Detection of debugger aware malware

Project Report

1013f15

Aalborg University
Department of Computer Science
Selma Lagerlöfs Vej 300
DK-9220 Aalborg Øst
http://www.cs.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerlöfs Vej 300
DK-9220 Aalborg Øst
http://www.cs.aau.dk

Title:
Detection of debugger aware malware

Theme:
Malware

Project Period:
Spring Semester 2015

Project Group:
des1013f15

Participant(s):
Sergey Gurkin

Supervisor(s):
Rene Rydhof Hansen
Matija Stevanovic

Copies: 4

Page Numbers: 41

Date of Completion:
June 4, 2015

Synopsis:

A debugger is a tool that is often employed by a security researcher to analyse malware. Sophisticated samples often employ different variety of anti-tampering techniques, which often include debugger detection. This can be detected by a researcher, but it is impossible to analyse every piece of malware due to high volume, and an automatic solution is needed.
Two approaches to the problem of automatic detection of anti-debugger malware were researched and discussed during this project - instruction tracing and differential analysis. Based on the research, a utility tool and a Cuckoo Sandbox module were implemented as a solution.

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.

Abstract

Computer security researchers are often placed in a difficult situation - their tools need to be perfect, in terms of possible exploits, in order to not raise suspicions in analysed malware. It is in the interest of malicious code to shut down if an analyst's presence is detected as the longer it runs, the more revenue can be generated. While malware only needs to find one weakness in the analysis tool, the tool needs to cover all possible angles of attack, which is often impractical. It is not possible to analyse every discoverd malware sample by hand, because of large numbers, and automation is required. Malicious code that employs anti-tampering techniques can often mimic as a non-threatening application to slip past the automatic analyser. Such samples require more fine-grained approach, which is also more costly. A system that can detect these samples can improve the overall performance of the malware analysis process.

Differential analysis converts the weakness of an analyst's tool into a strength and can yield better performance than analysing every sample in a fine-grained manner. This runs a malware sample in two machines and records it's actions. One of the machines is clean while the other contains traces of an analysis tool, such as debugger. Malware is declared as evasive if a difference is spotted in it's behavior. This workflow was implemented in an automatic dynamic malware analysis environment - Cuckoo Sandbox.

Preface

The following report is a product of SW10 specialization project for the fifth-year student of Software Engineering at Aalborg University. This project is the continuation of work performed during SW9th pre-specialization semester. Sources can be found in the last part of the report.

Sergey Gurkin
sgurki08@student.aau.dk

Contents

List of Listings 9

1 Introduction 11
1.1 Initiating Question 12

2 Malware analysis 13
2.1 Debuggers 13
2.2 Virtualization 16
2.3 Differential analysis 19
2.4 Cuckoo sandbox 21
2.5 Discussion 23

3 Related work 25
3.1 Debugging malware 25
3.2 Differential analysis 29

4 Solution 33
4.1 Obtaining samples 33
4.2 Differential analysis 35

5 Conclusion 41

Chapter 1

Introduction

According to the experiment performed in [6], 40% of malware samples exhibit less malicious behavior when executed with an attached debugger. This is a serious problem because debuggers are often employed by security researchers to understand the behavior of malware so that counter-measures can be developed.

Malware and security solutions are in a state of continuous arms race. Every new exploit is often soon countered with a patch which is then countered by a newer exploit. There exist several ways in which malware can be analysed, but these are often separated into static and dynamic techniques. Static approach analyses malware without executing it, e.g. disassembling the binary files. This is countered by different obfuscation techniques, that make static analysis and extraction of assembly code very difficult [26].

Dynamic analysis techiques counter this by letting the malware run while observing its behavior that can be expressed in a form of e.g. manual debugging, Application Programming Interface (API)-calls trace, a memory or network dump analysis, changes to registry and file structure. Malware responds to this by detecting the tools and techniques that are used to capture this data and appearing harmless if that is the case. Such samples require a more thorough but also a more costly analysis technique. It is not possible to manually analyse all the malware samples because of sheer volume and much of this task is automated. Automatically detecting whether an executable is truly harmless or only appears to be so can be difficult. Automation can be done by employing sandbox analysis systems such as described in [9].

The presented work is a continuation of [15], a project which evaluted popularity of Virtual Machine (VM) detection techniques. This was done by identifying popular methods and querying a database containing analysis reports of 80000 malware samples obtained in the wild. The reports were generated using Cuckoo Sandbox (Cuckoo)[13], an open source customizable automatic dynamic malware analysis system that is still being actively developed. It was discovered that some approaches, e.g. [39] which do not use APIs but inline as-

sembly techniques, could not be viewed in the Cuckoo reports as the required data was lacking. Approaches that use inline assembly techniques to detect debuggers also exist [10]. This problem is not local to Cuckoo and many sandboxes suffer from same issues, but Cuckoo was developed from the start to be expandable through some customizable modules which could be a source of solution.

1.1 Initiating Question

How to improve Cuckoo to detect malware samples that use advanced anti-debugging methods that rely on techniques such as inline assembly checks?

Chapter 2

Malware analysis

Malware that uses inline assembly to test for debuggers can be detected in several ways. In an ideal world, researchers would have access to the malware source code but this is not the case and only raw machine code binary files are available. One possible solution is to obtain the assembly code of the sample and look for specific patterns, signatures. This can be done statically by applying tools such as disassemblers. A disassembler generates assembly code from binary files which contain raw machine code. This can fail if the code is "armored" - malware creator used e.g. obfuscation techniques or packer software. A dynamic approach would let the malware run and record all the assembly level instructions as they are executed by hardware - an instruction trace. An instruction trace can be obtained by using some debugging functionality encoded directly in hardware.

An alternative approach that does not rely on instruction tracing is differential analysis. This is a dynamic approach that requires execution of the sample in different environments - with and without a debugger. Behavior across both environments is then compared and a sample is declared as evasive if deviations are found.

2.1 Debuggers

A debugger is a piece of software that utilizes Central Processing Unit (CPU) facilities that were specifically designed for the purpose. A debugger provides an insight into how a program performs its tasks, allows the user to control the execution, and provides access to the debugged program's environment. It is possible, using a debugger, to manually step over every instruction executed by the analysed binary and observe the changes done to memory and registers. An instruction trace can be generated by automating this process. This project focuses on Windows Operating Systems (OSs) using Intel CPUs.

Windows OS supports debuggers in the form of specific events, APIs and structures [23]. A debugger either starts a new process or attaches to a running

one and then enters an event handling loop. To manually step over every instruction, a specific CPU functionality is used - single-stepping mode. A CPU generates exceptions and waits for user response after execution of every instruction when it runs in single-stepping mode. These exceptions are delivered to the debugging software by the OS in form of events. Every exception has an EXCEPTION_RECORD structure associated with it. This structure is used to describe events to the handlers. Exceptions generated by a CPU running in single-step mode are described by EXCEPTION_SINGLE_STEP value of ExceptionCode field of EXCEPTION_RECORD structure. The GetThreadContext function returns the CONTEXT structure describing the state of the CPU at the moment of the exception.

A solution that either uses an existing debugger or Windows API for the purpose of instruction tracing is going to face the issue of privileges and detection. Windows OS separates running code in several rings or privilege levels. Code running in ring 0 is closest to hardware, has the highest privilege and is called kernel mode code while code running at ring 3 has less privileges and is called user mode code. The concept of privilege rings is illustrated in Figure 2.1. A program for Windows OS runs in either of two modes - user or kernel. Code that runs at the higher privilege has more control and can hide better. If an analyser component, such as a debugger, runs at the same privilege level as malware, there is a possibility of information leakage that can be used by malware to detect the debugger. The first requirement of transparency, according to [8], states that an analyser component (a debugger) must run above the highest privilege level malware can attain. This requirement is satisfied if debugger runs in kernel mode, while malware runs in user mode. Malware that runs in kernel mode, such as ZeroAccess rootkit [37], does exist and it can potentially discover an analyser that runs in kernel mode as well.

