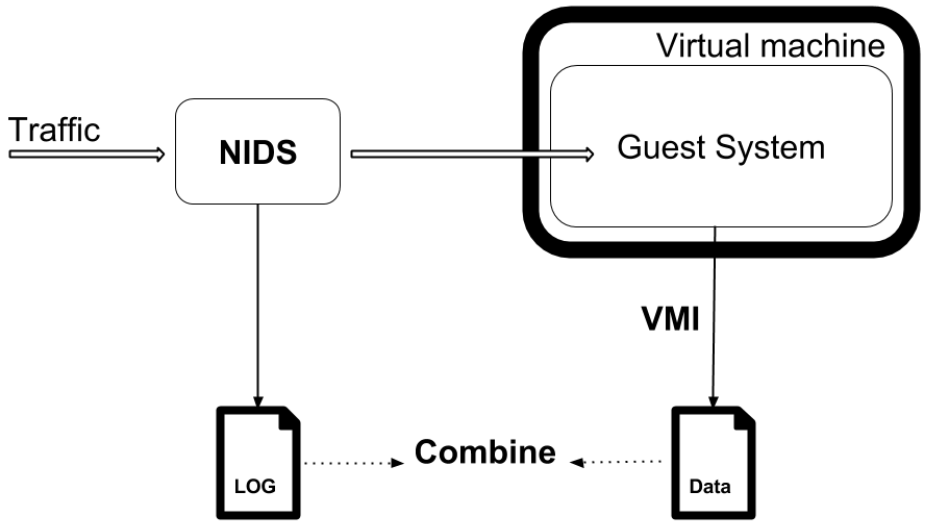

Combining Virtual Machine Introspection with Network-Based Intrusion Detection Systems

Towards a more Secure Environment for Cloud Applications

Master's thesis in Computer Systems and Networks

JULIA GUSTAFSSON

MAHBOOBEH DAFTARI

Department of Computer Systems and Networks
CHALMERS UNIVERSITY OF TECHNOLOGY UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

Master's thesis 2016:NN

Combining Virtual Machine Introspection with Network-Based Intrusion Detection Systems

Towards a more Secure Environment for Cloud Applications

JULIA GUSTAFSSON

MAHBOOBEH DAFTARI

Department of Computer Science and Engineering.
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden 2016

Combining Virtual Machine Introspection with Network-Based Intrusion Detection Systems
Towards a more Secure Environment for Cloud Applications
JULIA GUSTAFSSON MAHBOOBEH DAFTARI

Supervisor: Magnus Almgren, Department of Computer Science and Engineering
Examiner: Vincenzo Gulisano, Department of Computer Science and Engineering

Master's Thesis 2016:NN
Department of Computer Science and Engineering.
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An overview over a combined system of a network-based intrusion detection system (NIDS) and a guest system running in a virtual machine. The dotted arrows show the combining of the logged data from the NIDS and the data gathered from virtual machine introspection (VMI).

June, Sweden 2016

Combining Virtual Machine Introspection with Network-Based Intrusion Detection Systems
Towards a more Secure Environment for Cloud Applications
JULIA GUSTAFSSON
MAHBOOBEH DAFTARI
Department of Computer Systems and Networks
Chalmers University of Technology and University of Gothenburg

Abstract

An increasing number of systems are running as guest systems in virtual machines, for example, applications are moving to be running in the cloud. As the number of cyber attacks is rising, there is a need for a more secure environment. Virtual machines have the advantage that it is possible to inspect the content of the guest systems, called virtual machine introspection. This thesis aims to investigate a new way of securing systems - by combining virtual machine introspection and network-based intrusion detection systems.

Network-based intrusion detection system can inspect the content of the network packets going to all the systems in a network in real-time, they quickly can detect potential attacks. However, network-based intrusion detection systems have problems with false-positive alarms and to discover zero-day exploits. However, by combing virtual machine introspection with a network-based intrusion detection system the data from the virtual machine introspection could be used to provide more information about potential attacks and improve the network-based intrusion detection system at the same time. The goal of this thesis is to investigate how virtual machine introspection could be combined with network-based intrusion detection systems to produce a more secure system. By selecting an application and attacks to test, test cases were performed and data could be gathered from the two systems.

The result showed that several of the attacks was fully detectable by virtual machine introspection. However, the data gathered from the network-based intrusions detection system showed that even if the network-based intrusion detection system could, in this case, detect the chosen attacks, it could not provide any details about the result of the attack. Hence, virtual machine introspection is a great extension to the network-based intrusion detection system. However, a performance analysis of the virtual machine introspection platform was performed, which showed the it has several performance issues. Due to the performance of the platform, we recommend that a combined system should only be used during certain circumstances, such as when the network-based intrusions detection system raises an alert.

Keywords: network-based intrusion detection systems, virtual machine introspection, virtual machine, cloud security, cyber attacks, cloud computing

Acknowledgements

We would like to express our sincere thanks to our supervisor Magnus Almgren, who has guided us continuously throughout this thesis. This thesis would have never been accomplished without his idea to this thesis, thorough knowledge, and valuable advice. Furthermore, we thank our examinator Vincenzo Gulisano, for his valuable input. Besides, we are also grateful to the members of the PANDA community, especially Brendan Dolan-Gavitt, for all help. Last but not least, we would like to send our warm appreciation to our family and friends for their kind support.

Julia Gustafsson, Mahboobeh Daftari, Gothenburg, June 2016

Contents

List of Figures

List of Tables

1
Introduction

In a time where the number of cyber attacks is increasing, there is a need for a more secure environment to protect systems. At the same time, the use of virtual machines are increasing, which have advantages that could be utilized to protect systems. This thesis aims to utilize one of the advantages of virtual machines, virtual machine introspection, and combine it with network-based intrusion detection systems. Network-based intrusion detection systems can be used to monitor the network traffic for all systems in a network. However, network-based intrusion detection systems of today have several issues, such as a high rate of false-positive alarms and not being able to discover zero-day exploits [1]. As network-based intrusion detection systems exclusively monitor the network packets and often are placed isolated from the other systems in the network, they have a very limited view of the monitored systems. However, by running systems as guest systems on a virtual machine, it is possible to utilize a technique called virtual machine introspection in order to inspect the state of the systems. Even though this a great advantage, it comes with a penalty in performance. This may restrict the usage of virtual machine introspection. The aim of this thesis is to investigate how virtual machine introspection could be combined with network-based intrusion detection systems, in order to provide a more secure full system.

1.1 Background and Motivation

An increasing amount of systems and applications are running as guest systems on virtual machines, such as servers in cloud data centers [2]. Examples of cloud data centers are Amazon Web Services, Google Cloud Platform and Microsoft Azure [3], [4], [5].

Virtual machines have several advantages from a security perspective, such as that the guest systems can be isolated from the underlying host system, which means that the host system will not be affected if the guest system is compromised. Besides, virtual machines have the functionality of taking a snapshot of the guest system state at a certain point and later revert the guest system to the snapshot. This means that a compromised system can quickly be reverted to a normal operating state again, which aids the system availability. Furthermore, by running systems isolated in virtual machines, it is possible to perform virtual machine introspection, i.e. inspecting the content of the guest system without its knowledge or permission. This technique can be used for increasing the security of the guest systems, which is

an important field of application as the number of cyber attacks in the society are increasing and there exist security issues in cloud environments [6].

Virtual machine introspection has been used for developing security applications, such as host-based intrusion detection systems that can run outside the system they are monitoring, which means that they are not affected if the monitored system is compromised [7], [8]. However, as a host-based intrusion detection system is built for running on a specific system, it can only monitor one system at a time [9]. There exist another kind of intrusion detection system, network-based intrusion detection systems. Network-based intrusion detection systems monitor the network packets in real time to discover potential attacks and one network-based intrusion detection system can basically monitor all systems in a network at the same time [9]. Hence, the network-based intrusion detection systems are more beneficial for protecting larger system environment, such as cloud server environments.

There exist two different techniques to detect malicious network traffic; signature-based detection and anomaly-based detection [9]. Network-based intrusion detection systems that are using signature-based detection techniques are called signature-based and the network-based intrusion detection systems using anomaly detection techniques are called anomaly-based. Signature-based network-based intrusion detection systems monitor the network traffic by comparing signatures of known attacks and raise an alarm if they find a signature of any of the known attacks [9], [10]. Anomaly-based network-based intrusion detection systems monitor network traffic by observing if there are any deviation in the traffic compared to the defined normal traffic patterns [9], [10].

The main issue with network-based intrusion detection systems is the problem with raising accurate alarms. If a network-based intrusion detection system is using signature-based techniques for discovering potential attacks it will not discover zero-day exploits due to non-existing signatures. However, if the network-based intrusion detection system is using anomaly-based techniques, it might detect zero-day attacks, but it will have a rather high false-positive rate of alarms. Many false alarms decrease the credibility of the network-based intrusion detection system, and an attack might be missed due to the sheer number of false alarms. Network-based intrusion detection systems are often placed isolated from the other systems on the network as they monitor the network traffic. This means that a network-based intrusion detection system is not affected if one or more systems on the network is compromised. However, it does not have access to the internal part of the systems on the network, so the network-based intrusion detection systems can only analyze the content of the network packets and can not evaluate the actual result of the network packets in the systems. This makes it difficult for a network-based intrusion detection system to give accurate and informative alerts. Due to the many existing vulnerabilities and the occurrence of zero-day exploitations, network-based intrusion detection systems can not detect all existing attacks.

This thesis aims to investigate in how virtual machine introspection could be com-

bined with network-based intrusion detection systems, in order to investigate in how such a combined system could increase the security of the monitored guest system. The systems complement each other as network-based intrusion detection systems can discover potential attacks, while virtual machine introspection can be used to extract data about the state of the guest system. This should make it possible to gather more details about the potential attacks and dismiss false-positive alerts. Besides, by using virtual machine introspection, attacks that were not discovered previously could potentially be detected, and the network-based intrusion detection system could be improved. This would increase the accuracy of the network-based intrusion detection system, which improves the security of all the systems in the monitored network. In the future, by combining network-based intrusion detection systems and virtual machine introspection, it might be possible to instantly update the network-based intrusion detection system by continuously analyzing the produced data by the virtual machine introspection. This thesis aims to lay the foundation of research towards such a combined system of virtual machine introspection and network-based intrusion detection systems. Figure 1.1 demonstrates an overview over such a system, where the dotted lines show the potential field of application of combining the two systems.

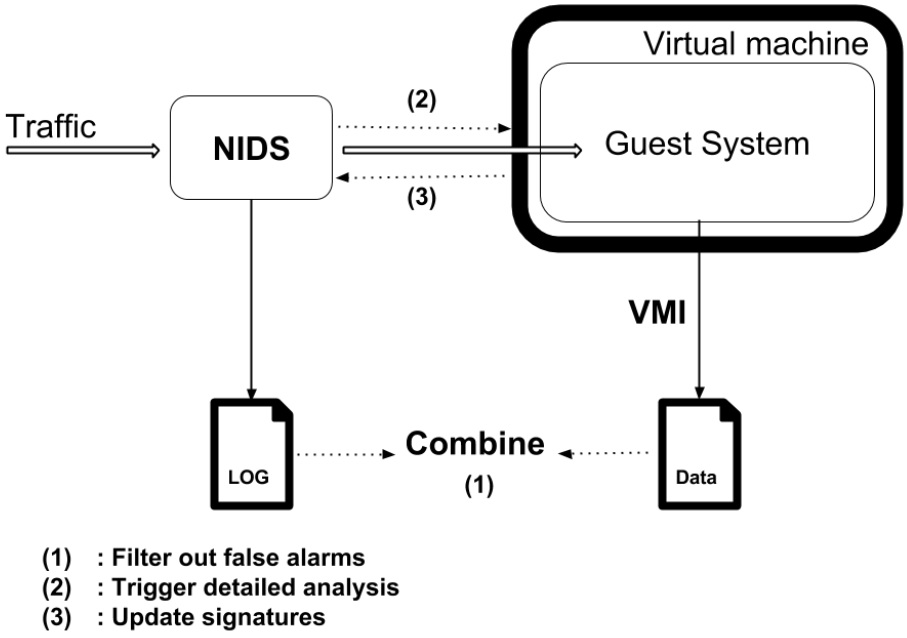

Figure 1.1: Overview over a combined system of a network-based intrusion detection system (NIDS) and virtual machine introspection (VMI).

1.2 Goals and Research Questions

The goals of this thesis are to investigate in how virtual machine introspection can provide a broader view of the monitored systems to state-of-the-art network-based

intrusion detection systems of today and how network-based intrusion detection systems and virtual machine introspection can be combined.

This goal can be divided into several concrete research questions:

- The first research question is if it is beneficial to use virtual machine introspection as an extension to a network-based intrusion detection system.
- The second question relates to what kind of information about the state of monitored systems and their applications that can be gathered using virtual machine introspection.
- The third research question relates to how and when these systems should be combined.

1.3 Limitations

The focus of this thesis is to explore the possibilities of combining data from network-based intrusion detection systems and virtual machine introspection. Hence, this thesis does not aim to build a combined system, but rather to analyze the combined data from an existing network-based intrusion detection system and the data that is possible to extract from a state-of-the-art platform for performing virtual machine introspection. Lastly, due to time constraints this thesis will focus on a few interesting test cases and do not aim to provide a generic study for all kind of vulnerabilities.

1.4 Thesis Organization

The organization of the thesis is presented in chronological order below.

Chapter 1 An introduction to the topic of this thesis is presented.
Chapter 2 Related work in the area is presented. Prior work within the area of combining virtual machine introspection and intrusion detection system, as well as related work for performing virtual machine introspection with the aim to provide high-level state information are presented.
Chapter 3 The necessary background about virtual machines, virtual machine introspection and network-based intrusion detection systems is presented. Also, the utilized tools are presented for enabling further discussion about why these tools were chosen.
Chapter 4 Surveys of the chosen platforms for virtual machine introspection and network-based intrusion detection are presented. Besides, the platforms for this thesis are discussed.
Chapter 5 Experiments and the evaluation of the experiments are presented.
Chapter 6 The system setup for performing the test cases is presented in detail.
Chapter 7 Results of test cases are presented and discussed.
Chapter 8 Conclusions of the thesis and future work within the area are presented and discussed.

2
Related work

This chapter presents related work in the area of combining virtual machine introspection (VMI) and intrusion detection systems (IDSes). It also includes papers presenting relevant techniques and systems for gaining useful information from VMI. There exist several papers about combining VMI and IDSes. However, none of these papers have been focusing on how VMI could be combined with network-based intrusion detection systems (NIDSes), to the authors best knowledge. Instead, the focus of the papers is how host-based intrusion detection systems (HIDSes) could be secured by utilizing VMI. The papers about combining HIDS and VMI are presented in Section 2.1. During the last years, some systems have been presented, which perform VMI and provide high-level information. Such systems are utilized to obtain important data for the analysis in this thesis. Section 2.2 presents the research and existing systems for performing VMI. Lastly, Section 2.3 presents a paper discussing how forensics memory analysis (FMA) can be used in combination with VMI for providing high-level state information about the system.

2.1 Systems combining Virtual Machine Introspection and Intrusion Detection Systems

There has been some research in combining VMI and HIDSes, in order to preserve a HIDS's view of the system state while running the HIDS outside of the monitored system. The first paper to describe a system combining VMI and IDS was presented by Garfinkel and Rosenblum [7]. They presented an architecture where a system was running in a virtual machine (VM) and where the system was monitored by a HIDS that was running in another VM. This was the first system using introspection in conjunction with an IDS. This setup makes it harder for an attacker to compromise the HIDS during an attack of the system. By using VMI it is possible for the HIDS to keep the view of the monitored system's state, even if the HIDS and the monitored system is running in isolated guest systems. Laureano et al. presented a similar system as Garfinkel and Rosenblum presented, where the system to be monitored was running in a VM, but instead of having the HIDS on another VM it was placed on the host system [8]. The HIDS could operate in the anomaly-based fashion by using information about the executed system calls that could be collected from the VM monitor. While the presented papers are utilizing VMI for securing HIDSes, this thesis aims to investigate in how VMI could provide NIDSes with a broader view of the monitored systems as well as investigate if these systems could be combined.

2.2 Virtual Machine Introspection Systems Applicable for Security Applications

There are several studies that investigate how to obtain useful data from virtual machine introspection, which could be utilized by security applications to analyze the state of the monitored system.

Garfinkel and Rosenblum presented a system called 'Livewire', which runs on top of WMware's virtual machine monitor (VMM) to detect attacks in Linux operating systems (OSes) by comparing data from VMI with data obtained by running Linux commands in a remote shell [7], [11].

Jiang et al. presented 'WMwatcher' for performing VMI in Linux OSes that utilizes a technique called guest view casting to reconstruct the view of the OS from the outside [11], [12].

Payne et al. presented 'XenAccess', which is a VMI library for Xen VMM [11]. It could be used without changing the VMM or the VM, and could monitor both Linux and Windows OSes [11], [13]. This makes it possible to perform VMI and virtual disk monitoring [11], [13]. Payne et al. presented another extended VMI library to XenAccess, called 'LibVMI' that could be used for the VMMs XEN and KVM [11]. The library can access physical and virtual addresses as well as kernel symbols [14]. It also allows for utilizing the forensics memory analysis tools Volatility for analyzing the memory, by a physical memory snapshot [11].

Dolan-Gavitt et al. presented a system called 'Virtuoso', which automatically can create tools to obtain useful introspection data from a system [15]. The tools could be used by security applications such as intrusion detection systems. Furthermore, Dolan-Gavitt et al. presented a technique to extract an executable binary file of the entire system [11]. This makes introspection programs more reliable as no heavy reverse engineering for a specific operating system is necessary [11]. Furthermore, another system for virtual machine introspection called 'PANDA', 'a Platform for Architecture-Neutral Dynamic Analysis', was presented by Dolan-Gavitt et al. [16]. In their paper they described the entire PANDA platform as well as several of its introspection plugins and systems. Besides, they showed how PANDA can be used for performing vulnerability analysis of specific programs. In relation to PANDA, Dolan-Gavitt et al. presented a system called 'Tappan Zee (north) Bridge' [17]. The system uses several techniques of mining memory operations made by the operating system, in order to automatically extract useful introspection information about user-level applications, by utilizing active monitoring [17]. Virtuoso and other similar systems focus on kernel-level information, while the most important data for security application is user-level information. By mining memory operations, it is possible to find convenient points to perform active monitoring, which is called tap points in their paper. The system runs on top of the PANDA platform.

This thesis utilizes such implemented systems as these presented above for perform-

ing virtual machine introspection to obtain useful information, in order to gain data for the analysis if such systems could be used in conjunction with network-based intrusion detection systems.

2.3 Forensics Memory Analysis's applicability for Virtual Machine Introspection

While VMI has problem extracting useful high-level information, another technique called FMA have techniques for gathering high-level information about the internal state of a system. This is useful for detecting attacks against the system etcetera. Dolan-Gavitt et al. presented how some of the techniques of FMA systems could be applicable for providing high-level state information from VMI [18]. This thesis will utilize FMA to achieve a great high-level view of the system as an extension to the data provided by the VMI.

3
Technical Background

This thesis utilizes techniques of virtual machine introspection (VMI) and network-based intrusion detection systems (NIDSes) in order to analyze the potential benefits of running the techniques in conjunction. The platforms that implement these techniques are used and analyzed, both separately and in combination. That requires in-depth knowledge of the platforms as well as their tools. In the following sections, we present the techniques, platforms, and vulnerabilities used for this thesis.

3.1 Virtual Machine

There exist two different types of virtual machines (VMs); system virtual machines and process virtual machines. A system virtual machine emulates a real machine, hence complete operating systems (OSes) can operate on such a VM. A process virtual machine is an application that supports the execution of a single application, which allows for cross-platform execution of platform-specific applications. This thesis will focus on system virtual machines and when referring to virtual machines, we mean the system virtual machines. A VM is a program running on a host system, which emulates hardware and allows different operating systems to run as guest systems. The guest systems are operating in the same way as any other operating system on a physical machine. The hardware devices of the VM are virtual and mapped in a certain way to the host system hardware. The state of a VM can be saved by taking a snapshot of the guest system. Later, the system can be reverted to the snapshot and system state will be reverted to the same state as when the snapshot was taken. Then, the guest system can continue to operate normally, ensuring quick recovery and thus, remain good availability.

3.1.1 Virtual Machine Introspection

Virtual machine introspection (VMI) is the technique for inspecting the content of a VM from the outside in order to analyze the guest system running inside the VM. This is done without the guest system's knowledge or permission. There exist mainly two types of VMI techniques, either monitor the VM from the virtual machine monitor (VMM) level or from a privileged VM [11]. The VMM technique is tamper resistant as it is isolated from the guest system and can inspect any OS activity. However, the VMM has a low-level view of the monitored guest system and it accesses the information about the guest system's CPU register, memory and devices as raw bytes [11]. As the useful information is high-level information, such as OS

abstraction, processes running in the system and information about files et cetera, this is called the semantic gap problem [17], [11]. In order to provide high-level state information about the monitored system, which is called bridging the semantic gap, additional VMI techniques are needed to extract high-level information [11].

There exist different techniques of performing VMI in order to bridge the semantic gap; in-VM, two different types of out-of-VM or a hybrid of the previous techniques [11]. The in-VM technique is based on an agent running within the VM that provides the VMM with information. This bridges the semantic gap, but at the same time, it breaks the isolation between the VM and the VMM [11]. The second technique, out-of-VM, VMI is performed by the VMM by utilizing information from the OS source code, or the underlying hardware architecture [11].

There have been several papers suggesting different usage of VMI. Hebbal et al. presented a survey of the applications of VMI [11]. VMI could, for example, be used for intrusion detection, intrusion prevention, malware analysis, memory forensics and for user-level applications introspection [11]. VMI could be used in two different modes, active or passive monitoring [11]. When using active monitoring there need to be so called hooks placed in the security-critical part in the monitored VM, to discover attacks [11]. Passive monitoring is when the system is externally scanned [11].

