DISTRIBUTED ON-DEMAND INTEGRITY MONITORING OF LEGACY APPLICATIONS AND RELIABLE ANALYSIS OF MIXED-MODE USER-KERNEL LEVEL ROOTKITS

by

SHABNAM ABOUGHADAREH

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2015

Copyright © by Shabnam Aboughadareh 2015

All Rights Reserved

To my love, Mehdi Azarmi, and my family for their huge love, support and encouragement.

ACKNOWLEDGEMENTS

First of all, I would like to express my deepest appreciation to my advisor Prof. Christoph Csallner for his huge support and his profound knowledge. Christoph has all the qualities that a world-class computer scientist must have. His trustworthiness makes him an ideal mentor and a true friend. I should give him credit for my success in receiving a best paper award as well as getting several job offers from top technology companies. Without Christoph's support, none of these successes would have been possible.

Beside my advisor, I thank the rest of my PhD dissertation committee, Prof. Matthew Wright, Prof. Donggang Liu and Prof. Jeff Lei. I took a course with Prof. Matthew Wright. His class was helpful to extend my knowledge in the area of computer security. Moreover, I really appreciate his useful comments which made me a better computer security researcher and also a better person. Thank you Matt for all your kind support. I would like to thank Donggang for dedicating his time to review some of my presentation slides and offering useful tips for an effective scientific presentation. Also, I thank Prof. Jeff for his insightful comments for my PhD proposal which helped me to improve my research work.

I sincerely thank Dr. Bahram Khalili, the graduate advisor at the CSE Department of University of Texas, Arlington. Bahram kindly shares his industrial and academic experiences with students to help them thrive in their research work and their future career. I cannot repay all he has done. I try to use his insight in my career as well as I can.

Dr. Matthew Elder hired me as a research intern at Symantec Research Labs (SRL). He was always available to help and his encouragements made me learn a lot. Thank you Matt for all you have done for me. I thank Dr. Nathan Evans and Dr. Azzedine Benameur, the principal researchers at SRL for their insightful comments as well as Martin Schulman, the technical director at Symantec, for his support during my internship.

I would like to thank Prof. Ramez Elmasri, Prof. Gautam Das, Prof. Manfred Huber, David Levine and Prof. Chengkai Li for all I learned from them during my PhD course work. Thank you Prof. Babak Sadeghiyan at Amirkabir University of Technology for the computer security foundation I got from you. Thank you Prof. Hossein Pedram, Bahman Pourvatan and Prof. Saeed Shiry at Amirkabir University of Technology for your encouragements.

Thank you Laleh Ghandehari, Mehra Nouroz Borazjany, Soheil Shafiee, Ishtiaque Hussain, Mainul Islam, Jing Xu, Tuan Anh Nguyen, Azade Nazi, Abolfazl Asudeh, Zahra Ghorban Oghli and Hossein Atashi my labmates, classmates and friends. Also, I would like to say thank you to the staff at CSE Department of University of Texas, Arlington, Sherri Gotcher, Pamela Mcbride, Camille Costabile and Irie Bito.

I am grateful to my family for their huge love and support. I sincerely thank my parents Mansoureh (Suri) Yaghini and Homayoun Aboughadareh and my brother Reza Aboughadareh. Thank you Batoul Rabbi Yar, Minoo Yaghini, Mahboubeh Yaghini, Jacob Yaghni, Joseph Yaghini, Batoul Ghorbian and Khosro Azarmi.

Finally, I would like to express my gratitude to my love, Mehdi Azarmi. Mehdi's talent, kindness, patience, hard work and enthusiasm have been always inspiring for me.

December 1, 2015

ABSTRACT

DISTRIBUTED ON-DEMAND INTEGRITY MONITORING OF LEGACY APPLICATIONS AND RELIABLE ANALYSIS OF MIXED-MODE USER-KERNEL LEVEL ROOTKITS

Shabnam Aboughadareh, Ph.D.

The University of Texas at Arlington, 2015

Supervising Professor: Christoph Csallner

The increasing number of malicious programs has become a serious threat. The growth of malware samples has led computer security researchers to design and develop automatic malware detection and analysis tools. At the same time, malware writers attempt to develop more sophisticated malware that makes detection and analysis hard or impossible. In my dissertation I explore the problems of current malware detection and analysis techniques by providing the proof-of-concept implementation of malware samples that cannot be detected or fully analyzed by current techniques. My dissertation identifies three problems in the current solutions.

First, regarding the limitations of monitoring the integrity of legacy programs such as expensive cost of migrating to modern and more secure platforms, code injection rootkit attacks on legacy applications are hard to detect. Second, the complex malware codes manipulate or intercept the malware analysis components which reside on their execution domain (user-mode and kernel-mode).Third, a mixed-mode

malware, which contains interdependent user-mode and kernel-mode components, misleads or foils single-domain analysis techniques.

To address the first problem, I propose TDOIM (Tiny Distributed On-Demand Integrity Monitor). TDOIM is a client-server scheme that periodically monitors applications to detect the malicious behavior injected by an attack. Specifically, it periodically compares the runtime state of all instances of the legacy application. If some instances start to diverge from the rest, this is an indication that the diverging instances may have been manipulated by malware. In other words, the server periodically infers and updates a white-list directly from the monitored application instances and checks all clients against this dynamic white-list. TDOIM installs a tiny client-side agent on legacy platforms with minimum attack surface and it does not require recompilation or restart of the monitored legacy application.

In order to address the problems of the current malware analysis techniques, I present the first mixed-mode automatic malware analysis platform called SEMU (Secure Emulator). SEMU is a binary analysis framework that 1) it operates outside the operating system and thereby outside the domain of user-mode and kernel-mode malware. 2) it deploys a novel mixed-mode monitoring of malware operations that is effective against sophisticated user-kernel level rootkit samples and kernel-mode exploits.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iv

ABSTRACT vi

LIST OF ILLUSTRATIONS xii

LIST OF TABLES xiii

Chapter Page

1. INTRODUCTION 1

1.1 Malware Detection Techniques and Motivation of TDOIM Approach . 1

1.2 Malware Analysis Techniques and Motivation of SEMU Approach . . 6

1.3 Dissertation Overview 8

2. BACKGROUND 9

2.1 Memory Management and Memory Layout 9

2.2 Code-Injecting Rootkits 10

2.3 Attack Surface of Current Anti-malware Approaches 11

2.4 Malware Analysis Tools 12

2.4.1 Virtual Machine and Introspection 12

2.4.2 Inside- vs. Outside-the-Guest VMI 13

2.4.3 Scope: Single-Domain vs. Whole-System 15

3. TDOIM APPROACH AND DESIGN 17

3.1 Assumptions and Threat Model 17

3.2 TDOIM Architecture Overview 19

3.3 Use-Case Scenarios 20

3.4 Monitoring Partially Swapped-Out Memory Segments 21

3.5 Monitoring Dynamically Loaded Code 23
3.6 TDOIM Agent's Attack Surface 25
3.7 TDOIM's Server Component 25
4. TDOIM IMPLEMENTATION 27
4.1 TDOIM's Client-Side Agent 27
4.2 Obtaining Virtual Address Ranges in the Agent 27
4.3 De-Relocating Virtual Addresses for Hashing 29
4.4 Hashing Physical Memory 31
4.5 Sending Hashes to the Back-end 31
4.6 TDOIM's Back-End Application 32
5. TDOIM EVALUATION 33
5.1 Research Questions 33
5.2 Subjects: Kernel and User Level Rootkits 34
5.2.1 Exchanging Libraries via LD_PRELOAD 35
5.2.2 Jynxkit: Exchanging libraries via ld.so.preload 35
5.2.3 Patching the Standard User-Mode Program Loader 36
5.2.4 InjectSO: Diverting Process Execution 36
5.2.5 Hooking the System Call Table 37
5.2.6 In-line Function Patching with Suterusu 37
5.3 Experiments 38
5.3.1 RQ1: TDOIM's Runtime Overhead 39
5.3.2 RQ2: TDOIM's False Positives 40
5.3.3 RQ3: TDOIM's Rootkit Detection Performance 41
6. TDOIM LIMITATIONS AND DISCUSSION 42
6.1 Stack: Return-Oriented Programming (ROP) 42
6.2 Kernel Heap: Attacks on Dynamic Data Structures 42

6.3 Potential Attacks on TDOIM 43
7. MIXED-MODE MALWARE AND SEMU ANALYSIS REQUIREMENTS . 44
7.1 Misleading User-Only Analysis 45
7.2 Misleading Kernel-Only Analysis 47
7.3 Misleading Inside-the-guest VMI 49
7.4 Analysis Requirements 50
8. SEMU APPROACH 53
8.1 Reverse-engineered OS Model: PDB 55
8.2 Pre-Execution: Create Shadow Memory 56
8.3 Whole-System Malware Analysis 58
8.3.1 Malware Logging 60
8.4 Post-Execution: Log Analysis 61
8.5 Implementation in QEMU 62
9. MIXED-MODE MALWARE SAMPLES 64
9.1 Misleading User-Only VMI 64
9.2 MDL System Call Semantic Modification 65
9.3 User-Level Acts on DKOM Attack 66
9.4 User-Level Acts on DKSM Attack 67
9.5 Stuxnet's Kernel Exploit 68
9.6 User-Level Malware Acts on User-Mode Unhooking of Mapped Kernel SSDT 68
10. SEMU EVALUATION 71
10.1 Analyzing Mixed-Mode Malware (RQ1) 71
10.2 Malware Analysis Execution Time (RQ2) 72
10.2.1 SEMU vs. TEMU's Inside-the-Guest VMI 72
10.2.2 SEMU vs. Ether's Single-Domain Analysis 74

11. SEMU LIMITATIONS AND DISCUSSION 76
11.1 Run-time Patching Support 76
11.2 Taint Analysis Capability 77
11.3 Security Applications 77
11.4 Analysis Speedup. 77
11.5 Handling Anti-emulation Attacks 78
11.6 Execution Paths Coverage 78
11.7 Defending Against Obfuscated Malware. 79
12. CONCLUSION 80
13. RELATED WORK TO TDOIM 81
13.1 Detecting Attacks on the Kernel 82
13.2 Leveraging Virtualized OS Stacks or Special Hardware 83
13.3 Comparing Memory with Disk Contents 84
14. RELATED WORK TO SEMU 86
14.1 Attacks on Trusted Kernel Code. 87
14.2 Evading Virtual Machines. 87
14.3 Rootkit Analysis Techniques. 88
14.4 Automatically Generating Outside-the-Guest VMI Tools. 88
14.5 Protecting Against Kernel Exploits. 89
14.6 Offline and Interactive Analysis Tools. 90
Appendix
A. Linux Implementation of Client-Side Agent 92
REFERENCES 94
BIOGRAPHICAL STATEMENT 108

LIST OF ILLUSTRATIONS

Figure Page

2.1 Two dimensions of dynamic malware analysis 12

2.2 Architecture comparison 14

3.1 Example configuration of TDOIM monitoring four instances of a legacy application. 19

3.2 Single-hash of available pages (left) vs. hash of hashes (Hash_T) plus individual page hashes Hash#1 to Hash#N (right). 23

3.3 Two application instances yield different "hash of hashes" values (Hash_T1 $\neq$ Hash_T2), as only one has loaded the benign_lib2.so dynamic library. 24

4.1 Address space of the APP1 example application, including the APP1 executable and several dynamically loaded libraries. 29

7.1 Example malware misleading Ether's user-only VMI 46

7.2 Stuxnet kernel exploit 48

7.3 Example malware payload that misleads TEMU 52

8.1 SEMU architecture and main execution phases 53

9.1 MDL system call semantic modification. 66

9.2 Unhooking system call table by a user-mode malware. 69

LIST OF TABLES

Table Page

5.1 Rootkit subjects run either in user (u) or kernel (k) mode; KV = Linux kernel version; CPU = 32 vs. 64 bit; Loc = main memory (m) vs. disk (d). 34

5.2 Breakdown of TDOIM's average runtime overhead; Loc = TDOIM client (c) vs. server (s). 39

5.3 The 7 kernel modules that produced false alerts and their number of pages; FH = false hashes; FP = false positives. 41

8.1 Kernel execution trace format 60

10.1 Results of analyzing six mixed-mode malware samples 72

10.2 Performance comparison of TEMU and SEMU 73

10.3 Fine-grained VMI: Instruction tracing in Ether and SEMU 75

CHAPTER 1

INTRODUCTION

Malware poses significant challenges to modern society. Among others, malware can take control of a victim system and perform arbitrary actions such as logging individual keystrokes to steal online banking passwords as wells as changing operating system's critical codes and data structures to remain undetected for years.