Debugging kernel code is a difficult task even when code under analysis is not malicious. It requires two machines - one to run the debugger while the kernel of the other is halted. It is possible to perform kernel debugging locally, but even then, it is a very delicate procedure and some of the usual debugger functionality, such as breakpoint and trace, is not available [24]. Using raw debugging functionality of existing solutions or Windows API with no additional enhancements is not optimal as it can be detected by malware running in kernel mode and is difficult to perform. A CPU's debugging support can be used with other solutions that modify the privilege levels structure.

2.1.1 Hardware debugging support

A single-core CPU executes processes one at a time but the execution can be diverted to another place in memory if an exception or an interrupt occurs. A process that generates an interrupt or an exception is suspended and the corresponding handler is executed. Every exception is identified by a unique number which is used as a lookup in Interrupt Descriptor Table (IDT) that links excep-

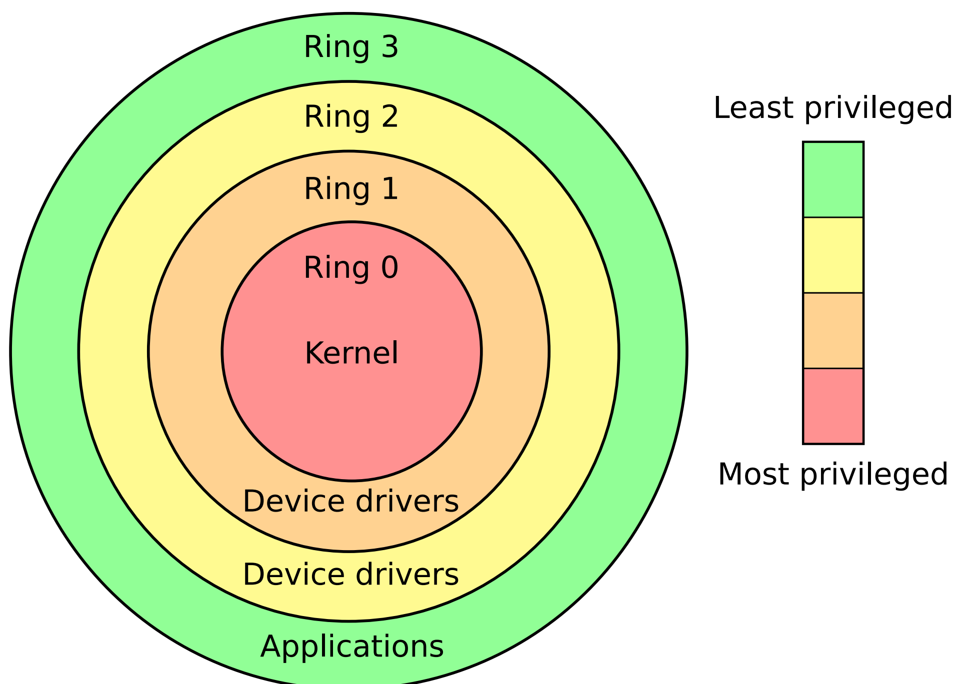

Figure 2.1: OS privilege levels[2]

tions and their handlers together. The suspended process is resumed after the handler is executed. The interrupt and exception handling mechanism plays a major part in how a debugger controls a program [19, Ch. 6].

Interrupt 1 - debug exception and Interrupt 3 - breakpoint exception are directly related to debugging and are supported by the Intel processors. The former exception can be raised as a result of several conditions, active single-stepping mode being one of them. Debug Status Register (DR6) is used to store the conditions that were sampled at the time the exception was generated, one of the conditions being the BS bit which tells whether the processor is in the single-stepping mode. This makes it possible to distinguish between the causes of the debug exception. When a processor runs in single-stepping mode, a debug exception is generated after execution of every instruction. The Trap flag (TF) of the EFLAGS register (EFLAGS) must be set in order to enter this mode. When a debug exception is raised, the content of the Extended Instruction Pointer (EIP) register is saved. The EIP register contains the address of the next instruction to be executed. The information stored in the EIP and DR6 makes it possible to produce a complete instruction trace [19, Ch. 17].

2.2 Virtualization

Using physical machines for the purpose of malware analysis is possible, but it involves some risks and VMs are often used instead [36, Ch. 2]. VMs simplify containment and the process of restoring the machine to a clean state after the analysis. It is possible to use virtualization to move an analyser component, such as debugger, out of malware's reach by either introducing new privilege levels or de-privileging the OSs.

A VM is an environment created by Virtual Machine Monitor (VMM), also known as a hypervisor. The main purpose of a VMM is to keep the VMs isolated from each other so that every single virtualized OS remains under illusion that it has full control of underlying hardware. The x86 architecture was not designed to handle several OSs running on a single processor and the VMM must emulate results of OS kernel mode code that would otherwise conflict. This was previously done by a technique called trap-and-emulate [28]. A VMM would run in ring 0 while de-privileging its VMs to ring 3. Every time a VM tried to execute its kernel mode code, the underlying hardware would generate an exception because virtualized OS would lack privileges as it was running in user mode at ring 3. This exception would then be handled by the VMM which runs in ring 0 and is authorized to do so. It would emulate results of exception-causing instruction and return it to the VM to maintain the illusion. This concept is illustrated by system 2 in Figure 2.2.

It is not possible to virtualize the x86 architecture using the classical trap-and-emulate technique because there exists a number of instructions executable from ring 3 that can expose the underlying VMM [34]. A number of approaches that successfuly achieve this task do exist and are described in [15]. Every one of these techniques - paravirtualization, binary translation and Hardware-assisted virtualization (HVM), either modify the privilige of the guest (virtualized) machine in relation to host (the machine on which the VM runs) or introduce new privilege levels to hardware, as it is the case with HVM. Figure 2.2 illustrates how different privilege levels are used in different x86 virtualization techniques. System 1 represents a normal machine with a single OS, system 2 is a classical trap-and-emulate machine, and system 3 is a machine virtualized with HVM. Even though classical trap-and-emulate is not possible on x86, both paravirtualization and binary translation use the same concept of de-privileging. Paravirtualization modifies the kernel code of the guest to not trap while binary translation involves translating the instruction that would trap into a non-trapping sequences before the actual code is run on the hardware. System 3 uses HVM that introduces a new privilege level in which the VMM runs - ring -1. The new privilege level allows the VMM to maintain control over VMs at all times and it also enables guest software, including OS, to run in privilege levels it was designed for.

An analyser component residing at ring -1, would satisfy the transparency requirement of [8]. A solution that either relies on paravirtualization or binary

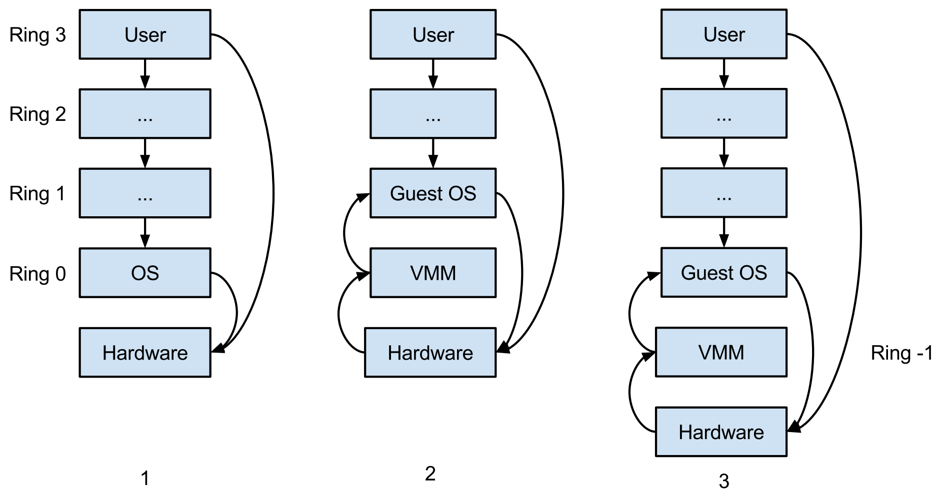

Figure 2.2: De-privileging in virtualization

translation is not optimal because it would lack the new powerful hardware features introduced by HVM. Paravirtualization also requires modification of OS code. Using a VM provides malware with additional detection possibilities compared to if a bare-metal system was used. However, it would be the VM that is detected, not the analyser.