Next section, will present the powerful platform PANDA, which is built upon a VM named QEMU and can perform dynamic analysis of the entire guest system [16]. Additionally, it has functionality for gaining information about user-level applications [16], [17].

3.1.2 PANDA - a Platform for Architecture-Neutral Dynamic Analysis

PANDA - a Platform for Architecture-Neutral Dynamic Analysis - is a relatively new open-source platform for performing dynamic analysis of an entire system. Currently, it is only officially working on Debian 7/8 and Ubuntu 14.04. However, it is under constant development by a small team of researchers. PANDA is built upon the QEMU system emulator version 1.0.1. PANDA has several features for performing dynamic analysis, including record and replay functionality, Low Level Virtual Machine (LLVM) features and a plugin architecture [16]. By recording an execution, it is possible to analyze the execution by replaying the recording and use plugins for specifying what events to log for further investigation [16]. There exist several plugins in PANDA and new plugins can be added to it. Some of these plugins rely on some concepts used by PANDA, like tap points and the pandalog file format. A tap point is a triple of the caller, program counter and the physical address space. It defines a point in the system, and can be used for performing memory access introspection at that specific point [17]. The pandalog format is a fast and small log format, that is programmatically flexible to read and use [19].

3.1.2.1 The Virtual Machine QEMU

PANDA is built upon version 1.0.1 of the QEMU system emulator, which supports over ten different CPU architectures [17]. QEMU translates all instructions to an intermediate language (IL) for using Tiny Code Generator (TCG) as a back-end code generator [17]. If LLVM is not enabled in PANDA, the QEMU's Just-In-Time (JIT) compiler generates the code and executes it, while when LVM is enabled, the TCG is translated by PANDA into LLVM code [16]. This is done using a module from S2E. Then the code is generated and executed by the LLVM JIT instead of the QEMU's JIT compiler [16].

3.1.2.2 Record and Replay Functionality

PANDA has the ability of recording an execution, which is stored on the disk. The recording can later be replayed to perform dynamic analysis of the execution. This works on guest system of these three architecture; x86, x86_64, and ARM. The record and replay functionality is deterministic, which means that the non-deterministic input to the system during a recording is captured in order to allow a deterministic replay [16]. This includes network packets, mouse and keyboard input among other things. The replay can be repeated as many times as necessary. The record and replay functionality does not support the possibility of interfering or interacting in any way with the recording, i.e. "go live", except from performing analysis with the PANDA plugins [16]. PANDA is slower than standard QEMU, and it has been observed that a replay is about four times slower compared to standard QEMU, while the recording is two times slower [16]. This is however, considered beneficial compared to how slow the system would be if PANDA's sophisticated analysis were run in real-time [16].

During the recording, PANDA first creates a snapshot of the system's state, which includes the registers as well as the memory [16]. Second, PANDA writes three different kind of non-deterministic inputs to a log file [16]. The inputs are the data sent to the CPU as input, RAM read and writes as well as hardware interrupts. Along with them, it records the trace points, which are triples consisting of the program counter, the instruction count since the beginning of the record, and the implicit loop variable [16]. Together, the snapshot and the non-deterministic data along with the trace points make it possible to perform repeatable replays of the execution.

3.1.2.3 Plugin Architecture for Performing Analysis

In PANDA it is possible to analyze the recordings made with PANDA's record and replay functionality using plugins. There exist several plugins within PANDA, and new plugins can be added to PANDA's plugin architecture. PANDA supports plugin-plugin-interaction, which means that plugins can be used in combination with each other. Plugins are written in C or C++ [17]. The plugins are tracking specific events in the system that are specified within the plugin's initialization function [16]. Plugins can track different parts of the execution, from when the

guest code is translated and executed as well as the loads and stores to the memory, typed command line arguments and track the program counter [16]. PANDA also has the ability to save raw memory snapshots that can be used for further analysis with Volatility, which is presented in Section 3.3.1.

3.1.2.4 Available PANDA plugins

PANDA has a distinct number of existing built-in plugins, which are used for analyzing the executions. These plugins can be fetched from PANDA's Github. However, some of the existing plugins are old versions of newer improved plugins. Also, a few of the plugins are 'helper-plugins', i.e. those plugins are not useful by themselves but are used in conjunction with other plugins to produce data to analyze. In this section, all the latest versions of all well-documented PANDA plugins and their helper-plugins that exist on PANDA's Github are presented in Table 3.1 [20]. 'STUW' is not considered a plugin in PANDA, rather a tool. However, we present it as a plugin due to simplicity. Plugins that are only used to test how other plugins work are not presented. Several of the plugins touch a subject that is not a part of this thesis, such as data tainting. Data tainting is a concept where values are tainted if they are private, and tainting can be used for security assertions.

Table 3.1: A list of all available well-documented PANDA plugins from PANDA's github. The 'Helper'-column describes if a plugin is a helper-plugin or not.

Plugins	**Helper**
asidstory	
bigrams	
bufmon	
callstrack_inst	✓
correlatetaps	
coverage	
dead_data	
file_taint	
fullstack	
ida_taint2	
keyfind	
llvm_trace	
memdump	
memsavep	
memsnap	
memstrings	
network	
osi	✓
osi_linux	✓
osi_winxpsp3x86	
printstack	
rehosting	
replaymovie	
STUW	
stringsearch	
scissors	
syscalls2	✓
taint2	✓
tainted_branch	
tainted_instr	
tapindex	
textprinter	
textprinter_fast	
tstringsearch	
unigram	
useafterfree	
win7proc	
win7x86intro	✓

We will now briefly describe the plugins listed in Table 3.1. This section summarizes the plugins existing on PANDA's github [21].

- **asidstory** produces output to a file in the current directory containing information about the processes in the replay and the instruction range when they were active. The information provided about a certain process is its process identifier (PID), process name and address space identifier (ASID). In Linux-based recordings the provided ASID is the virtual address and in Windows-based recordings the given ASID is the physical address.

- **bigrams** writes output to the terminal including statistics for all tap points in the replay. The statistics are based on the memory writes and could be used for clustering the tap points.

- **bufmon** is used for tracking buffers and the memory accesses to the buffers during a replay. The buffers is given as input to the plugin via a text file including necessary information about the buffers. The plugin produces an output file including in-depth information about the memory accesses as well as tap points.

- **callstrack_instr** is a helper-plugin that is only used with other plugins. It is tracking function calls and returns.

- **correlatetaps** produces a binary file, where each row includes two tap points and the number of times that they have been writing to the contiguous parts of the memory. By analyzing the output, it is possible to say if different tap points correlate.

- **coverage** writes output to the terminal including a list of all unique basic blocks in the LLVM IR code that was executed by a specific process. The specified process is given as an argument to the plugin.

- **dead_data** measures the "deadness" of the data by performing an analysis of how often tainted data is used to decide branches. This plugin is intended to be used in conjunction with other taint-related plugins.

- **file_taint** is a plugin for tainting a file, which is only supported for Linux-based recordings. It can be used for tainting files in order to, for example, to see if the file is sent out on the network or if the file was decrypted. Some of the tainting analysis might require additional plugins. File_taint uses the introspection plugins for gaining information about file objects as well as syscall2 for gaining information about file-related system calls such as open and read.

- **fullstack** generates the full callstack for the first time each tap point appears in the replay. This information is written to a file. The tap points are specified in a list of tap points.

- **ida_taint** produces information in a pandalog file about process introspection among other things. It can be used for tainting by using additional taint

plugins in conjunction. Also, taint analysis of the output file could be performed with commercial IDA Pro if one is running PANDA on Windows.

- **keyfind** locates the point in memory where the transport layer security's (TLS's) master secrets, i.e. private keys, are generated. The output produced is a text file containing the tap points where the keys were generated and the keys will be printed to standard error.

- **llvm_trace** creates a trace of the LLVM instructions executed during a replay, including the dynamic values as memory operations, a well as the result of operations in LLVM code. This could be saved in two ways. The first way is to save it to log files and one bitcode file with the LLVM bitcode. The log files includes one with the memory values, one with the other dynamic values and one with a list of all LLVM basic blocks. The files can be analyzed with a tool called "dynslice". The other way is to save it to two files; the trace and dynamic values to a TUBTF log file, and another file with the LLVM bitcode.

- **memdump** writes all the content of memory accesses from the given tap points. The output is placed in two binary files, one for reads and one for writes. The plugin operates the best when tapindex is used before.

- **memsavep** takes a snapshot of the memory from the beginning of the replay until a specified percentage of the instructions are executed. The snapshot is a raw memory dump, which can be analyzed with the memory forensic tools such as Volatility.

- **memsnap** dumps memory into a raw memory file once it is encountering a tap point in the replay. The output that is produced is one file with raw memory data from each tap point that was encountered. The output files from this plugin can be analyzed using memory forensic tools like Volatility.

- **memstrings** gives a searchable text file as the output that includes all the printable strings that have been read or written to the memory. The output file includes an indicator of whether the string is an ASCII or a Unicode string, which are the supported formats. Also, the instruction count where the string was read or written to the memory is included.

- **network** creates a PCAP-file of the network traffic from the replay, which can be further analyzed with the program Wireshark [22].

- **osi** is a helper plugin for performing introspection, that is used together with other OS-specific introspection plugins.

- **osi_linux** is a helper-plugin for performing introspection of recordings of Linux systems. It is used with other plugins for performing introspection analysis.

- **osi_winxpsp3x86** is a helper-plugin for performing introspection of recordings of Windows XP SP3 systems. It is used with other plugins for performing introspection analysis.

- **printstack** takes a program counter as an argument and when the program counter is reached in the replay it writes all the called functions, i.e. the stack, to the terminal.

- **rehosting** allows custom architecture to run in PANDA, by allowing execution of specified raw firmware images of 32-bit x86 systems.

- **replaymovie** produces sequences of ppm files, which together with the software 'ffmpeg' can create an mp4-file. Then, it is possible to see what happened on the screen during the execution even if one only has the replay of an execution.

- **scissors** produces a new shorter replay of the given replay. The start and the end instructions of the new replay is specified by the plugin's arguments.

- **STUW** is a tool that can be used for analysis of recordings of 32-bit Windows 7 systems. By first running the Windows-specific introspection plugin 'win7proc' to obtain data containing system calls and specify the output to be put into a 'pandalog'-file, 'STUW' can analyze the output from the 'pandalog'-file. This tool analyses the system calls, and based on those provides an overview of the interprocess-communication.

- **stringsearch** searches for specified strings that were read or written in the memory during a replay. The output consists of a file that contains the tap points where the string was encountered as well as the number of matches for each of the given strings at different tap points. Besides, output to the terminal includes the tap points, the instruction counts and information about which of the strings were matched and whether the match was a read or write to memory.

- **syscalls2** is a helper-plugin for information about the system calls in the system. This plugin is used in conjunction with other plugins to produce useful data.

- **taint2** is a helper-plugin for other plugins that performs taint analysis. This plugin tracks the data flow and can query data as well as label data.

- **tainted_branch** lists all the addresses of the branch instructions that are affected by tainted data. The output is placed in a 'pandalog'-file.

- **tainted_intrs** can, while used in conjunction with the 'file_taint'-plugin, pro-

vide the instructions that handles tainted data from the file given as an argument to the 'file_taint'-plugin. These instructions are written to a 'pandalog'-file.

- **tapindex** produces two different files, one for read and one for write, that are indexes of how many bytes were read and written to the memory at each tap point.

- **textprinter** generates two log files including the data read or written in the memory at the given tap points. These files include entries with information about callstack, program counter, ASID, virtual address, access count, and byte value. The log files are analyzed by first using a script to generate the raw data for each tap point. By the ASID given in the log files it is possible to know which process were reading or writing the specific data, by using the 'asidstory'-plugin.

- **textprinter_fast** is a special version of the 'textprinter'-plugin, but can only be used while a system is running, not during a replay. It only generates data read from the memory and is only working for one specified tap point. It can generate the data for writes if it is reconfigured.

- **tstringsearch** taints a string specified by the plugin 'stringsearch'. The tainting will be performed when the string is involved in memory operations.

- **unigrams** produces histograms by collecting statistics for the memory reads and writes at each of the tap points. The output, which is saved as a histogram, is saved into two files, one for reads and one for writes.

- **useafterfree** was written by the PANDA-developers after analyzing the outcome of PANDA after a use-after-free attack and can detect such an attack. The plugin analyzes low-level memory allocation functions and detects use-after-free attacks when freed memory is dereferenced. This plugin requires arguments to specify virtual addresses for the malloc, free and realloc functions as well as the address space where the use-after-free might happen. This plugin has some problems with false-negatives.

- **win7proc** is a Windows-specific plugin for performing introspection on several specified system calls, related to processes, registry, shared memory, file system and local procedure call. It can only be utilized to analyze recordings of 32-bit Windows systems. The outcome from this plugin includes the system calls made, overview over the processes' states and the data written by the system call "NtWriteFile" during the replay.

- **win7x86intro** is a helper plugin for performing introspection of recordings of executions from 32-bit Windows 7 systems. It is used together with other plugins to perform introspection analysis.

3.2 Network-based Intrusion Detection Systems

Network-based intrusion detection systems monitor the incoming and outgoing network traffic to discover potential attacks against systems on the network. However, unlike other security systems, NIDSes do not apply any defence mechanisms to protect the monitored systems against the attacks. Instead, they raise an alarm when a potential attack is discovered, in order for the administrators to decide on a proper countermeasure. NIDSes also log information about the network packets, which a system administrator can read to get additional information about the network traffic.

The result of a network traffic analysis can be divided into four categories; false-positive, false-negative, true-positive or true-negative. True-positive means that the NIDS raised an alarm when there actually was an attack, while false-positive means that the NIDS raised an alarm when there was no attack. True-negative means that there was no attack and the NIDS did not raise any alarm. False-negative means that the NIDS did not raise an alarm, however, in this case there was an actual attack that the NIDS missed.

3.2.1 Techniques for Discovering Attacks

There exist mainly two different techniques for discovering attacks in network traffic; signature detection and anomaly detection [9]. Signature-based NIDSes monitor the network traffic by comparing signatures of known attacks and raise an alarm if they find a match [9], [10]. On the other hand, anomaly-based NIDSes monitor network traffic by observing if there are any deviation in the traffic compared to the defined normal traffic patterns [9], [10]. A problem with the anomaly-based NIDSes is that along with the correct alarms they raise, they make a considerable number of false-positive alarms that take time and attention from network administrators [9]. Signature-based NIDSes generally produce less number of false alarms, due to the fact that their alarms correspond to signatures of existing attacks. However, that means that they will fail to detect zero-day exploits, as they are relying only on the existing signatures for known attacks.

3.2.2 Snort - an Open Source Network-based Intrusion Detection System

Snort is a cross-platform, lightweight, open source NIDS that can be used as a packet sniffer, packet logger and as a NIDS on IP networks. Its packet sniffer and logger are based on the libpcap library [23]. Snort performs real-time analysis of the network packets and their contents. Besides, it can analyze pre-made PCAP-files of network packets during a certain time. The analysis is based upon pre-defined rules and a match of a rule results in an alert. Alerts and the network packets causing the alerts are logged. If no alert is raised, no data is logged. Snort has the capability

to perform this real-time analysis of the network traffic with negligent overhead to the network [23]. There exist several different sets of pre-defined rules on the Snort project's web page, some sets are written by the community, some sets by the team supporting Snort and some commercial sets are written by IT-security professionals. Besides, users can add new rules themselves.

3.2.2.1 Architecture

Snort consists of different subsystems: the packet decoder, pre-processors, the detection engine, and the alert and logging subsystem. These subsystems work on top of the libpcap packet sniffing library. The packet decoder decodes the different layers in the TCP/IP protocol stack, by calling decoding routines for the protocol stack, from the bottom layer to the application layer [23]. The main function of the packet decoder is to set pointers to different parts of the packet data, which will be used by the detection engine [23].

There exist two types of pre-processors, one pre-processor is examining packets for non-signature-based attacks and the other pre-processor is used for modifying the packets so the packets can be interpreted correctly by the following subsystem for further rule-based analysis. The detection engine is the subsystem that is parsing the rules and loads them, and performs signature detection. The rules are split into the rule head and the rule options. The rule head contains information about protocol, IP address ranges and ports while the rule option includes the string to be matched, the priority etc. The detection engine processes these rule headers and options. If the packet does not match any of the signatures, it will be dropped and no alarm will be raised. However, if the packet matches a signature, an alert will be raised. Currently, only one alert per packet is logged. The alerts that are logged are chosen according to the highest order rule the packet matches [24]. The rule set are divided into content rules and non-content rules, where the content rules have higher order priority than the non-content ones [24]. The rules are described more in-depth in the next Section 3.2.2.2.

The last subsystem is the logging and alerting subsystem for managing output data. For managing the data output in a flexible way, output modules that are also called output plugins are used [25]. There exist different output plugins for writing data output, and the output plugins are chosen in the command line before running Snort. Logging can be performed using different formats, either the tcpdump format or a binary format. The alerts can be written to a text file, in text format or CSV format, or sent to the syslog facility [25]. Text-file alerts could be done in two different modes, either fast or full alerting. In full alert mode packet header information and alert message are written to the log, while in fast alerting mode only a subset of the packet header information are written.

3.2.2.2 Attack Detection Techniques

Snort analyzes the network packets by using rules. The rules are written in a simple description language. The rules consist of two different parts; the rule header and

the rule options. The first part, i.e. the rule header, consists of a specified protocol, the source and destination IP addresses and net masks, together with their port numbers. The second part, i.e. the rule option, consists of several different categories of options. The first category includes options for specifying information about the rule. The next categories consist of several options for specifying how to inspect the packet payload as well as several other options for specifying how to inspect the other parts of the packet. The last category includes options for events to happen after a rule was matched. There are several ways to perform payload detection within the rule option part of a rule. One way is to specify a keyword to search for in the packet payload, either in its raw format or in the decoded format. It is also possible to search for a URI keyword in the normalized request URI field in the packets. It is also possible to use additional options for specifying where in the packet to look for a specific keyword. Additionally, it is possible to compare a byte field as well as other byte options.

Regular expressions could be used for writing rules within the rule option. There also exist options for decoding packets and detecting malicious encoding et cetera. [26]. Some options are also specific for certain preprocessors.

3.3 Forensics Memory Analysis

Forensics memory analysis (FMA) is a scientific field, where memory images are analyzed for gathering information about a system's state, and its OS and applications. Mostly, this analysis is highly OS-dependent. The next section describes an advanced framework within the field.

3.3.1 Volatility - An Advanced Memory Forensics Framework

Volatility is an open source framework for advanced memory forensics. It supports memory images from several different version of the following OSes; Windows, Linux and Mac OS X. By using extraction techniques of the data from volatile memory (RAM) snapshots, this framework makes it possible to investigate the state of the system during the snapshot. The analysis of the snapshots is made with plugins, that are built-in into the framework.

4

Survey of Platforms Required for Data Acquisition

The goals of this thesis are to investigate in how virtual machine introspection (VMI) could provide a network-based intrusion detection system (NIDS) with a broader view of the system as well as investigate how a NIDS and VMI could be combined. Hence, a NIDS and a virtual machine (VM) platform for performing VMI should be selected. This chapter discusses the choice of platforms for performing experiments.

4.1 Survey of Platforms for performing Virtual Machine Introspection

Virtual machine introspection is one of the two main components of this thesis. There exist libraries and platforms that are based on certain VMs, which can perform VMI [19], [14], [27]. However, there only exist a few and most of them have restricted usage.

4.1.1 Requirements

It is important that the platform or library for performing VMI as well as the VM itself is open source and has a community, in order to be able to analyze the platform thoroughly and to use state-of-the-art software. Additionally, this makes it easy to discuss potential issues or the tool itself can be discussed. It is important that there exist tools for extracting useful data from the VMI, as VMI per default provide low-level information and there is a semantic gap problem in order to extract useful high-level information from the low-level information. The data provided by the tools should be high-level state information, such as information about user-level applications and files. It is important, due to restrictions in the utilized host computer, that the VM platform works on Unix and can run a Unix operating system (OS). Lastly, it would be beneficial if there was a tool available to reproduce the network traffic performed during the execution in the VM.

These are the aforestated requirements:

- Open source software.
- Run on Unix.
- Ability to run Unix OS as guest system.
- Provide high-level state information.

- Include tools for obtaining useful data from VMI.
- Include a tool to reproduce network packets of the execution.

4.1.2 Available Platforms

This section presents the two available open source platforms for gathering useful information by performing VMI. There also exists a third open source library, VM-Safe, which was made by VMware for VMware products [27]. However, it could only be used with a third-party security tool and is no longer supported.

4.1.2.1 LibVMI

LibVMI is an open-source virtual machine introspection library [28]. It is a successor to the XenAccess libary. It is built to simplify the introspection with the focus on memory introspection by using physical addresses, virtual addresses and kernel symbols [28], [14]. On Linux, LibVMI accesses the files, and it can access the VMs; Xen and KVM. On MAC OS X it can access the files. It can be used on memory snapshots. It can introspect both Linux and Windows operating systems. For high-level state information it should be used with PyVMI, that utilizes Volatility live, or Volatility itself [28]. Except from PyVMI, there are no other tools. However, LibVMI provides an API for writing new tools. Table 4.1 shows the result of the survey of LibVMI.

Table 4.1: Survey of LibVMI.

Requirement	**Fulfills Criteria**
Open source software	✓
Run on Unix	✓
Ability to run Unix OS as guest system	✓
Provide high-level state information	✓
Include tools for obtaining useful data from VMI	✓
Include a tool to reproduce network packets of the execution	

4.1.2.2 PANDA

PANDA is a relatively newly developed open-source platform PANDA, that can perform VMI by the use of plugins, which provide the user with introspection data useful for program analysis. PANDA was presented more in-depth in Chapter 3, Section 3.1.2. PANDA is a unique platform that utilizes VMI for performing program analysis. Currently, it only works on certain version of Linux OSes [21]. PANDA includes a record and replay functionality as well as a powerful plugin architecture to perform VMI [16]. The plugins are applied to recordings and can provide various information, such as process information, memory access information and network packets [21]. Furthermore, some PANDA plugins can produce a raw memory file of the memory, which could be used by forensics memory analysis tools such as Volatility for further analysis [21]. There are APIs for writing new plugins. The

PANDA community is small, but active. Table 4.2 shows the result of the survey of PANDA:

Table 4.2: Survey of PANDA.