These challenges are real and affect many people. For instance, a survey conducted in January 2011 found that *one third* of the 2,089 U.S. online households surveyed had been victims of malware in the previous year [1]. The survey estimated that in the previous year malware cost U.S. consumers overall USD 2.3 billion.

The motivation of this dissertation is to address some of the problems in the state-of-the-art malware detection and analysis techniques. In the following, I discuss the motivation of TDOIM and SEMU research works, my proposed malware detection and analysis approaches for resolving some of the problems of the current techniques.

1.1 Malware Detection Techniques and Motivation of TDOIM Approach

Detecting code-injection rootkit attacks that are conducted with new malware is notoriously hard. Such attacks may be launched by zero-day attacks or other avenues and are generally not detected by current antivirus approaches. New malware may inject malicious behavior into a trusted application, may be persistent on the infected system over long periods of time, and may manipulate the infected system in ways that make it very hard to be detected. Recent examples of code-injection rootkit attacks include Stuxnet, Duqu, and Flame [2, 3, 4]. Detecting such attacks is hard,

even if an administrator has great flexibility in preparing the system for malware attacks [5].

Administrators of legacy applications, however, are more constrained, which poses the following five additional key challenges for detecting such attacks. (1) First, legacy applications are usually kept around because they deliver high value. As a baseline, a security solution for such a legacy application should thus have a minimal attack surface, to not expose the application to additional threats. (2) Second, if such a high-value legacy application gets infected, the response should be immediate, to minimize negative consequences.

Sometimes an infection of a high-value legacy application is suspected but cannot be established with existing tools. (3) Then it is often important to be able to install an additional malware detection system on demand, even after the application may have been infected, and ideally without having to restart the possibly infected application. (4) Fourth, legacy applications are often not well maintained. Their source code may have been lost and it may no longer be clear which version of the application has been installed on which machines.

Finally (5), a legacy application is often tightly integrated with legacy platforms and operating systems. Such platform dependencies may make it prohibitively expensive to migrate such an application to the modern platforms that are often assumed by anti-malware research. Example assumptions include certain hardware such as secure crypto co-processors (e.g., TPM) and hardware extensions for virtualization (e.g., intel VT-x or AMD-V) as well as modern virtualized operating system stacks.

High-value legacy applications are used widely in all industries (including health care, banking, transportation, and national security). Attacks with new malware can be costly for any application. For example, an attack may eventually lead to a data breach, which in 2014 cost an affected company in the U.S. on average over one

million dollars [6]. Given that legacy applications are often high-value and detecting malware attacks is more challenging in legacy applications, effective new-malware detection approaches for legacy applications are urgently needed.

Existing approaches are not effective as they do not meet all five challenges. For example, a widely accepted best practice for detecting code-injecting rootkits in legacy applications is running host-based third-party antivirus tools and they are in wide use [7, 8]. However this straightforward approach does not address challenge (1). Current antivirus tools are essentially large and complex high-privilege extensions of the operating system that is running the protected application. Such an extension dramatically increases the malware attack surface—many attacks on antivirus tools have been described [9, 10, 11, 12].

Also, (challenge 2), it may take weeks before a current antivirus tool detects a new malware attack, since most antivirus tools work by comparing application files to a blacklist of known malware signatures. For a new malware attack it takes time for antivirus vendors to discover the malware, distill it into signatures, and push the signatures to the blacklist of the protected hosts. For example, in a 2007 study on some 8k malware samples, young (less than one week old) malware samples went undetected at a rate from over 20% to over 60%, depending on the tool vendor [13]. A 2014 study had similar results, e.g., antivirus tools could still not reliably identify malware samples that were several months old [14].

Cloud-based antivirus approaches address challenge (1) [13, 15, 16, 17]. But they do not fully address challenge (2), because they remain blacklist-based, which may leave high value legacy applications exposed to new malware attacks for weeks.

To support legacy applications, extensions of host-based antivirus tools catalog the files of all applications into a whitelist [18, 8]. However this approach does not

meet challenges (1) or (3), as it still has a large attack surface and may add to the whitelist malware that is already on the host.

Other anti-malware work also does not adequately support legacy applications, since it places strong assumptions on the monitored applications and therefore struggles with challenges (3), (4), and (5). For example, recent techniques assume virtualization [19, 20, 21, 22, 23, 24], certain VMs [19], or special hardware such as TPM or PCI add-in cards [25, 26, 27, 28, 29, 30].

To address all five challenges of detecting new code-injection malware attacks, I propose a Tiny Distributed On-demand Integrity Monitor (TDOIM), an approach that works well with legacy applications. At a high level, my approach is a client-server scheme that periodically monitors applications to detect the malicious behavior injected by an attack. Specifically, I periodically compare the runtime state of all instances of the legacy application. If some instances start to diverge from the rest, this is an indication that the diverging instances may have been manipulated by malware. In other words, the server periodically infers and updates a white-list directly from the monitored application instances and checks all clients against this dynamic white-list.

Specifically, on each platform running the legacy application dissertation install a tiny client-side agent. I keep the agent's functionality minimal to minimize the attack surface added to the machines running the application (requirement 1). This agent does not require special hardware or virtualization, which together with its minimal functionality allows deployment on a wide variety of legacy platforms (requirement 5). This client-side agent can be installed on a platform during a malware attack and does not require recompilation or restart of the monitored legacy application (requirement 3).

Each client-side agent periodically computes hash values of the memory allocated for kernel, application, and device drivers and sends the hashes to the server-component. The server detects outliers only within the group of reported hashes and thereby does not require any prior knowledge about original binaries, file signatures, blacklists, or whitelist (requirement 4). More importantly, the server detects potential malware attacks immediately and does not have to wait until a third party has released corresponding malware signatures (requirement 2).

The voting-based scheme works because a malware attack usually spreads relatively slowly across the various locations running the monitored application. While the malware has only infected a minority of the monitored application instances, the voting scheme can detect the malware infection as outliers.

To evaluate TDOIM, I implemented TDOIM for Linux and conducted several small experiments. For the experiments I used different user-mode and kernel-mode rootkits that perform code injection, hooking, and in-line patching to infect legacy applications. In the experiments TDOIM could always pinpoint the compromised systems and the infected memory regions in these systems. In the experiments the runtime overhead on the client machines was moderate. The amount of false positives was relatively low for kernel modules (4% of the modules' pages) and zero for the OS kernel and user-space applications.

To summarize, this dissertation makes the following major contributions.

- This dissertation describes a set of five key challenges faced when detecting rootkits in legacy applications.
- The dissertation describes TDOIM, an approach for detecting user-mode and kernel-mode rootkits in legacy applications. TDOIM has a tiny footprint and therefore small attack surface on the monitored machines and enables online rootkit detection.

- To evaluate TDOIM, the dissertation describes a prototype implementation of TDOIM for recent versions of Linux.
- The dissertation provides an initial empirical evaluation of TDOIM on several user and kernel mode rootkits, where with moderate overhead and a relatively low false positive rate TDOIM achieved a 100% rootkit detection rate.

1.2 Malware Analysis Techniques and Motivation of SEMU Approach

Malware analysis has been studied widely, using many real-world malware samples. However I am not aware of existing work that exposes malware analysis to *mixed-mode* malware. Mixed-mode malware is a type of malware that (a) has interdependent user- and kernel-mode components and (b) may actively attack or subvert malware analysis components. I say that malware components are interdependent if the second component performs its main malicious payload only if the kernel manipulation of the first component succeeds.

Given the big impact malware has, it is important for malware analysts to analyze malware and develop countermeasures. For such malware analysis, an important technique is to monitor the execution of actual malware with state-of-the-art dynamic malware analysis tools such as those based on TEMU [31], Anubis [32], and Ether [33]. Monitoring malware executions allows malware analysts to reverse-engineer and understand the subtle details of how a concrete malware instance functions. Analysts can leverage such understanding when designing and deploying malware countermeasures. Such countermeasures can ultimately protect a wide range of computers from both the specific malware analyzed and from similar, derived classes of malware [32].

Current dynamic malware analyses [31, 33, 32, 34, 35] are not effective for analyzing mixed-mode malware. The reason is that existing malware analysis tools suffer from one or both of the following shortcomings.

(1) First, many current analysis techniques place analysis components in the domain in which the malware is executing and thereby expose the analysis to malware manipulations. These approaches are referred to as *inside-the-box*, *inside-the-guest*, on *in-guest*. For example, popular analysis platforms such as TEMU and Anubis run the malware in a virtual machine. To inspect the state of the malware and the VM, such malware analyses often place some virtual machine introspection (VMI) components inside the VM, which exposes VMI and thereby the entire analysis to malware manipulation.

(2) Second, many approaches focus on a single domain, either kernel-mode or user-mode, but fail to fully capture malware that operates in both modes [33, 36]. For example, Ether leverages hardware virtualization extension to operate outside-the-guest but focuses only on user-mode analysis. Ether relies on the integrity of the kernel when inspecting the system state and malware behavior. However, mixed-mode malware manipulates the OS kernel and thereby foils such single-mode analysis.

To address the limitations of the existing techniques and analyze mixed-mode malware effectively, I propose a novel dynamic malware analysis tool called SEMU (*Secure EMUlator*). SEMU operates both outside-the-guest and across kernel and user modes. In my experiments I also found that SEMU's overhead was in line with closely related existing tools, i.e., Ether and TEMU. While my current SEMU implementation is for Windows, my analysis approach could also be implemented for other operating systems such as Linux. To summarize, the SEMU approach makes the following major contributions.

- I describe and provide practical implementations of several mixed-mode malware samples. Mixed-mode malware cannot be fully analyzed with current state-of-the-art dynamic malware analysis tools such as those built on TEMU, Anubis, and Ether.

