtual Machine Monitor (VMM). Hypervisors can be divided into two major groups:

Hypervisor type 1 uses a technique where the hypervisor runs directly on top of the hardware. With this technology, the hypervisor must support tasks that are usually performed by the host OS.

Hypervisor type 2 uses a technique where the hypervisor runs on top of a host OS[14], which provides access to the underlying hardware. (this is also called middleware virtualization or hosted virtualization) [11].

These two architectures are illustrated in figure 2.3.

The following describes how access to the different physical resources are administered when talking about virtualization, based on information found in [12] and [13].

[13]The term "guest OS" refers to an operating system running inside a virtualized environment, on a virtual machine.

[14]The host OS is an ordinary OS such as Windows or Linux running on top of the hardware.

2.2.1 CPU virtualization

CPU virtualization deals with how guest OS's access to the CPU resources on the physical machine is handled. One difference between hypervisor implementations is where the instructions of guest OSs are executed. Instructions may be executed exclusively in the virtual machine software or there may be a separation where some instructions are emulated in software while others are executed in the physical CPU[15]. This is a typical safety vs. performance trade-off, as emulation introduces significant overhead, while allowing critical operations to execute directly on the CPU may allow an application running on the virtual machine to break out of the virtualized environment and take control over the host machine. Allowing the guest OS access to the processor while the VMM maintains complete control is a relatively complex task [14].

One major issue when doing CPU virtualization is that many processor architectures differentiate privileged and non-privileged instructions. The standard x86 architecture family has two modes, Ring 0 ("privileged mode" or "kernel mode") and Ring 3 ("unprivileged mode" or "user mode"). Ring 0 is usually used by the OS kernel and device drivers running in kernel mode, Ring 3 by the applications. Trying to execute privileged instructions from a program running in unprivileged mode will cause a general protection fault, indicating that the operation failed due to lack of privilege. Privileged instructions can for example be operations manipulating the OS's memory area. Operating systems assume that they are able to perform these privileged operations, although this will not be the case for OSs running on virtual machines. Guest OSs are executed as applications on top of the host OS/hypervisor. The x86 architecture operating systems were originally designed to have complete ownership of the hardware, and were until 1998 thought to be impossible to virtualize [13].

As mentioned, privileged instructions issued in non-privileged context cause general protection faults, which can be caught by the hypervisor. The hypervisor can then emulate the system call and return the result to the guest OS, a technique known as "trap-and-emulate". The problem with the x86 architecture is, however, that some instructions are *sensitive*, meaning that the result might depend on the processor mode in which they are executed. These instructions do not return general protection faults when executed in non-privileged mode, but rather a different response than expected. A guest OS issuing sensitive instructions will expect the instruction to be executed in

[15]Typically, one wishes to executa as many instructions as possible on the CPU for performance reasons, while still keeping the system secure.

privileged mode, while the guest OS is in fact running in non-privileged mode and the reply will thus be that of a non-privileged call. As an example of these challenges, Intel's IA-32 architecture as implemented on their Pentium processor family contains at least seventeen instructions that would cause problems in a virtualization context [15].

To mitigate the issue of privileged and sensitive instructions, CPU virtualization on x86 architecture can be done using one of the following three techniques:

- Full virtualization using binary translation.
- OS-assisted virtualization - paravirtualization.
- Hardware-assisted virtualization.

Full virtualization using binary translation

This technique, also known as native virtualization, is used about a setup where a hypervisor, type 1 or type 2, is doing the task of mediating between an unmodified guest OS and the underlying hardware. This mediation is done by the VMM component of the hypervisor. Full virtualization offers the guest OSs access to the underlying hardware by simulating one instance of the hardware resources to each of the guest OSs. Each instance of the installed guest OSs will then see a full "normal" set of hardware, which it believes it owns wholly. This means that any OS can be installed without having any special virtualization support. A constraint is that the guest OS must support the underlying hardware architecture (for example x86) so that no instruction set emulation is necessary.

Full virtualization implementations must solve the problem with privileged and non-privileged instructions. [13] describes a solution for this using *binary translation* of instructions and address space. This way the guest OS perceives that its operations are performed directly on the processor in privileged mode, while it in reality is being controlled by the hypervisor. VMware's hypervisor performs binary translation for privileged code and executes unprivileged instructions directly on the processor [16]. The binary translation typically manipulates the address space instructions are allowed to access or replace the system call with one or more other system calls, limiting one guest OSs abilities to write to memory space owned by other virtual machines or the host OS.

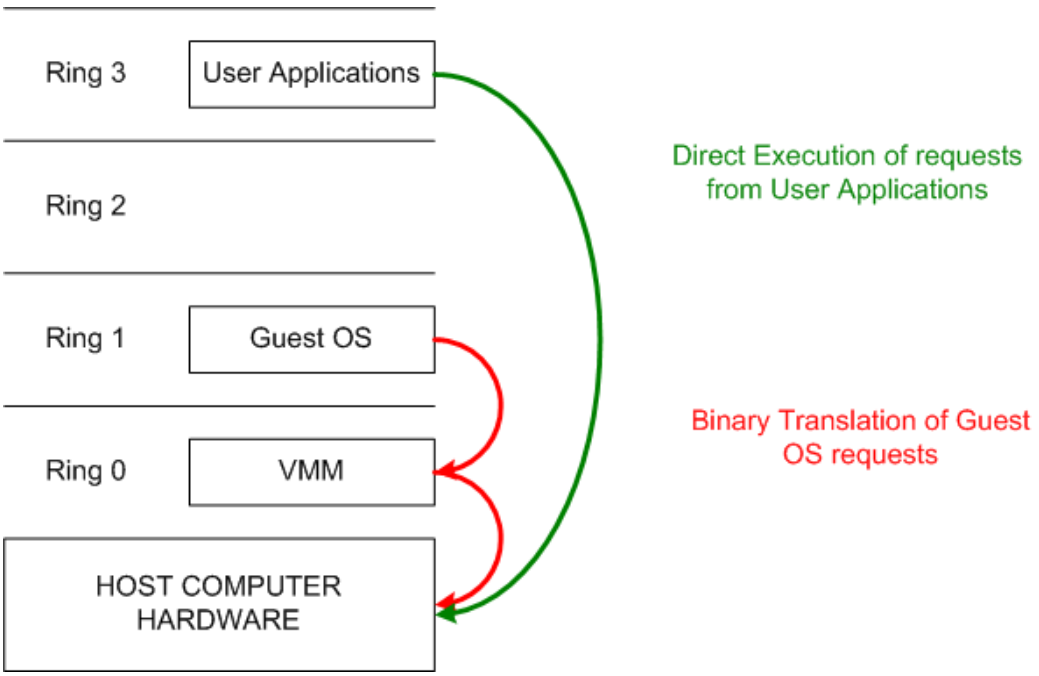


Figure 2.4: Full virtualization using binary translation, from [13].

This principle is illustrated in figure 2.4.

In the test setup during the implementation in this thesis, Ubuntu Server Edition was used as a host OS and VMware Server (former VMware GSX Server) as a full virtualization environment.

OS-assisted virtualization - Paravirtualization

Paravirtualization is similar to full virtualization, but differs in one major area. While the guest OS on a full virtualization system can be any operating system made for the appropriate processor instruction set, the guest OS on a paravirtualization systems is modified to work with the hypervisor. The gain is that the performance approaches that of non-virtualized systems. The disadvantage is that paravirtualization systems does not natively support unmodified OSs. Instead of doing the relatively resource consuming tasks of trap-and-emulate or binary translation, the OS is rewritten to make calls to a special API in the hypervisor. This API implements the privileged and sensitive instructions in a more efficient way than is the case with

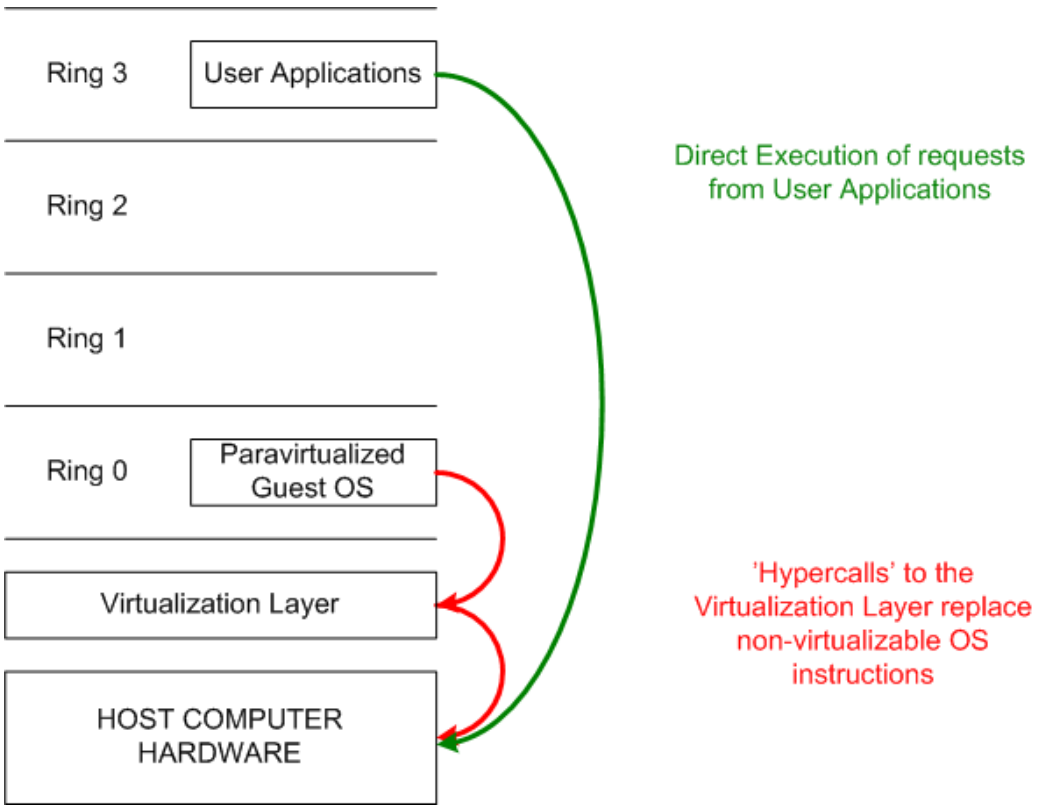


Figure 2.5: Operating system assisted virtualization, from [13].

full virtualization. This principle is illustrated in figure 2.5. VMware uses elements of paravirtualization techniques in some of their products, like for instance VMware Tools and the "Vmxnet" I/O device driver. Note, that this is not CPU virtualization, although it uses the same technique of making calls from the guest OS to the hypervisor.

Hardware-supported virtualization

To better be able to handle virtualization, both Intel and AMD offer processors with hardware support for virtualization techniques, called Intel VT and AMD-V, respectively. The hardware support includes some new instructions that help a VMM and a guest OS execute privileged instructions in the appropriate level [12], and enable full virtualization without using binary translation. Intel VT offers support for 64-bit guest OS, and allows a VMM and a guest OS to share access to the hardware using a new privi-

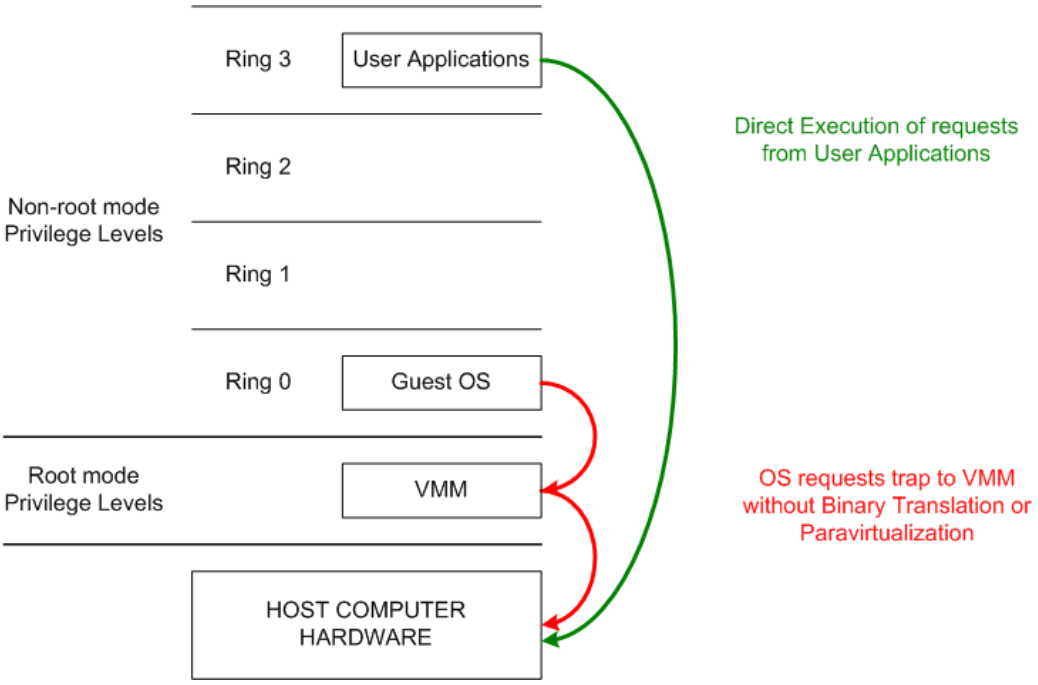


Figure 2.6: Hardware assisted virtualization, from [13].

lege level/ring [15]. AMD introduces two new modes, called Host Mode and Guest Mode, and a new instruction called *VMRUN* as an extension to their AMD64 architecture [14]. The *VMRUN* command switches the processor into Host Mode, which is used by the hypervisor to manage its guest OSs. This principle is illustrated in figure 2.6.

2.2.2 Memory virtualization

Beyond the CPU, the memory is another important resource that needs to be virtualized in virtualization systems. The available physical memory of the host must be made available to the virtual machines when needed. The virtualization technique used to share memory between virtual machines is similar to the functionality that present day operating systems use to allocate memory between its applications [13]. This virtual memory scheme allows an application to see a contiguous address space, which does not necessarily reflect the true physical address allocation. To make memory management work in a virtualized environment, it is necessary to perform management at two levels. The guest OS must perform memory management for the

applications running within it in the regular manner of OSs, while the hypervisor/VMM needs to manage memory mapping for the various virtual machines. The latter level of virtualization is achieved by virtualizing the Memory Management Unit (MMU) found on all modern x86 architecture CPUs [13]. VMware uses shadow page tables to provide a mapping between the virtual memory pages of the guest OSs and the actual pages in the physical memory. Using such shadow page files allows the linear address space of the virtual machine to be mapped directly to the real machine addresses, hence increasing the speed of memory operations. Another technique, used for instance by the Xen hypervisor[16], is to use a paravirtualized approach to allow the guest OS partial direct access to the physical memory page tables [17].

2.2.3 I/O and device virtualization

Device virtualization is about presenting each virtual machine with a set of generic virtual devices, and translating the requests going to and from the real hardware. When performing device virtualization, there are two main decisions that have to be made; where to place the drivers for the physical hardware and which virtual hardware should be presented to the virtual machines. Regarding the first issue, VMware places the device drivers in the hypervisor while for instance Xen uses an indirect architecture where the drivers are placed within a privileged virtual machine[17] [17]. The second will typically be a configuration issue.

[16]Produced by XenSource Inc., now a part of Citrix Systems.

[17]Actually, Xen uses a back-end device driver in the privileged virtual machine and a front-end device driver in the guest OS.

2.3 Hiding Virtualization from Attackers and Malware

Virtual machine environments are quite common tools for malware analysis and classification [18]. However, it is an increasing trend that malware authors include code to detect the presence of such environments [19]. The consequence is that the malware might choose to hide its true behavior, or even remove itself completely when detecting analysis environments. It is likely, however, that as virtualization becomes more common in production systems, malware will want to run in virtual environments as well.

[18] shows some possible techniques used by malware to detect whether it is executed within a virtualized environment, emphasizing the VMware product line. Two concrete techniques used to detect the presence of VMware, and ways to mitigate these weaknesses, are described. The first detection technique is to look for the presence of the VMware internal communication channel, which is used to communicate between the host and the guest OSs. This channel is used to move data between host and guest when using for example the clipboard application or simple drag-and-drop functionality. The test is simply to look for a known value in specific registers, the presence of which indicate that the channel is being used. If this test is issued on a non-VMware system, the instruction will trigger an exception handling routine. [18] states that inserting execution of the malicious code into the exception handling routines could be used to make sure that the malware is only executed in a non-virtualized environment.

[18] also describes another detection technique using the location of tables such as the *Interrupt Descriptor Table* (IDT), *Global Descriptor Table* (GDT) and *Local Descriptor Table* (LDT). These tables are three of the four memory management tables used for segmented memory management in the x86 architecture. The GDT and LDT are tables that contain segment information giving the base address, access rights, type, length and usage information about memory segments. The LDT was used on early x86 architecture processors that did not have any paging feature, and described privately held memory per user process. The GDT data structure is used to hold information about shared global memory segments, while the IDT holds information about interrupts and exceptions. These tables are located at predictable addresses in the virtual machine that differs from the addresses found in the host OS, due to the fact that the x86 architecture processors only has one register to store the address of each data structure table. This causes prob-

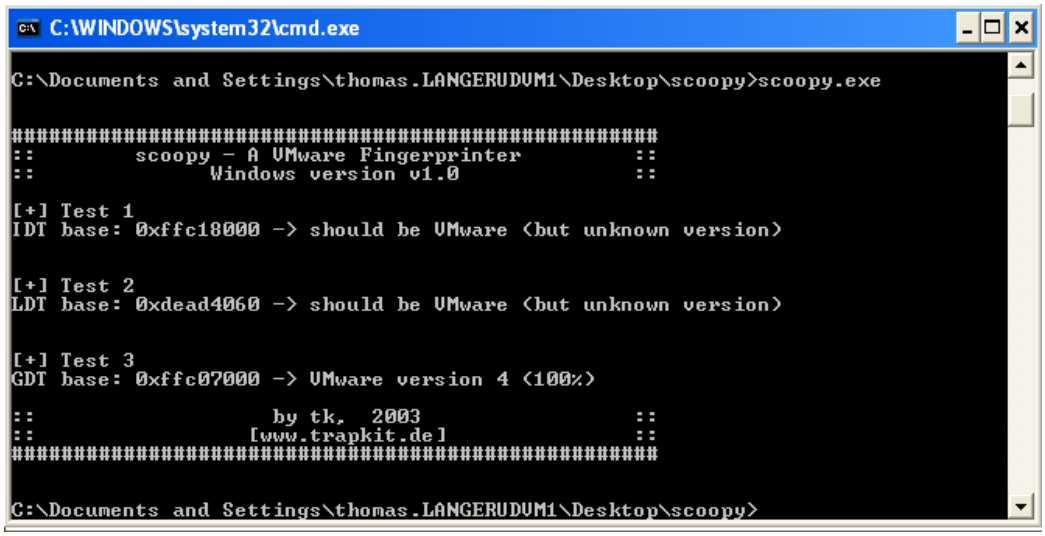


Figure 2.7: "Scoopy" application run inside a virtual machine.

lems when more than one OS tries to use the same registers. An OS running in a virtual machine (thus in Ring 3) trying to write to any of these registers will cause a general protection fault, which the VMM can handle. Reading from the registers is however not restricted to Ring 0, and an OS in a VM would get the information belonging to the host OS. To avoid this from happening, the VMM must provide each virtual machine with its own separate virtual IDT, LDT and GDT register. For more information on register issues with virtualization on Intel processors, see [20].

Examples of implementations that use the IDT detection technique are *Red Pill*[18] and *Scoopy*[19]. Usage of Scoopy in native Windows XP and in Windows XP on VMware Server is shown in figures 2.7 and 2.8 respectively.

The author of the *Scoopy* program mention other characteristics that could be used to detect the presence of VMware, including:

- Copyright notes/vendor strings in various files and registry keys.
- VMware specific hardware drivers, such as "VMware Virtual IDE Hard Drive" or "VMware Virtual IDE CDROM Drive".
- VMware specific BIOS.

[18]Invisiblethings.org - Red Pill - `http://www.invisiblethings.org/papers/redpill.html`.

[19]trapkit.de . Scoopy Doo - `http://www.trapkit.de/research/vmm/scoopydoo/`

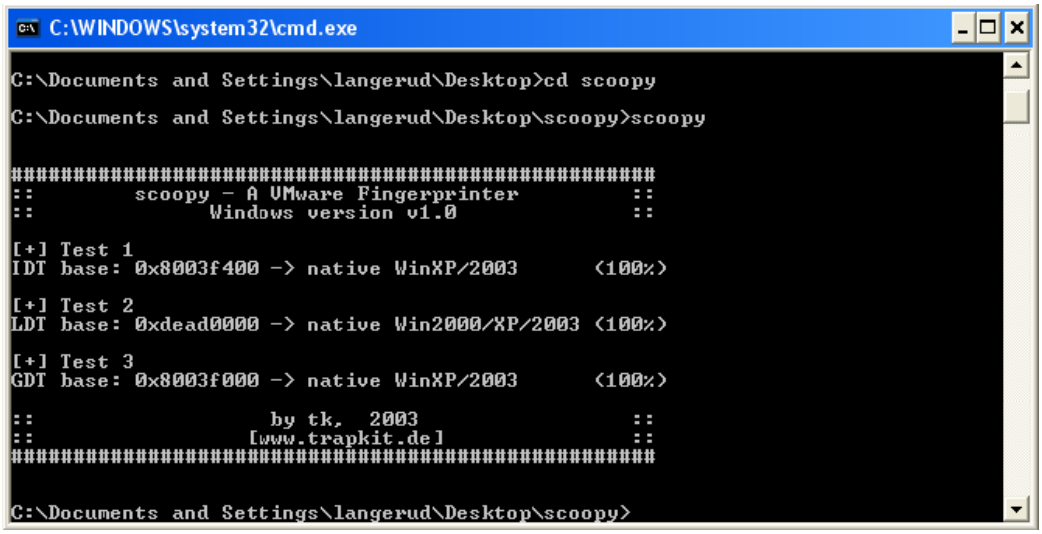


Figure 2.8: "Scoopy" application run outside a virtual machine.

- VMware specific MAC addresses, in the 00:0C:29 or 00:50:56 address range.
- Installed VMware tools.
- Sensitive system calls that need to be trapped and emulated, will take longer time when handled by the hypervisor than they would when executed directly on the CPU. This requires some kind of baselining to compare execution time. One such technique is described in [19].

For mitigation, [18] presents a list of VMware configuration options to alter the memory relocation functionality and the binary translation functionality. Some of these options are not documented by VMware, but the effect is nevertheless that VMware is no longer detected by Red Pill or Scoopy. One consequence of this is that the drag-and-drop and copy/paste functionality is disabled. Another solution to the detection issue is to alter or remove the *magic value* associated with the internal communication channel, but this requires patching of the VMware binary and the VMware disk image and is quite complex.

2.4 Malware obfuscation

Malware authors mainly seek to obfuscate their code for two reasons, both related to obscuring the malware's true purpose and nature. The first is related to hamper static analysis of the malware by malware experts such as researchers and anti-virus vendors. The other is to avoid detection or recognition by automated tools, typically anti-virus engines. In practice, many of the same techniques are applied for both purposes, but only the ones relevant for avoiding automated detection will be discussed here, as manual malware analysis is outside the scope of this thesis. As malware authors generally can be seen as a creative and heterogeneous group, the listing given here is mainly an effort to give an introduction to the main principles and techniques and is not necessarily complete.

This section is closely related to the anti-virus techniques discussed in section 2.5; the techniques utilized for obfuscating malware are adapted to the techniques for detecting malware and vice versa in a never-ending race.

The reason that these techniques are presented here is to show that a regular signature based surface malware scan may often be insufficient even to identify relatively simple malware families; even well known malware may pass surface scanning if it is obfuscated using one of the techniques listed below.

The techniques presented here are the ones given in [1], which are written with viruses in mind, but the same techniques could very well be applied for other kinds of malware (especially for other kinds of automatically reproducing code, but also for creating new species of existing malware).

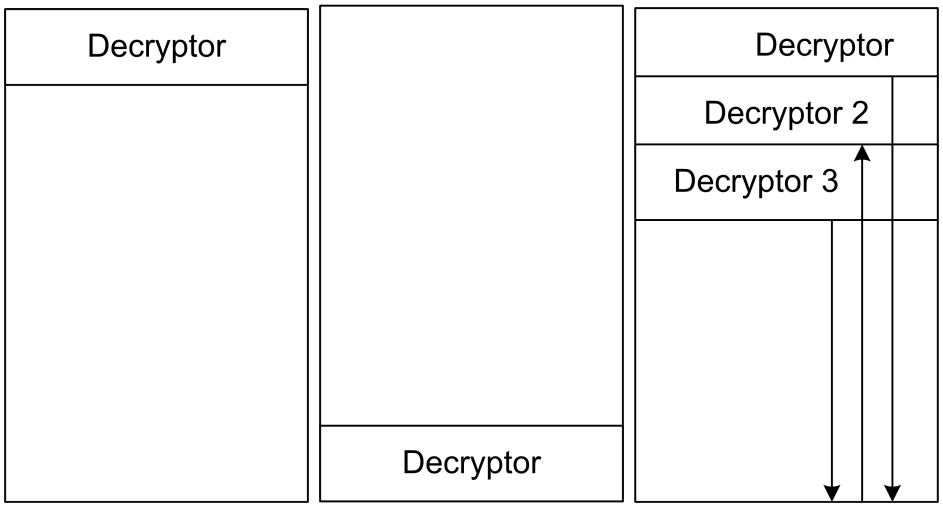
The reason for executing the malware samples with real-time anti-virus scanning software running in the background, is that known malware[20] often is distributed with a new packer, using for example polymorphic techniques[21]. This means that samples that on the surface bears little or no resemblance may in fact be the exact same malware. The reason for this is that malware creators may use several packers (typically using encryption) on the same sample, making old signatures for the malware family useless. Practical experience in this field shows that it may take a significant period of time before the anti-virus vendors update their signatures to capture a new version of an already known malware.

[20] Known malware is in this case malware that has already been analyzed and created a signature for by anti-virus vendors.

[21] A introduction to polymorphic techniques will be given later.

2.4.1 Encrypted malware

Encryption is the simplest technique for malware obfuscation presented here. The point of malware encryption is, obviously, to encrypt the malware body so that it is not recognized as what it actually is. Additionally, the encryption key may be changed between every infection (for self-copying malware), so that creating a catch-all signature of the malware body may be non-trivial.



(a) Encrypted malware with decryption header.

(b) Encrypted malware with decryptor at the end.

(c) Malware encrypted in layers.

Figure 2.9: Encrypted Malware with decryption header, adapted from [1].

Common to all encrypted malware is that it must, obviously, have some means to decrypt itself prior to performing reproduction, execution, infection and similar operations. This is done using a decryption header which is attached to the decrypted body in some fashion (usually before as shown in figure 2.9(a), but it might be placed other places to confuse analysts).

Several techniques can be utilized to make analysis and detection of encrypted malware harder (see for example the layered encryption shown in figure 2.9(c)), but in the end detection of encrypted malware is relatively simple; instead of creating a signature for the actual malware body (which would be very difficult, as the body varies depending on the key), signatures are created for the decryptors themselves, which at some point must be a static recognizable header. The weakness with this technique is that classification of the malware may be very inaccurate, as several viruses may

use the same encryption/decryption code. Indeed, some malware packers are distributed to the entire community, providing possibility to encrypt existing unencrypted malware.

Another possibility is to include decryption routines for the most common encryptors in the anti-virus engine, providing the engine with the possibility to perform on-the-fly decryption of the most commonly seen encryption techniques. Obviously, this can be problematic if many malware creators create their own encryption/decryption routines or employ advanced key hiding schemes, but the general trend seems to be reuse of many of the same techniques and implementations.

2.4.2 Oligomorphic code

Oligomorphic code is the next step in code protection and addresses the detection technique for encrypted malware described in the previous section. Oligomorphic malware has the possibility of replacing or altering its decryptor in new generations. The malware typically carries a number of decryptors it can use, and each time it reproduces it uses a different header. Due to the limited number of decryptors available, signature based detection of the header is still possible, although it can be somewhat risky as some of the decryptors may be used more seldom and thus be missed during analysis.

As with encrypted malware, it is possible to use decryption routines in the scan engine, especially if the behavior of the different decryptors are identical[22].

2.4.3 Polymorphic code

Polymorphic code further reduces the effectiveness of signature based scanners by introducing mutating decryptors. Polymorphic malware automatically generates new decryptors for new generations. The algorithm remains the same, but different techniques are applied to give the code a new appearance. One typical technique for doing this is insertion of some kind of junk operation, such as for example the assembler *NOP* instruction or manipulating registers or variables that are never used for anything useful. The actual

[22]It is, naturally, possible to create decryptors with identical factual behavior and different signatures.

decryptor code may only be single instructions padded with any number of junk code on every side, making signature based detection very difficult, not to say impossible. For an example of this, see code listing 2.1, which shows how the garbage operations can be mixed with the real code. Notice that there are hardly any occurrences of two consecutive non-junk instructions.

```
; Group 1 Ű Prolog Instructions
inc     si          ; optional, variable junk
mov     ax,0E9B     ; set key 1
clc                 ; optional, variable junk
mov     di,012A     ; offset of Start
nop                 ; optional, variable junk
mov     cx,0571     ; this many bytes - key 2

; Group 2 Ű Decryption Instructions
Decrypt:
xor     [di],cx     ; decrypt first word with key 2
sub     bx,dx       ; optional, variable junk
xor     bx,cx       ; optional, variable junk
sub     bx,ax       ; optional, variable junk
sub     bx,cx       ; optional, variable junk
nop                 ; non-optional junk
xor     dx,cx       ; optional, variable junk
xor     [di],ax     ; decrypt first word with key 1

; Group 3 Ű Decryption Instructions
inc     di          ; next byte
nop                 ; non-optional junk
clc                 ; optional, variable junk
```

Listing 2.1: Illustration of a mutated simple XOR decryption routine of the 1260 virus [1].

There are several possible ways to try to fight polymorphic code. A conceptually relatively simple way is to run the code through optimizers, which should ideally remove all non-relevant instructions, such as *NOP*. However, this is in practice not as simple as it sounds, as garbage instructions may include pointless registry modifications and similar seemingly meaningful tasks, so separating useful and garbage instructions may be a complex task.

A more reliable method is running the code in simple emulators[23], and thus letting the malware do the dirty work of decrypting the malware body itself.

[23]Which in a sense can be seen as minimized version of what is attempted done when executing the malware on a virtual machine running real-time anti-virus software in the implementation described later in this thesis.

2.4.4 Metamorphic code

Metamorphic malware takes the process of modifying code all the way. While polymorphic malware is able to obfuscate both its header and body, it will always have the weakness of the decrypted malware body which sooner or later will have to be written to disk or memory prior to execution. Through either emulation or real-time monitoring, it will always be possible for anti-virus software to detect this body. Metamorphic malware modifies its own code more or less randomly in every generation, while still keeping the same functionality. This modification can be done in several ways. The example of inserting garbage code as described for polymorphic code is one way, another is separating the code into blocks[24] and recombining them using jump instructions, as was done with the badboy virus shown in figure 2.10. Metamorphic malware may also carry its original source code and randomly modify it by generating junk code or performing other modification operations prior to recompiling with any appropriate compilator installed on the infected system[25].

When fighting metamorphic malware, traditional detection techniques are rarely sufficient: the malicious code may look completely different from generation to generation, with the actual malicious operations being intermixed between myriads of innocent (and useless) operations. Typically, detection of metamorphic malware must be done by looking for the "typical malicious behavior" such code will perform, such as infecting other files, performing operations to hide itself and so on.

Note that it naturally is possible to combine all of the techniques above. Such that polymorphic malware may also metamorphically drop a different payload in every generation, in order to further hamper detection.

2.4.5 Behavior modification

The techniques above can also be combined with behavior modification to further enhance the obfuscation. This will typically be related to detecting virtualized and emulated environments and showing benign (or no) behavior when such environments are detected. In the case of the techniques described

[24]These blocks should be too small or generic to be made signatures for.

[25]This is especially relevant to system which often come with pre-installed compilators, such as *nix systems.

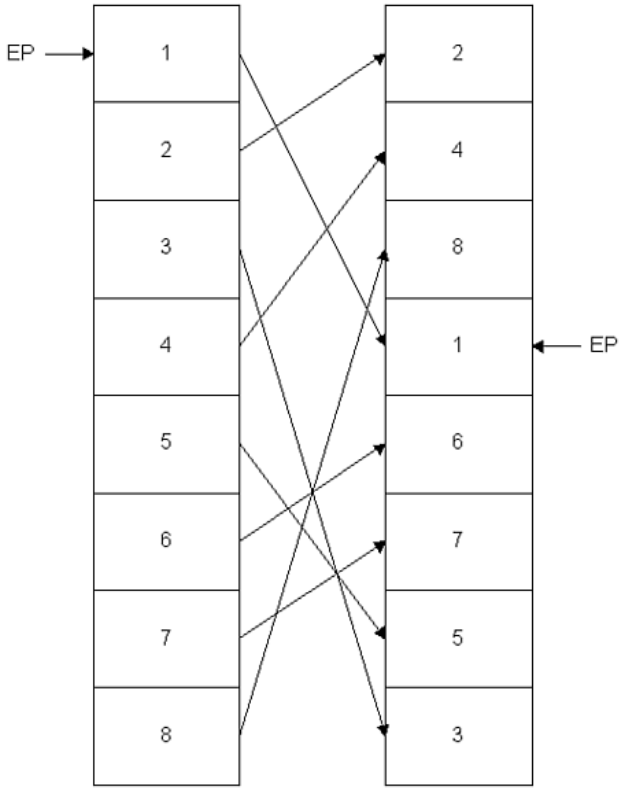


Figure 2.10: Example of reordering of modules in the metamorphic *Badboy* virus, from [1].

above, malware may choose not to decrypt itself if a virtualized environment is detected. Both techniques for detecting virtualization and countermeasures for detection are discussed in section 2.3. Luckily for the implementation done here, VMware is being used more and more for hosting servers, and can as such be expected to be a more and more likely target of attack itself. This means that VMware detection in the long run is unlikely to be a high priority for malware creators, as systems running on VMware may be seen as desirable targets in themselves.

2.5 Anti-virus technology

This section gives an overview over the technologies and techniques used in anti-virus solutions. Although combating malware consists of a myriad of different instruments, including code quality assurance techniques, firewalls, intrusion prevention systems, organizational policies and more, the focus will be on detection techniques used in what is traditionally considered anti-virus software. The reason that these techniques are described here is that anti-virus engines are an essential part of the functionality of the PowerScan framework.

Initially, note that there are relatively few good sources on just how commercial anti-virus solutions work; the industry is highly competitive and any new technology that can gain one vendor a few percents higher detection rate than its competitors is unlikely to be published and shared. In addition to the competition faced by other vendors, anti-virus software developers are in a never-ending race with the malware creators, who are constantly trying to come up with ever new ways to defeat the detection and prevention techniques. These people will often use any knowledge about anti-virus software to defeat it. Therefore, most of the information presented in this section is based on [1], which appears to be one of the very few thorough presentations on how anti-virus engines work.

Anti-virus detection techniques involve several different techniques with different complexity and different detection capacities, starting from simple signature-based malware detection to detection of unknown malware and malware that uses obfuscation techniques such as encryption, metamorphism and polymorphism.

When talking about anti-virus scanning techniques, there are basically two different kind of scanners that are employed; on-demand scanners and real-time scanners. The on-demand scanners are executed by the user or some kind of scheduling mechanism, and can usually be configured to scan different parts of the file system or memory. The other type of scanner is the real-time kind. This scanner is usually loaded into memory at system boot time, and aims to scan every piece of code being read, written and executed. This could be achieved for instance by hooking into system calls related to opening, reading, writing or closing files, or memory read and write. Real-time scanners can also hook directly into the file system. The scanner can be implemented as an application or register itself as a device driver.