Requirement	Fulfills Criteria
Open source software	✓
Run on Unix	✓
Ability to run Unix OS as guest system	✓
Provide high-level state information	✓
Include tools for obtaining useful data from VMI	✓
Include a tool to reproduce network packets of the execution	✓

4.1.3 Discussion of chosen Platform

PANDA was chosen as it provides more tools than LibVMI, which could provide high-level state information and can reproduce the network packets. Besides, PANDA can produce the same data as LibVMI, as it can inspect the memory by using Volatility. However, while PANDA mainly supports analysis of recordings, LibVMI could produce live inspection via Volatility. Due to PANDA's potential to produce more data via its plugin architecture, PANDA is still considered a better choice.

4.2 Survey of which Operating System to Introspect

The PANDA platform supports several OSes, where the restrictions are basically the supported architectures for being able to use PANDA's record and replay functionality. However, the available PANDA plugins support different operating systems. Based on the information available on PANDA's Github it is mostly Linux-based and 32-bit Windows 7 architectures that are supported. There exist distinct plugins for each of the architectures. This makes it interesting to run test cases for both Windows-based and Linux-based OSes. Due to restrictions in the utilized host computer as well as PANDA's lack of support for recording via hardware virtualization i.e. KVM, a rather simple 32-bit Linux OS was chosen for the Linux-based test cases.

Due to the restrictions in PANDA and the utilized host system, no Windows test cases were performed on our host computers. However, due to PANDA's large community it was possible to find recordings of well-known attacks targeting Internet Explorer on their web page, 'PANDA SHARE', which is a web page were recordings are distributed [29]. These recordings could be used for gathering data from PANDA as well as Snort for further analysis.

4.3 Survey of Beneficial Tools

By running test cases with different events it is possible to obtain data from both NIDS and VMI for analysis. The test cases will be performed by sending traffic to an application, recording the execution to do virtual machine introspection, and providing captured traffic to the NIDS to analyze.

4.3.1 Survey of Virtual Machine Introspection Tools

In Chapter 3, Section 3.1.2.4 an overview of all of PANDA's plugins was presented. Based on this overview, we hereby present the following subset of plugins that were chosen for our analysis.

4.3.1.1 General Tools

The plugin *replaymovie*, that creates a video of the desktop during the recording is a useful tool. This plugin makes it possible to get a better view over what happened in the desktop of the system. In order to be able to analyze the network traffic with Snort, the plugin *network*, which creates a PCAP-file with the network traffic during a recording, is necessary. The plugin *scissors* can produce shorter recordings, which is beneficial if a recording is long or if a certain part of the recording is interesting.

4.3.1.2 Process Tools

The plugin *asidstory* is used for obtaining process information. Also, there exists a Windows-specific plugin for obtaining process information, which is presented in Section 4.3. The *asidstory*-plugin produces useful information about each process such as its PID and address space ID (ASID) as well as high-level information about the processes' execution in the system. Hence, this plugin provides basic knowledge about the processes in the system.

4.3.1.3 Memory Tools

PANDA has several plugins for obtaining data written into or read from memory. Memory operations are useful in order to obtain information about user-level applications, which the researchers behind PANDA discussed in their paper "Tappan Zee (North) Bridge: Mining Memory Accesses for Introspection" [17].

The most basic memory plugin is *memstrings,* which prints all strings from the recording that was read from or written into the memory. This plugin makes it possible to search for strings, for example "Internet Explorer has stopped working", which originates from the message that is created when Internet Explorer no longer responds. This is an easy way of getting a better view over what has happened in the system, for example, if Snort raised an alert. By searching for such strings, one can conclude that a specific application did or did not crash. These kinds of error messages differ between OSes. However, such strings could also be searched for by

using the plugin *stringsearch*, which creates tap points related to the strings that are given as input. The tap points include the process reading or writing the string in memory. Besides, *stringsearch* writes to the command line the instructions count of each of the read and write matches of the searched strings. These instruction counts can be used to create shorter recordings using the plugin *scissors*, which is beneficial if the recording is large and if one is interested in analyzing a particular part of the recording. This might also be useful if the tap points produced by *stringsearch* is going to be analyzed more in-depth by using other plugins such as *textprinter* and *memsnap*. Then it might be beneficial to make a shorter recording in order to only get the relevant tap points.

The *textprinter*-plugin can produce the data read and written in memory at the given tap points. This is beneficial when, for example the cause of an Internet Explorer crash is not known, but one can guess that it for example has to do with HTML data received. Then it is possible to create tap points by using *stringsearch* and search for the start of the HTML messages "<html" or "<HTML". Then the *textprinter*-plugin will recreate the HTML file, which can be further analyzed manually. The plugin *memdump* can produce all data written or read in the memory by tap points. This plugin should only be used after the plugin *tapindex* has been deployed. The data is obtained by using commands like "grep" for certain strings. This might not however be an effective way of analyzing the memory accesses, as one most likely wants to see all the data at certain tap points. But if one only wants to find out if a specific string is written or read in the memory it might be useful.

The *memsnap*-plugin can produce the raw memory data from specific tap points for further analysis with Volatility, which provides in-depth analysis of the data obtained from the memory. Another plugin *memsavep*, also produce raw memory data, but for a specific percent of the recording. If a recording is rather small it can produce raw memory data for the whole recording, which can be analyzed with Volatility.

4.3.1.4 Windows-specific Tools

The plugins *STUW* and *win7proc* are Windows-specific plugins. *STUW* produces information about system calls made within the interprocess communication and *win7proc* produces information about different kinds of system calls in the recording. Besides, *win7proc* produces information about a process life and spawned child processes. This plugin can also produce the data written during the recording. Hence, these Windows-specific plugins are very useful for gaining in-depth information about the system and specific applications.

4.3.1.5 Linux-specific Tools

The taint plugin, *file_taint* is only supported for Linux-based systems. This could be used for tainting, but also for obtaining data about system calls involving file management, and discovering which processes that made the system calls.

4.3.1.6 Not Deployed Tools

Lastly, there are several other plugins that are not discussed here, due to several reasons. Some plugins provided by PANDA are helper-plugins and are only used in conjunction with other plugins. Several plugins such as *unigrams*, *tapindexes*, *bigrams* and *correlatetaps* provide statistics about tap points, which is not useful while trying to analyze the result of the selected events. There exists two plugins that provide low-level information *llvm_trace* and *coverage*. By using these plugins it is possible to obtain all the traces of LLVM instructions during the replay as well as all the basic blocks in LLVM IR code. However, the data generated by *coverage* makes it hard to follow the execution trace of the process as the plugin only prints all unique basic blocks, which are a lot for even a small recording. In order to be able to analyze the basic blocks, one needs to use the *llvm_trace*-plugin, which produces extremely large output. Even a 15 second recording of the system being idle ends up with 100 GB produced output. The large size of these output files, produced by both *coverage* and *llvm_trace*, make them difficult to analyze and also, crash the particular computers dedicated for this thesis. Hence, such plugins are not used.

Plugins that write information about the stack, i.e. *fullstack* and *printstack*, are not used either since they produced quite large amount of data that is hard to analyze. As the selected attacks do not include tainted data the taint plugins; *ida_taint2*, *dead_data*, *tainted_branch*, *tainted_instr* and *tstringsearch* are not used. There are several plugins found irrelevant as they do not produce data that is applicable for the selected events; *keyfind*, *osi_winxpsp3x86*, *bufmon*, *rehosting* and *textprinter_-fast*. Lastly, there exists a plugin called *useafterfree*, which is written specifically to analyze if a use-after-free attack has occurred. However, as this plugin requires in-depth memory inspection in order to find a specific address space with Volatility, it is not used.

4.3.2 Survey of Forensics Memory Analysis Tools

Due to the plugins that produce raw memory files, Volatility needs to be used to analyze these files. Volatility has a similar plugin architecture as PANDA, hence different plugins can be chosen due to different circumstances. Moreover, the plugins exist in different versions depending on the OS. Hence, the interesting plugins will be presented separately for Windows and Linux.

The Windows-specific plugins that provide process information are interesting. The following plugins process information; *pslist*, *psscan*, *pstree* and *psxview*. The plugins differ in how they present the processes and what kind of information about the processes are displayed. Information about files can be shown using the plugin *filescan*, which is beneficial if the attack is caused by received files. The *dllist*-plugin provides information about the DLL files, which is interesting due to security issues involving DLL files.

Besides, there exist plugins specific for the Windows OS, *iehistory*, *windows* and *wintree*. The first one, *iehistory*, produces information about the history and cache

memory of Internet Explorer and the other Windows plugins produce desktop information about the desktop of the system.

The Linux-specific plugins that provide process information are relevant to deploy. For Linux the following plugins produce information about processes; *linux_psaux*, *linux_psenv*, *linux_pslist*, *linux_pstree*, *linux_psxview* and *linux_proc_maps*. As for the Windows-specific plugins, the Linux-specific plugins differ in which kind of information about the processes that are shown. The file plugin *linux_enumerate_files* is useful for gaining information about files that recently have been changed. For the attacks that involve opening new ports the plugins *linux_netscan* and *linux_netstat* are useful as they provide information about the network connections.

4.4 Survey of Network-based Intrusion Detection Systems

One of the main components of this thesis is the NIDS platform. There exist different types of NIDSes, anomaly-based, signature-based and a combination of these two. The way the alerts are raised also differs among the NIDSes, and the definition of alerts can be more or less complex. This section describes the selection of the NIDS based on the requirements of a NIDS for this thesis.

4.4.1 Requirements

There are several requirements for the NIDS that will be used for this thesis. It is important that the NIDS is an open source project and has a rather large community, in order to be up-to-date with the state-of-the-art technology. Besides, an open source community enables a way to discuss details about the NIDS and its alarms. Due to restrictions of the host computer used for this thesis, the NIDS needs to able to be configured for Unix. It is important to know why it should be straightforward an alarm was raised, and it is a requirement that the NIDS should use signature-based techniques. Besides, a large amount of existing signatures is preferable to be able to find existing alarm for vulnerabilities and shellcode, i.e. attack code, that is utilized in the test cases (described in Appendix A). Furthermore, the NIDS needs to have an accessible way of defining new signatures for vulnerabilities, if the NIDS do not have signatures for vulnerabilities that is utilized for the test cases. Preferably, it should be able to analyze the PCAP-files of the network traffic in the same way it normally analyze traffic.

These are the aforestated requirements:

- Open source software.
- Run on Unix.
- Utilizing signature-based techniques.
- Including a large amount of existing signatures.
- Provide an accessible way to define new alerts.

- Ability to analyze PCAP-files.

4.4.2 Available Platforms

This section presents three open source NIDSes that according to an online search and by reading forums seem to be the most utilized today.

4.4.2.1 Bro

Bro is an open-source NIDS, which was developed as a research tool [30]. It is Unix-based and is not yet deployed to work with other operating systems [31]. It utilizes both the techniques of an anomaly-based NIDS and a signature-based NIDS. It is based on events, where events in the network traffic are analyzed by Bro policy scripts. Bro policy scripts are written in the Bro scripting language and could be implemented to raise alerts among other things. Bro is rather complex, but have a lot of features. Signatures can be used in Bro, but it is not recommended to use [30]. It supports analysis of PCAP-files. Table 4.3 shows the result of the survey of Bro.

Table 4.3: Survey of Bro.

Requirement	**Fulfills Criteria**
Open source software	✓
Run on Unix	✓
Utilizing signature-based techniques	
Including a large amount of existing signatures	
Provide an accessible way to define new alerts	
Ability to analyze PCAP-files	✓

4.4.2.2 Snort

Snort is one of the most well-known and deployed NIDSes in the world. It is mostly a signature-based NIDS [23]. It was presented in detail in Chapter 3, Section 3.2.2. Snort have existed since the 90s and is continuously developed [23]. It has a large community that provides great support regarding alerts, i.e. alarms. The alerts are defined by rules [23]. Snort have a very large set of rules, both community rules, rules provided by Snort developers and commercial rules provided by external companies, which continuously are updated. The alerts provides the IP addresses and port numbers, a message, a category, a priority et cetera as well as a reference to the rule, which makes is to interpret why the alarm was produced [23]. Older versions of rules are always available. The rules are defined in a accessible way, which makes it easy to write new rules [23]. Snort analyzes the captured traffic in the same way as the live traffic and works on many OSes [23]. Table 4.4 shows the result of the survey of Snort.

Table 4.4: Survey of Snort.

Requirement	Fulfills Criteria
Open source software	✓
Run on Unix	✓
Utilizing signature-based techniques	✓
Including a large amount of existing signatures	✓
Provide an accessible way to define new alerts	✓
Ability to analyze PCAP-files	✓

4.4.2.3 Suricata

Suricata is a powerful cross-platform open source NIDS, which was released in 2009. It has several advantages, such as being multi-threaded, therefore it could benefit from more than one CPU core in the system and could operate faster with heavy traffic [32]. Furthermore, it can extract malicious files that is downloading, identify protocols and log more data than just the network packets [32]. However, these are not requirements for this thesis. Alerts, i.e. alarms, are defined by rules in a similar manner to Snort, and Snort's rules can be used in Suricata. However, there are only a few rules provided by Suricata The alerts are defined in an accessible way. Suricata supports analysis of PCAP-files. Table 4.5 shows the result of the survey of Suricata.

Table 4.5: Survey of Suricata.

Requirement	Fulfills Criteria
Open source software	✓
Run on Unix	✓
Utilizing signature-based techniques	✓
Including a large amount of existing signatures	
Provide an accessible way to define new alerts	✓
Ability to analyze PCAP-files	✓

4.4.3 Discussion of Chosen Platform

All the three presented NIDSes have advantages, however, Snort was chosen as the NIDS for this thesis as it fulfilled the stated requirements for this thesis. Suricata and Bro have smaller communities compared to Snort. Suricata has only a few ruleset of its own. As Snort is more deployed and has a larger community it was chosen over Suricata. Bro has a more complex way of defining alerts and should preferably not be used as a signature-based NIDS.

4.4.4 Survey of Network-based Intrusion Detection System Tools

The chosen NIDS, Snort, deploys rules to analyze the network traffic and to be able to raise alerts. There exist many rules and rulesets, both community rules as well

as Snort's own rulesets. Besides, people publish rules online at code sites.

5
Methodology

This chapter consists of two parts. The first part describes the experimental phase in this thesis, i.e. what kind of experiments that are performed as well as how the experiments are performed. The second part of this chapter describes the evaluation of the experiments.

5.1 Experimental Phase

The setup of this thesis includes the two main components; a network-based intrusion detection system (NIDS) and a platform for performing virtual machine introspection (VMI). In order to meet the goals of this thesis, that is to investigate in how virtual machine introspection can give a broader view to state-of-the-art network-based intrusion detection systems of today and to investigate if network-based intrusion detection system and virtual machine introspection can be combined, data will be gathered from both components and a performance analysis of the VMI platform will be conducted. Network traffic with various payload will be sent to an application running in a guest system, in order to cause different attacks. In order to gather relevant data, the NIDS will analyze the traffic and VMI will be performed on the guest system. As discussed in Chapter 4, the chosen NIDS is Snort and the chosen VMI platform is PANDA. The setup of the experiments can be seen in Figure 5.1, where more in-depth information about the setup is described in Chapter 6.

The chosen platform for VMI has different support for which data that can be gathered from different operating systems (OSes). The concerned OSes are 32-bit Windows 7 and Linux systems, which were discussed more in-depth in Chapter 4.1. Due to this fact, both Windows-based and Linux-based test cases will be performed. However, not all of the tests will be performed in both due to restrictions in the utilized host computer in the setup as discussed in Chapter 4.1. The main difference between the support of the two OSes is that Windows can provide more information about the performed system calls and how they affect the system. Only necessary to have a few test cases that affect the life of a process and involves files.

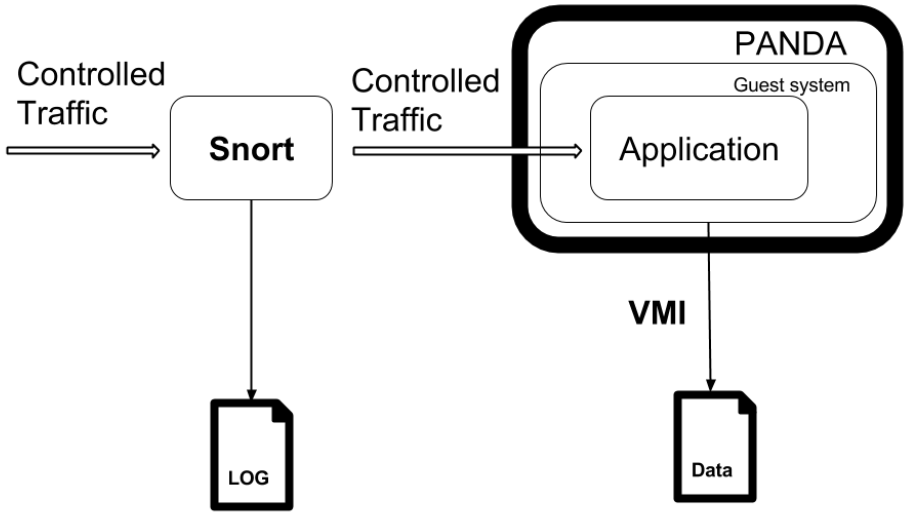

Figure 5.1: Overview over the experimental setup.

5.1.1 Choice of Application

An increasing amount of hosts are running in virtual machines (VMs) due to the many advantages and field of applications. One of the fields in which VMs are used more and more is for running servers [33]. One example of this is cloud data centers, where servers are running on virtual machines due several reasons such as lower cost, fault-tolerance and system maintenance [2]. These servers are under an increasing amount of security threats, which increases the need of NIDSes [6]. Hence, system administrators for such systems could utilize the information from VM, i.e. VMI, as an extension to the NIDS to gather data about the servers' state. This thesis will therefore utilize a server as the application to test the selected attacks.

5.1.2 Performance Tests

To answer one of the goals of this thesis, how VMI and NIDS could be combined, and to be able answer the research question if it is beneficial to use VMI as an extension to a NIDS as well as the research questions about how and when NIDS and VMI could be combined, it is necessary to conduct performance test of the VMI.

It is important to test the user-experience of running the VMI platform, as the VMI platform directly affects the monitored guest system. By measuring the round-trip time of running in an ordinary VM and the chosen platform for VMI, it is possible to draw conclusions about how the performance is affected. The chosen platform requires that executions are recorded in order to analyze the data. Hence, the round-

trip time for running the VMI platform while recording will be measured. Also, the memory usage will be measured for the different test cases.

The recordings of executions on the VMI platform are saved to the host system and these recordings are used to perform VMI. Therefore, it is important to measure their size. How are the recordings increasing with time, and how much are the recordings increasing when applications are operating.

The recordings created by the VMI platform are analyzed by different plugins that perform program analysis using VMI. In order to draw conclusions about these plugins with regards to performance, it is necessary to perform different performance test cases. The different aspects that are interesting to analyze are the time it takes to gather data, how much data that is produced by the plugins, and the memory usage for the plugins. The chosen VMI platform produces raw memory files that can be analyzed with a forensics memory analysis (FMA) tool. The chosen FMA tool will also be included in these performance test cases. The FMA tool as well as the plugins of the VMI platform was presented and discussed in the previous chapters, Chapter 3 and Chapter 4. The plugins could be run on any host system using the VMI platform as long as the recordings are accessible.

5.1.3 Data Acquisition from the Platforms

To be able to meet the goals of this thesis as well as answering the research questions, data from the two platforms needs to be gathered. These require various test cases of different attacks, and in order to test the NIDS thoroughly, various payload for the test cases are needed. The data gathering will be performed by running the network traffic in both of the platforms, which allows analysis of the data the two platforms would provide if they always were used in conjunction.

5.1.3.1 Virtual Machine Introspection

Several test cases that affects the guest system in various ways will be performed, in order to investigate what kind of information about the monitored systems' and its applications' state that can be gathered using VMI. A certain number of attacks are needed in order to test how the VMI platform can provide data about different parts of the system, with different levels of severity. Besides, normal execution is needed in order to analyze how a normal execution is shown by the VMI platform.

Due to the fact that attacks can be performed in vastly different ways, it is infeasible to test all kind of attacks. As VMI provides data about the state of a system and its applications, attacks that affect different parts of the system are chosen and attacks that affect, for example, the visuals of an application are not included. The attacks have different severity and the result of them are widely different. Attacks with lower severity could be part of a large, and more severe attack and are therefore important to evaluate. The test cases that were chosen are presented below. Figure 5.2 shows an overview of the attacks.

- **Attack: Add a new File** Add a file to the systems. Except from unauthorized adding files to the targeted system, this kind of attack could result in malicious code being placed on the system. It could also be part of an attack where a remote file is fetched and saved to a new file in the targeted system to be executed [34].

- **Attack: Add a new User** Add a new user to the system. This could lead to unauthorized persons accessing the system by using the newly added user.

- **Attack: Change Privilege of a File** Change privilege of a file, in the system. This could lead to unauthorized persons accessing a file they should not be able to touch or that authorized persons no longer being able to access the file. An example of such a file can be a text file with passwords or an application.

- **Attack: Crash a Process** Crash a process in the system. This is a denial-of-service attack that could cause major issues for an individual.

- **Attack: Kill all Processes** To kill all processes in the system can be devastating if crucial processes are running on the system.

- **Attack: Read a File** Read a file in the system. This can provide the attacker, and potentially more persons, with unauthorized information.

- **Attack: Reboot the System** Reboot the system, which can be devastating if crucial processes are running on the system.