- I present SEMU, a whole-system outside-the-guest dynamic malware analysis tool that can effectively analyze mixed-mode malware. For example, SEMU *detects* and *analyzes* kernel exploits that cannot be analyzed by current kernel-mode analysis approaches.
- I provide the first empirical evaluation of the runtime characteristics of a whole-system outside-the-guest dynamic malware analysis tool such as SEMU.

1.3 Dissertation Overview

In the remainder of this dissertation, I explore current well-known malware analysis and detection approaches and their limitations in chapter 2. I discuss TDOIM, the malware detection approach in chapter 3, 4, 5 and 6. I describe SEMU approach, its design and implementation and proof-of-concept implementations of mixed-mode malware samples, in chapter 7, 8, 9, 10 and 11. Finally in chapter 12, I conclude the dissertation.

CHAPTER 2

BACKGROUND

This chapter provides necessary background information on memory management and memory layout techniques that are common across many platforms and operating systems, how rootkits take advantage of these existing detection techniques and how current malware analysis are designed.

2.1 Memory Management and Memory Layout

While details differ among the various platforms and operating systems, there are two broad categories of memory address space, kernel and user. Most widely used operating systems are monolithic and thus the kernel and its extensions have full access to the same address space. Each user-mode application has its own address space and cannot directly access the memory of other applications or the kernel.

Code and its data exist in two main forms, on disk in files and loaded in main memory. The compiler typically places both code and data in a number of segments or sections. While the terms have well-defined meanings that differ across platforms, in this dissertation I use *segment* and *section* interchangeably to refer to a chunk of either code or data either on disk or in main memory. For each segment the compiler can set access rights (read, write, and execute), which most operating systems enforce via the platform's memory management hardware support. For example, code is typically placed in executable non-writable segments and constant data values are typically placed in non-executable non-writable segments.

On most platforms, each application is allocated in a contiguous block of virtual memory, which the platform maps to a possibly non-contiguous set of pages in physical (main) memory. While each machine running a given application may place a given page at a different physical address, the page's virtual address is fixed at load-time and therefore identical across machines. A program can run with only some of its pages in physical memory, the remaining pages are swapped out to secondary storage, such as a disk. Kernel addresses are often an exception, as several operating systems such as Linux make sure that their kernel code and read-only data segments are both in contiguous chunks of physical memory and never swapped out.

Most platforms have mechanisms for dynamic code loading, e.g., to support a device the user plugged in at runtime or to handle various input values. This code is loaded into an existing address space and adds additional code and data segments. Common examples in the kernel are kernel-level device drivers. In user-mode, many platforms have mechanisms to dynamically load code libraries.

2.2 Code-Injecting Rootkits

Rootkit traditionally meant software that is used in an initial attack to elevate a user to root access on a victim system. However in the current literature rootkit refers to software that an attacker uses after gaining the desired level of access through some other attack such as a zero-day exploit. In this new definition, which I use in this dissertation, a rootkit uses the existing level of access to create a more persistent backdoor to retain access in the long term and possibly hides itself and malicious payloads from anti-malware tools.

Many rootkits work by injecting code into the victim system [37, 38]. To inject code, rootkits use their high OS privilege level to overwrite code segments on disk or in main memory or load additional (malicious) code segments (e.g., as a dynami-

cally linked library) and change read-only data segments (i.e., that contain function pointers) to link to them. A rootkit may inject code at any level, i.e., in the kernel, in kernel extensions such as device drivers, and in user-mode applications.

Since code-injecting rootkits are common and hard to detect in legacy applications, this dissertation focuses on detecting rootkits in systems running legacy applications. Since buffer overflow attacks have received a lot of attention, I explicitly distinguish them from code-injecting rootkits. A buffer overflow typically overwrites parts of the call stack and then may cause control-flow to jump to this new material in the stack, which may also jump to some more material in the heap. However in this dissertation I focus on rootkits that inject code into code segments.

2.3 Attack Surface of Current Anti-malware Approaches

Many current anti-malware tools have a large attack surface and thus are vulnerable to many attack vectors [9, 10, 11, 12]. Due to their large number and complexity, in this section I do not attempt to enumerate all such attack vectors. Instead I list a few of the attack vectors that are common in existing tools but are absent in TDOIM.

An example functionality is that current antivirus tools store application white-lists and malware black-lists on each monitored client. (1) First, such list stores can be manipulated on disk or in memory. (2) Second, the process of adding new elements to the lists can be intercepted or subverted, at the source server, during transmission, or at the client-side antivirus tool destination. (3) Finally, retrieving elements from the lists can be intercepted.

In addition to these direct attack vectors, there are also the following well-known indirect attacks. Since each such functionality is implemented in code, existing anti-malware tools contain a lot of code, which presents many opportunities for code vulnerabilities and zero-day exploits. The functionality is also spread across many

places such as disk, memory, and registry, with their own attack vectors. To access such system resources, current anti-malware tools use a large number of OS APIs, such as system calls, which can be manipulated [39].

2.4 Malware Analysis Tools

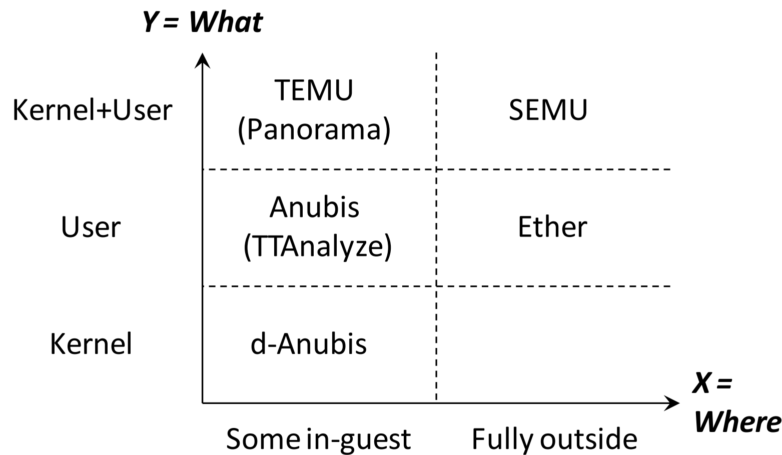

Figure 2.1. Two dimensions of dynamic malware analysis: Scope (y-axis) and whether an analysis places some of its components inside-the-guest (x-axis)..

I can classify dynamic malware analyses along two dimensions. Figure 2.1 shows on the x-axis whether an analysis places some of its components inside the guest OS. Having components inside-the-guest makes an analysis vulnerable to malware attacks. The y-axis captures the scope of the malware analysis. SEMU is the only analysis that combines a kernel+user scope with being fully outside-the-guest.

2.4.1 Virtual Machine and Introspection

Malware may corrupt the OS it is running on and may in turn corrupt other programs running on the OS, including malware analyses. To retain control of the machine, malware is thus typically run on a (guest) OS that is installed on a hardware

emulator or *virtual machine* (VM) [40, 31, 33]. The VM runs on another OS called the host OS. A VM also enables an analyst to inspect all interactions between malware and guest OS.

To be useful, a VM-based malware analysis tool has to query the current VM state. This state is readily available in a low-level form, i.e., in terms of register values and main memory bytes. But a malware analysis tool is typically written in terms of higher-level OS abstractions such as threads and processes. This gap between the readily available low-level hardware state and the desired high-level operating system state is called the *semantic gap* [41].

Virtual machine introspection (VMI) bridges the semantic gap, by reconstructing high-level OS information from low-level hardware state [42]. Without bridging the semantic gap, even basic analysis tasks such as logging the execution trace of a malware sample become very hard if not impossible.

2.4.2 Inside- vs. Outside-the-Guest VMI

Since malware may corrupt anything within its reach, it is useful to determine if a VM-based malware analysis places any of its components inside the guest OS. This classification notably differs from the more common classification of where the majority of the malware analysis components reside. While TEMU and Anubis are often described as outside-the-guest [43], they place at least some of their VMI components inside-the-guest.

Figure 2.2 gives an overview of state-of-the-art tools including TEMU [31] and Anubis [32] that both place a custom kernel-mode VMI driver inside the guest OS. Such a driver is outside the malware's reach if the malware is restricted to user-mode. The existing drivers differ in how they pass OS state to the analysis. The TEMU driver writes to a predefined I/O port. The Anubis driver reports to a user-

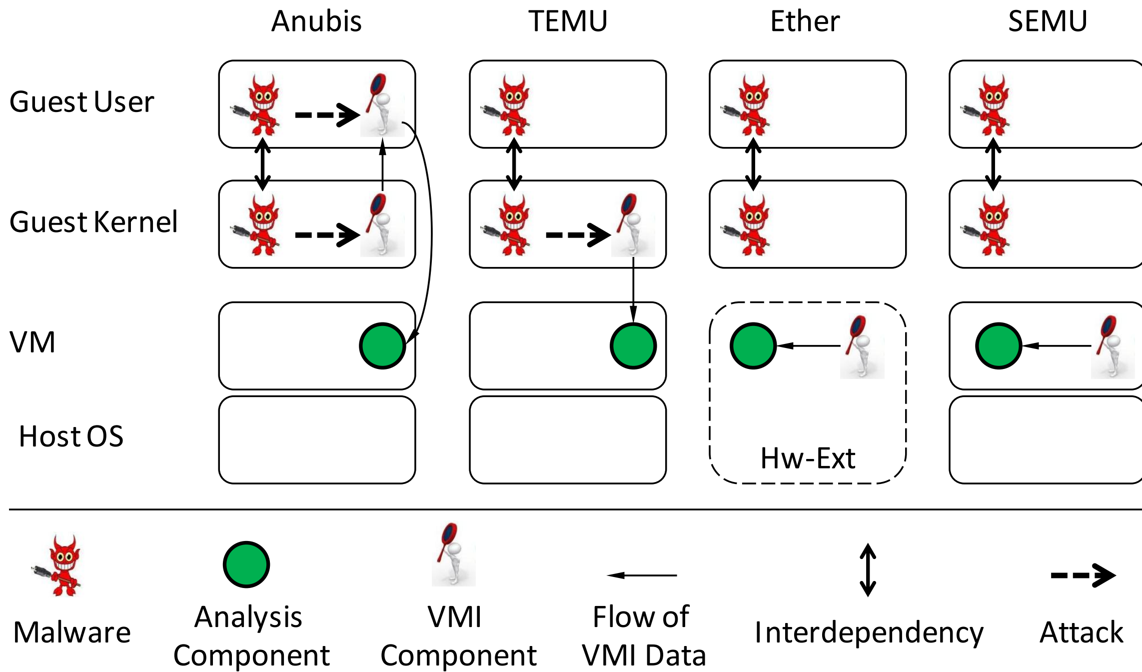

Figure 2.2. Architecture comparison. Not shown is malware's main payload attacking the guest OS. Hw-Ext = Hardware virtualization extension. .

mode application, which communicates with the analysis component over a virtual network. The Anubis extension dAnubis [44] patches the kernel functions that load kernel-mode modules. The patched functions then notify dAnubis when kernel-mode malware is loaded.