2.5.1 Signature scanning - First generation scanners

Searching files or memory locations for byte sequences extracted from known malware using simple string matching is one of the simplest malware detection mechanisms. Signatures are made from sequences or patterns of instructions that identify a malicious block. One important challenge with using signature scanning is that the signature patterns must be long enough to detect a malicious sequence uniquely, avoiding producing false positives. A signature for one particular malware variant might detect other variants or even new but closely related malware, given that the same signature is present in both. However, there is also the possibility that other and completely different malware are incorrectly classified as a known malware even if the overall behavior is different. Incorrect classification may again lead to disinfection attempts resulting in corrupted files. If the code being disinfected is part of the boot sector or some way critical to the OS, the system may crash or be rendered useless. To help improve accuracy of detection, signature scanners may support wildcards in the signature pattern or allow for several mismatches within the string. To speed up the string matching process, scanners may employ optimizing techniques such as hashing of code, scanning only the beginning and end of files or using information in executable file headers to start scanning at the entry point of the code.

2.5.2 Smart scanning - Second generation scanners

Smart scanning manages to mitigate obfuscating techniques such as inserting *NOP* or other bogus instructions in the assembly code. The smart scanning technique ignores such instructions when applying the malware signature. The same technique can be used with malicious code in textual format, such as scripts, that is obfuscated by white spaces or newlines. Another smart scanning-like technique, called "Skeleton detection," works by removing all non-essential instructions or code lines such that only important instructions make up a signature. These bogus code removing operations can generally be performed using assembly level optimizers on the code prior to performing scan.

Some second-generation scanners use a technique known as "nearly exact identification", which makes use of more than one search string. The classification is then based on the number of matches found in a given file. Determining a correct match is essential if attempting disinfection. In the worst

case, erroneous identification may an incorrect disinfection routine being initiated, can cause corruption or deletion of benign files or system services. Another method of doing "near exact identification" is relying on checksums over parts of the malware code. Using hashes improves the scanning rate, and is done by most scanners. "Exact identification" takes this technique one step further, by taking checksums over all segments of a malware found to be constant. This means that all parts of a malware sample containing variable data must be identified and excluded. This is not a trivial task, as code and data might be tightly intermixed.

2.5.3 Algorithmic scanning

If the standard search algorithm of a scanner is not able to detect a given malware sample, there is a need for a more specialized detection algorithm. This is often referred to as "algorithmic scanning" or "virus-specific detection". This technique utilizes a scan definition language to define detection routines for individual malware, which are executed in a predefined order on the objects opened by the scanner. The detection routine might look for numerous strings in one search, or perform address translation between virtual and physical addresses. Algorithmic scanning is found in for instance the Kaspersky and Norton anti-virus engines [1]. To speed up the scan operation, filtering techniques might be used in combination with algorithmic scanning. Filtering is justified in the fact that one type of malware will usually only infect one or a few objects, such as for example only given files or the boot sector. This means that the algorithm for detecting one type of malware is only done on objects that it can infect.

Algorithmic scanning can also be used to attack malware obfuscated with techniques such as encryption. An algorithm might detect known packers (by using signatures for the decryptors) and perform decryption of the body before applying normal signature scan. Attacking the encryption of code is called "X-RAY scanning".

2.5.4 Code emulation

It is also possible for scanners to perform code emulation in order to combat malware obfuscation techniques. Code emulation in anti-virus engine is a similar technique to the one implemented in this thesis. By using a minimal

sandbox environment, the code can be allowed to execute for a short period of time, so that any encrypted code will be decrpyted and written to memory or disk. When code is written, it can be scanned using an ordinary signature scanner.

The greatest limitation of code emulation solution is performance. For scanning a single file running it in an emulator is no problem, but when performing for example a complete system scan, doing a full emulated execution of every single (executable) file is not a realistic option. One possible remedy for this problem is to assume that the decryptor will be the first thing to execute in the malware, and only execute the files for a very limited period of time. This can be attacked by malware that executes some relatively significant number of (processing expensive) garbage instructions prior to decrypting itself or by using techniques such as staged/layered decryption. While an extra loading time in the order of 1-3 seconds is insignificant for a single malware while unpacking, it is intolerable for a scanner to allow each sample more than one second when performing scan of a significant number of files (several tens of thousands is definitely not unusual here). Additionally, ordinary sandbox/emulation discovery techniques can be used to detect the sandbox and hide the malicious behavior if an analysis environment is detected.

As mentioned earlier, there is released very little information about how individual anti-virus engines works, so there is no way to accuratly assess how widespread this technique is. However, considering the significant processing required to set up a sandbox for every single executable file, it is unlikely that all scanners use it for every file. As malware packers are very popular[26], it seems that they are still an efficient tool for trying to avoid detection by anti-virus engines. This indicates that code emulation implementations either are not being used to a large degree or are not efficient enough to detect most packed malware. It also indicate, however, that the malware packers themselves are being detected by the anti-virus products. This may possibly be mitigated by creating custom packers.

2.5.5 Metamorphic malware detection

Providing reliable detection of metamorphic malware using the techniques listed above is hard, so other techniques are required. Typically, it is nec-

[26]Reuters: BitDefender Lab's top 10 malware list for April (2008) dominated by malware packers - `http://www.reuters.com/article/pressRelease/idUS166334+09-May-2008+MW20080509`.

essary to analyze the behavior of the malware instead of relying on given strings existing in its code. This can for example be done by disassembling the malware instructions and look for vital instructions, possibly combined with a state machine in order to track the order of instructions. This way, malware signatures can be built based on the central operations it performs on the infected system, rather than on the actual strings it contains. Heuristic analysis, presented in the next section, is a good example of a technique that can be used against metamorhpic malware.

Once again, recall that knowledge about the inner workings of individual anti-virus engines is hard to obtain, so different engines might have good solutions for detecting metamorphic code.

2.5.6 Heuristic analysis

The principle of heuristic analysis is to look for malicious behavior, rather than looking for known patterns of a given malware family based on previous analysis. Instead of searching for string signatures, a heuristics scanner looks for combinations of instructions, events and other indicators that a program is not legitimate. The main reason for using this technique is to detect previously unknown malicious code for which there exists no specific signature. In order to analyze behavior, heuristic scanners need to have access to the unencrypted code of the malware. For this reason, heuristic engines need to be combined with other techniques that can unpack (and, if needed, decrypt) packed code. Another possibility is to implement the heuristic engine as a real-time service, which monitors executing threads and tracks their operations on the system.

A heuristics engine usually combines parsers, flow analyzers and disassemblers in addition to a weighting or rule-based decision engine to evaluate how likely a given file is to be malicious [21]. The first set of tools is used to scan the code and correlate events and calls performed to determine how likely the actual code is to be malicious, for example by looking for virus or worm reproduction mechanisms. This can then be combined with other factors which *might* imply that the program is malicious, such as presence of packers, anti-debugging techniques, memory-resident code and similar things [1]. The results from all of these operations are then fed to the weighting or rule-based decision engine, which then adds up the “danger score” for the entire situation and decides if action should be taken.

It is worth noting that heuristic scan methods potentially may produce false positives. This is justified by the fact that heuristics scanners have the ability to detect previously unknown malware, which represents a significant advantage compared to other techniques. The number of false positives can also be adjusted by modifying the threshold of the decision engine.

Another problem with heuristic scanner is that they are (obviously) unable to provide detailed classification of malware, so they will typically not be able to provide disinfection strategies. For this reason, heuristic scanners will usually be most interesting to use in a real-time scanning context, as suspected malware then may be blocked, preventing the infection from happening in the first place.

2.5.7 Memory scanners

Some malware might avoid initial detection and get loaded into memory and executed. Once it is there, it need not necessarily be executed as a user-space application, but possibly also as kernel modules or device drivers. Once loaded, malware can be memory resident, meaning that the code remains active even after the application has seemingly been terminated. This kind of malware is called Terminate and Stay Resident (TSR). Once malware is active in memory, the corresponding executable can not be deleted from the file system, making disinfection using regular techniques impossible. Moreover, a memory resident malware may hide itself from scanners using stealth techniques, and also monitor its own system objects such as registry keys, protecting them from manipulation or removal.

In order to detect and thus have a chance to remove such malware, memory scanners are required. As TSR malware can run as kernel modules or device drivers, a memory scanner should run in kernel mode in order to be able to access all relevant memory pages. Once a memory scanner detects malicious code executing, it should start a disinfection routine which will terminate the thread or process and perform the required disinfection operations.

2.6 Malware naming and classification

The lack of standardization of naming conventions is, especially for malware analysts, a significant motivation for running a sample through several anti-virus engines in order to accurately classify a malware sample. This section gives an introduction to issues regarding naming conventions in the anti-virus industry. First, some issues regarding malware classification are presented. Then, a widely used [1] naming convention from the Computer Antivirus Researcher Organization (CARO) is introduced before some malware classification coordination efforts are presented. Lastly, some automated classification schemes are presented.

Ideally, there should exist some unambiguous and preferably automated process for classifying malware in such a manner that once a new sample is found, its full and unique name is implicitly given. Unfortunately, creating such a tool is practically impossible due to the immense diversity and volume of released malware. Some efforts have been made in this field, a few of which will be presented later in this section. To date, however, all the available tools solve only some subset of the problem[27], lacks support for intuitive human-readable format or has some other issue which discourages its usage [22].

The will to attempt to coordinate classification efforts seems to be present among most anti-virus vendors, but the tools to do so efficiently still seems to be missing. Although manual naming conventions such as the CARO Virus Naming Convention exists (and indeed are attempted used), anti-virus researchers are still unable to coordinate their names fully. With tens of thousands of new malware variants released monthly (or about 2500 daily [23]), researchers have their hands full coping with designing detection for new samples and have little or no time for coordinating names between vendors. Obviously, with the amount of released malware, no central authority can decide and assign names for all detected malware. As the vendors use their names in several different places such as in the anti-virus engines, in security bulletins and many other places, adapting to other vendors' naming after a name has been chosen is usually not desirable either.

Still, an established naming convention will provide a common standard and understanding for the names of malware, and will contribute to reduce the

[27]For example, there are tools that can classify a malware sample in relation to others, that can classify macro viruses and so on, but none that can create a unique, meaningful name for any malware type.

confusion and problems resulting from inconsistent naming.

2.6.1 CARO Virus Naming Convention

The CARO Virus Naming Convention was first introduced by CARO in 1991 [24] in order to try to coordinate the virus naming schemes of different anti-virus vendors. As CARO is not a standardization body, the scheme is not forced on the anti-virus vendors, but rather strongly recommended. The original scheme was superseded by a revised version in 2002 [25], which greatly expands the convention and attempts to adapt it to recent times. The revised CARO NC is used to classify several types of malware. Although few anti-virus vendors use the convention directly today, it still remains the only convention that most anti-virus companies ever tried to adopt [1]. Elements of the scheme can be clearly identified in almost every vendors' scheme today. The presentation given here is only a very simple overview of the relatively complex conventions, for a full description refer to the original papers [24,25]. Note that CARO does not provide any central naming coordination authority, but rather offers a common way of constructing names.

Unlike some other naming conventions, especially automated, the CARO scheme relies on textual readable descriptions. The full name is constructed as follows:

```
<malware_type>://<platform>/<family_name>.<group_name>
  .<infective_length>.<variant><devolution><modifiers>
```

Of the fields here, <*family_name*> is the only one strictly required.

The following gives a short description of the fields. In [24] rules and advice for the different fields are given as well, but this will largely not be discussed here.

<**malware_type**> This part of the name indicates which type of malware one is dealing with. This will typically be *virus*, *trojan*, *backdoor* and so on.

<**platform**> This part of the name describes the minimum platform the malware requires to function. If the malware can run on several platforms, the set of platforms is given, such as for example *{Linux,W32}*.

<**family_name**> The family name represents the minimum a virus scanner must report when it detects malware in order to correspond to the CARO naming convention. This is the component that is typically seen as the name of the malware, such as for example *MyDoom* or *BadBoy*.

<**group_name**> The group part of the malware name is used when a large subset of a malware family contains members that are sufficiently similar to each other and sufficiently different from other members of the family. Usage of group name is deprecated in the revised convention.

<**infective_length**> This part of the name indicates the length of the infection of mobile malicious code. It is used to distinguish malware within a family or group based on their typical infective lengths, when appropriate[28].

<**variant**> This field separates minor variants of the same malware family with the same infective length. The value of this field is typically *A*, *B* and so on.

<**devolution**> Devolution indicates the opposite of evolution, and is used to indicate a subset of the original malware which was created due to a bug in the reproduction mechanism. This is valid for malware which reproduces differently based on the environment, where the buggy reproduction is only one of the possible reproduction schemes. The devolution field adds a number behind the variant, typically beginning at *1*.

<**modifiers**> The modifier field is used to convey any particularly important information about the malware and the way it reproduces. This can for example be locale (dependency of a specific language version of the target platform), which medium it uses to spread (such as p2p networks, an exploit, IRC and so on) or other characteristics (such as fast-spreading mass-mailer, slow-spreading mass-mailer, macro and so on).

[28]The infective length indicates the amount of data added to infected files.

2.6.2 Common Malware Enumeration

The Common Malware Enumeration (CME)[29] initiative is a project started in 2005 by the private non-profit MITRE organization and is headed by the US Computer Emergency Readiness Team (US-CERT). CME's editorial board consists of representatives from anti-virus vendors and other relevant organizations. The central point of CME is that it does not aim at solving the problem of malware naming, but rather seeks to provide a central identifier during major malware outbreaks.

Malware indexed by the CME initiative is assigned an identifier on the form `CME-x`, where x is a numeric identifier. By the time of writing, 39 outbreaks have been assigned a CME identifier. As mentioned, the reason that there are so few entries in the CME list is that it focuses on reporting major malware outbreaks.

2.6.3 The WildList Organization International

The WildList Organization International[30] is a non-profit research organization founded in 1996. Its primary purpose is to identify, track, and report on active computer virus threats. The WildList is a cooperative listing of positively identified and verified viruses reported as being in the wild by virus professionals. The list is updated monthly, and consists of in the wild virus outbreaks that are not confined to small regions.

WildList does not attempt to provide a naming scheme for listed viruses, nor does it create names; the aim of WildList is to pick a name among those available and assign it as *the* name for a given virus family or variant. The organization attempts to adopt a CARO-formatted name if it can be verified quickly enough. However, establishing a correct CARO name might take too long time, as there is no central authority for verifying these names. In these cases the most widely adopted name from the industry is used. If none such is established relatively quickly, the name given by the first individual or organization that discovered the virus is used.

There has been suggested that the WildList be expanded to also include other kind of malware[31].

[29] CME website - `http://cme.mitre.org/`.
[30] The WildList Organization International website - `http://www.wildlist.org/`.
[31] Is The Wild List too Tame? - `http://www.appscout.com/2007/02/is_the_wild_`

2.6.4 The VGrep database

The VGrep database[32] is a database generated by scanning samples using different anti-virus scanners from different vendors. The results are extracted and added to a text database. This database can then be used to provide cross reference between malware identification given by some of the more popular anti-virus products. At the time of writing, the list includes 14 different anti-virus products[33].

2.6.5 Automated classification schemes

Automatic classification schemes are, as the name implies, automatic ways to classify a given malware sample. This can be done with two different goals in mind; one is to compare different samples in order to classify them in relation to each other, the other to create a unique ID in order to coordinate efforts. The former can be used to simplify analysis of new samples which are similar to existing ones (as the analyst will know that the new sample is a variant of a given existing one).The latter can be used to create centralized databases which can be used to coordinate manual naming or other coordination efforts among anti-virus researchers and vendors.

In [26], an automated virus classification scheme which compares unknown malware with existing samples and returns a possible match based on behavior is described. This method is unable to classify metamorphic malware, as it relies on the malware code being non-changing. Non-altering code is classified using the notion of *basic blocks*, which is a continuous section of code that does not contain a jump instruction. To extract these, it is necessary to have the code in a state where it is unencrypted, unpacked and not obfuscated. A control flow graph is then constructed, relative addresses normalized and external library code identified. Then, one of three distance algorithms (*edit distance*, *inverted index* and *bloom filters* are the suggested algorithms) is employed to calculate the distance between the content of these basic blocks and the basic blocks of already analyzed malware. This way, the malware sample is compared to the previously existing samples, and is thus classified with respect to, and comparison with, these.

`list_too_tame.php`

[32]The VGrep database home at Virus Bulletin - `http://www.virusbtn.com/resources/vgrep/`.

[33]List of anti-virus products used to produce the VGrep database - `http://www.virusbtn.com/resources/vgrep/which_products/`.

A commercial implementation that seems to utilize many of the principles described in [26] has been created by Sabre Securities. Their *VxClass*[34] software automatically groups malware into families based on behavior. According to their webpage, "*VxClass can structurally compare executables and thus ignore byte-level changes such as instruction reordering or string obfuscation. Small changes in the code or changed compiler settings will not fool VxClass.*" VxClass automatically recognizes and removes most packers, before grouping malware into families based on similarities of functionality. These similarities are calculated by breaking down the malware into directed graphs consisting of code/functionality blocks which are then compared. This way, VxClass can identify whether a malware sample is something brand new or simply a deviation of something existing (in which case existing analysis results to a large degree can be reused). An example output from VxClass can be seen in figure 2.11.

A system for creating a malware ID based on the changes a sample causes on the infected system is suggested in [27]. This approach is different from systems classifying the malware according to system calls issued, as it catches the actual effects the malware has on the target system, rather than simply its behavior. The presented technique is based on executing malware samples in a virtualized environment and tracking alterations and creation of system objects such as spawned processes, registry keys, files and network connections. These system state changes are the basis for fingerprint creation. Similarly to the approaches present above, the authors of [27] proposes that the malware can be grouped based on behavior and distance algorithms for easier overview. Note that the implementation described uses VMware, and as such is vulnerable to attacks on virtualization such as those described in section 2.3 of this thesis.

[34] VxClass - `http://vxclass.com/`.

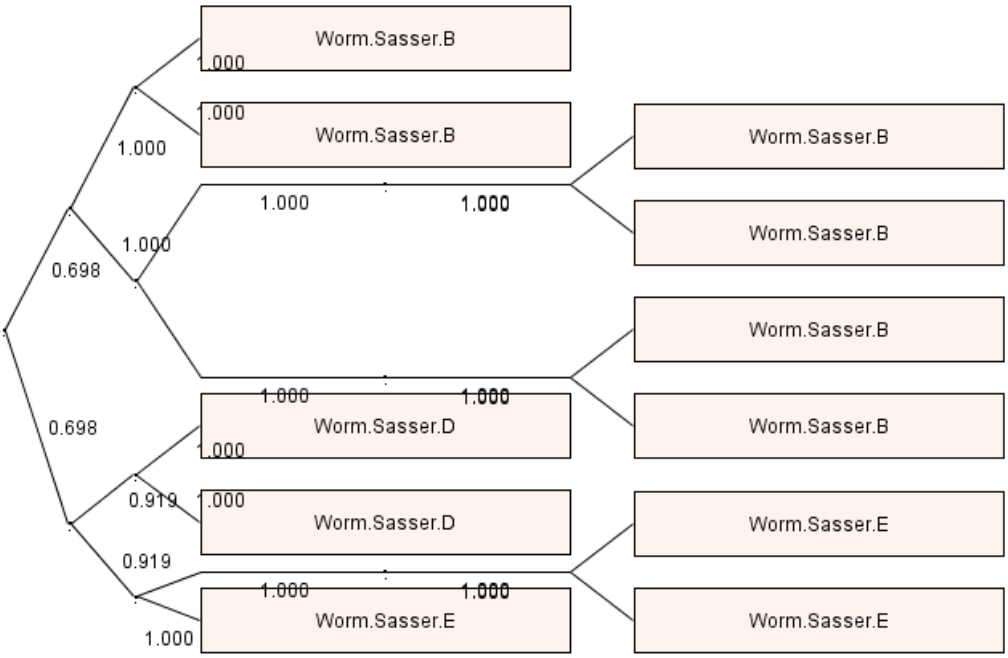


Figure 2.11: Part of a result output from the VxClass application. Illustrates how malware from the same family is correctly grouped together. The full figure, found at `http://www.sabre-security.com/files/rwth_named_colors.png` also shows that other families are grouped together similarily. Family and variant names have been assigned using the ClamWin anti-virus engine.

2.7 Sandboxing and code analysis

Sandboxing is a way of analyzing the behavior of an executable inside an isolated environment. Sandboxing can be done in different granularity, for example by looking at all system calls or just tracking the resulting changes made to the sandbox environment. This section introduces malware analysis in a controlled environment, as this is a technique utilized in this framework. Focus will be on 32bit Windows systems, as this by far is the most widespread platform, and thus the one targeted by most malware. An introduction to the Windows architecture is given in section 2.1.

This section will give an idea about how dynamic analysis can be carried out in a sandbox environment, and how the behavior of malicious code can be studied. There are essentially three ways of mapping changes made to a system by malicious code [28]:

Post-mortem Create an image of the system state before and after the code is allowed to execute, and compare the images.

Static analysis Manually analyze the malicious code to recognize known malware and determine what actions the code is supposed to take once executed.

Dynamic analysis Monitor the actions taken by the code when executing it.

2.7.1 Post-mortem

There exist several tools to take snapshots of file systems and registry for the purpose of comparing a "before" image with an "after" image to detect changes. Doing so gives the analyst information about what changes has been made to the registry and file system by the malware. This post-mortem approach has the disadvantage that temporary changes will not be detected. One example of this is temporary files that are created and removed before the malicious code terminates. One way to mitigate this is to track all changes being done to a file system or registry, which also includes temporary files being created and then deleted. This technique can for instance be implemented using a VMware setup with *undoable* disks[35]. All changes are

[35]Which means that all changes to disks are tracked in log files rather than performed directly on the virtual file system, giving the user the option to either commit or reject

in this case temporarily written to REDO-files, which can be monitored to capture the changes. Another issue is that many changes to the registry and file system are legitimate changes, meaning analysis and filtering is required to pin-point the changes actually made by the malware.

2.7.2 Static analysis

Static analysis is a technique which is about manually analyzing the program code of a malware sample. The challenge is to understand the true nature of a program from looking at the available code. In most cases, the uncompiled source code is not available to the analyst, leaving him or her with the challenge of reconstructing the source code from a compiled binary using disassemblers, decompilers and other tools. As malware authors have a large arsenal of obfuscation techniques at hand, such as anti-debuggers and encryption, reconstructing the source code can be hard. Even if no such techniques have been used, decompilers can not be expected to recreate the source code in a structured and easy-readable form. This will differ according to the level of abstraction of the relevant programming language, but in practice, analysts are usually left analyzing assembly language code. This is especially the case for malware written in low-level languages. Utilizing static analysis enables the analyst to understand much more of the program details than what is gained by simply running the code and observing what happens.

2.7.3 Dynamic analysis

Although static analysis of source code is the best way to ensure completeness, it is often useful to execute the code and analyze its operation in order to get an understanding of its behavior. To be able to do real-time interception of system calls, it is necessary to hook the Windows APIs and redirect the calls to customized code for logging. System call monitoring is about looking at information crossing from a process to some part of the kernel. This includes the name of functions calls, arguments and return values. In Windows, calls both to the Windows Native API (*ntdll.dll*) and the user mode Win32 API dll-files, described in more detail in section 2.1, yields a lot of information about the behavior of the process that is being analyzed, such

the changes after performing some operation.

as which files it creates, which registry entries it creates/manipulates, network activities and so on. It is also possible to perform dynamic analysis at the machine instruction level [28]. This is a very time consuming operation which will not be explained any further in this thesis.

Available tools for dynamic analysis include debuggers, function call tracers, emulators and also possibly network traffic analyzers. One challenge when doing dynamic code analysis is that the code is actually executed and can potentially cause harm to the host system. To be able to safely execute the code and perform dynamic behavior, it is desirable to create a secure environment to contain the malware and its actions - a sandboxed environment. A sandbox can be implemented using different techniques, from using a real system that can be restored to a system copy or similar to using an emulated system where every system call is controlled by the software. A common way of creating a sandbox is by using hardware or software virtualization to split a computer system into several compartments (virtual machines) in which the malware sample may be executed. Virtual machines might, in addition to offering separation and isolation, have a snapshot functionality which allows for easy recovery of a system back to a working state. To serve as a testing environment for malicious code, it is vital that the sandbox offers complete isolation from the host system and other virtual machines running on the same host. It is also possible to employ system call hooking to restrict the actions malicious code is allowed to perform in a host environment. This way it is possible to execute the code and monitor its actions without giving the code any chance to break confinement or do damage to the analysis system.

2.8 Multiple execution paths for dynamic malware analysis

This section introduces an important issue when performing dynamic analysis; the fact that running a program in a given environment often makes the code follow only one of many possible paths of execution.

Program code can branch on a variety of different conditions during execution. This means that just running a suspected malware sample in an analysis environment might not uncover the true nature of the code, and even may end up mislabeling it as innocent. Branching conditions can for example be availability of an Internet connection, existence of specific system objects, files or registry entries, detection of specific AV engines, service packs, virtualization technology and other information [29]. Other conditions may be system time, the contents of a file or registry entry or the result of a URL read operation or download.

[29] describes a test of 308 malware samples from the wild, and concludes that 172 of these branch or make control flow decisions based on the existence of or result from interaction with objects such as system time, files or network access. It also points out that some malware try to detect virtualization or other analysis attempts in order to hide its presence and purpose in such environments. It further describes an implementation of a framework that attempts to run through as may executions paths as possible when performing malware analysis.

The framework in [29] aims to broaden the test coverage when analyzing malware. This is done using solution that tracks certain program input values and identifies points in the program flow where these input values are used as branch conditions. The system is built as an extension to the TTAnalyze application presented in section 1.7.5, and uses TTAnalyze's features for analysis of operating system calls and appurtenant arguments and ability to filter out system calls originating from the code undergoing analysis. Whenever a control flow decision is made, the system takes a snapshot of the content of the process address space and the processor registers. The program is then allowed to continue its original path. When the code reaches a termination point, the system is reverted to the stored snapshot, and the branch condition value inverted before execution starts again.

This approach has some weaknesses. For example, reverting to snapshot may disrupt synchronization with external processes such as network protocols.

The chosen solution to this is to intercept all network system calls and just return a success code. A read operation from a network resource returns a random sting. Obviously, this limits the solution's possibility to determine behavior of malware employing staged downloaders and other malware depending on input from the network.

Exploration of multiple execution paths may also be done manually by instrumenting the malware sample with a debugger and manually setting the values of different variables as the code is executed. Doing this manually can, however, require significant effort as the possible number of execution paths can be significant. Still, it gives the analyst the opportunity to control central critical branch conditions.

Note that this problem is not solved in PowerScan, and that this discussion is only included for reference and information.

2.9 Sandnets - network behavior analysis

Knowing what actions a malware sample is taking locally on the computer on which it is executed is important. However, some malware variants, such as bots, worms, spyware and staged downloaders, may use a network connection as part of their behavior. This can for example be to report back to the malware creator, wait for commands, update itself, download new exploits and so on. Although observing that a malware sample attempts to establish a network connection gives some information about its behavior, discovering how malware interacts with other network services, especially other Internet services, may be essential to discovering its real behavior and intent.

Malware that downloads additional components that wholly or partly depends on an Internet connection for operation is increasing in numbers [30]. Behavioral analysis of such malware samples is challenging, both because of the multiple path issue discussed in section 2.8 and because samples require interaction with resources outside control of the analyst. Ideally, the malware should be allowed to connect to the required external resources and perform its full exchange. However, doing this prior to having an idea of what is going to happen may be somewhat risky from a security point of view[36].

[30] introduces the concept of a *Sandnet*, which is a sandbox that can emulate a full network - or even the entire Internet. The Sandnet concept is realized by running the malware on a system connected to a Sandnet server that is set up to respond to every IP packet and emulate protocols like TCP, UDP, ICMP and DNS. The only configuration needed in the system to be infected is to set the Sandnet server as default route and primary DNS server. Since the malware does not receive any proper response to its packets[37], the entire code will probably not execute. Still, seeing which request the malware initially sends once its TCP, UDP or similar session is establish helps the analyst a long way toward understanding its behavior. If URLs and/or IRC server and channels used by the malware are known, this information may be used to manually investigate what they contain and how they would respond.

As for regular sandboxes, it is possible to perform detection of Sandnets. This could for example be done by resolving a hostname corresponding to a known IP in DNS, test ping RTT or IP TTL using traceroute or other

[36]For example, the malware may try to infect some other host, making it appear as if the attack is originating from the home domain of the analyst!

[37]Obviously, the Sandnet is not able to provide the response the malware expects, it only emulates the basic network protocols.

diagnosis tools. However, this is not very likely to be implemented in real malware, as the effort of writing reliable detection would probably outdo the gain (as a malware analyst may simply just execute the sample, possibly under the control of a debugger, in order to see its true behavior).

An example of an implemented Sandnet tool is the open source tool Truman[38] from SecureWorks.

As for multiple path analysis, the Sandnet solution presented here is not implemented in the PowerScan framework, but merely discussed for information. It would also be possible to set up a Sandnet encompassing a virtual machine running in PowerScan.

[38] `http://www.secureworks.com/research/tools/truman`.

2.10 API hooking

This section gives an introduction to the concept of API hooking, which is a technique that can be used to monitor system calls on the windows platform. This discussion is included as monitoring system calls is highly relevant to performing dynamic analysis, which is one of the things that can be done using the PowerScan framework. Obtaining an understanding of API hooking can help simplify the process of choosing which tools to use for dynamic analysis in PowerScan.

API hooking involves intercepting calls made to a, for the calling application, external library. Once the call has been intercepted, the code execution flow is usually transfered to a section of customized code. This code may for instance perform logging of the API function name and parameters, manipulate input parameters or perform any other optional action. The customized code which is being run is referred as the *hook*. After the customized code has been executed, control is transferred back to the original API function or the calling function. On the Windows platform, hooking is usually done into the Win32 API DLLs such as *kernel32.dll, ntdll.dll, user32.dll and gdi32.dll* described in section 2.1[39]. Rerouting of code execution is supposed to be hard to detect for the malware function. There are, however, ways to detect the hooking[40], which might be employed by malware authors. The rest of this section gives an introduction to ways of redirecting control and injecting code in order to perform hooking.

A prepared library for redirecting system calls in Windows have been made available by Microsoft[41].

The following introduces some techniques for intercepting API calls:

Import table modification is achieved by overwriting the Import Address Table (IAT) found in every Portable Executable (PE) file. The IAT contains the entry-point addresses of all API functions that the executable or library make use of and is used every time an external API is called. By replacing the entry-point address of an API with the entry-point address of the hook, control can be redirected to the replacement

[39]Note that it might be possible for malware to make calls directly to the undocumented *ntdll.dll* API, so it might be desirable to hook this API as well.

[40]For an example implementation of an API hook detector, see `http://www.security.org.sg/code/apihookcheck.html`.

[41]`http://research.microsoft.com/sn/detours/`

code (the hook).[42]

Export table modification is an alternative to IAT modification. The Export Address Table (EAT) is found in every DLL file and gives the entry-point address of every function that is offered by the particular DLL. This table can be modified to transfer control to the hook. Modifying the EAT of a DLL also causes the *GetProcAddress* API call to return the entry-point address pointing to the hook. A process may choose to use the Win 32 API GetProcAddress to get the entry-point address of an API function instead of using the IAT. This way, the EAT technique mitigates a potential weakness with IAT modification, where the hooking can be bypassed using GetProcAddress.[42]

Overwrite beginning of API code in memory with a assembly language *JMP* instruction. This simple technique redirects control to a section of custom code. It does, however, require location of the API code in memory and overwriting non-vital instructions.[42]

Using a proxy DLL with the same name as the real one. This proxy DLL must implement all of the API functions of the original DLL. This technique utilizes *function forwarder* feature. A function forwarder is an entry in the DLLs export section that delegates a function call to another function in another DLL.[43]

Note that the hook code will have to be injected into the context of the application that is being hooked[44] in order to avoid triggering a security exception. This can be done in several ways, but in general, it is about making the application load the hook code into its context. This can for example be done by using the *CreateRemoteThread* API call, which is used to create a thread within the context of another application. This thread can then load the hook code using the *LoadLibrary* API call. Another possible way of doing this is to write the hook DLL into a given registry key (`KEY_LOCAL_MACHINE¬\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_¬DLLs`), which will be loaded by *user32.dll* (*user32.dll* loads all libraries at the specified key during initialization). Obviously, this technique can only be used for hooking applications which already use the *user32.dll* API.

[42]Technique described at `http://www.security.org.sg/code/apihookcheck.html`.
[43]Technique described at `http://www.codeproject.com/KB/system/hooksys.aspx`
[44]Within the *context* of an application means within the memory space allocated to it.

Chapter 3

Implementation

"The flim is okee-dokee."

- The Swedish Chef (The Muppet Show)

3.1 Preliminary development

This section describes the general high-level considerations and decisions made during the development of the PowerScan framework. This includes interpretation of the problem description, derivation of concise requirements for the implementation, choice of development process methodology, a brief discussion of the high-level architecture and finally some pseudo-code for core functionality.

3.1.1 System description and requirements

The main purpose of the system can be derived from the assignment text given in section 1.2:

Phase 1 Create functionality for automatically scanning a file with several anti-virus engines. The result should be aggregated and presented to the user.

Phase 2, part 1 Execute the malware while running real-time anti-virus software (on-access scanner). Aggregate and present the result.

Phase 2, part 2 Execute the malware and perform a dynamic analysis of its actions on the environment in which it is run.

The following sections further specifies the requirements and wishes for each of these phases.

Phase 1

Phase 1 does not offer any new functionality compared to existing products. One example of software which supplies this functionality is OPSWAT's Metascan, described in section 1.7.1. Although the functionality required here is relatively simple and could easily be implemented using one computer, a more comprehensive solution using distributed scanning in several virtual machines should be chosen, in order to accommodate for the later phases.

The following core requirements are identified for this phase:

1. The application must accept a file input.
2. The application must be able to return scan results from a number of different scan engines.

The following functionality *should* be implemented:

1. Result filtering for each anti-virus engine (meaning that the result should be presented in some relatively easy-to-read manner).
2. Parallelization of scan operations.
3. Support for an arbitrary (and unrestricted) number of scan engines.
4. Possibility to plug in new anti-virus engines without recompiling code.
5. Extensibility to allow relatively simple transition to phase 2 (code-wise).
6. Automatic update of virus definitions.
7. Registration and representation of date of last update of malware definitions when showing the scan results.
8. Time-outs for potentially blocking operations.

The following functionality *may* be implemented:

1. Graphical user interface.
2. Automatic setting of scan machines in *ready* state, i.e. with fully updated virus definitions and ready to perform scans (powered up, logged in and with fully updated definitions).

Phase 2, part 1

Given good design choices in phase 1, phase 2 part 1 should be relatively simple to implement. Within a distributed environment, the necessary operations will be copying and executing files and reading results of operations. The required operations are similar for these phases, only some details such as the order of the operations and the set-up of the environment differs.

The following core requirements are identified for this phase:

1. The application must accept a malware sample and execute it while a real-time anti-virus engine is running.

2. The application must be able to report the result of this operation for a number of different anti-virus engines.

3. The application must reset the system after executing the malware to prevent any permanent harm to the host system.

Additionally, the system should display the general characteristics described for phase 1.

Phase 2, part 2

The following core requirements are identified for this phase:

1. The application must accept a malware sample, execute it and perform dynamic analysis on some aspects of its execution.

The following functionality *should* be implemented:

1. Support for dynamic analysis of several aspects of the malware's execution:

    - Changes to the local file system.
    - Registry changes.
    - Network traffic (what is sent, which protocol is used and where is it sent).

The following functionality *may* be implemented:

1. Support for extensible plug-in of command-line based dynamic analysis tools.

3.1.2 Choice of high level architecture

In the early stages of the project, two distinct high level architectures are discussed. The first relies on resident code in each guest OS, with the architecture shown in figure 3.1, while the second fully depends on VMware and the functionality offered through the VMware VIX API, as shown in figure 3.2. Each architecture has its strengths and weaknesses, as discussed below.

The main strength of the first architecture, shown in figure 3.1, is that it does not rely much on the underlying technology; all necessary code is custom written as part of the application and the only external functionality needed is the ability to capture and restore the state of the OS (before and after running the malware, respectively). This also mitigates some of the problems discussed in section 2.3, where malware hides its functionality when it detects that it is being run in a virtualized environment, as this architecture does not require that the malware is executed in a virtual machine. In practice, however, running such a system without using virtualization would be highly impractical, as the administrative overhead of maintaining snapshots on several physical computers simultaneously would be significant. Weaknesses of the design also include the fact that resident code on the machine on which the malware is copied and executed may be attacked and results manipulated. Also, significant work would have to be put into implementing functionality already offered by virtualization software such as VMware (file transfer, file execution, state restoration and more).