- **Attack: Start a new Process** Start a new process in the system. This could result in that a malicious process is started that harms the computer or that a shell is started, allowing the attacker to access the system from the shell.

- **Attack: Write to a File** Write to a file in the system. This can affect the system in various ways, such as purely sabotage or add malicious code.

- **Normal Execution** A normal execution of the processes in the system. This event is important to evaluate in order to be able to discover the difference between a normal execution and an attack.

Figure 5.2: Overview of the selected attacks.

5.1.3.2 Network-based Intrusion Detection Systems

In order to answer the research questions about how and when NIDS and VMI should be combined it is beneficial to run tests that result in the different outcome of the NIDS. Hence, the test cases, presented in Section 5.1.3.1, will be performed several times in different ways, each in order to provide a view of what kind of data the NIDS provides. The NIDS's analysis of incoming and outgoing network traffic always results in one of the four following categories: false-positive, false-negative, true-positive or true-negative. The aim is to produce all categories of output from the NIDS during the test cases if it is possible. This will show how data from VMI could be used in the different categories of output. Figure 5.3 shows the attacks and the different output that the NIDS can produce.

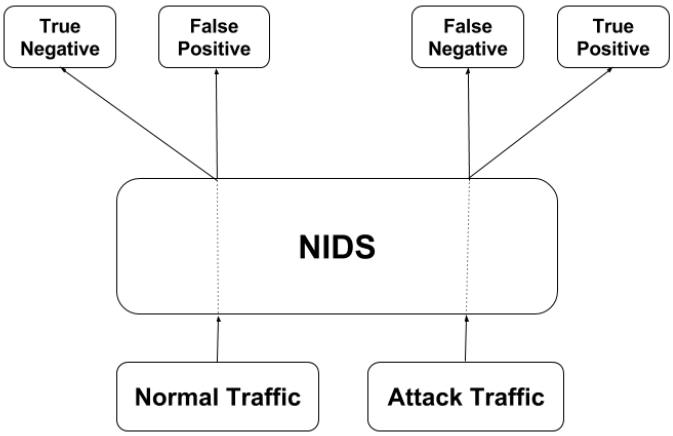

Figure 5.3: Overview of attacks and the different categories of output from the NIDS.

Furthermore, a performance analysis is needed to be able to fully answer the research question if it is beneficial to use VMI as an extension to a NIDS as well as the research question about how and when NIDS and VMI should be combined. The performance tests are described in the next section, Section 5.2.1

5.2 Evaluation Phase

The performance test cases will be analyzed for each of the different test cases in order to draw conclusions about how it is beneficial to use the VMI platform. The data gathered by the chosen NIDS and the chosen platform for performing VMI will first be analyzed separately and then, combined.

5.2.1 Performance Analysis

For each of the different performance test cases, the result of the tests will be compared. This means that for the first test case, user experience, the round-trip time as well as the memory usage for all the tests will be compared and analyzed. For the second test case, the sizes of the recordings with will be compared to analyze the growth of the recordings with regards to time. Then the recordings with network traffic respective without network traffic will be compared in order analyze how computations affects the size of a recording. In the third test case, the result from the tests involving plugins of the VMI platform as well as the FMA tool, will

be compared. Hence, the size of the data produced by the plugins and FMA tools will be compared and analyzed as well as the memory usage and the time they take to execute. By analyzing the outcome from the performance analysis it will be possible to draw conclusions about how and when the VMI and the NIDS should be combined.

5.2.2 Analysis of the Data from the Platforms

The data produced by the VMI platform for the different test cases will be compared in order to analyze what view of the system that can be provided by the VMI. Also, the data produced by the NIDS and the VMI platform will be compared in order to investigate how the outputs from these systems differ and how could they be combined.

Due to the fact that different data can be gathered from Windows-based and Linux-based system in PANDA, the outcome from the Windows-based and Linux-based test cases will be compared. The comparison will be performed by analyzing what kind of information can be extracted from the Windows-based test cases in comparison to the Linux-based test cases.

The data produced by the VMI platform for each test case will be analyzed and compared, in order to investigate what kind of attacks that can be discovered by using VMI and how much data about each test case that is produced. Also, the test cases involving normal traffic will be compared to the attack test cases. This makes it possible to analyze how normal executions are presented by the data produced by the VMI platform.

The data from the NIDS and VMI platform will be compared differently for each category of output from the NIDS. The test cases within the category true-positive will be analyzed by comparing the data produced by he NIDS respective the VMI platform during the different attacks. The data will also be analyzed in a combined fashion, in order to analyze how beneficial it is to combine the data. The gathered data about the attacks will be compared to the actual result of the attacks, in order to state how accurate information about the actual attacks the data provides.

The data from the false-positive test case will be analyzed by comparing the data from the NIDS and the VMI platform. The data from the VMI platform will be analyzed by investigating if it provides correct information about the system state, i.e. if it is possible to state that an alert from the NIDS was a false-positive alert. The false-negative test case will be analyzed by analyzing the produced data from the VMI platform and investigate if it provides the correct information about the system state, i.e. if it is possible discover the result of the test case that the NIDS missed. Lastly, for the true-negative, the produced data from the VMI platform will be analyzed to determine if it is possible to conclude that no attacks were performed, i.e. that the alert was an authentic true-negative. For all the described test cases, it is interesting to analyze how much in-depth information about each attack the

platforms can provide, and what conclusions that can be drawn from the data.

6
System Setup

This chapter presents the systems, platforms and applications used for performing the test cases needed to obtain relevant data for the analysis of the network-based intrusion detection system (NIDS) and virtual machine introspection (VMI).

6.1 Overview over the Test Systems and the Test Applications

The main operating system (OS) for the test cases is Linux, and the main application is a server. However, due to that the chosen VMI provide different data for Linux and Windows, two Windows test cases targeting Internet Explorer were included.

6.1.1 Windows-based Tests

The Window-based tests were pre-made recordings by Saumil Shah and Ryan Whelan from PANDA's web page for distributing recordings for further analysis [29]. These recordings were originally performed on a 32-bit Windows 7 system.

6.1.1.1 Application

The client application that was attacked in the pre-made recording was Internet Explorer version 8, which is vulnerable to the use-after-free vulnerability CVE-2012-4792. For the interested reader, use-after-free attacks and the vulnerability CVE-2012-4792 are described more in details in Appendix A.

6.1.2 Linux-based Tests

The Linux-based tests were performed on a 32-bit Debian Squeeze system, by using a pre-made image with the 32-bit Debian Squeeze installed [35]. This image was used to run the OS in PANDA's virtual machine. The selected application was installed in the system and by using PANDA's record and replay functionality different execution traces of the application could be recorded and later replayed for in-depth analysis.

6.1.2.1 Application

A vulnerable echo server made by Jeffrey A. Turkstra [36] was used to perform the test cases. The server simply echo back the input given by the client. The

echo server is vulnerable against buffer overflow attacks, which can make the server crash or execute code sent by an attacker. For the interested reader, buffer overflow attacks are described more in details in Appendix A, Section A.2.

6.2 Overview over the Test Environment

The chosen operating system (OS) for performing the test cases was 64-bit Ubuntu version 14.04, due to restrictions in supported OSes of the chosen VMI platform. Ubuntu was running in Parallels, with 9884 MB RAM, four cores and it was run in a nested virtualization mode to fully support VM in VM.

6.2.1 PANDA for Performing Virtual Machine Introspection

The latest version of PANDA existing on Github in February 2016 was employed for program analysis by performing VMI. PANDA includes a virtual machine (VM), QEMU, where the system to perform tests was run. In order to obtain data from the introspection of the VM, several of PANDA's plugins were used. The choice of the plugins was discussed and presented in Chapter 4, Section 4.3.1.

6.2.2 Volatility for Performing Forensics Memory Analysis

Volatility version 2.5 was used for forensics memory analysis.

6.2.2.1 PANDA plugins for Data Acquisition for Windows-based Test Cases

This section presents the plugins, see Table 6.1, that can be used to obtain useful data for the analysis. Table 6.1 does not include helper-plugins. It should be remarked that *memsavep* could be used in shorter recordings instead of *memdump* or *memsnap*. Besides, the *scissors-plugin* is not necessary in shorter recordings.

Table 6.1: PANDA Plugins for Windows-based Test Cases. A list of all used PANDA plugins from PANDA's github. The 'Windows-specific'-column describes if the plugin is Windows-specific or not.

PANDA Plugins	Windows-specific
asidstory	
memdump	
memsavep	
memsnap	
memstrings	
network	
replaymovie	
STUW	✓
stringsearch	
scissors	
textprinter	
win7proc	✓

6.2.2.2 Volatility plugins for Data Acquisition for Windows-based Test Cases

This section presents the Volatility plugins that produce useful data for the analysis of the Windows-based test cases, which are shown in Table 6.2.

Table 6.2: Volatility plugins for Windows-based Test Cases.

Volatility Plugins
dllist
iehistory
filescan
pslist
psscan
pstree
psxview
windows
wintree

6.2.2.3 PANDA plugins for Data Acquisition For Linux-based Test Cases

This section presents the plugins that can be utilized to obtain useful data for the analysis of the Linux-based test cases. These plugins are presented in Table 6.3. This table does not include helper-plugins. It should be remarked that *memsavep* could be used in shorter recordings instead of *memdump* or *memsnap*. Besides, the *scissors*-plugin is not necessary in shorter recordings.

Table 6.3: PANDA plugins for Linux-based Test Cases. A list of all available well-documented PANDA plugins from PANDA's Github. The 'Linux-specific'-column describes if the plugin is Linux-specific or not

PANDA Plugins	Linux-specific
asidstory	
file_taint	✓
memdump	
memsavep	
memsnap	
memstrings	
network	
replaymovie	
stringsearch	
scissors	
textprinter	

6.2.2.4 Volatility plugins for Data Acquisition For Linux-based Test Cases

This section presents the Volatility plugins that produce useful data for the analysis of the Linux-based test cases. Table 6.4 shows the utilized plugins.

Table 6.4: Volatility plugins for Linux-based Test Cases.

Volatility Plugins
linux_enumerate_files
linux_netscan
linux_netstat
linux_proc_maps
linux_psaux
linux_psenvt
linux_pslist
linux_pstree
linux_psxview

6.2.3 The Network-based Intrusion Detection System Snort and its Rule Sets

Snort version 2.9.8.0 was employed for analyzing the network packets. For each test, Snort analyzed a PCAP-file of the network traffic sent during each test. There exists several different rule sets. For this thesis the rule set *snort-snapshot-2980* was used. Specifically, the rules called *browser-ie* and *shellcode* were used for the Windows test cases targeting Internet Explorer. The *browser-ie*-rules contain rules for the vulnerability that is attacked in Internet Explorer. The *shellcode*-rules include rules for detecting different signs of shellcodes, i.e. attack code, as well as specific attacks.

These *shellcode* rules were also used for the Linux test cases targeting a small server.

Furthermore, a new rule was deployed for the vulnerability in the server, as no existing rules were deployed for this server's specific vulnerability. According to Snort's manual, "good" rules catch attacks targeting the vulnerability instead of specific exploits that easily could be changed by the attacker [37]. The added rule checks that the IP is the home network, the port is the port of the server and that the size of the package is not larger than the critical size that makes the buffer overflow. The priority is one. There are two different versions of this rule employed, one with the category "Attempted User Privilege Gain" and one with "Attempted Administrator Privilege Gain". This is due to that the server will be tested both by running it with root privileges and without. "Attempted Administrator Privilege Gain" is utilized when the server is running with root privileges and "Attempted User Privilege Gain" is utilized when the server is running with user privilege, i.e. normal execution. Snort was performing the analysis by reading from PCAP-files, one PCAP-file for each attack.

7
Evaluation and Discussion

This chapter presents the performance analysis, the result of the attacks and the applicability of combining the systems. First, the performance analysis of the VMI platform is presented. Second, the result of the performance analysis and how it affects the usage of the VMI platform is discussed. Then, the performed attacks are presented. First, the data from the network-based intrusion detection system (NIDS) is presented and discussed. Second, the data that could be gathered from virtual machine introspection (VMI) is presented and discussed. Then, the data from different operating systems (OSes) are compared and discussed. Lastly, the applicability of combining the system is discussed, with regards to previous presented result in this chapter.

7.1 Performance Analysis of the Virtual Machine Introspection Platform

This section presents the performance analysis of the VMI platform with regards to the user experience, data usage of the recordings and the time consumption, data usage and memory usage of the VMI platform's plugins.

The memory usage was gathered with the tools *top*, *ps_mem*, and PANDA's plugin's output. For *top* and *ps_me*, the memory usage of a process per second where gathered. top shows different parts of the memory usage for each process as a whole. The most interesting parts are the ones showing the percentage of available physical memory used by the processes, the amount of non-swapped physical memory that the processes are using and the amount of memory that could be shared that is used by the process. The *ps_mem* tool provides information about the total memory usage of an entire process with regards to physical memory and shared memory. In this thesis, we refer to the sum of the amount of physical memory used by a process and the shared memory that is used by a process as the total memory usage of the process. *top* and *ps_mem* calculate the shared memory in different ways, and it should be notices that *top* has some problems with showing the accurate memory usage on Linux [38]. There exist more advanced tools within the field that can show memory usage more exact for different threads within a process. However, the tools used in this thesis provide a statement about the memory usage of a process.

The performance test cases were performed on a 64-bit Ubuntu version 14.04. Ubuntu was running in Parallels, with 9884 MB RAM, four cores and it was run in

a nested virtualization mode to fully support VM in VM. It should be noted that VM in VM might affect the performance. However the result still gives indication of the performance of the platforms and plugins.

7.1.1 User Experience of Running in the Platform

This section presents the user experience of utilizing the platform for running an application in a guest system, by measuring the round-trip time of the server that was utilized in the Linux test cases. A Python script measures the wall time for 1,000 messages that are sent from the client to the server until the client gets the messages echoed back from the server. This test was performed 100 times for each test case. Besides, the memory usage was measured to show the potential differences between the platforms. This was measured five times using *top* and five times using *ps_mem*. Both of the tools were used to provide an accurate view of the memory usage as this affects the host system running the VM. The test cases where performed on the different VM platforms; the current latest version of QEMU; QEMU-2.6.0, PANDA and in PANDA while utilizing PANDA's functionality of recording. In all cases, the image of the Linux OS that was utilized for the Linux-based test cases was used.

7.1.1.1 Result and Discussion

The Round-trip time for each of the three test cases are presented in Table 7.1 and the confidence intervals are presented in Figure 7.1. This is the mean value of 100 tests. As can be seen in the Table 7.1, the difference between the round-trip time of QEMU 2.6.0 and PANDA is rather small. The difference is about 0.013 seconds, and PANDA is about four percent faster. However, the round-trip time for PANDA while utilizing its recording feature is longer. The round-trip time is about 21 percent slower than QEMU 2.6.0 and 26 percent slower than PANDA. Figure 7.1 shows the confidence intervals of the round-trip time for the three platforms. The confidence intervals were obtained by using 100 samples and using a 95 percent confidence.

Table 7.1: The mean values of the round-trip time for the VM platforms.

System	**Time[s]**
QEMU2.6.0	0.31861
PANDA	0.30569
PANDARecording	0.38396

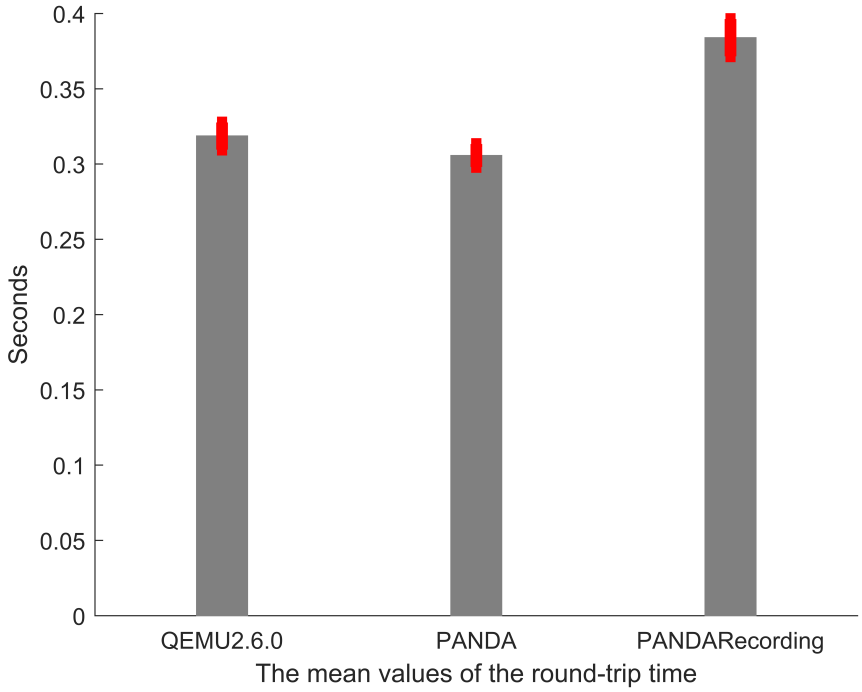

Figure 7.1: Overview of the confidence intervals for the round-trip time. 100 samples, 95 percent confidence.

The Memory Usage was measured during the test cases. The memory usage measured by *top* was constant and is presented in Table 7.1, Table 7.2, Table 7.3 and Table 7.4. The memory usage measured by *ps_mem* is presented in Table 7.5 and Table 7.6. An overview of the result of the memory usage is presented in Table 7.7.

As can be seen, PANDA requires almost 4 times more memory usage than QEMU 2.6.0. However, the memory usage does not differ that much between using PANDA with or without its recording functionality. The result from *top* and *ps_mem* were slightly different.

Table 7.2: The memory usage of the physical memory of the platforms, presented in percent of the different platforms. Total of amount of RAM available is 9634 MB. Measured by top.

System	**Memory Consumption [Percent]**
QEMU2.6.0	4.5
PANDA	16.0
PANDARecording	16.0

Table 7.3: The mean value of constant memory usage in MB of the physical memory in the different platforms. Total of amount of RAM available is 9634 MB. Measured by top.

System	Memory Consumption [MB]
QEMU2.6.0	393
PANDA	1503
PANDARecording	1505

Table 7.4: The mean value of constant memory usage of the shared memory in MB of the different platforms. Total amount of RAM available is 9634 MB. Measured by top.

System	Memory Consumption [MB]
QEMU 2.6.0	11.02
PANDA	4.60
PANDA recording	4.66

Table 7.5: The mean value of constant memory usage in MB of the physical memory in the different platforms. Total amount of RAM available is 9634 MB. Measured by ps_mem.

System	Memory Consumption [MB]
QEMU 2.6.0	370
PANDA	1500
PANDA recording	1500

Table 7.6: The mean value of constant memory usage of the shared memory in MB of the different platforms. Total amount of RAM available is 9634 MB. Measured by ps_mem.

System	Memory Consumption [MB]
QEMU 2.6.0	2.6
PANDA	0.627 - 0.663
PANDA recording	0.699-1.200

Table 7.7: Total memory usage, shared plus physical. Rounded off to closest 100 based on the previous numbers. Measured by ps_mem and top.

System	Memory Consumption
QEMU 2.6.0	400
PANDA	1500
PANDA recording	1500

7.1.2 Data Usage of the Virtual Machine Introspection Platform's Recordings

This section presents how the size of the recordings of the guest system increases with regards to time and computations. The recordings were first recorded in the absence of network traffic, and then when network traffic was enabled. In both cases, several recordings of different lengths were recorded. The lengths were five, ten, fifteen, twenty and twenty-five seconds. For each length, five recordings were made. The server was running in both cases. In the case when the network traffic was enabled, a Python script in the client sent 1,000 packets to the server, which also was echoed back from the server to the client. This was executed in all the recordings with network traffic enabled. The time estimated to stop the recordings was measured manually. However, the exact time provided by PANDA shows the the length of thhe recording when a recording is stopped. For each case, three files were produced by a recording and the total size of these three files were measured in MB.

7.1.2.1 Result and Discussion

Table 7.8 presents the mean values of the size of the recordings when network traffic is disabled. Table 7.9 presents the result achieved in the presence of network traffic. The results show that the size of a recording slowly increases with the length of the recordings and that the size of a recording increases when computations in the VM are performed. Figure 7.2 shows how the recordings increase with time and computations, Figure 7.3 and Figure 7.4 show the confidence interval for the recordings with and without network traffic. To obtain the confidence intervals, 5 samples were used and 95 percent confidence was used. The reason why the recordings start at a rather large size is that a snapshot of the VM is taken in the beginning at each recording [19]. Except from this it is the file including the non-deterministic input that grows [19].

Table 7.8: The mean values of the size of the recording in MB in the absence of network traffic.

Time	**Size [MB]**
5	266.56
10	267.38
15	268.00
20	268.70
25	269.58

Table 7.9: The mean values of the size of the recordings in MB in the presence of network traffic.