2.2.1 Hardware assisted virtualization

The x86 architecture is made virtualizable by introducing two processor operation modes - VMX root and non-root, which are meant respectively for VMM and guest. New instructions are available to processors running in VMX root mode, compared to normal ring 0 operation mode. Figure 2.3 illustrates a VMM that runs two guests machines. The processor enters the VMX root mode with VMXON instruction and allows the guests to run with VM-enter. When a guest finishes execution or an event that stops a guest occurs, control is transfered back to the VMM by a VM-exit instruction.

A special data structure, Virtual Machine Control Structure (VMCS), is associated with every guest. This structure is used to control VM entries and exits as well as define processor behavior when it runs in VMX root or non-root mode. The data stored in the VMCS is divided into following groups:

- Guest-state area
- Host-state area
- VM-execution control fields
- VM-exit control fields

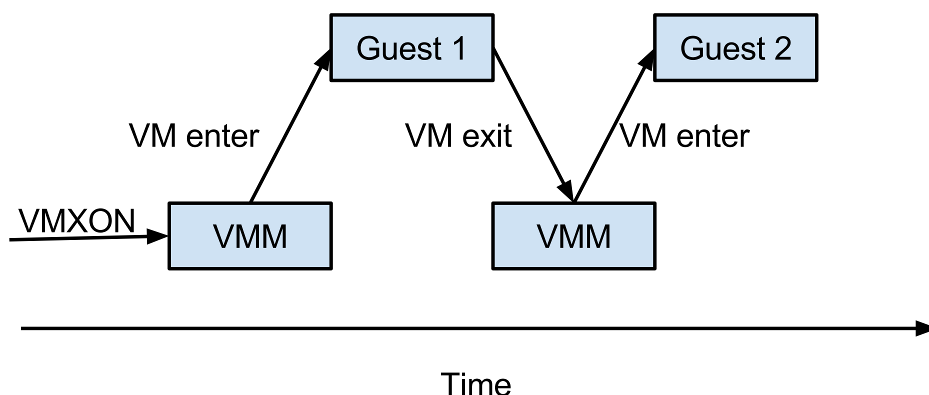

Figure 2.3: VMM running two guests with HVM

- VM-exit information fields

One way of procuring an instruction trace is by leveraging the processor debugging facilities described in Section 2.1.1 as well as VMCS. The guest-state area of the VMCS stores both register and non-register state of the guest VM. Register state has the registers of the CPU used by guest and non-register state has information such as whether the CPU is active, halted or can be interrupted e.g. VMM can set the TF of EFLAGS and call VM-enter which would result in the guest machine running in single-step mode. In order to cause VM-exits on Interrupt 1 - debug exceptions, the VMM also has to modify the exception bitmap, located in VM-execution control fields of the VMCS. The exception bitmap is a 32 bit field, each bit corresponding to one of 32 pre-defined Intel interrupts. If a bit is set to 0, an interrupt is delivered normally through guest IDT, but if the bit is set to 1, the interrupt causes VM-exit and return of control to the VMM. It is then possible to examine VM-exit information fields to determine the cause of VM-exit, which in the case of single-stepping would be a debug exception, and to examine the guest register state saved in VMCS to extract the instruction information.

An instruction trace can also be produced by using one of new functionalities of HVM - Monitor Trap Flag (MTF). This mode can be activated by setting the 28th bit in the VM-execution control fields. MTF causes VM-exits after execution of every instruction.

The two methods differ in visibility from the guest system and the verbosity of the results. A TF explicitly states that CPU is running in a single-step mode. A debug exception caused by single-stepping is not handled immediately. The state of the program is changed before the exception is raised, meaning that an instruction has to be executed in order for the single-step debug exception to be generated. In the context of a VM-entry, this means that the single-step debugging exception does not happen immediately and another exception, regardless of its priority, can be pending. If exception bitmap for this pending exception is

0, it will be handled by guest IDT, meaning that no VM-exits will be generated. Contents of CPU registers, EFLAGS amongst others, is saved to the stack right before the exception or interrupt handler code is run. The TF is also set to 0, so as to not trap on every instruction of handler code. If malware gains access to the running handler, it can see the state of TF on the stack and determine that it is being traced and it would not be possible to see as the handler code is not traced if TF is used.

The MTF approach is transparent to the guest as it is controlled through the VMCS. It causes VM-exit after execution of every instruction, unless an event of higher priority, such as system-management interrupt, occurs before CPU had a chance to execute an instruction. This means that interrupt handler code with lower priority will be traced.

Both of the described approaches to produce an instruction trace can be automated, but they suffer from performance issues. This happens because every instruction executed by guest means a VM-exit which involves:

- Record the cause of VM-exit
- Record the state of guest processor
- Load the state of host processor

The cost of VM-entry is similar as host state has to be saved and guest state has to be loaded. Optimizations are possible as demonstrated by [1] who note that performance of VM-entries between P4 and “Core” Intel architectures improved from 2409 to 937 cycles.

2.3 Differential analysis

This is an alternative approach to detect debugger aware malware. At least two environments are created - reference and comparison. Reference environment is used as a baseline while comparison environment is “tainted” by the presence of a debugger. A malware sample is executed in both environments and behavioral logs are generated for each. Behavior of the sample is compared across the environments and it is declared evasive if the difference is significant.

Reference machine needs to be convincing enough for malware to show its true malicious behavior. One option is to use bare-metal machines both as reference and as comparison. This approach is more transparent than if VMs were used but it is also more costly as restoring bare-metal machines to a clean state is more difficult than doing so with a VM, as snapshotting is a core feature that is readily available out of the box for many VM products. Using VMs introduces another problem - malware that detects virtualization and shuts down. This makes it difficult to decide whether a sample behaves equally in both reference and comparison environments because anti-debugger techniques are abscent or because it detects virtualization before anti-debugger checks are performed

and stops executing. It is possible to hide some signs of virtualization. The VM-Cloak project [5], for example, does this by harvesting real world hardware information from registry and applying it to the generated VMs. While it is possible to maintain transparency against some approaches, methods like local [31] or remote timing analysis [21] are difficult to defend against and achieving complete VM transparency is not yet possible [11].

Whether a bare-metal or VM approach is chosen, the machine itself needs to be convicing. Signs of a real world user activity, such as presence of cookies, popular applications, documents and some modifications to default settings can improve the illusion of a real machine. Making the illusion completely transparent is difficult as at least one component of the analysis system - the initiator that starts the malware, must be present. The signs of this component can be hidden by automatically uninstalling it after every run, such as the case with [20].

One telltale sign of an analysis environment is the absence of internet access. Internet is used by malware for propagation and as a Command and Control (C&C) channel - which is the case with botnets. A bot would not be able to function properly without internet to receive commands from the master. The propagation technique is also difficult to observe without at least a local network. Giving malware full network access is dangerous and unethical but several solutions exist. One way to deal with this problem is to only grant partial network access to malware so that messages which are perceived as malicious are blocked. For example, a single Domain Name Service (DNS) lookup and a connection to the resolved Internet Protocol (IP)-address is probably an attempt to connect to a C&C server and should be allowed. Repeated attempts to establish and drop a Transmission Control Protocol (TCP) connection is a sign of an ongoing Distributed Denial of Service (DDoS) and should be blocked. This approach is used by Honeywall, a 2nd generation honeypot [29] and GQ [22]. Automating policy generation is difficult and most work has to be done by hand. Another approach to the network problem is to try and fool the malware by emulating populare network services such as Hypertext Transfer Protocol (HTTP), DNS and Internet Relay Chat (IRC). Tools like fakeNet [35] and INetSim [17] try to perform intelligent emulation of these services by satisfying the malware's requests. For example, if malware requests a .jpg file from an emulated HTTP service, the response would include a stock image or one located at user-configured path. It is possible to defeat such approaches and sophisticated malware will not be fooled. For example, a botherder can generate a list of email addresses, including some of his own to test if everything is working properly, to send spam to.