The main strength of the second architecture, shown in figure 3.2, is that it utilizes the already implemented functionality in VMware to perform tasks that would be relatively time-consuming to implement from scratch. Also it does not rely on any resident code in the guest OSs, leaving no code for malware to attack. Thus the only way the system can be compromised when allowing the malware to execute is through a direct attack on the virtualization environment. Although possible, this is one of the things that environments such as VMware aim to prevent. A weakness of this design is that it to a large degree relies on a vendor-specific implementation (namely VMware's VMware Server and VIX[1]). This can be mitigated somehow through compartmentalization of the design, which means that another virtualization solution may be implemented without too much effort - typically by replacing the VMware VIX API module and keeping the same interface[2].

[1]VIX is an automation API for VMware Server and Workstation, described in some more detail in section 3.1.3.

[2]Obviously, such a replacement operation would require that the new virtualization

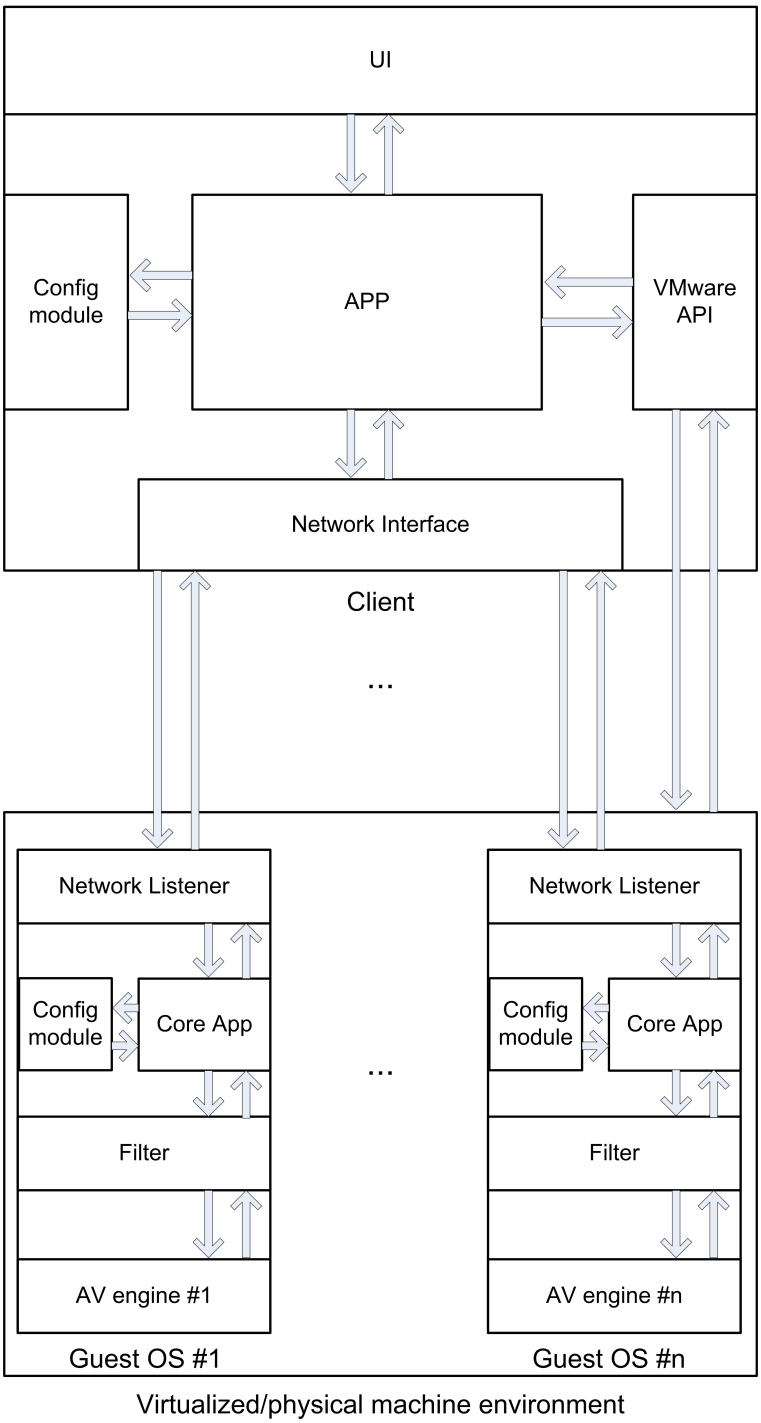


Figure 3.1: High-level sketch for the first (and unused) architecture.

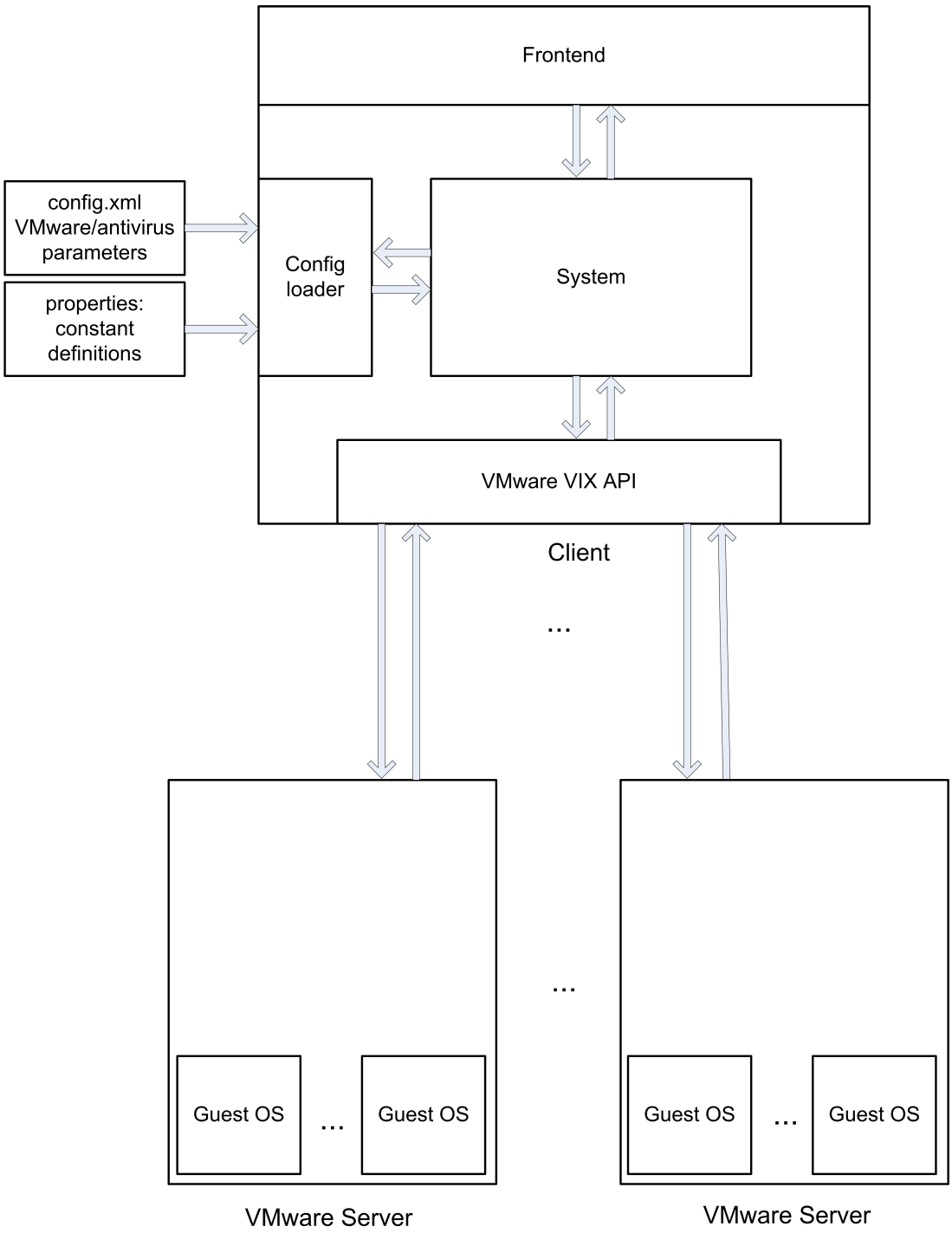


Figure 3.2: High-level sketch for the chosen architecture.

Although there exists several possible full machine emulators[3] and virtualization environments[4], VMware's solution is chosen, as it has a comprehensive user community and therefore is likely to offer good security (which obviously is vital when executing files known or suspected to contain malicious code on the guest OSs) in addition to a well-documented API with automation for the functionality needed.

3.1.3 Architecture description

As described in the previous section and shown in figure 3.2, the architecture chosen for this system does not rely on any code residing on the OS responsible for executing malware. The only requirement is that the virtual machines have an add-on application called VMware Tools installed. This program is part of the VMware application suite and not part of the PowerScan framework. In this manner, the PowerScan framework is independent of the virtualization servers. The following elements can be identified from the high-level architecture:

VMware VIX API The VMware VIX API[5] is a library that offers automated access (meaning that it can be called from other applications) to the functionality offered by VMware Server and VMware Workstation software. VIX comes in three different versions, one for C, one for Perl and one for COM. Thus, an application using the library would either have to use one of these languages or use some kind of binding. A weakness with VIX is that it does not support interaction with the GUI of the guest OSs, meaning that any tool used will have to support command line usage.

Config loader The config loader is responsible for two different configuration files; one XML file containing info about the operational environment (which anti-virus engines are running, which VMware hosts they are running on, usernames, passwords, executable paths and so on) and one file containing constants needed in the system (such as time-outs, log paths and so on).

solution supports similar functionality to that of VIX.

[3]See for example Qemu at `http://fabrice.bellard.free.fr/qemu/`.

[4]See for example the Xen Hypervisor at `http://www.xen.org/` or the Real-Time Systems Hypervisor at `http://www.real-time-systems.com/real-time_hypervisor`.

[5]The VIX API Reference Documentation can be found at `http://pubs.vmware.com/vix-api/ReferenceGuide/`.

Frontend The frontend (in addition to the config loader) is responsible for handling the interaction between the user and the system. Initially, the frontend will be implemented as a command-line program which accepts a number of command-line arguments and prints the result to console. This component should be written in a fashion which makes implementing other interfaces, such as a GUI, an overcomable task.

System The relatively undescriptivly named "system" component should contain the logic needed to perform the desired tasks by combining the other components of the application. An example of a typical "system" operation would be to read in environment configuration from the config reader, connect to the appropriate VMware hosts, copy a malware sample to the guest OSs running on the hosts, perform scan on the guest OS and return the result to the user.

3.1.4 Pseudocode for system operations

This section contains pseudo code showing the general system behavior for some selected tasks. The abstraction level of the pseudo code is based on the functionality offered by the VIX API [31] and thus shows how VIX can be used to offer the desired functionality of the system.

Perform scan in guest OS

```
scan(scanEngine, malwareSample){
  // Take snapshot to ensure that there exist a snapshot
  // of the clean state.
    if(noSnapshotPresent)
    vix.takeSnapshot();

  vix.copyFileToGuestOS(malwareSample);

  vix.executeFile(scanEngine +" "+ malwareSample +"> result.log");

  resultFile = vix.copyFileFromGuestOS("result.log");

  vix.revertToSnapshot();

  result = LogFilter(resultFile);

  return result;
  }
```

Listing 3.1: Pseudocode for scan operation

Execute malware in guest OS running real-time antivirus software

```
executeMalware(logLocation, malwareSample){
  // Take snapshot to ensure that there exist a snapshot
  // of the clean state with a real-time scanner running.
  if(noSnapshotPresent)
    vix.takeSnapshot();

  vix.copyFileToGuestOS(malwareSample);

  // Assume real-time scanner is already running
  vix.executeFile(malwareSample);

  // Allow malware to execute some given time
  sleep(executionTimer);

  resultFile = vix.copyFileFromGuestOS(logLocation);

  // Interrupt execution of malware sample by reverting to snapshot
  vix.revertToSnapshot();

  result = LogFilter(resultFile);

  return result;
  }
```

Listing 3.2: Pseudocode for the execute malware operation

Execute malware in guest OS running analysis tools

```
executeTools(malwareSample, tools, sleepTime){

  // Take snapshot to ensure that there exist a snapshot
  // of the clean state.
  if(noSnapshotPresent){
    vix.createSnapshot();
  }

  // Copy the malware sample to the remote system
  vix.copyFileToGuest(malwareSample, remotePath);

  // Log in with the console user so that the running
  // programs are visible to the operator.
  vix.logInGUIConsoleUser();

  // Start tools
  for(analysisTool in tools){
    vix.runProgramInGuestNonBlocking(analysisTool);
  }

  if(malwareSampleNeedExplicitExecution){
    vix.runProgramInGuestNonBlocking(malwareSample);
  }

  // Allow the operator time to interact with the
  // analysis tool and the malware.
  sleep(sleepTime);
```

```
for(analysisTool in tools){
  results.add(vix.copyFileFromGuest(analysisTool.getResultFilePath()));
}

vix.revertToSnapshot();
}
```

Listing 3.3: Pseudocode for the dynamic anlysis operation

3.2 Implementation description

This sections describes how the system was implemented and provides discussion on some of the decisions made during the implementation.

3.2.1 Choice of programming language

In a development project, deciding on the programming language to use for the implementation is one of the first decisions that has to be made. Two main factors influencing the choice in this implementation can be identified:

1. The chosen language must be able to communicate with the VIX API.
2. The developers' (that is, the authors of this thesis) experience with the language.

The first factor did not severely reduce the options; as mentioned in section 3.1.3, VIX is implemented in C, Perl and COM. A Python wrapper called pyvix[6] has been written for the C library, enabling Python support. Similarly, a C++ application should be able to call the C library without too much effort. Lastly, Java offers connectivity with (among others) C or C++ libraries through the usage of the Java Native Interface (JNI). There are probably several other languages that could be used to call the C library as well, but this is not discussed any further, as the number of languages already identified should suffice.

The second factor was in reality far more limiting than the first; although the developers possess some knowledge in C, C++ and Python, the team's programming experience is heavily biased toward Java. Although learning more of one of these languages would have been an overcomable task, it was decided that it was better to stick with the most well-known language and thus be able to direct greater efforts at actually writing good code and maintaining a good design. For connecting Java to the C library, external code was decided used, as direct usage of JNI is a non-trivial task[7]. Two third-party solutions were considered for this part; the commercial J/Invoke[8]

[6] `http://sourceforge.net/projects/pyvix`

[7] For example, it requires the programmer to program C or C++ code to create bindings between different data types.

[8] J/Invoke - easy Java native interoperability, `http://www.jinvoke.com`.

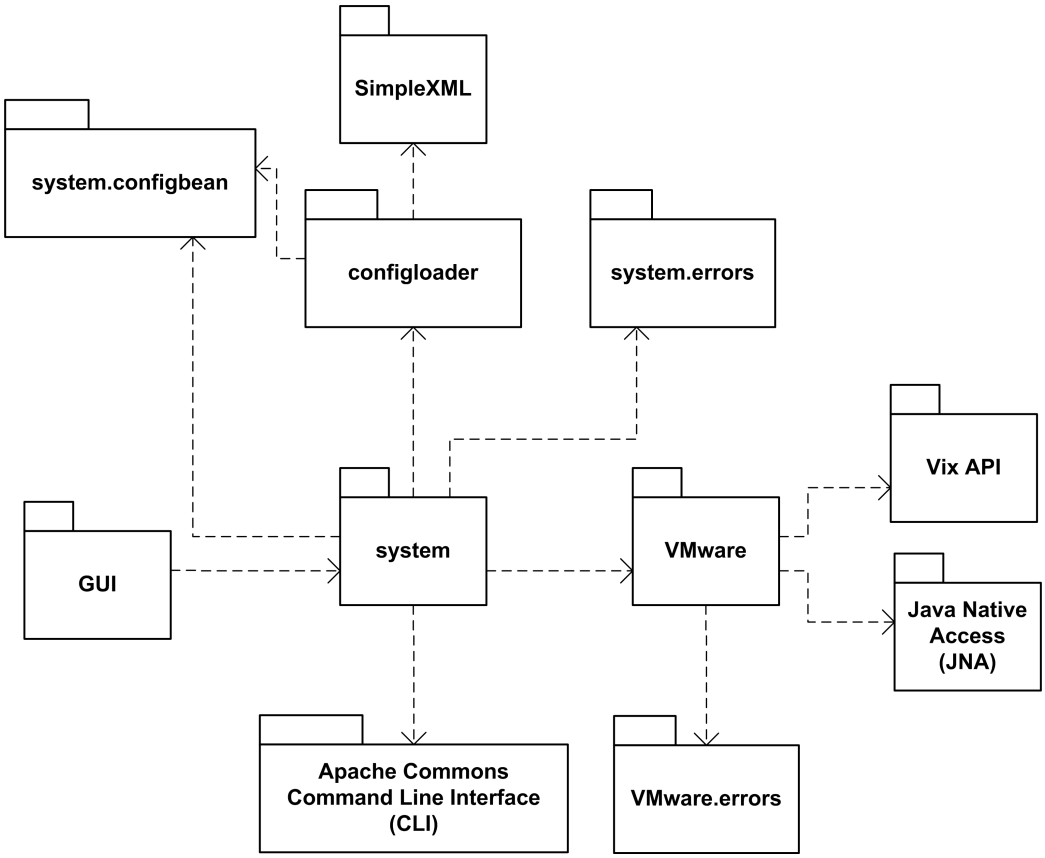


Figure 3.3: Overall package diagram of the PowerScan framework.

and the open source Java Native Access (JNA)[9]. Both were tested, and it was decided to use JNA as it is free and open source and offers almost the exact same functionality and usage value as J/Invoke.

3.2.2 Overall design

The chosen architecture closely reflects the preliminary one discussed in section 3.1.3. The overall composition of packages and external frameworks can be seen in figure 3.3. For a more comprehensive description of each package and its classes, refer to section 3.2.3.

[9]Java Native Access (JNA): Pure Java access to native libraries, `https://jna.dev.java.net`.

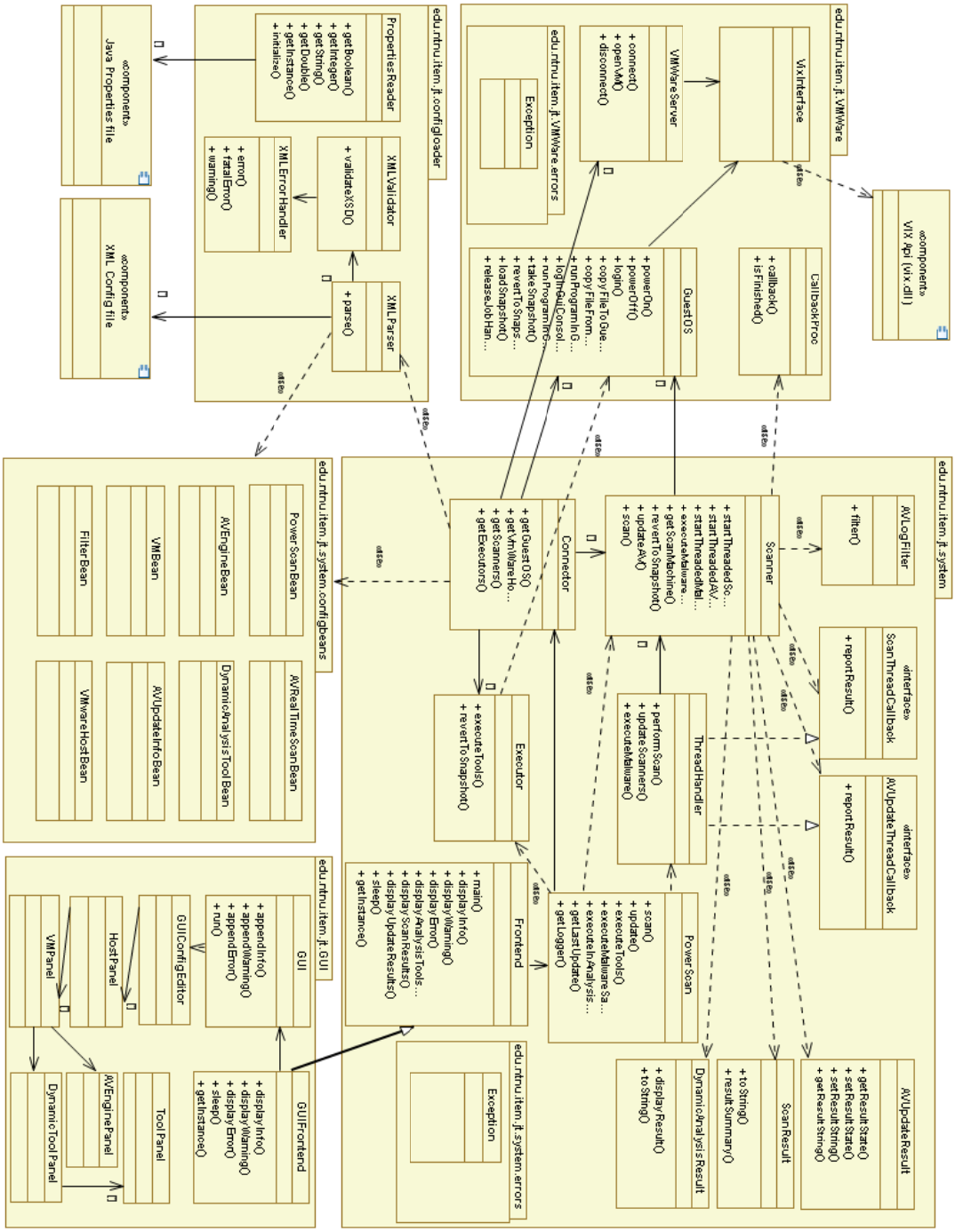


Figure 3.4: Class diagram showing the PowerScan framework.

3.2.3 Component description

This section describes the different packages involved in the PowerScan system implementation. The section is structured as follows; first, each package is given as a separate section, which starts with an introduction to the package and its functionality. For each package/section, a description of the most significant classes is given (if any are deemed significant enough to warrant a separate introduction at all). Note that the Javadoc documentation describing the different methods in the various classes are attached this thesis.

System package

The system package corresponds to the system component described in section 3.1.3; it contains the functionality needed to make the other components work together. Although this sounds like a simple part, it definitely is not; the system package is by far the most complex of the packages. This is in reality not so surprising, as the main purpose of the PowerScan system is to act as a framework and tie other components together. A class diagram of the package is given in figure 3.5.

The following gives a description of the most significant classes in the system package:

Scanner The Scanner class contains all logic associated with anti-virus engines. It combines parameters read from the XML configuration file with functionality offered by VMware VIX via the VMware package described later in this section. Significant public methods offered by Scanner include:

- scan() performs a scan operation of the supplied file with the associated anti-virus engine.
- startThreadedScan() starts a scan operation in a thread and reports result to a given callback function. For more information about threaded operations in PowerScan, see the discussion about threading in section 3.2.5.
- updateAV() updates the anti-virus engine's anti-virus definitions (if automated updates are supported by the engine associated with the scanner).

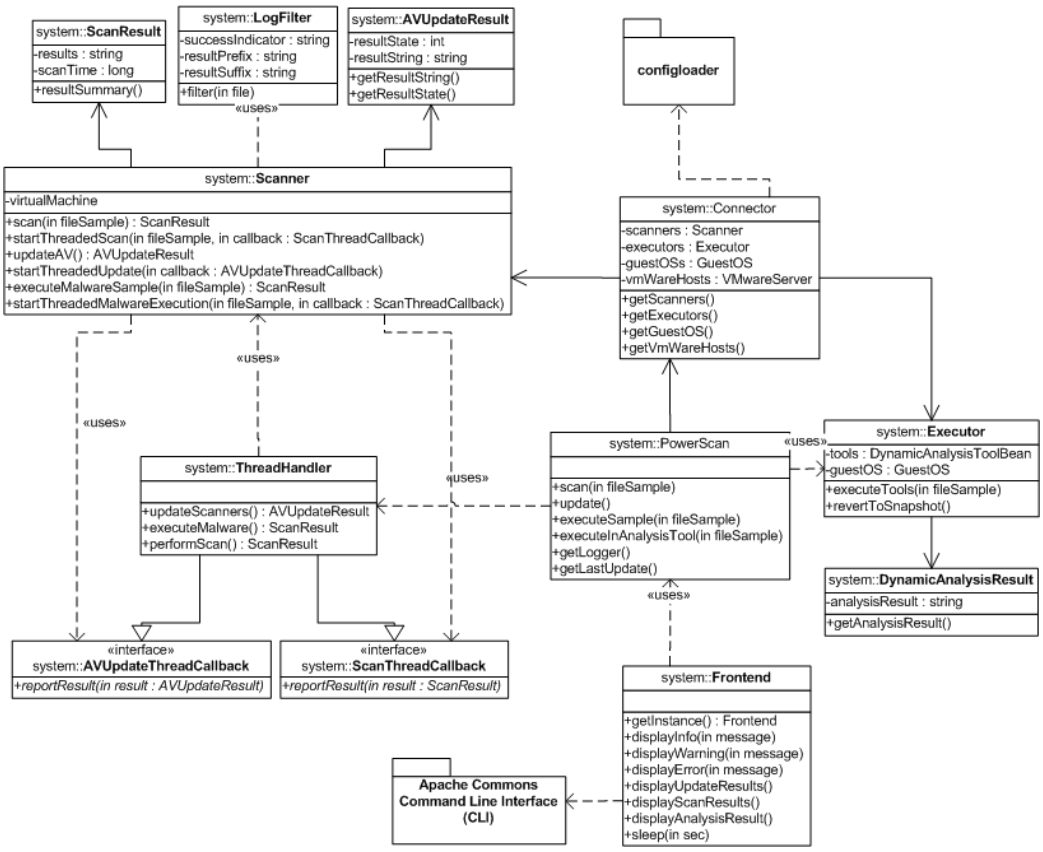


Figure 3.5: Class diagram for the system package. Note that not all functions and attributes are shown, only the ones deemed most relevant for understanding the package.

- startThreadedAVUpdate() starts an update operation in a thread and reports result to a given callback function.
- executeMalwareSample() executes a malware sample with a real-time anti-virus scanner running in the background. The intention of this operation is to increase the likelihood to detect and correctly classify malware employing the hiding techniques described in section 2.4.
- startThreadedMalwareExecution() starts malware execution in a thread and reports the result to a given callback function.

Executor The Executor class is similar to the scanner class, in that it represents the set of dynamic analysis tools running on one virtual machine. Although this can also be seen as a difference; a Scanner object represents exactly *one* scanner running at exactly *one* virtual machine, while an Executor represent an arbitrary number of tools running on *one* virtual machine (this is done as different analysis tools usually are not conflicting, while anti-virus engines usually are). Note that unlike the Scanner class, the Executor class does not perform (or support) operations in parallel. This is done to give the malware analyst the possibility to interact with the system[10] while the sample is running, in order to manipulate both the analysis tools and the malware itself. The sleep period that the system allows interaction is configurable, and may be interrupted using the console or GUI when the system is running and the operator does not need any more interaction time. Some important public methods in the Execution class include:

- executeTools() start execution of all the analysis tools registered with one virtual machine.
- revertToSnapshot() reverts the virtual machine to a previously stored snapshot.

PowerScan The PowerScan class is a central class used to tie the different system components together. This typically means that it accepts UI parameters such as paths to the configuration files from the Frontend, instantiates the Connector (see description below), starts the appropriate action (scan, update, execution or dynamic analysis) and returns the result. This is the class that should be used if PowerScan is to be used as a library. Significant public methods offered by PowerScan include:

[10]This means logging in to the virtual machine using for instance the VMware Server Console.

- scan() performs a surface-scan on the given malware sample file using the registered anti-virus engines.
- update() attempt to update all scanner registered in the system.
- getLastUpdate() get the date of the last call to PowerScan.update() from this client.
- executeMalwareSample() execute the supplied malware sample on all registered scanners.
- executeInAnalysisTool() run the supplied malware sample in all the registered dynamic analysis tools.
- getLogger() returns a logger object for the calling class.

Connector The Connector is responsible for translating the simple configuration beans representation of the surrounding environment to actual live and connected objects. This is done by reading information about the available anti-virus engines, analysis tools and so on from the configuration beans (which are created by the XMLParser class using the Simple XML framework) and instantiating the corresponding objects.

ThreadHandler The ThreadHandler is responsible for managing threaded scanner operations. This includes starting the requested operations (such as for example scan), playing the role as callback function for the threads when the operation finishes, keeping track of which scanners has completed, keeping track of timeouts, performing necessary operations on timed out scanners (typically reverting them to previously taken snapshots) and so on. Significant methods offered by ThreadHandler include:

- performScan(), updateScanners() & executeMalware() which perform the self-explanatory operations on all Scanner objects registered with the ThreadHandler ("registration" of scanners is done in the constructor).
- reportResult() which are two callback functions (with different parameters) used by the different scanner thread executions. The functions are introduced through implementing two interfaces, ScanThreadCallback and AVUpdateThreadCallback (these interfaces will not be discussed in any further detail).

LogFilter The LogFilter is a generic filter that can be used to filter results from different operations performed by scanners. The log filter is presented with the result of an operation as a reference to a file containing

the log. The filter has three modes of operation; it can search for a line containing a given string (a success indicator) and return the string between some delimiters in that line (every line containing the success indicator will be processed), it can search for a given success indicator and return that entire line (every line containing the success indicator will be returned) or it can return every line in the result (in which case, its only functionality is to read a file to a string).

Frontend The Frontend represents a presentation layer between the user and the PowerScan system. The Frontend class itself comes with a command line interface, using the standard out stream (stdout) and the standard error stream (stderr), but it can also be extended to support other types of UI, such as for example GUI (and, indeed it *has* been extended to use a GUI, for more information see the GUI package). Significant classes of the Frontend are the different functions used for presenting the user with info, such as:

- displayInfo() which displays a regular information message.
- displayWarning() which displays a warning message.
- displayError() which displays an error message.
- sleep() which is a method that halts execution of the running thread for a given amount of seconds. This method is used when executing the analysis tools to allow operator interaction. The sleep method may be interrupter by user interaction.
- displayUpdateResults() which displays the results of the update operations on each guest OS.
- displayScanResults() which displays the results of the scan operations on each guest OS.
- displayAnalysisToolsResult() which displays the results of the analysis operations on each guest OS.

It was considered separating the Frontend class from the system package into a separate package called UI (along with the GUIFrontend and GUI classes), but as the different display functions of Frontend are called from throughout the system package, it was decided that it can be seen as a natural part of the system.

The Frontend is implemented using the singleton pattern, providing a single point which all in- and output must pass through. This also makes it relatively simple to implement other types of UIs as these can

simply be returned by the singleton's *getInstance()* function, as long as they inherit from the Frontend[11]. Making calls to the Frontend's display methods from different locations in the code is also simple, as it can be done on the form *Frontend.getInstance().displayInfo("The scan is okee-dokee.")*.

The System package also contains a subpackage, errors, which containd the relevant exceptions that can be thrown from the system package. These are simple self-explanatory exceptions which describe different error situations and will not be discussed in further detail.

VMware package

The VMware package contains the logic needed to perform the needed operations against VMware hosts and virtual machines. This is done by calling the VIX C API [31] using a JNA binding, as discussed in section 3.2.1. The package only contains four classes, but it is still central to the operation of the PowerScan framework; all the functionality offered by PowerScan requires the usage of VMware Virtual Machines, and the VMware package contains all the code needed to "talk to" VMware. For a class diagram of the VMware package, see figure 3.6.

The classes of the VMware package are:

VixInterface The VixInterface is an interface for the functions from VIX that are used in the system. This interface is required by JNA in order for it to be able to do mapping between the C API and Java code. The class simply contains skeleton functions corresponding to the similar ones in VIX, only with Java data types. VixInterface is discussed in some more detail in the JNA description in section 3.2.4.

GuestOS The GuestOS class represents an instance of a virtual machine associated with the system. It implements a subset of the VIX API, namely the subset which represents the (relevant) operations that can be performed on a virtual machine. These functions include:

[11]Note that this was not as great a success in practice, as inheritance from singletons is problematic (because private constructors can not be overridden). In consequence, the Frontend has a protected constructor, which violates the singleton pattern. In retrospect, it is seen that other solutions for this, such as the usage of a static class, should have been researched.

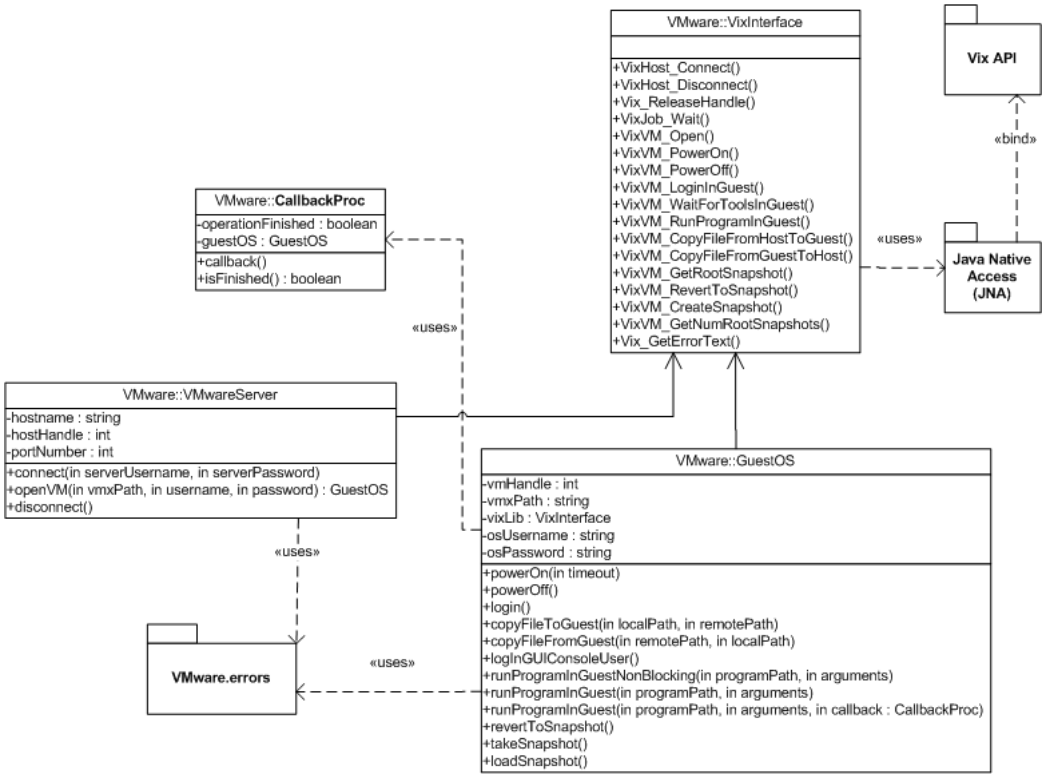


Figure 3.6: Class diagram for the VMware package. Note that not all functions and attributes are shown, only the ones deemed most relevant for understanding the package.

- powerOn() powers on the virtual machine.
- powerOff() powers off the virtual machine.
- login() logs in to the virtual machine with the username and password given in the constructor of the GuestOS class.
- copyFileFromGuest() copies a file from the virtual machine guest OS to the client.
- copyFileToGuest() copies a file from the client computer to the guest OS on the virtual machine.
- runProgramInGuest() runs a program in the guest OS.
- takeSnapshot() takes a snapshot.
- revertToSnapshot() reverts to snapshot.

The GuestOS class attempts to release all resources associated with it in VIX in its destructor.

VMwareServer The VMwareServer class represents a server on which virtual machines are hosted. Similarly to GuestOS, each instance of VMwareServer represents an instance of VMware Server running on a physical server, and implements a subset of relevant operations on VMware Servers from the VIX API. These operations are:

- connect() connects to the VMware Server.
- openVM() loads a given virtual machine running on the VMware Server. This operation returns a GuestOS instance.
- disconnect() disconnects from the VMware Server.