Time	Size [MB]
5	270.00
10	272.04
15	274.82
20	276.36
25	278.66

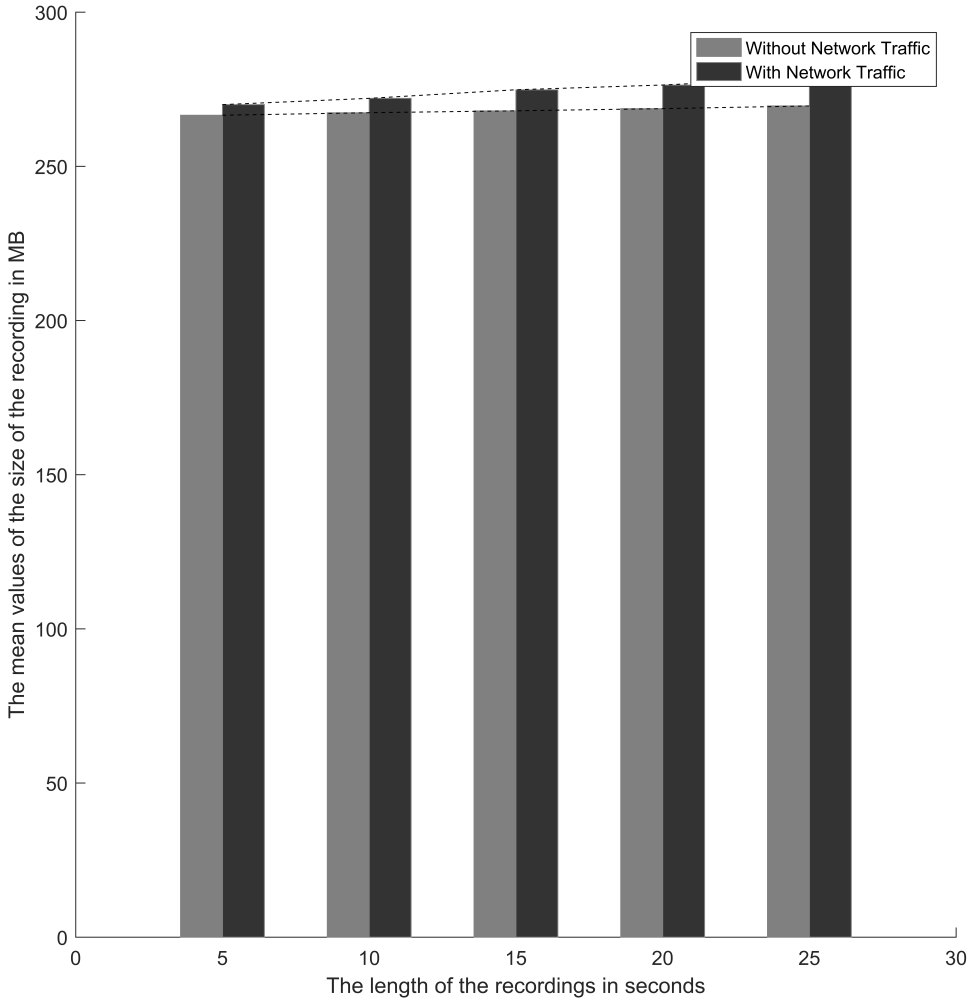

Figure 7.2: Overview of the size of the recordings with and without network traffic.

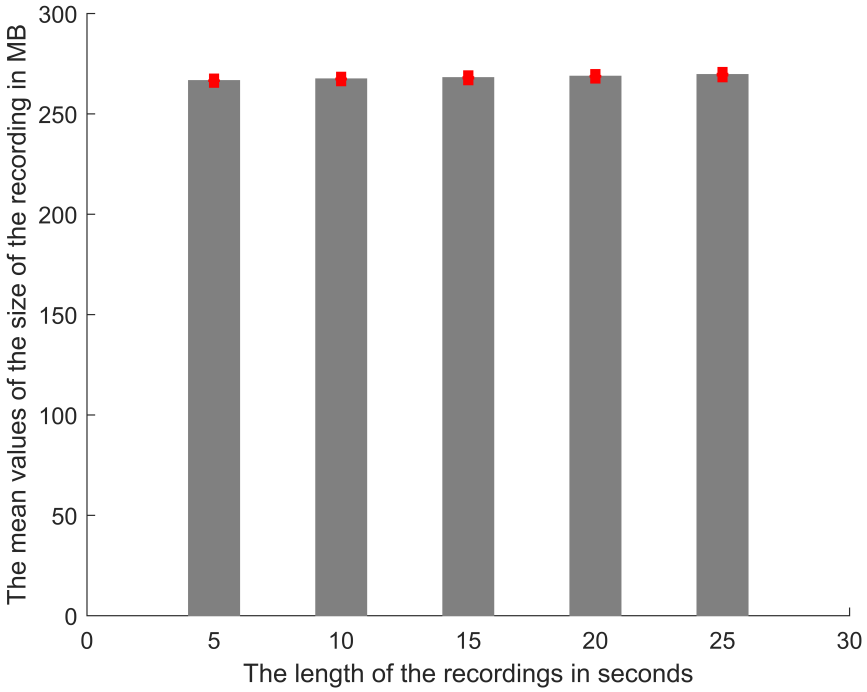

Figure 7.3: Overview of the confidence intervals of the recordings without network traffic. 5 samples, 95 percent confidence.

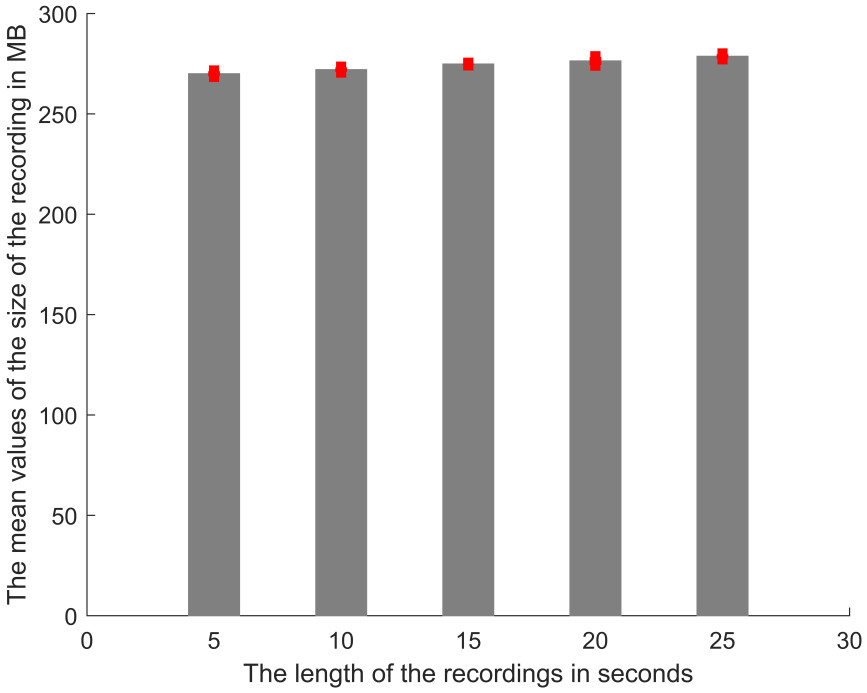

Figure 7.4: Overview of the confidence intervals of the recordings with network traffic. 5 samples, 95 percent confidence.

Remark, the recordings were recorded manually and therefore, the times are not exactly the stated five, ten, fifteen seconds, et cetera. First, the recordings were recorded five times for each of the selected time slots. However, then the size of the five seconds recording was larger than the other size of the longer recordings. As the difference between the recordings larger than five seconds showed a linear increasing pattern, we realized that there must have been some background process running during the five seconds recordings. Instead we made new recordings, were the recordings were recorded from five seconds up til 25 seconds at once and then, we started again from five seconds. Then, it could be showed that the size of the recording was increasing linearly.

7.1.3 Performance of the Virtual Machine Introspection Platform's Tools

This section presents the result of the performance evaluation of the PANDA plugins and the Volatility plugins. The performance evaluation was performed by applying PANDA's plugins and Volatility's plugins for analyzing a recording. The plugins that were chosen are plugins that are not dependable on arguments that differs from each case, such as *stringsearch* does as it requires a document with specific strings to search for. The recording was the same as in Section 7.1.2, where a Python script sends 1,000 messages from the client to the echo server and wait for the client to get the messages echoed back from the echo server. There were several test cases performed on this recording. The first test case includes measuring both the time to execute each plugin separately and the time to execute all plugins combined. This was performed by measuring the time in seconds using a bash script. This was performed 100 times for each test case. Besides, the time to run plugins are printed by the PANDA plugins. This time is presented as well. The plugins were run five times each to gather the mean value of the time provided by PANDA to run each plugin. Besides, the sizes of the data produced by the plugins are presented, which was measured by a bash script. The data produced by the plugins are deterministic, hence it is only necessary to measure it once. Lastly, the memory usage of the plugins are presented, which was measured using the Linux command *top*. It was measured five times to ensure a correct result. In this case the physical memory usage of each plugin is presented, as these plugins could be run on another system than the system running the VM and hence, it is only necessary to show the approximate memory usage of the plugins. Additionally, PANDA prints the memory usage in percent for each percent of the instructions performed for each plugins and it shows the seconds that have elapsed since the start of the plugin. This memory usage is presented and compared to the memory usage gathered for each second of execution by *top*. The memory usage was gathered by running the plugins five times.

7.1.3.1 Result and Discussion

The time for running the PANDA plugins and the Volatility plugins are presented in Table 7.10, Table 7.11, Table 7.12 and Table 7.13. Besides, the confidence intervals of the times are presented in Figure 7.5, Figure 7.6 and Figure 7.7. There

were 100 samples used and a confidence level of 95 percent to obtain the confidence intervals. Table 7.10 shows the total mean time for the PANDA plugins *asidstory*, *memstrings*, *file_taint*, *memsavep* and the total mean time of the Volatility plugins; *linux_enumerate_files*, *linux_proc_maps*, *linux_psaux*, *linux_psenv*, *linux_pslist*, *linux_pstree*, *linux_psview*, *linux_netstat* and *linux_netscan* are shown in Table 7.10. Besides, Table 7.10 shows that the PANDA plugins in general demand more time than the Volatility plugins. However, it is necessary to run the plugin *memsavep* to use the Volatility plugins.

Table 7.11 shows the mean time for some of the utilized PANDA plugins, measured via the script. Table 7.12 shows the mean time for the plugins calculated from the time provided by PANDA. Table 7.12 shows shorter mean time value than Table 7.11, as the script measures the time until the PANDA plugin have stopped running completely. Table 7.11 and Table 7.13 show that there is a great variation among the time consumption in both PANDA's and Volatility's plugins. In general, PANDA's plugins require more time. Besides, table 7.11 shows the time to run the *replaymovie* plugin, that creates a movie, as well as the replay of the system desktop during the recording, itself without applying any plugins. By Figure 7.8 and Figure 7.9 one can see a timeline for how much time each plugin takes in relation to the total amount of time, for both PANDA and Volatility. The time presented in this figure is based on the mean values of the time to run each plugin respective all plugins calculated by running the script.

Table 7.10: All plugins: The mean value of the time to run PANDA plugins and Volatility plugins.

System	**Time** [s]
PANDA plugins	166
Volatility plugins	43
All	209

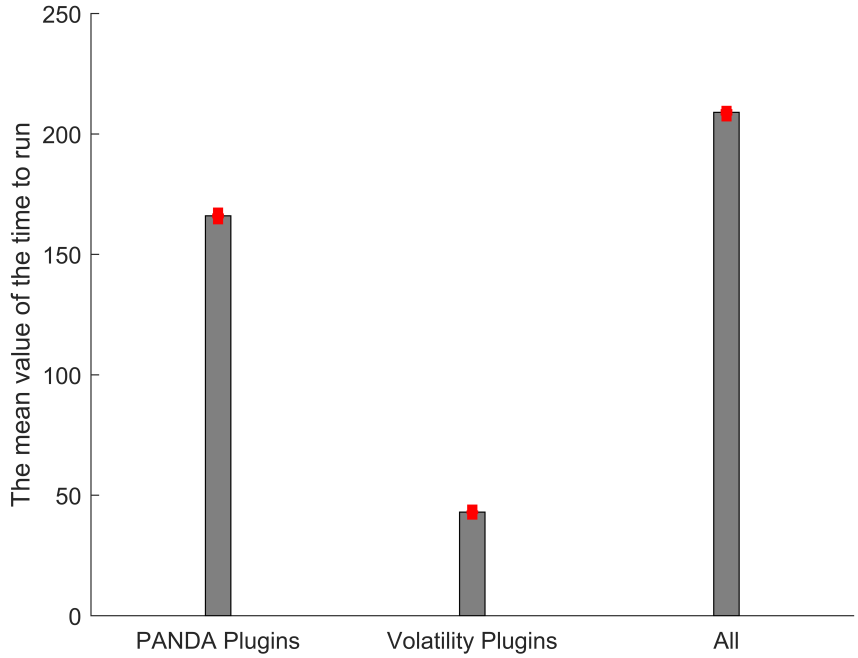

Figure 7.5: All plugins: Confidence intervals for the total time for the PANDA plugins, Volatility plugins and all the plugins. 100 samples, 95 percent confidence.

Table 7.11: PANDA plugins: The mean values of the time to execute PANDA plugins. The time for running replaymovie plugin and create a movie as well as the time to run the replay without any plugins are included.

Plugin	**Time** [s]
asidstory	12
file_taint	64
memsavep	12
memstrings	80
replaymovie	21
replay	10

Table 7.12: PANDA plugins: The mean values of the time to execute PANDA plugins by PANDA. The replaymovie plugin is not included as it requires additional commands.

Plugin	**Time** [s]
asidstory	10
file_taint	62
memsavep	10
memstrings	77
replay	9

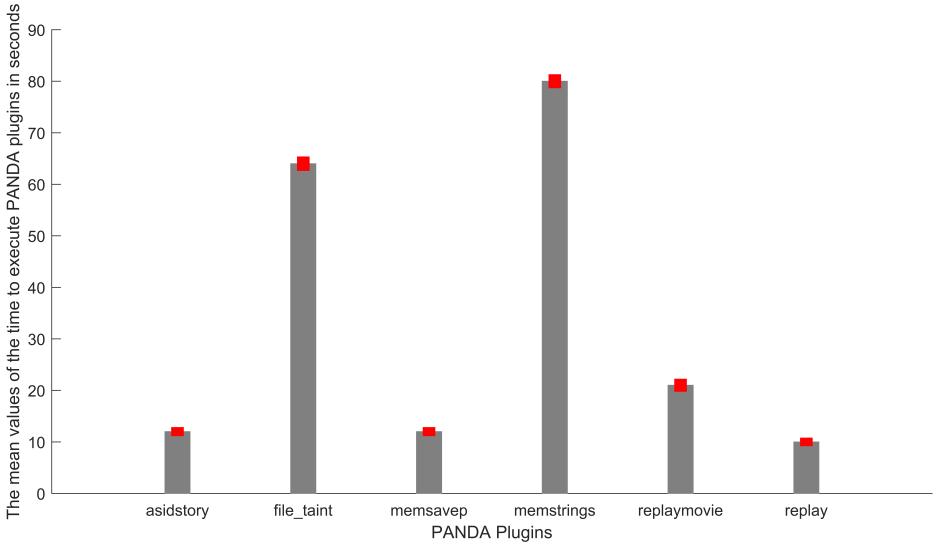

Figure 7.6: PANDA plugins: Confidence intervals of the time to execute each plugin. 100 samples, 95 percent confidence.

Table 7.13: Volatility plugins: The mean values of the time to execute Volatility plugins.

Plugin	**Time** [s]
linux_enumerate_files	6
linux_netscan	4
linux_netstat	8
linux_proc_maps	18
linux_psaux	2
linux_psenv	2
linux_pslist	2
linux_pstree	2
linux_psxview	2

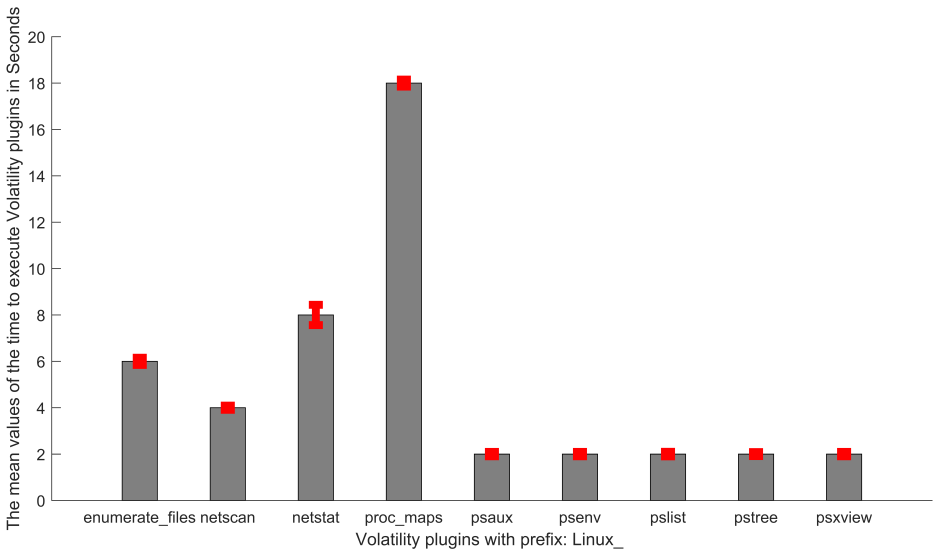

Figure 7.7: Volatility plugins: Confidence intervals of the time to execute each plugin. 100 samples, 95 percent confidence.

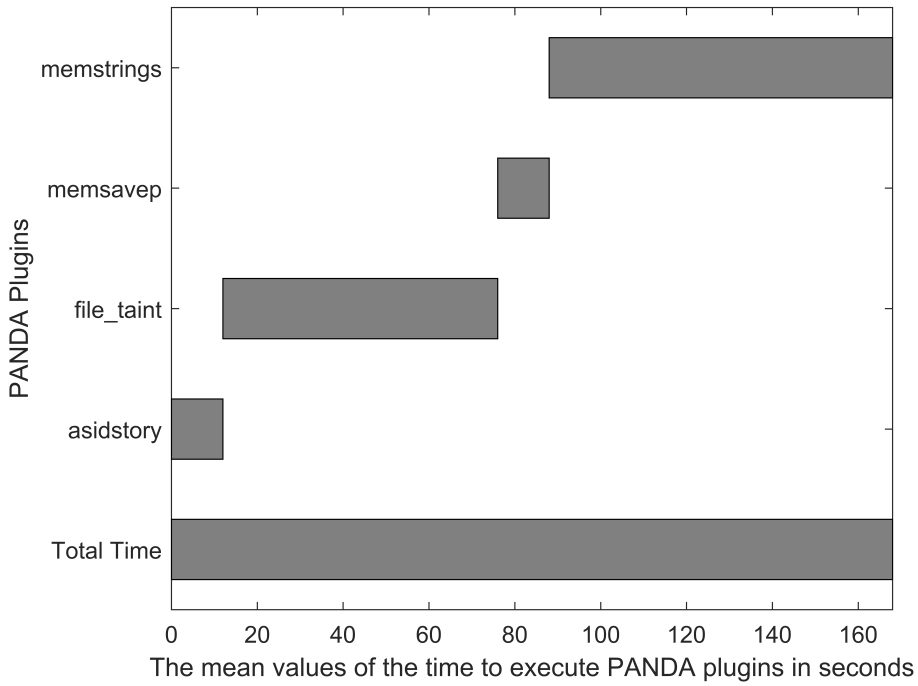

Figure 7.8: PANDA plugins: Overview of the PANDA plugin times in relation to the total amount of time the PANDA plugins take to execute each plugin.

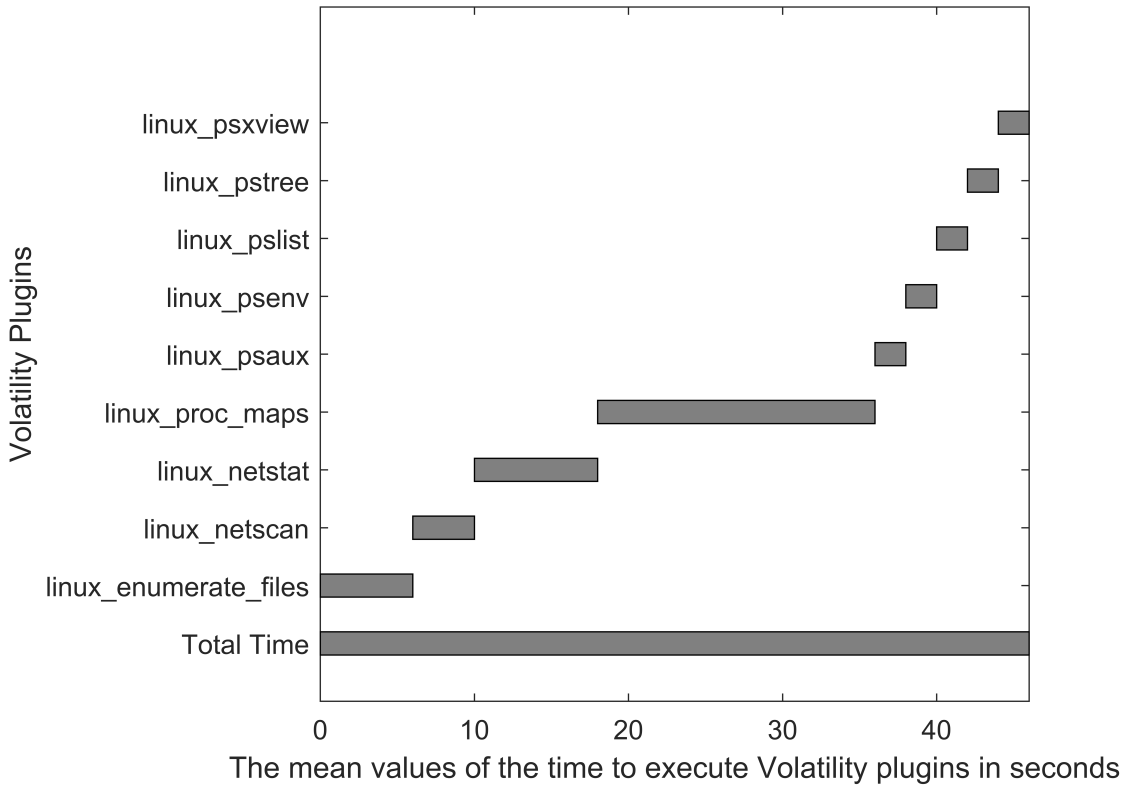

Figure 7.9: Volatility plugins: Overview of the Volatility plugin times in relation to the total time the Volatility plugins take to execute each plugin.