An important part of the environment for this project is the debugger itself. Popular methods of debugger detection are known and while it is possible to emulate the signs of debugger presence, like setting the BeingDebugged flag of a process to true, it is also cumbersome as debuggers can be detected in numerous ways [10]. Another disadvantage of such approach is that previ-

ously unknown methods will not be detected. This can be solved by using actual debugging software. Merely installing it is often not enough and a debugger must take an active role by running the malware samples. It is possible to automate this process to a degree as some debuggers, such as GNU Project debugger (GDB), have a Command Line Interface (CLI) that allows to simply run a selected sample. Making the debugger perform more advanced actions, such as setting breakpoints or passing exceptions to the sample, is difficult. This would require a standalone emulation component using a dedicated debugger's API or using some debugging functions provided by the OS, such as Windows debugging API [23].

It is also necessary to gather enough information so as to generate an accurate behavioral profile for later comparison. Presence of analyser components in the machine reduce authenticity but increase information yield. The most stealthy approach to generate a malware's behavioral profile is to base it only on persisten system changes as this does not require an in-guest analyser component, reducing risk of detection. A change is persistent if it is still present after a system shutdown, e.g. changes to Windows registry or filesystem, such as created, edited or deleted files. It is not possible to detect memory-only malware using this approach. Network activity can also help in generating a malware behavioral profile. If the traffic is captured from outside the analysis environment, e.g. at the gateway, filtering is necessary to generate traffic produced specifically by the sample. It is possible to obtain more information from a malware run by using in-guest components. For example, a list of all API-calls and parameters performed by malware can be obtained by hooking the APIs.

The comparison of malware behavioral data can take several forms. A cheap solution is to compare the raw data in a bruteforce manner - each API call from one environment is directly compared to an API call from a second environment e.g. The sample is labeled as evasive if there is a slightest mismatch. This is a very rigid approach and a slightest difference, that might have been completely unrelated, can set it off. For example, a fast-flux network [32] could have been used by malware author. In this case, the IP-address associated with the hostname can change very often, meaning that if malware analysis was not performed simultaneously at both reference and comparison machines, the extracted network logs would show different IP-addresses. This difference is superficial and does not have any semantic meaning, but it would be detected by the bruteforce comparison nonetheless. This can be solved by abstracting from the raw behavioral data, as suggested by [4].

2.4 Cuckoo sandbox

Cuckoo [13] is an open source customizable automated malware analysis system. It dynamically analyses a sample by running it in a controlled VM. A Cuckoo analysis yeilds reports such as API-calls trace, network and memory

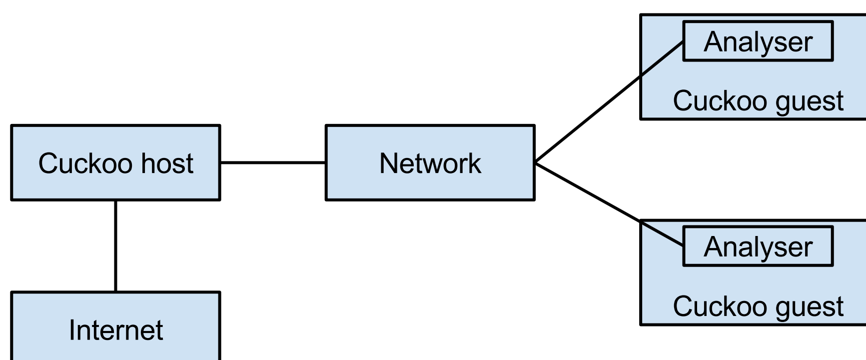

Figure 2.4: Overview of Cuckoo architecture

dumps, changed files and extracted ASCII strings. It is possible to extend the stock functionality with user-written plugins.

Figure 2.4 illustrates a general overview of Cuckoo architecture. The system is separated in two parts - core (host) and guest. Communication happens through a network, which can be virtual if both components run on the same physical machine. The host controls the analysis guest machines, manages submissions and processes the results submitted by the guests. Every guest environment has an analyser component which is responsible for the malware samples it receives from the host. The analyser injects the received samples with a .dll that contains inline function hooks of normal and native Windows APIs, executes malware according to its type and sends the gathered data back to the host. The data is sent over a TCP connection as events happen and not at the end of analysis, making the analysis process more resilient to crashes.

Functionality of both core and analyser components of Cuckoo can be extended through user-written modules. The following list of modules describes the extensions possible to the core component:

- Machinery - layer between Cuckoo and VMM of the guest machine
- Auxiliary - run on host machine concurrently to each analysis, e.g. capturing network traffic
- Processing - process the raw results received from Analyser, e.g. memory dump and API-calls analysis
- Signatures - capture events specified by the signature at the end of analysis, e.g. to identify a certain type of malware based on characteristics of its behavior
- Reporting - translate results of an analysis into different formats, e.g. Javascript Object Notation (JSON), HyperText Markup Language (HTML)

or MongoDB.

Machinery modules are a way for Cuckoo to connect to the VMM of choice in order to leverage its functionality for the purpose of analysis VMs management, e.g. to reboot the VMs and restore them to a clean snapshot. Machinery modules allow Cuckoo to connect to desired VMM directly or by using LibVirt [16] library. LibVirt is an open source API the goal of which is to provide common and stable layer for accessing different VMMs through a single library. In order to generate an instruction trace of a malware sample, the code of VMM itself needs to support this functionality. If the VMM support is present, a machinery module can be used in order to start the tracing and an auxiliary module can receive the reports generated by the VMM through the network. A processing module for analysing the high volume of assembly instructions would also be required.

Extending the analyser component through the following user modules is also possible:

- Analysis packages - define how to run malware in guest, e.g. how to open a .pdf or .exe files.
- Auxiliary - run concurrently with analysis in guest, e.g. emulating human presence by moving mouse and clicking on “Accept” or “OK” windows

An analysis package component is responsible for starting the actual malware process. It is possible to modify the process environment to make it look like it is being debugged or to attach an actual debugger to the process. A differential analysis workflow is made of executing a sample in different environments, comparing results and producing a report. A Cuckoo workflow works only with a sample running in one environment per analysis. Automating the process of differential analysis only with Cuckoo modules might be difficult because of the difference in workflows. The customizable modules allow some change in how each individual analysis step is performed, but it is not possible to add new steps or completely redefine old ones. It is possible to add some comparison and automation functionality to a core reporting module, as processed malware execution results are first available at reporting stage and also because every reporting module is run by Cuckoo every analysis. A standalone component that uses Cuckoo only to produce malware analysis reports and then performs the comparison and reporting tasks on its own is also a possible solution.

2.5 Discussion

Instruction tracing, a feature present in many debuggers, produces a trace of assembly instructions performed by a sample. It is possible to automate the task of obtaining instruction traces by leveraging a CPU’s support for debugging and the new hardware features introduced by HVM. An instruction trace can be

useful in different cases of malware analysis, but it does not in itself tell an analyst whether a sample employs anti-debugging techniques or not and further analysis is required. This approach produces a lot of data and is also computationally expensive as each instruction executed by malware results in several context switches in the CPU with VM-exits and VM-enters. It requires support from a VMM which makes the complete solution possibly dependent on one VM vendor. An instruction trace also only covers a branch of execution. If malware stops its execution before it tests for debuggers, e.g. it spots virtualization, anti-debugging behavior would not be visible.

Differential analysis relies on comparison of execution of a malware sample in several environments. This approach does not require support from low-level components, such as VMM. Detection of samples that perform 0-day exploits is also possible for further, possibly manual, analysis. Whereas instruction tracing can be of benefit for any malware researcher, the results of differential analysis are only of interest for researchers interested in anti-debugging techniques. Both approaches require an environment that appears as trustworthy to malware as possible. This raises several ethical and technological difficulties, mainly associated with internet access and VM transparency.

A solution using the differential analysis approach is the focus of this project. While it also requires more computation than a normal analysis, it can spot malware that tries any of detection techniques, including the 0-day exploits. It can be viewed as a filtering step that separates trivial malware from armored samples. These anti-tampering samples can then be sent for a more costly but also more informative analysis.