Ether [33] performs outside-the-guest VMI by relying on a processor-specific hardware virtualization extension (Intel VT-x [45]). The hardware extension allows Ether to run in a privileged hypervisor mode. Certain events in the monitored VM such as some exceptions trigger the hypervisor mode and return control to Ether. For example, to log an instruction, Ether sets a trap flag before the instruction to force a debug exception that returns control to Ether.

While outside-the-guest VMI protects Ether from some malware attacks, relying on hardware virtualization extensions has two drawbacks. First, for a fine-grained

analysis that logs every instruction, Ether has to install a trap flag before every instruction and handle the resulting debug exceptions. This single-step mode slows down execution by three orders of magnitude [46]. Second, Ether requires that the underlying processor supports hardware virtualization extensions, but many processors do not meet this requirement.

Besides well-known analysis tools I described, there are several VM-based approaches that focus on malware and rootkit detection or protection [47, 48, 49, 20, 50, 51, 52, 19, 53]. However none of these existing tool can *monitor* and *log* the execution of kind of mixed-mode malware or kernel exploits described in this document.

2.4.3 Scope: Single-Domain vs. Whole-System

Many analyses capture either user-mode or kernel-mode malware activities, but not both. Such a single-mode focus is insufficient for fully analyzing mixed-mode malware.

An example whole-system analysis is Panorama. It is a part of BitBlaze [31] and uses TEMU for performing whole-system taint analysis [54]. But as TEMU, Panorama performs some VMI tasks inside the guest OS. Panorama and similar approaches track the information flow of sensitive data such as keystrokes and network packets to detect and analyze malware [55, 21, 56, 57, 58]. But mixed-mode malware may not have such data flows and thus cannot be fully analyzed by such approaches.

An example user-only analysis is Ether. It assumes the integrity of kernel data (i.e., the system call table) and control-flow. A mixed- mode malware can drop a kernel-mode malware that manipulates the system call table and mislead Ether's analysis.

An example kernel-only analysis is the Anubis extension dAnubis [44]. It is notified whenever malware is loaded into kernel memory. But being single-mode,

dAnubis may miss attacks from user-space. For example, user-mode malware can access the kernel via zero-day exploits, such as bugs in standard device drivers. In such cases a malicious payload executes with kernel privileges without loading a kernel-mode module.

CHAPTER 3

TDOIM APPROACH AND DESIGN

This chapter gives an overview of the main assumptions behind TDOIM and its resulting architecture. For example, TDOIM monitors efficiently code that can be loaded dynamically and get partially swapped out.

3.1 Assumptions and Threat Model

I assume that the adversary attempts to compromise a legacy application. A legacy application may run on a user's system in user-mode, kernel-mode, or both. A user system is the operating system (OS) and all applications running on the OS. The OS may run directly on hardware, in a virtual machine (VM) instance, or in a container [59].

I also assume that many instances of a legacy application are running at the same time. For example, this assumption is met by a large distributed application that contains many instances of the same legacy application (perhaps running in data centers or on the cloud) or by many users running instances of the same stand-alone application on their respective machines. A malware attack is carried out by executing a malicious payload on one or more machines running a given legacy application.

As common in rootkit-type attacks, I assume the adversary has full access to the whole attacked machine, including the file system and all memory address spaces. The adversary may exploit this access and continuously hack a victim machine's operating system and the legacy application running on the machine. As part of such

an attack, the attacker injects code for both malicious payloads and to change the OS in order to hide the payload.

The adversary may carry out the attack by manipulating binaries on the disk, by infecting loaded images in memory, or both. For code injection into the different kinds of software running on the victim's machine, a concrete attack may include a combination of the following common rootkit techniques [39].

- For injecting code into the kernel, the adversary may obtain a higher privilege level (such as root access) and inject the malicious payload by patching the binaries on disk, hooking the system call table, or patching (overwriting) internal operating system code sections and read-only data sections in memory.
- For injecting code into kernel extensions, the adversary may similarly patch the binaries on disk or manipulate the code section or read-only data section of legitimate device drivers or kernel-mode services in memory.
- Finally, for user-mode applications, the adversary may patch a binary on the disk or inject a malicious library in the address space of a running legitimate user-mode process or service, by manipulating the code section in memory.

TDOIM relies on a hash function for hashing memory values with relatively few collisions. That is, I assume that two memory regions with different contents will produce different hash values with very high probability. An attacker could carefully craft a manipulation such that both the original and the manipulated memory yield the same hash value. But by using a good hash function this would be hard.

TDOIM also works best if there are relatively few variants of a given legacy application. Specifically, I assume that the legacy applications are relatively homogenous in terms of their application version and how each version has been compiled. This is often the case, as developers tend to use the same compiler for longer periods of time and only a few versions of a given legacy applications are in wide use.

3.2 TDOIM Architecture Overview

Since the TDOIM client component resides in the address space of the OS that runs the legacy application, there is a risk that a rootkit attacks the TDOIM client. To minimize this risk, the TDOIM client has a tiny feature set and therefore a tiny attack surface. Important TDOIM features that related monitoring approaches include on the client TDOIM therefore places on its back-end server component.

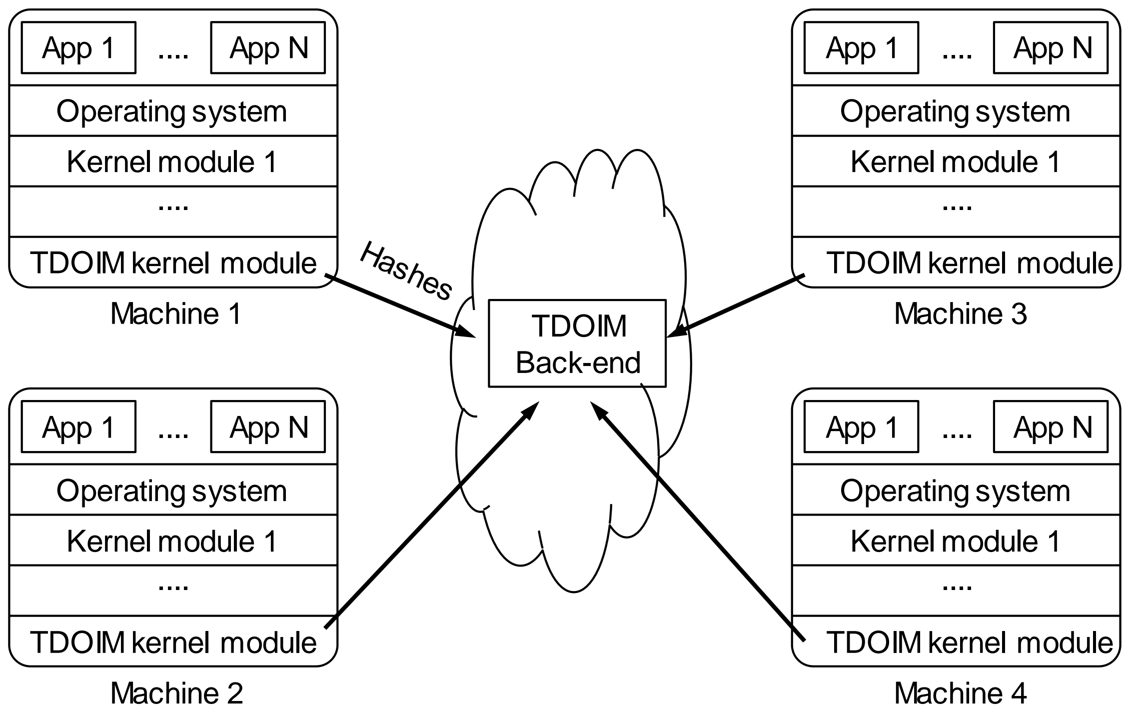

Figure 3.1. Example configuration of TDOIM monitoring four instances of a legacy application. .

Figure 3.1 gives a high-level overview of TDOIM's architecture, using an example setup of a legacy application that runs on four machines, which may be virtualized. Each of the four machines has installed a TDOIM client component, i.e., the TDOIM

agent. The TDOIM agent is a kernel-mode module that over a network connection sends at short random intervals to the TDOIM back-end server its machine's configuration. The configuration includes its processor name, OS version, compiler, and a hash of all code segments and all read-only data segments of all processes.

The TDOIM back-end is a user-mode application that resides outside the monitored system, e.g., on a remote host. Based on the configuration data received from clients, the TDOIM back-end divides the user systems into groups. Within a group, each member has the same combination of architecture, OS version, and compiler. Within each group, TDOIM compares the hashes of the various applications to detect outliers.

TDOIM takes steps to make this communication efficient. For example, by default a TDOIM agent only transmits a single hash (of all hashes) to its backend. Only if the back-end flags a hash as an outlier, the back-end requests more detailed hash values of the various applications to investigate why the hash of all hash values diverged.

TDOIM-like monitoring tools are susceptible to *scrubbing attacks*, in which a memory resident malware predicts monitoring time and attempts to conceal it's trace in memory at the time the monitoring agent computes a memory hash [27]. To address such attacks, TDOIM obtains its memory hashes in random intervals, which are by default 30–90 seconds. To further address the threat of scrubbing attacks and to reduce the TDOIM client's attack surface, TDOIM places its random interval generator on its back-end component.

3.3 Use-Case Scenarios

TDOIM has two main use-cases, which differ in the type of legacy applications monitored and their owners. (A) In the first use-case, an administrator is responsible

for a large number of instances of a legacy application, possibly distributed over several data centers. The admin may suspect that some of these instances are being attacked by malware. So the admin installs the client-side kernel agent on each machine, e.g., by remotely installing it as a device driver or other kernel extension. Upon installation the TDOIM agent will take periodic memory hashes and send them back to the TDOIM server component. The server component detects outliers, which may trigger the admin to either inspect the outliers more closely for possible malware infection or to shut down the outlier instances for further analysis.

The second use-case (B) differs in that an end-user may suspect a malware infection on her machine and install the TDOIM client module locally on her machine. TDOIM will then compare her hash values with those produced by other end-users running the same legacy application. This use-case has stronger privacy requirements for transmitting hashed memory regions to the server and a lower tolerance for false warnings, as the end-user can only inspect a single instance more closely for possible malware infection.

Both use-cases have in common that upon completion the TDOIM client component can be uninstalled from the affected client machines. Neither installation nor uninstallation require recompilation or restart of the monitored application.

3.4 Monitoring Partially Swapped-Out Memory Segments

When characterizing a program's code and read-only data segments as a hash value, it is important to determine which parts of the code and read-only data segment to include in the hash. Since at any time some pages of the program's segments may be swapped out to disk, the brute force solution would be to first swap all segments back into main memory and then compute the hash values. However this approach would be very inefficient, as each swap can be time intensive and displace pages other

processes may need, triggering further system slowdown. Furthermore, pages on disk are less interesting as they are currently not in use, by neither the application nor the rootkit.

To compute hash values efficiently, previous work such as SVV therefore just computes the hash value of those pages that are currently in physical memory [60]. However this approach may produce false positives, as on different machines different pages may be swapped out to disk. This results in different hash values even if the underlying memory contents have not been manipulated.

