Riku Peltonen

Automated Testing of Detection and Remediation of Malicious Software

Helsinki Metropolia University of Applied Sciences

Master of Engineering

Information Technology

Master's Thesis

22 November 2017

Helsinki Metropolia University of Applied Sciences

Author Title Number of Pages Date	Riku Peltonen Automated Testing of Detection and Remediation of Malicious Software 71 pages 22nd November 2017
Degree	Master of Engineering
Degree Programme	Information Technology
Instructor	Ville Jääskeläinen, Principal Lecturer

In recent years, automation has become commonplace in the field of malware analysis and development of anti-malware software. Automated testing of anti-malware software relies heavily on simulated malware samples and infections, due to the security considerations related to handling real malware. However, using simulated malware does not produce realistic data about the actual protection capabilities of the anti-malware software when facing real threats in real end-user environments.

The aim of this thesis was to design and implement a system for automated testing of anti-malware products and technologies against real malicious software. It combined the common methodologies of automated software testing with the special considerations related to handling and analysis of malware.

The thesis starts with background research in the domain of malware and anti-malware, describing the common methods malicious software uses to infect computer systems, and how anti-malware software attempts to counteract them. Further information is provided on how to automate the testing of different anti-malware techniques and features against different malware infection scenarios.

A suitable architecture for the system, and the technologies used to implement it, are drafted and evaluated, followed by detailed steps on how each functional part of the system was implemented. The automated tests and their coverage are described in detail, including how malware is used, detected and remediated in the test environment.

Keywords	malware, anti-malware, software testing, test automation

Table of Contents

List of Abbreviations

AoE	ATA over Ethernet
API	Application Programming Interface
APT	Advanced Packaging Tool
AR	Action Research
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
EICAR	European Institute for Computer Antivirus Research
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
KVM	Kernel-based Virtual Machine
MIT	Massachusetts Institute of Technology
MPoE	Message Passing over Ethernet
MSF	Metasploit Framework
NTP	Network Time Protocol
OAS	On-access Scanning
ODS	On-demand Scanning
SCP	Secure Copy
SSH	Secure Shell
PDF	Portable Document Format
PE	Portable Executable
PyPI	Python Package Index
QEMU	Quick Emulator
RPC	Remote Procedure Call
SHA	Secure Hash Algorithm
SSL	Secure Sockets Layer
TA	Test Automation
TLS	Transport Layer Security
UAC	User Access Control
WMI	Windows Management Instrumentation
WTS	Web-traffic Scanning
XML	Extensible Markup Language

1 Introduction

Testing of modern anti-malware software is predominantly automated. This is primarily due to the speed and volume provided by automated testing, which are highly beneficial in the context of anti-malware testing. Also executing and verifying anti-malware operations is possible and generally preferred to be done programmatically, as verifying changes in the computer memory and file system is not an operation that requires specific human interaction or judgement.

Automated anti-malware testing is commonly performed against both real malware samples and specifically crafted test samples, which are identified as malware but are not actually malicious and contain no extensive logic. Using test samples is convenient in that they are safe to use in regular test- and network environments and provide some rudimentary feedback about the functionality of anti-malware software. However they do not provide extensive code coverage in the targeted protection features and technologies, so the testing cannot entirely rely on them. The added code coverage that using real malware samples instead of test samples provides resides in functional areas such as file parsers, unpackers and disinfectors.

The bulk of anti-malware testing that is performed against real malware targets signature-based detections, by scanning large sets of files in specific high-volume environments, with the goal of identifying which files are detected as malicious and which are not. Automated functional testing of the actual client-side protection and remediation features using real malware is rarer, and is commonly done with test samples. This is because producing and verifying real malware infections adds a lot of additional requirements for the test environment and infrastructure. The malware handling needs to be automated but secure, and the test environment, or sandbox, needs to be able to use and execute malware without allowing it to escape to other computers and networks. Also verifying anti-malware operations against real malware can be tricky, as unlike test samples they do not always behave in a predictable and expected manner, unless they have been thoroughly reverse-engineered.

Automated functional testing against real malware is necessary for ensuring and improving the effectiveness of anti-malware software, but there are no comprehensive generic solutions available for it, so every anti-malware vendor and security research organization needs to implement their own. This study aims to design and implement a solution that satisfies the basic requirements for such a system, that is how such a system should be designed so that it is secure, accessible and efficient, and produces useful data about the functionality and effectiveness of anti-malware software. The intended outcome of the study is a functional test automation framework and supporting end-to-end infrastructure for testing the detection and remediation of malicious software, using real malware samples.

This study was commissioned by an anti-malware vendor, with the intent of using the system for anticipating and improving the scores of their software in malware protection tests performed by third-party security software review organizations, such as AV-TEST [1] and AV-Comparatives [2]. These scores are often referred to in online- and magazine reviews, and can affect the public perception and sales of anti-malware software significantly.

The solution of this study involves both research and development. The research focuses on the classification and behavior of malware, with the goal of identifying what kind of malware is currently prevalent and how the system should utilize it in a way that is secure and predicable, but also realistic. The study also introduces some common anti-malware techniques and technologies, how they work and how they can be tested in an automatic fashion. The body of knowledge for the malware and anti-malware research will come both from existing literature sources and consultation with expert malware analysts.

The development phase aimed to implement a fully functional test automation framework and supporting infrastructure, and to integrate it with existing systems and continuous integration flows. The development effort consisted of three areas: the virtual sandbox, the end-to-end infrastructure and the client-side test automation. Some parts of the system, for example the physical network environment, the underlying virtualization platform and the malware sample storage, were provided as existing services and were not implemented or configured in the scope of this study. However their basic layout, functionality and usage is explained in some detail.

The study also explores different technology choices, and weighs their functionality and suitability for implementing such a system. This includes the programming- or scripting language for implementing the client-side test automation and the infrastructure automation, the continuous integration system for implementing and hosting the test jobs, and other possible technologies for supporting and tooling functionality.

As the majority of prevalent in-the-wild malware targets the Windows operating system, it is also the main focus platform of this study. However Linux and Apple OS X -based systems were also employed, with coverage matching the lower number of anti-malware features and threats present on those platforms.

The system produces different types of data, which can be used for evaluation and assurance of the quality of anti-malware software. This includes but is not limited to: data about malware detection rates, the performance of different scanning operations, and memory dumps with crash information. The system is not meant for malware analysis as such, and the malware samples used in it are generally already known and analyzed, with available detections. However, one could theoretically drop any malware sample into the system and observe how the anti-malware software interacts with it.

1.1 Research Design

This study starts with a brief introduction into the world of malware and anti-malware; what it means, why it is done and how it works in practice. It categorizes malware in broad terms, introduce some common anti-malware techniques, and explain different ways computers can get infected and how anti-malware tries to counteract them. It then evaluates how the different malware detection and remediation scenarios could be automated, in practical and technical terms.

The following chapters draft out the proposed architecture and infrastructure of the system, including the virtual sandbox, the test job flow, and the backend automation for handling malware and other artifacts in the system.

Next the study describes the actual client-side test automation implementation; what kind of test cases are supported, how they were executed, how the results can be verified, processed and presented, and how the data produced by the system can be utilized in practice.

The last chapter of the report describes how the system was tested, evaluated and re-iterated to reach the final state and acceptance, followed by conclusions and lessons learned during the study. Figure 1 illustrates the research design of the study.

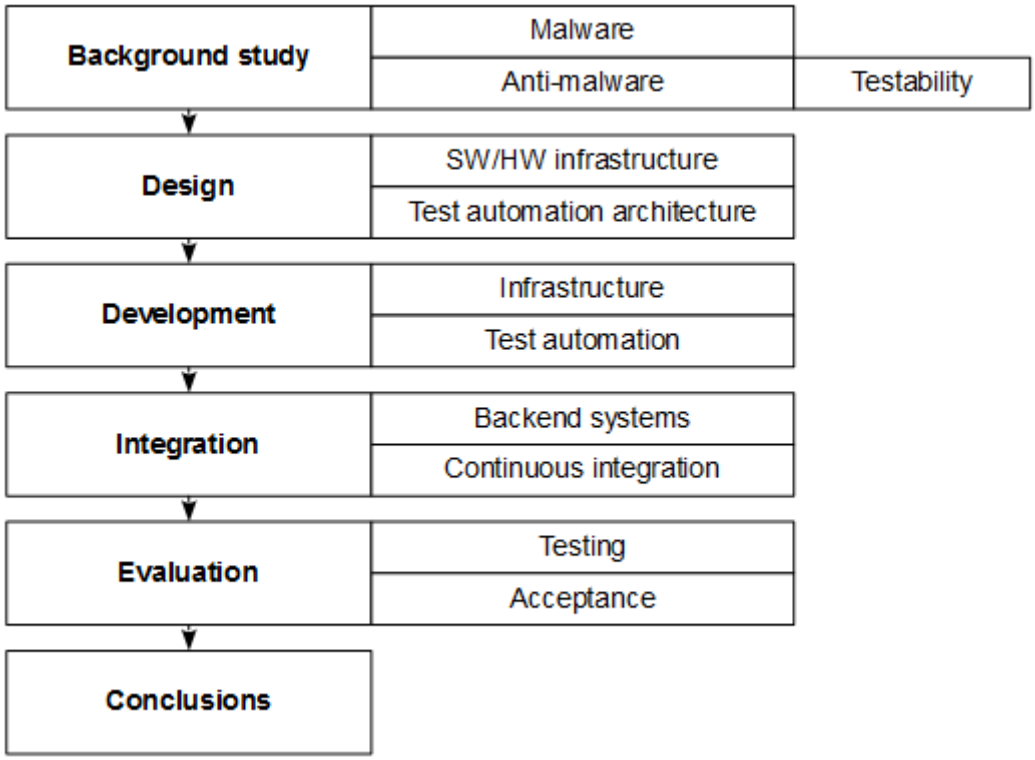

Figure 1. Research Design.

Figure 1 describes the main sequential phases of the study, and the major items researched or implemented in each phase.

2 Research Methodology

The research method of this study relates closely to Action Research (AR). It is an empirical research method that attempts to solve a real-world problem in collaboration with the “owner” of the problem, which in the case of this thesis is the commissioner organization.

The Action Research methodology was originally introduced by Kurt Lewin (1890-1947), from the notion of a researcher becoming immersed in a real-world situation and following it through to wherever it may lead. [3]

The Action Research method is common among research projects relating to software engineering, due to the iterative nature of software development (see Figure 2). It also suits to this study, which tried to solve a real tangible problem and did not deal in experimental research as such. Also, following the AR methodology, this study was performed in the same environment in which the results, that is the completed system, will be applied and used.

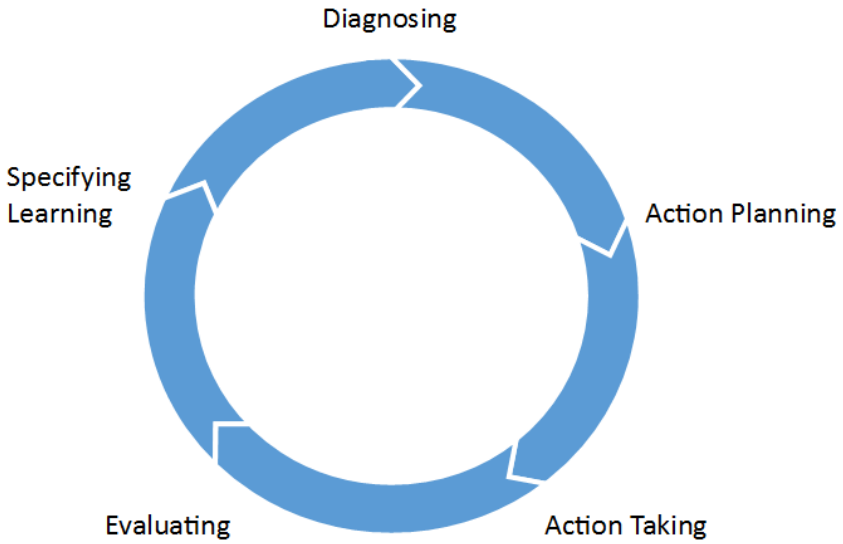

Figure 2. Action Research Cycle. [4]

The research was conducted in the following sequential phases:

1. Collect requirements from stakeholders.
2. Review viable technologies for implementing the required functionality, with emphasis on technologies that are already utilized in the commissioner organization.
3. Design software architecture, and how to utilize existing hardware architecture.
4. Develop solution.
5. Test and evaluate solution, re-iterate until acceptable.

The following chapters describe the methodology of the requirement collection for the different phases of the study.