Chapter 3

Related work

3.1 Debugging malware

Debuggers are useful tools that allow analysis of code at low level. One of the most important functionality of a debugger is breakpoint. When a breakpoint is hit, execution of program is stopped and control is given to the debugger, allowing analysis of the environment at the time. This could be very helpful when analysing malware, as it would be possible to see how it tries to detect tampering and to skip the garbage instructions inserted on purpose. Breakpoints can be of two types - software and hardware. Hardware breakpoints use CPU's debug registers to store a memory address value, which can for example be of code or data type, on which to trigger. The disadvantage of hardware breakpoints is the low number of debug registers - 4. Software breakpoints can be implemented in two ways - either by inserting debugging instructions at compile-time or by modifying binary at run-time. The former is often not possible when analysing malware, as this requires access to source code. The latter involves saving the instruction at the desired address and overwriting it with a special instruction that would generate an exception if executed - int3 on x86 architecture. When this happens, the control is given to the debugger program and the original instruction is restored to be executed. Number of software breakpoints is not limited as their hardware counterpart. Both types of breakpoints can be detected by malware. Hardware breakpoints can be counteracted by malware that fills the debug registers with some of its own values e.g. Sofware breakpoints can be detected by checksum approaches, as they modify program's code.

VAMPiRE is one of the subsystems of dynamic malware analysis framework developed by [38]. The main purpose of this subsystem is to implement "stealth breakpoints" - unlimited breakpoint functionality that is not detectable by common malware approaches. This is done by leveraging virtual memory system abstraction of modern computers. One of the purposes of an OS is to keep running processes separated from each other. This involves managing their memory, so that processes do not write over other processes memory. This is managed by a

virtual memory system - a mapping between a virtual and physicall addresses. A process sees the virtual address space as continuous piece of memory while OS uses special hardware such as Memory Management Unit (MMU) to translate virtual addresses to real ones, which can be fragmented. The smallest unit on which virtual memory system can operate is a page, which has several attributes available to it. One of the attributes is present or non-present which means whether a page is actually present in the physical memory or if it was swapped and is stored on disk. A page fault is generated if it is the latter and control is handled to the page fault exception handler.

Manipulation of page attributes to cause invocation of a page fault handler is the main mechanism behind breakpoints implemented by the VAMPiRE system. A user specifies the memory location on which to set the breakpoint. This location is added to a breakpoint table and the page associated with the address is set to be non-present. The default page fault handler redirects execution to the code implemented by VAMPiRE. This is done in binary at run-time so as to cover it up from malware. If the address of the page that generates a page fault exception and any of the addresses stored in the breakpoint table match, a breakpoint is reached and the target instruction is executed in single-step mode.

VAMPiRE modifies the page table and can be detected by malware that attains kernel level privilege. It must also run its handler on every page fault, which can incur a performance overhead. The transparency requirements of Ether [8] are not satisfied either, as the system runs in guest.

A similar approach, but using HVM, is presented in SPIDER [7]. The main concept of SPIDER is illustrated in Figure 3.1. A guest physical page is translated into two host physical pages - code and data. The host physical pages have mutually exclusive attributes. The host page 1, the code page, is executable but not readable. The host page 2, the data page, is readable but not executable. None of the pages are writeable. These page attributes ensure that if guest tries to e.g. read the code, a page fault exception will be generated. In contrast to VAMPiRE where page fault handler code was placed in guest, the SPIDER's page fault handler is placed in the hypervisor, transparent to malware. The SPIDER hypervisor will then return appropriate page, hiding the int3 instruction which is used to generate software breakpoints.

Transparent breakpoints implemented by VAMPiRE or SPIDER are useful in manual analysis. Automating the analysis is difficult, because the anti-instrumentation code needs to be detected first. Another approach is to generate an instruction trace of malware and analyse the results afterwards. An instruction trace is a log of commands executed by a CPU. It can be a useful resource in malware analysis as it provides the lowest abstraction level expression of a program behavior and it is more complicated for malware to hide its behavior. Instruction tracing is a fine-grained analysis technique that also comes at the cost of performance. Every executed instruction results in several actions that handle e.g. collecting the required log information and allowing for execution to continue.

Ether [8] is a dynamic malware analysis system capabale of producing an in-

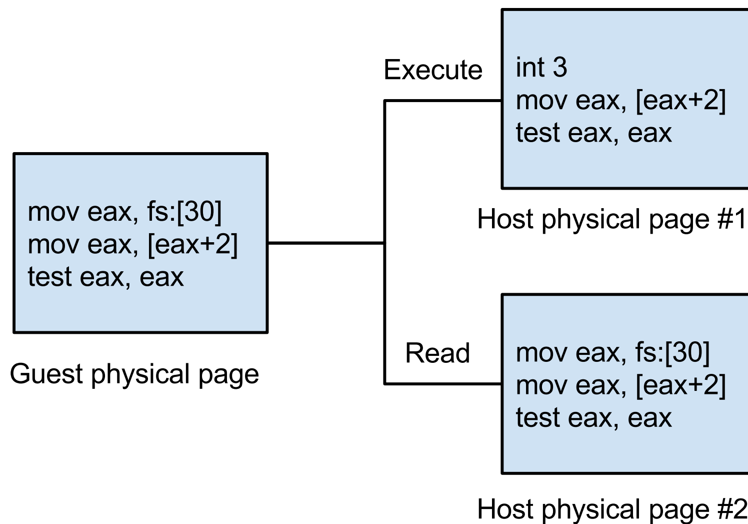

Figure 3.1: Main concept of SPIDER [7]

struction trace for research purposes. It is built upon Intel Virtualization Technology (Intel VT) [18] and Xen hypervisor [40]. Intel VT is a HVM that allows for virtualization of x86 architecture using trap-and-emulate technique described in [28]. This is achieved by creating a new privilege level for VMM to run at and by implementing explicit customizable trapping controls at hardware level. A more in-depth description can be found in Section 2.2. Ether is built upon Xen version 3.0 which added support for HVM.

Figure 3.2 provides an overview of Xen and Ether architectures. Xen's terminology is different from other VM vendors and guest systems are called unprivileged domains - DomUs. DomUs do not have privileges to access hardware directly and are completely isolated from each other and Dom0. Dom0 is the control domain, a host system the main purpose of which is to expose Xen VMM control interface. The Xen's architecture is expanded by Ether - two new components are added. Events such as instruction tracing, system calls and memory writes are detected by the Ether VMM component. The handlers for these events are located in Ether userspace component.

Instruction tracing is implemented by setting the TF of EFLAGS register in the VMCS of the guest machine. When TF is set, interrupt 1 - debug exception is generated by the CPU after execution of every instruction. This exception is forwarded to the Ether VMM component which also has the task of filtering instructions that are not generated by the malware process itself. This is done by checking the value of Control Register (CR)3 which contains the physical adress of page directory. Page directory is the base of the paging structure hierarchy and its value is unique for every process [19, Sec. 2.5]. If the CR3 value matches the one observed during execution of malware process, the event information is

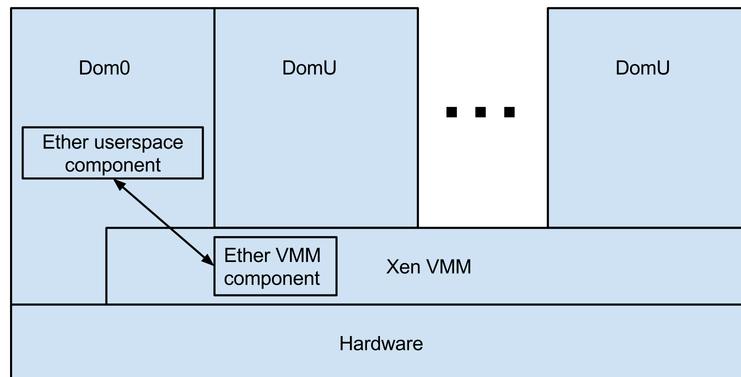

Figure 3.2: Ether and Xen architecture [8]

forwarded to the Ether userspace component which collects and logs the data. It has to translate the opcode of the instruction into a readable format, e.g. 88 to MOV, and collect the information stored at registers at the time of the event. When this is completed, the Ether component allows the execution of DomU analysis machine to continue until next debug exception.

Memory writes and system calls are monitored in a similar way. Ether VMM creates shadow page tables that guests use for address translation. The shadow page entries lack write permissions and an exception is generated every time a guest tries to write. System calls monitoring is a more coarse-grained analysis than monitoring memory writes and tracing instructions and is less expensive. It is done by overwriting memory pointed to by SYSENTER_EIP_MSR which is a model specific register holding a memory address for SYSENTER command. SYSENTER is used from user level to request a system call, it changes privilege level to kernel and jumps to the specified kernel memory address which performs the required system call. This value is overwritten to generate a page fault which is then handled by Ether.