The size of the data produced by the PANDA plugins and Volatility plugins is presented in Table 7.14, Table 7.15 and Table 7.16. These values are deterministic, so it is not necessary to measure them more than once. As can be seen in Table 7.14 the total size of PANDA's plugins are rather larger. However, in Table 7.15 and Table 7.16 one can see that the size of the data produced by different PANDA plugins and Volatility plugins vary quite a lot. Note that the Volatility plugins rely on the raw memory dump produced by *memsavep*. Furthermore, the table shows that the size of the output from *memsavep* is much greater than any other output produced by PANDA plugins or Volatility plugins. Moreover, the outputs from the PANDA plugin *memstrings*, and the Volatility plugins *linux_proc_maps*, and *linux_enumarate_file* are quite large comparing to the rest of the plugins in respective table.

Table 7.14: The total size of the data produced by the PANDA and Volatility plugins.

Platform	**Size [MB]**
PANDA	257
Volatility	2

Table 7.15: Size of the data produced by PANDA plugins. The replay analyzed by the plugins itself is 258,704 MB.

Plugins	**Size [MB]**
asidstory	0.008
file_taint	0.024
memsavep	256.0
memstrings	0.592
replaymovie	0.052

Table 7.16: Size of the data produced by Volatility plugins. The raw memory file that is analyzed is 256 MB.

Plugins	**Size [MB]**
linux_enumerate_files	0.684
linux_netscan	0.004
linux_netstat	0.020
linux_proc_maps	1.200
linux_psaux	0.012
linux_psenv	0.056
linux_pslist	0.016
linux_pstree	0.012
linux_psxview	0.020

The memory usage of the PANDA and Volatility plugins are presented in percent in Table 7.17 and Table 7.20. It is the highest measured memory usage of each plugin that is presented. Worth to note that for *asidstory*, *memsavep*, *memstrings*, and *replaymovie*, the memory usage is constant except from the first one to four seconds. During those seconds the memory usage is 0.1 - 0.2 percent lower than the value presented in Table 7.17 for *asidstory*, *memsavep*, and *replaymovie*, and 0.2 - 0.9 percent lower than presented in 7.17 for *memstrings*. The plugin *file_taint* is constantly increasing its memory usage starting from 3.3 percent and ending at 86.3 percent, after about one minute of execution. In Table 7.17 one can see that the differences in memory usage between the chosen plugins are quite small, except from *file_taint*. However, *file_taint* is not implemented for obtaining information about system calls, as it is originally made for tainting. Table 7.19 shows that the output from PANDA shows a similar memory usage of the plugins. In these cases, the first seconds or percentage of instructions executed by the plugin are also a bit lower than the memory usage during the rest of the execution. For *asidstory*, *memsavep*, and *replaymovie* the memory usage is about 10-20 MB lower during the first 2-4 seconds and for *memstring* the memory usage us about 10-60 MB lower for the first 5 seconds. The plugin *file_taint* grows continuously from 330 MB to 8040 MB. These results are similar to the results from *top*.

Besides, the majority of the Volatility plugins' memory usage is constant, except from the very first second in the majority of the plugins. Then, it is 0.1 - 0.3 percent

lower. *linux_enumerate_files* utilized 1.3 percent of the memory and in the very last second it utilized 1.4 percent. In Table 7.20 one can observe that the memory usage is rather small for the Volatility plugins and the difference between their memory usage is small.

Table 7.17: PANDA plugins: The mean value of the highest measured memory usage of physical memory in percent. Measured by top.

Plugin	**Memory Consumption [percent]**
asidstory	3.3
file_taint	86.3
memsavep	3.3
memstrings	4.5
replaymovie	3.3

Table 7.18: PANDA plugins: The mean value of the highest memory usage of physical memory in MB. Total amount of RAM available for processes is 9634 MB. Measured by top.

Plugin	**Memory Consumption [MB]**
asidstory	322
file_taint	7756
memsavep	318
memstrings	440
replaymovie	325

Table 7.19: PANDA plugins: The mean value of the highest memory usage in MB. Total amount of RAM available for processes is 9634 MB. Measured by PANDA's output

Plugin	**Memory Consumption [MB]**
asidstory	310
file_taint	8040
memsavep	310
memstrings	420
replaymovie	310

Table 7.20: Volatility plugins: The mean value of the highest memory usage in percent. Measured by top.

Plugin	Memory Consumption [percent]
linux_enumerate_files	1.4
linux_netscan	1.6
linux_netstat	1.3
linux_proc_maps	1.3
linux_psaux	1.1
linux_psenv	1.1
linux_pslist	1.0
linux_pstree	1.3
linux_psxview	1.3

Table 7.21: Volatility plugins: The mean value of the highest memory usage of physical memory in MB. Total amount of RAM available for processes is 9634 MB. Measured by top.

Plugin	Memory Consumption [MB]
linux_enumerate_files	136
linux_netscan	157
linux_netstat	126
linux_proc_maps	126
linux_psaux	111
linux_psenv	109
linux_pslist	100
linux_pstree	128
linux_psxview	128

7.1.4 Performance Problems with PANDA

PANDA is not impeccable. By the performance test that was conducted it is clear that the use of PANDA will degrade the performance of the system running as a guest systems. The memory usage increases while using PANDA. Additionally, the execution needs to be recorded in order to be able to perform VMI, which affects the performance. The recording of the system increases the round-trip time, i.e. the responsiveness of the system, which leads to a degradation of the user-experience. Besides, the recordings take up large amount of space and the sizes of the recordings increase with both time and computations in the execution. This means that is is not feasible to record the execution at all time as space will be an issue. The performance analysis also showed that the PANDA platform, with or without recording enabled, also requires additional memory usage compared to a modern version of the VM PANDA is built upon. However, in a server environment with powerful dedicated servers, this might not be such a large problem.

The time, size and memory usage varied among the PANDA and Volatility plugins. It was only one plugin, *file_taint*, that could cause major performance problems. However, this plugin is built for the purpose of tainting, and in this thesis it is used for gathering information about the file system, which means that a more optimized plugin for gathering information about the file system most likely could be implemented. Besides, the plugin architecture allows for selection of which plugins to use and this plugin could be skipped if it causes problems. This is important to think about before using it, as it freezed the whole host system when trying to run it with too little RAM. However, in order to run the plugins one needs the platform and the recording to analyze, which means that the analysis could be done on another host and the use of the plugins will not affect the monitored guest system.

During the time of using PANDA several other problems have been observed. First, there are restrictions in PANDA's performance. It does not support record and replay functionality when hardware simulation via KVM is enabled. Instead, only CPU emulation is used. This leads to restrictions as our dedicated computer was able to run 64-bit Ubuntu 16.04 in PANDA with KVM enabled, but when it was not enabled it was too slow to use and there were also graphical problems. However, this is highly dependable on the host system that is used. With a dedicated host with better hardware this might not be a problem. However, it restricted the choice of OS for this thesis as discussed in Chapter 4, Section 4.2.

Another problem is that PANDA assumes that there exist enough RAM and space for the different plugins, which have led to crashes of the entire host system. In particular, this happened while testing the not deployed plugin *llvm_trace* and by running the plugin *file_taint* without disabling tainting. No error messages or warnings were provided by PANDA either.

Lastly, the recordings sometimes are erroneous, without any specific error message or known reason.

7.2 Overview of the Attack Test Cases

This section presents an overview of the performed test cases of the selected attacks. The majority of the test cases are attacks targeting the echo server in a Linux OS. The two Windows-based test cases, with attacks targeting Internet Explorer, are stated in the text. For most of the test cases, several tests were performed. Some of the tests were performed by using Metasploit, a well-known security project [39].

7.2.1 Attack: Create a New file

A new directory was added to the current directory, i.e. the same folder as the server. The server was closed after the attack with its application specific error message "send: Bad file descriptor".
A new file was added to the current directory, i.e. the same folder as the server

executable. The server was closed after the attack with its application specific error message "send: Bad file descriptor".

7.2.2 Attack: Create a New User

This test was performed three times with different shellcodes, that each add a user with different user names and passwords. The server was running with root privileges. In each test a user was added, which means that the user is attached to the "/etc/passwd"-file. Two of the tests was performed with shellcode generated by Metasploit. The server was closed after the attacks with its application specific error message "send: Bad file descriptor".

7.2.3 Attack: Change Privilege of a File

This test was performed two times with different shellcodes. The server needed to run with root privileges. In the first test case the "/etc/shadow"-file's privilege was changed, from root privilege, to no restrictions. The shellcode of the second test case was generated by Metasploit. The privilege of the "/etc/shadow"-file was changed, to allowing all user to read and write. The server was closed after the attacks with its application specific error message "send: Bad file descriptor".

7.2.4 Attack: Crash a Process

7.2.4.1 Linux

The server application crashed with the application specific error messages "send: Bad file descriptor" and "Segmentation fault".

7.2.4.2 Windows

Internet Explorer crashed, with resulted in that Internet Explorer freezed and a pop-up window showed the error message "Internet Explorer has stopped working".

7.2.5 Attack: Kill all Processes

All processes on the guest system were killed.

7.2.6 Attack: Read a File

Two tests were performed and the server needed to run with root privileges. The first test case prints the content of the file "etc/passwd" to the terminal. The second test case was performed with data generated by Metasploit. The "/etc/shadow" file was printed to the terminal. The server was closed after the attacks with its application specific error message "send: Bad file descriptor".

7.2.7 Attack: Write to a File

This test case is the same as "add a new user" due to the fact that in Linux a new user is added by changing the "/etc/passwd"-file.

7.2.8 Attack: Reboot the System

The system was rebooted. In order to perform this test, the server needs to run as root.

7.2.9 Attack: Start a new Process

7.2.9.1 Linux

Shell This attack started a shell. Due to the fact that the server was running in the terminal, the shell starts in the same terminal and replaces the server. This attack was performed two times, the second time with shellcode generated by Metasploit. The server shows the application specific error message "send: Bad file descriptor".
Bind Shell to Port This attack started a shell on a specific port, which then waited for incoming connections. The port number was given in the shellcode. This attack was performed two times, the second time with shellcode generated by Metasploit. The server prints the application specific error message "send: Bad file descriptor".
Reverse Shell This attack created a reverse shell, i.e. it started a shell that connects to the attacker by IP address and port number given in the shellcode. This attack was performed twice, the second time with shellcode generated by Metasploit. The server prints application specific error message "send: Bad file descriptor".

7.2.9.2 Windows

The Windows standard calculator application started while the attacked application, Internet Explorer, continued to run.

7.2.10 Normal Execution

Two tests with non-malicious data was sent to the server. In the first test short text messages was send to the server. In the other test a string of x86 NOOPS, i.e. "90", was sent.

7.3 Evaluation of Data Gathered from the Network-based Intrusion Detection System

This section presents the result of the data gathered by the chosen NIDS from the different attacks. This data will then be discussed with regards to the accuracy.

7.3.1 Result of the Test Cases

The result of the tests, even for the same attack, varied quite a lot. Several different shellcodes, i.e. attack codes, were deployed for almost every attack and the output from the NIDS depended on the way the attacks were constructed. Some of the attacks were produced by the well-known software Metasploit, and these are marked in the text. Alerts are presented by the given message, category, and priority. The attacks were targeting the server or Internet Explorer. Note that Snort additionally produces a PCAP-file each time an alert is raised, including the suspicious packet.

7.3.1.1 Attack: Create a New file

Test 1 Server: This attack generates two alerts.
`Message : INDICATOR-SHELLCODE x86 NOOP, Classification: Executable Code was Detected, Priority: 1.`
`Message: "Echo Server: Buffer Overflow Detected!", Classification: Attempted User Privilege Gain, Priority: 1.`
Test 2 Server: This attack generates two alerts.
`Message : INDICATOR-SHELLCODE x86 NOOP, Classification: Executable Code was Detected, Priority: 1.`
`Message: "Echo Server: Buffer Overflow Detected!", Classification: Attempted User Privilege Gain, Priority: 1.`

7.3.1.2 Attack: Create a New User

Test 1 Server: This attack generates two alerts.
`Message : INDICATOR-SHELLCODE x86 NOOP, Classification: Executable Code was Detected, Priority: 1.`
`Message: "Echo Server: Buffer Overflow Detected!", Classification: Attempted Administrator Privilege Gain, Priority: 1.`
Test 2 Server - Metasploit: This attack generates two alerts.
`Message : INDICATOR-SHELLCODE x86 NOOP, Classification: Executable Code was Detected, Priority: 1.`
`Message: "Echo Server: Buffer Overflow Detected!", Classification: Attempted Administrator Privilege Gain, Priority: 1.`
Test 3 Server - Metasploit: This attack generates three alerts.
`Message : INDICATOR-SHELLCODE Metasploit payload linux_x86_adduser, Classification: Executable Code was Detected, Priority: 1.`
`Message : INDICATOR-SHELLCODE x86 NOOP, Classification: Executable Code was Detected, Priority: 1.`
`Message: "Echo Server: Buffer Overflow Detected!", Classification: Attempted Administrator Privilege Gain, Priority: 1.`

7.3.1.3 Attack: Change Privilege of a File

Test 1 Server: This attack generates two alerts.

```
Message : INDICATOR-SHELLCODE x86 NOOP, Classification: Executable Code
was Detected, Priority: 1.
Message: "Echo Server: Buffer Overflow Detected!", Classification: At-
tempted Administrator Privilege Gain, Priority: 1.
```

Test 2 Server - Metasploit: This attack generates three alerts.

```
Message : "INDICATOR-SHELLCODE Metasploit payload linux_x86_chmod", Clas-
sification: Executable Code was Detected, Priority: 1.
Message : INDICATOR-SHELLCODE x86 NOOP, Classification: Executable Code
was Detected, Priority: 1.
Message: "Echo Server: Buffer Overflow Detected!", Classification: At-
tempted Administrator Privilege Gain, Priority: 1.
```

7.3.1.4 Attack: Crash a Process

Test 1 Server: This attack only generates the alert made for this server.

```
Message: "Echo Server: Buffer Overflow Detected!", Classification: At-
tempted User Privilege Gain, Priority: 1.
```

Test 2 Internet Explorer : One alert is generated for this attack.

```
Message : "BROWSER-IE Microsoft Internet Explorer deleted button use af-
ter free attempt", Classification: Attempted User Privilege Gain, Pri-
ority: 1.
```

7.3.1.5 Attack: Kill all Processes

Test 1 Server: This attack generates two alerts.

```
Message : INDICATOR-SHELLCODE x86 NOOP, Classification: Executable Code
was Detected, Priority: 1.
Message: "Echo Server: Buffer Overflow Detected!", Classification: At-
tempted User Privilege Gain, Priority: 1.
```

7.3.1.6 Attack: Read a File

Test 1 Server: This attack generates two alerts.

```
Message : INDICATOR-SHELLCODE x86 NOOP, Classification: Executable Code
was Detected, Priority: 1.
Message: "Echo Server: Buffer Overflow Detected!", Classification: At-
tempted Administrator Privilege Gain, Priority: 1.
```

Test 2 Server - Metasploit: This attack generates two alerts.

```
Message : INDICATOR-SHELLCODE x86 NOOP, Classification: Executable Code
was Detected, Priority: 1.
Message: "Echo Server: Buffer Overflow Detected!", Classification: At-
tempted Administrator Privilege Gain, Priority: 1.
```

7.3.1.7 Attack: Reboot the System

Test 1 Server: This attack generates two alerts.
`Message : INDICATOR-SHELLCODE x86 NOOP, Classification: Executable Code was Detected, Priority: 1.`
`Message: "Echo Server: Buffer Overflow Detected!", Classification: Attempted Administrator Privilege Gain, Priority: 1.`

7.3.1.8 Attack: Start a New Process

7.3.1.8.1 Server: Start Shell

Test 1: This attack generates two alerts.
`Message : INDICATOR-SHELLCODE x86 NOOP, Classification: Executable Code was Detected, Priority: 1.`
`Message: "Echo Server: Buffer Overflow Detected!", Classification: Attempted User Privilege Gain, Priority: 1.`
Test 2 - Metasploit : This attack generates three alerts.
`Message : "INDICATOR-SHELLCODE Metasploit payload linux_x86_exec", Classification: Executable Code was Detected, Priority: 1.`
`Message : INDICATOR-SHELLCODE x86 NOOP, Classification: Executable Code was Detected, Priority: 1.`
`Message: "Echo Server: Buffer Overflow Detected!", Classification: Attempted User Privilege Gain, Priority: 1.`

7.3.1.8.2 Server: Bind Shell to Port

Test 1: This attack generates two alerts.
`Message : INDICATOR-SHELLCODE x86 NOOP, Classification: Executable Code was Detected, Priority: 1.`
`Message: "Echo Server: Buffer Overflow Detected!", Classification: Attempted User Privilege Gain, Priority: 1.`
Test 2 - Metasploit: This attack generates three alerts.
`Message : "INDICATOR-SHELLCODE Metasploit payload linux_x86_shell_bind_tcp", Classification: Executable Code was Detected, Priority: 1.`
`Message : "INDICATOR-SHELLCODE x86 NOOP", Classification: Executable Code was Detected, Priority: 1.`
`Message: "Echo Server: Buffer Overflow Detected!", Classification: Attempted User Privilege Gain, Priority: 1.`

7.3.1.8.3 Server: Reverse Shell

Test 1: This attack generates one alert.
`Message : "INDICATOR-SHELLCODE possible /bin/sh shellcode transfer attempt", Classification: Executable Code was Detected, Priority: 1.`
Test 2 - Metasploit This attack generates one alert.
`Message : "INDICATOR-SHELLCODE possible /bin/sh shellcode transfer attempt", Classification: Executable Code was Detected, Priority: 1.`

7.3.1.8.4 Internet Explorer: Start Calculator

Several alert is generated for this attack.
`Message : "BROWSER-IE Microsoft Internet Explorer deleted button use after free attempt", Classification: Attempted User Privilege Gain, Priority: 1.`
`Message : "INDICATOR-SHELLCODE JavaScript var shellcode", Classification: Executable Code was Detected, Priority: 1.`
Four of the following alert: `Message : "INDICATOR-SHELLCODE unescape encoded shellcode", Classification: Executable Code was Detected, Priority: 1.`
`Message : "INDICATOR-SHELLCODE Feng-Shui heap grooming using Oleaut32", Classification: Executable Code was Detected, Priority: 1.`

7.3.1.9 Attack: Write to a File

This test is the same as add a new user, see Section 7.3.1.2.

7.3.1.10 Exchange the NOOPS

The NOOPS was changed from a string of "90" to four different strings of instructions that are no-operations (NOOPS); "4149", "4048", "5058", and "575F". These NOOPs are less deployed for attacks [40]. When the NOOPS were changed to one of the four strings of instructions, no alert for the NOOPS were raised i.e. the message "INDICATOR-SHELLCODE x86 NOOP" did not appear.

7.3.1.11 Normal Execution

Test 1 Server: Sending simple text messages, such as "hello server" to the server. No alert was generated.
Test 2 Server: This test was performed by sending a string of NOOPS, the instruction 90, without any attack code. This generates an alert, even though no attack was made.
`Message : "INDICATOR-SHELLCODE x86 NOOP", Classification: Executable Code was Detected, Priority: 1.`
None of these four strings, no alert about the NOOPS are raised in any of the attacks.

7.3.2 Discussion about the Detection of Attacks

Most of the alerts raised for the attacks against the server were more or less the same alerts. Some attack raised quite specific alerts, which will be discussed in-depth. For each attack an alarm was raised, due to the alert produced for the server vulnerability. Based on the deployed alert for the server vulnerability, it was not possible to produce a false-negative output from the NIDS. However, true-positive, true-negative and false-positive outcomes were possible to obtain.

7.3.2.1 General Alerts

All of the attacks, except the reverse shell attack, raised at least the following two alerts, one that the payload includes x86 NOOP and that buffer overflow was detected. The last alert was the alert produced specifically for the vulnerability in the server. These two alerts provide the information that a buffer overflow is detected as well as that the payload includes NOOPs, which might indicate malicious payload.

7.3.2.2 Detailed Alerts

The reverse shell attacks, included in the attack *start a new process*, raised the same alert stating that there might be an attack that attempts to transfer the shell. This is exactly what the reverse shell tests did, which means that this is a correct and descriptive alert. Furthermore, some versions of the different attacks provide additional alerts. These attacks were; *add a new user*, *change privilege of a file*, and two of the start a new process-tests. For the attacks add a new user, one of the two Metasploit tests raised an alert. The test adds a user named 'metasploit' and raises an alert with a message that the payload includes Linux x86 code with the system call adduser. The second attack to raise an additional alert was *change privilege of a file*. The Metasploit version of the attack raises an alert that the payload includes Linux x86 code with the system call that change the access permissions to files. The third attack to raise additional alerts was *start a new process*. The Metasploit version of the attack that starts a shell raises an alert with a message that Linux x86 payload including the exec system call was detected. Lastly, the attack *start a new process*, raised an alert additionally for the Metasploit test case. The alert provided information that the payload includes Linux x86 shellcode for binding a shell to a TCP port. These additional alerts provide correct and detailed information about the attacks. However, it was only for some of the Metasploit-versions of the attacks that these detailed alerts were raised. This means that only during certain circumstances these alarms are raised, and an attacker that want to avoid these kind of alerts can do so by easily changing the shellcode.

7.3.2.3 Avoiding Alerts

The NOOPs that were used during the previous attacks were exchanged to four different strings of NOOPs. Each of these NOOPs were used in a sequence and replaced the NOOPs in the previous attacks. Then, no alerts that the payload included x86 NOOP was raised. This means that by basically changing the NOOPs to something more less deployed, the NOOPs will not raise an alert. Again, this shows that an attacker can easily avoid alerts by changing the shellcode, in this case the NOOPs.

7.3.2.4 Normal traffic

The normal execution did not raise any alerts. However, when a seres of NOOPs were included in a non-malicious payload to the server, an alert about that the x86 NOOP was included in the payload was raised. This is a false-positive alert. In this case one could argue that as an alert for the buffer overflow vulnerability in the

server was not raised, one could ignore the alert. However, this is not the general case as it might not always be possible to construct alerts that detects all attacks against a vulnerability.