2.1 Requirements Overview

As this study aimed to implement a fully functional system, not just a design, it needs to fulfill several technical and operational requirements. These requirements were derived from a combination of industry standard practices and wishes of the commissioner organization.

The system had two specific sets of requirements. The first was for the functional test automation framework, which were mostly generic and common with other typical test automation implementations. These requirements came mostly from common practices and industry experience, with some degree of custom tailoring for the needs and existing conventions of the commissioner organization.

The second set of requirements related to the security and isolation of the system, due to the presence and use of live malware. These requirements came primarily from secure systems engineering experts within the organization. Malware handling within the organization also has strict guidelines, which included undertaking a certifying training session.

The core requirements of the complete system were identified as following:

- Test coverage; the tests executed by the system need to cover all the major features and functionality of the software-under-test. The process of adding additional test coverage after the completion of this study should be simple and fast.
- Reliability; the results produced by the system need to be stable and trustworthy. The system needs to operate autonomously for extensive periods of time without significant downtime or maintenance required.
- Efficiency; the execution time of the tests needs to be as small as possible, to minimize the quality feedback loop for developers and other stakeholders depending on the results.
- Performance; the system should not consume excessive amounts of hardware resources. This includes resources consumed by both the process infrastructure and the virtualized test environments.
- Security; the system needs to be secure, so the malware used by it cannot escape the test environment to other systems and network environments.
- Maintainability; the system needs to be easy to maintain, and any need for manual maintenance should be indicated in a visible and understandable way.

Chapter 2.2, Evaluation Criteria, describes in more detail how the fulfillment of the requirements can be evaluated.

2.1.1 Technology Requirements

The technologies utilized in the study needed to, in addition to being technically viable for implementing the required functionality, conform to the technologies currently used in the commissioner organization. This ensured the solution can be further developed and maintained with knowledge and tools already present in the organization.

The technologies that were chosen based on prior presence in the organization require no further viability evaluation or justification for utilization in this thesis, and will thus be introduced only from a functional point-of-view. For technologies not already present in the organization, the evaluation was conducted based on industry standard options and feasibility for this study, in addition to possible preferences from the commissioner organization.

Some infrastructure and services used in this study were already present in the organization and were thus not deployed and configured in the scope of this thesis. This includes the network environments, the virtualization platform, the malware storage, and some additional services and servers. The thesis report introduces these technologies and services from a functional usage point-of-view, and how the system integrates with them.

The exact collection of technologies utilized in this thesis is introduced and described in detail in Chapter 6.4, Technology Review.

2.1.2 Output Requirements

As the intended outcome of this thesis was a fully functional system, the output requirements relate primarily the output of the system itself. The output produced by the system is primarily quality-related data, to be used in evaluating whether the software-under-test is of sufficient quality for customer release.

The most significant data produced by the system is the results of test cases, which is often relatively binary: a test either passes or fails. The significance of a test result depends on the test case, some of which are more critical than others. The exact methodology of the test cases is described in detail in Chapter 5.2, Test Cases.

Some tests can collect and produce additional data, which can be used to further evaluate the software-under-test and the test results. The additional data can include:

- Measured execution speed of different operations.
- Performance metrics of the test environment, for example memory consumption and processor usage data.
- Screenshots taken in various phases of the tests.

Additional data can also be collected from the test environment after testing, which can provide pointers to the exact location and reason of problems encountered in the tests. Such data can include:

- Log files
- Memory/crash dumps
- Operating system event logs

As the scope of the tests primarily related to detection and remediation of malware, related information also needed to be collected and presented. This included:

- System changes performed by the malware
- Malware detection rates
- Malware detection misses, with information on the missed samples
- Malware disinfection failures

The results and artifacts produced by the system need to be visible and easy to parse. The overall status of the system and tests needs to be presented in a simple information radiator

""Information radiator" is the generic term for any of a number of handwritten, drawn, printed or electronic displays which a team places in a highly visible location, so that all team members as well as passers-by can see the latest information at a glance: count of automated tests, velocity, incident reports, continuous integration status, and so on." [5]

2.2 Evaluation Criteria

The evaluation criteria for the system closely followed the requirements listed in Chapter 2.1. Most of the requirements were reasonably measurable, though some require additional interpretation. The evaluation of the system was performed in co-operation by stakeholders in the commissioning organization and the author of this study.

The reliability of the test results is a key factor in determining the usefulness of any test system; a test should only fail when it encounters a functional problem or unexpected behavior in the software-under-test. Failures due to problems in the framework or test environment produce noise, and waste a significant amount of time from the person evaluating the results. The reliability was evaluated by observing the consistency of the results and the causes of possible failures.

The precise test coverage of any test automation system is commonly difficult to measure and present, but with proper naming of test jobs and test cases the major feature- and component-level coverage can be presented. The solution presented in this thesis did not include tests performed against instrumented software, which could produce exact code coverage reports.

The efficiency and performance of the system are demonstrable by monitoring and analyzing the execution times of the tests and different flows in the infrastructure, and the hardware resource usage of the different components of the system during execution.

The security of the system was evaluated on a conceptual level, by systems engineers responsible of the broad environment the system will reside in, and software security experts within the organization.

3 Existing Implementations

There are many literary sources and existing implementations for automated analysis of malware, but few for automated functional testing with real malware. Such systems undoubtedly exist, however they are primarily developed internally in anti-malware vendor- and security research organizations. The exact design and details of such systems can thus be considered company-internal information, and fall under company non-disclosure agreements. This is primarily due to how such systems integrate with the existing infrastructure in the organization, most notably the malware storage- and handling systems.

Existing open-source implementations may be available for some individual parts of the system, but none that cover all the required functionality as a complete end-to-end system. As such the existing knowledge for this study primarily only relate to descriptions and classifications of malware, functionality of different anti-malware techniques and technologies, and automated software testing methodologies.

Some of the existing knowledge comes from internal sources within the commissioner organization, and as such the exact source of the information cannot be disclosed. Such information primarily relates to the anti-malware techniques and technologies. Most of the information applies to all major anti-malware products in the market, but as this study is conducted using only one specific product, such generalizations cannot be made.

The design and implementation steps performed in this study are, while taking pointers from industry-standard practices, original.

4 Malware and Anti-Malware

The following chapters provide a brief introduction into common types of malware; how they appear in computer systems, and how they are commonly counteracted by anti-malware software.

4.1 Malware

Malware, or malicious programs, refers to software that cause harm or otherwise compromise the security of computers without the knowledge of the user, commonly for either monetary or destructive reasons. Though there are several types of malware, categorized by their behavior and purpose, such as Trojans, Worms, Backdoors, Key Loggers, Downloaders and Ransomware, most of them can be put into a few high-level categories by how the infection occurs and how they can be detected: memory infections, system infections, rootkits and exploits. This study does not attempt to target all possible types of malware, but only those that are currently prevalent and commonly encountered by typical Internet users. This chapter covers the high-level categories of malware, and how the infections are commonly detected and remedied by anti-malware software.

Malware infections, regardless of type, commonly occur when the user of the computer is tricked into running a malicious executable file, often downloaded from the Internet or delivered as an email attachment. Infections that do not require the user to run an executable are also possible, for example by accessing websites that host malicious web browser scripts.

The following chapters describe some common malware infection scenarios. How they can be tested in automatic fashion will be further described in Chapter 4.2, Anti-Malware Techniques and Testing.

4.1.1 Memory Infections

A memory infection refers to malware that runs in the computer memory, more specifically in a process in the computer operating system. The process in which the malware resides in can either be a new one created by the malware, or an existing one to which the malware injected itself. Injecting existing processes is more common, as it makes it more difficult for the user to notice that an infection has taken place.

Memory infections can be detected by scanning the running processes, memory-block by memory-block, and looking for code signatures that could identify the malware. Identifying the malware in a process requires there to be an existing signature detection for it, so only known and analyzed malware can realistically be detected in the memory.

When a memory infection is detected, the process in question is commonly terminated, and the file which launched it is further analyzed. If a malware detection is found in the originating file, it then needs to be remedied. The remediation method depends on the type and method of the infection. If the file is entirely malicious, it is often deleted or quarantined, depending on the settings of the anti-malware product. If the infection was injected into an otherwise legitimate process, it needs to be disinfected. The disinfection is done by attempting to remove the injected malicious code from the file.

4.1.2 System Infections

A system infection refers to a malware infection that has compromised the operating system installation, by injecting itself into the processes, file system or registry of the operating system. The injection can happen for example by dropping or replacing files in the system folders, making it look like a part of the operating system instead of malware.

System infections can be detected by scanning the operating system folders and files. If detections are found, the files should be disinfected or restored, for example from a System Restore Point in the Windows operating system.

4.1.3 Rootkits

Rootkits target the kernel of the operating system, which, if successful, gives it more freedom and capabilities than typical malware infections. Rootkits are often used to hide malicious processes and network connections from the user and anti-malware software. [6]

The prevalence of rootkits has significantly diminished in recent years, and thus they will not be specifically targeted in this study

4.1.4 Exploits

Exploits target security vulnerabilities in computer software, using them as an attack vector into the targeted system. Exploits are not technically malware in themselves, but as they are often used to deliver malicious payloads into computers, they are serious security threats.

Common targets for vulnerability exploitation include operating systems, web browsers and widely used file formats such as Adobe Flash and PDF documents [7]. The organizations maintaining the most popular exploitation targets, such as Microsoft and their Windows operating system, commonly react quickly to discoveries of new vulnerabilities, and promptly develop and distribute security fixes. This means the time window for exploiting such software is small, but the large number of users not promptly installing security fixes whenever they become available more than compensates for it.

Preventing the exploitation of vulnerable software is difficult, if not impossible, and the focus is instead in quick response. The easiest way to protect computers from exploits is to keep all software, especially third-party software, up-to-date with latest updates and security patches. Many real-world exploit attacks occur using so-called 0-day exploits, that is exploits that have only recently been found and not yet fixed by the software vendor.

Anti-malware software cannot do much to prevent exploitation, but they can instead attempt to block the types of payloads the exploits can deliver. For testing exploit protection features, this study uses exploits targeting the Adobe Flash multimedia file format. The exploitation will be established by accessing a malicious web address with a web browser running a vulnerable version of the Adobe Flash Reader software.

4.1.5 Malware Persistence

Regardless of the type, one of the main objectives of malware is to persist itself into the target system, so it can stay operational after the system is rebooted or power cycled.

The most common method of persistence is the operating system registry [8]. The malware can create a launch point in the registry, which will automatically execute a file upon startup. The file to be executed can be hidden in the file system, and named such that it does not raise suspicion even when the launch point is observed by the user.

Another way of achieving persistence is “trojanizing” system binaries. In it, the malware injects code into a system binary, which in turn executes the malware the next time the system binary is run. [9]

Detecting and removing the malware persistence mechanisms is critical in the malware disinfection process. If it is not done properly, the malware will eventually re-infect the system.

4.2 Anti-Malware Techniques and Testing

This chapter introduces some common techniques and operations that anti-malware software performs to detect malware, how they work and how those operations could be automatically executed and verified for the purpose of testing. Some of the operations involve direct interaction with the anti-malware product, which requires that some of its functionality is possible to be automated or scripted. Preferably this is done by interacting with the product programmatically via an Application Programming Interface (API), or Remote Procedure Call (RPC), access, but if this is not possible or supported by the product, a typical way of automating the functionality is by scripting command-line operations in the product. In this study we utilize direct API access with the anti-malware product under testing, and drive operations programmatically whenever possible.

4.2.1 On-demand Scanning

On-demand Scanning (ODS) refers to scanning files that reside in a local or remote file system. The contents of the files are scanned and checked against signature detections in the detection database of the anti-malware product. The malware files are not executed, so no infection takes place, which makes testing on-demand scans relatively easy and safe, though the test machine does need to have the malicious files present in its file system. When simply testing whether the anti-malware product has detections for certain malware samples, on-demand scans are the quickest and most efficient way of verifying it.

On-demand scans are performed by telling the anti-malware software to scan locations in the file system containing the malware files. Possible malware detections in the files are then reported by the anti-malware product. The reports, depending on how they appear to the user, can then be parsed and evaluated by the tests. If no detection information can be extracted from the product by the tests, it is still possible to verify what detections took place by configuring the product to delete infected files, and then verifying that all the expected malware files were removed from the file system. On-demand scans can be easily tested against any malware, and requires no prior knowledge of their type or behavior.

In most anti-malware products on-demand scans can be initiated with a command-line tool, making it easy to automate the related tests. In the anti-malware product used in this study, on-demand scans with a command-line tool also produce a report with details about possible detections, which will be utilized in reporting the results of the tests.