To distinguish swapped-out pages from manipulated pages efficiently, TDOIM uses the following multi-level hashing scheme. TDOIM first locates all the pages of a given program's code and read-only data sections that are currently in physical memory and therefore not swapped out. TDOIM then computes the hash of each of these pages and associates each hash value with the page's virtual address. Further TDOIM computes the hash of these hash values. In an initial comparison, the TDOIM back-end compares these "hash of hashes". If they diverge, the back-end compares all component hash values.

Figure 3.2 shows an example of hashing the N pages in virtual address range A to B. On the left, SVV computes a single hash of all the pages that are currently in physical memory. On the right, TDOIM first computes the hash of hashes Hash_T of the pages currently in physical memory. If the Hash_T values differ, TDOIM compares the hashes of all pages in physical memory (within A to B) across different systems. In addition to removing the false warnings of SVV-type approaches, TDOIM's hash of hashes technique can therefore also pinpoint the pages that have been manipulated, if any.

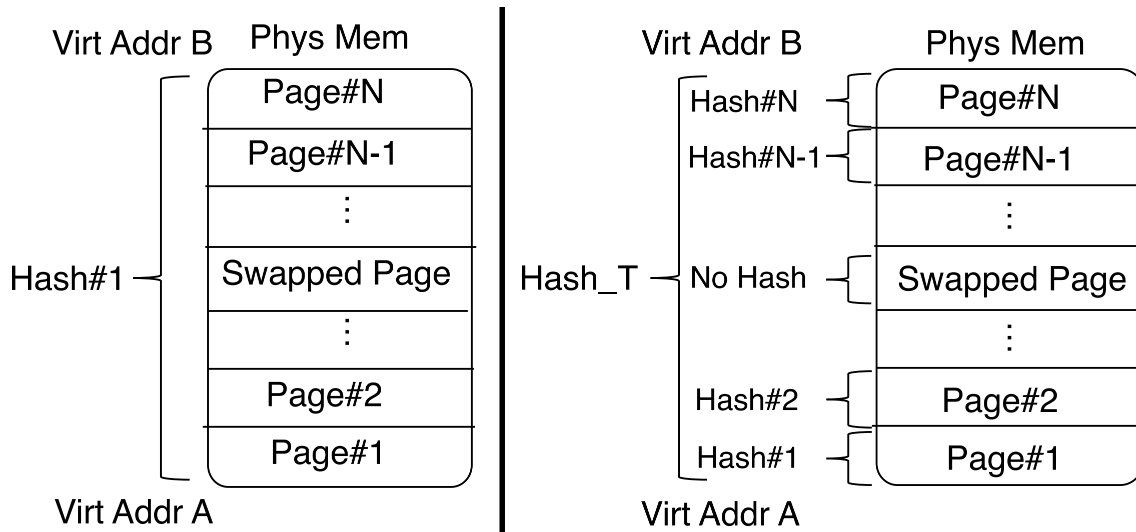

Figure 3.2. Single-hash of available pages (left) vs. hash of hashes (Hash_T) plus individual page hashes Hash#1 to Hash#N (right). .

3.5 Monitoring Dynamically Loaded Code

At any given time two instances of an application may differ in the libraries they have loaded. This may happen when the two instances solve different tasks, each of which may require its own libraries. So if I would map an entire application to a single hash value, this again would lead to false warnings, as different libraries lead to different hash values. For example, the two applications instances in Figure 3.3 have not been manipulated. But they yield different hash values (Hash_T1 $\neq$ Hash_T2), as only one of them has loaded the benign_lib2.so dynamic library.

While the Figure 3.3 problem could be resolved with the Section 3.4 two-level hashing technique, dynamically loaded code may also be placed at different virtual addresses across instances. For example, if one instance has loaded L1 before L2, but another instance has loaded L1 after L2, then the libraries are likely placed at different virtual addresses. In this case both the hash of hashes value and the page-level hashes of the two instances would differ, even though there is no rootkit manipulation.

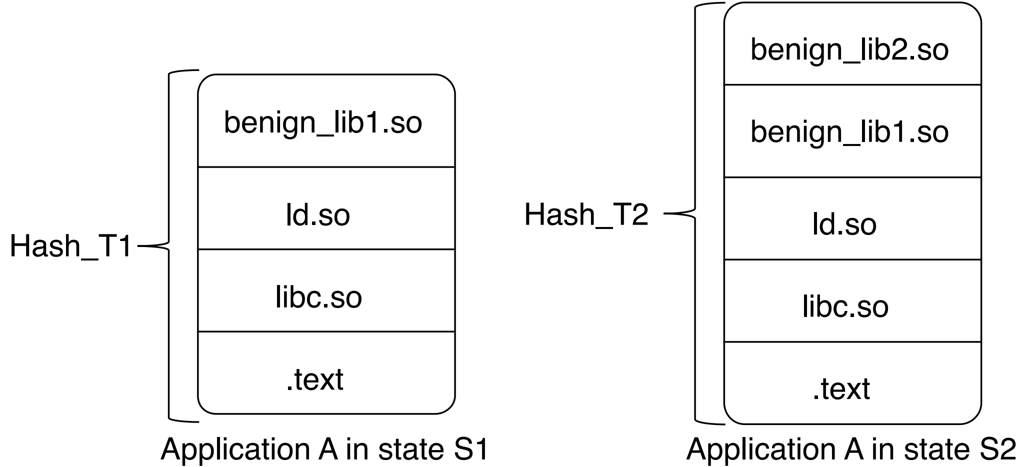

Figure 3.3. Two application instances yield different "hash of hashes" values (Hash_T1 $\neq$ Hash_T2), as only one has loaded the benign_lib2.so dynamic library..

To distinguish such cases from rootkit attacks, TDOIM associates each page hash with the name of the segment the page belongs to. TDOIM also computes one hash per segment. This way, TDOIM can quickly compare segments across machines and locate manipulated pages, even if different libraries are loaded or they have been loaded in different orders. For example, the two application instances in Figure 3.3 have different hash of hashes (Hash_T1 $\neq$ Hash_T2). TDOIM thus compares the segments' hashes (.text, libc.so, ld.so, benign_lib1.so, and benign_lib2.so) and identifies benign_lib2.so as an outlier.

Figure 3.3 is also an example of a false warning, since in this example benign_lib2.so is a benign library. However a real-world TDOIM setup has likely more instances than the two shown in Figure 3.3. Such a benign library is therefore typically running on a larger number of application instances. Since rootkits typically spread relatively slowly across machines, TDOIM has a time window in which a rootkit is only present on a relatively small number of machines. Thus TDOIM uses a configurable threshold value (by default 10%). An occurrence above the threshold

indicates a benign library, whereas a below-threshold occurrence indicates a rootkit infection.

3.6 TDOIM Agent's Attack Surface

The TDOIM client-side component is designed to have a minimal feature set and therefore a minimal attack surface. Specifically, TDOIM's client-side agent calls APIs and macros only to access and hash memory, manipulate strings, communicate via the network, and perform OS-level synchronization.

As a concrete example of TDOIM agent's small footprint, Appendix A lists all OS functions called by TDOIM's prototype implementation for recent versions of Linux.

3.7 TDOIM's Server Component

At a high level the server's main tasks are communicating with TDOIM's client-side agents and dealing with the hashes it receives from these clients. To keep the attack surface of TDOIM's clients as small as possible, the server also takes on such tasks as keeping track of all participating machines, initiating client-server communication, and deciding random communication intervals.

After sending a request to its clients, the server waits for a configurable duration for client responses (5 seconds by default). If the server only receives partial (or no) hashes from some clients, it sends another request to these systems. If a client does not respond to server requests for a long time (e.g., 10 requests), the server produces an alert. This alert indicates that the client is shut down, has a network problem, or is possibly under malware attack.

For each participating client, the server aims to maintain at least two hash values, the current and the previous one. So after the defined time interval has passed, the server compares the hashes it has received.

CHAPTER 4

TDOIM IMPLEMENTATION

To evaluate TDOIM, I have implemented its main components as prototypes for the Linux platform. This section describes key implementation details that allow an initial empirical evaluation.

The goal of this implementation is not to provide a highly efficient and secure large scale distributed software. So the prototype implementation is missing advanced features such as failure recovery, security and authentication, high availability, scalability and performance management. As an example, my implementation does not actively prevent denial of service attacks on the server component. Addressing these issues is part of future work.

4.1 TDOIM's Client-Side Agent

The TDOIM agent is a kernel module and therefore has full access to the user system's memory. When the agent starts executing it creates a single-threaded workqueue [61] that listens to a predefined port and operates based on the commands it receives from the TDOIM back-end.

4.2 Obtaining Virtual Address Ranges in the Agent

To obtain the virtual address range of Linux kernel code and read-only data, existing rootkit detection and protection tools consult the Linux kernel's symbol table [20, 62], which is stored in the *system.map* file. As usual, for each kernel

function and variable the symbol table maps between name and address. However this approach does not work if the kernel uses run-time address randomization [63, 64].

Instead, memory dumping tools traverse the I/O memory resources to find the kernel code's physical address range [65]. The Linux kernel keeps track of the I/O operations that occur within the address range of physical resources such as RAM. For the OS the RAM's child resources include kernel code, data, and uninitialized data sections (bss). However this approach does not work for the kernel's read-only data.

To obtain the virtual address range of Linux kernel code and read-only data, even under run-time address randomization, TDOIM calls the well-known *kallsyms* function. The kallsyms function extracts all symbols (e.g., functions and variables) from the kernel and is therefore commonly used by Linux debuggers. The resulting file maps between symbol address and name. For instance, a developer can use this file to extract the entry point address of an internal kernel function within the OS's virtual address space.

In most Linux distributions kallsyms is available. The TDOIM agent thus calls kallsyms to extract the address of the symbols that mark the start and end addresses of the kernel code and read-only data segments (`_stext`, `_etext`, `__start_rodata`, and `__end_rodata`).

Outside the kernel, TDOIM extracts code and read-only data address ranges by traversing the kernel's data structures in the kernel's heap. For example, the Linux kernel maintains linked lists of all processes (`task_struct`) and drivers (`module_struct`) and TDOIM interprets them to find the required virtual address ranges.

Specifically, TDOIM traverses the above heap structures to identify each segment or *virtual memory area* (VMA) of the kernel, each kernel module, and each user-mode process. Figure 4.1 shows an the memory layout of the APP1 example

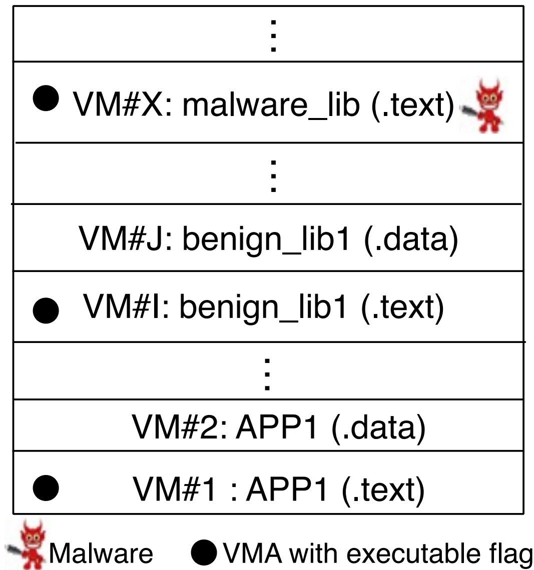

Figure 4.1. Address space of the APP1 example application, including the APP1 executable and several dynamically loaded libraries. .

process, which contains several VMAs. TDOIM checks if a VMA is flagged as executable, which indicates a code segment.