CallbackProc The callback procedure is an implementation of the callback functionality offered by the JNA API, which allows an operation to be performed against VIX in a non-blocking manner. This allows the calling function to perform other operations while waiting for the current operation to finish. It can also be used to prevent the system from locking in case the called operation never returns (which can happen when using VIX). In practice, the common usage of this functionality within PowerScan is to start one or more operations, then sleep for some time, check whether the operation(s) has finished, if not, sleep again. This is then repeated either until timeout or the operation(s) finish(es).

Note that it would be possible to interact with the VIX API using only the VixInterface class - the GuestOS and VMwareServer classes are there to add object orientation and to simplify usage[12].

Similarly to the system package, the VMware package has a subpackage called *errors* which contains simple self-explanatory exceptions. These also have the capability of carrying an error-code and the corresponding textual description of the error, as returned from VIX.

configbeans package

The configbeans package[13] contains simple beans which store an object oriented view of the information represented in the XML configuration file. The

[12]Indeed, the introduction of these classes heavily simplifies the usage, as it allows the programmer to ignore the finer details of the non-object oriented C API.

[13]Which in fact is a subpackage of the System package.

config beans are used as a means to be able to use the Simple framework discussed in section 3.2.4 to automatically parse the XML configuration. The config beans are then later used by the Connector class of the system package to instantiate the needed objects. The beans are built to closely reflect the relations between different elements in the PowerScan in the same manner as the XML. For a class diagram over the configbeans package, refer to figure 3.7.

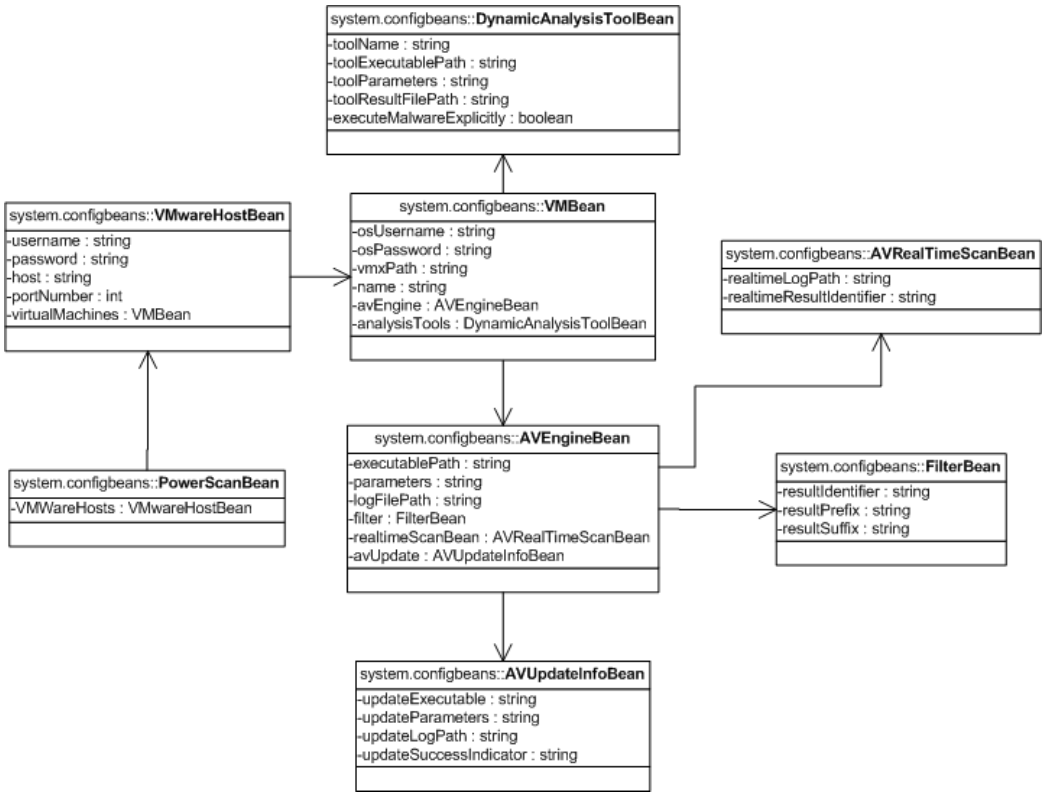


Figure 3.7: Class diagram for the configbeans package. Note that the only functions in this package are setters and getters, which are ommitted in this diagram.

configloader package

The configloader package is the implementation of the config loader described in the high level architecture discussed in section 3.1.3. Thus it is responsible both for reading constant definitions from the properties file and parsing the XML to a usable format for the system. The latter operation is definitely

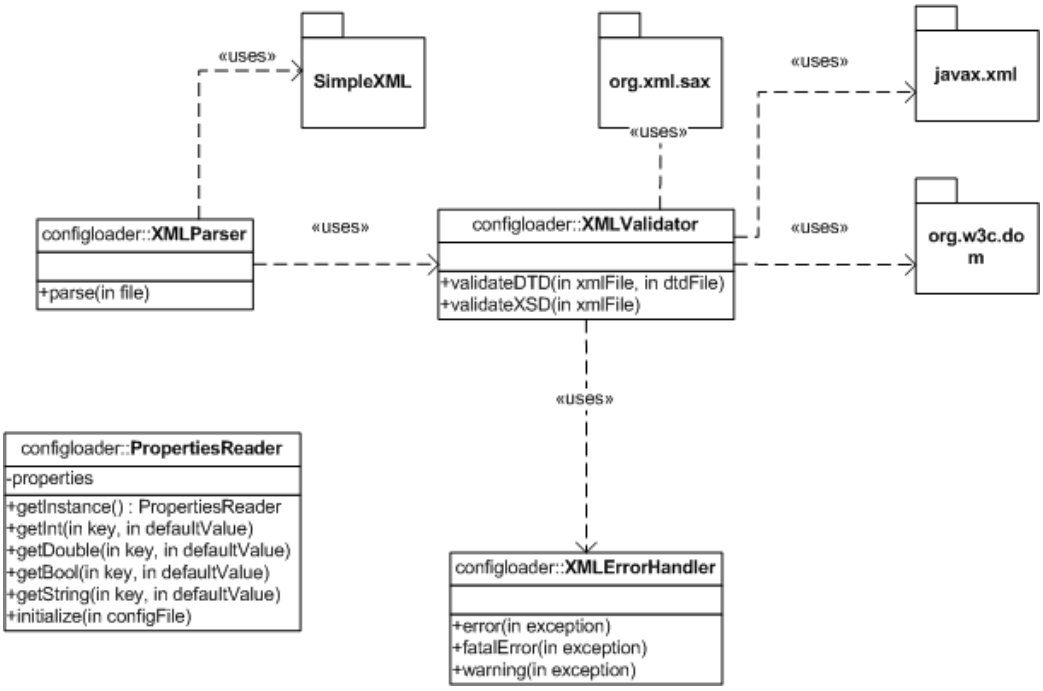


Figure 3.8: Class diagram for the configloader package. Note that not all functions and attributes are shown, only the ones deemed most relevant for understanding the package.

the most complex of the two, and involves the usage of the Simple framework described in section 3.2.4 and the configuration beans described earlier in this section. The package also contains functionality for verifying the correctness of the XML using both the XSD and the DTD validation schemes. For a class diagram of the config loader, refer to figure 3.8. A description of the validation scheme used for the XML configuration can be found in section 3.2.5.

The most significant classes of the config loader package are:

PropertiesReader The PropertiesReader is responsible for reading properties from a Java properties file. For an example properties file and a description of its fields, refer to section D.4. The properties reader is implemented as a singleton, meaning that there can only exist one instance of it at the time. The properties are either read from a explicitly given file given as an initialization parameter or attempted read from the default properties file location (`config\powerscan.proper-`

`ties`[14]). The constants are read from the properties file using a key and a hard coded default value (used in case the key can not be found or the properties file can not be read), with different getter functions based on the type of value (*int*, *boolean* and so on). The properties reader is implemented as a wrapper with some extra functionality on top of *java.util.Properties*.

XMLParser The XML parser is basically just a wrapper for the Simple XML framework described in section 3.2.4 and does not provide much extra functionality besides wrapping in the framework and calling the validator. See section 3.2.4.

XMLValidator Performs validation an XML file based on an XSD or DTD schema[15]. See section 3.2.4 for details.

GUI package

The GUI package provides a graphical user interface extension to the Frontend class described in the system package. This includes a main window for performing the major operations of PowerScan and a configuration editor. The configuration editor provides a graphical user interface for editing the XML configuration file which describes the operational environment of the framework. This functionality is naturally closely integrated with the config beans, which are used for storing information both as it is written to and read from the GUI[16]. Using the config beans in this manner allows seamless integration between the GUI and the actual XML file, utilizing the Simple framework described in section 3.2.4 via the XMLParser class. Before XML created or modified using the configuration editor is saved to file, it is validated using the XMLValidator from the configreader package. The class diagram for the GUI package is shown in figure 3.9.

The most important classes of the GUI package are:

GUI This GUI class contains all the code needed for the main GUI window.

[14]The separator is adjusted to the platform, so the default file will be read correctly on other OSes as well.

[15]During implementation of the system, both XSD and DTD schemes were created, but in the end only the XSD schema was used. XSD was chosen as it is the newest of the two schemes and is generally perceived to be the most powerful.

[16]In fact, classes extending Java's *JPanel* keeps references their related beans for easy access.

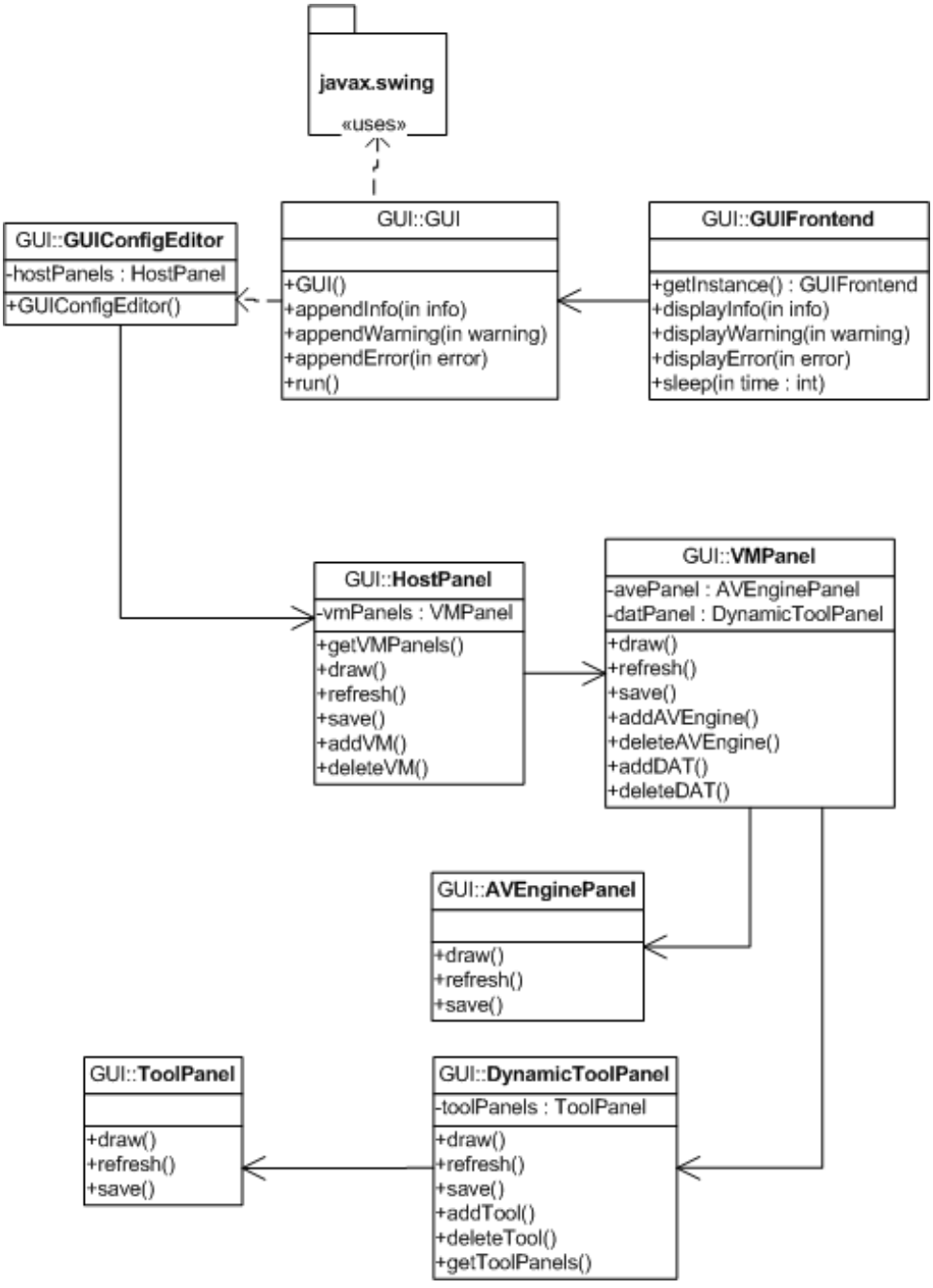


Figure 3.9: Class diagram for the GUI package. Note that not all functions and attributes are shown, only the ones deemed most relevant for understanding the package.

It starts operations by making calls to the PowerScan class in the same manner as the Frontend class. Additionally, it provides functionality for storing the text written to the console window. Whenever an operation is started from the GUI (such as scans, updates and so on) a new thread is launched for execution. This is done to ensure that the GUI does not lock up while the execution is controlled by the started operation, and allows the GUI to display progress messages during execution. The GUI has the following important public methods:

- appendInfo() is used by the GUIFrontend class to append information messages to the user in the console text area.
- appendWarning() is used by the GUIFrontend class to append warning messages to the user, by appending the messages to the console text area.
- appendError() is used by the GUIFrontend class to append error messages to the user, by appending the messages to the console text area and in addition present the user with an error dialog box.

GUIFrontend This GUIFrontend class extends the Frontend class, and overrides the display methods defined there, so that the information is shown in the GUI console area instead of being sent to the standard out stream.

GUIConfigEditor This is the main class in the config editor system, which also represents its main window. Additionally, it holds all the menu items needed to edit the XML configuration.

3.2.4 Frameworks and third party code used

The PowerScan framework utilizes a number of third party frameworks in order to offer the desired functionality. It has been sought to as often as possible use publicly available open-source frameworks which have been around for a while, in order to minimize cost and maximize quality.

Java Native Access

As mentioned earlier, communication between the PowerScan framework and the VMware virtual machines is achieved using the VMware VIX C API. To

be able to communicate with the virtual machines, it was necessary to use this API, available as a dll-file for Windows. The chosen solution was to write a wrapper class in Java and call the VIX C API. Three solutions were considered for this problem; Java Native Interface[17] (JNI), Java Native Access[18] (JNA) or J/Invoke[19]. JNI requires knowledge of the languages on both sides, as type conversions and so on must be done manually. JNA and J/Invoke removes this knowledge requirement by largely performing conversion automatically[20]. J/Invoke and JNA offers basically the same functionality, the biggest difference being that J/Invoke is a commercial product, while JNA is an open-source project. After performing some tests and realizing that both products suited the needs here well, it was decided to use the open source alternative.

JNA requires a wrapper class, used for conversion, which is written as a Java interface. An example of how the wrapping can be done is shown below[21]:

```
public interface VixInterface extends com.sun.jna.Library{

    VixInterface INSTANCE = (VixInterface) Native.loadLibrary("vix",
        VixInterface.class);

    public int VixVM_Open(int hostHandle,
          String vmxFilePathName,
          Callback callbackProc,
          byte[] clientData);

}
```

Listing 3.4: Example of a JNA interface - VixInterface - with one function.

The actual implementation of the interface defines all relevant functions from the vix.dll file, redefined to Java data types. The mapping and conversion of Java data types to native C data types is done by the JNA framework.

[17]Sun Microsystems JDK6 Java Native Interface -related APIs and Developer Guides - `http://java.sun.com/javase/6/docs/technotes/guides/jni/index.html`

[18]Java Native Access (JNA): Pure access to native libraries - `https://jna.dev.java.net/`

[19]J/Invoke | easy Java native interoperability - `http://www.jinvoke.com/`

[20]Some mapping is required, but this is trivial compared to the effort required when using JNI.

[21]Naturally, the actual interface contains more functions, the code listed is only used for illustrating the principle.

Simple

According to the SimpleXML website, "*Simple is a high performance XML serialization and configuration framework for Java. Its goal is to provide an XML framework that enables rapid development of XML configuration and communication systems. This framework aids the development of XML systems with minimal effort and reduced errors.*" In other words, the Simple framework focuses on providing an easy entry point to XML when using Java. In this usage, this was initially a significant advantage, but it later turned out that Simple may have been a bit too simple, as it did not provide any functionality in terms of validation.

In PowerScan, Simple is used to deserialize XML to JavaBeans[22] (located in the configbeans package), which are then used by the Connector class to instantiate the actual objects representing the entities used by the rest of the system. It is also used for serialization by the XML config editor in the GUI, when storing the entered information to file. Simple does not require any configuration besides annotations in the config beans that tells the framework which elements and attributes from the XML file maps to which objects in the bean classes.

The following is an example of annotations from the AVEngineBean class, which carries information about an anti-virus engine[23]:

```
@Attribute
private String name;

@Element(name="avExecutablePath")
private String executablePath;

@Element(name="avParameters")
private String parameters;
```

Listing 3.5: The AVEngineBean annotations

Apache Commons CLI

The Apache Commons Command Line Interface[24] (CLI) is used to offer the command line interface capability of PowerScan. The Apache Commons CLI

[22] JavaBeans are objects that are used to carry other objects, and contains no other functions than setters and getters for the encapsulated objects.

[23] Note that the setters and getters are omitted.

[24] Apache Commons CLI home - `http://commons.apache.org/cli/`

API provides a simple way to automatically parse command line parameters, print help messages and more. The API is used by defining a set of command line options, which can be optional or mandatory, may accept a parameter and has a corresponding description. The following is an example of usage of options taken from the Frontend class of PowerScan[25]:

```
Options options = new Options();

options.addOption("GUI", false, "Determines whether to use a GUI.");
options.addOption("scan", true, "Performs a surface scan of the supplied
    malware sample with all scan engines configured in the XML config file.
    ");
```

Listing 3.6: Apache Commons CLI usage example part I

After having defined the options, a parser is constructed to parse the Java main function parameters against the options[26]:

```
public static void main(String[] args){

 try{
  CommandLineParser parser = new PosixParser();
  CommandLine cmd = parser.parse(options, args);

  if(cmd.hasOption("GUI")){
   //Initialize GUI
   return;
  }

  if(cmd.hasOption("SCAN")){
   String malwareSample = cmd.getOptionValue(Frontend.AV_SCAN_OPTION);
   //Perform scan operation
   return;
  }

 }catch(ParseException e){
   // Inform user that parsing failed and print usage instructions.
   new HelpFormatter().printHelp("java -jar powerscan.jar [arguments].",
       options);
 }

 //If no recognized parameters was found, print usage information.
 new HelpFormatter().printHelp("java -jar powerscan.jar [arguments].",
     options)
}
```

Listing 3.7: Apache Commons CLI usage example part II

Execution would then result in the following output[27]:

[25]Note that the actual implementation contains more options than this example - it has been shortened for simplicity.

[26]This is also heavily simplified compared to the implementation.

[27]Obviously, running the command line with the -GUI or -SCAN switches would produce other outputs.

```
> java -jar powerscan.jar
usage: java -jar powerscan.jar [arguments] <malware sample>.

 -GUI                   Determines whether to use a GUI.
 -SCAN <arg>            Performs a surface scan of the supplied malware
                        sample with all scan engines configured in the XML
                            config file.
```

Listing 3.8: Apache Commons CLI usage example part III

Apache Xerces XML Parser

The Apache Xerces Parser is used to perform validation of the XML configuration file against a referred XML Schema Definition (XSD) file. This is done by running the XML document through the Xerces parser with the validation feature turned on. The validator will then check both that the XML document is well-formed and that it is valid. A well-formed XML document is a document that conforms to the XML language rules. This involves checking that all tags are ended and that nesting of elements is done properly. Validation, on the other hand, is done against a referred schema, and is about verifying that the XML document conforms to the schema. Restrictions that can be included in a schema are number of instances of an element or attribute, whether the elements and attributes are required or optional and allowed data types. If any of these these checks fail, an exception is thrown and handled by the application according to the severity of the error.

Although Xerces could probably be used for parsing the XML as well, it was decided to use Simple for this, as it provides the functionality in a very hassle-free way. Unfortunately, Simple does not support any strict validation (although some validation is done using annotations), which is where Xerces comes to its right. In a sense, using two frameworks for XML may be unfortunate, but the decision has pragmatic reasons: the framework which provided the desired functionality with the least effort was chosen in both cases.

3.2.5 Operation description

This section describes how significant components in the PowerScan framework works together to offer the functionality described in section 3.1. This section will mostly be related to significant functionality which has not been

described in sufficient detail in the package and framework descriptions, typically because it spans over several classes and/or packages.

Malware Execution

This section illustrates how execution of malware with a real-time anti-virus solution running in the background is performed. This is the only operation described here, as the others are conceptually similar, only with other parameters and function orders. The sequence chart can be seen in figure 3.10 and consists of the following phases[28]:

1. The Scanner object receives a call instructing it to start malware execution, with the malware sample and two booleans as parameters. The two booleans indicate whether snapshot should be taken before execution and reverted to after.

2. The scanner optionally takes snapshot before starting the operations.

3. The malware sample is copied to the virtual machine.

4. A callback procedure is instantiated and the guest OS is instructed to start execution of the malware sample and inform the callback procedure when execution has finished. The VIX API (via JNA) will call a given function in the callback procedure object once execution of the malware thread has finished. The malware is allowed to execute either until callback is received or until timeout.

5. The real-time anti-virus log file is then copied back to the PowerScan client machine and written to the scanner log file. A filter with the information given in the XML is then created and used to parse the log file to obtain the desired result. Then, a snapshot is optionally taken before the scan result is returned to the caller.

Note that the snapshot functionality of these functions are not used in ordinary PowerScan execution, as the ThreadHandler class handles snapshots (to prevent these operations from delaying presentation of operation results).

[28]Note that the sequence has been simplified to only show the relevant operations.

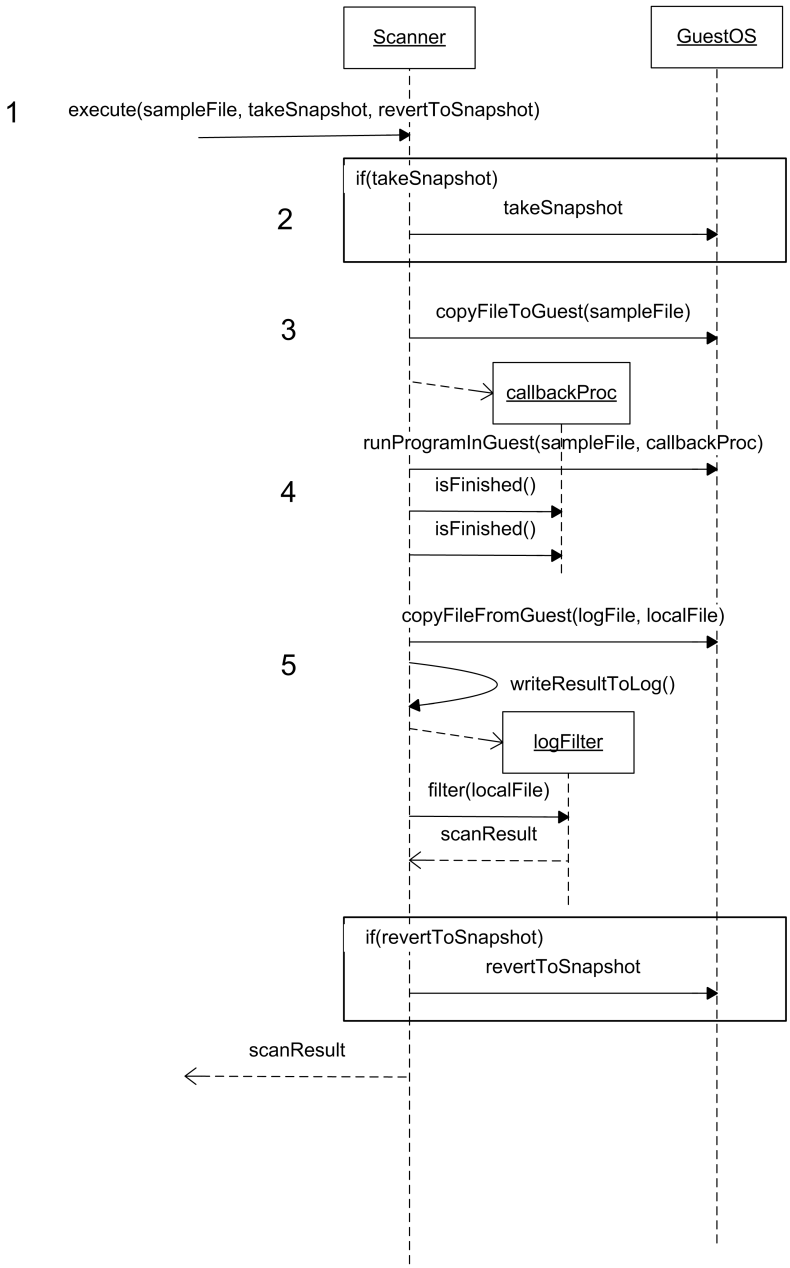


Figure 3.10: Sequence diagram showing malware execution with real-time anti-virus software running. Presumes that the system is already "up and running". Note that calls going to VixInterface and to/from the actual VIX API are omitted.

Threading

In order to be able to fully utilize the parallel nature of the involved virtual machines, PowerScan performs certain operations in parallel. As PowerScan supports an arbitrary number of virtual machines running on an arbitrary number of VMware hosts, using the virtual machines in sequence would be very inefficient. This section presents how scans are performed in parallel, but the exact same principles are utilized when updating the anti-virus software, executing malware samples with real-time anti-virus scanners running in the background and performing non-blocking operations in VIX. Thus, parallelization of the two latter operations will not be described in any further detail. For reasons discussed is section 3.2.3, dynamic tool analysis of malware is performed in sequence.

Figure 3.11 shows the (somewhat simplified) message sequence for a threaded scan. Note that this figure presumes that the system is up and running, meaning that the XML configuration has been parsed and objects has been instantiated accordingly. Similarly, some functionality has been omitted for readability. The operation consists of the following phases:

1. The Frontend class receives input from the user, instructing the system to perform a scan of a given file. The Frontend forwards this request to the central PowerScan class, which is responsible for "getting the job done". The PowerScan class then instantiates a ThreadHandler instance, which is responsible for managing the threads that will later be used for execution, and instructs it to start the scan.

2. The ThreadHandler instructs all registered scanners to start a threaded scan of the supplied file. The ThreadHandler also supplies a reference to itself, which will be used for sending callbacks when the threads finish (this reference is not shown in the figure).

3. After having initiated scans in all scanners, the ThreadHandler polls itself after a time interval (set in the properties file) to check whether all scanners has completed. This is repeated either until timeout or until all scanners are done (whichever occurs first).

4. If the timeout is reached before all the scanner threads have called back, the ThreadHandler assumes the operation has timed out, and reverts the virtual machines running any remaining scanners to its previously taken snapshot before returning the results to the PowerScan class.

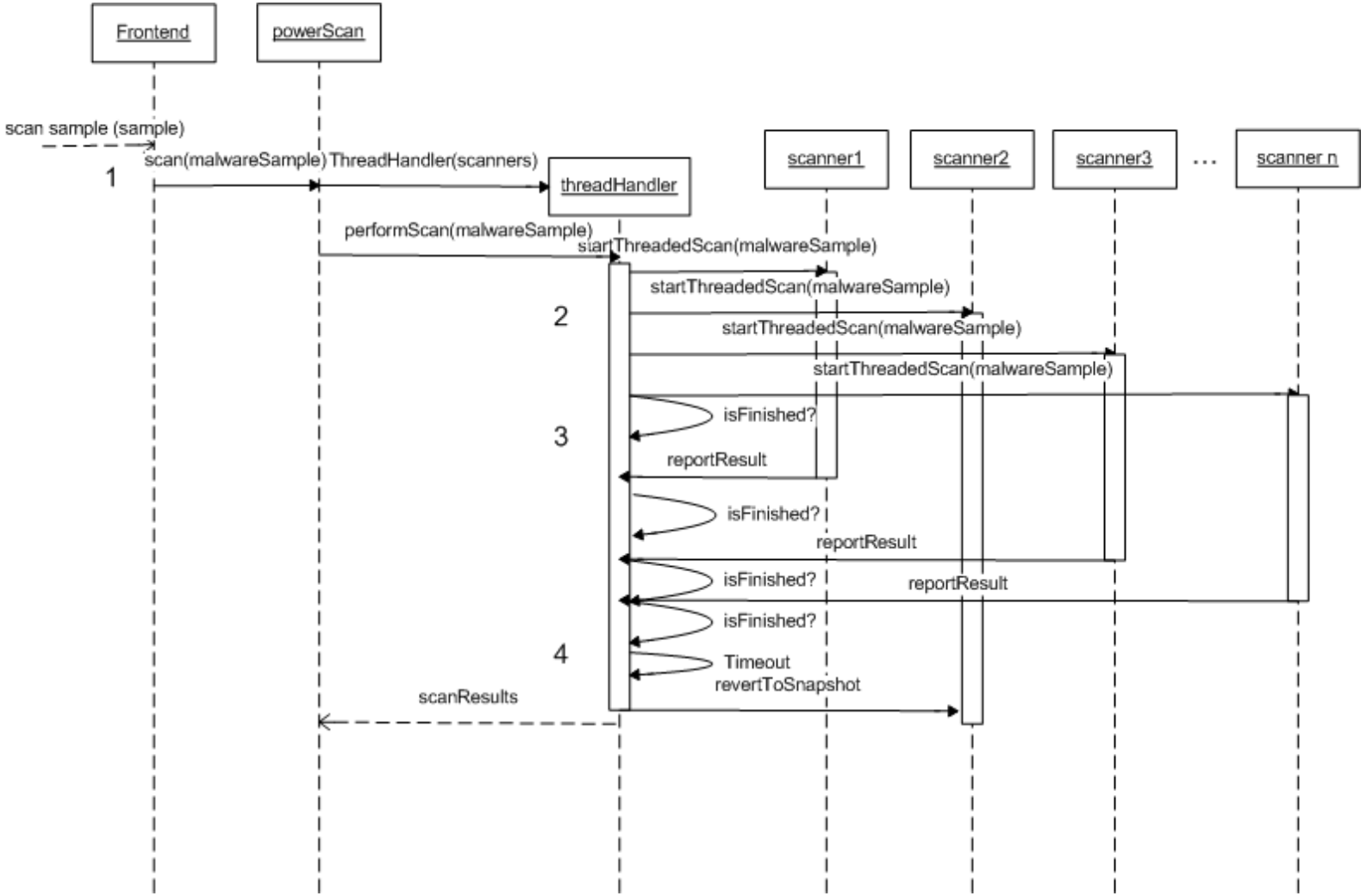


Figure 3.11: Sequence diagram showing a threaded scan operation. Presumes that the system is already "up and running," meaning that all scanners are loaded and ready.

Logging

Most significant classes in the PowerScan framework implement logging. The logging is implemented using the java *java.util.logging* package which comes with Java Standard Edition. The logging uses three of the six default levels supplied in Java, namely *Info*, *Warning* and *Severe*[29]. Logging classes fetch a static reference to the appropriate loggers and write to these objects[30]. Although the simple form *log.info("The scan is okee-dokee!")* can be used, most logging objects define their own private logging function, as it is useful to write some information about the context of the log message, especially when several threaded instances of the same objects (for example different Scanner objects) are writing to the same log file simultaneously. These private logger help functions append some context information and write the message to the log file.

XML config file

Configuration of the operational environment in the PowerScan framework is represented in the form of an XML-formatted configuration file. XML was chosen as configuration representation due to the fact that it is highly extendable, structured and gives a good overview in that it gives a logical presentation. It also allows the usage of validators to more or less automatically verify that the XML document follows given rules. The chosen solution for validation was to use W3C XML Schema Definition (XSD) as the description language, and hence the XML file is described with the help of an .xsd file. The W3C XSD language is based on XML, and is a more powerful and rich alternative to its predecessor, the Document Type Definition (DTD) language.

A description of the PowerScan XML based on the XSD schema can be found in section D.3. A description of the XMLParser and XMLValiator classes can be found in section 3.2.3.

[29] The log levels *Config*, *Entering* and *Exiting* are not used

[30] The *Logger* instance reference is fetched using the PowerScan class' *getLogger()* function.

Properties file

The other configuration file, properties, is used to set timeout values for different operations and some global paths. This, among other things, includes the log file paths to be used for the PowerScan log files on the local system. The properties file is read by the PropertiesReader class, which can parse the values from the properties file into strings, integers, doubles or booleans. Every call to the config reader contains a default value, such that if the config reader is unable to find the requested value the default value will be returned and used.

An example properties file can be found in section D.4. A description of the PropertiesReader class can be found in the component description in section 3.2.3.

3.3 System analysis and considerations

This section describes the characteristics of the implemented product. Where section 3.2 dealt with *how* the system is implemented, this section focuses on the implementation and its results and consequences.

3.3.1 Requirement analysis

This section performs a simple analysis of which of the initial requirements formulated prior to starting this project has been met in the PowerScan framework implementation.

Requirement analysis for phase 1

Recall the description of phase 1 identified in section 3.1.1: *"Create functionality for automatically scanning a file with several anti-virus engines. The result should be aggregated and presented to the user"*. For a description of each of the individual requirements, please refer back to section 3.1.1. The following describes the fulfillment of core requirements identified for phase 1:

Core requirement 1 This requirement is fulfilled. The application can accept any file input.

Core requirement 2 This requirement is fulfilled. The application can use multiple scan engines at once.

The following describes the fulfillment of recommended requirements (requirements that *should* be fulfilled):

Recommended requirement 1 This requirement is fulfilled. The application supports simple string-based filtering which in most cases should be sufficient to display the isolated result string.

Recommended requirement 2 This requirement is fulfilled. Scan operations are performed in parallel.

Recommended requirement 3 This requirement is fulfilled. Given a sufficient number of servers running VMware, any number of scan engines can be run at the time.

Recommended requirement 4 This requirement is fulfilled. New antivirus engines are plugged in using XML, and can thus be done without recompiling.

Recommended requirement 5 This requirement is fulfilled. Usage of one VMware virtual machine for each scan engine is an overkill for part 1, but makes the transition to part 2 relatively trivial.

Recommended requirement 6 This requirement is partly fulfilled. PowerScan supports automatic updates for most engines, although some need user interaction. This is because PowerScan is only able to perform operations via the command line.

Recommended requirement 7 This requirement is partly fulfilled. PowerScan stores the time and date of the last initiation of the update command in the registry of the client machine. Although this may not be the last time and date of the actual update operation for each scan engine, it should give a good estimate. This is especially true if only one client computer is used, as the time and date then represents the last time an update operation was explicitly initialized. It is, however, worth noting that the time and date may not be correct, as some antivirus engines update automatically when given the chance and because update operations may have been initiated through PowerScan on other client machines.

Recommended requirement 8 This requirement is partly fulfilled. The most likely operation to block, file execution, has been made non-blocking. Other time-consuming operations such as file transfers are not. This is because an unfinished (and thus aborted) file transfer is deemed less critical than execution of malware on the virtual machine.

The following describes the fulfillment of optional requirements (requirements that *may* be fulfilled):

Optional requirement 1 This requirement is fulfilled. Although not really planned implemented before the project started, a GUI has been implemented. This is made possible partly through the extendable implementation of the generic Frontend class.

Optional requirement 2 This requirement is fulfilled. Each time a PowerScan operation is started, the virtual machines are powered on and

logged in to. If the virtual machines are powered on before the operation starts, this step simply returns immediately. Performing the update operation will power up all virtual machines, log in, perform updates for all scan engines supporting this operation and then take a snapshot. This will lead to the "*ready state*" described in the requirements.

Requirement analysis for phase 2, part 1

Recall the description of phase 2, part 1 identified in section 3.1.1: *"Execute the malware while running real-time anti-virus software. Aggregate and present the result"*. For a description of each of the individual requirements, please refer back to section 3.1.1. The following describes the fulfillment of core requirements identified for phase 2, part 1:

Core requirement 1 This requirement is fulfilled. The system accepts a file input and executes it on every registered virtual machine running a scan engine that supports real-time scan.