4.2.2 On-access Scanning

On-access Scanning (OAS) occurs when anti-malware software intercepts file-related events in the operating system, such as opening, copying or moving files in the file system. The event is blocked until the file in question is scanned for malware, after which it is either allowed or denied, depending on the result of the scan. If the file is found to be malicious, it is handled according to the choice of the user or the settings of the anti-malware product, which usually in the context of on-access detections means quarantining the file.

Quarantined files are placed in a special isolated location in the file system, from where they can be either restored, in case the detection is later determined to be a false positive (see Chapter 4.2.12), or deleted by the user. Quarantined files can also be submitted to the anti-malware product vendor for more thorough analysis.

On-access scans are among the most common ways of preventing malware from entering systems, as they occur every time a new file lands in the file system, either by copying or downloading. As such the on-access scanning tests are also among the most important tests to be performed in any automated anti-malware testing.

Automating on-access scan tests mostly involves generating and verifying file system events, and requires no interaction with the anti-malware product, except for the purpose of extracting detection information. As with on-demand scans, on-access scans can be tested against any malware without any additional preparation or prior knowledge.

4.2.3 Memory Scanning

Memory scanning scans the running processes in the operating system in attempt to detect memory infections, as explained in Chapter 4.1.1. If a process is found to be infected, it is terminated and the originating file is remedied according to the type of the infection and settings of the anti-malware product.

Testing memory scanning, especially with real malware, is significantly more complex than the other anti-malware features, as it requires the presence of an active memory infection, and knowledge of the malware's behavior. Producing memory infections in a controlled and secure manner can be difficult, which are explored in more detail in Chapter 7.8.4, Memory Scan.

4.2.4 System Scanning

A system scan scans the folders and registry of the operating system, in order to detect and remedy system infections, as explained in Chapter 4.1.2. Depending on the anti-malware product and the operating system, a system scan might also attempt to restore compromised operating system files. Infected system files have to be handled carefully, as simply removing them might render the operating system un-functional.

Scanning the operating system registry is the most important function of the system scan, as registry launch points are a very common method of malware persistence.

4.2.5 Web-traffic Scanning

Web-traffic Scanning (WTS) is similar to on-access scanning, but instead of intercepting file system events, it intercepts network traffic packets. Incoming network packets are intercepted by the anti-malware product before they reach the disk, and scanned for malware. If malicious content is found, the network request is blocked.

Also similar to on-access scans, web-traffic scan tests can be automated without interaction with the anti-malware product. A web browser or some other software capable of generating HTTP requests can be automated to download malicious files, while the tests verify that the requests are blocked accordingly.

4.2.6 Heuristic Protection

Heuristic protection refers to detecting and blocking malware based on behavioral analysis instead of signature detections. While signature detections require the exact malicious file to be known to the anti-malware software, heuristic protection can detect variations and targeted mutations of the malware by its behavior. A heuristic engine monitors events in the file system, observing operations such as file, process and network socket creations, and uses heuristic detection patterns to identify malware based on them.

Heuristic protection can be tested by simply executing the malware file, and observing whether a detection event takes place. The testing can be enhanced by using malware that is known to perform certain operations, such as creating a process or accessing a network address, and verifying that they are detected and blocked accordingly.

4.2.7 Exploit Protection

Exploit protection features in anti-malware software attempt to mitigate the damage caused by exploitation of software. Exploit protection functionality in anti-malware products cannot realistically prevent the exploitation itself, as exploits commonly utilize vulnerabilities in other software, the blocking of which is beyond the capabilities of the anti-malware software.

As the primary objective of exploits is to deliver and execute malicious payloads, exploit protection features can minimize the damage caused by them by detecting and blocking the delivery and execution of the payloads, and alerting the user to the presence of the exploit. It is then up to the user to remedy the exploited software, primarily by upgrading it to the latest version, where the vulnerability that enabled the exploit is hopefully fixed.

Testing exploit protection is most convenient by utilizing an exploit toolkit, such as the Metasploit Framework, described in the following chapter.

4.2.8 Metasploit

Metasploit, or Metasploit Framework (MSF), is a penetration testing toolkit developed by Rapid7. It maintains a database of known exploits, and an extensive collection of exploit delivery methods and payloads. [10]

As Metasploit is a reputable penetration testing toolkit and not a malicious attack tool, it only provides exploits that have already been fixed by the software vendors. Thus to test exploits with Metasploit, the target machine needs to have an older version of the exploitable software installed.

The deployment and usage of Metasploit is covered in more detail in Chapter 7.5.1, Metasploit Framework.

4.2.9 Reputation

In a reputation check, the anti-malware software calculates the cryptographic hash, for example SHA-1 or SHA-256 [11], of a file, and sends it to a backend to be checked against a reputation database. If a match is found, that is if the file has been seen before, the backend returns information about the file, and whether it is classified as malicious or not.

Only files previously encountered by the anti-malware vendor can be identified with a reputation check, so it does not protect against new unknown threats or mutations of existing ones.

A reputation check can return different types of classifications, such as:

- Known clean; the file is known to be clean and is safe to execute.
- Known malicious; the file is known to be malicious and should be blocked.
- Unknown; the file has not been seen by the backend before, and should be scanned for malware locally.

4.2.10 Malware Remediation

Once the presence of malware has been detected, it needs to be handled properly. In the event of a detection, the anti-malware software commonly asks the user how to remedy it. In the context of automated testing, the remediation method is commonly preconfigured in the product, to remove the need for additional GUI interaction.

There are several different methods of handling detected malware, depending on the context in which the detection occurred. For example in the event of an on-access scan detection, the most common remediation methods are:

- Delete; the malicious files are removed from the file system.
- Quarantine; the malicious files are stored in a secure location in the file system, for possible further analysis.
- Disinfect; the anti-malware software attempts to remove the malicious code from the infected file. This is only applicable for files that have been injected by malware, rather than files that were created to be malware in the first place.
- Report only; the user is informed of the detection but no automatic action is taken.

Verifying the success of the remediation operation is an important part of anti-malware testing, as malware that has been detected but not remedied properly remains a threat to the system.

4.2.11 Whitelisting

Anti-malware products commonly keep a list of files and software that are known to be clean and trustworthy, also known as a whitelist. Whitelisting is used to avoid unnecessary scanning of files, reducing the load the anti-malware product causes to the system.

Whitelists primarily contain software from known and reputable software vendors, and files belonging to operating systems. Also files belonging to the anti-malware product itself are commonly whitelisted.

4.2.12 False Positives

A false positive refers to the event in which a clean file is mistakenly detected as malware by the anti-malware product. This might happen if a piece of software has behavior similar to that of some known malware, causing it to be detected by heuristics. False positives can also occur if a signature detection is too generic, leading to unintended matches.

False positives do not compromise the security of a system, but are a significant inconvenience for the user. In the worst case scenario, a false positive might render some software unusable, if some critical file in it was mistakenly detected as malware and removed.

Any false positives encountered in anti-malware testing need to be handled properly, preferably by reporting it to the author of the faulty detection.

5 Software Testing

This chapter introduces and explains some common software testing terms and concepts that will be referred to later in this report, with emphasis on how they are applied in the context of automated software testing.

Software testing is a crucial part of the software development process. Its primary purpose is to find defects and failures in the software before they are released to customers. The defects can range from minor inconveniences and cosmetic issues to serious flaws that can, depending on the software, even cause accidents or fatalities [12]. In addition to finding defects, testing also evaluates the confidence in the quality of the software, to assist in decision making in the software development- and release process [13].

In the context of anti-malware software, defects can expose the customer systems to malicious programs, while giving a false sense of security to the user. For a typical home user this might be a mere inconvenience, but for a large organization or a corporation, such as a bank, the consequences of the defects can be catastrophic.

As anti-malware software is directed more towards home users and corporations instead of critical infrastructure, it is not regulated in the same way as for example software for power plants. As such the quality control processes for anti-malware software do not need to adhere to formal standards, such as the ISO-standards.

5.1 Automated Software Testing

Automated software testing, also known as test automation (or TA), refers to a method of software testing that uses programmed scripts to automatically execute the different steps of software testing, with no human interaction required.

Automated testing can be split to two distinct categories: functional and non-functional test automation. Functional test automation refers to tests that verify specific functionality against a set of pre-defined expected results. Functional tests are generally quick to execute and can be run repeatedly several times a day.

Non-functional test automation refers to tests that focus more on observing and analyzing behavior and functionality than verifying it. The most common form of non-functional testing is performance testing. Performance tests generally monitor, record and analyze the resource usage of the software, such as memory, processor and I/O usage, while performing different functional operations with the software. Also the execution speed of the functional operations is generally recorded and analyzed.

Test automation scripts can be considered as software programs themselves, often implementing concepts of object-oriented programming. However the scripts are commonly relatively light-weight in terms of programmed logic, as they only aim to perform tasks of limited scope. Test automation development should follow common software development conventions, and the skill-set of an experienced test automation developer is very similar to that of a regular software engineer. Ideally software engineers working on the product development should also participate in the development of test automation.

The main advantages of automated testing compared to manual testing relate to the speed and repeatability of the tests. An automation script can execute and verify practically all functional operations in the software-under-test faster than a human can. Also automated tests can be repeated for every code change in the software-under-test, which would be a very monotonous task for a human tester to perform.

Though automated testing can be employed extensively to test software, there are some forms of testing that are not viable for automation, such as user experience- and language testing. User experience testing requires human judgment that cannot be realistically scripted, while automating language testing would require extensive effort in establishing the required language rules and dictionaries.

Automated tests are especially suitable for anti-malware testing, as all the operations and verifications in this context can generally be performed programmatically, and much faster than a human tester could. Though human judgment might be needed in analyzing the behavior of malware, in this study all the malware used in the testing are already known, and as such their behavior can be anticipated.

In the organizational unit where this study was performed practically all functional and non-functional testing is automated. Most of the automated test cases have been written by software engineers, while test automation specialists focus on the frameworks and systems that enable and support the testing.

Automated tests are commonly defined and organized in entities called in test cases. The following chapters offer a brief explanation on how test cases are formed, and how they are organized into test sets.

5.2 Test Cases

In software testing, a test case is “a document, which has a set of test data, preconditions, expected results and post-conditions, developed for a particular test scenario in order to verify compliance against a specific requirement” [14].

In the context of automated testing a test case has a similar purpose, but instead of only documenting the preconditions, steps and verifications, it also contains the implementation for each of them, written with a programming- or scripting language.

The programmed implementation of a test case often resides in a programming entity called a class, also known as a test class in the context of automated testing. A test class can contain one or more test cases, organized in functions, also called test functions or test methods. Not every method in a test class necessarily defines a test case, as a test class can also contain and implement different preparatory operations.

In this thesis a test class contains one or multiple test cases of similar target and scope, and a number of complementary setup methods. The test classes commonly follow the following sequential steps:

1. Configure logging, to allow the test cases to write information to a log file.
2. Preparatory steps that configure the system for the test case, or set up some data that is required by the test case. These steps can reside in the same function as the actual test cases, or in a separate functions that are executed prior to the tests.
3. Execute the test cases, in a pre-defined sequence.

4. Verify the results of the test cases. Though the preparatory steps are often shared between the test cases, each test case is responsible for verifying its own results.
5. Collect and store log files and other artifacts, both from the test cases and the software-under-test.

The technical implementation of test cases is described in detail in Chapter 7.7, Test Cases.

5.3 Test Sets

In testing operations that contain a large number of test cases, it is not often viable to execute every test case in every test session. To split the tests into smaller and more manageable units, test sets can be employed. A test set is a not a common software testing term as such, though it bears some resemblance to a test suite.

In this thesis, a test set is a collection of test cases that target similar features or functionality in the software-under-test. The implementation of a test set is a file that contains a list of test case names, and possible additional instructions. When the test automation framework runs the test set, it executes the test cases defined in it in sequential order as they appear in the test set file.

See Chapter 7.6.2 for more information about the technical implementation of test sets, and how they are executed.

6 Architecture Overview

The system this thesis implements consists of three main functional areas: the end-to-end backend infrastructure, the virtual environment and the client-side test automation. This chapter drafts the broad architecture of each area, followed by a review of the different technologies chosen for the implementation, and the rationale for each choice. The actual implementation steps of the system will follow in Chapter 7, Solution Implementation.

6.1 Backend Infrastructure

The purpose of the backend infrastructure is to deliver artifacts to and from the test environment, and to enable management of the execution and lifecycle of the tests.

The artifacts delivered to the test environment include:

- Software builds to be tested
- Test cases
- Tools utilized by the tests

Artifacts retrieved from the test environment include:

- Test results
- Log files
- Memory dumps
- Additional metrics collected from the test environment

The infrastructure spans several network environments, due to the considerations related to the handling of live malware. The network environments are already present in the organization, and this study utilizes them only as a typical user.