One of the main Ether features is the lack of any in-guest analyser component. Monitoring happens outside of guest environment - a technique also known as VM introspection. Ether presents 5 transparency requirements that it attempts to satisfy:

- Higher privilege - it should not be possible for malware to achieve privilege level of the analyser component. This requirement is satisfied because VMM implemented with HVM support is located at privilege level -1 and this is enforced by hardware.

- No non-privileged side effects - the side effects possibly induced by the analyser component should only be accessible through privileged handlers. This requirement is satisfied because analysers memory is isolated by shadow page tables mechanism. Sensitive CPU instructions also generate exceptions that are only handled by the VMM.

- Identical basic instruction execution semantics - the instructions executed by guest should have the exact same semantics as if guest machine was not virtualized. This requirement is satisfied because guest OS with HVM runs in privilege level 0, just as a normal non-virtualized machine would. The exceptions generated by the CPU are transparent to the guest because they are handled by the VMM at privilege level -1.

- Transparent exception handling - the guest exception handling should not be affected in any way - no existing handlers should be modified and no new handlers should be added by the analyser component. This requirement is satisfied because exception handling happens at VMM which has access to the VMCS structure. This makes it possible for VMM to distinguish between "true" exceptions generated by guest, which are delivered, and exceptions that are specific to the Ether, which are handled transparently.

- Identical measurements of time - timing information available to guest should be the same with and without analyser component. Ether analyser componet incurs timing penalties as it needs time to run its code. This requirement can be satisfied for local timing sources as these instructions are privileged and generate exceptions when executed from guest. The VMM would have to to calculate the performance hit incurred by exception handling and modify the timing information available to guest accordingly. It is not possible to modify remote or external timing sources and this can lead to detection.

Authors note that Ether could be detected because of side effects incurred by the first generation Intel Intel VT that have to do with memory and Transaction lookaside buffer (TLB), which is flushed on every VM-exit. This issue has been fixed in later HVM implementations [12]. Instruction tracing in Ether is also limited by poor performance and authors have stated that it not meant as a real-time analysis tool.

VAMPiRE, SPIDER and Ether are all vulnerable to timing analysis, one of detection methods that is most difficult to defend against. An analysis component, located inside the guest or in hypervisor, needs time to run if it is to be of any use. This incurs a timing penalty which can be detected by malware. The tools presented here hide their presence by modifying the local time stamp counter register to appear as if they never ran. While this makes it possible to avoid malware that only relies on local timing sources, it is not possible to avoid malware that uses remote timing.

3.2 Differential analysis

A different way to detect anti-debugging capabilities, without suffering from large performance overhead induced by fine-granularity analysis, is by compar-

ing the behavior of malware in a clean reference and in an instrumented analysis environments. Any deviations in behavior imply that the modified analysis environment is detected. This method does not show exactly how a debugger is detected. The exploit can be deduced from analysing the malware behavioral log that was used for comparison, but if the granularity of information in the log is not fine enough, the exploit might slip through. This method was used in [6] to determine popularity of anti-tampering techniques in malware. 6900 distinct malware samples were run in a clean, debugged and virtualized environments while having their behavior recorded. The results showed that 40% of samples executed fewer malicious actions in debugged and virtualized environments compared to the referenced one. A malicious event is defined by authors as an event that causes persistent changes to the system. A one-to-one comparison was done on the produced behavior logs.

Comparing behavior of malware in this fashion might not yield reliable results, as stated by [3], who implement a "record and replay" system in their work. They state that Windows OS is very complex and it is possible for the same malware sample to exhibit slightly different behavior when executed in the same environment. This could happen if, for example, the execution of malware relies on external sources, such as internet, some parameters change, such as CPU load. The authors define the behavior of malware to be a sequence of system calls with their arguments. System calls provide means for malware to communicate with its environment and to cause persistent changes. The authors record the system calls with arguments and output values in the reference system and then "replay" this information on the analysis system.

The main part of architecture of the [3] is a kernel driver installed on both reference and analysis systems. The driver hooks the System Service Descriptor Table (SSDT), which is a structure that is used as an entry point to every system call. On the reference system, this makes it possible to record the system calls performed by malware with the according arguments and outputs. When malware sample is run in the analysis system, the driver detects the identical system calls, but it replaces the results of these calls with values from the reference system, instead of passing them to the OS. Small timing differences can cause temporal deviations in behavior, e.g. asynchronous function calls. The system call matching algorithm is implemented in a flexible way because of that. It uses two queue structures and has "lookahead" functionality. One of the queues holds the system calls that appeared on the analysis system but not on the reference one, while the other queue holds the opposite - system calls that appear on the reference system but not in the analysis. Some system calls have side effects and can't be replayed - memory allocation e.g. These have to be forwarded to the OS

"Replaying" malware actions is a more complex task than simply observing its behavior, as it is necessary to handle all the possible side effects. The system also completely relies on in-guest components, which could be detected. This could be solved by employing HVM, but that could also decrease performance.

[30] is a differential analysis system built on top of Cuckoo. It is argued that

organizations employ different environments in their infrastructure, e.g. Linux servers and Windows hosts. The main goal of the system is to automatically execute a malware sample in different environments in order to assess the damage. A more hollistic view of malware behavior can be gained by executing it in different environments. The main difference from the work done in this project is the fact that the environments employed by authors are static, meaning that malware is allowed to run loose in VMs prepared on beforehand. The environment used in this project is more dynamic in a sense that a debugger must actually run and execute the malware. This project is also more focused on detecting anti-debugging capabilities in malware, while [30] is more focused on gaining a complete picture of a malware's sample behavior.

Chapter 4

Solution

4.1 Obtaining samples

In order to evaluate if the solution is working, a ground truth set of malware samples is needed. This set contains samples that certainly do have observable anti-debugging behavior. It can then be used as an input to the created solution to see if it can identify the samples as evasive. Two possibilities were explored for creation of this set.

It is possible to create a non-malicious sample with anti-debugging capabilities in the lab environment. This allows much more control than if a sample was found in the wild and it is possible to focus only on anti-debugging. The downside is that the methods used in the benchmark sample are the ones that have been already discovered and studied and newer methods might be missed. Another approach is to obtain samples from the wild, as infrastructure for doing so already exists in the form of malware research repositories, such as VirusShare [33]. The samples obtained in this way would also need to be studied in order to confirm the presence of anti-debugging methods.

4.1.1 Benchmark sample

A sample that performs a debugger check using an inline assembly function and simulates malicious activity by checking a registry key and creating a file was written in C++. The being_debugged function, shown in Listing 4.1, is used for debugger detection. This type of detection method was chosen because it is one of the simplest methods an attacker can use to detect if his malware is being debugged yet it is also stealthy enough to not use any APIs, rendering API - hooking instrumentation methods, such as ones used in Cuckoo and other sandboxes, blind to this attack. The goal of this method is to check whether BeingDebugged flag of Process Environment Block (PEB) structure [25] is set. The FS register is used by Windows OSs to store the Thread Environment Block (TEB), a data structure that describes currently running thread. The address of PEB is stored at offset 0x30h. The square brackets used with mov command mean that instead

of copying the address itself, the content stored at that address is copied into the EAX register. The BeingDebugged flag is stored at offset 0x02h and this value is moved into EAX. The last bit denotes whether the flag is set and so a bitwise and operation is used and the answer, either 0 or 1, is stored in the result variable.

Listing 4.1: Debugger detection function

```
bool being_debugged(){

  int result = 0;
  //Bitwise and operation is used to get the least significant bit(being debugged)
  _asm{
    mov eax, FS:[0x30]
    mov eax, [eax + 0x02]
    mov ebx, 01
    and ebx, eax
    mov [result], ebx
  };

  if (result == 1){
    return true;
  }

  return false;
}
```

The main function of the sample can be seen in Listing 4.2. The sample first checks if it is being debugged by calling being_debugged function and stops execution if that is true. If the sample is not being debugged, it will call the vbox_reg_check function and create a new text file to simulate malicious activity. The vbox_reg_check function uses RegOpenKeyEx and RegQueryValueEx, which are Windows API functions used for working with register. When this sample is run by Cuckoo, it becomes obvious if a debugger was detected by looking at API call trace, accessed registry keys and modified files lists of the analysis report, because the registry checks and file creation attempts will be absent.