Core requirement 2 This requirement is fulfilled. As for phase 1, PowerScan supports any number of scan engines.

Core requirement 3 This requirement is fulfilled. Both in case of success and failure, PowerScan reverts every virtual machine to the previously taken snapshot after performing its operations. This goes for all phases.

As the implementation is common, the system also display the characteristics described for phase 1.

Requirement analysis for phase 2, part 2

Recall the description of phase 2, part 2 identified in section 3.1.1: *Execute the malware and perform a dynamic analysis of its actions on the environment in which it is run.* For a description of each of the individual requirements, please refer back to section 3.1.1. The following describes the fulfillment of core requirements identified for phase 2, part 2:

Core requirement 1 This requirement is fulfilled. PowerScan supports insertion of dynamic analysis tools which can perform analysis of several aspects of its execution.

The following describes the fulfillment of recommended requirements (requirements that *should* be fulfilled):

Core requirement 1 This requirement is partly fulfilled. Due to the fact that the majority of analysis tools are GUI based, finding such tools which can be plugged easily into the PowerScan framework is somewhat problematic (see section C.1 for an analysis of some of the tools available). Luckily, such tools as the Norman Sandbox Analyzer gives a comprehensive analysis which can be copied back.

The following describes the fulfillment of optional requirements (requirements that *may* be fulfilled):

Core requirement 1 This requirement was fulfilled. PowerScan uses the same XML configuration for all phases, and thus any number of (command line interface based) dynamic analysis tools may be plugged in.

3.3.2 Functionality tests

When developing software, testing is a vital activity in all development phases. The objecting of testing is to gain confidence within two areas; validation and verification. Validation is about assuring that the software has the right functionality; that the software is written according to the requirement specifications. Verification is about testing that the implemented functionality is done correctly; that it works as required and actually produces correct output without errors.

Testing should be carried out at different stages in a development process, the most common being:

Unit testing Done at unit/class level and aims at testing all functions to verify the correctness of the implemented functionality in each component.

Integration testing After the various units/classes have been tested, interaction between the different units is tested.

System testing Tests the entire system in its intended production environment. System testing can be divided into several sub tests focusing on particular aspects of test system. Examples include security testing and stress/high load testing.

Acceptance test Carried out by the users/customers before the product is handed over from development to management/maintenance and put in production.

Testing can be conducted in two different ways, namely black box and white box testing. The former is carried out without any knowledge about the internal structure and workings of the system; the point is to check if a given input returns the expected output. The latter is planned and carried out with knowledge about the internals of the system test target.

In this project, the various classes and components have been tested at three different abstraction levels. First, classes and functions were tested during the development process in a white box manner. Then integration testing was performed as the components were combined into a system. These tests were performed during development using the main functions in the classes being tested. The main testing effort was however put in system tests, which have mostly been carried out in a black box fashion. Test specifications have been prepared and performed for testing the entire system using the command line interface, shown section E.1 and the graphical user interface, shown in section E.2. Additionally, tests were created for the GUI configuration editor, which can be found in section E.3.

Bugs and weaknesses discovered during testing were, naturally, sought corrected. Some weaknesses, however, were not rectified, either as they were outside the scope of this implementation[31] or because the effort required to correct them outweighs the advantages based on the limited time scope. These weaknesses are discussed in section 3.3.6.

[31] For example slight weaknesses in used frameworks and so on.

3.3.3 Requirements for external anti-virus software and analysis tools

In order to be used seamlessly in the PowerScan framework, there are some requirements that anti-virus software and dynamic analysis tools must fulfill. These are described in the following sections.

Requirements for anti-virus software to be added

There are some requirements for an anti-virus engine in order for it to be usable in the PowerScan framework. As the framework does not support GUI interaction with scanners in any way, invocations of the engine or the interpretation of the results must be doable via command line. This has some consequences, as listed below.

Requirements for adding a new anti-virus engine to be used for surface scan operations (phase 1):

1. It must be possible to initiate on-demand scan of a malware sample by executing some file with the malware sample (and potentially some other parameters) as a parameter.

2. Execution of the file that performs the scan must return the result either to a log file or stdout/stderr.

3. The log file must be parsable in some manner, so that it is possible to automatically extract a summary of the result.

Note that the above requirements does not heavily limit the range of scanners that can be used; although not all anti-virus solutions support what would be the most simple case, i.e. some "*scanner.exe -file=malwaresample.exe*" syntax, most provide some combination of parameters that allows for on-demand scan via the command line. Furthermore, it should be possible to write macros that interact with the interface of purely GUI-based scanners such that even these may be automated, although this may require considerable effort and is not covered in any further detail here.

Requirements for performing automatic update of virus definitions in anti-virus software (relevant for phase 1 and phase 2 part 1):

1. It must be possible to invoke a virus definition update by executing a file on the remote OS with a given set of parameters.
2. The update operation should preferably indicate the result of the update operation in some parsable manner, i.e. in a log file or written to stdout/stderr (which could be redirected to file).

As for surface scan, these requirements are not too hard in most cases. However, the operation will often require some degree of work during set-up. This may include operations like creating setup files[32], investigating the involved parameters and so on. Some engines do not support updating via the command line at all. In these cases it may be possible to use macros or write a some simple code/script which connects to the update server, downloads the definitions and writes them to the proper location (this would then be the file that is executed to perform the update operation).

Requirements for executing malware with real-time anti-virus software running:

1. It must be possible to configure the anti-virus software to monitor the system and output any incident (i.e. discovered infection) to a file.
2. The anti-virus software output must be written in a plain text format which can be parsed as simple strings.

These requirements enforce somewhat stronger limitations on the possible anti-virus solutions; as the monitoring process must be running in the background, it is very hard to interact with it and modify its behavior without tampering with the source code. It might be possible to process any pop-ups that appear upon executing a malware sample, but this is not investigated in any further detail here. Most anti-virus engines write information about real-time scan to log files, so the results could be parsed from these.

For an analysis of a relatively comprehensive set of anti-virus software and their behavior according to the requirements above, refer to appendix B.

Requirements for dynamic analysis tools to be added

As for the former described operations, the dynamic analysis functionality depends on interaction with one or more tools running in a guest OS in a

[32]Some tested solutions rely on setup files for updating.

virtual machine. Analysis tools that are to be plugged into this framework need to have the following properties:

1. It must be possible to invoke the analysis tool from the command line, or it must be possible to have it running as part of a snapshot.

2. The analysis tool must be able to report the result of their operation to a file or to stdout/stderr (which may be redirected to a file).

The PowerScan framework can be configured to sleep its operations for some period of time while analysis tools are being executed. Thus, it is actually possible to use analysis tools which require human interaction in PowerScan. This sleep period is configurable via the properties file. The sleep may be interrupted at any time if the operator does not require any more time by pressing the Cancel button on the progress monitor in the GUI or pressing the Enter button when using the command line. When the sleep period is over or interrupted, the result file is copied back to the local system and the tools in the next virtual machine are executed. When all the analysis tools have completed, the results are presented to the operator.

3.3.4 Security

When dealing with potentially harmful code[33], it is very important that the system as a whole is protected against the code taking control or causing permanent harm. In PowerScan, the decidedly greatest risk is the execution of the malware sample in the virtual machine, both for the purpose of analyzing it using dynamic analysis tools and for analyzing it using a real-time anti-virus engine. Although virtualization greatly reduces the risk of harmful code performing any real damage, there are a few things that can go wrong:

1. The malware can break out of the virtualized environment and attack the host OS.

2. The malware can break out of the virtualized environment using the network connection.

[33]Indeed, the code is not only potentially harmful - it is *probably* harmful!

Against risk 1, one will have to rely on the implementor of the virtualized environment to prevent break-outs. In the case of VMware, this generally should not be a big problem, as the software is very wide-spread and subject to relatively close scrutiny from both the user community and the security community in general. There has been a few incidents in the past[34], but hardly any of these apply to the usage described in this thesis. Should malware targeting a flaw in VMware software be released, the weakness is likely to be detected sooner rather than later, as VMware is also the weapon of choice for many malware analysts. Keeping an eye for security advisories and keeping VMware Server up to date should in general be sufficient to mitigate this threat.

Risk 2 can easily be mitigated through disabling the network interface for the guest OSs. A problem is that this process is not supported by the VIX API, meaning that the user will have to manually enable the network interfaces prior to performing update of virus definitions and then disable it again before executing malware samples. Needless to say, forgetting this can have dire consequences, as it may enable malware to spread on the local network and even initiate attacks on other systems/networks.

In this implementation not too much emphasis has been put on protecting the client itself, as it is supposed to be used in closed environments without being exposed as an external service. Still, the client's security is enhanced through the usage of the inherently secure Java language. Additionally, the interface to the client is relatively limited, and external users have few opportunities to manipulating execution without being able to modify the configuration or properties files.

3.3.5 Performance

This sections deals with performance related aspects of the implementation. In general, the usage of virtual machines generally makes the system too slow for anything but checking of critical files and usage related to malware analysis. This is due to the fact that operations on the virtual machines are done on an OS scale, meaning that they involve the entire dataset (disk operations, storing of memory state and so on) of the OS. Examples of very time consuming operations are powering on and off the virtual machines,

[34]For a list of issued VMware security advisories, see `http://www.vmware.com/security/advisories/.`, for security alerts, see `http://www.vmware.com/security/alerts/`

taking snapshots and reverting from snapshots.

Scan

As the scan operation may be performed relatively often, all scans are performed in parallel. This leads to significant performance improvement compared to a simple sequential scan. As the virtual machines themselves are parallel (and may even be executing on different physical serves) there is no reason that the scanning should be performed in sequence. This goes for both on-demand scan and execution of malware samples with real-time anti-virus software running.

File copying

Copying files to virtual machines using VIX is a relatively slow operation. For example, copying a file of 21ă751ă685 bytes to two virtual machines running in parallel on the same host took 590 seconds, yielding a transfer speed of approximately 38kB/s. This is not a very impressive feat given that the actual bandwidth bottleneck is the 54Mbit WLAN interface of the client. In most cases, however, this will not be a severely limiting factor, as malware tend to be relatively small in size in order to be able to spread unnoticed.

3.3.6 Known weaknesses

During development and testing, some issues arised that may somewhat limit the behavior and functionality of the framework. Some of these issues may be possible to mitigate, but during the work on this thesis the time was not sufficient to rectify all. This sections lists some of the known issues which has not been rectified, some of which may be subject to further work.

Some threads may hang if a global timeout occurs. This comes from the fact that the global timeout triggers enforced restoration to a previously taken snapshot (to avoid inconsistencies in the scan machines), even though other operations, such as file copying or file execution, are still processing. The reason that the restoration of snapshots is done in such a “hard” manner is that some operations may crash in a blocking manner without timing out, depending on more or less unpredictable events in the virtual machine. When

this happens, simply kill the process manually (for example by pressing ctrl-z in the shell), wait for the virtual machines to restore snapshots, edit the settings file to a higher full timeout and retry. This is especially important when processing large files as file copying is relatively slow when using VIX.

When result files use non-standard character encoding, the result will be parsed incorrectly by Java. This means that the automatic filtering will fail, and any output reported to the user will probably be garbled. A work-around for this is to get the file name of the copy of the log file on the client machine and inspect the file manually. The problem here is that file encoding is not given explicitly for files, so if this is not known in advance it can be very hard to determine (it must typically be done using stochastic methods). Different solutions to this problem have been tested, but none satisfactory working were found.

Directory names used with VIX may not contain special characters (for example the Norwegian “æ”, “ø” and “å”).

The PowerScan system must be restarted between different operations when using the optional GUI. The is probably due to state created in VIX when reverting the virtual machines to snapshots. This issue has been attempted resolved by closing all the connections to the VMware Server manually, but it has not been successful.

The framework does not consider scan engine version. For some anti-virus solutions this may lead to problems, as the newest signature may sometimes require updates of the scan engine as well. Automatization of this process would be challenging, both practically related to implementation but also with respect to licensing for commercial products, where obtaining updates and so on may not be free. As updates of the scan engine is something that only needs being done at some interval (say, once a month), it can be done manually without too much hassle.

During testing, there appeared to be some issues with the layout of the GUI, resulting in that some of the input field and the console text area appeared stretch outside the boundaries of the GUI window. However, this issue did only show up on one of five computers when testing. Since this issue has not been possible to reproduce on the other computers used during development and testing, it is not suspected to cause any trouble in the future.

Chapter 4

Conclusion

"Have no fear of perfection, you'll never reach it"

- Salvador Dali

4.1 A look back

Recall the different phases of the implementation in this thesis from section 3.1.1:

Phase 1 Create functionality for automatically scanning a file with several anti-virus engines. The result should be aggregated and presented to the user.

Phase 2, part 1 Execute the malware sample while running real-time anti-virus software. Aggregate and present the result.

Phase 2, part 2 Execute the malware sample and perform a dynamic analysis of its actions on the environment in which it is run.

As has been shown throughout chapter 3, all of these phases have been fully implemented. By utilizing VMware's *VMware Server*, a distributed environment of virtual machines is set up for hosting anti-virus software and analysis tools, which can be used for analyzing malware samples. In this manner, a sample can be scanned using on-demand surface scanners, executed while running real-time anti-virus software and analyzed using dynamic analysis tools. For all of these cases, any number of anti-virus scanners and analysis tools may be used, greatly enhancing the probability of exact identification and classification.

4.2 Strengths

The PowerScan framework as implemented here has a number of strong sides. Nearly all the requirements listed in section 3.1.1 were met. This means that all the intended and desired functionality, and more, of the framework is present in the final implementation. This includes:

High extensibility Usage of XML configuration allows high extensibility when adding new servers, scanners and tools to the PowerScan framework. This also includes support for any tool that can be used via the command line, meaning that both present and future tools can be plugged into the framework. The XML configuration can be modified directly by accessing the file, or by using the GUI config editor offered as part of the graphical user interface.

High customizability All relevant constants in PowerScan can effortlessly be modified by editing the supplied properties file.

Wide interface support PowerScan comes with both graphical user interface and command line interface support. Additionally, the system is written in a way that allows usage of the services as a library by calling the relevant functions in the PowerScan class.

Dependability PowerScan is written in a manner which makes sure that virtual machines are always restored to a previously taken snapshot after performing potentially harmful operations, such as malware copying and execution. As long as PowerScan is allowed to execute to an end, meaning that it is not forcefully terminated, the virtual machines will always be reverted to snapshot if a malware operation has been performed.

Nifty extra features Additionally, PowerScan has a number of small convenient features which eases use, such as:

- User interaction during dynamic analysis tool execution. PowerScan sleeps for a predefined amount of time for every virtual machine used for analysis, allowing the analyst to interact both with the malware itself and the analysis tools during execution. This interaction period can also be skipped by pressing the cancel button in the GUI or pressing enter in the command line interface.

- PowerScan is distributed, meaning that it supports multiple VMware Server machines. This means that a server farm of several less powerful machines can be used for analysis.
- PowerScan performs logging in all central classes, meaning that execution and results can be inspected closely in case of unexpected behavior and/or results.
- PowerScan performs validation of the XML configuration file, in order to verify that it conforms to the specified schema.

4.3 Weaknesses

Despite the strong sides listed in the previous section, there are some weaknesses with the PowerScan implementation:

All used tools must support command line usage As PowerScan uses VMware Vix for communication with the virtual machines, all tools that are used must support interaction through the command line usage, as Vix does not provide any means for interacting with GUIs of running programs. This limitation is also present in similar existing commercial tools, such as VirusTotal described in the related work section of this report[1]. It might be possible to write custom scripts for interaction with the GUIs of individual tools, but this has not been investigated any further.

Performance PowerScan requires every anti-virus scanner to run on a separate virtual machine and requires all virtual machines used to revert to snapshot after operation. Naturally, this introduces a significant processing overhead.

Additionally, some more specific known weaknesses of the implementation are discussed in section 3.3.6.

[1] Stated in `http://www.virustotal.com/sobre.html` under credits.

4.4 Possible usages

Considering the weaknesses listed, and particularly the performance part, there are some limitations to the usage of PowerScan. As operations will usually take a significant period of time, PowerScan is not well suited for use in real-time systems such as personal computers or mail servers. However, in systems where accurate identification and classification is essential, such as for example in malware analysis (which indeed is the intended usage of PowerScan) or at the border of highly sensitive networks or systems PowerScan comes to its right. In such situations, accurate identification and classification will usually be more important than real-time response.

Another interesting usage for PowerScan is comparison of anti-virus engines. To date, there are relatively few good, objective comparisons of anti-virus engines available. By using PowerScan as a library, it would be possible to automate the process of comparing detection rates of several engines for malware samples, both 0-day and older.

4.5 Lessons learned

During development of PowerScan and writing of this report, some lessons were learned. One such is that it is desirable to have close collaboration and contact with the future users of the product. This goes for defining concise requirements, holding regular demonstrations and receiving feedback based on these and performing acceptance testing in the end. Due to time constraints on both sides as well as geographical distance, this was not done to as large a degree as desired during this implementation.

In general, one might also say that the approach to theory/background in this thesis (and in particular this report) is sub-optimal. Because of the limited timeframe, theoretical aspects were investigated in parallel with implementation, meaning that not all discovered aspects were implemented. Also, some of the pure background theory may seem less relevant, as it only provides understanding of the involved concepts, without actually being used in the implementation.

On a more practical note, it can be argued in hindsight that other virtualization/emulation environments than VMware should have been considered before starting the implementation. As discussed several places in chapter 2, there are some weaknesses with using VMware in a malware analysis context. However, with the knowledge gained throughout the work, it is still hard to see that any other environment could have provided a better overall package with respect to security, programming interface, distributed capabilities and so on.

As with most projects, there are some aspects that the authors feel should have been done that were not due to lack of time. This especially goes for doing a feasibility analysis of execution with real-time anti-virus scanners running in the background as compared to on-demand scanning, which is the established way of doing anti-virus scan today (see for example MetaScan and VirusTotal in the related work section). Doing a comprehensive study of detection rates of fresh samples using this technique compared to on-demand scanning might have yielded some interesting results. For some suggestions for further work, see the following section.

4.6 Further work

The PowerScan framework is a prototype. This means that most of the emphasis in this thesis has been on implementing the actual program, making it work and documenting the theory behind it. There are quite a few aspects of it that can still be investigated.

From an academic point of view, it would be interesting to assess how efficient the malware execution with real-time anti-virus engine solution is compared to a straight forward scan for fresh (0-day) malware samples. Using the PowerScan framework it should be relatively straight forward to scan a significant number of samples using several anti-virus engines in both real-time and on-demand mode and compare the results.

Another interesting possibility for the future would be to try to integrate solutions for some of the weaknesses of the analysis techniques used. In the background section of this thesis, some shortcomings of PowerScan's setup are discussed and although mitigations techniques are described, none of them have been implemented. An interesting task would be to try to implement an automated solution to these challenges. Examples of these challenges are malware that detects virtualized environments, discussed in section 2.3 and multiple execution path analysis discussed in section 2.8.

Other possibilities include setting up the system for usage in a real environment such as a malware analysis lab or at the perimeter of a high-security system. This would not require a lot of research work, but some effort would still be required. Setup of the system is described in the user guide in appendix A.

Finally, a solution to the problem of PowerScan only supporting command line tools could be created. This limitation of PowerScan reduces the number of tools that can be plugged in, and decreases its overall value as a framework. It should be possible to write relatively simple code for interacting with relevant GUIs, but this has not been investigated in any detail in this thesis.

Bibliography

[1] P. Szor, *The Art of Computer Virus Research and Defense.* Addison Wesly for Symantec Press, 20035.

[2] M. Erbschloe, *Trojans, Worms, and Spyware: A Computer Security Professional's Guide to Malicious Code.* Newton, MA, USA: Butterworth-Heinemann, 2004.

[3] R. A. Grimes, *Malicious mobile code: virus protection for Windows.* Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2001.

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

[5] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," *Security & Privacy Magazine, IEEE*, vol. 5, no. 2, pp. 32–39, March-April 2007.

[6] U. Bayer, C. Krugel, and E. Kirda, "TTAnalyze: A tool for analyzing malware," 2006. [Online]. Available: http://citeseer.ist.psu.edu/bayer06ttanalyze.html

[7] S. Roman, "Microsoft technet windows architecture," World Wide Web electronic publication, Microsoft, Tech. Rep., 2000. [Online]. Available: http://www.microsoft.com/technet/archive/ntwrkstn/evaluate/featfunc/winarch.mspx

[8] J. Bacon and T. Harris, *Operating Systems : Concurrent and distributed software design.* Pearson Education Limited, 2003.

[9] Microsoft Corporation, "Microsoft portable executable and common object file format specification, revision 8.1," World Wide Web electronic publication, 2008. [Online]. Available: http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

[10] A. Singh, "An introduction to virtualization." World Wide Web electronic publication, 2004. [Online]. Available: http://www.kernelthread.com/publications/virtualization/

[11] IBM, "IBM Systems Virtualization," 2005. [Online]. Available: http://publib.boulder.ibm.com/infocenter/eserver/v1r2/topic/eicay/eicay.pdf

[12] M. T. Jones, "Virtual linux; an overview of virtualization methods, architectures, and implementations." World Wide Web electronic publication, 2006. [Online]. Available: http://www.ibm.com/developerworks/linux/library/l-linuxvirt/

[13] "Understanding full virtualization, paravirtualization and hardware assist," World Wide Web electronic publication, 2007. [Online]. Available: http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf

[14] A. Zeichick, "Processor-based virtualization, amd64 style, part ii," World Wide Web electronic publication, AMD, Tech. Rep., 2006. [Online]. Available: http://developer.amd.com/TechnicalArticles/Articles/Pages/630200615.aspx

[15] I. Cooperation, "Enhanced virtualization on intelő architecture-based servers," World Wide Web electronic publication, Intel Cooperation, Tech. Rep., 2006. [Online]. Available: http://www.intel.com/business/bss/products/server/virtualization_wp.pdf

[16] J. Lo, "VMware and CPU Virtualization Technology," World Wide Web electronic publication, 2005. [Online]. Available: http://download3.vmware.com/vmworld/2005/pac346.pdf

[17] VMware Inc., "Virtualization: Architectural considerations and other evaluation criteria," World Wide Web electronic publication, 2005. [Online]. Available: http://www.vmware.com/pdf/virtualization_considerations.pdf

[18] M. Carpenter, T. Liston, and E. Skoudis, "Hiding virtualization from attackers and malware," *Security & Privacy Magazine, IEEE*, vol. 5, no. 3, pp. 62–65, May-June 2007.

[19] P. Ferrie, "Attacks on virtual machine emulators," World Wide Web electronic publication, Symantec Advanced Threar Research, Tech. Rep., 2006. [Online]. Available: http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf

[20] J. S. Robin and C. E. Irvine, "Analysis of the intel pentium's ability to support a secure virtual machine monitor," in *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2000, pp. 10–10.

[21] M. Schmall, "Heuristic techniques in av solutions: An overview," World Wide Web electronic publication, Tech. Rep., 2002. [Online]. Available: http://securityfocus.com/infocus/1542

[22] V. Bontchev, "Current Status of the CARO Malware Naming Scheme." [Online]. Available: http://www.people.frisk-software.com/~bontchev/papers/pdfs/caroname.pdf

[23] Symantec, "Symantec Global Internet Security Threat Report Volume XII," World Wide Web electronic publication, 2008. [Online]. Available: http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xiii_04-2008.en-us.pdf

[24] V. Bontchev, F. Skulason, and A. Solomon, "A Virus Naming Convention," World Wide Web electronic publication, 1991. [Online]. Available: http://www.caro.org

[25] N. FitzGerald, "A Virus by Any Other Name: The Revised CARO Naming Convention," in *Proceedings of the 5th Anti-Virus Asia Researchers conference 2002*, Seoul, 2002, pp. 141–166.

[26] M. Gheorghescu, "An automated virus classification system," in *Proceedings of the fifth annual Virus Bulletin conference 2005*, 2005.

[27] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, "Automated classification and analysis of internet malware," *LNCS*, vol. 4637, pp. 178–197, 2007.

[28] D. Farmer and W. Venema, *Forensic Discovery*. Addison-Wesley Professional, 2005.

[29] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," *2007 IEEE Symposium on Security and Privacy*, vol. 00, pp. 231–245, 2007.

[30] J. Stewart, "Behavioural malware analysis using Sandnets," *Computer Fraud & Security*, vol. 2006, no. 12, pp. 4–6, 2006.

[31] VMware, "Vix API Reference Documentation." [Online]. Available: http://www.vmware.com/support/developer/vix-api/

Appendix A

User Guide

> "The question of whether a computer can think is no more interesting than the question of whether a submarine can swim."

- E. W. Dijkstra

This user guide describes how to set up and use the PowerScan analysis framework and is intended to be as detailed as necessary to enable a user to utilize the capabilities of the framework without knowledge of the source code. The XML configuration and properties files are described in detail.

A.1 Requirements

To be able to use the PowerScan framework, there are some requirements that will have to be fulfilled. The following are the system requirements for the different components.

A.1.1 Client

The client is the computer that is to be used to run the PowerScan jar-file, upload the malware samples and read the results. The following are the requirements to the client:

- Java Runtime Environment: Minimum JRE 6.
- Operating system: Only Microsoft Windows XP has been tested, although it will probably work on all Microsoft Windows platforms. Theoretically, PowerScan should work on any platform as it is written in Java, but this has not been tested.

A.1.2 Virtualization servers

The virtualization servers are the servers hosting the virtual machines used by the PowerScan framework. The following are the requirements to the virtualization servers:

- Operating system: The system has been tested with Ubuntu Server Edition 7.10 with Linux kernel 2.6.22-14-server as host operating system, although any OS supported by VMware Server should work.
- Virtualization software: The system requires VMware Server 1.x or any other VMware Server version supported by VMware Vix. Has been tested with VMware Server 1.0.6 for Linux.

A.1.3 Usage

The following requirements apply for the usage of the PowerScan framework:

- When the jar file is to be executed, it should be executed with the directory containing the jar file as the working directory. This ensures that relative references to configuration files etc. will work. The shortcuts in supplied with PowerScan ensures the correct working directory on Windows platforms.

A.2 Environment Setup

The PowerScan malware analysis framework is made up of an analysis environment, an XML-formatted configuration file and a text formatted properties file in addition to the PowerScan jar-file running from a client workstation.

The malware sample scanning environment is made up of one or more hosts running VMware Server. By default VMware Server listens on port 902, but it is possible to use customized port numbers. Once the server(s) have been set up, virtual machines need to be created and a Microsoft Windows OS installed. PowerScan requires that VMware Tools are installed on each of the virtual machines, which can be done via the VMware Server interface. The system supports each virtual machine (and thus Windows installation) running one anti-virus engine and/or one or more dynamic analysis tools. To avoid the tools interfering with each other, it is recommended to use only one anti-virus engine or a few tools on each virtual machine. After the anti-virus engine and/or analysis tools have been installed, the corresponding paths and parameters must be written into the XML config file.

The following is a step-by-step installation instruction:

1. Install VMware Server on hosts running x86 architecture OSs, such as Microsoft Windows or Linux/Unix. The test setup during the implementation in this thesis used Ubuntu Server, which provided very good performance and stability. Host system requirements can be found in section “Host System Requirements” of chapter 1 of the VMware Server

Online Library[1].

2. A serial number is needed when installing VMware Server, this is issued by VMware when registering for download at *Download VMware Server - http://www.vmware.com/download/server*. Note that there are different serial numbers for Windows and Unix/Linux systems. Help on installation may be found in chapter 2 of "Administration Guide" on the VMware Server Online Library[1].

3. Install VMware Server Console on the computer that should function as the client. VMware Server Console is part of the "VMware Server Windows client package"[2]. Requirements for the client is also found in the "Administration Guide" on the VMware Server Online Library[1].

4. Create as many virtual machines as desired on each host. Instructions are given in chapter 2 of the "Virtual Machine Guide" on VMware Online Library[1].

5. Install Microsoft Windows XP[3] as guest OS on the virtual machines. Help on this operation may be found in the "Choosing and Installing Guest Operating Systems" section of the "Guest Operating System Installation Guide" on the VMware Online Library[1]. This guide also lists known issues and compatibility between VMware products and various guest OSs.

6. Create a user with administrator access on each guest OS. Note that the users must have a password set in order to work with PowerScan.

7. Install the desired anti-virus engines and analysis tools on the designated virtual machines. This installation process is vendor specific, and is not described here.

8. Take a snapshot of each VM using the VMware Server Console, as shown in figure A.6. The snapshot should, for performance reasons, be taken when the guest OS is running, the user logged[4] in and no

[1]VMware Server 1 online library - `http://pubs.vmware.com/server1/wwhelp/wwhimpl/js/html/wwhelp.htm`.

[2]VMware Server registration - `http://register.vmware.com/content/download.html`.

[3]Or other supported OS. Although only tested with Win XP, PowerScan should work with any Win NT OS. 64 bits guest OSs are also supported.

[4]If not, the virtual machine must be powered on and the user logged in every time PowerScan is run.

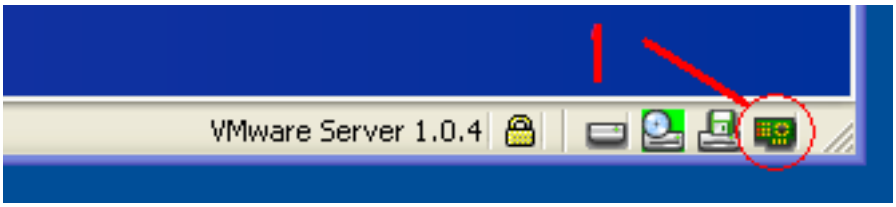


Figure A.1: VMware Server Console status line.

windows open[5].

9. Once the virtual machines are set up, an appropriate config file and properties file must be prepared. An example XML file is shown in section D.1. For an explanation of the XML, see section A.11.

Note that the VMware environment need to be secured before uploading any suspected malware samples. In particular, the network feature of the virtual machine must be set up properly. Typically, before a malware sample is executed, it should be ensured that the virtual machine does not have access to the Internet or any other potentially unsecured networks. The network options, shown in figure A.2, are available by double clicking the "network card" icon on the VMware Server Console status bar, as shown in figure A.1. Before running a malware sample, the network should be set to "*Host-only*", so that the malware is unable to communicate with the Internet.

A.3 User Interface

The PowerScan framework currently supports two interfaces; a graphical user interface (GUI), a command line interface (CLI) and an application programming interface (API)[6]. The command line interface is invoked by default when launching the application. When using the CLI, all output from the program is written to stdout, meaning that it will usually be directed to the command line console. When using GUI, all output will appear on the console area of the GUI window.

[5]As open windows may cause real-time scanners to interfere with the on-demand scan operation.

[6]The API can be called by using the PowerScan as a library and calling functions in the `edu.ntnu.item.jt.system.PowerScan` class.

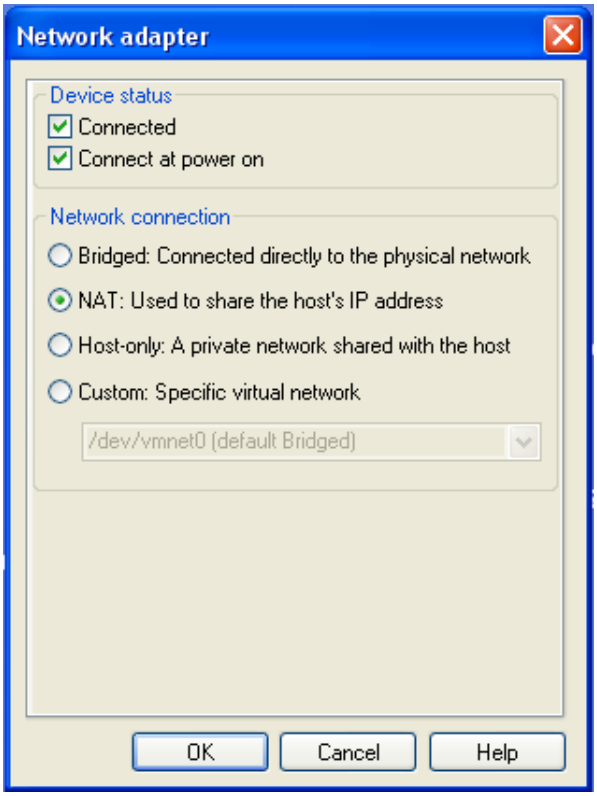


Figure A.2: VMware Server network options.

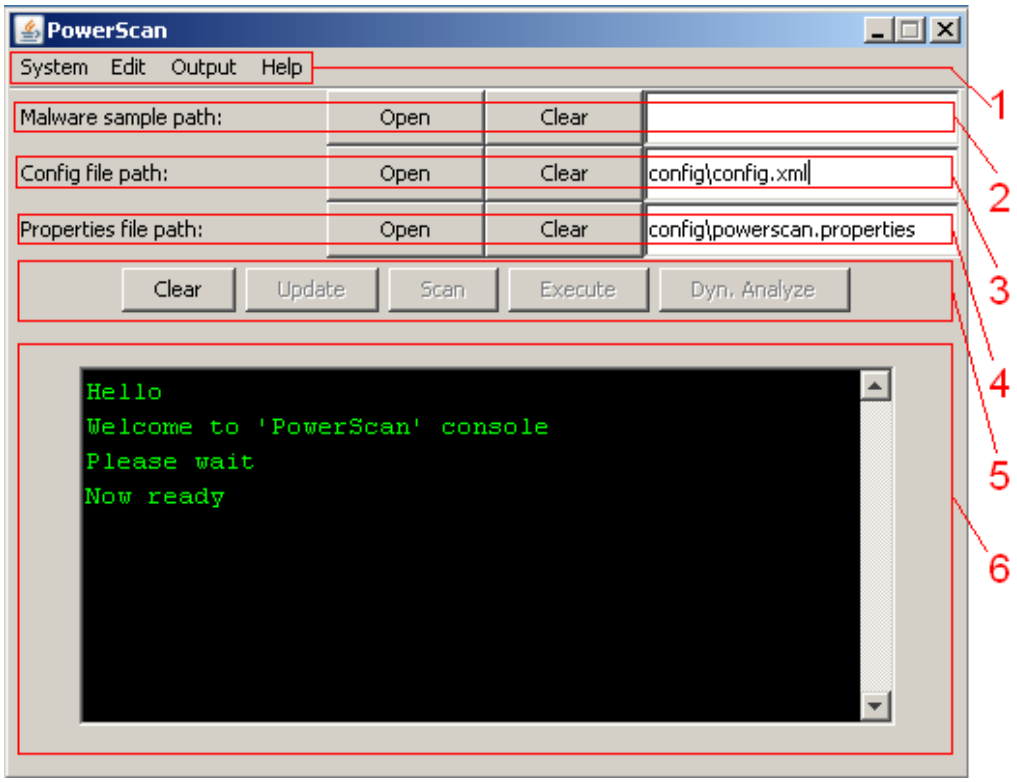


Figure A.3: The GUI main window.

A.3.1 Graphical user interface

To use GUI, the application must be started with the "-GUI" command line switch[7] or by using the included Windows shortcut. The various areas of the main GUI window are described below:

1. The menu line gives access to various options as shown in figures A.4, A.5, A.7 and A.8. The various menu options are described later in this document.

2. The *Malware sample path* area contains a field for manual path entry, a "Clear" button to clear the field and an "Open" button that allows the user to browse the client file system. This field must contain the path to the malware sample file to be analyzed (unless the operation to be performed is update).

[7]Meaning that PowerScan must be launched as `java -jar powerscan.jar -GUI`.

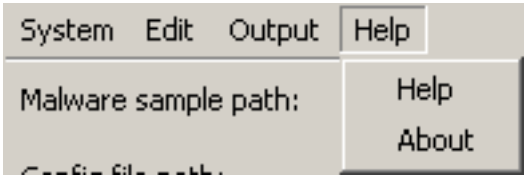


Figure A.4: The GUI "Help" menu.

3. The *Config file path* area contains a field for manual path entry, a "Clear" button to clear the field and an "Open" button that allows the user to browse the client file system. This field must contain the path to the XML config file described in section A.10 of this document.

4. The file properties path area contains a field for manual path entry, a "Clear" button to clear the field and an "Open" button that allows the user to browse the client file system. This field should contain the path to the properties file described in section A.13 of this document.