The Green network is the common test network in the organization, where most of the functional and non-functional testing takes place. No malware is allowed in this network environment, except for non-malicious malware test samples such as EICAR [15]. All typical network services, such as Domain Name System (DNS), Dynamic Host Configuration Protocol (DHCP) and Network Time Protocol (NTP) are available in the Green network.

The Orange network contains the main malware storage systems of the organization. Malware storage and scanning is allowed in this network environment, but execution of malware and live infections are not. The Orange network has a limited set of network services available, such as DNS.

The Red network is the main malware analysis and testing network. This network has no malware-related restrictions, so storage, scanning and execution of live malware is allowed. The Red network provides no network services, only the core connectivity. As such, every machine wishing to connect to the network needs to have a pre-acquired IP address.

As the test system implemented in this thesis has to integrate with the continuous integration flows in the organization, the management of the tests has to reside in the Green network. The actual tests involve execution of live malware, and thus have to reside the Red network. The Orange network is utilized for retrieving malware samples to be used in the tests.

The backend infrastructure needs to perform the following functions:

- Automation server for hosting test jobs in the Green network.
- Integration with the continuous integration flows in the Green network.
- Retrieve malware from the malware storage in the Orange network.
- Deliver test artifacts to the Red network.
- Setup and execute tests in the Red network.
- Return artifacts from the Red network.

Test jobs are a way of splitting the testing operation into multiple independent and parallel units. Instead of running all the tests in one session, which would take an excessively long time and make it difficult to differentiate and digest the results, separate test jobs should be created for each major feature or sub-component of the software-under-test. Having multiple test jobs targeting different parts of the software also ensures that faults in one feature or component in the software does not cause all tests to fail.

Due to security concerns, there is no direct connectivity between the Green and Red networks. As this study needs to transmit artifacts between them, a special channel had to be implemented to enable it. This channel is named Rabbithole, and is described in more detail in Chapter 7.2.

As the Rabbithole channel can only be used to transfer files, it cannot route synchronous connections from the automation server in Green network to the test environment in Red network. This means the test jobs cannot access the test environment directly, and a new controller service, henceforth referred to as Controller, is needed in the Red network to manage the communications and execution. The Controller will receive artifacts through the Rabbithole, parse instructions from them and execute tests accordingly. Figure 3 shows the high-level architecture of the system, across the different network environments.

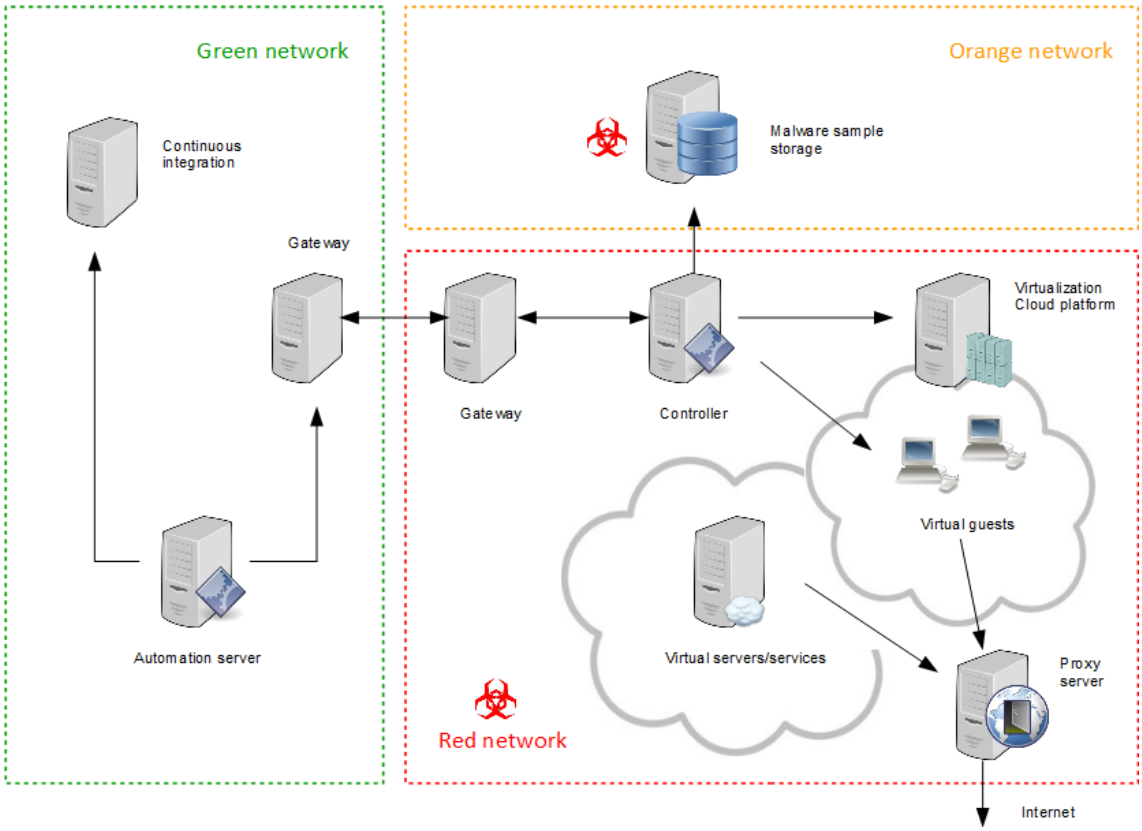

Figure 3. Architecture Overview.

Figure 3 drafts the hardware, software and network infrastructure of the system. The full details of the infrastructure are described in more detail in Chapter 6 and Figure 4.

6.2 Virtual Test Environment

The tests performed by the system were chosen to be executed in a virtual computing platform. This is due to the scalability and flexibility provided by virtualization compared to physical hardware.

As the software-under-test has clients for various platforms, the tests also need to cover different operating system platforms and -versions. Virtualization makes this simpler by using virtual machine images. The virtual platform can maintain an extensive collection of images for different operating systems and -versions, allowing tests to choose which platform to run on. As virtualization also enables concurrency, the tests can be executed on targeted operating system versions simultaneously (depending on the capacity of the virtual platform.)

Virtual environments are not ideal for testing with real malware. Some more advanced malware families can detect they are being run in a virtual environment, react by skipping the infection or otherwise changing their behavior to evade malware analysis. However there are techniques for obfuscating the virtualization, depending on the virtual platform being used. This study attempts to obfuscate the virtualization by ensuring no device or driver names in the test machine contains known references to virtualization. This can be done by configuring the virtual machine settings in the OpenStack platform accordingly. Chapter 7.8.4 contains more information on how anti-virtualization measures affected this study.

6.3 Client-side Test Automation

The client-side test automation defines and executes the actual tests performed by the system, making it the most critical functional area in the overall system. The tests are defined as test cases and test sets. Test cases commonly test a single feature in the software-under-test, while test sets define a collection of test cases to be executed in the same test session. In general test cases should be self-sufficient, doing all required preparation and verification steps without relying on possible previous steps in the test set. The test set can also include test cases that perform different setup operations in the test environment in preparation for the tests that follow. Such cases will be referred to as "setup cases" in this study.

Each test case should collect all the data and artifacts needed to investigate the result that specific case. This, instead of collecting all artifacts as the last step in a test set, ensures that possible interruptions of the test set execution does not invalidate the tests that were already executed. The test case should also notice if a crash occurred in the software-under-test during it, and handle the memory dumping and dump collection accordingly. This ensures the correct context for the crash is known already before it is investigated.

The test cases in this system primarily interact with the software-under-test programmatically. In cases where the desired functionality is not available via API or RPC access, command-line interfaces are utilized. Only if neither of these are viable is the Graphical User Interface (GUI) automation utilized. This is primarily to minimize the execution time of the tests, as GUI automation is generally much slower than a programmatic approach.

To avoid duplicating test code, often repeated operations, such as interfacing with the anti-malware product or performing operations in the file system, are organized into libraries.

6.4 Technology Review

The automation framework chosen for hosting the test jobs is Jenkins [16]. It is an open-source automation server, developed by the Jenkins Project and distributed under the MIT license [17]. The primary reason for choosing Jenkins is its prevalence in the industry, as well as its existing utilization in the commissioner organization.

Jenkins works with a master/slave server architecture, where the master instance hosts the test jobs, configurations and stored artifacts, while one or more slave instances drive the actual execution. Each slave instance has a configurable number of executor threads, each driving a separate test session.

The virtualization platform used to host the test environment is OpenStack, an open-source cloud computing platform managed by the OpenStack Foundation [18]. An existing instance was already available in the Red network, and thus the deployment and configuration of OpenStack is not in the scope of this study. The creation and configuration of the virtual machine images used in the system are covered in detail in Chapter 7.4, Virtual Test Environment.

All data transfers in the infrastructure are done using the Secure Copy (SCP) method of the Secure Shell (SSH) protocol [19]. SSH was chosen due to its broad utilization in Linux-based systems, which consist most of the machines involved in the hardware infrastructure of the test system.

The programming language chosen for implementing the client-side test automation is Python, a high-level dynamic programming language developed and maintained by the Python Software Foundation [20]. The following characteristics of Python make it deal for implementing test automation:

- Interpreted language; the source code is compiled at runtime, removing the need for pre-compiling it into executable binaries.
- Multi-platform; Python is available on all major operating systems: Windows, Linux and OS X.
- Large standard library, which allows relatively complex implementations without additional libraries.
- Extensible; Python has a very large collection of third-party libraries, also known as modules, available in services such as the Python Package Index (PyPI). Also installing new libraries is easy and fast with tools such as pip and setuptools.

The following third-party Python modules are utilized in the implementation of the client-side test automation:

- nose; extends the Python standard library unittest module with additional features, making it easier to write, find and run tests.
- pywin32; Windows extensions for Python, which allows access to Windows API and registry.
- WMI; a library for accessing the Windows Management Instrumentation (WMI) system.
- pywinauto; a Windows graphical user interface (GUI) automation library.

All the modules, and the Python installation itself, are utilized together with another third-party Python module named virtualenv. It allows creating stand-alone Python environments, with entire module libraries packaged into it, that can be deployed on isolated systems such as the environment defined in this thesis.

7 Solution Implementation

This chapter describes in detail the steps taken to implement each functional part of the system, and how they work together to form the full end-to-end flow of the system. Figure 4 provides the detailed architecture overview of the system.

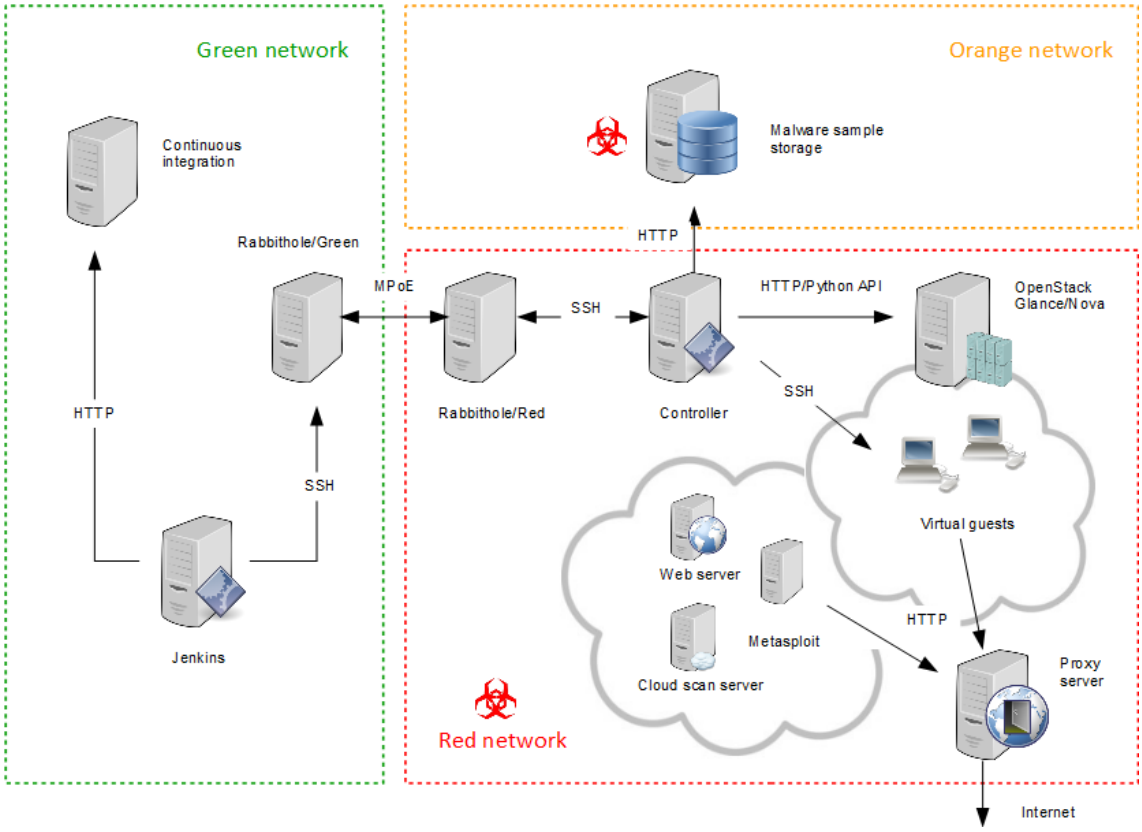

Figure 4. Architecture Overview Details.