Listing 4.2: The branching of behavior in the benchmarking sample

```
int _tmain(int argc, _TCHAR* argv[])
{
  if (being_debugged()){
      return 0;
  }
  else{
    bool result = vbox_reg_check();
    std::ofstream myfile;
    myfile.open("log.txt");
    myfile << "Malicious activity\n";
    myfile.close();
  }
  return 0;
}
```

4.1.2 Real world samples

The SW9 project [15] researched VM detection methods that could be used by malware to avoid being analysed. Malware samples were run in Cuckoo which generated behavior reports that were uploaded to MongoDB. The database contains analysis reports of 80000 malware samples. This resource is reused in this project in order to generate a ground truth of samples. The data stored in Cuckoo reports is not enough to detect malware that uses inline assembly methods for debugger detection, such as the benchmarking sample, but it is possible to find samples that check for debuggers in other ways. The reasoning is that if a sample checks for a debugger in a way that is detectable with Cuckoo data, it might also perform other checks that remained undetected so far.

Cuckoo reports contain a very rich API trace and debug detection methods that use Windows API were queried for first. The interesting API functions are IsDebuggerPresent and FindWindow, because these are the easiest for an attacker to use - one API call reveals whether a debugger is present or not. The former function checks the PEB structure and returns the value of BeingDebugged byte which is set by the debugger when it starts or attaches to a process. The latter function can be used to find open windows in the desktop. If supplied with arguments such as “OllyDbg”, it will return a handle to the window with that name. Lastly, MongoDB was queried for samples that check HKEY_LOCAL_MACHINE (HKLM) \SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug registry key. This key is associated with automatic postmortem debugger in Windows - a debugger that starts automatically if an application crashes. These 3 queries yielded 17 unique results from a total of 80000 samples.

The actual samples were obtained by searching VirusShare repository for matching md5-checksums present in corresponding Cuckoo reports. A Windows 7 VirtualBox VM with OllyDbg was created and samples were analysed by hand looking specifically for assembly code that checked BeingDebugged flag of PEB, as this is one of the easies checks for a malware writer to perform. The attempt was unsuccessful and no traces of inline assembly checks were found during manual analysis. Some of the analysed samples also made system unstable and crash. This can probably be attributed to other malware defense mechanisms, such as packing.

4.2 Differential analysis

Two environments are needed in order to perform differential analysis - one for reference and one for analysis. Both environments use VirtualBox for virtualization and Windows 7 32 bit edition with network simulated by INetSim with default configuration and no real access to the internet. The only difference between the two environments is the presence of GDB on the analysis machine, which was installed as a part of Minimalist GNU for Windows (MinGW) [27]. GDB was chosen for this project because of authors basic familiarity with it and

knowledge that it is possible to automate the process of running an application under it.

Just having GDB installed on one of the system would not be useful for the purpose of anti-debugger malware detection. A Cuckoo analysis package was created in order to automatically start the malware sample under the debugger and allow it to run. A Cuckoo analysis package tells the in-guest agent how to process certain types of malware, e.g. .pdf samples are opened with Adobe Reader. The .exe package was taken as a base for this project. It starts a malware process in a suspended state and injects it with the instrumentation dll to hook the API functions in default reference environment. The modified package, which is used in the analysis environment, attaches the GDB after the process is created but before it is resumed. GDB knows which process to attach to because it is supplied with a path to the process as well as the process identifier in the form of arguments. To automatically continue running the process, the debugger is supplied with "-ex cont" command. "-ex" executes one GDB specific instruction which in this case is "cont". When "cont" instruction is executed without parameters, the process is allowed to execute until a breakpoint, error or termination is encountered.

To generate the reports, a malware sample is run once in the reference and once in the analysis environments. The work performed in [3] states that a comparison based on only two executions is not reliable, and it is possible for the analysis results to be different even if the environment and the sample remained the same. This can be caused by e.g. a sample relying on external conditions, such as time or different command from the botmaster or internal conditions, such as CPU or memory load. The authors of [3] avoid this problem by using "record and replay" functionality of their system, as described in Section 3.2. This approach is very complex as not all system calls can be replayed, as some have important side effects, such as memory allocation. This is not possible to implement by only using Cuckoo modules and the source code would have to be modified or completely rewritten. Cuckoo injects a dll into malware process, CuckooMonitor, in order to hook APIs. This is the part that could have been modified to implement the "replay" functionality. It would have to be modified heavily as it would need access to the analysis report generated in the reference environment. An alternative solution would be to implement the "record and replay" by using HVM. It is impractical with the current Cuckoo architecture, which relies on an in-guest agent and a TCP connection to the host machine for communication. The strongest point of HVM, in the context of malware analysis, is the possibility of an agentless (no in-guest components) analysis system. As Cuckoo relies on an agent component, using HVM would mean redefining and reimplementing a big part of the system in addition to a necessary VMM component. The authors of Cuckoo have expressed an interest in the VM-introspection technique - analysing the malware with no in-guest components [14]. Implementing VM-introspection without support from VMM in general and **hvm!** (**hvm!**) specifically can be very difficult and it is possible that

Cuckoo will support this feature in the future. As of now, the "record and replay" functionality is out of scope for this project.

Listing 4.3: API call 34 in reference environment

```
{
    "category": "registry",
    "status": true,
    "return": "0x00000000",
    "timestamp": "2015-05-05
        03:51:13,251",
    "thread_id": "916",
    "repeated": 0,
    "api": "RegOpenKeyExW",
    "arguments": [
        {
            "name": "Handle",
            "value": "0x0000007c"
        },
        {
            "name": "Registry",
            "value": "0x80000002"
        },
        {
            "name": "SubKey",
            "value": "Hardware\\
                Description\\System"
        }
    ],
    "id": 34
},
```

Listing 4.4: API call 34 in analysis environment

```
{
    "category": "system",
    "status": false,
    "return": "0xc0000135",
    "timestamp": "2015-05-05
        03:52:17,496",
    "thread_id": "2412",
    "repeated": 1,
    "api": "LdrGetDllHandle",
    "arguments": [
        {
            "name": "ModuleHandle",
            "value": "0x02100210"
        },
        {
            "name": "FileName",
            "value": "mscoree.dll"
        }
    ],
    "id": 34
},
```

The comparison of two reports is implemented as a standalone utility in the form of a python script. Implementing comparison functionality in Cuckoo reporting stage was refrained from. As discussed in Section 2.4, this does not fit the Cuckoo workflow of analysing one malware sample at the time. The comparison utility takes two arguments - the IDs of the runs that are to be compared. These IDs denote the folder names where the analysis results are stored. The utility loads both reports and compares the API call trace, the modified files and registry keys. The results are stored as a text document that contains the differences between the reference and analysis runs. The current comparison mechanism goes through API calls line by line without considering the arguments. If a difference is spotted, both API calls are written down in the log. The registry keys and modified files are compared using the set-intersection operation, meaning that the order in which these were stored in the analysis file does not matter, compared to the API call comparison. Figure 4.3 and Figure 4.4 are two API calls of the benchmarking sample that was run in the reference and analysis environments respectively. The line by line API comparison mechanism spots a difference here, because the behavior of the sample branches at this point in the

execution and it calls two different APIs. This branching is clearly visible at line 6 of the source code demonstrated in Figure 4.2. The branching happens because the analysis environment runs the sample with a debugger attached and this is spotted by the being_debugged function. Listing 4.5 is the result of the comparison utility executed on benchmarking sample reports. The "-" means that a file or a registry key was present in the reference run but not in the analysis run, while a "+" means the opposite.