5. The buttons in this area are used to invoke the various operations of the PowerScan system.

6. The console text area is where the progress messages and result outputs are printed.

A.3.2 Command line usage

The following is the text shown when using the CLI:

```
usage: java -jar powerscan.jar [arguments].
 -ANALYZE <arg>          Perform dynamic analysis of the supplied sample
                         with all (if any) registered dynamic analysis tools
                             from XML.
 -c,--CONFIG <arg>       XML Config file location. Default location:
                         config\config.xml
 -EXECUTE <arg>          Executes the supplied malware sample on any virtual
                         machine supporting real-time anti-virus detection and
                              extracts results.
 -GUI                    Determines whether to use a GUI.
 -s,--SETTINGS <arg>     Properties file location. Default location:
                         config\powerscan.properties
 -SCAN <arg>             Performs a surface scan of the supplied malware
                         sample with all scan engines configured in the XML
                             config file.
 -UPDATE                 Performs a update of all the AV engines listed in
                         the XML configuration file. If update can not be done
                              for a given scanner,
```

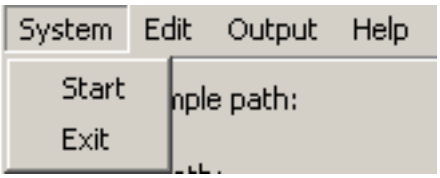


Figure A.5: The GUI "System" menu.

```
                    the user will be instructed on how to perform it.
```

Listing A.1: The CLI help text

This shows that all operations are started by running the application with an appropriate switch. The location of the XML config file and properties file should also be given to the application using an argument. If not, the default locations are assumed.

A.4 Malware sample scan

Performing a malware sample scan in PowerScan means scanning the suspected malware sample with the on-demand scan feature of the installed anti-virus scanners.

Starting a malware sample scan using GUI is performed using the following steps:

1. Input the paths to the config and properties files in fields 3 and 4 shown in figure A.3.

2. Input the path to the malware sample on the local system in field 2 shown in A.3.

3. To read the configuration and initialize the system, choose "Start" from the "System" menu choice, shown in figure A.5.

4. Once "Now ready" appears in the text area, press the "Scan" button.

5. When the operation finishes, the result will appear in the text area.

Starting a malware sample scan using the CLI is done using the following command:

```
> java -jar powerscan.jar -c <configFilePath> -s <propertiesFilePath>
    -SCAN <malwareSamplePath>
```

Listing A.2: Invocation of the PowerScan scan operation using the Command Line Interface.

A.5 Malware sample execution

Some malware samples may use packers and avoid detection by the on-demand surface scan, but might be detected when executed under surveillance of a real-time anti-virus scanner. Executing a malware sample in the test environment requires the following steps:

1. Input the paths to the config and properties files in fields 3 and 4 shown in figure A.3.

2. Input the path to the malware sample on the local system in field 2 shown in figure A.3.

3. To read the configuration and initialize, choose “Start” from the “System” menu choice in field 1, shown in figure A.5.

4. Once “Now ready” appears in the text field, press the “Execute” button.

5. When the operation finishes, the result will appear in the text area.

To start execution of a malware sample using the command line, use the following command:

```
> java -jar powerscan.jar -c <configFilePath> -s <propertiesFilePath>
    -EXECUTE <malwareSamplePath>
```

Listing A.3: Invocation of the PowerScan execute operation using the Command Line Interface.

A.6 Malware sample analysis

When dynamic analysis tools are installed as a part of the testing environment, they can be used to analyze a malware sample. To execute the installed analysis tools on the sample, requires the following steps:

1. Input the paths to the config and properties files in fields 3 and 4 shown in figure A.3.
2. Input the path to the malware sample on the local system in field 2 shown in figure A.3.
3. To read the configuration and initialize, choose "Start" from the "System" menu choice in field 1, shown in figure A.5.
4. Once "Now ready" appears in the text field, click the "Dyn. analyze" button.
5. When the operation finishes, the result will appear in the text area.

One important difference between the scan and execute operation on the one hand, and the analysis operation on the other, is that while the scan and execute operations are executed on all the virtual machines in parallel, the analysis operation is carried out on one virtual machine at the time. This means that first, all the tools registered on one virtual machine is started and the malware sample optionally executed, before the system sleeps for the indicated amount of time (given in the properties file). The result files (if any) are copied back, and the virtual machine is reverted to snapshot. While the first virtual machine is reverted to snapshot, the tools registered with the next virtual machine are started, and a new sleep period is started. This is done to allow an operator time to interact with the analysis tool and/or malware sample at one virtual machine at the time. The interaction sleep period is set in the properties file, but the sleep can be skipped at any time by pressing the "Cancel" button in the GUI process monitor or pressing the *Enter* key when using the command line.

When installing analysis tools, it is worth noting that some tools take the name of the executable to be monitored as a parameter, while others need to be started before the malware sample is executed on the system. In the configuration file, there is an element associated with the analysis tool that indicates whether the malware sample should be executed. For tools

taking the path to the sample as part of their parameter, the placeholder "$samplePath" should be used in the parameter string to indicate insertion of the malware sample path.

To start execution of a malware sample using the command line, use the following command:

```
> java -jar powerscan.jar -c <configFilePath> -s <propertiesFilePath>
    -ANALYZE <malwareSamplePath>
```

Listing A.4: Invocation of the PowerScan analyze operation using the Command Line Interface.

A.7 Update AV definition files

The definition files of the anti-virus engines needs to be updated relatively frequently. This operation includes taking snapshot of the system after definition files have been updated[8] To update the virus signature files, the following steps should be performed:

1. Input the paths to the config and properties files in fields 3 and 4 shown in figure A.3.

2. To read the configuration and initialize, choose "Start" from the "System" menu choice in field 1, shown in figure A.5.

3. Once "Now ready" appears in the text field, press the "Update" button.

4. When the operation finishes, the result will appear in the text area.

To initiate update of the anti-virus engines using the command line, use the following command:

```
> java -jar powerscan.jar -c <configFilePath> -s <propertiesFilePath>
    -UPDATE
```

Listing A.5: Invocation of the PowerScan update operation using the Command Line Interface.

[8]This is done automatically unless otherwise is stated.

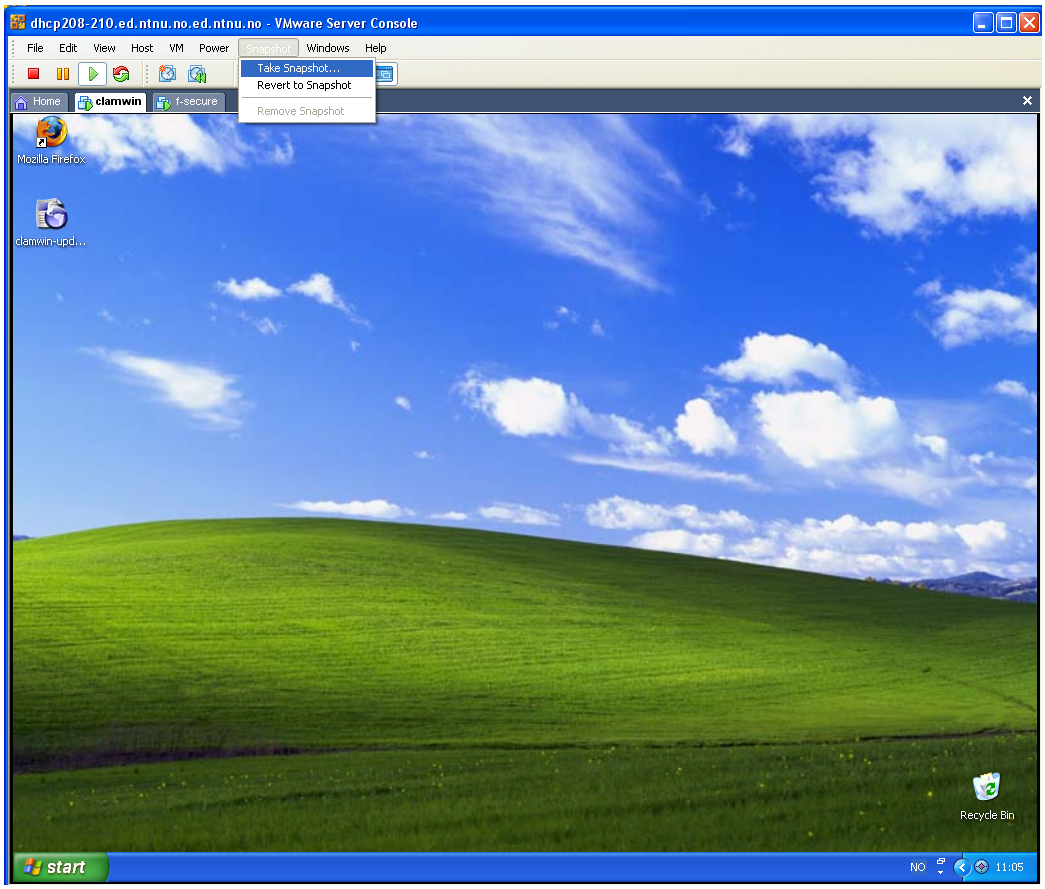


Figure A.6: Taking snapshot using VMware Server Console.

A.8 Adding new anti-virus engines or tools

Adding new tools to the analysis environment requires installing the anti-virus engine or tool on a virtual machine, and taking a snapshot after the installation is complete. The various configuration parameters must then be added to the XML configuration file to enable the framework to utilize the tool/scan engine. The installation in the guest OS should follow an ordinary Windows installation process, and is not described in any further detail here. Taking snapshot using the VMware Server Console is shown in figure A.6.

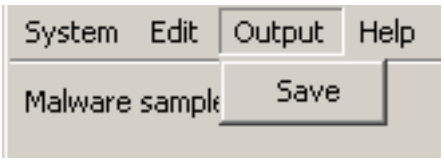


Figure A.7: The GUI "Output" menu.

A.9 Saving console output

When using the GUI, it is possible to save the text output that is currently in the console text area. This is done by choosing "Output" from the menu and then "Save", shown in figure A.7. This option saves the text currently displayed in the text area to a file.

The similar operation when using the Command Line Interface would be redirecting the output to file by appending "`$>$ output.log`" to the given command so that for example the scan operation becomes:

```
> java -jar powerscan.jar -c <configFilePath> -s <propertiesFilePath>
    -UPDATE > output.log
```

Listing A.6: Redirection of the update operation out using CLI.

A.10 Editing the XML configuration file

As the configuration file is written in XML, it is human-readable and could be edited using any text editor as described in sectionA.11. Another, hopefully more user friendly, means to edit the configuration is to use the built-in graphical config file editor. The config editor is launched from the "Edit" menu, and opens in a separate window. The main window is shown in figure A.9. This figure shows the config editor in the "Host/VM" view.

The menus shown in field 1 of figure A.9 are described in the following sections.

A.10.1 The "File" menu

The "File" menu offers the following choices, as shown in figure figure A.10:

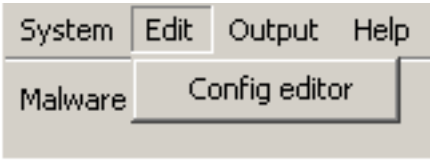


Figure A.8: The GUI "Edit" menu.

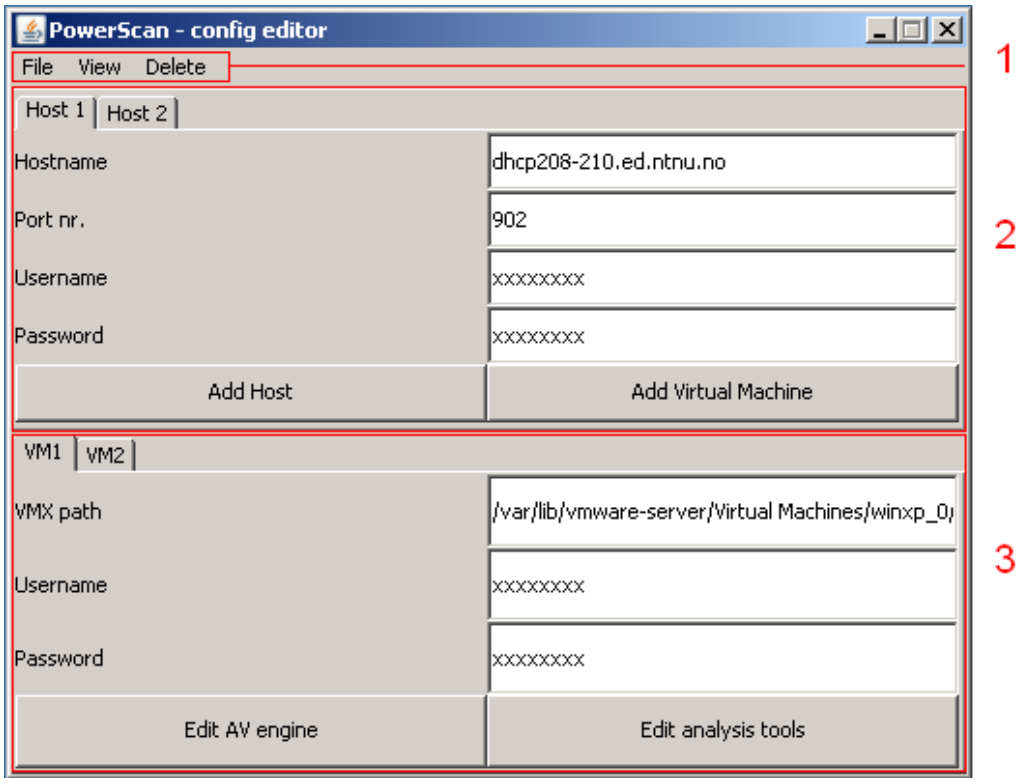


Figure A.9: The config editor main window in "Host/VM view".

New config opens a new, empty configuration file.

Open config... launches a dialog box to select an XML file to edit.

Save config... launches a dialog box for saving the open XML file.

Exit closes the Config Editor.

A.10.2 The "View" menu

The "View" menu offers the following choices, shown in figure A.12:

Host/VM view brings the user back to the host and virtual machine overview window.

A.10.3 The "Delete" menu

The "Delete" menu offers the following choices, shown in figure A.11:

Delete active host Only available in "Host/VM" view, this choice deletes the host that is selected in the upper half of the window, field 2 in figure A.9.

Delete active VM Only available in "Host/VM" view, this choice deletes the virtual machine that is selected in the bottom half of the window, field 3 in figure A.9

Delete active AVE Only available in "AVE" view, this choice deletes the currently active anti-virus engine.

Delete active tool Only available in "Tools" view, this choice deletes the currently active tool.

A.10.4 The "Host/VM" view

The "Host/VM" view shows the defined hosts in the upper half of the window, field 2 in figure A.9, and the defined virtual machines defined for the selected

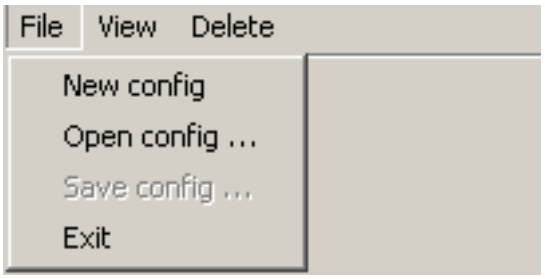


Figure A.10: The config editor “File” menu.

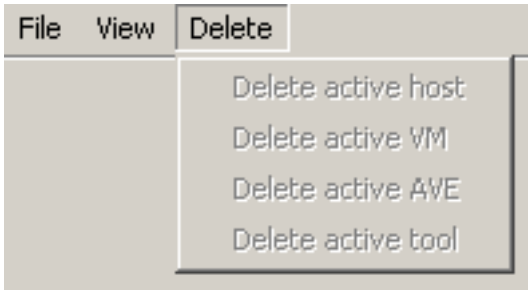


Figure A.11: The config editor “Delete” menu.

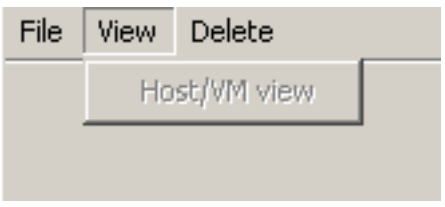


Figure A.12: The config editor “View” menu

host in the lower part, field 3. It is always possible to return to this view by choosing the "View" menu on the menu bar and selecting "Host/VM view". Field 2 also contains buttons to add a new host, or to add a new virtual machine to an already existing host. When a new host entry is created, a new, empty virtual machine entry is also created, as a VMware host offers no functionality in PowerScan unless it has a virtual machine registered.

When a host is selected in field 2, all the virtual machines that belong to that particular host are shown in the field 3. A virtual machine may run an anti-virus engine, and/or a set of analysis tools. If a host has an anti-virus engine or tools entry registered, an "Edit AV engine" or "Edit analysis tools" button is shown. If the host does not have an anti-virus engine or tool set registered, field 3 will have an "Add AV engine" and an "Add analysis tool" button.

Both the "Edit AV engine" and the "Add AV engine" button will change the editor to the "AVE" view, described in section A.10.5. The "Edit analysis tools" and "Add analysis tool" buttons change to the "Tools" view, described in section A.10.6.

A.10.5 The "AVE" view

The config editor "AVE" view is shown in figure A.13. This view allows editing of all elements related to an anti-virus engine. In addition to the element fields, the view has a "Save changes" button that saves the changes made within the view. Note that this does not write the changes to file - the only way of permanently storing the changes is via the "Save config..." File menu choice! If the "AVE" view is left using the "Host/VM" choice on the "View" menu, the changes are discarded.

An entire "AVE" element can be deleted by choosing the "Delete active AVE" command on the "Delete" menu. If no changes have been made, or the changes have been saved by clicking the "Save changes" button, it is possible to navigate back to the "Host/VM" view by selecting the "Host/VM" command from the "View" menu.

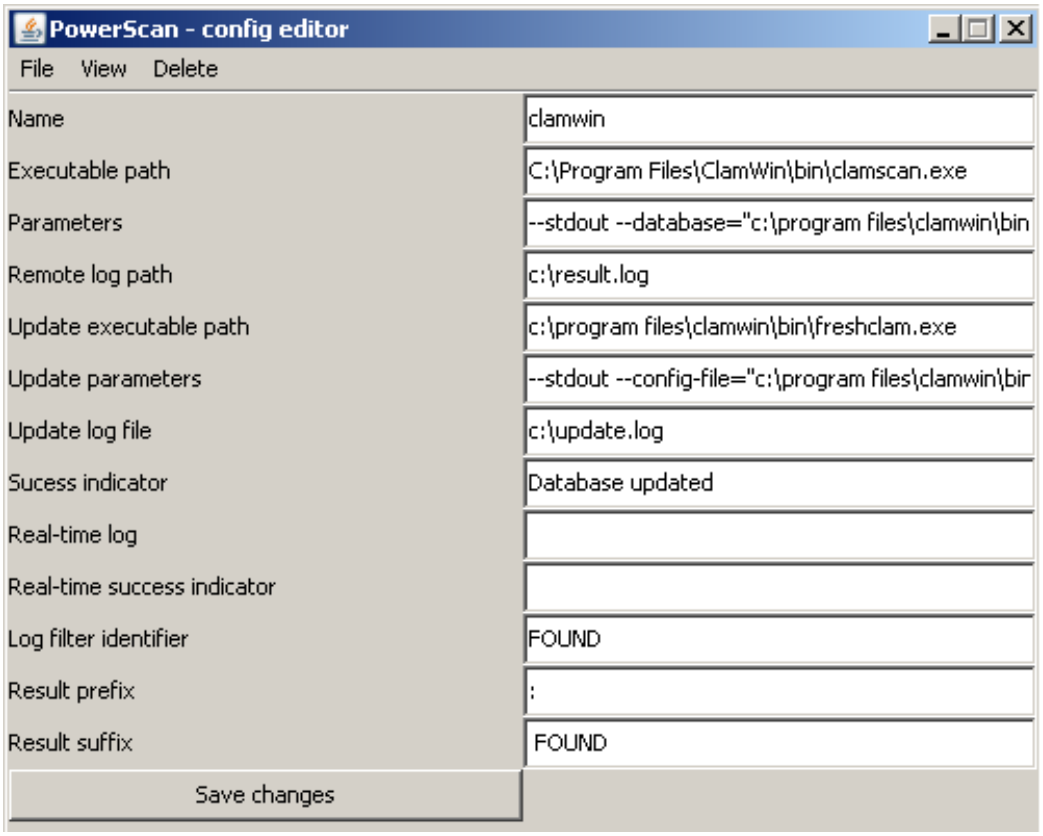


Figure A.13: The config editor “AVE” view.

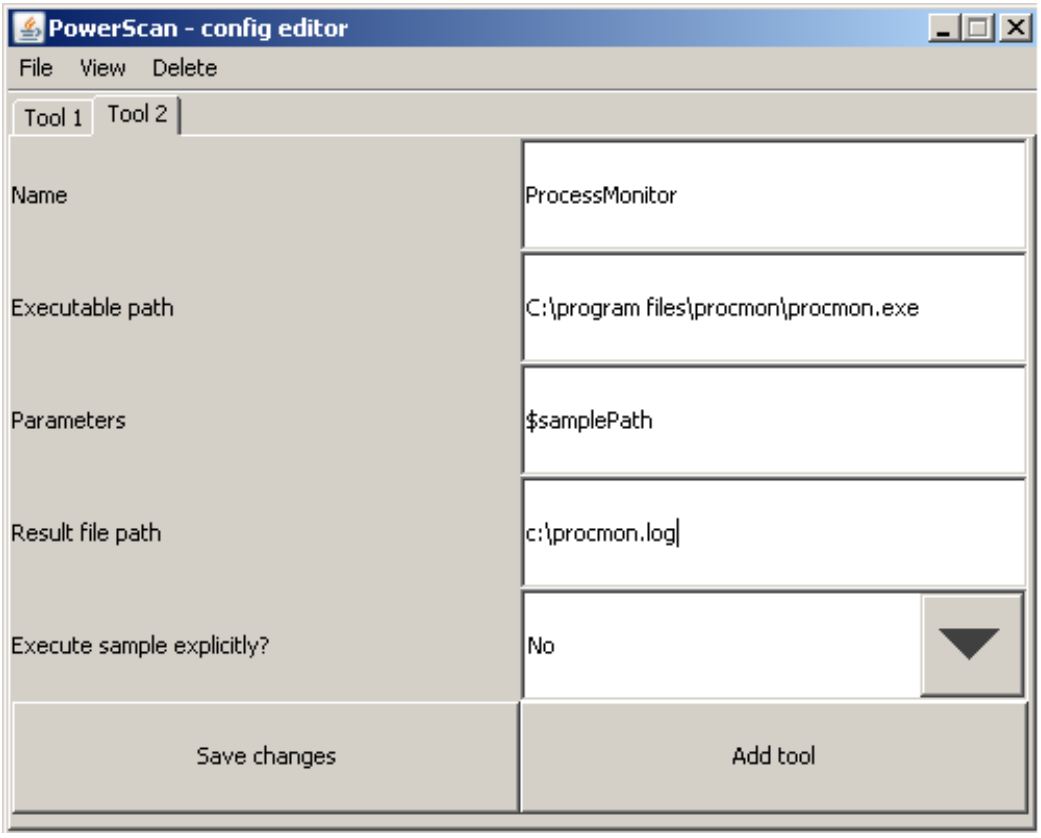


Figure A.14: The config editor “Tools” view.

A.10.6 The "Tools" view

The config editor "Tools" view is shown in figure A.14. This view allows editing of all tools associated with a virtual machine. Switching between different tools is done by selecting the appropriate tab at the top of the view window. In addition to the element fields, the view has a "Save changes" button that stores the changes in the system. Note that this does not write the changes to file - the only way of permanently storing the changes is via the "Save config..." File menu choice! If the "Tools" view is left using the "Host/VM" choice on the "View" menu, the changes are discarded. This view also has an "Add tool" button that adds a new tool element to the configuration.

The currently active tool can be deleted by choosing the "Delete active tool" command on the "Delete" menu. If no changes have been made, or the changes have been saved by clicking the "Save changes" button, it is possible to navigate back to the "Host/VM" view by selecting the "Host/VM" command from the "View" menu.

A.11 Understanding the XML configuration file

As mentioned earlier, PowerScan uses an XML structured configuration file to represent its execution environment (meaning the associated VMware hosts, virtual machines, installed scanners, tools and so on). Although the file can be manipulated in the config editor GUI, it is also possible to edit the XML directly using any text editor. This section describes the different parts that make up the XML, and how they should be interpreted.

The skeleton of the XML config file is shown below.

```
1: <PowerScan>
2:  <VMwareHostList>
3:    <VMwareHost host="">
4:      <hostPortNumber></hostPortNumber>
5:      <hostUsername></hostUsername>
6:      <hostPassword></hostPassword>
7:      <VM vmxPath="">
8:        <vmUsername></vmUsername>
9:        <vmPassword></vmPassword>
10:       <avEngine name="">
11:         <avExecutablePath></avExecutablePath>
12:         <avParameters></avParameters>
```

```
13:         <avLogFilePath></avLogFilePath>
14:         <avLogFilter>
15:           <avResultIdentifier></avResultIdentifier>
16:           <avResultPrefix></avResultPrefix>
17:           <avResultSuffix></avResultSuffix>
18:         </avLogFilter>
19:         <avUpdateInfo>
20:           <avUpdateExecutable></avUpdateExecutable>
21:           <avUpdateParameters></avUpdateParameters>
22:           <avUpdateLogPath></avUpdateLogPath>
23:           <avUpdateSuccessIndicator></avUpdateSuccessIndicator>
24:         </avUpdateInfo>
25:         <realTimeScan>
26:           <rtLogPath></rtLogPath>
27:           <rtResultIdentifier></rtResultIdentifier>
28:         </realTimeScan>
29:       </avEngine>
30:       <analysisTools>
31:         <dynamicAnalysisTool toolName="">
32:           <toolExecutablePath></toolExecutablePath>
33:           <toolParameters></toolParameters>
34:           <executeMalwareExplicitly></executeMalwareExplicitly>
35:         </dynamicAnalysisTool>
36:       </analysisTools>
37:     </VM>
38:   </VMwareHost>
39: </VMwareHostList>
40:</PowerScan>
```

Listing A.7: Skeleton of the XML config file

The <PowerScan> element is the root element of the PowerScan XML configuration and signifies that the XML indeed is a PowerScan configuration file. It contains only one element; the <VMwareHostList>, which is, as the name implies, a list containing one or more <VMwareHost> elements. The <VMwareHost> element has a mandatory attribute "host" which contains the host name or IP address of a host running VMware Server. The <VMwareHost> element contains two elements, <hostUsername> and <hostPassword>, which are required to connect to the server. Further, a <VMwareHost> element contains one or more <VM> elements representing a virtual machine installed on the given host. The <VM> element has a mandatory attribute "vmxPath", which gives the path on the server to a vmx file that contains configuration info for the VM. This path is needed to be able to boot the VM without using the GUI-based VMware Server Console. A <VM> element contains a <vmUsername> and a <vmPassword> element which contain user credentials for a user that must exist on the virtual machine guest OS. In addition to the "vmxPath", <vmUsername> and <vmPassword> elements, a <VM> contains information about what tools are installed on the virtual machine. A tool in this context can be an anti-virus engine and/or one or more analysis tools. XML description of

the installed tools of the virtual machine are then given within the <VM> element. The description of an anti-virus engine is shown below.

```
10:       <avEngine name="">
11:         <avExecutablePath></avExecutablePath>
12:         <avParameters></avParameters>
13:         <avLogFilePath></avLogFilePath>
14:         <avLogFilter>
15:           <avResultIdentifier></avResultIdentifier>
16:           <avResultPrefix></avResultPrefix>
17:           <avResultSuffix></avResultSuffix>
18:         </avLogFilter>
19:         <avUpdateInfo>
20:           <avUpdateExecutable></avUpdateExecutable>
21:           <avUpdateParameters></avUpdateParameters>
22:           <avUpdateLogPath></avUpdateLogPath>
23:           <avUpdateSuccessIndicator></avUpdateSuccessIndicator>
24:         </avUpdateInfo>
25:         <realTimeScan>
26:           <rtLogPath></rtLogPath>
27:           <rtResultIdentifier></rtResultIdentifier>
28:         </realTimeScan>
29:       </avEngine>
```

Listing A.8: The AV engine element of the XML config file

As can be seen, the <avEngine> element has a "name" attribute that is used for identification of the engine. The value of this attribute could for example be set to "F-secure" or "ClamWin". Note that the element is only used for visual identification, and can theoretically be set almost any value. Then there follows four elements used when running an on-demand scan of a sample file.

- The <avExecutablePath>element on line 11 gives the path to the executable used to initiate the on-demand scan.
- The <avParameters> element on line 12 holds any parameters or arguments that should be passed to the executable. To insert the malware sample path in the parameters, use the string "$samplePath".
- The <avLogFilePath> on line 13 gives the path to a log file that is retrieved from the VM after the scan has completed.
- The <avLogFilter> element on line 14 contains strings to look for in the retrieved log file when parsing the result. These strings are used as follows:
  - The string given in the <avResultIndentifier> element on line 15 is used to fine the correct line in the log file. If this element is empty, the entire log file is printed.

- The string given in the <avResultPrefix> element on line 16 is used to determine from which position in line the result is starting. If this element is empty, it is assumed that the result text starts at the beginning of the line. If the element is set, the text AFTER the prefix is extracted.
- The string given in the <avResultSuffix> element on line 17 is used to determine at which position in the line the result text ends. If the element is empty, it is assumed that the result text ends at the end of the line. If the element is set, the text starting with the given string is removed.

For example, the interesting line in a log file may look like:

VIRUS FOUND: eicar_test_file found in selected file(s)

A filter definition like:

```
14:         <avLogFilter>
15:           <avResultIdentifier>VIRUS FOUND:</avResultIdentifier>
16:           <avResultPrefix>: </avResultPrefix>
17:           <avResultSuffix>found</avResultSuffix>
18:         </avLogFilter>
```

Listing A.9: The AV log filter element of the XML config file

would give the result *eicar_test_file*.

The next part of the <VM> element is used to hold information needed to run virus definition database updates on the anti-virus engines. The <avUpdateInfo> element contains the following elements:

- The <avUpdateExecutable> element on line 20 should contain the path to the executable that performs the update.
- The <avUpdateParameters> element on line 21 holds any parameters required by the update executable.
- The <avUpdateLogPath> element on line 22 holds the path to the log file where the result of the update operation is written. If this tag is not present (or empty), the update operation is started and no further action is taken[9].

[9]This again means that the user will have to manually check the result of the update operation. This might be desirable for updates performed on engines without support for reporting the update result to stdout.

- The <avUpdateSuccessIndicator> element on line 23 contains a string that is searched for in the log file to determine the result of the update operation. If the success indicator is found, the operation is assumed to have succeeded, and the line(s) containing the success indicator are printed. If the <avUpdateLogPath> element is present, but no <avUpdateSuccessIndicator>, the entire log file is printed for manual analysis by the user.

The final element within the <avEngine> element is used when a malware sample is executed in the OS installed on the virtual machine to see if the sample is detected by the real-time functionality of an anti-virus engine. Following the convention of the other elements, the <realTimeScan> elements contains a log file path and a result identifier. The <rtLogPath> element holds the path to the real-time scanner log file, and the <rtResultIdentifier> element a string used to find the line containing the result in the log file.

If a VM is used to perform dynamic analysis of malware, the <VM> element needs to contain an <analysisTools> element. Although it is possible for virtual machine to have both analysis tools and an anti-virus scanner, it is generally recommended that they are installed on separate machines, as they may interfere with each other. The <analysisTools> element is a list containing information about one or more tools installed on the virtual machine. The <analysisTools> element is shown below.

```
30:       <analysisTools>
31:         <dynamicAnalysisTool toolName="">
32:           <toolExecutablePath></toolExecutablePath>
33:           <toolParameters></toolParameters>
34:           <executeMalwareExplicitly></executeMalwareExplicitly>
35:         </dynamicAnalysisTool>
36:       </analysisTools>
```

Listing A.10: The analysis tools element of the XML config file

The <dynamicAnalysisTool> element is the elements that represents one tool. As seen on line 31, the element has an attribute "toolName" that is used for identification[10]. The elements contained within the <dynamicAnalysisTool> elements are the following:

- The <toolExecutablePath> element holds the path to the tool executable.

[10]As for anti-virus engines, the name is merely a visual identificator, and has no real impact on the execution.

- The <toolParameters> element holds the parameters that are supplied to the executable. An important feature with this string is that occurrences of the string "$samplePath" is replaced with the path to the malware sample on the virtual machine.

- The <executeMalwareExplicitly> element is a boolean value, holding either *true* or *false*. *true* indicates that the malware sample should be executed on the remote system after the tools have been started. Some tools do not require this, as they take the malware sample file as a parameter. In the latter case, the value should be *false*. If there is more than one tool installed on a single virtual machine, the malware sample is executed after the tools have been started if *any* of the tools have the <executeMalwareExplicitly> value *true*.

A.12 Redirection of console output

Some applications print their output to the command line console and use a log file format that is not human readable. This can make interpretation of the result hard. There is, however, a means for redirecting the command line console text output to a file that can be copied back to the client for further examination. This is done by running the application through the Microsoft Windows Command Prompt on the virtual machine. The Microsoft Windows Command Prompt executable, *cmd.exe*, has a switch that executes whatever is given after the switch and terminates. What makes this useful is that the Command Prompt allows for redirection of output from application run under it using the ">" operator. The following is an example XML configuration of a tool using output redirection[11]:

```
<dynamicAnalysisTool toolName="ipconfig">
  <toolExecutablePath>c:\windows\system32\cmd.exe</toolExecutablePath>
  <toolParameters>/C ipconfig.exe > c:\ipconfig.log</toolParameters>
  <toolResultFilePath>c:\ipconfig.log</toolResultFilePath>
  <executeMalwareExplicitly>false</executeMalwareExplicitly>
</dynamicAnalysisTool>
```

Listing A.11: Redirection of Command Prompt output on a virtual machine

[11]Note that the parameters contain both the file to be executed (ipconfig.exe) and the > operator. Any tool can be launched via the Command Prompt in a similar manner.

A.13 Understanding the properties file

The properties file is a simple text-based configuration file that is used to set global PowerScan constants. The format is a simple "key = value" scheme, where the values are parsed from string to other data types by the system, as shown in the table below. The following values are set in the properties file:

Property	Type	Default	Description
vmware.tools- .timeout	int	60	Timeout for VMware Tools to start in the Virtual Machines (VMware Tools are required for operation of PowerScan).
scanner- .executionpath	string	c:\\	Specifies where in the virtual machine the malware should be copied (and executed).
snapshot.before.scan	boolean	false	Specifies whether snapshots should be taken before performing scans and malware execution. This is generally NOT recommended, as storing snapshots is a time consuming operation, which should only be performed when changes are made to the Virtual Machine.
polling.interval- .minor	double	0.250	Sets polling interval for minor operations. This indicates how often the system should check for completion of minor non-blocking tasks, such as file execution.
polling.interval- .major	double	1.000	Set polling interval for major operations. This indicates how often the system should check for completion of major non-blocking tasks, such as full scan operations.
scanner.timeout	int	60	Maximum time to allow for individual scan operations to finish.
update.timeout	int	600	Maximum time to allow for updating to finish.
full.scan.timeout	int	600	Timeout for full scan. Used to prevent entire system from crashing following failure in a single scan thread. Should be greater scanner.timeout value above, since copying of files etc. is included in this timeout in addition to the actual scan.
malware.execution- .timeout	int	25	Maximum time to allow malware to execute with real-time anti-virus scanner running in the background.
full.execution- .timeout	int	240	Maximum total time for malware execution operation (for all registered scanners). Used to prevent the entire system from crashing following failure in a single thread.
log.overwrite	boolean	false	Specifies whether log files should be overwritten when program is executed. If set to false, time and date are appended to log files. Note that log files may take up significant resources over time.
log.vmware	string	logs\\vmware.log	Path for the VMware log file, where VMware-related events are logged.