Figure 4 expands on the architecture overview introduced in Chapter 6.1, adding in the interfaces and protocols used by the different parts of the infrastructure. The following chapters describe the different components of the infrastructure in detail.

7.1 Test Jobs

Due to the architecture of the system, where the actual client test environment is located in a different physical network environment than the Jenkins instance hosting the test jobs, the test jobs themselves are relatively lightweight and have no extensive testing logic. Their purpose is to, in sequence:

1. Collect testable artifacts from internal and external sources, primarily other Jenkins jobs.
2. Package the artifacts into a ZIP payload archive.
3. Send the payload archive to the test environment, though the Rabbithole channel.
4. Wait for a response archive from the Rabbithole channel.
5. Extract the response archive to a local workspace.
6. Record the test results.
7. Create and offer artifacts.

The testable artifacts packaged into the payload can include:

- Anti-malware- and other product installers
- Offline-installable anti-malware engine- and virus definition updates
- Test cases
- Additional tools utilized by the tests

The payload archive is transmitted to the Rabbithole channel using the Secure Copy (SCP) method of the Secure Shell (SSH) protocol.

In addition to the testable artifacts, the payload contains a configuration file for the Controller in Red network. Table 1 lists the parameters defined in the configuration file.

Table 1. Configuration File Parameters.

Parameter	Definition
TEST_NAME	Name of the test session, combined from the name of the test job and current build number. The name is used as the name of the virtual machine instance in OpenStack.
TEST_SET	Name of the test set to execute.
TEST_IMAGE	Name of the virtual machine image to use.
TEST_FLAVOR	Hardware specifications to be assigned to the virtual machine instance.
MALWARE_SET	Name of malware set to use.
MALWARE_SAMPLES	List of hashes of malware samples to use. Mutually exclusive with MALWARE_SET.
KEEP_INSTANCE	What to do with the test virtual machine after testing: • "no": always delete the virtual machine • "yes": always keep the virtual machine running • "crash": keep the virtual machine only if a crash was detected • "timeout": keep the virtual machine only if the tests hit timeout
TIMEOUT	How long to wait for the tests to finish, in minutes, before forcefully terminating the virtual machine instance.

After the payload archive has been created and sent to the Rabbithole, the test job remains in a loop waiting for a response archive of a certain name to appear in the Rabbithole. If the response archive does not arrive within a set amount of time, the wait loop is aborted and the test job is terminated.

The test results are recorded in XUnit XML format. XUnit is a unit testing framework widely utilized in software development. A plugin for the Jenkins framework allows recording and storing test results in the XUnit XML format, which the utilized Python testing module, nose, is capable of outputting.

Following the conventions of the Jenkins framework, the execution of a test job can result in one of four result states:

- Blue; all operations and tests in the test job passed successfully.
- Yellow; one or more test cases failed.
- Red; an infrastructure- or some other unexpected failure occurred, which prevented the test job flow from executing correctly.
- Grey; the job execution was aborted before it could finish.

A test job can offer files as artifacts, which can be downloaded from the test job after it has finished executing. The files offered as artifacts often include different log- and other files collected from the test environment by the Data Collector module, introduced in Chapter 7.6.3, and malware scan reports, introduced in Chapter 7.9.

7.2 Rabbithole

The Rabbithole is a secure channel implemented to enable file transfer between the safe Green network and the malicious Red network. It was setup for the purposes of this study by a network infrastructure team, and as such its exact configuration steps will not be described in detail in the scope of this study.

The channel is implemented with a physical link between two gateway servers, one in each network. The Linux-based servers communicate with each other using a file transfer method called Message Passing over Ethernet (MPoE) of the ATA over Ethernet (AoE) network protocol. [21] It is a relatively obscure network protocol, which does not use the Internet Protocol (IP), and as such cannot be utilized by most known malware.

As a secondary security mechanism, all files that are transferred from the Red network to the Green network are scanned for malware by the Linux machine that implements the Green network side of the Rabbithole. If a file coming from the Red network is detected to be malicious, it is deleted and a note informing the user of the detection is left in its place.

The Rabbithole is accessed using the SSH communications protocol. Files are copied to- and from two directories that are present on both sides of the channel. Files to be transferred to the other side of the channel are placed in the export-folder. After the file transfer has completed, the file will appear in the import-folder on the other side of the channel. To prevent partial transfer of files that have not yet been fully copied to the export-folder, only files with a certain prefix in the filename are transferred. The prefixes are named to-red-network and to-green-network accordingly.

The test jobs utilize the channel as follows:

1. Copy the payload archive to the export-folder on the Green network side of the channel.
2. Add the to-red-network prefix to the filename.
3. Start waiting for a file with a certain filename (including the to-green-network prefix) to appear in the import-folder.
4. Copy the file to the job workspace with SCP.

7.3 Controller

As introduced in Chapter 6.1, the Controller is a service that manages the test automation flow on the Red network side of the infrastructure. It implements the following primary functions:

- Monitor and retrieve files coming through the Rabbithole.
- Configure test virtual machine instances and manage their lifecycle (creation, deletion) in OpenStack.
- Setup test sessions.
- Retrieve artifacts from virtual machine instances.
- Create results payload and send it to the Rabbithole.
- Perform other infrastructure jobs, such as checking the network connectivity to different parts of the infrastructure.

The Controller was implemented as a multi-threaded Python application, running in a Linux machine in the Red network. When it is started up, the following setup sequence is executed:

1. Initialize a Python Queue object, for distributing work to the worker threads.
2. Start a dedicated thread, named RabbitholeWatcher, for managing Rabbithole communications (monitoring, sending and retrieving files.)
3. Start a dedicated infrastructure worker thread, named InfraWorker, for various infrastructure jobs, such as updating the Controller, checking the heartbeat of various related systems, and creating pre-defined malware sample sets.
4. Start a variable number of Worker threads, for running the actual test sessions.
5. Start the main program loop.

When an incoming payload file appears in the Rabbithole, a sequence of actions takes place, as listed in Table 2.

Table 2. Controller Workflow.

#	Actor	Action
1	RabbitholeWatcher	Copy the payload from the Rabbithole with SSH.
2	RabbitholeWatcher	Place the payload into the Python Queue object.
3	Worker	First available worker thread picks up the payload from the Queue.
4	Worker	Extract the payload to a temporary folder.
5	Worker	Read the configuration file from the payload.
6	Worker	Provision new virtual machine in OpenStack.
7	Worker	Retrieve malware files (see details in Chapter 7.3.1, Malware Handling.)
8	Worker	Copy malware files to the virtual machine with SCP.
9	Worker	Copy testable artifacts to the virtual machine with SCP.
10	Worker	Initiate tests on the virtual machine, depending on guest operating system: • Windows: set up a bootstrap script, reboot machine. • Linux/OS X: run tests remotely with SSH.
11	Test virtual machine	Activate Python virtual environment.
12	Test virtual machine	Run tests.
13	Test virtual machine	Collect product logs and other artifacts.
14	Worker	Copy results and other artifacts from the virtual machine

		with SCP.
15	Worker	Tear down the virtual machine, according to the KEEP_INSTANCE parameter.
16	Worker	Create results payload.
17	Worker	Copy results payload to the Rabbithole.

If any of the steps described in Table 2 fail, the session is aborted and a notification about it is sent back to the test job. Aborted sessions are reported differently from failed ones, to make it easier to quickly recognize infrastructure-related failures.

7.3.1 Malware Handling

The Controller retrieves malware for the tests from two possible sources: local storage in the Controller file system, or the main malware sample storage server. The used source depends on the contents of the provided configuration file. If the MALWARE_SET parameter is defined, the Controller attempts to find a pre-defined malware set from the Controller's local storage. Alternatively, if the MALWARE_SAMPLES parameter is defined, the Controller will export the requested malware samples from the malware sample storage server.

The malware sample storage server is located in the Orange network, however a mirror server exists in the Red network, directly accessible by the Controller. The sample storage server is accessed and used with a pre-existing Python library, which is not defined in the scope of this study.

The pre-defined malware sample sets in the Controller local storage are stored as tar.gz archives, containing one or more samples. Tar.gz is a common Linux archive format that combines two archiver utilities, tar and Gzip. The sets are created using a dedicated job in Jenkins, which tells the InfraWorker to export defined samples and package them as a new tar.gz archive.

7.4 Virtual Test Environment

The virtual machines used in the testing are run in an OpenStack private cloud environment in the Red network. All the OpenStack services are configured and managed by the in-house network infrastructure team, and only the management of the virtual machine images and instances is done in the scope of this study.

Communication with the OpenStack instance is done through Python-based API clients for two OpenStack services, Nova and Glance. Nova is the virtual machine provisioning service, for creation and lifecycle management of virtual machine instances. Glance is the image service, for uploading and managing virtual machine images. [22]

The provisioning of virtual machines is done by the Controller, in the following sequence:

1. Use Nova Python client to create virtual machine instance, using the virtual image defined in the configuration file.
2. Wait for the virtual machine instance to finish building, and to get the ACTIVE status
3. Get IP address of the virtual machine instance.
4. Assign a floating IP address to virtual machine instance.
5. Assign the virtual machine instance to the correct security group.

The virtual machine images used in this study are KVM/QEMU images, created as raw risk images and converted into the QCOW2 disk image format [23]. Creating a test-ready virtual machine image requires several setup steps, though most of them are only required for Windows-based virtual machines.

When creating a virtual test image of a Windows operating system, the following steps are required:

1. As a requirement from OpenStack, the virtual machine guest operating system needs to have Virtio disk and network drivers installed [24]. This is done by injecting a Virtio driver ISO image into the virtual image creation process, to be selected when the guest operating system is installed.
2. Install guest operating system.
3. Setup automatic logon, so the operating system can boot to desktop without asking for credentials.
4. Disable screensaver, so possible GUI-based tests are not interrupted.
5. Disable power save options, so the operating system does not go into sleep mode during extensive wait periods.
6. Setup Windows paging file. This is required for dumping process memory during crashes. The page file size should be at least the size of the RAM memory, so a full kernel dump can be created.
7. Set Windows to test mode, to accept test-signed binaries.
8. Disable Windows User Access Control (UAC), to remove the need to elevate processes as an administrator user.
9. Install Cygwin [25].
10. Install OpenSSH module in Cygwin, to enable SSH connectivity to the virtual machine [26].
11. Add the Controller public key to the SSH authorized_keys file, to enable passwordless SSH login to the virtual machine.
12. Install Python.
13. If testing an anti-malware product without automatable installation process, preinstall the product.

Linux and OS X-based images only require the following setup steps:

1. Install guest operating system.
2. Install SSH.
3. Add the Controller public key to the SSH authorized_keys file.
4. Install Python.

7.5 Virtual Services

The tests implemented in this study utilize different server-side services in their functionality. As the test environment is isolated from other network environments and the public Internet, these services need to be available in the local cloud environment in the Red network. To remove the maintenance cost of the test system, the choice was made to dynamically deploy these services to the Red network OpenStack environment whenever they are needed, and then removing them when they are no longer used.

The deployment of these services required implementing automation to handle their configuration and management. This automation was implemented in the Controller service. The following sections describe the different services available for the tests, and how they are deployed and used.

7.5.1 Metasploit Framework

Metasploit Framework, as introduced in Chapter 4.2.8, is an exploit testing toolkit and service, available for various operating systems. It is utilized in this study by exploit protection tests, to be introduced in Chapter 7.8.8. In the tests, the Metasploit Framework is used to configure, deploy and deliver exploits to the test machines.

In this study the Linux version of Metasploit was used, primarily due to its ease of installation and management. It is deployed to OpenStack in a Debian Linux virtual machine instance. The installation was done using Debian software packages, known by the .deb file extension, available for download from the Rapid7 website.

To make repeated deployments of Metasploit easier, a Linux Bash shell script was developed to automate the installation. Metasploit installation requires certain other software packages to be available on the system, most notably Ruby and PostgreSQL. The installation of the prerequisite software packages was done using the Linux Advanced Packaging Tool (APT).