7.3.2.5 Attacks Against Internet Explorer

Both the Windows attacks raised an alert about the use-after-free attempt in Internet Explorer. This was the only alert generated for the crash attack, while the attack *start a new process* raised additionally alerts. The additional alerts include information about that the payload included a specific heap grooming, Javascript shellcode, and unescape encoder shellcode. These alerts do not provide any information the result of the attacks, but they do indicate that the payload might be malicious. However, based on that the *crash a process* attack only raised the alert about the vulnerability and that the *start a new process* attack raised several additional alerts about shellcode being included in the payload, one can argue this can be interpreted that the first attack is a crash attack without any shellcode. However, due to the fact that an attacker always can reconstruct the shellcode, there is always, more or less, ways to avoid raising these alerts about shellcode.

7.4 Evaluation of the Data Gathered from the Virtual Machine Introspection

This section presents the data that could be gathered from the VMI platform in the test cases of different attacks. Most of the test cases were performed in Linux, and two are performed in Windows as well. For the majority of were test cases performed in Linux, several tests were performed. This is due to the fact that different data was sent to the server, for example, different shellcodes for the attacks. The result of the tests are presented. Then, the differences in the data that can be gathered from Linux and Windows are discussed. Lastly, the ability to discover the result of the attacks from the data gathered by VMI are discussed.

7.4.1 Result of the Test Cases

The plugins that was utilized for these test cases are presented in Chapter 6, Section 6.1.

7.4.2 Result of Linux-based Test Cases

The result of the Linux-based test cases against a server and the Windows-based test cases against Internet Explorer are presented.

7.4.2.1 General remarks

Some general remarks about the data produced by the VMI platform can be summarized from the tests.

The plugin *memstrings* can provide information about each attack such as paths to commands and files. These are presented for each test case; however it is hard to draw any conclusions about specific attacks against the server by this. By using the plugin *stringsearch* it is possible to search for these strings to find out which process that is reading or writing the string to memory. However, this is an inconvenient method as one first needs to find file names manually, before using *stringsearch*. In order to use *stringsearch* a document with the file names and command paths to search after needs to be made and the document is one of the arguments to *stringsearch*. The plugins *memstrings* and *stringsearch* could also be used to show the attack code. Besides, the plugin *network* can also be used to recreate all network packets during a recording, which makes it possible to inspect the attack code.

After each attack the server terminated, which were possible to observe by utilizing one of the following plugins: *linux_psaux*, *linux_psenv*, *linux_pslist*, *linux_pstree*, *linux_psview*, *linux_netstat* or *linux_netscan*, which showed that the server and the server's ports are no longer active. By using the plugin *asidstory* it is also possible to observe that the server was not active during the last part of the recordings when it was terminated. However, as this plugin shows processes as inactive when they are not computing something, this does not indicate that the server was terminated. The *asidstory* plugin also provides information about each process such as its ASID identification, which is useful when using plugins such as *file_taint* and *stringsearch*.

Before the server terminated after each attack, it printed a specific error message. By using *memstrings* the server specific error message can be seen, or searched for. This error message could also can be searched for by using *stringsearch*, which also provides additional information such as if it was a read or written string to memory and by which process. By using the *replaymovie* plugin it is possible to see that the server prints the application specific error message before it shuts down.

7.4.2.2 Attack: Create a New file

- **The memstring plugin** shows the new file names.
- **The stringsearch plugin** provide a way to search for the file names and one can see that the server process has been writing and reading the file names in memory.
- **The memsavep plugin** creates a raw memory file including 99.9 percent of the recording. The raw memory file can be analyzed with Volatility.
- **The Volatility plugin linux_enumerate_files** shows the new paths to the files that were created in the test cases.

7.4.2.3 Attack: Create a New User

- **The memstring plugin** shows the new user names, the path to the password file et cetera.
- **The stringsearch plugin** provides a way to search for the user names and command paths and one can see that the server process has been writing and

reading the user names and command paths in memory.
- **The asidstory plugin** can extract the processes's ASIDs.
- **The file_taint plugin**, with no arguments except notaint, can show that the password file that was opened and the process's ASID. The ASID corresponds to the server, which normally should never interfere with this file.

7.4.2.4 Attack: Change Privilege of a File

Could not be detected.
- **The memstring plugin** shows the paths to the changed files.
- **The stringsearch plugin** provides a way to search for the path to the altered files and one can see that the sever process has been writing and reading the path to altered files in memory. However, as *file_taint* do not provide any information about the server opening and reading the altered file it is hard to draw any conclusions about this.

7.4.2.5 Attack: Crash a Process

- **The memstring plugin** shows the "Segmentation fault" string.
- **The replaymovie plugin** shows that the server process closes with a segmentation fault.
- **The stringsearch plugin**, by searching after the "segmentation-fault string", can show if the "segmentation fault"-string where written/read into memory by the server.

7.4.2.6 Attack: Kill all Processes

For this attack it is only possible to utilize the *asidstory* plugin as well as the *replaymovie* plugin, as the recording is erroneous.

- **The asidstory plugin** shows that all the processes stopped being active after approximately a third of the recording. This is abnormal compared to a normal execution and indicates that the system is shutdown.
- **The replaymovie plugin** shows that the normal window of the system is replaced with a full screen picture of the background, which happens when the system either is starting or terminating.

7.4.2.7 Attack: Read a File

- **The asidstory plugin** shows the ASID of the server.
- **The memstrings plugin** finds the path to the read files and paths to the commands, as well as the full content of the read files presented coherently. The last thing is abnormal as it means that the content is read or written into memory.
- **The stringsearch plugin** provides a way to search for the paths to the read files and paths to the commands as well as the content of the file that was read, and shows that the server was writing/reading the strings in memory.

- **The file_taint plugin**, with no arguments except notaint, shows the file that was opened and that the files are read by an ASID that corresponds to the sever's ASID.
- **The replaymovie plugin** shows how the file's content is printed to the terminal and that the server closes.

7.4.2.8 Attack: Reboot the System

For this attack it is only possibly to run the *asidstory* plugin, as the other plugins fails while running the plugin or create erroneous files.

- **The asidstory plugin** shows that all processes stops being active in the beginning of the recording. This is abnormal compared to a normal execution, and indicates that the system is shut down.

7.4.2.9 Attack: Start a New Process

Several attacks were performed that started different processes; a local shell, a shell that binds to a port and a reverse shell.

7.4.2.9.1 Start Local Shell

- **The asidstory plugin** shows the started shell.
- **The memstrings plugin** shows the path to the command to start a shell.
- **The stringsearch plugin** provides a way to search for the paths to the command as well and displays that the server was writing/reading the string in memory.
- **The replaymovie plugin** shows the server it printing the specific error message and then, a shell is started in the terminal.
- **The memsavep plugin** creates a raw memory file including 99.9 percent of the recording. The raw memory file can be analyzed with Volatility.
- **The Volatility plugins linux_psaux, linux_psenv, linux_pslist, linux_pstree, linux_psview** show that the new shell process is active. linux_pstree does not show the parent-child relationship between the processes as the server has terminated. linux_pslist shows the exact start time of the process.
- **The Volatility plugin linux_netstat** shows that the shell is listening on the same port as the server did.
- **The Volatility plugin linux_netscan** shows that the port is still active.

7.4.2.9.2 Bind Shell to a Port

The outcome of this attack depends on if the attacker connects to the port or not during the recording. If the attacker connects to the port the following can be seen using plugins:

- **The asidstory plugin** shows the started shell.
- **The memstrings plugin** shows the path to the command to start a shell.

- **The stringsearch plugin** provides a way to search for the paths to the command as well and see that the server was writing/reading the string in memory.
- **The memsavep plugin** creates a raw memory file including 99.9 percent of the recording. The raw memory file can be analyzed with Volatility.
- **The Volatility plugins linux_psaux, linux_psenv, linux_pslist, linux_pstree, linux_psview** show that the new shell process is active. If the attacker has not connected to the port these plugins show the server as active. linux_pstree does not show the parent-child relationship between the processes as the server has terminated. linux_pslist shows the exact start time of the process.
- **The memsavep plugin** creates a raw memory file including 99.9 percent of the recording. The raw memory file can be analyzed with Volatility.
- **The Volatility plugin linux_netstat** shows that the shell is listening on the same port as the server did and it also shows the newly opened port the shell is bound to.
- **The Volatility plugin linux_netscan** shows that the attacked application's port is still active and also shows the new port.

7.4.2.9.3 Reverse Shell

- **The asidstory plugin** shows the started shell.
- **The memstrings plugin** shows the path to the command to start a shell.
- **The stringsearch plugin** provides a way to search for the paths to the command as well as showing that the server was writing/reading the string in memory.
- **The memsavep plugin** creates a raw memory file including 99.9 percent of the recording. The raw memory file can be analyzed with Volatility.
- **The Volatility plugins linux_psaux, linux_psenv, linux_pslist, linux_pstree, linux_psview** show that the new shell process is active. linux_pstree does not show the parent-child relationship between the processes as the server has terminated. linux_pslist shows the exact start time of the process.
- **The Volatility plugin linux_netstat** shows that the shell has established a connection, and to which IP address and port number. It states that the shell also listens to the server's port.
- **The Volatility plugin linux_netscan** shows the server's port as well as the new connection established by the attack.

7.4.2.10 Write to a file

See Section 7.4.2.3, as this is the same test case.

- **The memstring plugin** shows the path to the files etc.
- **The stringsearch plugin** provides a way to search for the paths to the files as well and shows that the server was writing/reading the strings in memory.
- **The asidstory plugin** can extract the processes' ASID.

- **The plugin file_taint** shows that the file is opened and the identification of the process that opened the file (the ASID-identification, where the ASID of a process is provided by asidstory).

7.4.2.11 Normal Execution

This section describes the data generated by the plugins during a normal execution. Several processes including the server are active during various parts of the recordings, which can be seen in the output of *asidstory*. *memstrings* does not generate output including any server specific error messages or "Segmentation fault". *file_taint*, with no arguments except notaint, does not generate any data about the server being involved in open or reading files. *replaymovie* shows that the server is running as normal.

By creating a raw memory file with the *memsavep* plugin, including 99.9 percent of the recording, it is possible to observe that the server is active by utilizing one of the following Volatility plugins: *linux_psaux*, *linux_psenv*, *linux_pslist*, *linux_pstree* or *linux_psview*. Besides, the server, including its IP address and port number and the state of the connections made to the server, is visible in the plugins *linux_netscan* and *linux_netstat*.

7.4.3 Result of Windows-based Test Cases

The result of the Windows-based test cases against Internet Explorer.

7.4.3.1 Crash a Process

- **The plugin replaymovie** shows that the user visits a malicious web page in Internet Explorer and clicks on a malicious link. The browser crashes, and a new window is showing with the message "Internet Explorer has stopped working", where the user can chose to close the program or check online for solutions. The recording stops when this dialog was shown.
- **The plugin asidstory** shows that Internet Explorer was not active the last part of the execution. Nonetheless, this does not prove that Internet Explorer crashed.
- **The plugin memstring** makes it possible to extract the string "Internet Explorer has stopped working".
- **The plugin stringsearch** makes it possible to extract the string "Internet Explorer has stopped working", by searching after the string or a part of the string. The plugin shows that Internet Explorer was reading and writing the string in memory.
- **The win7proc plugin** can be used with the script "procstory" to see started and terminated processes, however the crash of Internet Explorer is not visible. This is due to the way Windows handles applications that crash, i.e. the user needs to choose to terminate the process manually. The win7proc plugin can regenerate the system calls during the execution and it is possible to see that the malicious file was created under the directory "Temporary Internet files".

The win7proc plugin can regenerate the files written in the system during the execution. The output included the malicious file causing the attack.
- **The memsavep plugin** creates a raw memory file including 99.9 percent of the recording. The raw memory file can be analyzed with Volatility.
- **The Volatility plugins iehistory, windows, wintree** show the last visited web page.
- **The Volatility plugin filescan** shows the malicious file causing the attack was saved into "Temporary Internet Files".
- **The plugins pslist and pscan** show start and exit times for processes, but the crash of the IE is not visible here due to the Internet Explorer not being properly terminated.

7.4.3.2 Start a New Process

One test that started a new process was performed.

7.4.3.2.1 Calculator Application

- **The plugin replaymovie** shows the user being on a malicious web page. The recording ends after the user have clicked on a malicious link, which made the calculator application start running.
- **The plugin asidstory** shows that the calculator process is running in the system, and that it was active during the second part of the recording.
- **The win7proc plugin** can be used with the script "procstory". It is possible to see that the calculator process was created after approximately half of the time in the recording. In the list of "interesting" and "boring" processes one can see that the Internet Explorer was starting the calculator. The win7proc plugin can regenerate the files written in the system during the execution. The output included the malicious files causing the attack. Besides, it was possible to read one of the files using a text editor. Additionally, the win7proc can generate the system calls during the execution. From the system calls from the "win7proc" plugin one can observe that Internet Explorer writes to and read the malicious files causing the attack to "Temporary Internet files"-directory. It also shows that Internet Explorer calls "nt_create_user_process" to create the calculator process, as well as interacting with the calculator process via the system calls "nt_open_key" and "nt_write_virtual_memory".
- **The tool STUW** generates the interprocess system calls between the calculator and the Internet Explorer "nt_open_key" and " nt_open_key_ex". The output from the same tool also shows the interprocess communication between IE and the malicious files causing the attack from the "Temporary Internet Files" directory.
- **The memsavep plugin** creates a raw memory file including 99.9 percent of the recording. The raw memory file can be analyzed with Volatility.
- **The Volatility plugins iehistory, windows, wintree** one can conclude the last visited web page.

- **The Volatility plugin filescan** shows that the files that caused the attack, were saved into "Temporary Internet Files".
- **The Volatility plugins pslist, psscan, pstree, and psxview** it is possible to see that the calculator process is running in the system. The first three plugins also produce information about when the process started running. The plugin "pstree" prints the processes as a tree, shows Internet Explorer and calculator as independent processes. This contradicts the information obtained by the "win7proc" plugin in PANDA, where it was Internet Explorer that started the calculator process.

7.4.3.3 Additional Results

The plugin win7proc that shows the system calls executed in the system shows that it is possible to extract which process that is reading, writing or creating a file

7.4.4 Comparison of Data from Different Operating Systems from the Virtual Machine Introspection

The data that could be gathered for Windows-based respective Linux-based attacks *Crash a process* and *Start a new process* were quite different. The main difference is that for recordings of execution in 32-bit Windows 7 operating system, the very useful plugin *win7proc* exists. It utilizes system calls for providing an overview of the system's state. It is possible to get a great overview of the start and termination of processes in relation to the time elapsed during the recording. However, the crash of a process in Windows is not visible here. The start of a new process is visible. Besides, the parent-child relationship between the processes are visible, which can show if a process started another process. If a process have started another process this can be seen both illustrated, and it can also be seen by studying the system calls themselves. The relationship between processes and the start and termination of processes are not possible by the cross-platform plugin *asidstory*. *asidstory* only shows all processes that existed during the execution as well as when they were active. From the system calls, which among other things shows system calls related to the file-system, it is possible to see which process that wrote or read a file. Besides, it is also possible to see which process that created a file. It might be possible to see additional interesting system calls related to the file-system. Lastly, the plugin regenerates the files written during the execution. Depending on the file itself, it might be possible to read it as well. Besides, the tool *STUW* can, based on the output from *win7proc*, additionally show the interprocess communication.

In Linux, a similar plugin such as *win7proc* does not currently exists. By the plugin *file_taint*, which actually is made for tainting, it is possible to see which process opens or reads which files. However, the result of the attack that created a file was not shown by this plugin. Moreover, this plugin is not optimized for showing system calls and it does also not show the processes directly, only the ASID. If a plugin like *win7proc* was implemented for Linux, given that it is possible, it would provide a better view of the result of a lot of the attacks that were tested. Based on the data

provided by *win7proc*, at least the following attacks might be better detected by PANDA; create a file, write to a file, read a file, crash a process and the parent-child relationship between the attacks that start a new process.

Lastly, the plugins provided by Volatility are different for different OSes. However, for most of the ones utilized in this thesis, there exist a similar version of each plugin for both Windows and Linux systems.

7.4.5 Ability to Detect Result of Attacks

This section discusses the result of the different test cases, and the conclusions that are possible to draw by analyzing the data gathered by VMI. The output of the different attacks varied, which means that some attacks could be detected easier than others. Besides, the output varied depending on the OS.

7.4.5.1 File System

The attacks involving files had various results. The attack *create a file* was only possible to observe by using Volatility, which provided data including the new paths to the files created. In the attack *read a file*, it was possible to extract the files that was opened and read by the server process from the PANDA plugin *file_taint*. The same plugin shows the attack *write to a file*, that the file was opened by the server process. However, this does not imply that the file was written to. The attack *change privilege of a file* was not possible to observe at all using PANDA or Volatility.

7.4.5.2 System User

It was not possible to extract the new user or observe that the user file was changed in the attack *add a new user*. However, by the PANDA plugin *file_taint*, one can see that the password file was opened by the server process. This does not necessarily imply that a new user was added, but it is suspicious as the server should not interfere with this file.

7.4.5.3 Crash of Processes

The crash of a process was only visible in PANDA by utilizing the plugin *memstring*, or *stringsearch*, to search for the standard error message by the specific plugin and OS. In Volatility for Linux it was possible to see that the server process has stopped running, as all the process-related plugins do not show the server application as an active process. However, due the fact that Internet Explorer process in Windows does not terminate when it has crashed, it was not visible in the Volatility plugins for Windows. Besides, it was not visible in the Windows specific plugin *win7proc* either, which otherwise shows the termination of processes. However, it was still

possible to search for the error message in *memstring* or *stringsearch*. As *memstrings* do not show which process that wrote the error message, *stringsearch* could be used if it is necessary to see which process that wrote the error and hence, crashed.

7.4.5.4 Start of New Processes

The start of a new process was visible by both PANDA and Volatility. In PANDA the plugin *asidstory* provides a list of all processes that existed during the recording. However, no parent-child relationship was presented and when the process was started was not presented. For Windows, an additional plugin is available, *win7proc*, which presents the parent-child relationship between processes and presents the time when a new process started during the execution. This provides useful information about which process that started a process and when. By the process-related plugins in Volatility, for both the Windows-based and Linux-based test cases, it was possible to see the new processes as well as their start time. One of these plugins provides information about the relationship between the processes. However, this plugin did not show the relationship between Internet Explorer and the newly started process. The server application stopped when it started a new process, and hence the parent-child relationship was not shown either. Besides, for the attacks that started a shell, the shell process was visible in Volatility's network connection plugins.

7.4.5.5 System Shutdown

In the attacks where the system is rebooted or forced to shut down, PANDA stops working during the close down. This means that no plugins, except from the ones writing output such as *asidstory*, is working. *asidstory*, shows that all processes stop being active during the same time during the recording, which is exactly what happens. One could argue that it would be enough to draw the conclusion that PANDA has been shut down by observing the output from the plugins in the terminal, which provides an error message. However, due to unknown circumstances, recordings in PANDA are erroneous. These recordings have the same problems with not being able to run any plugins. However, in these cases, the output from *asidstory* is empty and are not showing any processes at all. Hence, the output from *asidstory* here provides more certainty to the fact that the system has been shut down. However, there is no data provided that shows the difference between a reboot and a forced shut down.

7.4.5.6 Termination of Processes

The server terminated after all the attacks due to an error when the server tries to send the received data back. This results in that the server prints an error message to the terminal and terminates. Hence, the server was shown as idle in the last part of the recordings in the output from the plugin *asidstory*. By searching in the output from *memstrings*, or by utilizing the plugin *stringsearch*, it was possible to

see the error message from the server. In some cases it was possible to see that the server has stopped working as a string including the server's error message directly followed by a new prompt is written to the memory. Besides, the process-related plugins in Volatility showed the server as a non-active process, i.e. terminated.

7.4.5.7 Normal Traffic

For a normal execution, processes should be running and be active, i.e. visible in both *asidstory* and the process plugins provided by Volatility. No signs of crucial error messages by the server or crash messages should be visible by using *memstrings* or *stringsearch*, and *file_taint* should not show any interference with suspicious files. Also, the Volatility plugins that show network connections should show the sever as an active process on the correct port number. However, certain attacks such as *change privilege of a file* was not possible to observe in PANDA, which means that it is not possible to be certain that the execution is normal.

7.4.5.8 Ambiguous Results

Lastly, a discussion about the results from the plugins *asidstory*, *memstrings* and *stringsearch* follows. By studying the output from *asidstory* one can see during which times of the recordings that each process was active. Hence, one might assume that a certain process has crashed or terminated if it stops being active during the recording. However, this is not always the case as the processes are marked as non-active while they are idle. This means that only during certain circumstances, such as when having a process that normally should always be active, the output from *asidstory* is useful. However, for most cases this could not be used to prove that an application have terminated. Another problem with *asidstory* is that it sometimes shows several instances of the same processes, and that the process name might vary slightly. This could create problems when creating scripts or programs for automatically extracting the information from the data.

memstrings which prints everything written to or read from the memory is able to show the path in the shellcode, the shellcode itself as well as the new user names, and files. However, it is very hard to discover these things as it requires manually searching through the output. It is also not possible to extract which process that were reading or writing the strings in *memstrings*. However, this could be done by using *stringsearch*, by searching for the abnormal strings. The main problem is still to find the more specific strings, such as file names, in the first place. It is also hard to know if the string belongs to a malicious behaviour or if it is normal input. However, paths to command is more suspicious, even though it could be a part of normal input. The problem with searching for paths to different commands is that the numbers of matches are so large, that the plugin becomes really slow and the size of the output increases. But, both *memstring* and *stringsearch* are beneficial to be used for searching for crash error messages or similar.