Continued on next page

Property	Type	Default	Description
log.system.scanner	string	logs\\scanner.log	Path to the log file for Scanner operations.
log.system	string	logs\\system.log	Log path for system classes.
log.system.executor	string	logs\\executor.log	Log path for the executor class (dynamic analysis tools log).
executor.execution-path	string	c:\\	The path on the virtual machines where the malware sample is copied to and executed from when using the dynamic analysis functionality.
executor.localResult-Path	string	results\\	Directory on the client machine where log files should be copied when using the dynamic analysis functionality.
executor.overall-.timeout	int	330	Sets the time the operator has to interact with the analysis tool or malware sample logging in to the virtual machine before the result is copied back to local system.
xml.xsd.path	string	config.xsd	Sets the path for the XSD file used to validate the XML config file

All times are given in seconds. Note that backslashes ("\") must be escaped by another backslash, such that `c:\program files\` becomes `c:\\program files\\`. Also note that directories given on the remote machine must already be created on the virtual machine, as Vix 1.0 and 1.1 do not support directory creation.

A.14 PowerScan files

The following files are found in PowerScan:

`PowerScan.jar` The PowerScan jar file. Contains the PowerScan code. Executed by running *java -jar PowerScan.jar* (requires that Java 6 or newer is installed on the client machine). Note that the PowerScan file must be executed with the root directory of PowerScan as the working directory. In other words, the working directory must contain the other files described in this section.

`PowerScan GUI` Microsoft Windows shortcut which launches the PowerScan graphical user interface.

`PowerScan CLI` Microsoft Windows shortcut which launches the PowerScan command line interface using `cmd.exe`.

`vix.dll` The Vix C library used for communication with VMware components such as VMware Server hosts and virtual machines.

`ssleay32.dll` & `libeay32.dll` The OpenSSL toolkit for SSL/TLS. Extension used by Vix to enable secure communication.

`userguide.pdf` User manual of the PowerScan framework - this document.

`eclipse_project.zip` Eclipse project containing the needed files to import PowerScan in Eclipse.

config\ Configuration related files.

- `powerscan.properties` Default properties file for PowerScan, defining constants.
- `config.xml` Default XML configuration file for PowerScan.
- `config.xsd` XML Schema Definition (XSD) file used to validate the XML configuration.

lib\ External framework library files.

- `jna.jar` Java Native Access framework, used to communicate with C libraries.
- `simple-xml-1.7.1.jar` Simple framework, used to serialize and deserialize configuration to and from XML.
- `commons-cli-1.1.jar` Apache Commons Command Line Interface framework, used to offer the command line capability of PowerScan.
- `xercesImpl.jar` Xerces Java Parser framework, used to validate the XML configuration file against XSD.

logs\ Log files created by the PowerScan framework.

src\ The PowerScan source code.

javadoc\ The PowerScan javadoc files.

A.15 Understanding the log files

The PowerScan framework creates plain text formatted log file when executing operations. Paths for the log files are given in the .properties-file,

see section A.13 for details. It is important to notice that the property *log.overwrite* determines whether the log files should be overwritten, or if new log files should be created every time the PowerScan program is executed. Not overwriting the files is smart to keep history, but the number of log files can easily become overwhelmingly large.

Log files are written to `$workingDirectory\logs`. Files with the following prefixes are created:

system Used by the XML error handler, XML reader and Connector classes. These files contain log messages containing information about functionality such as reading from configuration file, initialization and connections to the hosts.

executor Used by the Executor class. An Executor object is a representation of a virtual machine with analysis tools installed. Information in these files includes copying of files, execution of programs and snapshot handling.

scanner Used by the Scanner class. A Scanner object is a representation of a virtual machine with an anti-virus engine installed. Execution of files, on-demand scans and the copying of files to and from a virtual machine is logged in these files.

vmware Used by GuestOS and VMwareServer classes. Contains information about operations against virtual machines, such as power-ons, login attempts, copying of files and the taking of and reverting to snapshots.

Appendix B

Compatibility testing of anti-virus engines

> "Reality is that which, when you stop believing in it, doesn't go away."

- Phillip K. Dick

B.1 Introduction

This appendix shows an overview of different commercial and non-commercial anti-virus engines that were researched during the development of PowerScan to get an idea of how well different anti-virus products fit into an automation framework. Note that the list is in no way exhaustive; it is an effort to test the solutions of the major vendors. However, a requirement for being in the list is that a license could be obtained - trial or full.

The following table lists the anti-virus products that are tested:

Product	*Producer*	*License*
Avast! Home Edition	Alwil	Free
AVG AV Free Edition	Grisoft	Free
Avira Antivir Personal Ed.	Avira	Free
BitDefender Free Ed.	Softwin	Free
BullGuard Internet Sec. 8.0	BullGuard	NOK 499/year
CA AntiVirus 2008	CA	USD 39,99/year
ClamWin AV	Open Source	GPL
Comodo AV 2.0 beta	COMODO	Free
F-prot	Frisk Software	USD 29/year
F-secure AV 2008	F-secure	NOK 595/year
Kaspersky AV 7.0	Kaspersky	USD 39,95/year
McAfee virusscan	McAfee	£31,99/year
Norman AV	Norman	NOK 549/year
Norton AV 2008	Symantec	NOK 559-649/year
Panda AV 2008	Panda Software	€39,95/year
Sophos AV SBE	Sophos	£133/year (5 users)
Trend Micro AV 2008	Trend Micro	£29,95/year
ZoneAlarmő AV	CheckPointő	€19,95/year

Prices are for a one PC one year subscription of the stand-alone installation of the AV engine when applicable. Some licenses offer installation on several computers. Different pricing models apply when the anti-virus engine is part of protection packages including firewall, email filtering, pop-up blocker and other features. Different pricing per unit also apply when buying licenses for multiple installations.

B.2 Anti-virus engine survey

Mapping the possibility to trigger on-demand scan operations and anti-virus definition file update using the command line is quite a time-consuming task;

each anti-virus engine must be downloaded and installed on a clean Windows XP SP2 installation (for example running on virtual machines to avoid interference between different installs). For each installation, command line on-demand scan possibilities must be tested, and console output or log files collected and analyzed in order to create scan log filters. The various commands, switches and parameters must be mapped by using trial and error, reading forum posts and guidelines and by issuing support request to the various vendors' customer support services.

On-demand scan is tested using a benign custom made file *test.bat* containing a simple echo command and the "malicious" *eicar.com* standard virus test file. Real-time scan is tested by downloading the *eicar.com* from `http://www.eicar.org/anti_virus_test_file.htm` onto the test machine. Log files for on-demand scanners must then be located and reviewed.

The following sections describe the results obtained from the anti-virus engine survey.

Avast! Home Edition

On-demand scan: `.\Alwil Software\Avast4\ashQuick.exe $sampleFile`. However, this does not report result to console.

On-demand update: `.\Alwil Software\Avast4\ashUpd.exe vps /silent`

Real-time log file: `.\Alwil Software\Avast4\DATA\log\Warning`

AVG AV Free edition

On-demand scan: On-demand scan can be initiated by using `.\AVG\AVG8\avgscanx.exe /SCAN $sampleFile`

On-demand update: `.\AVG\AVG8\avgupd.exe`

Real-time log file: `C:\Documents and Settings\All Users\Application Data\avg8\Log\history.xml`

Avira AntiVir

On-demand scan: `.\Avira\AntiVir PersonalEdition Classic\avcls.exe $sampleFile < c:\input.txt`. This requires that the `c:\in-`

put.txt file exists and contains a single DOS-style newline[1]. This is to avoid hang at the "Please press the Enter key to exit" message. Note that the avcls.exe program is an addition to the Avira AntiVir-suite and needs to be downloaded separately from http://www.avira.com/en/support/support_downloads.html.

On-demand update: .\Avira\AntiVir PersonalEdition Classic\preupd.exe. This launches GUI, which closes automatically after 10 sec (so no interaction is needed).

Real-time log file: C:\Documents and Settings\All Users\Application Data\Avira\AntiVir PersonalEdition Classic\LOGFILES\¬ avguard.log

BitDefender

On-demand scan: C:\Program Files\Common Files\Softwin\BitDefender Scan Server\bdc.exe $sampleFile /f

On-demand update: C:\Program Files\Common Files\Softwin\BitDefender Scan Server\bdc.exe /update

Real-time log file:

Bullguard AV

On-demand scan: .\BullGuard Ltd\BullGuard\BgScan.exe $sampleFile

On-demand update: .\BullGuard Ltd\BullGuard\BullGuardUpdate.¬ exe

Real-time log file: C:\Documents and Settings\All Users\Application Data\BullGuard\Logs\OnAccess.log

CA AV

On-demand scan: .\CA\CA Internet Security Suite\CA Anti-Virus¬ \caavcmdscan.exe $sampleFile

[1]Create new file and press the "enter" key once. This produces the CR+LF sequence.

On-demand update: `.\CA\CA Internet Security Suite \ccupdate\¬ ccupdate.exe`. Launches updater, but requires GUI user interaction. `.\CA\CA Internet Security Suite\CA Anti-Virus\caav.exe /¬ UPDATE` also launches GUI. According to customer support, it is not possible to update using only command line. However, it should be possible to script this functionality[2].

Real-time log file: `C:\Documents and Settings\All Users\Application Data\CA\Consumer\AV\RealTimeScannerLog.txt`.

ClamWin

On-demand scan: `.\ClamWin\bin\clamscan.exe $sampleFile`. Requires some additional parameters, such as redirection of output, path to database files and so on.

On-demand update: `.\ClamWin\bin\freshclam.exe`. Requires a freshclam.conf file in the same directory as the executable file. The .conf file must at least contain the following lines:

```
DNSDatabaseInfo current.cvd.clamav.net
DatabaseMirror database.clamav.net
MaxAttempts 3
```

Real-time log file:

Comodo AV

On-demand scan: `.\Comodo\Comodo AntiVirus\CAVSCons.exe $sample-File`

On-demand update: `.\Comodo\Comodo AntiVirus\CavMUD.exe`. Launches updater, but requires GUI user interaction. Support stated that stand-alone command line usage will be considered implemented in the future.

Real-time log file: `C:\Documents and Settings\All Users\Application Data\Comodo\Comodo AntiVirus\Reports\OnAccessReport.¬ Log`

[2]See `http://homeofficekb.ca.com/CIDocument.asp?KDId=2898&Preview=0&Return=0&GUID=51C871EC7AE74D04929BF0E824FE7D7F`.

F-prot

On-demand scan: `.\FRISK Software\F-PROT Antivirus for Windows¬ \fpscan.exe -o logPath $sampleFile`

On-demand update: Unknown, support has been contacted but no reply received.

Real-time log file: F-prot logs all events to the Windows Event Viewer.

F-Secure AV Client Security

On-demand scan: `C:\Program Files\F-Secure\Anti-Virus\fsav.exe $¬ sampleFile`

On-demand update: `C:\Program Files\F-Secure\Anti-Virus\getdb-htp.exe -url=http://avupdate.f-secure.com/updates/ -gui=1 -ver=FSAV6`.

Real-time log file: `C:\Program Files\F-Secure\common\LogFile.log`

Kaspersky

On-demand scan: `.\Kaspersky Lab\Kaspersky Anti-Virus 7.0\avp¬ .com SCAN "$sampleFile" /i0 /R:"$logPath"`

On-demand update: `.\Kaspersky Lab \Kaspersky Anti-Virus 7.0\¬ avp.com UPDATE`

Real-time log file: Log files from on-demand scan is in ok format, however the event log is not human readable (`C:\Documents and Settings\¬ All Users\Application Data\Kaspersky Lab\AVP7\`).

McAfee AV

On-demand scan: Command line is according to customer service not available in home-editions, but is possible in enterprise-edition with "some modification"[3].

[3] `http://community.mcafee.com/showthread.php?t=215621`

On-demand update:

Real-time log file:

Norman

On-demand scan: `.\Norman\nvc\bin\Nvcc.exe /U /LF:$logFile $sampleFile`.

On-demand update: `C:\Program Files\Norman\Npm\Bin\niu.exe`

Real-time log file: Real-time scanner does not create any log files in the current version. According to customer support, this is a planned feature in future versions[4].

Norton

On-demand scan: According to customer support, command line on-demand scan of individual files is not possible in home-version, without the GUI being launched. Works with full system scan.

On-demand update:

Real-time log file:

Panda AntiVirus 2008

On-demand scan: Has a separate command line tool[5].

On-demand update: The command line scanner does not seem to support automatic update, but can be scripted[6].

Real-time log file: An inquiry about the location of log files was sent to customer support, but no reply has been received.

[4]See the norman support forum at `http://forum.norman.com/viewtopic.php?p=8640`.

[5]Can be downloaded from `http://research.pandasecurity.com/blogs/images/pavcl.zip`.

[6]An excellent description of how this can be done for Panda is found at `http://forums.theplanet.com/index.php?showtopic=42950`. The script described in this post can probably be used for other engines as well.

Sophos

On-demand scan: Initiate on-demand scan: `.\Sophos\Sophos Anti-¬ Virus\sav32cli.exe -p=$logFile $sampleFile`

On-demand update: Automatic update can be scripted. Need to download and execute a self-extracting executable archive file[7].

Real-time log file: Real-time scan events are written to `C:\Documents and Settings\All Users\Application Data\Sophos\Sophos Anti¬ -Virus\logs\SAV.txt`

Trend Micro AntiVirus plus AntiSpyware 2008

On-demand scan: Trend has a command line scanner,[8], but is unable to unpack the required files for testing. Received from customer support: To run a Scan, use `.\Trend Micro\Internet Security\UfNavi.exe /a UfSNavi.ini`.

On-demand update: Not possible.

Real-time log file: Not human readable.

ZoneAlarm Antivirus

On-demand scan: `.\Zone Labs\ZoneAlarm\multiscan.exe $sampleFile`. This launches GUI. The application has no switch to disable GUI.

On-demand update:

Real-time log file: `c:\windows\internet logs\ZALog.txt`

B.3 Summary

The following table summarizes the anti-virus engine survey.

[7] See the Sophos knowledge base at `http://www.sophos.com/support/knowledgebase/article/10378.html`

[8] `http://esupport.trendmicro.com/support/viewxml.do?ContentID=en-117058`

Product	Scan[a]	Update[b]	Real-time scan[c]	Notes
Avast! Home Edition	Y	Y	Y	
AVG AV Free Edition	Y	Y	Y	
Avira AntiVir Personal Ed.	Y	Y	Y	
BitDefender Free Ed.	Y	Y	N	
BullGuard Internet Sec. 8.0	Y	Y	Y	
CA AntiVirus 2008	Y	Y	Y	
ClamWin AV	Y	Y	N	
Comodo AV 2.0 beta	Y	N	Y	*[d]
F-prot	Y	N	N	
F-Secure AV 2008	Y	Y	Y	
Kaspersky AV 7.0	Y	Y	N	*[e]
McAfee virus scan	N	N	Y	*[f]
Norman AV	Y	Y	Y	
Norton AV 2008	N	N	Y	
Panda AV 2008	Y	Y	N	*[g]
Sophos AV SBE	Y	Y	Y	*[h]
Trend Micro 2008	Y	N	N	
ZoneAlarmő AV	N	N	Y	

[a] Usable on-demand command line scanner

[b] Usable on-demand command line update feature

[c] Usable real-time scanner with parsable log files

[d] On-demand update not possible in current release, will be considered for further versions.

[e] Event logs are written in a seemingly unparsable proprietary format, so the on-demand scanner can not be used.

[f] According to customer support, no command line possibilities in home edition. Is possible in enterprise edition with "some modification".

[g] Uses a separate command line tool, but this tool does not support automatic updates. Automatic updates must be scripted so that the downloads are automatically downloaded and installed.

[h] A guide for automatic updates can be found at `http://www.sophos.com/support/knowledgebase/article/10378.html`.

Appendix C

Dynamic malware analysis tool survey

> "In the struggle for survival, the fittest win out at the expense of their rivals because they succeed in adapting themselves best to their environment."

- Charles Darwin

C.1 Overview

There exist a lot of malware analysis tools. However, to be suitable for use with the PowerScan framework, the tools should require low interaction and have the ability be initiated via the command line and print results to file. The following sections look into some different tools that have been tested to see if they would fit into the framework. Note that the overview is in no way exhaustive, but it still gives an introduction to the domain.

C.2 Sandbox solutions

The following presents an overview of the analysis of sandbox analysis tools:

Product	Command line?	Suitable?
CWSandbox	Unknown	Possibly
Norman Sandbox Analyzer	Y	Y
Norman Sandbox Analyzer Pro	N	N

CWSandbox

URL: `http://www.cwsandbox.org/orhttp://www.sunbelt-software.com/Developer/Sunbelt-CWSandbox/`

Usage: On-line web interface is available. Has got quite high system requirements for installation, see `http://www.sunbelt-software.com/Developer/Sunbelt-CWSandbox/Requirements/`. Is likely possible to run from the command line, but this has not been tested. Obtaining a custom use license does not appear to be possible.

Note: Complete sandbox solution that uses API hooking and DLL code injection to analyze behavior. Presents results in plain text or HTML.

Norman Sandbox Analyzer

URL: `http://www.norman.com/microsites/malwareanalyzer/Products/analyzer`

Usage: GUI, but also supports command line usage for automation. Command line parameters well documented in the user manual. Using `.\Norman SandBox Analyzer\analyzer.exe /a:$logPath /p:c:\¬ NSA\filter.ini /d:c:\nsa\ $samplePath`

Note: Has summary view, but also API log, dropped files overview, IRC server and visited URL view. Summary may be written in

Norman Sandbox Analyzer Pro

URL: `http://www.norman.com/microsites/malwareanalyzer/Products/analyzer-pro`

Usage: GUI. User manual states: "Analyzer Pro is a highly specialized tool that was never meant to be used as a command line or automation tool. Analyzer Pro will accept a few parameters from the commandline."

Notes: In addition to functionality of the Norman Sandbox Analyzer, the Pro edition also include a disassembler view, created thread view, memory dump view and a Live Internet Communicator analyzer module.

C.3 Registry monitors

The following presents an overview of the analysis of registry monitors:

Product	Command line?	Suitable?
RegShot	N	Y
RegMon	N	Y
Registry Watch	N	Y
SpyMe Tools	N	Y

RegShot

URL: `https://sourceforge.net/projects/regshot`

Usage: GUI only.

Notes: Takes snapshots of the registry before and after an action, and compares them producing a pure text or HTML formatted report on added and deleted registry keys.

Sysinternal RegMon

URL: `http://www.microsoft.com/technet/sysinternals/ProcessesAndThreads/Regmon.mspx`

Usage: GUI only.

Notes: Regmon have been replaced by Process Monitor on versions of Windows starting with Windows XP SP2. Works by performing system call hooking on registry related calls. Has filtering option to filter processes regarding only one process, but this can not be given as parameter. Log can be saved from GUI.

Easydesksoftware Registry Watch

URL: `http://www.easydesksoftware.com/regwatch.htm`

Usage: GUI only.

Notes: Compares snapshots of the registry. Produces text formatted report showing old and new values of changed keys.

LC IBros Solutions SpyMe Tools

URL: `http://www.lcibrossolutions.com/spyme_tools.htm`

Usage: GUI only.

Notes: Compares two snapshots of the registry or file system, and produces a differences list.

C.4 File system monitors

The following presents an overview of the analysis of file system monitors:

Product	Command line?	Suitable?
System Change Log	Y	Y
DiskMon	N	N
FileMon	N	Y
hIOmon	N	N
Handle	Y	N

System Change Log

URL: `http://www.greyware.com/software/systemchangelog/`

Usage: Installs as a service. Look at `http://support.microsoft.com/?kbid=310399` for instructions on how to enable auditing in Win XP Pro

Notes: This system runs as a service, and logs file system activity. Logs to `c:\windows\system32\scl.log`

Sysinternals Disk Monitor (DiskMon)

URL: `http://technet.microsoft.com/nb-no/sysinternals/bb896646(en-us).aspx`

Usage: GUI only.

Notes: Uses kernel event tracing. Monitors read and write operations to hard drive sectors, not files and directories.

Sysinternals File Monitor (FileMon)

URL: `http://technet.microsoft.com/nb-no/sysinternals/bb896642(en-us).aspx`

Usage: GUI only.

Notes: Uses a file system API hook. Filemon have been replaced by Process Monitor on versions of Windows starting with Windows XP SP2. Monitors operations performed on files in the Windows file system. It is possible to filter on specific files or directories. Starting in version 4.1 FileMon is able to monitor named pipe and mail slot file system activity on Windows NT/2K.

hIOmon

URL: `http://www.hyperio.com/hIOmon/hIOmon.htm`

Usage: Requires GUI, the presentation client requires Java to run. Allows exporting (to a CSV-formatted file) the file or disk I/O trace data that it has collected.

Notes: Intended used as a performance monitor, but can be used for tracking file operations.

Sysinternals Handle

URL: `http://technet.microsoft.com/nb-no/sysinternals/bb896655(en-us).aspx`

Usage: `.\handle.exe -p \$sampleFile`

Notes: Displays information about open file handles for any process in the system. Only reports the *current* open handles without providing functionality for tracing over time. Handle has to be started as the process to be monitored is executing. Thus not that usable in our context. Has a GUI version also, see `http://technet.microsoft.com/nb-no/sysinternals/bb896653(en-us).aspx`.

C.5 Process monitors

Product	Command line?	Suitable?
Process Monitor	Y	Y
PsTools	Y	Y
PrcView	Y	Y

Sysinternals Process Monitor

URL: `http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx`

Usage: Example of .bat-file:

```
set PM=C:\Progra~1\procmon\Procmon.exe
start %PM% /Quiet /minimized /backingfile $logPath\$logFile.pml
%PM% /waitforidle
$program_to_be_monitored.exe
%PM% /terminate
%PM% /saveas $logfile.xml /openlog $logPath\$logFile.pml
```

Notes: Possible to get log files in XML, but must be converted after monitoring is completed. See Sysinternals forum - `http://forum.sysinternals.com/forum_posts.asp?TID=13843&KW=command+line&PID=64569#64569`. NB! Extensive logging!. This tool combines features of the legacy tools Sysinternal Filemon and Regmon.

Sysinternals PsTools

URL: `http://technet.microsoft.com/nb-no/sysinternals/bb896682(en-us).aspx`

Usage: `.\pslist.exe -t > c:\pslist.log`

Notes: The tool *PsList* in the *PsTools*-package has an option to get process information from a remote computer. Can also be used to get prosess tree from the local computer.

PrcView

URL: `http://www.faratasystems.com/pview/prcview.htm`

Usage: `.\pv.exe [arguments]`

Notes: This tool has a command line utility that allows the user to extract extended information about running processes.

C.6 Network activity monitors

For list; see `http://www.woodmann.com/collaborative/tools/index.php/Category:Network_Sniffers`

Product	Command line?	Suitable?
TCPView	Y	Y
TCPDump	Y	Y
WireShark	Y	Y
Sniff Hit	N	Y

Sysinternals TCPView

URL: `http://technet.microsoft.com/en-us/sysinternals/bb897437.aspx`

Usage: `.\TcpView\tcpvcon.exe -a $sampleFile > $logFile`

Notes: TCPView is a tool to monitor all open TCP and UDP endpoints. The TCPView tool is based on a GUI, but contains a command-line tool called Tcpvcon.

TCPDump for Windows (Windump)

URL: `http://www.winpcap.org/windump/install/default.htm`

Usage: `C:\windump\WinDump.exe -i\$interfaceNr > \$logFile`

Notes: Windump is Windows version of the *tcpdump* tool for Unix, and is built on the Windows version of the Unix *libpcap* API.

WireShark

URL: `http://www.wireshark.org/`

Usage: `.\wireshark.exe [options]`

Notes: WireShark can be run from the command line, and supports a large number of parameters. Output can be redirected to file (using the -w switch) The filters will probably have to be prepared in advance. Output files needs to be opened in WireShark for analysis.

Sniff Hit

URL: `http://labs.idefense.com/files/labs/releases/previews/map/sniff_hit.html`

Usage:

Notes: Is a HTTP, IRC, and DNS sniffer. Part of the iDefence Malcode Analysis Pack.

C.7 Packages and tool sets

Product	Command line?	Suitable?
All-seeing Eye	N	N
Windows Resource Tool Kit	Y	N
SysAnalyzer	N	Y
Malcode Analysis Pack	both	Y

All-seeing Eye

URL: `http://www.fortego.com/en/ase.html`

Usage: Based on GUI, and requires user interaction to create rules. Shows GUI warnings for events like loading of DLLs, registering of services etc.

Notes: A tool that monitors processes, DLL usage, drivers and services, file system, registry, browser helper objects and network sockets. Based on a learning mechanism, the different modules tries to separate normal behavior from malicious actions.

Windows Resource Kit Tools

URL: `http://www.microsoft.com/downloads/details.aspx?FamilyID=9d467a69-57ff-4ae7-96ee-b18c4790cffd&DisplayLang=en`

Usage: Various command line tools.

Notes: Contains process monitoring tools, but is unable to redirect output.

iDefence SysAnalyzer

URL: `http://labs.idefense.com/files/labs/releases/previews/SysAnalyzer/`

Usage: Requires GUI.

Notes: Works mainly by comparing system snapshots, this is due to efficiency reasons. However, the tool also include some real-time logging to avoid missing critical operations. Snapshots cover processes, network endpoints, loaded system drivers, DLLs loaded into Explorer and certain registry keys.

iDefence Malcode Analysis Pack

URL: `http://labs.idefense.com/files/labs/releases/previews/map/`

Usage:

Notes: Collection of various tools, including TCP client, sniff_hit[1], mail server capturer, fakeDNS spoofing tool, a hidden process detector and various other shell tools.

C.8 System call analysis

Product	Command line?	Suitable?
KAM	N	Y
Strace 0.3	Y	Y
StraceNT0.8	Y	N
APIScan	Y	N
Detours	Y	Y
ListDLLs	Y	N
Omega Red syscall	??	N
Rohitab API mon	N	Y

[1]IRC, HTTP and DNS sniffer

KaKeeware Application Monitor (KAM)

URL: `http://www.kakeeware.com/i_kam.php`

Usage: Requires GUI.

Notes: Is a DLL/API spy application.

Strace for NT 0.3

URL: `http://www.woodmann.com/collaborative/tools/index.php/Strace_for_NT`

Usage: `.\strace-0.3\app\Release\strace.exe -e [APIs to monitor] -o $logFile $samplePath`

Notes: Strace has preliminary support for Windows XP. On Windows XP, it is necessary to set the registry key `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\EnforceWriteProtection` to REG_DWORD 0. and reboot before using strace. This disables the kernel from checking for errant memory overwrites, and is not a good thing, in general. It is currently necessary for strace because the system call table is write protected, and strace needs to modify it. Shows some parameters as readable text. Filtering can be done using the *-e* command line switch with the system call names as arguments.

IntellectualHeaven StraceNT 0.8

URL: `http://www.intellectualheaven.com/default.asp?BH=projects&H=strace.htm`

Usage: `.\strace\StraceNt.exe -f stFilter.txt $samplePath > $logPath`

Notes: Is possible to filter system calls via a separate text filter file, but does not show arguments in human readable format.

APIScan

URL: `http://www.woodmann.com/collaborative/tools/index.php/APIScan`

Usage: `.\apiscan\APIScan.exe $samplePath`

Notes: Creates a log file in the directory of the target application containing all the API calles issued from that process. Does not show the APIs in sequencial order, and does not show arguments passed with the API calls.

Detours

URL: `http://research.microsoft.com/sn/detours/`

Usage:

Notes: Must be compiled from C++

ListDLLs

URL: `http://technet.microsoft.com/nb-no/sysinternals/bb896656(en-us).aspx`

Usage: `.\listdlls\Listdlls.exe $samplePath > $logFile`

Notes: Must be run while the monitored process/application is running. This means timing issue when automating.

Omega Red syscall

URL: `http://ry.pl/~omega/progs/syscall.zip`

Usage:

Notes: Have tried installation on several machines, but the program seems to fail every time.

Rohitab API monitor

URL: `http://www.rohitab.com/apimonitor/`

Usage: Requires GUI

Notes: Seems to have good filtering capabilities, possible to filter both processes and APIs. The arguments are in human readable form, and status codes are given for errors.

C.9 General analysis tools

Product	Command line?	Suitable?
Red Curtain	N	Y

MANDIANT Red Curtain

URL: `http://www.mandiant.com/redcurtain.htm`

Usage: Requires .NET framework.

Notes: MRC examines executable files to determine how suspicious they are based on a set of criteria such as the entropy (in other words, randomness), indications of packing, compiler and packing signatures, digital signatures, PE anomalies and other characteristics to generate a threat score.

Appendix D

Configuration file examples

> "Rules make the learner's path long, examples make it short and successful"

- Seneca (Roman philosopher, mid-1st century AD)

D.1 Example XML config file

```
<PowerScan>
   <VMwareHostList>
      <VMwareHost host="dhcp208-210.ed.ntnu.no">
         <hostPortNumber>902</hostPortNumber>
         <hostUsername>XXXXXXX</hostUsername>
         <hostPassword>XXXXXXX</hostPassword>
         <vmList>
            <VM vmxPath="/var/lib/vmware-server/Virtual Machines/winxp_0/
                Windows XP Professional.vmx">
               <vmUsername>XXXXXXX</vmUsername>
               <vmPassword>XXXXXXX</vmPassword>
               <avEngine name="clamwin">
                  <avExecutablePath>C:\Program Files\ClamWin\bin\clamscan.
                      exe</avExecutablePath>
                  <avParameters>--stdout --database="c:\program files\
                      clamwin\bin" --log=c:\result.log "$samplePath"</
                      avParameters>
                  <avLogFilePath>c:\result.log</avLogFilePath>
                  <avLogFilter>
                     <avResultIdentifier>FOUND</avResultIdentifier>
                     <avResultPrefix>: </avResultPrefix>
                     <avResultSuffix> FOUND</avResultSuffix>
                  </avLogFilter>
                  <avUpdateInfo>
                     <avUpdateExecutable>c:\program files\clamwin\bin\
                         freshclam.exe</avUpdateExecutable>
                     <avUpdateParameters>--stdout --config-file="c:\program
                         files\clamwin\bin\clamd.conf" --datadir="c:\program
                          files\clamwin\bin" --log="c:\update.log" </
                         avUpdateParameters>
                     <avUpdateLogPath>c:\update.log</avUpdateLogPath>
                     <avUpdateSuccessIndicator>Database updated</
                         avUpdateSuccessIndicator>
                  </avUpdateInfo>
                  <realTimeScan>
                     <rtLogPath></rtLogPath>
                     <rtResultIdentifier></rtResultIdentifier>
                  </realTimeScan>
               </avEngine>
               <analysisTools>
                  <dynamicAnalysisTool toolName="IPconfig">
                     <toolExecutablePath>c:\windows\system32\cmd.exe</
                         toolExecutablePath>
                     <toolParameters>/C ipconfig > c:\ipconfig2.log</
                         toolParameters>
                     <toolResultFilePath>c:\ipconfig2.log</
                         toolResultFilePath>
                     <executeMalwareExplicitly>false</
                         executeMalwareExplicitly>
                  </dynamicAnalysisTool>
                  <dynamicAnalysisTool toolName="Netstat">
                     <toolExecutablePath>c:\windows\system32\cmd.exe</
                         toolExecutablePath>
                     <toolParameters>/C netstat -a > c:\netstat.log</
                         toolParameters>
                     <toolResultFilePath>c:\netstat.log</toolResultFilePath>
                     <executeMalwareExplicitly>false</
                         executeMalwareExplicitly>
                  </dynamicAnalysisTool>
```

```
            </analysisTools>
         </VM>
         <VM vmxPath="/var/lib/vmware-server/Virtual Machines/
             winxp_default/Windows XP Professional.vmx">
            <vmUsername>XXXXXXX</vmUsername>
            <vmPassword>XXXXXXX</vmPassword>
            <avEngine name="F-Secure 7.10">
               <avExecutablePath>c:\Program files\f-secure\Anti-Virus\
                   fsav.exe</avExecutablePath>
               <avParameters>/REPORT=c:\result.log "$samplePath"</
                   avParameters>
               <avLogFilePath>c:\result.log</avLogFilePath>
               <avLogFilter>
                  <avResultIdentifier>Infection</avResultIdentifier>
                  <avResultPrefix>: </avResultPrefix>
                  <avResultSuffix></avResultSuffix>
               </avLogFilter>
               <avUpdateInfo>
                  <avUpdateExecutable>c:\program files\f-secure\anti-
                      virus\getdbhtp.exe</avUpdateExecutable>
                  <avUpdateParameters>-url=http://avupdate.f-secure.com/
                      updates/ -gui=1 -ver=FSAV6</avUpdateParameters>
                  <avUpdateLogPath></avUpdateLogPath>
                  <avUpdateSuccessIndicator></avUpdateSuccessIndicator>
               </avUpdateInfo>
               <realTimeScan>
                  <rtLogPath></rtLogPath>
                  <rtResultIdentifier></rtResultIdentifier>
               </realTimeScan>
            </avEngine>
         </VM>
         <VM vmxPath="/var/lib/vmware-server/Virtual Machines/winxp_1/
             Windows XP Professional.vmx">
            <vmUsername>XXXXXXX</vmUsername>
            <vmPassword>XXXXXXX</vmPassword>
            <avEngine name="AVG">
               <avExecutablePath>c:\Program files\avg\avg8\avgscanx.exe</
                   avExecutablePath>
               <avParameters>/REPORT=c:\report.txt /SCAN=$samplePath</
                   avParameters>
               <avLogFilePath>c:\report.txt</avLogFilePath>
               <avLogFilter>
                  <avResultIdentifier>identified </avResultIdentifier>
                  <avResultPrefix></avResultPrefix>
                  <avResultSuffix></avResultSuffix>
               </avLogFilter>
               <avUpdateInfo>
                  <avUpdateExecutable>C:\Program Files\AVG\AVG8\avgupd.
                      exe</avUpdateExecutable>
                  <avUpdateParameters></avUpdateParameters>
                  <avUpdateLogPath></avUpdateLogPath>
                  <avUpdateSuccessIndicator></avUpdateSuccessIndicator>
               </avUpdateInfo>
               <realTimeScan>
                  <rtLogPath>C:\Documents and Settings\All Users\
                      Application Data\avg8\Log\avgrs.log</rtLogPath>
                  <rtResultIdentifier>EID_Id_vir</rtResultIdentifier>
               </realTimeScan>
            </avEngine>
         </VM>
      </vmList>
   </VMwareHost>
```

```
      <VMwareHost host="129.241.208.158">
         <hostPortNumber>902</hostPortNumber>
         <hostUsername>XXXXXXX</hostUsername>
         <hostPassword>XXXXXXX</hostPassword>
         <vmList>
            <VM vmxPath="C:\Virtual Machines\Windows XP Professional\Windows
                 XP Professional.vmx">
               <vmUsername>XXXXXXX</vmUsername>
               <vmPassword>XXXXXXX</vmPassword>
               <analysisTools>
                  <dynamicAnalysisTool toolName="Norman Sandbox Analyzer">
                     <toolExecutablePath>C:\NSA>analyzer.exe</
                         toolExecutablePath>
                     <toolParameters> /d:c:\nsa /a:c:\nsa.log $samplePath</
                         toolParameters>
                     <toolResultFilePath>c:\nsa.log</toolResultFilePath>
                     <executeMalwareExplicitly>false</
                         executeMalwareExplicitly>
                  </dynamicAnalysisTool>
               </analysisTools>
            </VM>
         </vmList>
      </VMwareHost>
   </VMwareHostList>
</PowerScan>
```

Listing D.1: A sample XML configuration file

D.2 PowerScan XML Schema Definition

The following listing shows the PowerScan XML Schema Definition (XSD), which defines the rules for the PowerScan XML configuration file.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<!-- Define the simple elements first -->
<xsd:element name="hostPassword" type="xsd:string"/>
<xsd:element name="hostUsername" type="xsd:string"/>
<xsd:element name="avResultIdentifier" type="xsd:string"/>
<xsd:element name="avResultPrefix" type="xsd:string"/>
<xsd:element name="avResultSuffix" type="xsd:string"/>
<xsd:element name="avUpdateExecutable" type="xsd:string"/>
<xsd:element name="avUpdateParameters" type="xsd:string"/>
<xsd:element name="avUpdateLogPath" type="xsd:string"/>
<xsd:element name="avUpdateSuccessIndicator" type="xsd:string"/>
<xsd:element name="avExecutablePath" type="xsd:string"/>
<xsd:element name="avParameters" type="xsd:string"/>
<xsd:element name="avLogFilePath" type="xsd:string"/>
<xsd:element name="vmUsername" type="xsd:string"/>
<xsd:element name="vmPassword" type="xsd:string"/>
<xsd:element name="toolExecutablePath" type="xsd:string"/>
<xsd:element name="toolParameters" type="xsd:string"/>
<xsd:element name="toolResultFilePath" type="xsd:string"/>
<xsd:element name="rtLogPath" type="xsd:string"/>
<xsd:element name="rtResultIdentifier" type="xsd:string"/>
<xsd:element name="executeMalwareExplicitly" type="xsd:boolean"/>
<xsd:element name="hostPortNumber" type="xsd:integer"/>

<!-- Define attributes -->

<xsd:attribute name="host" type="xsd:string"/>
<xsd:attribute name="name" type="xsd:string"/>
<xsd:attribute name="vmxPath" type="xsd:string"/>
<xsd:attribute name="toolName" type="xsd:string"/>

<!-- Define the complex elements -->

<xsd:element name="VMwareHostList">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" minOccurs="1" ref="VMwareHost"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="VMwareHost">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element maxOccurs="1" minOccurs="1" ref="hostPortNumber"/>
      <xsd:element maxOccurs="1" minOccurs="1" ref="hostUsername"/>
      <xsd:element maxOccurs="1" minOccurs="1" ref="hostPassword"/>
      <xsd:element maxOccurs="1" minOccurs="1" ref="vmList"/>
    </xsd:sequence>
    <xsd:attribute ref="host" use="required"/>
  </xsd:complexType>
</xsd:element>
```

```
<xsd:element name="vmList">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" minOccurs="0" ref="VM"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="avLogFilter">
  <xsd:complexType mixed="true">
    <xsd:sequence>
      <xsd:element maxOccurs="1" minOccurs="1" ref="avResultIdentifier"/>
      <xsd:element maxOccurs="1" minOccurs="0" ref="avResultPrefix"/>
      <xsd:element maxOccurs="1" minOccurs="0" ref="avResultSuffix"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="avUpdateInfo">
  <xsd:complexType mixed="true">
    <xsd:sequence>
      <xsd:element maxOccurs="1" minOccurs="0" ref="avUpdateExecutable"/>
      <xsd:element maxOccurs="1" minOccurs="0" ref="avUpdateParameters"/>
      <xsd:element maxOccurs="1" minOccurs="0" ref="avUpdateLogPath"/>
      <xsd:element maxOccurs="1" minOccurs="0" ref="avUpdateSuccessIndicator
          "/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="avEngine">
  <xsd:complexType mixed="true">
    <xsd:sequence>
      <xsd:element maxOccurs="1" minOccurs="1" ref="avExecutablePath"/>
      <xsd:element maxOccurs="1" minOccurs="1" ref="avParameters"/>
      <xsd:element maxOccurs="1" minOccurs="1" ref="avLogFilePath"/>
      <xsd:element maxOccurs="1" minOccurs="0" ref="avLogFilter"/>
      <xsd:element maxOccurs="1" minOccurs="0" ref="avUpdateInfo"/>
      <xsd:element maxOccurs="1" minOccurs="0" ref="realTimeScan"/>
    </xsd:sequence>
    <xsd:attribute ref="name" use="required"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name="realTimeScan">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element maxOccurs="1" minOccurs="1" ref="rtLogPath"/>
      <xsd:element maxOccurs="1" minOccurs="0" ref="rtResultIdentifier"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="analysisTools">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" minOccurs="0" ref="
          dynamicAnalysisTool"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

```
<xsd:element name="dynamicAnalysisTool">
<xsd:complexType mixed="true">
  <xsd:sequence>
    <xsd:element maxOccurs="1" minOccurs="1" ref="toolExecutablePath"/>
    <xsd:element maxOccurs="1" minOccurs="0" ref="toolParameters"/>
    <xsd:element maxOccurs="1" minOccurs="0" ref="toolResultFilePath"/>
    <xsd:element maxOccurs="1" minOccurs="0" ref="executeMalwareExplicitly"/
        >
  </xsd:sequence>
  <xsd:attribute ref="toolName" use="required"/>
</xsd:complexType>
</xsd:element>

<xsd:element name="VM">
<xsd:complexType mixed="true">
  <xsd:sequence>
    <xsd:element maxOccurs="1" minOccurs="1" ref="vmUsername"/>
    <xsd:element maxOccurs="1" minOccurs="1" ref="vmPassword"/>
    <xsd:element maxOccurs="1" minOccurs="0" ref="avEngine"/>
    <xsd:element maxOccurs="1" minOccurs="0" ref="analysisTools"/>
  </xsd:sequence>
  <xsd:attribute ref="vmxPath" use="required"/>
</xsd:complexType>
</xsd:element>

<!-- Define the overall structure -->
<!-- Starting with one VMwareHostList element -->
<xsd:element name="PowerScan">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element maxOccurs="1" minOccurs="1" ref="VMwareHostList"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

</xsd:schema>
```

Listing D.2: The W3C XML Schema definition

D.3 Description of PowerScan's XML with respect to the XSD schema

This section describes the setup of PowerScan's XML configuration file, with respect to the defined XSD schema. It gives a description of the XML elements and their limitations, as they are described in the XSD. For the actual XSD schema and an example XML file, refer to the sections above.