Once Metasploit has been installed, it can be used in various ways. Usage through Remote Procedure Calls (RPC) is supported, which provides a good programmatic way of usage for the purposes of test automation. The remote procedure calls are done using the HTTP-protocol, using HTTP POST requests.

7.5.2 Cloud Scan Server

The Cloud Scan Server is an additional component of the anti-malware product used in this study, which provides a cloud-based file scanning service for the clients. The service is used to offload scanning logic and load from clients, to enable building more lightweight end-point clients. Testing the Cloud Scan Server is not in the scope of this study as such, however it needs to be present to enable the full technology stack of the anti-malware product.

The Cloud Scan Server is a Linux-based service, deployed on a Debian Linux virtual machine in OpenStack. Communication with the service is done using the HTTP-protocol. The communications for a service such as this need to encrypted, however the Cloud Scan Server itself does not provide a Transport Layer Security (TLS) or Secure Sockets Layer (SSL) implementation that would enable HTTPS-based communication. The encryption is provided by another service, HAProxy [26], acting as a frontend and load-balancer for the Cloud Scan Server. The HAProxy instance is also dynamically deployed to OpenStack as a Debian Linux virtual machine.

7.5.3 Web Server

To enable testing web-traffic scanning of malicious content, a web server is needed to host and provide the content. As the test system is in an isolated network environment, the web server hosting the malicious files needs to be purpose-built for the tests. As the server needs to be accessible by the test machines, it needs to reside in the same OpenStack cloud environment and network segment.

The web server is deployed to OpenStack as a Debian Linux virtual machine every time the web-traffic scan tests are executed. The web server itself was built using the HTTPS-erver module from the Python standard library. This lightweight and maintenance-free approach was chosen due to the fact that the server does not need to implement any logic beyond hosting an index of files. Also the web server does not need to provide HTTPS-based encrypted communications, as web-traffic scanning cannot scan encrypted content. The usage of the web server is described in detail in Chapter 7.8.6, Web-traffic Scan.

7.5.4 Internet Proxy Server

Though the test system resides in an isolated network environment by design, it needs Internet connectivity in certain operations. For example certain types of malware attempt to connect to a server in the public Internet, and will not function correctly if the connectivity cannot be established.

The Internet connectivity was provided using a proxy server. As the test system deals with real malware, it should attempt to hide itself from malware developers, to deny them the knowledge of where and by whom their malware is being analyzed and tested. As such using a normal proxy server hosted within the organization is not feasible. Instead a Tor-based proxy implementation was employed.

Tor is a network software that attempts to anonymize all Internet traffic passing through it [28]. The Tor-based proxy implementation was provided for this study by a network infrastructure team, and as such its exact implementation and deployment steps are not documented here.

7.6 Client-side Infrastructure

The client-side infrastructure is written with the Python programming language, and contains three core components: bootstrap, test runner and data collector, which are described in detail in the following chapters.

7.6.1 Bootstrap

The bootstrap script is responsible for managing the client-side infrastructure workflow, and for initiating the testing. The execution method of the bootstrap depends on the platform. On Windows, it is placed in the “Startup Programs” system folder by the Controller, causing it to be automatically executed when the operating system starts up. On Linux and OS X, it is executed directly and synchronously by the Controller using SSH.

The bootstrap performs the following sequential operations:

1. Extract the testable artifacts payload archive to the local file system.
2. Activate the Python virtual environment.
3. Execute the test runner script (described in the following chapter.)
4. After testing has concluded, execute the data collector script, described in Chapter 7.6.3, Data Collector.
5. Indicate to the Controller service that testing has concluded, using a status file written to the local file system.

7.6.2 Test Runner and Test Sets

The test runner script manages the test flow by interpreting test set files and executing test cases. The test sets are text-based files that contain a list of sequential instructions, with one row containing one instruction. The instructions can be either framework commands or names of test case files to be executed. Test cases and framework commands in a test set are executed in order until the end of the set is reached, unless a test case marked as critical fails. Table 3 lists the currently implemented framework commands.

Table 3. Framework Commands.

Command	Function
!reboot	Restarts the test machine
!shutdown	Shuts down the test machine
!winxp	Only execute the following command on Windows XP operating system

Additional commands may be added to introduce further control flow logic to the test sets. The name of the test case in a test set can be preceded by a +-sign, which marks the test case as critical. If a critical test case fails, the execution of the remaining instructions in the test set is aborted and the testing session is concluded.

The test runner script logs the progress and results of the executed test set into a log file, using the Logging module found in the Python standard library. This makes it possible to monitor the progress of the test set execution while the tests are running.

7.6.3 Data Collector

The data collector script is responsible for collecting all relevant artifacts from the test machine, and packaging them into a ZIP archive file.

The artifacts include:

- Test results
- Log files created by the software-under-test
- Log files created by the test cases
- Memory/crash dumps generated by the operating system or the software-under-test
- Operating system logs
  - Windows Event Log
  - Linux/OS X SysLog
- Version information of the software-under-test

The resulting archive file is sent through the Rabbithole by the Controller, and stored as a build artifact in the Jenkins test job. The contents of the archive can then be analyzed to investigate test failures and other results.

7.7 Test Cases

The automated test cases, as introduced in Chapter 5.2, were implemented with the Python programming language. The implementation and execution of the test cases utilizes two existing Python modules: the unittest module of the Python standard library, and nose, a third-party test execution module.

Similar to any typical programming class object, a test case has a constructor function that can contain different initialization steps performed before the actual tests are executed. The most common initialization step performed in this system is establishing the logging. This is done by using the Logging module of the Python standard library.

The actual test steps performed in the test case are organized into functions, each named with the prefix “test_”. The naming convention is a requirement of nose, the test execution module, which allows it to discover executable tests from the test class. The test functions are executed in alphabetical order, so naming conventions can be used to control the order in which the tests are executed. The common steps performed by each test function were described in detail in Chapter 5.2.

A test case verifies the result of operations and the correctness of data by using a programming concept called assertion. This is typically done by comparing the value of a variable to a pre-defined reference value. If the values do not match, an assertion error is raised by the test framework, which is reported as a test failure. Figure 5 contains a pseudo-code sample of the typical structure of a test case.

```
from product_library import test_product
from test_library import base_test

class TestSample(base_test.BaseTest):

    def setUp(self):
        self.testProduct = test_product.Product()
        self.startLogging()

    def test_01_test_product_mode_1(self):
        """Test that product works correctly in mode 1."""
        self.testProduct.setMode(1)
        result = self.testProduct.someOperation()
        assert result == True, "Operation failed in mode 1"

    def test_02_test_product_mode_2(self):
        """Test that product works correctly in mode 1."""
        self.testProduct.setMode(2)
        result = self.testProduct.someOperation()
        assert result == True, "Operation failed in mode 2"
```

Figure 5. Sample Test Case Structure.

The actual test coverage provided by the test cases implemented in this study is explained in detail in the following chapter.

7.8 Test Coverage

This chapter provides descriptions of the most important tests initially implemented in this study. As with all test automation systems, the test coverage will evolve and improve over time, so the tests documented here are not the only coverage this system will provide over its lifetime.

7.8.1 Installation and Update

Install tests test the fresh installation of the anti-malware product into a system. This is primarily done using a "silent" install option from the command-line, meaning the product is installed without using the graphical user interface (GUI) or requiring interaction with the installer. This makes the test faster and more reliable compared to a test that would navigate the user interface and interact with the dialogs.

For an anti-malware product, two distinct installation tests are required; one in which the product is installed into a clean machine, with no malware present, and one into a machine that has been infected with malware. Installation into a clean machine is tested in various other tests in the organization, and as such is not required in this test system from a test coverage point-of-view. However, as having an installed and functional product present in the test environment is a prerequisite for all the tests that follow, it needs to be included.

Testing installation into an infected machine shares much of the implementation with the memory scan tests, which are introduced in Chapter 7.8.4. Most anti-malware products run a scan of the system they are being installed into prior finishing the installation, to detect and clean any existing malware infections already present in the system. The scan, in the context of this specific anti-malware product, is a memory scan of the processes running in the system. The mechanics of the scan, and how the malware infection is produced for the test, will be explained in detail in Chapter 7.8.4.

Update tests test the live update of an anti-malware product already installed on a system. In the context of this test system, this refers to updating the anti-malware engines and components, and the virus definition databases, not the product as a whole. As the test environment is located in an isolated network with no connectivity to update servers, the updates have to be delivered to the tests in an offline-installable form. This is a supported use-case for the anti-malware software in question, so no further development is required to implement the support for offline updates.

7.8.2 On-demand Scan

The on-demand scan (ODS) tests test scanning of a collection of malware files on the local disk, on a network drive, or on a portable storage device attached to the system. The malware files do not represent a malware infection as such, as none of them have been executed. The purpose of the test is to verify the detection capabilities of the anti-malware product against a large collection of files representing different file types and malware families. Generally, the on-demand scan test answers the question "can the anti-malware product detect a certain family of malware?"

The usefulness of the on-demand scan test depends heavily on the composition of the malware sample collection used in the test. As explained in Chapter 4.1, using malware samples that are old and no longer prevalent in the real world provides no useful information about the current detection capabilities of the anti-malware product. As the on-demand scan tests scan a large number of malware files and thus trigger a large number of detections in the product, it is the test where the composition of the malware collection is the most important.

The primary verification performed by the on-demand scan test is that each scanned file in the collection triggers a malware detection. The exact method of how the test observes and verifies the detection determines how quick and efficient the tests are. It should be possible to perform the verification programmatically, as using the graphical user interface to verify a large number of detection events is very slow by comparison. In the anti-malware product used in this study, the programmatic approach is supported through an application programming interface (API) to monitor detection events triggered by the product.

The full functional flow of the on-demand scan test is as follows:

1. Disable on-access scanning in the anti-malware product. This ensures the malware files are not scanned while they are being handled, before the intended on-demand scan takes place.
2. Establish a malware file collection in a location accessible by the test machine, primarily the local disk.
3. Configure the anti-malware product to delete infected files.
4. Scan the malware file location using a command-line scanner.
5. Verify that every file in the malware file location triggered a detection event.
6. Verify that every file in the malware file location was deleted.

The on-demand scan test also has coverage for scanning for malware inside archive files. For such a test, the flow, compared to a normal ODS test, is as follows:

1. Same as normal ODS test.
2. Same as normal ODS test.
3. Create a ZIP archive file, and package a number of malware files inside it. The ZIP archive is created using the Python zipfile module, which is a part of Python standard library.
4. Scan the archive file using a command-line scanner.
5. Verify that the archive file triggered a detection event.
6. The anti-malware product in question does not delete infected archive files, to ensure the end-users don't lose the contents of entire archives because of one infected file found inside them. Therefore the infected archive file is not expected to be deleted.

7.8.3 On-access Scan

The primary purpose of on-access scan (OAS) testing is to verify the integration of the anti-malware product into the operating system, and its capability to intercept file events in the file system.

In terms of protecting uses from getting malware on their systems, on-access scanning, along with web-traffic scanning, is probably the most important feature in the anti-malware product. When on-access scanning is enabled and functional, every file that lands on the file system, whether copied or downloaded, is intercepted and scanned for malware. This makes it the last line of defense should other methods of protection fail to block the threat.

The full functional flow of the on-access scan test is as follows:

1. Establish a malware file collection in a location accessible by the test machine.
2. Configure the anti-malware product to quarantine infected files.
3. Copy or move files from the malware file location to a directory on the local disk.
4. Verify that every copy or move operation was intercepted and blocked, by checking that no files landed in the destination directory.
5. Verify that every file copied or moved triggered a detection event.
6. Verify that every detected file was quarantined. As the quarantine location is protected by the anti-malware product, this has to be done by checking the correct action listed in the detection event.

The anti-malware product used in this study does not scan inside archive files in on-access scanning, due to performance considerations. If the contents were scanned, users dealing with large numbers of archive files would have their file operations slowed down significantly. Thus the on-access scan tests do not target archived files as such. Archive files still trigger malware detections though, if the container itself has a detection written for it.

7.8.4 Memory Scan

Memory scan tests are where the actual live malware infections are tested. They are the only tests in the scope of this study where malware is allowed to execute and infect the memory of a computer. As such they are also the most dangerous tests, where malware could potentially escape the test environment. However the design of the test system makes this very unlikely.

Unlike other tests where practically any types and families of malware can be used, memory scan tests require some effort in selecting suitable malware for the tests. Executing a malicious program does not always result in a successful infection, due to various environment-related factors. For example the malware might have been written for a different version of the operating system than what is being used in the tests, and as such might not be compatible with it.