7.5 Applicability of Combining the Systems

This section discusses the applicability of using VMI as an extension to NIDS and how NIDS and VMI could be combined in the future, with regards to the research questions. While the result of the different attacks showed that the alerts raised by the NIDS varied depending on how the attacks were constructed, VMI can always show the state of the system. However, the detectability of the different attacks by using the data gathered from the VMI varied. The data gathered from the Windows-based test cases showed potential to provide more detailed information about the result of the attacks, than the data gathered from Linux-based test cases did. However, more tools need to be developed for PANDA to be able to provide more data about different attacks for all platforms.

7.5.1 Research Question: Is it Benefical to Use Virtual Machine Introspection

The first research question is if it is beneficial to use VMI. To be able to answer this question, a performance analysis of PANDA was performed in Section 7.1, and the result will be discussed here. The first test showed that the responsiveness of running in the PANDA platform, while recording was about 21-26 percent slower than running PANDA without recording. As it is necessary to record the execution for performing VMI, the system's round-trip time will be slower. Also, PANDA requires about three-four times more memory usage than running in a modern version of the VM PANDA is built upon. The second test case showed that the output from the recordings grow linear both with regards to time and computations. The third test case showed the performance of the plugins, with regards to time to execute, size of output and memory usage. The time, size and memory usage varies quite a lot between the plugins. The result of the performance analysis shows that by using the VM platform for VMI, the responsiveness and memory usage are worse compared to other modern VMs. Additionally, recordings will require an increasing amount of space. Plugins for gathering interesting data can be chosen based on the performance and the data they produce, which means that this is highly modular.

7.5.2 Research Question: What Kind of Data Can be Gathered from Virtual Machine Introspection

The second research question was what kind of data that can be gathered from VMI, i.e. PANDA. The result of this was discussed in detail in Section 7.4.5. To conclude the result of the test cases, the result is presented in short. For the Linux-based test cases the following attacks where detectable; add files, crash a process, start of a new process. For the following attacks useful information was provided; add a new user, reboot, kill all processes. The attack *change privilege of a file* was not detectable. However, due to the fact that the server terminated due to the buffer overflow, the data produced by PANDA could still provide useful information. That

a recorded execution is a normal execution can not be guaranteed by using VMI, due to certain attacks such as *change privilege of a file* are not detectable by the data produced by PANDA. In this specific case, the server terminates after all attacks due to buffer overflow, but this does not apply to all applications. Both of the attacks that were tested in Windows OS, i.e. crash a process and start a new process, were detectable. Based on the data gathered from the output of those, it might be possible to discover more attacks performed in a Windows OS, such as which processes that read, write, and create files. As the data includes system calls involving processes, files, et cetera, it might be possible to detect even more attacks. The data gathered from PANDA can provide us with a broad view of the system state, and the Windows-specific plugin that could provide data based on system calls shows the potential of showing even more specific information.

7.5.3 Research Question: How and When to Combine the Systems

The last research question is, based on the result of the previous research questions, how and when the two systems should be combined. The data gathered from PANDA provides useful information about the system's state and could detect several of the attacks. PANDA provided accurate information for each of the categories of NIDS output, and it could be used to discover that an alert was false-positive, confirm that the system were operating as normal in case of a true-negative result and could provide details about the state of the system in case of a true-positive alert. False-negatives were not tested, however the result shows that it should be possible to detect that as long as the attack is one of the attacks that was tested. Hence, it could be used as an extension to Snort in order to always provide a view of the state of the monitored system as Snort did not provide that much detailed information for most of the alerts. However, in order to perform VMI, i.e. utilize the plugins, the monitored OS needs to be run in PANDA and a recording needs to be made of the part of the execution that is interesting. This is a problem as it makes the responsiveness of the guest system shrink with over 20 percent, the memory usage of PANDA is about four times higher than other modern VMs, and the recordings increases continuously with regards to time and computation. This means that it might be unfeasible to always record the monitored system. In addition to this, the data gathering by the plugins require time, space, and memory usage. However, due to the plugin architecture of PANDA, it is possible to choose which plugins to run, depending on what information one wants to gather, and the performance of the plugins. The plugins could be used on another host with PANDA installed, so it does not have to run on the same host as the monitored system.

Because of PANDA's performance, its VMI functionality should only be used during certain circumstances. It could be used only when alerts are raised or when other suspicious network traffic is appearing, such as external network traffic or any other traffic that is not considered trusted. This would make it possible to detect the state of the system after a potential attack, as PANDA could detect the result in

the system of several of the attacks tested while Snort only could mainly detect attacks targeting vulnerabilities. Besides, false-positive alerts could be dismissed. However, false-negative outcome from Snort would be missed. The rules in Snort could be improved by using the data from PANDA in addition to the network packets that caused the alert. Additionally, it could be more limited to only be used for a certain application on a certain port in the system. Figure 7.10 shows an example of how the system could be combined.

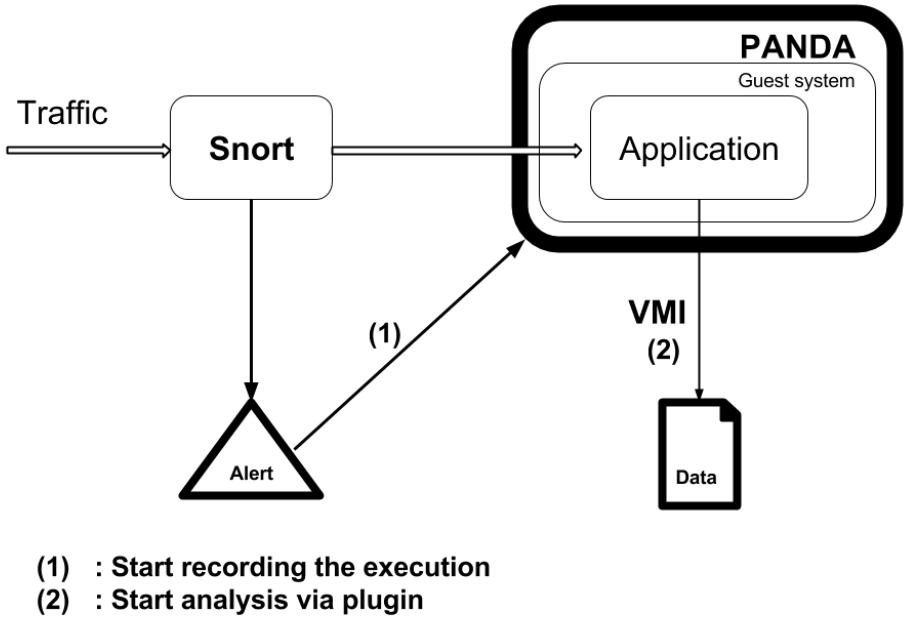

Figure 7.10: Overview of the suggested combined system.

Lastly, if it was beneficial to use PANDA constantly, it would be possible to catch both false-negative and false-positive, and get an accurate view of the system's state after each true-positive alert. It would also be possible to define new rules for false-negative results, by comparing the outcome from SNORT and PANDA. Note that this means that each network packet would trigger data from PANDA to be produced. Depending on how many network packets the monitored system receives and what data that is selected to be produced by the VMI, i.e. the plugins in PANDA, a lot of data could be produced constantly. This requires a future combined system, where VMI could be performed live and the system would need to analyze the data produced rather quickly in order to be able to provide accurate information within a certain amount of time to discover attacks and provide accurate information about the potential attacks discovered by Snort. Currently, PANDA has only a few, very limited, plugins that can run live and for the most of the systems of today, it would be rather hard to analyze the outcome of all incoming network packets.

8

Conclusion and Future Work

This chapter discusses essential future work in the research area towards a combined system of virtual machine introspection and network-based intrusion detection systems, and then concludes this thesis with a summary of the work and the result.

8.1 Future Work

The result in this thesis shows that virtual machine introspection could be used to get a broader view of the system, and detect the result of certain attacks. However, it would be beneficial to test more attacks to continue the research about how virtual machine introspection can provide useful data about the system's state. Due to the fact that Windows has an additional plugin that showed potential to be able to detect more of the selected attacks, it would be beneficial to test all the selected attacks in this thesis in a Windows operating system as well. Furthermore, more applications need to be selected to provide a broader view of when virtual machine introspection can produce useful information.

As additional future work, more virtual machine introspection plugins, dedicated for providing information required to detect the result of attacks, could be implemented. The main problem is that the data provided by virtual machine introspection is low-level information that needs to be extracted by the plugins to provide high-level information, i.e. there is a semantic gap problem. Such plugins are needed to produce useful data involving the file system and processes. One example of a plugin that would provide useful information is a system calls plugin for Linux, similar to the existing system calls plugin available for Windows-based recordings. Another benefical plugin would be one that can detect different types of attacks, i.e. buffer overflow et cetera, by utilizing the low-level information provided by PANDA. Such plugins would be beneficial in a future system where virtual machine introspection and network-based intrusion detection systems could be running in conjunction and continuously update the rules in the NIDS. There exists one such plugin, *useafterfree*, which where discussed in Chapter 3 Section 3.1.2.4

Lastly, as discussed in Chapter 7, Section 7.1.3 there are some performance issues in PANDA, mainly due to the fact that the execution needs to be recorded to be able to perform virtual machine introspection, i.e run plugins. This restricts the usage of PANDA's virtual machine introspection. Hence, more research in how to make the plugins run live is needed in order to be able to utilize virtual machine introspection

at any time. PANDA currently includes some experimental plugins that can run their analysis live on a system. However, these plugins have limited functionality in order to not slow down the introspected system.

8.2 Conclusion

This thesis lays the foundation for research towards a combined system of virtual machine introspection and network-based intrusion detection systems. The goals of this thesis were to investigate how virtual machine introspection could provide a broader view of the state-of-the-art network-based intrusion detection systems of today, and how data gathered from virtual machine introspection and network-based intrusion detection systems could be combined. By analyzing available state-of-the-art network-based intrusion detection systems and virtual machine introspection in detail, the platforms of this thesis were chosen. In order to gather useful data, a selection of an interesting application to run the tests against was chosen. Furthermore, the interesting events i.e. attacks and normal execution were chosen. Besides, in order to test the network-based intrusion detection systems thoroughly, different versions of each test case were tested. Also, different outputs from the network-based intrusion detection systems were tested with regards to true and false outputs. By analyzing the output from the virtual machine introspection thoroughly, and compare it to the output from the network-based intrusion detection systems. Also, the data gathered from different operating systems was compared. Moreover, a performance analysis was performed by measuring the responsiveness of the application running in PANDA, data usage of the recordings, memory usage of the platform and its plugin as well as additional tests of the performance of the plugins.

The result showed that while the network-based intrusion detection system provides various results in the alerts produced, the virtual machine introspection could provide useful information for virtually all the attacks. The attacks that were fully detectable were; *add a file*, *crash a process* and *start a new process*. However, based on the Windows-based test cases, more attacks might be detectable by running a Windows guest system rather than running a Linux guest system. The performance analysis showed that utilizing the virtual machine introspection functionality made the responsiveness of the monitored system slower and that the memory usage was higher. As the execution needs to be recorded, the recordings require additional space. Additionally, in order to gather the data from the virtual machine introspection, the program runs slower and additional memory consumption is needed. This means that there will be a certain delay before the data can be accessed. Based on the result, it is suggested that virtual machine introspection only should be utilized during certain circumstances, such as when alarms are raised by the network-based intrusion detection system or when suspicious network packets arrive.

As virtual machines are increasingly employed in the industry, such as in cloud data centers, while the security threats in the society are expanding, the need for a more secure environment is growing. This thesis presents the first step towards such a secure environment based upon combining virtual machine introspection and

network-based intrusion detection systems.

Bibliography

[1] A. Kott, C. Wang, and R. F. Erbacher, eds., *Cyber Defense and Situational Awareness*, vol. 62 of *Advances in Information Security*. Springer, 2014.

[2] R. W. Ahmad, A. Gani, S. H. A. Hamid, M. Shiraz, A. Yousafzai, and F. Xia, "A survey on virtual machine migration and server consolidation frameworks for cloud data centers," *Journal of Network and Computer Applications*, vol. 52, pp. 11 – 25, 2015.

[3] "Amazon Web Services," 2014 (Accessed: 2016-05-25). `https://aws.amazon.com/`.

[4] "Google Cloud Platform," 2014 (Accessed: 2016-05-25). `https://cloud.google.com/`.

[5] "Microsoft Azure," 2014 (Accessed: 2016-05-25). `https://azure.microsoft.com/`.

[6] Cisco, "Network-based intrusion detection overview," tech. rep., Cisco, (Accessed: 2016-05-20).

[7] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *In Proc. Network and Distributed Systems Security Symposium*, pp. 191–206, 2003.

[8] M. Laureano, C. Maziero, and E. Jamhour, "Intrusion detection in virtual machine environments," in *Euromicro Conference, 2004. Proceedings. 30th*, pp. 520–525, Aug 2004.

[9] H. Debar, M. Dacier, and A. Wespi, "A revised taxonomy for intrusion-detection systems," *Annales Des Télécommunications*, vol. 55, no. 7, pp. 361–378, 2000.

[10] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung, "Intrusion detection system: A comprehensive review," *Journal of Network and Computer Applications*, vol. 36, p. 16–24, 2013.

[11] Y. Hebbal, S. Laniepce, and J. M. Menaud, "Virtual machine introspection: Techniques and applications," in *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, pp. 676–685, Aug 2015.

[12] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, (New York, NY, USA), pp. 128–138, ACM, 2007.

[13] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Secure and flexible monitoring of virtual machines," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pp. 385–397, Dec 2007.

[14] "LibVMI: Simplified Virtual Machine Introspection," 2016 (Accessed: 2016-05-25).
https://github.com/libvmi/libvmi.

[15] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Security and Privacy (SP), 2011 IEEE Symposium on*, pp. 297–312, May 2011.

[16] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering with PANDA," in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, PPREW-5, (New York, NY, USA), pp. 4:1–4:11, ACM, 2015.

[17] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan Zee (North) Bridge: Mining memory accesses for introspection," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, (New York, NY, USA), pp. 839–850, ACM, 2013.

[18] B. Dolan-Gavitt, B. Payne, and W. Lee, "Leveraging forensic tools for virtual machine introspection," tech. rep., Georgia Institute of Technology. School of Computer Science, 2011.

[19] PANDA, "PANDA User Manual | Pandalog," 2016 (Accessed: 2016-04-04).
https://github.com/moyix/panda/blob/master/docs/manual.md/.

[20] PANDA, "PANDA | panda_plugins," 2016 (Accessed: 2016-04-04).
https://github.com/moyix/panda/tree/master/qemu/panda_plugins/.

[21] PANDA, "PANDA," 2016 (Accessed: 2016-04-04).
https://github.com/moyix/panda/.

[22] "Libpcap File Format," (Accessed: 2016-07-16).
https://wiki.wireshark.org/Development/LibpcapFileFormat.

[23] M. Roesch, "Snort - lightweight intrusion detection for networks," in *Proceedings of the 13th USENIX conference on System administration*, (Seattle, Washington, USA), pp. 229–238, ACM, November 1999.

[24] Snort, " Snort FAQ | README.alert_order | ALERT ORDERING," 2016 (Accessed: 2016-04-04).
https://www.snort.org/faq/readme-alert_order/.

[25] Snort, " Output Modules," 2016 (Accessed: 2016-04-04).
http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node21.html/.

[26] Snort, " Payload Detection Rule Options," 2016 (Accessed: 2016-04-04).
http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node32.html/.

[27] "VMware News Releases," 2016 (Accessed: 2016-05-25).
https://www.vmware.com/company/news/releases/vmsafe_vmworld.

[28] B. D. Payne, "Simplifying virtual machine introspection using LibVMI," tech. rep., Sandia National Laboratories, 2012.

[29] PANDA, "PANDA Share," 2016 (Accessed: 2016-04-04).
http://www.rrshare.org/.

[30] "The Bro Network Security Monitor," 2014 (Accessed: 2016-05-25).
https://www.bro.org/.

[31] J. T. Rødfoss, "Comparison of Open Source Network Intrusion Detection Systems," Master's thesis, Oslo University College, 2011.

[32] "Suricata," (Accessed: 2016-05-25). https://suricata-ids.org.

[33] J. Mudigonda and P. Ranganathan, "Server switch integration in a virtualized system," Mar. 24 2015. US Patent 8,990,801.

[34] J. Foster and J. Deckard, *Buffer Overflow Attacks: Detect, Exploit, Prevent*. Elsevier Science, 2005.

[35] Debian, "Debian Squeeze and Wheezy i386 images for QEMU or KVM," 2016 (Accessed: 2016-04-04). https://people.debian.org/~aurel32/qemu/i386/.

[36] Jeffrey A. Turkstra, "Buffer Overflows and You | Exploit," 2016 (Accessed: 2016-04-04). https://turkeyland.net/projects/overflow/exploit.php/.

[37] "Writing Good Rules," 2014 (Accessed: 2016-05-25). http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node36.html.

[38] "TopSharedMemoryBug," 2014 (Accessed: 2016-06-05). http://wiki.apache.org/spamassassin/TopSharedMemoryBug.

[39] "Metasploit," (Accessed: 2016-07-16). https://www.metasploit.com.

[40] "ASCII shellcode," 2014 (Accessed: 2016-05-25). https://nets.ec/Ascii_shellcode.

[41] Peter Vreugdenhil, " HAPPY NEW YEAR ANALYSIS OF CVE-2012-4792 ," 2016 (Accessed: 2016-04-04). https://blog.exodusintel.com/2013/01/02/happy-new-year-analysis\-of-cve-2012-4792//.

[42] CVE, " CVE Details," 2016 (Accessed: 2016-04-04). https://www.cvedetails.com/.

[43] CVE, " Internet Explorer : Security Vulnerabilities (Memory Corruption)," 2016 (Accessed: 2016-04-04). http://www.cvedetails.com/vulnerability-list.php?vendor_id=26&product_id=9900&version_id=&page=1&hasexp=0&opdos=0&opec=0&opov=0&opcsrf=0&opgpriv=0&opsqli=0&opxss=0&opdirt=0&opmemc=1&ophttprs=0&opbyp=0&opfileinc=0&opginf=0&cvssscoremin=0&cvssscoremax=0&year=0&month=0&cweid=0&order=1&trc=526&sha=16cb9a3cb15717c5411f39789472de5b82ad8e6c/.

[44] Mark Yason, " Use-after-frees: That pointer may be pointing to something bad | Example 2: CVE-2012-4792 (IE CButton UAF) ," 2016 (Accessed: 2016-04-04). https://securityintelligence.com/use-after-frees-that-pointer\-may-be-pointing-to-something-bad//.

[45] OWASP, "Buffer Overflow," 2016, (Accessed: 2016-04-04). https://www.owasp.org/index.php/Buffer_Overflow/.

[46] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: Automatic adaptive detection

and prevention of buffer-overflow attacks," in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, (Berkeley, CA, USA), USENIX Association, 1998.

A

Vulnerabilities Used for Analysis

Two types of vulnerabilities are presented, which are utilized in this thesis in order to collect relevant data. Firstly, the vulnerability use-after-free is presented. Secondly, the vulnerability buffer overflow is presented in detail.

A.1 Use-after-free

A use-after-free vulnerability depends on a memory corruption flaw [41]. The reason is that the application allows both freeing memory space and accessing memory space, which can lead to unexpected behaviour if the application tries to access the freed memory space [41]. A freed memory space can be assigned to new pointers, while the application that used the freed memory space is still running and can point to the freed memory space again. By doing this the application will return to a completely other place in memory, which might lead to an attacker being able to perform remote code execution. Accessing freed memory might also lead to a crash of the application or the application executing arbitrary code [41]. Many such vulnerabilities exist in software. They are listed on the well-known Common Vulnerabilities and Exposures (CVE) web page of known information security vulnerabilities from the National Vulnerability Database [42]. Several such vulnerabilities have been discovered in previous versions of Internet Explorer, such as the vulnerability CVE-2012-4792 presented in Section A.1.1 [43].

A.1.1 The Internet Explorer Vulnerability CVE-2012-4792

This vulnerability was found in Microsoft Internet Explorer, version 6-8. This specific use-after-free vulnerability occurs due to a CDoc object containing an allocated CButton object that is freed by setting the ".outerText=". The CDoc object's pointer to CButton is not changed and its pointer becomes a dangling pointer after the CButton is deleted with garbage collection. When the CButton pointer is dereferenced, the vulnerability is exploited [44].

A.2 Buffer Overflow

Buffer overflow vulnerabilities have historically been common, and there still exist such software flaws today. A buffer overflow vulnerability is caused when a user

provides input to a program that overflows the allocated memory space, which results in memory addresses being overwritten. Each process has its own view of the memory and the process has direct access to only a portion of this memory. A part of this memory is the stack where all the user provided data are stored as well as the local variables, return addresses, and the frame pointers next to them. There exists different kind of buffer overflow attacks. This thesis refers to stack-based buffer overflow attacks when mentioning buffer overflow. A program, which is vulnerable to a stack-based buffer overflow attack, performs operations that lack crucial checks of the size of the user inputs [45]. Therefore, if the size of the input data provided by the user exceeds the size of the buffer and if the concerned functions do not validate the size of the data, the user will be able to overwrite the return address and frame pointers that leads to the interruption of the normal program execution and cause a buffer overflow attack [45].

A.2.1 Constructing Attacks against Buffer Overflow Vulnerabilities

Buffer overflow vulnerabilities can be exploited in different ways. An attack targeting a buffer overflow flaw can affect the availability of the vulnerable program. A vulnerable program will crash if the return address or frame pointers are overwritten arbitrarily [45]. However, buffer overflow vulnerabilities can also be exploited for gaining access control, which will allow the attacker to execute code. If a return address is crafted in a way such that it points to an address in memory where the attacker's code is placed, the buffer overflow vulnerability results in the program interruption and the attacker's code being executed [45]. The attacker's code is called shellcode. Due to the fact that it is hard to determine the offset to where the shellcode is placed, NOOP instructions are usually utilized. These NOOP instructions are named a NOOP sled. If the return address points to anywhere in the NOOP sled the shellcode will eventually be executed. The location of the return address is also needed, and it can be determined by, for example, reverse engineering [46].