Schema languages are useful, as they can be used to verify that the given XML config file is not only well-formed, meaning that it conforms to the XML standard, but also that it conforms to specified rules concerning which elements are optional, the number of allowed sub elements within elements and so on.

When initiated, the PowerScan framework needs to be told how the lab environment it is supposed to utilize is set up. The required information includes how many VMware hosts are supposed to be used, their IP address or host name, username and password for the host, how many VMware guest OS instances are running and which AV engines or analysis tools are installed on them and how the tools are to be executed. The layout of the XML file is described below:

The XML file starts with one root element called <PowerScan>. A <PowerScan> element is defined in XSD as follows:

```
<xsd:element name="PowerScan">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="VMwareHostList" minOccurs="0" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Listing D.3: XSD PowerScan element

This means that a <PowerScan> element contains one and only one element of the type <VMwareHostList>. The <VMwareHostList> element is also a complex element[1], defined as

```
<xsd:element name="VMwareHostList">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="VMwareHost" minOccurs="1" maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
```

[1] A complex element is an element that contains other elements and/or attributes.

```
</xsd:element>
```

Listing D.4: XSD VMwareHostList element

This shows that a <VMwareHostList> element is a list containing at least one but possibly an arbitrary high number of <VMwareHost> elements. The <VMwareHost> element contains the following elements:

```
<xsd:element name="VMwareHost">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element maxOccurs="1" minOccurs="1" ref="hostPortNumber"/>
      <xsd:element maxOccurs="1" minOccurs="1" ref="hostUsername"/>
      <xsd:element maxOccurs="1" minOccurs="1" ref="hostPassword"/>
      <xsd:element maxOccurs="1" minOccurs="1" ref="vmList"/>
    </xsd:sequence>
    <xsd:attribute ref="host" use="required"/>
  </xsd:complexType>
</xsd:element>
```

Listing D.5: XSD VMwareHost element

As seen, a <VMwareHost> contains the password and username for the host, and an element called <vmList>, which is a list one or more <VM> elements. Each <VM> element represents a guest OS running Windows XP, and has the following elements

```
<xsd:element name="VM">
<xsd:complexType mixed="true">
  <xsd:sequence>
    <xsd:element maxOccurs="1" minOccurs="1" ref="vmUsername"/>
    <xsd:element maxOccurs="1" minOccurs="1" ref="vmPassword"/>
    <xsd:element maxOccurs="1" minOccurs="0" ref="avEngine"/>
    <xsd:element maxOccurs="1" minOccurs="0" ref="analysisTools"/>
  </xsd:sequence>
  <xsd:attribute ref="vmxPath" use="required"/>
</xsd:complexType>
</xsd:element>
```

Listing D.6: XSD VM element

The last two elements of the <VM> shows that a guest OS can run one AV engine, or one or more analysis tools. An AV engine has the following configurable properties;

```
<xsd:element name="avEngine">
  <xsd:complexType mixed="true">
    <xsd:sequence>
      <xsd:element maxOccurs="1" minOccurs="1" ref="avExecutablePath"/>
      <xsd:element maxOccurs="1" minOccurs="1" ref="avParameters"/>
      <xsd:element maxOccurs="1" minOccurs="1" ref="avLogFilePath"/>
      <xsd:element maxOccurs="1" minOccurs="0" ref="avLogFilter"/>
      <xsd:element maxOccurs="1" minOccurs="0" ref="avUpdateInfo"/>
      <xsd:element maxOccurs="1" minOccurs="0" ref="realTimeScan"/>
```

```
    </xsd:sequence>
    <xsd:attribute ref="name" use="required"/>
  </xsd:complexType>
</xsd:element>
```

Listing D.7: XSD avEngine element

The AV engines are used for on-demand scan, and the first two parameters are used to configure the command used to initiate the scan. The third parameter, <avLogFilePath>, is the path to the log or result file in which the result of the on-demand scan is written. This may be a standard log file that the AV engine uses, or the console output piped to a temporary file. The <avLogFilter> element is used to filter the result file looking for specific words that indicate a malware hit. It is defined as follows:

```
<xsd:element name="avLogFilter">
  <xsd:complexType mixed="true">
    <xsd:sequence>
      <xsd:element maxOccurs="1" minOccurs="1" ref="avResultIdentifier"/>
      <xsd:element maxOccurs="1" minOccurs="0" ref="avResultPrefix"/>
      <xsd:element maxOccurs="1" minOccurs="0" ref="avResultSuffix"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Listing D.8: XSD avLogFilter element

At regular intervals, the AV engines need to update their definition/signature database. This is hard to schedule using automatic updaters, as the guest OSs are reverted to snapshot at unpredictable intervals (typically after each use). To deal with this, PowerScan supports automatic update of all registered engines. The <avUpdateInfo> element contains path to the update executable and parameters, a path to a log file and some word or words to look for in the log that indicate a successful update operation. Note that this element is optional, as some engines may not support automatic updates via the command line.

```
<xsd:element name="avUpdateInfo">
  <xsd:complexType mixed="true">
    <xsd:sequence>
      <xsd:element maxOccurs="1" minOccurs="0" ref="avUpdateExecutable"/>
      <xsd:element maxOccurs="1" minOccurs="0" ref="avUpdateParameters"/>
      <xsd:element maxOccurs="1" minOccurs="0" ref="avUpdateLogPath"/>
      <xsd:element maxOccurs="1" minOccurs="0" ref="avUpdateSuccessIndicator
          "/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Listing D.9: XSD avUpdateInfo element

The <realTimeScan> element of the <avEngine> is, as the name indicates, configuration parameters for the use of real-time scan functionality. The malware sample is executed within the guest OS, and the real time scan log is parsed to look for indicators showing the the executable has been recognized as containing malware. Although real-time scanner alerts often are shown in GUI pop-ups, PowerScan relies on parsing the log file of the anti-virus solution in order to determine if an infection was noted. This is because the Vix framework does not support GUI interaction.

In addition to scanning the malware sample with an AV scanner, it is also possible to set up guest OSs running one or more analysis tools. These tools are often executed with the malware sample as a parameter, but in some cases the analysis tool is started before the sample is executed. The <VM> element contains an <analysisTools> element, which is a list containing one or more <dynamicAnalysisTool> elements. The <dynamicAnalysisTool> element contains the configuration parameters for an analysis tool, and is defined as

```
<xsd:element name="dynamicAnalysisTool">
<xsd:complexType mixed="true">
  <xsd:sequence>
    <xsd:element maxOccurs="1" minOccurs="1" ref="toolExecutablePath"/>
    <xsd:element maxOccurs="1" minOccurs="0" ref="toolParameters"/>
    <xsd:element maxOccurs="1" minOccurs="0" ref="toolResultFilePath"/>
    <xsd:element maxOccurs="1" minOccurs="0" ref="executeMalwareExplicitly"/
        >
  </xsd:sequence>
  <xsd:attribute ref="toolName" use="required"/>
</xsd:complexType>
</xsd:element>
```

Listing D.10: XSD dynamicAnalysisTool element

This element contains a path to start the tool, a string that may contain parameters, a log file path to be able to copy back the result and an element called <executeMalwareExplicitly> which is used to indicate whether the malware should be executed explicitly, or if it is supplied as a part of the parameter string. If the malware path needs to be included as part of the parameters, it can be inserted using the string $malwarePath.

An example XML config file can be found in section D.1, and the appurtenant XSD file in D.2.

D.4 Properties file example

```
## VMware host setup
## Note that all times are given in seconds.

## Virtual machine setup
# Specify timeout for waiting for tools to start in Virtual
# Machines (tools is required for operation)
vmware.tools.timeout = 30

## Scanner parameters
# Specify the path on the virtual machines in which to execute the malware
    sample.
# Note that backslashes (\) will have to escaped by another backslash, such
# that c:\program files\ becomes c:\\program files\\.
# Also note that the directory must already be created on the virtual
    machine,
# as VIX 1.0 and 1.1 does not support directory creation.
# The execution path must end with a file separator (typically slash (/) or
    backslash (\)).
scanner.executionpath = c:\\

# Specify whether snapshots should be taken before performing scans.
# This is generally NOT recommended, as storing snapshots is a time
    consuming
# operation, which should only be performed when changes are made to the
    Virtual Machine.
# It may also lead to instabilities in the virtual machines and cause scan
    failures.
snapshot.before.scan = false

# Set polling interval for minor operations
polling.interval.minor = 0.250

# Set polling interval for major operations
polling.interval.major = 1.000

# Maximum time to allow for individual scan operations to finish.
scanner.timeout = 60

# Maximum time to allow malware to execute with real-time anti-virus scanner
     running
malware.execution.timeout = 25

# Maximum total time for malware execution operation (for all registered
    scanners)
full.execution.timeout = 240

# Maximum time to allow for updating to finish
update.timeout = 600

# Timeout for full scan. Used to prevent entire system from crashing
# following failure in a single thread.
full.scan.timeout = 600

# Determine whether log files should be overwritten when program is executed
    .
# Note that log files may take up significant resources over time.
log.overwrite = false

## Log files
```

```
# VMware calls
log.vmware = logs\\vmware.log

## Scanner log (scan logic)
log.system.scanner = logs\\scanner.log

## System log (main thread)
log.system = logs\\system.log

## Executor log
log.system.executor = logs\\executor.log

## Executor path to location of malware sample
# Note that backslashes (\) will have to escaped by another backslash,
# such that c:\program files\ becomes c:\\program files\\.
# Also note that the directory must already be created on the virtual
# machine, as VIX 1.0 and 1.1 does not support directory creation.
# The execution path must end with a file separator (typically slash (/) or
    backslash (\)).
executor.executionpath = c:\\

# Path to which the result files from the analysis tools should be copied
# on the local system
executor.localResultPath = c:\\test\\

# Sets the time in seconds for which the operator is allowed to interact
# with the malware sample/analysis tools
executor.overall.timeout = 30

# Sets the path for the XSD file used to validate the XML config file
xml.xsd.path = config.xsd
```

Listing D.11: Example PowerScan properties file

Appendix E

Test case specifications

> "The test is to recognize the mistake, admit it and correct it. To have tried to do something and failed is vastly better than to have tried to do nothing and succeeded."

- Dale E. Turner

E.1 System test using command line interface

Test specification for	System test
Module	system
Responsible for test specification	Thomas Langerud
Date	19 May 2008
Estimated hours to carry out test	
Objective of test	Test the complete system, using the command line interface
Test sequence number	1
Comments	This tests basic functionality using the command line interface. The special cases are tested using the GUI (sequence 2).

Test carried out by	Thomas Langerud
Date	23 May 2008
Build	
Configuration file version	1.30
Properties file version	1.18
Comments	

1.1 Test the scan functionality with multiple virtual machines on one host			
Prerequisites	System set up with Windows XP installed on virtual machines		
[sequence number]	[action]	[expected result]	[result, error no.]
1.1.1	Execute the system using the following command switches: "-s config/powerscan.properties –c config/config.xml –SCAN c:\test\eicar.com"	The system executes the scan operation and the following status messages should be printed to the console: - *Validating XML config* - *Connected to VMware Host [host]* - *Performing scan on [sample] using X scan engines.* - *[anti-virus engine] starting scan.* The result of the scan operation should be printed. - *Successfully reverted virtual machine to snapshot for [anti-virus engine]*	OK

1.2 Test the update functionality			
Prerequisites	System set up with Windows XP installed on virtual machines		
[sequence number]	[action]	[expected result]	[result, error no.]
1.2.1	Execute the system using the following command switches: "-s config/powerscan.properties –c config/config.xml –UPDATE"	The system executes the update operation, and the following status messages should be printed to the console: - *Validating XML config.* - *Connected to VMware Host [host].* - *[anti-virus engine] starting update operation.* - *Starting update of virus definitions for [anti-virus engine].* - *Automatic snapshot taken for VM running [anti-virus engine].* The update result should be printed. The results will depend on the command line capabilities of the anti-virus engine. The following is an example: - *Successfully updated engines:* *clamwin* *Database updated (295835 signatures) from database.clamav.net (IP: 62.236.254.228)* *Engines that does not provide verbose update log and should be reviewed manually:* *F-Secure 7.10 running at [host] with OS login/pass*	OK

		[username]/[password]. *Please login to these virtual machines with the provided usernames/passwords and assure that they are updated and ready to perform scans. Remember to take snapshots if any further changes are made.*	

1.3 Test the execute malware functionality			
Prerequisites	System set up with Windows XP installed on virtual machines		
[sequence number]	[action]	[expected result]	[result, error no.]
1.3.1	Execute the system using the following command switches: "-s config/powerscan.properties -c config/config_with_new_test.xml -EXECUTE c:\test\eicar.com"	The system executes the execute malware operation and the following status messages should be printed to the console: - *Validating XML config* - *Connected to VMware Host [host].* - *Executing [sample] on X virtual machines.* - *[anti-virus engine] starting execution of malware sample at [sample].* The result of the execute malware operation should then be printed. - *Reverted virtual machine running [anti-virus engine] to snapshot.*	OK

1.4 Test the dynamic analysis functionality, several tools in several VMs			
Prerequisites	System set up with Windows XP installed on virtual machines		
[sequence number]	[action]	[expected result]	[result, error no.]
1.4.1	Execute the system using the following command switches: "-s config/powerscan.properties -c config/config_with_new_test.xml -ANALYZE c:\test\eicar.com	The system executes the dynamic analysis operation and the following status messages should be printed to the console: - *Validating XML config* - *Connected to VMware Host [host].* - *Executing dynamic analysis on file [sample].* - *Copying sample file to VM [host]:[vmx path]* - *Done copying sample file to guest OS.* - *Tool [tool] is now running on VM [host]:[vmx path].* - *All analysis tools have been started on VM [host]:[vmx path], user interaction might be required. Log in to VM using username/password [username]/[password]* - *Now sleeping - awaiting user interaction.* Execution will now sleep for a configurable time period, and the console will show countdown as: - *Slept X seconds of total Y* each 10 seconds - *Done sleeping - completing task.* If a result file is given, the file should be copied back, showing the following status message: - *Copying result file back for tool [tool]* The sequence should then be repeated for the next VMs, starting from "Copying sample file…" After the last VM has completed copying back the result, the VMs should be reverted to snapshot. - *Reverting VM with handle [handle] on host [host] to snapshot* The result of the operation should then be printed. Depending on whether the tool creates a log file, the result may look like: - *Tool [tool] completed. No result file is declared.* *or* - *Tool [tool] completed, result file is located at [local*	OK

		result file path].	
1.4.2	Redo sequence 1.4.1, cancel the sleep period before it completes	The sleep period should be interrupted, and the system should proceed with the next action.	OK

E.2 System test using graphical user interface

Test specification for	System test
Module	system
Responsible for test specification	Thomas Langerud
Date	19 May 2008
Estimated hours to carry out test	
Objective of test	Test the complete system, using the graphical user interface
Test sequence number	2
Comments	The scan test should be carried out using the EICAR test file.

Test carried out by	Thomas Langerud
Date	23 May 2008
Build	[ ]
Configuration file version	1.30
Properties file version	1.18
Comments	

2.1 Test the initialization functionality with one host			
Prerequisites	System set up with Windows XP installed on virtual machines		
[sequence number]	[action]	[expected result]	[result, error no.]
2.1.1	Start PowerScan with the –GUI switch.	The GUI is shown, the "Open" and "Clear" buttons and the three input fields are available. The Start, Edit, Output and Help menus are available. The console text area displays *Now ready*.	OK
2.1.2	Insert path to valid XML config file and properties file, choose Start from the System menu.	The following status messages should be printed: - *Validating XML config* - *Connected to VMware Host [host]* No error messages should appear, except for message about virtual machines running both anti-virus engine and tools. - *Now ready* The input fields for config and properties, and their corresponding buttons, should now be unavailable, as should the Start option on the System menu.	OK

2.2 Test the initialization functionality with multiple host			
Prerequisites			
[sequence number]	[action]	[expected result]	[result, error no.]
2.2.1	Start PowerScan with the –GUI switch	The GUI is shown, the "Open" and "Clear" buttons and the three input fields are available. The Start, Edit, Output and Help menus are available. The console text area displays - *Now ready*.	OK
2.2.2	Insert path to valid XML config file and properties file, choose Start from the System menu	The following status messages should be printed: - *Validating XML config* - *Connected to VMware Host [host]* should appear once for each host in the config file No error messages should appear, except for message about virtual machines running both anti-virus engine and tools. - *Now ready* . The input fields for config and properties files, and their corresponding buttons, should now be unavailable, as should the Start option on the System menu.	OK
2.2.3	Redo sequence 2.2.2	The following text should be printed to the console text area:	OK

	using an XML config file which does not contain all required elements or attributes.	- *Warning: XML validation failed: [error message]*	
2.2.4	Redo sequence 2.2.2 using an XML config file which does not contain correct information in the 'host' attribute.	If only one host in the config file, the following message should appear: - *Error: No scanners were loaded from config* The system should then return to initial state. If more than one host in the config file, the following message should appear: - *Warning: unable to connect to VMware host at [incorrect hostname]. Proceeding without given host.* The system should then return to ready state.	OK [Failed: if the remaining host contains only VMs with analysis tools, the system would not run. Status: solved]

2.3 Test the scan functionality with multiple virtual machines running on one host			
Prerequisites	Sequence 2.2		
[sequence number]	[action]	[expected result]	[result, error no.]
2.3.1	Input path to the eicar test file on the local system. Press the Scan button in the GUI.	The console text area should show the following messages: - *Initiating scan* - *Performing scan on [sample] using X scan engines.* - *[anti-virus engine] starting scan.* The result of the scan operation should be printed. - *Successfully reverted virtual machine to snapshot for [anti-virus engine]*	OK
2.3.2	Redo sequence 2.3.1 with an XML file where the <avLogFilter> element is missing from one or more of the VM	As for sequence 2.3.1, but the result should be that the entire log file is printed for the anti-virus engine that is missing the <avLogFilter> element.	[Failed: a nullPointerException is thrown when the element is missing. Status: solved]

2.4 Test the malware execution functionality			
Prerequisites	Sequence 2.2		
[sequence number]	[action]	[expected result]	[result, error no.]
2.4.1	Input the path to the eicar test file on the local system. Press the Execute button in the GUI.	The console text area should show the following messages: - *Malware execution is requested* - *Executing [sample] on X virtual machines.* - *[anti-virus engine] starting execution of malware sample at [sample]* The result of the execute operation should be printed, e.g.: - *[anti-virus engine] running on [host] reported: No infection detected.* For engines that does not have an on-access scanner, or does not produce a human—readable log file, the following message should be printed: - *[anti-virus engine] running on [host] reported: Real-time scanning not supported.* - *Reverted virtual machine running [anti-virus engine] to snapshot.*	OK [Failed: the parsing of log files seems to be a problem due to character encoding issues. Status: pending]
2.4.2	Redo sequence 2.4.1 with an XML file containing the <realTimeScan> element without any data. The system will then	The console text area should show the following messages: - *Malware execution is requested* - *Executing [sample] on X virtual machines.* - *[anti-virus engine] starting execution of malware sample at [sample]*	OK

	attempt to execute the malware on the VM, but will not be able to copy the log file back to the local system.	The AV engines with incorrect <realTimeScan> element will cause the following warning message: - *Warning: Copy of malware sample to or result log file from guest OS for virtual machine running [anti-virus engine]. A common reason for this error is that the scan operation failed to write the log output to file (i.e. the file was not created). Error: One of the parameters was invalid* The result of the execute operation should be printed, e.g.: - *[anti-virus engine] running on [host] reported:* *No infection detected.* For the AV engines with incorrect <realTimeScan> element, the following result line should be printed: - [*av engine] running on [host] reported:* *Malware execution failed.*	

2.5 Test the dynamic analysis functionality with tools on one virtual machine.			
Prerequisites	Sequence 2.2.		
[sequence number]	[action]	[expected result]	[result, error no.]
2.5.1	Input path to the test file in the malware path field. Press the Dyn. Analyze button.	As the operation executes, the console text area should display the following status messages: - *Initiating dynamic analysis.* - *Executing dynamic analysis on file [sample].* - *Copying sample file to VM [host]:[vmx path]* - *Done copying sample file to VM.* - *Tool [tool] is now running on VM [host]:[vmx path]* If the malware sample is indicated to be executed explicitly, the following messages should be printed: - *Malware sample now executed.* When all tools have been stated, the status messages should state: - *All analysis tool have been started on VM [host]:[vmx path], user interaction might be required. Log in to VM using username/password [username]/[password]* - *Now sleeping - awaiting user interaction* At this time, a dialog box counting down the specified sleep period should pop up. Once the countdown is completed: - *Done sleeping - completing task* The result files (if indicated in config) are now copied back. Status messages should state: - *Copying result file back for tool [tool]* The VMs are now supposed to be reverted to snapshot, status message should say: - *Reverting VM with handle [vm handle] on host [handle] to snapshot* The results of the analysis tool operation should be shown. Dependent on whether the tool has a log file that is copied back, the result may look like: - *Tool [tool] completed* or - *Tool [tool] completed, result file is located at [local result file path.]*	OK
2.5.2	Redo sequence 2.5.1 using an XML file with analysis tool element	When trying to run an executable which does not exist, the following error message should be printed: - *Warning: Unable to execute analysis tool on remote*	[Failed: when one tool did not execute properly, the other tools on the same VM were not started.

	containing an erroneous path to the tool.	*system*	Status: solved]
2.5.3	Redo sequence 2.5.1, but press Cancel as the sleep time counter dialog box is showing.	- The sleep should be interrupted, and the system should continue with the next operation.	OK

2.6 Test the dynamic analysis functionality with tools on several virtual machines.			
Prerequisites	Sequence 2.2		
[sequence number]	[action]	[expected result]	[result, error no.]
2.6.1	Using an XML file with several analysis tools on more than one VM, start the analysis operation by pressing the Dyn. Analyze button.	The following status messages should be displayed in console: - *Initiating dynamic analysis.* - *Executing dynamic analysis on file [sample].* - *Copying sample file to VM [host]:[vmx path]* - *Done copying sample file to VM.* - *Tool [tool] is now running on VM [host]:[vmx path]* The following should be printed once for each tool: - *Tool [tool] is now running on VM [host]:[vmx path]* After all tools have been started: - *All analysis tool have been started on VM [host]:[vmx path], user interaction might be required. Log in to VM using username/password [username]/[password]* - *Now sleeping - awaiting user interaction* The progress bar should then be shown, counting up to the indicated sleep time. The result files (if indicated in config) are now copied back. Status messages should state: - *Copying result file back for tool [tool]* The same sequence of status messages, starting from "Copying sample path…", should then be printed again for the next virtual machine. When the last VM has completed execution and the result files are copied back, all the VMs should be reverted to snapshot. The results should then be printed. - *Tool [tool] completed. No result file is declared.* *Or* - *Tool [tool] completed, result file is located at [local result file path].*	OK

2.7 Test the antivirus definition update functionality			
Prerequisites	Sequence 2.2		
[sequence number]	[action]	[expected result]	[result, error no.]
2.7.1	Press the Update button in the GUI.	The following status messages should be displayed in console: - *[anti-virus engine] starting update operation.* - *Starting update of virus definitions for [anti-virus engine].* - *Automatic snapshot taken for VM running [anti-virus engine].* The result of the update operation should be printed. If any of the engines do not have parsable log or output, the following text should appear: - *Engines that does not provide verbose update log and should be reviewed manually:* *[anti-virus engine] running at [hostname] with OS login/pass [username]/[password].*	OK

		Please login to these virtual machines with the provided usernames/passwords and assure that they are updated and ready to perform scans. Remember to take snapshots if any further changes are made.	
2.7.2	Redo sequence 2.7.1 with an XML file where one or more AV engines does not have any <avUpdateInfo> element.	The following status messages should be printed: - [anti-virus engine] starting update operation. For those anti-virus engines that have an <avUpdateInfo> element, the following should be printed: - *Starting update of virus definitions [anti-virus engine]* - *Automatic snapshot taken for VM running [anti-virus engine]*. For those anti-virus engines that do not have an <avUpdateInfo> element, the following should be printed: - *Automatic update not supported for [anti-virus engine]*. The results of the update operations should then be printed. For anti-virus engines that do not have a <avUpdateLogPath> element, the result should be: - *Engines that does not provide verbose update log and should be reviewed manually:* *[anti-virus engine] running at [host] with OS login/pass [username]/[password]*. *Please login to these virtual machines with the provided usernames/passwords and assure that they are updated and ready to perform scans. Remember to take snapshots if any further changes are made.* For anti-virus engines that do not have a <avUpdateInfo> element at all, the result should be: - *The following engines does not support automatic updates and must be updated manually:* *[anti-virus engine] running at [host] with OS login/pass [username]/[password]*. *Please login to these virtual machines with the provided usernames/passwords and perform manual update. Remember to take snapshot after updating.*	OK

E.3 System test of the configuration editor

Test specification for	System test
Module	GUI Configuration editor
Responsible for test specification	Thomas Langerud
Date	19 may 2008
Estimated hours to carry out test	2
Objective of test	Test the functionality of the GUI configuration editor.
Test sequence number	3
Comments	

Test carried out by	Thomas Langerud
Date	27 May 2008
Build	
Configuration file version	
Properties file version	
Comments	

3.1 View/edit existing XML file			
Prerequisites	The PowerScan system is started with the –GUI switch.		
[sequence number]	[action]	[expected result]	[result, error no.]
3.1.1	In the GUI, select the 'Config Editor' choice from the 'Edit' menu.	The configuration editor opens in a new window.	OK
3.1.2	Open an existing XML configuration file by choosing 'Open config' from the 'File' menu.	The host(s) defined in the config file is shown in the upper half of the window, while the virtual machines associated with a host is shown in the lower half.	OK
3.1.3	Choose a random virtual machine.	- Observe that the 'Edit AV engine' button is shown if the VM has an anti-virus engine associated with it, or the 'Add AV engine' if it does not have one. - Observe that the 'Edit analysis tools' button is shown the VM has tools associated with it, or the 'Add analysis tools' of it does not have any.	OK
3.1.4	Select the 'Delete' menu	- Observe that the 'Delete active host' option is enabled. - Observe that the 'Delete active VM' option is enabled. - Observe that the 'Delete active AVE' and 'Delete active tool' options are disabled.	OK
3.1.5	Select the 'View' menu	- Observe that the 'Host/VM view' option is disabled.	OK
3.1.6	Choose a VM with an associated anti-virus engine, and press the 'Edit AV engine' button	The Edit AVE view is shown, listing all information about a given anti-virus engine. Observe the following changes to the menu: - In the 'View' menu, the 'Host/VM view' choice in now enabled. - In the 'Delete' menu, the 'Delete active AVE' choice is now enabled.	OK
3.1.7	From the 'View' menu, choose 'Host/VM view'.	The config editor returns to the previous view, showing the hosts and virtual machines.	OK
3.1.8	- Redo sequence 3.1.6. - From the 'Delete' menu, select the 'Delete active AVE' choice	The following should happen: - The config editor returns to the Host/VM view. - The 'Edit AV engine' button is now replaced with the 'Add AV engine' button.	OK
3.1.9	Choose a VM with one or more associated analysis tools and press the 'Edit analysis tools' button	The Edit tools view is show, listing all information about a given analysis tool, and when there are more than one tool it is possible to switch between the different tools using tabs at the top of the page. Observe the following changes to the menu: - In the 'View' menu, the 'Host/VM view' choice in now enabled. - In the 'Delete' menu, the 'Delete active tool' choice is	OK

		now enabled.	
3.1.10	From the ‘View’ menu, choose ‘Host/VM view’.	The config editor returns to the previous view, showing the hosts and virtual machines.	OK
3.1.11	- Redo sequence 3.1.9 - From the ‘Delete’ menu, select the ‘Delete active tool’ choice.	If more than one tool are shown in the Edit tools view, the following should happen: - The config editor returns to the Host/VM view. If only one tool is shown, the following should happen: - The config editor returns to the Host/VM view. - The ‘Edit tools engine’ button is now replaced with the ‘Add AV engine’ button.	OK
3.1.12	- Redo sequence 3.1.9 - Press the ‘Add tool’ button. Return to the Host/VM view, and then select ‘Edit analysis tools’ on the same VM	Observe that a new tab with the title ‘New tool’ appears. Observe that the new tool is now available as a tab with the name ‘Tool #’	OK

3.2 Add new host			
Prerequisites	The config editor launched, a config file is opened and the editor is in Host/VM view.		
[sequence number]	[action]	[expected result]	[result, error no.]
3.2.1	Press the ‘Add Host’ button.	Observe that a new host tab is added at the top of the page.	OK
3.2.2	Select the newly added host tab.	Observe that a new VM is added associated with the new host.	OK
3.2.3	Save the file selecting the ‘Save config’ option on the ‘File’ menu.	Review the saved XML file and verify that the new host and VM is present.	OK

3.3 Add new virtual machine			
Prerequisites	The config editor launched, a config file is opened and the editor is in Host/VM view.		
[sequence number]	[action]	[expected result]	[result, error no.]
3.3.1	Press the ‘Add Virtual Machine’ button.	Observe that a new host tab is added to the bottom half of the window.	OK
3.3.2	Select another host, and the select the host in which the new VM was just added.	Observe that the newly added VM tab is still present.	OK
3.3.3	Save the file selecting the ‘Save config’ option on the ‘File’ menu.	Review the saved XML file and verify that the new VM is present.	OK

3.4 Add new anti-virus engine to a virtual machine			
Prerequisites	The config editor launched, a config file is opened and the editor is in Host/VM view.		
[sequence number]	[action]	[expected result]	[result, error no.]
3.4.1	Select a VM which does not have an anti-virus engine associated with it. Press the ‘Add AV engine’ button. Enter text into some of the fields.	A new, empty Edit AVE view window is shown.	OK
3.4.2	Return to the Host/VM view selecting the ‘Host/VM view’ from the ‘View’ menu.	Observe that the ‘Add AV engine’ button is now replaced by the ‘Edit AV engine’ button.	OK
3.4.3	Press the ‘Edit AV engine’ button.	Observe that the previously entered text is still there.	OK
3.4.4	Save the config file by selecting the ‘Save config’ option on the ‘File’ menu.	Review the saved XML file and verify that the newly added anti-virus engine information is present.	OK

3.5 Add new analysis tools to a virtual machine			
Prerequisites			
[sequence number]	[action]	[expected result]	[result, error no.]
3.5.1	Select a VM which does not have an analysis tools associated with it. Press the 'Add analysis tools' button. Enter text into some of the fields.	A new, empty Edit tools view window is shown.	OK
3.5.2	Return to the Host/VM view selecting the 'Host/VM view' from the 'View' menu.	Observe that the 'Add analysis tool' button is now replaced by the 'Edit analysis tools' button.	OK
3.5.3	Press the 'Edit analysis tools' button.	Observe that the previously entered text is still there.	OK
3.5.4	Save the config file by selecting the 'Save config' option on the 'File' menu.	Review the saved XML file and verify that the newly added analysis tool information is present.	OK

3.6 Create new XML file from scratch			
Prerequisites	The config editor is launched, no file is open.		
[sequence number]	[action]	[expected result]	[result, error no.]
3.6.1	Select 'New config' from the 'File' menu.	A new, empty host with an empty VM is shown in the Host/VM view.	OK
3.6.2	Select 'Save config' from the 'File' menu.	Review the saved XML file and verify that the newly added information is present.	OK