Some malware families also implement counter-measures against malware analysis and testing, by attempting to detect virtual environments [29]. If the malware detects it is being executed in virtual machine, it can skip the infection and deny the analysis or testing, or even perform some destructive actions in the system.

Therefore when selecting malware to be used in the memory scan tests, they have to be first manually tested to verify they can successfully infect the system in the test environment. For this study, the selection contains malware families such as Zeus, ZeroAccess, Kelihos and Tofsee. The exact names of the malware families are often not industry-standard, with each anti-malware vendor and researcher using their own names and naming conventions. As such the names mentioned here might not appear in the context of other anti-malware products.

The memory infection is achieved by running a malicious executable file. On the Windows operating system, which is the primary focus of this study, this most commonly means a Portable Executable (PE) file, with an .exe file extension.

To reliably test the detection and removal of memory infections, the presence of the infection needs to be verified before the memory scan is executed, to ensure that the malware was successfully able to infect the system. Depending on the type of malware, this can be done by observing and comparing changes in the file system, registry and processes in the operating system.

In this study a tool originally developed by Trend Micro Inc., named HijackThis [30], is utilized to take and compare snapshots of the system. The snapshots are a text-based document that lists changes in the file system, running processes and the system registry. The tests use the tool to take three snapshots of the system, at different stages of the test, and compare them against each other to highlight changes. Table 4 lists the different snapshots and their purpose.

Table 4. HijackThis Snapshots.

Stage	Timing	Purpose
1	Before the malware infection	Baseline snapshot of the clean system, before any changes are expected.
2	After the malware infection	Compared to the baseline snapshot, to highlight changes done by the malware.
3	After the memory scan and disinfection	Compared to the baseline snapshot, to verify that changes done by the malware were detected and reverted.

The comparison of the snapshots is done by performing a file comparison operation, commonly known as a "diff", named after the Unix diff command [31]. The diff operation does a line-by-line comparison of two files and creates a file, known as the difference file, listing the differences. In this study the diff operation is performed using the difflib module from the Python standard library.

Once the malware has been executed, and it has been observed to make changes in the system, the memory scan is ready to be performed. In the anti-malware product used in this study, the memory scan scans the processes running in the operating system, looking for code signatures that match known malware.

If the memory scan successfully detects the infection, the anti-malware product is then instructed to attempt disinfection. The result of the disinfection is verified by taking another snapshot of the system, and comparing it to the baseline snapshot of the clean system.

The full functional flow of the memory scan test is as follows:

1. Use HijackThis to take a baseline snapshot of the clean system.
2. Infect the test environment's memory by executing a pre-selected malicious executable file.
3. Verify the infection by taking another HijackThis snapshot, and comparing it to the clean baseline snapshot.
4. Perform a memory scan using a command-line tool.
5. Verify that the process created by the malicious executable was terminated.
6. Verify that the malicious executable file was deleted.
7. Verify the disinfection by taking another HijackThis snapshot, and comparing it to the clean baseline snapshot.

7.8.5 System Scan

As introduced in Chapter 4.2.4, a system scan scans the system folders and registry of the operating system, looking for possible changes done by malicious software. The general term of a system scan can also contain a memory scan, but in the anti-malware product used in this study they are separate features.

To test system scan, the system needs to be prepared with malicious files in certain locations. In the tests implemented in this study, this includes:

- Copying malicious Portable Executable (PE) files to system folders, such as:
  - C:\Windows
  - C:\Windows\System32
  - %APPDATA% (a Windows alias to the Application Data folder)
  - C:\Program Files and C:\Program Files (x86)
- Replacing operating system binaries, such as .DLL files in C:\Windows\System32, with malicious files.
- Creating registry launch points pointing to malicious executables.

Registry launch points tell the operating system which programs to run when the operating system starts up, after the computer has been powered up or restarted.

When a system infection is detected, the anti-malware product used in this study attempts to restore the operating system into a previous state using system restore points, if one is available [32]. This allows restoring operating system binaries that may have been replaced with malware.

To verify that the operating system binaries were restored, they should be compared to a reference after the system scan. This can be done by comparing the hashes of the files to references that were calculated prior to establishing the malware.

The full functional flow of the system scan test is as follows:

1. Create a system restore point.
2. Establish malicious files in various system folders.
3. Create registry launch points pointing to malicious files.
4. Perform a system scan using a command-line tool.
5. Verify that the malicious files that were copied to system folders were removed, and that each file triggered a detection.
6. Verify that the operating system binaries that were replaced with malicious files were restored, by comparing their hash to the original.

7.8.6 Web-traffic Scan

Web-traffic scanning (WTS) testing tests the interception and scanning of file downloads. In the anti-malware product utilized in this study the interception happens on the protocol-level, before the file lands on the disk. The interception is not integrated into web browsers, which means it can be tested with any software capable of initiating HTTP-based downloads.

Not all file types are scanned, to ensure the web-traffic scanning does not slow down the web browsing too significantly. Most notably HTML-files are not scanned, even if they contain embedded scripts that could potentially be malicious.

Testing web-traffic scanning is done by downloading malicious files from a web server and verifying that the file downloads are blocked. In the anti-malware product used in this study a blocked download is indicated with a HTTP 503 (Unauthorized) error code. As the interception is done on the protocol-level with no dependencies to web browsers, the file downloads can be initiated programmatically directly from the Python code in the test case. The download is performed using the urllib2 module from the Python standard library. The deployment of the web server used by the tests was described in detail in Chapter 7.5.3, Web Server.

The full functional flow of the web-traffic scan test is as follows:

1. Deploy a new web server instance to OpenStack.
2. Establish a malware file collection on the web server.
3. Attempt to download the files.
4. Verify that each download attempt was blocked with a HTTP 503 error code.
5. Verify that each download attempt resulted in a detection event.
6. Verify that no files landed on the local disk.

7.8.7 Heuristic Protection

Heuristic protection testing tests the detection and blocking of malware based on behavior instead of existing signature detections in the virus definition database. As such the malware used in the test requires some pre-selection, to ensure it attempts to perform some operations that are monitored by the heuristic protection features in the anti-malware product.

As explained in Chapter 4.2.6., the test is performed simply by executing a malicious executable file and verifying that a detection and block event took place. The verification is done by observing detection events though the anti-malware product's API.

The expected result is that the malware is not allowed to run, but should the heuristic protection of the anti-malware product fail to detect the malware, a successful memory infection can occur. As such the heuristic protection tests bear similar risks as the memory scanning tests, explained in Chapter 7.8.4.

The full functional flow of the heuristic protection test is as follows:

1. Attempt to run a malicious executable.
2. Verify that the execution was blocked (no process was created.)
3. Verify that a heuristic detection event took place.

7.8.8 Exploit Protection

The exploit protection testing implemented in this study explore two distinct scenarios: exploitation of Adobe Flash multimedia file format through a web browser, and exploitation of Microsoft Office documents. They were chosen primarily due to the prevalence of the exploited software in end-user systems.

Both cases of exploitation require extensive preparatory steps in setting up the exploitable software and selecting suitable exploits and malware payloads to be used in the tests. As mentioned in Chapter 4.1.4., exploits targeting popular software are generally fixed quickly by the software vendor, making testing with the latest versions of the software non-viable.

7.8.8.1 Flash Exploit Test

For the Adobe Flash test, an exploit targeting a user-after-free vulnerability in the ActionScript 3 implementation in Adobe Flash Player was chosen [33]. The choice was made based on manual tests evaluating the suitability of different vulnerabilities and exploits. As the vulnerability was fixed in Adobe Flash Player version 18.0.0.194, the tests use an earlier version. The exploitation in the tests occurs through a web browser, however the type and version of web browser used is not relevant and should work with any of them.

The Flash exploit is prepared and delivered using the Metasploit Framework, introduced in Chapter 4.2.8. The deployment and usage of Metasploit was explored in detail in Chapter 7.5.1. The exploit is delivered to the test environment through a web server established by Metasploit, that serves the exploit as a .swf Flash file. The exploitation takes place when the tests access the Flash file URL using a web browser.

The payload used in the tests is non-malicious, and only launches the Windows Script Host console GUI. This approach was chosen due to the prevalence of the usage of Windows Script Host in actual real-world exploitations scenarios.

The Flash exploit test is executed twice; first to verify that the exploit and the delivered payload are functional, and second to verify that the exploit protection functionality in the anti-malware product is able to block them.

The steps of the exploit test are as follows:

1. Prepare the exploit and payload in Metasploit.
2. Start the web server hosting the exploit in Metasploit.
3. Disable exploit protection functionality in the anti-malware product (or disable the anti-malware product as a whole.)
4. Access exploit URL in the tests through a web browser.
5. Verify that the Windows Script Host console (wscript.exe) was launched.

The steps for the exploit protection test are as follows:

1. Prepare the exploit and payload in Metasploit.
2. Start the web server hosting the exploit in Metasploit.
3. Access exploit URL in the tests through a web browser.
4. Verify that an exploit detection event took place.
5. Verify that the Windows Script Host console (wscript.exe) is not running.

The verification of the exploitation, and whether it was successfully blocked, is done by simply checking whether the wscript.exe process is running.

7.8.8.2 Microsoft Office Exploit Test

The Microsoft Office exploitation is done using documents containing malicious macros. The macros use generic functionality available to macros in Microsoft Office documents, and are not, as of this writing, not version-specific. As such any recent version of Microsoft Office should be usable for the tests. For this study, Microsoft Office 2013 was selected, due to its prevalence in end-user environments [34].

To remove the need for static installations of Microsoft Office in virtual machine images, steps to dynamically install it before the tests were implemented. This was done using XML-based configuration files that can be passed to the Microsoft Office installer, which allows unattended installation without using the graphical user interface.

The malicious Microsoft Office documents selected for this study originate from an article by the cyber security firm FireEye, titled "Ghosts in the Endpoint." [35] The article introduces different types of malware not detected by traditional anti-malware products, which received significant publicity in the cyber security community.

In recent versions of Microsoft Office macro execution is blocked by default, unless the user gives a confirmation to allow them. To enable testing the macros automatically, macro execution needs to be explicitly enabled for the tests. This can be done though the Policies-key in the Windows system registry.

The full functional flow of the Microsoft Office exploit test is as follows:

1. Install Microsoft Office.
2. Enable automatic macro execution in Microsoft Office.
3. Open a document containing malicious macros in Microsoft Excel or Word.
4. Verify that an exploit detection event took place.

7.9 Scan Reports

Every test cycle executed in the system produces a scan report. It is a human-readable document presenting different details produced and collected from the tests. The report is implemented as a HTML-document, to be viewed in the web browser.

The scan report contains the following information:

- The type of test that was executed, for example "On-access Scan."
- Name of the product that was tested.
- Version information of the product that was tested, and version information of each individual component of the product.

- Duration of the test, in seconds. This duration refers to the duration of the actual anti-malware operation that was tested, not the duration of the entire test job and all its preparatory steps.
- The size of the malware file set that was used in the test.
- The number of malware detection events that were recorded in the test.
- The number of missed files, that is, the number of files that did not produce a detection event.
- The detection rate, as a percentage. The detection rate formula is as follows: 100 * (1 – (missed_file_count / file_set_size))
- Full listing of the missed files, with file name and path information.
- Full listing of the detection events, with the following information:
  - Full path name of the file that produced the detection.
  - Full name of the malware detection that was triggered.
  - Name of the anti-malware component that detected the malware.

The scan reports can be used to assess the result and details of the malware scans, especially in the event of missed detections. They can also be easily distributed to stake-holders, to share information about malware detection rates in the tests.

8 Evaluation

This chapter evaluates the solution of this thesis against the objectives defined in the requirements analysis, performed as explained in Chapter 2, and any other requirements and considerations that may have emerged during the development of the solution.

The technologies used to implement the solution conform to the technologies and conventions already utilized in the commissioning organization, the most significant of which are the Python programming language and the Jenkins automation server. The solution was built on top of the OpenStack cloud virtualization platform, which was already utilized and available in the organization. Also the Python modules utilized in programming the infrastructure and test cases were selected primarily based on prior usage in the organization.

As per the requirements, the test coverage implemented in the completed solution covers all the major anti-malware features of the software-under-test, which included: on-demand scanning, on-access scanning, memory scanning, system scanning, web-traffic scanning, heuristic protection and exploit protection. However the test coverage implemented in this thesis is only the initial baseline, which will continue to evolve as the system is adopted and utilized by the organization that commissioned it.

8.1 Reliability and Efficiency Evaluation

The reliability analysis of the solution was performed by monitoring the following metrics over the course of one calendar month:

- Execution time of the tests
- Consistency of the results
- Number and nature of incidents that required human action to resolve

From the collected data, execution time and consistency evaluation was performed.