4.3 De-Relocating Virtual Addresses for Hashing

Kernel modules and shared libraries can be loaded in any order and therefore at different addresses in a virtual address space. Within such code the compiler thus has to express some addresses relative to an assumed base address. To address this issue, recent user-level libraries mostly use position independent code (PIC). PIC adds a level of indirection and replaces addresses in the code segment with a lookup of the actual virtual target address. The lookup is placed in the (writable) data segment and is therefore not hashed by TDOIM. If a library is compiled to PIC it will therefore yield the same TDOIM hash value regardless of where it is loaded.

Instead of position independent code, kernel modules and legacy user-level libraries use the traditional technique of load-time relocation. When loading such code the operating system fixes the code's base address and replaces ("relocates") each relative virtual address with the then-known absolute virtual target address. After relocation, the same library or kernel module may therefore contain different (absolute) virtual addresses on different machines, which would yield different hash values for the same piece of code.

A straightforward solution to this problem would be to find all such addresses in kernel modules and legacy shared libraries and replace them with zero. While this approach would ensure equal hash values for different relocations, it may also produce false negatives if a rootkit only manipulates such addresses.

To address this problem, TDOIM "de-relocates" addresses for hashing. For this task TDOIM leverages the segments' virtual address ranges obtained in Section 4.2. TDOIM replaces each absolute virtual address that points to a library with the corresponding relative virtual address. Specifically, for each absolute virtual address, TDOIM determines the library and memory segment the address points to and subtracts from the address the segment's base address, thus yielding the de-relocated address.

To find addresses in a library, TDOIM customizes the Distorm disassembler [66], which supports 32-bit and 64-bit Intel processors. Distorm provides convenient access to opcodes and operands via its decomposer feature. While Distorm contains some Linux header files that can only be used in user space it is written in POSIX C (without using OS-dependent APIs). I could thus remove user-mode specific header files from Distorm and use it in TDOIM's kernel agent.

4.4 Hashing Physical Memory

With the virtual addresses de-relocated, hashing the physical Linux kernel memory pages is straightforward as the kernel makes sure that its code and read-only data are in contiguous physical addresses and are never swapped out. So the TDOIM agent takes the virtual address ranges obtained in Section 4.2 and maps them to physical addresses via standard kernel APIs.

Outside the kernel, however, obtaining the physical addresses is more interesting. Regarding the addressing limit in X86, 32 bit architectures, not all memory-resident pages of user-mode and kernel-mode applications (HIGH_MEMORY pages) are mapped [61] into virtual address space. That means there is no virtual address dedicated for unused pages of applications. However, since these pages are not used by any application, these pages are maintained in *page cache* as candidates for swapping to disk for the cases in which the number of pages in physical memory exceeds a particular limit. To solve this issue and obtain the hashes of all pages belonging to an application, TDOIM reads those pages directly from the page cache.

The current implementation uses the MD5 hashing algorithm. MD5's trade-offs are well-known, it is fast but has a non-zero chance of hash collisions [67]. Picking instead a stronger hash-function such as SHA-2 or SHA-3 would lower the chance of such hash collisions.

4.5 Sending Hashes to the Back-end

Each TDOIM agent has a unique *TDOIM identifier*. In the current implementation this identifier is fixed and assigned before the TDOIM agent is installed. The TDOIM agent produces one message to send the hashes to the back-end application. Every message contains the user system's TDOIM identifier and the names of kernel,

kernel extensions, and applications, followed by their respective 16 byte MD5 hashes. TDOIM names the hashes of the kernel "kernel code" and "kernel data".

As mentioned earlier, TDOIM sends the hashes of loaded libraries or injected code separately from the code segments of user-mode applications. Thus, after including the hash of text segment of a process, TDOIM writes names and hashes of loaded libraries or injected codes.

4.6 TDOIM's Back-End Application

In the current prototype implementation, the back-end is a C++ user-mode application that contains STL data structures to maintain the information about each user system and the hashes of OS and applications. For each user machine that is booted and connected to the network, the back-end creates an object called MA_INFO. The MA_INFO contains a member as a dedicated name to user machine and the data structures that maintain the name and hashes of OS, user-mode and kernel-mode applications.

In the current prototype implementation all communication is performed via UDP. This is however just an implementation choice and could be replaced with other protocols. To communicate with clients, the server uses non-blocking sockets.

CHAPTER 5

TDOIM EVALUATION

In this chapter, I provide some research questions and conduct several experiments to evaluate the effectiveness of TDOIM approach for rootkit detection.

5.1 Research Questions

To evaluate TDOIM, I ask if TDOIM presents a promising path toward online rootkit detection in legacy applications. This question has two main facets, runtime overhead and true vs. false positives. While also interesting, true and false negatives are less relevant than true and false positives, since rootkit detection is heuristic in nature and TDOIM can be combined with other rootkit detection approaches. I therefore investigate the following three research questions (RQ), expectations (E), and hypotheses (H).

- RQ1: Does its runtime overhead preclude TDOIM from detecting rootkits online in legacy applications?
  - E1: I do not expect TDOIM to be applicable for all legacy application settings. A key reason is that TDOIM performs a relatively expensive analysis to compare kernel addresses across clients that may differ due to their code load order and their OS's address space layout randomization.
  - H1: TDOIM can be useful in legacy application settings in which client machines still have significant available computational resources.
- RQ2: Does its false positive rate preclude TDOIM from being used in production?

Table 5.1. Rootkit subjects run either in user (u) or kernel (k) mode; KV = Linux kernel version; CPU = 32 vs. 64 bit; Loc = main memory (m) vs. disk (d).

Rootkit		KV	CPU	Loc	Attack
LD_PRELOAD	u	3/2.6	32/64	m	Exchange libraries
Jynxkit	u	2.6	32	m	Exchange libraries
Patch dynamic loader	u	2.6	32	d	Inject code
Attach to process	u	2.6	32	m	Divert execution
Syscall hooking	k	3	64	m	Change kernel data
Suterusu	k	3	64	m	Change kernel code

  - E2: Since disassembly of x86 binaries is undecidable and likely to produce erroneous results, I cannot expect zero false positives.
  - H2: TDOIM's false positive rate is typically greater than zero but on average can remain below 10%.

- RQ3: Can TDOIM detect common types of kernel and user level rootkit attacks?
  - E3: I expect TDOIM to detect common types of kernel and user level rootkits online in legacy applications.
  - H3: TDOIM can detect common types of kernel and user level rootkits online in legacy applications.

5.2 Subjects: Kernel and User Level Rootkits

Table 5.1 lists the rootkits used in the experiments. They are a mix of third-party samples and my own development. The rootkits operate both in user and kernel mode and manipulate both disk contents and main memory. The rootkits perform a variety of attacks—they exchange libraries, inject code, divert execution, and change both kernel code and data. Following is a high-level description of each rootkit, together with how TDOIM is expected to detect it.

5.2.1 Exchanging Libraries via LD_PRELOAD

In this approach the rootkit exchanges the program's libraries with alternative, malicious libraries, for example, to divert the system calls issued by a user-mode application. To make a system call, the application would normally call a function in the C standard library libc. However this rootkit can divert some or all such function calls to a malicious library that is loaded in the application's process.

To divert function calls, the rootkit changes the order in which a process loads libraries. To change the load order, the rootkit sets the LD_PRELOAD environment variable of a new process. Linux loads the libraries listed in LD_PRELOAD before all other libraries, even before libc. As an example implementation of this rootkit, in this experiment I set LD_PRELOAD to divert the execution of the open file system call to a malicious open file function implemented in my injected library. I expect TDOIM to detect this rootkit by comparing the hash code of two program instances, which will differ due to the different loaded libraries.

I evaluate the effectiveness of TDOIM for rootkit detection using the following scenarios. First, the open file API is replaced by a new open file function which returns NULL when it is called by the standard Linux (user-level) cat application. Second, if a file scanner application (e.g, an anti-virus) attempts to open a malicious file which malware has dropped in a victim system, the new fake open file function returns NULL. Otherwise, it invokes the original open file in libc. Therefore, malware can prevent the file scanner from opening the malicious file for scanning.

5.2.2 Jynxkit: Exchanging libraries via ld.so.preload

In addition to the process-specific LD_PRELOAD environment variable of Section 5.2.1, Linux also checks the contents of the /etc/ld.so.preload file for user-level

libraries that should be loaded before libc. Manipulating this file therefore changes the library load order of all future user-mode processes.

In this experiment I run the third-party Jynxkit rootkit. Jynxkit adds its malicious ld_poison.so library to ld.so.preload and thereby diverts several system calls including open, opendir, fdopendir, readdir, unlink, _xstat, _fxstat, and _lxstat. Jynxkit remains undetected by several common anti-rootkit tools, because they only detect library injection attacks via LD_PRELOAD.

5.2.3 Patching the Standard User-Mode Program Loader

To influence how the OS starts all future user-mode program executions, the ld.so.preload approach of Section 5.2.2 leaves the OS binaries intact. But a rootkit can also directly rewrite on disk the binary file of the OS's standard ld-linux.so dynamic user-mode program loader.

A proof-of-concept implementation of this approach is available [68]. This rootkit adds malicious shell-code to the loader binary on disk. This patched loader sets the entry point of a to-be-loaded program to some malicious code. The loader also adds code that after the malicious code execution transfers execution to the program's originally intended benign entry point. I obtained the rootkit's source code and use it to inject code into the cat application.

5.2.4 InjectSO: Diverting Process Execution

The previous attacks of Sections 5.2.1 to 5.2.3 manipulated future processes. Instead, this attack uses an OS's standard support for debuggers to attach to a process that is already executing and divert the processes's execution [69]. This type of attack typically diverts execution to the OS's loader and dynamically loads a

malicious library. The attack then changes some of the program's function pointers to divert function calls to the just injected malicious library.

Since Linux does not provide much library support for this kind of rootkit, implementing this attack is harder on Linux than on Windows. However I obtained the source code of the InjectSO rootkit prototype implementation. I thus injected into the address space of a running standard Linux cat application a malicious library that prints a message in the terminal.

In this type of attack, TDOIM notices that the cat instances running on different machines produce different hash values. The different hash values point to cat's code segment, which differs across machines due to the rootkit.

5.2.5 Hooking the System Call Table

A common rootkit technique first locates kernel data structures such as the system call table that point to important system functions. By changing the pointers, attackers can divert or "hijack" system calls to malicious code. In other words, these rootkits hijack the kernel execution without changing any kernel code.

In this experiment I used the implementation of the Linux syscall hooker [70] to divert system calls. For this rootkit I expect TDOIM to detect a difference in the kernels' read-only data segments.

5.2.6 In-line Function Patching with Suterusu

While the Section 5.2.5 rootkit changes kernel control-flow without changing any code, a rootkit can of course also directly change the kernel code to change its control-flow. For example, in in-line patching a rootkit may overwrite a function's prologue with a jump to malicious code and thereby divert kernel execution [39].