Listing 4.5: Comparison result of benchmarking sample that was run in reference and analysis

```
1 ========== API call names comparison ==========
2 Comparing process: 0
3 call number: 34 RegOpenKeyExW > LdrGetDllHandle
4 call number: 35 RegQueryValueExW > ExitProcess
5 ========== Files comparison ==========
6 − C:\Users\John\AppData\Local\Temp\log.txt
7 ========== Registry keys comparison ==========
8 − HKEY_LOCAL_MACHINE\Hardware\Description\System
```

4.2.1 Improvements

The comparison utility is a proof of concept and it suffers from some limitations. One of the possible improvements that could enhance the tool a lot is modification of the mechanism used for API comparison. There are two things that can be improved - the reliance on line-by-line comparison and arguments. The arguments of API calls are not considered in the current version and this can affect the detection rates. API functions such as LoadLibrary or GetProcAddress will be considered equal even if they are loading completely different modules. The comparison mechanism can also be improved by implementing a margin for error in form of buffers and lookahead functionality, as suggested in [3]. It is curently implemented in a very strict way - it compares API calls by their placement, meaning that API call number 1 of run 1 is compared only to API call number 1 of run 2. If the API call trace is identical for both runs, except that one of the runs has one less or one more call, the sample will be flagged as evasive, which is likely not the case.

Another weakness of the comparison mechanism, in case of files, is its reliance on names expressed in a string format. It is dangerous because malware can create files in analysis environment with the same name as in reference environment, but with completely different content. This can be avoided by using checksum techniques.

Because the sample is run by a real debugger in the analysis environment, all the exceptions generated by malware will be presented to the debugger first. This part is currently not automated, even though an exception generated by malware means that it has likely crashed and has to be restarted. A similar attack vector using int 3 instructions can be used by malware. Int 3 is an instruction

used by debuggers to generate software breakpoints. Malware can insert these instructions on purpose to slow down and confuse researchers doing manual analysis, but it will also stop the current system because skipping the breakpoint instructions is not yet automated.

Chapter 5

Conclusion

Arms race is the current state of computer security field and more malware are becomming "armored" with anti-tampering techniques, such as debugger detection. The goal of this project was to improve Cuckoo, one of the current open source sandbox analysis environments in active development, to be able to detect samples employing anti-debugging techniques. Two possible approaches to this problem were researched - instruction tracing and differential analysis.

Instruction tracing is based on CPU's debug facilities to log all the instructions executed by malware. This is a very fine grained analysis at low abstraction level. If this technique is implemented in a more stealthy fashion, using HVM, it becomes very expensive as every instruction executed by malware results in several context switches in CPU. It also requires a new analysis component which is capable of detecting whether the executed code was evasive or not. This solution is very complex as it would require modification of VMM code as well as the source code of Cuckoo. Differential analysis was chosen as an alternative. It requires execution of malware in two environments - reference and analysis. The analysis environment is modified to include traces of a debugger. After malware has been executed in both environments, the analysis reports are compared and a sample is declared evasive if there is a difference. This approach is cheaper than instruction tracing. It requires a comparison component in order to determine if the sample is evasive. This approach also captures all the detection methods, even 0-day exploits, but it is not possible to examine them in full detail like it could have been if an instruction trace was present.

An implementation that relies on Cuckoo customizable structure was created. An analysis package is used to attach a real debugger to the malware .exe file in the analysis environment. A utility tool that compares the results of two runs, one in reference and one in analysis environments, is implemented in python. The comparison mechanism is basic as it only considers API names, registry keys and files. The comparison of API calls can be further improved if arguments are considered and if the comparison is done in a more flexible way with lookahead functionality.

Bibliography

[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices*, 41(11):2–13, 2006.

[2] Hertzsprung at English Wikipedia. Privilege rings. `http://upload.wikimedia.org/wikipedia/commons/2/2f/Priv_rings.svg`. Retrieved 10.05.2015.

[3] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient detection of split personalities in malware. In *NDSS*, 2010.

[4] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11. Citeseer, 2009.

[5] Jurriaan Bremer and Thorsten Sick. Vmcloak. `http://vmcloak.org/`. Retrieved 10.05.2015.

[6] Xu Chen, Jonathon Andersen, Zhuoqing Morley Mao, Michael Bailey, and Jose Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 177–186. IEEE, 2008.

[7] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 289–298. ACM, 2013.

[8] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62. ACM, 2008.

[9] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6, 2012.

[10] Peter Ferrie. The ultimate anti-debugging reference. 2011.

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

[12] Gideon Gerzon. Intel virtualization technology processor virtualization extensions and intel trusted execution technology. https://software.intel.com/sites/default/files/m/0/2/1/b/b/1024-Virtualization.pdf.

[13] Claudio Guarnieri, Jurriaan Bremer, and Mark Schloesser. Mo malware mo problems - cuckoo sandbox. https://media.blackhat.com/us-13/US-13-Bremer-Mo-Malware-Mo-Problems-Cuckoo-Sandbox-Slides.pdf, . Retrieved 10.05.2015.

[14] Claudio Guarnieri, Allessandro Tanasi, Jurriaan Bremer, and Mark Schloesser. Cuckoo sandbox. http://www.cuckoosandbox.org, . Retrieved 10.05.2015.

[15] Sergey Gurkin. A study of virtualization detection methods in modern malware. Technical report, Aalborg University, December 2014.

[16] Red Hat. Libvirt. http://libvirt.org. Retrieved 10.05.2015.

[17] Thomas Hungeberg and Matthias Eckert. Inetsim. http://www.inetsim.org/. Retrieved 10.05.2015.

[18] Intel. Intel virtualization technology. http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html. Retrieved 10.05.2015.

[19] *Intel© 64 and IA-32 Architectures Software Developer's Manual*. Intel©, 1 2015. Volume 1,2,3.

[20] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barebox: efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 403–412. ACM, 2011.

[21] Tadayoshi Kohno, Andre Broido, and Kimberly C Claffy. Remote physical device fingerprinting. *Dependable and Secure Computing, IEEE Transactions on*, 2(2):93–108, 2005.

[22] Christian Kreibich, Nicholas Weaver, Chris Kanich, Weidong Cui, and Vern Paxson. Gq: Practical containment for measuring modern malware systems. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 397–412. ACM, 2011.

[23] Microsoft. Debugging reference. https://msdn.microsoft.com/en-us/library/ms679304%28v=vs.85%29.aspx,. Retrieved 10.05.2015.

[24] Microsoft. Local kernel-mode debugging. https://msdn.microsoft.com/en-us/library/windows/hardware/ff553382%28v=vs.85%29.aspx,. Retrieved 10.05.2015.

[25] Microsoft. Peb structure. https://msdn.microsoft.com/en-us/library/windows/desktop/aa813706%28v=vs.85%29.aspx,. Retrieved 10.5.2015.

[26] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pages 421–430. IEEE, 2007.

[27] Colin Peters. Minimalist gnu for windows. http://www.mingw.org/. Retrieved 28.05.2015.

[28] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7): 412–421, 1974.

[29] Honeynet Project. Know your enemy: Gen2 honeynets. http://old.honeynet.org/papers/gen2/. Retrieved 10.05.2015.

[30] Athina Provataki and Vasilios Katos. Differential malware forensics. *Digital Investigation*, 10(4):311–322, 2013.

[31] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting system emulators. In *Information Security*, pages 1–18. Springer, 2007.

[32] Jamie Riden. How fast-flux service networks work. http://www.honeynet.org/node/132. Retrieved 10.05.2015.

[33] J-Michael Roberts. Virusshare. http://virusshare.com/. Retrieved 28.05.2015.

[34] John S Robin and Cynthia E Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. Technical report, DTIC Document, 2000.

[35] Michael Sikorski and Adrew Honig. Fakenet. http://practicalmalwareanalysis.com/fakenet/. Retrieved 10.05.2015.

[36] Michael Sikorski and Andrew Honig. *Practical Malware Analysis*. No Starch Press, 1 edition, 2012. ISBN 1593272901.

[37] Symantec. Zeroaccess rootkit. http://www.symantec.com/security_response/writeup.jsp?docid=2011-071314-0410-99. Retrieved 10.05.2015.

[38] Amit Vasudevan and Ramesh Yerraballi. Stealth breakpoints. In *Computer security applications conference, 21st Annual*, pages 10–pp. IEEE, 2005.

[39] VMware. Vmware backdoor io port. http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1009458. Retrieved 20.12.2014.

[40] Xenproject. Xen. http://www.xenproject.org. Retrieved 10.05.2015.