8.1.1 Execution Time Analysis

To evaluate the execution time of the tests, one first needs to break down the factors that contribute to the full execution time of a test job. It was already known during the design phases of the solution that the architecture of the system will have a significant impact on the execution time. This is primarily caused by the fact that the virtual test environment is located in a separate network environment, not directly accessible from the test jobs. The biggest single factor increasing the execution time is the Rabbithole channel between the Green and Red networks. A typical payload archive, with a size of 200 megabytes, takes between one and two minutes to pass through the channel. Also polling for payloads on either side of the channel has a small delay, which is multiplied for each payload moving through the system.

The second major factor affecting the execution time is the provisioning of the virtual test environment in OpenStack. It takes on average 1 minute for OpenStack to build the virtual instance, and 1-2 minutes for the guest operating system in the virtual machine to boot up and become usable. Deleting the virtual machine instance after testing takes an additional 10 to 20 seconds.

Certain operations in the Controller service also take time, including:

- Unpacking the payload archive.
- Copying the tests and testable artifacts to the virtual machine instance.
- Copying the malware samples to the virtual machine instance.
- Copying test results, logs and other artifacts from the virtual machine instance after testing.
- Creating the response payload archive.

On average these operations combined take an additional 5 minutes, depending on the size of the payload archive and the size of the malware set used in the tests. Altogether the infrastructure of the system adds an average of 10 minutes to the execution time of the test jobs. This, considering the requirements set for the infrastructure, is an acceptable overhead.

From the test cases that do not deal with scanning malware, the tests for the installation or upgrade of the anti-malware product are the most time-consuming, taking up to several minutes. This is caused by the installation process of the software, and as such cannot be reduced or optimized in the tests. As running these tests is the prerequisite step for all the tests that follow, the time consumed is acceptable.

The execution time of the malware-related test cases depends significantly on the size of the malware sample set used. With a small sample set the average execution time can be as low as a few seconds, which increases linearly up to several minutes as the size of the malware set increases to up to several gigabytes of samples.

Certain test cases can require restarting the guest operating system, which in itself can take several minutes, depending on the speed of the virtual machine and the current load on the cloud virtualization platform. As such, the number of restarts required are minimized by running the test cases such that they share a restart, rather than allowing each test case to restart the system individually.

Overall no tests cases were found to have an unreasonably high execution time. The exact execution times are not always repeatable, as in a shared virtual cloud platform such as OpenStack the amount of virtual machine instances running on the platform, and their resource usage, can affect the performance of the test virtual machines and therefore the speed of the tests.

Table 5 shows the execution time of a typical test job, with all the intermediate steps included.

Table 5: Test Run Execution Times.

#	Step	Avg. execution time (seconds)
1	Collect testable artifacts for the payload archive	60
2	Create payload archive and send it to Rabbithole channel	40
3	Rabbithole file transfer	90
4	Retrieve payload archive from the Rabbithole channel, and extract it	20
5	Provision the virtual machine instance, and wait for guest operating system to start up	60

6	Copy artifacts and malware samples to the virtual machine	180
7	Execute test cases	480
8	Copy results and artifacts from the virtual machine	30
9	Delete the virtual machine	15
10	Create response payload archive and send it to Rabbithole channel	40
11	Rabbithole file transfer	90
12	Retrieve response archive from Rabbithole channel, and extract it	20

The combined execution time of the sample test run amounts to 1125 seconds, or just over 18 minutes. This, considering the complexity of the infrastructure, is an acceptable run-time for a test job.

8.1.2 Consistency Evaluation

Consistency of a test automation system primarily refers to the repeatability of the test results, and the absence of random failures in the tests and the test infrastructure. A test automation system that produces results that cannot be trusted, or one that requires extensive and continuous manual maintenance, can end up having a net negative effect on the whole software development process.

A test automation system can be considered fully consistent if test failures only originate from changes or problems in the software-under-test. Test failures caused by bugs or crashes in the software-under-test are naturally acceptable, as one purpose of automated testing is to uncover exactly such defects. A functional change in the software-under-test may change its behavior, and thus invalidate the expected results implemented in the tests. This is an acceptable reason for tests to fail, though the tests should then be promptly fixed to account for the new behavior and make them pass again.

Any failure state in the tests or the test infrastructure that does not reproduce on sequential executions can be considered a random failure. Random failures are the most time-consuming to investigate, as the cause of the failure does not reproduce consistently and thus cannot be easily investigated live during execution. A typical reason for a random failure can be for example some unexpected operation performed by the operating system in the test environment, such as Windows Update downloading and installing updates during the tests.

In the context of this study, the most common random failure that was observed related to archive file corruption during the Rabbithole channel file transfer. The exact cause could not be uncovered, but it is most likely caused by a timing-related race condition in the MPoE service. To mitigate the problem, a retry mechanism was implemented into the Rabbithole functionality. The Rabbithole will keep a copy of the payload archive on the Green network side of the channel. If the Controller in the Red network is unable to extract the payload archive it retrieved from the Rabbithole, it will request the Rabbithole to re-transfer the file from the Green network.

The base metric used in evaluating the consistency of the solution is a test run, which is one execution of one configuration in a test job. This solution implements 21 test jobs in its initial state. Each test job runs on average 4 parallel configurations, each targeting a different operating system type or version. Thus a full execution cycle of all tests in the solution consists of 84 individual test runs. All the test jobs are executed at least once a day, with some of them being executed up to 10 times a day, depending on the number of new available builds of the software-under-test. On a typical day, one can assume that on average 2 executions of each test job takes place. This amounts to an average of 168 test runs on any given day.

Over the monitoring period, the number of failed test runs due to random failures averaged between 0 and 3 per day. Assuming the worst case scenario of 3 failures per day, this results in a 0,0178% failure rate. The test cases themselves were found to be adequately consistent. That is, given no changes in the test environment and the software-under-test, the tests can be expected to pass indefinitely on repeated executions.

Consistency issues observed during the monitoring period primarily relate to the hardware- and network infrastructure of the system. As there are three servers and four network hops between the test job and the test environment, problems in any of them will cause the entire flow to fail. However during the monitoring period, only the Rabbithole channel was found to cause occasional problems. The problems related to the MPoE service, which would occasionally go down and require a manual restart. The frequency of the problem is relatively low though, and is not expected to reproduce more than once a calendar month.

9 Conclusion

The objective of this thesis was to design, implement and deploy into production a system for automated testing of anti-malware products and technologies against real malware sample collections. Before the work commenced, requirements and evaluation criteria for the system were drafted with the stakeholders in the commissioner organization.

The thesis started with background research on malware and anti-malware, with the goal of getting a deeper understanding of the domain, and how to apply automated software testing methodologies in it. Next, a high-level design for the complete system was drafted. Viability analysis was performed on potential technologies to be used for the implementation, and a suitable set of technologies was determined. The thesis successfully discovered a viable technical solution for achieving the objective, and the technical requirements set for the thesis were met successfully. The system was deployed into production, and the value and integrity of the data provided by it were deemed acceptable.

The initial test coverage of the automated tests implemented here is a baseline to be expanded on. As with any test automation system, further tests will need to be written to cover new features and functionality in the software-under-test. In the context of anti-malware testing, new tests will also need to be written to cover potential new malware threats that are discovered.

The project was completed within the agreed time frame, and further development and maintenance of the system were handed over to the commissioner organization. Potential areas of further development were discussed and drafted, but, to keep this thesis within the agreed limits, were not included in this report.

References

1 AV-TEST Institute. *About the Institute* (2017). **https://www.av-test.org/en/about-the-institute/** (Accessed Jan 9, 2017)

2 AV-Comparatives. *About us* (2017). **https://www.av-comparatives.org/about-us/** (Accessed Jan 9, 2017)

3 Kock, Ned. Information Systems Action Research (2007). p. 5

4 Kock, Ned. Information Systems Action Research (2007). p. 46

5 Agile Alliance. *Information Radiators* (2015). **https://www.agilealliance.org/glossary/information-radiators** (Accessed Jan 21, 2017).

6 Sikorski, Honig. *Practical Malware Analysis* (2012). p. 221

7 Özkan, Serkan. T*op 50 Products By Total Number of Distinct Vulnerabilities* (2016). **https://www.cvedetails.com/top-50-products.php?year=2016** (Accessed Feb 5, 2017)

8 Sikorski, Honig. *Practical Malware Analysis* (2012). p. 241

9 Sikorski, Honig. *Practical Malware Analysis* (2012). p. 243

10 Kennedy, O'Gorman, Kearns, Aharoni. *Metasploit, The Penetration Tester's Guide* (2011). p. xxii

11 Lynch, Vincent. *Re-Hashed: The Difference Between SHA-1, SHA-2 and SHA-256 Hash Algorithms* (2017). **https://www.thesslstore.com/blog/difference-sha-1-sha-2-sha-256-hash-algorithms/** (Accessed Sep 2, 2017)

12 Homes, Bernard. *Fundamentals of Software Testing* (2011). p. 1-3

13 Homes, Bernard. *Fundamentals of Software Testing* (2011). p. 9

14 Tutorials Point. *Test Case* (2017). **https://www.tutorialspoint.com/software_testing_dictionary/test_case.htm** (Accessed Mar 12, 2017)

15 European Expert Group for IT-Security. *Anti-Malware Testfile* (2017). **http://www.eicar.org/86-0-Intended-use.html** (Accessed Feb 19, 2017)

16 The Jenkins Project. *Jenkins User Documentation* (2017). **https://jenkins.io/** (Accessed Jun 4, 2017)

17 The Jenkins Project. *MIT License* (2011). **https://jenkins.io/license/** (Accessed Jun 4, 2017)

18 The OpenStack Foundation. *The OpenStack Foundation* (2017). **https://www.openstack.org/foundation/** (Accessed Mar 12, 2017)

19 SSH Communications Security. *SSH Secure Shell* (2017). **https://www.ssh.com/ssh/** (Accessed Jul 16, 2017)

20 Python Software Foundation. Python Software Foundation (2017). **https://www.python.org/psf/** (Accessed Sep 2, 2017)

21 Koutoupis, Petros. *Mastering ATA over Ethernet* (2017). **http://www.linuxjournal.com/content/mastering-ata-over-ethernet** (Accessed Jul 16, 2017)

22 Rhoton, John. *Discover OpenStack, The Compute components Glance and Nova* (2013). **https://www.ibm.com/developerworks/cloud/library/cl-openstack-nova-glance/cl-openstack-nova-glance-pdf.pdf** (Accessed Oct 14, 2017)

23 McLoughlin, Mark. *The QCOW2 Image Format* (2008). **https://people.gnome.org/~markmc/qcow-image-format.html** (Accessed Feb 6, 2017)

24 Jones, M. *Virtio: An I/O virtualization framework for Linux* (2010). **https://www.ibm.com/developerworks/library/l-virtio/index.html** (Accessed Feb 6, 2017)

25 Cygnus Solutions, Red Hat. *Cygwin* (2017). **https://www.cygwin.com/** (Accessed Jan 21, 2017)

26 OpenBSD Foundation. *OpenSSH* (2017). **https://www.openssh.com/** (Accessed Jan 21, 2017)

27 Tarreau, Willy. *HAProxy – The Reliable, High Performance TCP/HTTP Load Balancer* (2017). **http://www.haproxy.org/#desc** (Accessed Feb 25, 2017)

28 The Tor Network: *Tor Overview* (2017). **https://www.torproject.org/about/overview.html.en** (Accessed Oct 7, 2017)

29 Sikorski, Honig. *Practical Malware Analysis* (2012). p. 369

30 Sourceforge. *HijackThis* (2017). **https://sourceforge.net/projects/hjt/** (Accessed Feb 5, 2017)

31 Tutorials Point. *diff – Unix, Linux Command* (2017). **https://www.tutorialspoint.com/unix_commands/diff.htm** (Accessed Sep 16, 2017)

32 Microsoft Corporation. *Back up and restore your PC* (2016). **https://support.microsoft.com/en-us/help/17127/windows-back-up-restore** (Accessed May 14, 2017)

33 Rapid7. *Vulnerability & Exploit Database* (2017). **https://www.rapid7.com/db/modules/exploit/multi/browser/adobe_flash_hacking_team_uaf** (Accessed Jan 14, 2017)

34 Weyne, Felix. *Analyzing Malicious Office Documents* (2016). **https://www.uperesia.com/analyzing-malicious-office-documents** (Accessed Sep 2, 2017)

35 FireEye. *Ghosts in the Endpoint* (2016). **https://www.fireeye.com/blog/threat-research/2016/04/ghosts_in_the_endpoi.html** (Accessed Dec 28, 2016)