I evaluate the effectiveness of TDOIM against the proof-of-concept Linux kernel in-line patching rootkit Suterusu [71]. I expect TDOIM to notice a changed hash value of the code segment that contains the patched function.

5.3 Experiments

To explore the research questions I conducted a few small experiments. While limited, these small experiments provide interesting preliminary insights into the research questions. Specifically, I set up a testbed of 20 user systems and installed a TDOIM agent in each user system. These 20 user systems communicated with one TDOIM server instance.

For the scope of this dissertation I assume high-speed connections between server and clients, as it is typically found within a local machine. While this choice does not capture many legacy application installation scenarios, it is still valuable as a baseline for future experiments and approximates the setup of some legacy applications.

I further focus the experiments on a baseline via the following experimental choices. First, I run the same version of each application and each kernel module on each client. Second, for these experiments I do not explore situations in which applications dynamically load a large number of libraries during the experiments. On the other hand I do not prevent applications from loading libraries.

All components were hosted on the same physical machine. Each user system ran on an Ubuntu Linux VMWare VM with kernel versions 2.6 or 3, 32-bit or 64-bit Intel processor, and 515 MB RAM. The TDOIM server ran on a 64-bit Ubuntu VMWare VM with a version 3 kernel and 4 GB RAM. The host system was Debian Linux running on a 2.33 GHz Xeon machine with 32 GB RAM.

Table 5.2. Breakdown of TDOIM's average runtime overhead; Loc = TDOIM client (c) vs. server (s).

TDOIM activity	Target	Loc	Slowdown (%)
Hash	user-mode app	c	8
Hash	kernel code/data	c	3
Hash + de-relocate	kernel modules	c	20
Compare hashes	20 VMs	s	15
Log MA_INFO	20 VMs	s	42

The server and each client used their Ubuntu OS in its default installation and configuration. This meant that runnning in each OS were 100 user-mode applications and 41 kernel modules.

For each rootkit detection experiment, I divided the 20 user systems into two groups. Within each group, each user system has an identical configuration, i.e., the same hardware and OS configuration. For each experiment I further infected between one and three VMs with a given rootkit.

5.3.1 RQ1: TDOIM's Runtime Overhead

To evaluate the performance of the TDOIM agent, I measure how much time it takes to compute the page hashes of user-mode applications and kernel data and code. For kernel modules, I measure the overhead of page hashing and de-relocation.

To evaluate the performance of the TDOIM back-end application, I calculate the overhead of comparing hashes of the kernel and 141 applications (100 user-mode applications and 41 kernel modules) over 20 virtual machines. Also, I customize the back-end application to produce the log of hashes for each virtual machine and I measure the overhead of the log producing task.

Table 5.2 summarizes the measurements. The table contains two kinds of slowdown numbers. The server (s) slowdown measurements represent a given task's slow-

down of the TDOIM server component. The client (c) measurements are slowdown of the monitored application.

As a baseline for the server measurement, the biggest overhead (42% slowdown) was logging the hashes received from the clients to disk. This task is only turned on when debugging the TDOIM server component. Comparing the hash values resulted in a 15% slowdown. On the client side, the sum of all TDOIM tasks lead to a slowdown of 31% of the monitored client application. While this is a non-negligible slowdown, it is still feasible for many application scenarios. The TDOIM prototype implementation could also be further optimized, possibly leading to smaller slowdown numbers.

5.3.2 RQ2: TDOIM's False Positives

To explore research questions RQ2 and RQ3 I carefully analyzed each TDOIM-produced alert. Specifically, I dumped the pages that yielded different hash values on different clients. I then manually analyzed the pages to determine if different hash values were caused by a rootkit.

A false positive occurs when TDOIM produces an alert about different hash values from two or more client agents that are not caused by a rootkit manipulation. Such false positives could occur for different reasons, such as errors in my TDOIM prototype implementation. For the experiments I expect these false positives to stem from the insufficient reverse engineering of memory addresses in client side kernel modules.

Indeed, during the experiments all of TDOIM's false positives came from kernel modules. Recall that kernel modules are relatively challenging to handle due to the OS's address space layout randomization and possibly different module load order.

Table 5.3. The 7 kernel modules that produced false alerts and their number of pages; FH = false hashes; FP = false positives.

Kernel module	Pages	FH	FP (%)
e1000	22	3	13.6
vmwgfx	18	2	11.1
ttm	11	1	9.0
drm	33	1	3.0
bluetooth	55	3	5.4
rfcomm	9	1	11.1
psmouse	16	2	12.5
All 41 modules	314	13	4.1

Of the 41 kernel modules running in the standard 64 bit Ubuntu 12.04 LTS installation, I received in the experiments false positives on 7 kernel modules. These false positives are summarized in Table 5.3. For example, the e1000 kernel uses 22 pages for its code and read-only data sections. The size of each page is 4kB. TDOIM received conflicting hashes and thus false warnings on 3 of these 22 pages, which corresponds to a 13.6% ratio. Across all 314 pages of the 41 kernel modules, false warnings occurred in 13 pages or 4.1% of pages.

5.3.3 RQ3: TDOIM's Rootkit Detection Performance

Besides false positives, the experiments also produced many true positives. That is, in each experiment, TDOIM successfully pinpointed the VM, application, and page that were infected by a rootkit, both at the user and kernel level.

CHAPTER 6

TDOIM LIMITATIONS AND DISCUSSION

In this chapter I examine the limits of my current threat model (and therefore of TDOIM) and discuss how these limits could be addressed in future work.

6.1 Stack: Return-Oriented Programming (ROP)

Not covered by the threat model are stack manipulations. A well-known example stack manipulation is return-oriented programming (ROP), in which attackers hijack the application control flow without code injection [72, 73]. A ROP attack instead overwrites an application's stack.

Detecting ROP attacks with TDOIM would require monitoring and comparing the execution control-flow of applications across different machines, which is part of the future work. Moreover, several other approaches mitigate ROP attacks by monitoring the control flow or via compiler-level approaches for building less vulnerable binaries [74, 75].

6.2 Kernel Heap: Attacks on Dynamic Data Structures

Also not covered by the threat model are manipulations of dynamic data structures in the kernel's heap such as direct kernel object manipulation (DKOM). The most common DKOM attack is manipulation of the linked lists the kernel uses to keep track of kernel modules and processes, to hide the presence of malware in the victim system.

While TDOIM does not detect such heap manipulations, many real-world attacks use DKOM only to hide their presence. In addition, such attacks rely on rootkits covered by the threat model to attack applications. TDOIM can thus detect such combined attacks, even if their traces are hidden via DKOM.

6.3 Potential Attacks on TDOIM

Similar to related tools, if malware is aware of the TDOIM agent on the victim systems, malware can attack the TDOIM agent or intercept its network communication with the back-end. Standard approaches could be added to protect TDOIM from such attacks such as binary obfuscation, hiding the TDOIM agent, or hiding its network communication.

On user systems that have TPM, I could also extend TDOIM to run securely even during a malware attack. Such an extension could be built using Flicker [76].

CHAPTER 7

MIXED-MODE MALWARE AND SEMU ANALYSIS REQUIREMENTS

At a high level, mixed-mode malware runs in two phases. (1) In phase one, a (typically kernel-mode) malware component modifies a part of the OS kernel, i.e., kernel code, kernel data, or both. (2) In phase two, a (typically user-mode) malware component executes the main malicious payload. There are three key ideas behind these phases.

The first key idea is that the payload reads modified kernel data or invokes modified kernel code. The semantics of the executed payload is thus determined by the success of the phase 1 kernel manipulation attempt.

The second key idea is that a malware analysis can only observe the main malicious payload behavior if the phase 1 kernel modification attempt succeeds. If the phase 1 attempt did not succeed, then the phase-2 payload may amount to benign behavior or a different malicious behavior.

The third key idea is that the phase-1 kernel modification *may* also lead current malware analysis tools to incomplete or inaccurate analysis results. Mixed-mode malware thereby effectively prevents malware analysis with current tools.

As other malware, mixed-mode malware has various implementation options. Phase-1 can be carried out by a kernel-mode component that has been deployed by either a user-mode component or by a user-mode dropper application. But it may also be triggered from a user-mode component via a zero-day kernel exploit which manipulates kernel structures. Similarly, phase 2 is typically carried out by a user-

mode component, as it is more convenient for malware writers. But phase 2 could also be implemented by a kernel-mode component.

More formally, the state of the kernel at time T consists of code C_T and data D_T. Phase 1 performs operation M to manipulate the original kernel code C_O and data D_O, yielding C_N and D_N, or $M(C_O, D_O) = (C_N, D_N)$. If the manipulation succeeds then C_N is the manipulated code C_M and D_N is the manipulated data D_M. In the new system state, phase 2 executes a sequence of operations U, which invoke C_N and read D_N. The main malicious payload behavior, operation P, is the sequence of commands executed with successfully manipulated kernel data and code, $U(C_M, D_M)$.

$$U(C_N, D_N) = \begin{cases} \text{operation P} & \text{if } C_N = C_M \wedge D_N = D_M \\ \text{operation X} & \text{otherwise; } X \neq P \end{cases}$$

Optionally, operation M may also attack an inside-the-guest VMI component of a malware analysis A. In this case the analysis produces an incomplete, incorrect, or misleading report if M succeeded. Since the malicious payload is only observable if M succeeds, analysis A is not effective for analyzing such mixed-mode malware.

In the following I describe concrete mixed-mode malware examples that cannot be analyzed by current malware analyses. We present these examples in a minimal style for ease of exposition. The examples are minimal in the sense that each example exploits one weakness of one kind of current malware analysis technique. However these examples could be combined into integrated, comprehensive malware attacks that cannot be fully analyzed by several or all current malware analysis techniques.

7.1 Misleading User-Only Analysis

This section describes how mixed-mode malware can mislead user-only analysis, regardless of where the analysis performs VMI. These techniques thus affect both in-guest VMI such as Anubis and outside-the-guest VMI such as Ether. The high-level

idea is to modify those guest OS kernel entities in phase 1 that the phase 2 malicious payload uses. Since kernel modifications are outside the scope of a user-focused analysis, the analysis misses the true semantics of the malicious payload.

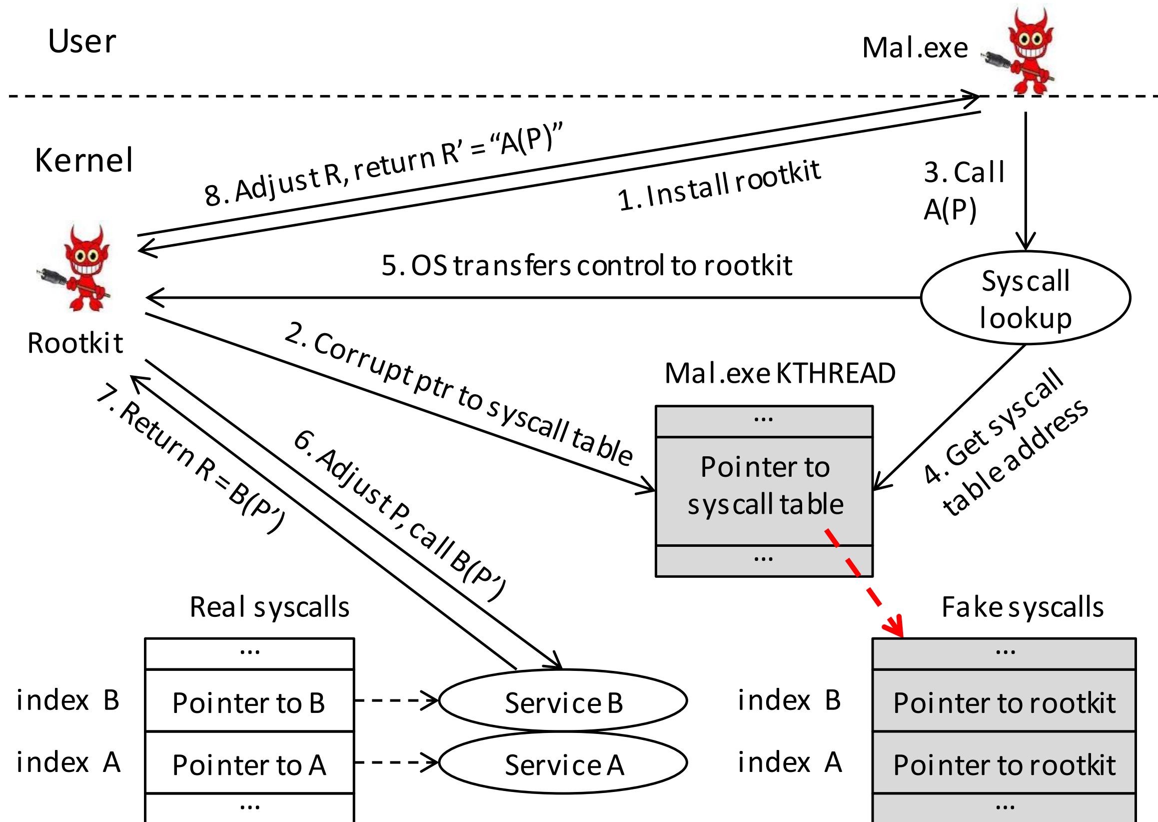

Figure 7.1. Example malware misleading Ether's user-only VMI; box = data; oval = function; solid arrow = attack; dashed arrow = original pointer; bold dashed arrow = manipulated pointer; gray = corrupted; P = system call A parameter values; P' = system call B parameter values. .

Figure 7.1 shows the main steps of misleading user-only analyses such as Anubis or Ether. (1) First the user-mode malware Mal.exe installs a rootkit (a kernel-mode component). The rootkit creates a fake system call table that contains pointers to itself instead of to the standard operating system services. (2) Then the rootkit

overwrites the pointer (within the `KTHREAD` object of Mal.exe) the operating system uses to find the address of the system call table with the address of the fake system call table.

(3) Each time Mal.exe calls system call A the operating system will now (4) follow the pointer to the fake system call table. (5) The OS thereby directs the control-flow to the rootkit instead of the original OS service. (6) The rootkit adjusts the input parameters to system call B and invokes the corresponding OS services. (7) After executing the invoked system service, the rootkit adjusts the return values to the original system call and (8) returns the results to the user-mode malware component.

Since a user-only analysis logs system calls at the interface between user- and kernel-mode, it will log the system calls as they are issued by Mal.exe. If the kernel modification succeeds, the analysis will log a sequence of system calls that differs from the actually executed system calls.

If the kernel modification does not succeed then an analysis tool cannot observe the main malicious payload, as the system call table will not point to the malware-created one. The commands executed by Mal.exe thus lead to system calls that differ from the malicious payload.

7.2 Misleading Kernel-Only Analysis

This section describes how a mixed-mode malware sample cannot be fully analyzed by a kernel-only analysis, regardless of where it performs VMI. The malware sample thus affects both in-guest and outside-the-guest VMI. We are not aware of any current outside-the-guest VMI kernel-only analysis systems. But dAnubis is a well-known in-guest VMI kernel-only system [77].

The Stuxnet example malware does not load any malicious kernel-level code and is thus not tracked by a kernel-level analysis tool such as dAnubis. Instead, Stuxnet

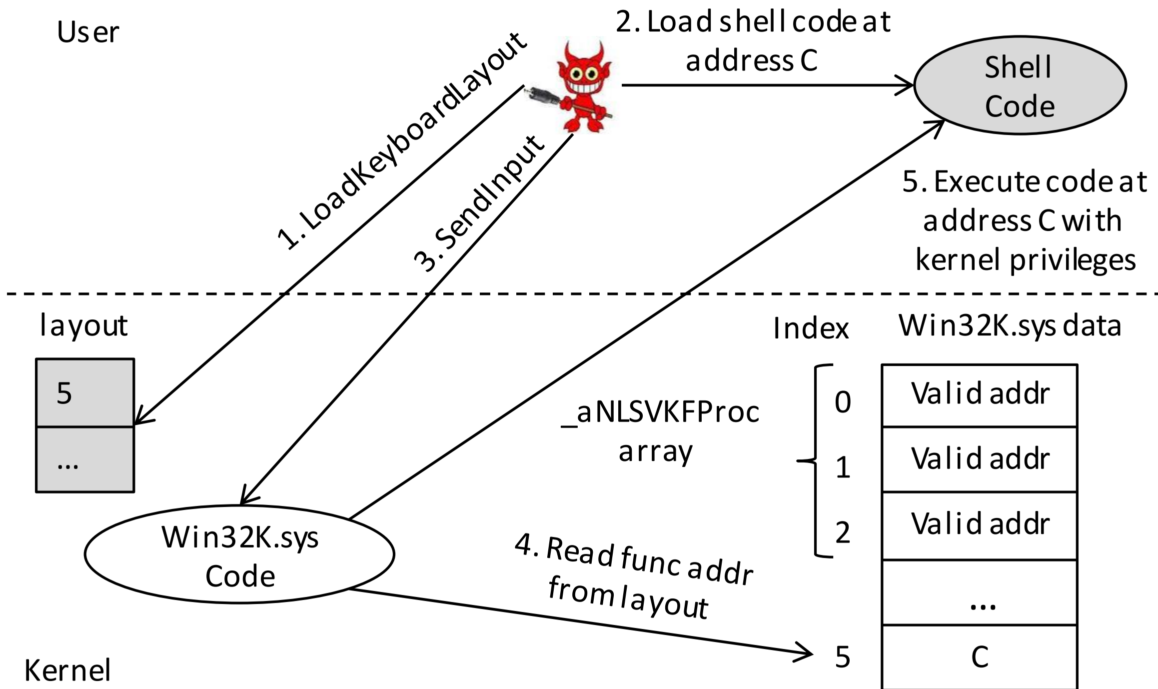

Figure 7.2. Stuxnet kernel exploit; box = data; oval = function; white = original; gray = corrupted..

runs in user-mode. To obtain administrator access to the victim system, it executes some instructions (shellcode) with kernel privilege. This attack uses a vulnerability in the Win32k.sys driver of Windows XP and Windows 2000. Win32K.sys is the driver that manages the graphical user interface environment, e.g., by dispatching keyboard and mouse inputs to applications. A Windows user-mode application can create a custom keyboard layout file and activate it via the LoadKeyboardLayout API.

To hijack the kernel execution, Stuxnet passes a malformed keyboard layout file as an input argument to a system call. Stuxnet exploits that the kernel does not check the bounds of an array index that is provided in an user-generated keyboard layout file.

For finding the virtual key that corresponds to a keystroke, Win32k.sys obtains an index into an array from the active keyboard layout file. Win32k.sys gets the address at the given index and executes the function at this address. The attack crafts a keyboard layout file that contains an out-of-bounds index such as 5, which points to C. When interpreted as a function pointer, C points within the range of the malware's user-mode address space, which effectively yields control to the malware.

Figure 7.2 summarizes the main steps. (1) The user-mode malware loads a crafted keyboard layout file by invoking LoadKeyboardLayout. (2) Then it allocates memory at address C and loads a malicious shellcode into this address. (3) To trigger the vulnerability, it invokes the SendInput API, which synthesizes keystrokes and triggers Win32k.sys to use the crafted layout file for extracting the virtual key for the requested keystrokes. (4) Therefore Win32k.sys reads the value C as the function address and (5) executes the malicious shellcode at location C with kernel privilege.

To summarize, if the kernel modification succeeds the malware executes code at C with kernel privileges, untracked by dAnubis. If the kernel modification does not succeed to provide administrator access for malware code, malware cannot execute its malicious payload.

7.3 Misleading Inside-the-guest VMI

This section describes how mixed-mode malware can defeat inside-the-guest VMI, regardless of the scope of the malware analysis. This approach can thus mislead user-only techniques such as Anubis, kernel-only techniques such as dAnubis, and whole-system techniques such as those based on TEMU.

Figure 7.3 shows an example malware that can mislead in-guest VMI techniques such as TEMU's Module Notifier VMI driver. In Figure 7.3 the main malicious payload is to call service A with certain parameters (step 3). The malware Mal.exe

triggers this payload only if the previous kernel manipulation attempt in step 2 succeeds. But step 2 also breaks TEMU's VMI scheme and thereby renders TEMU ineffective.

An analyst therefore has two options. She can allow step 2 to trigger the malicious payload, but this renders TEMU ineffective. The other option is to prevent the step 2 manipulation, but this prevents the main malicious payload from executing.

In more detail, Dropper.exe first drops the "Function Modifier" rootkit (step 1). This rootkit manipulates TEMU's VMI infrastructure (step 2). The rootkit checks the system call table and internal OS functions and unhooks and unpatches the functions TEMU uses. Then the rootkit manipulates these functions so they return false results to TEMU's VMI driver.

Specifically, the rootkit unhooks the internal OS function `MmLoadSystemImage` (Mm1), which TEMU hooks to detect driver loading by malware. The rootkit also unhooks the OS function `ZwQuerySystemInformation` (Zw1) of the system call table, which TEMU uses to retrieve driver module information. The rootkit then hooks Zw1 to point to an alternative malicious implementation Zw1'.

In step 3, the Dropper creates a new process for Mal.exe, which calls Zw1. If the phase-1 manipulation of step 2 does not succeed, then this results in a notification to TEMU, but no call to service A. If step 2 succeeds, then this call by Mal.exe results in the malicious payload and misleading TEMU's VMI.

7.4 Analysis Requirements

From the threat model and example mixed-mode malware attacks I can infer the following three requirements for effective analysis of mixed-mode malware. (1) The analysis must run outside the malware scope. For mixed-mode malware this means that the malware analysis cannot have any component such as VMI in the guest OS.

(2) A precise and comprehensive model of both kernel data and kernel code. This is a common malware analysis requirement. Without such a model an analysis tool cannot generate a precise and comprehensive log of potentially malicious activities.
(3) Log potentially malicious activities regardless of where they occur. This means monitoring and logging both kernel-mode and user-mode activities. Such logging requires identifying which instructions in user- and kernel-mode are run on behalf a malware component, for example, when the OS performs a requested system call.

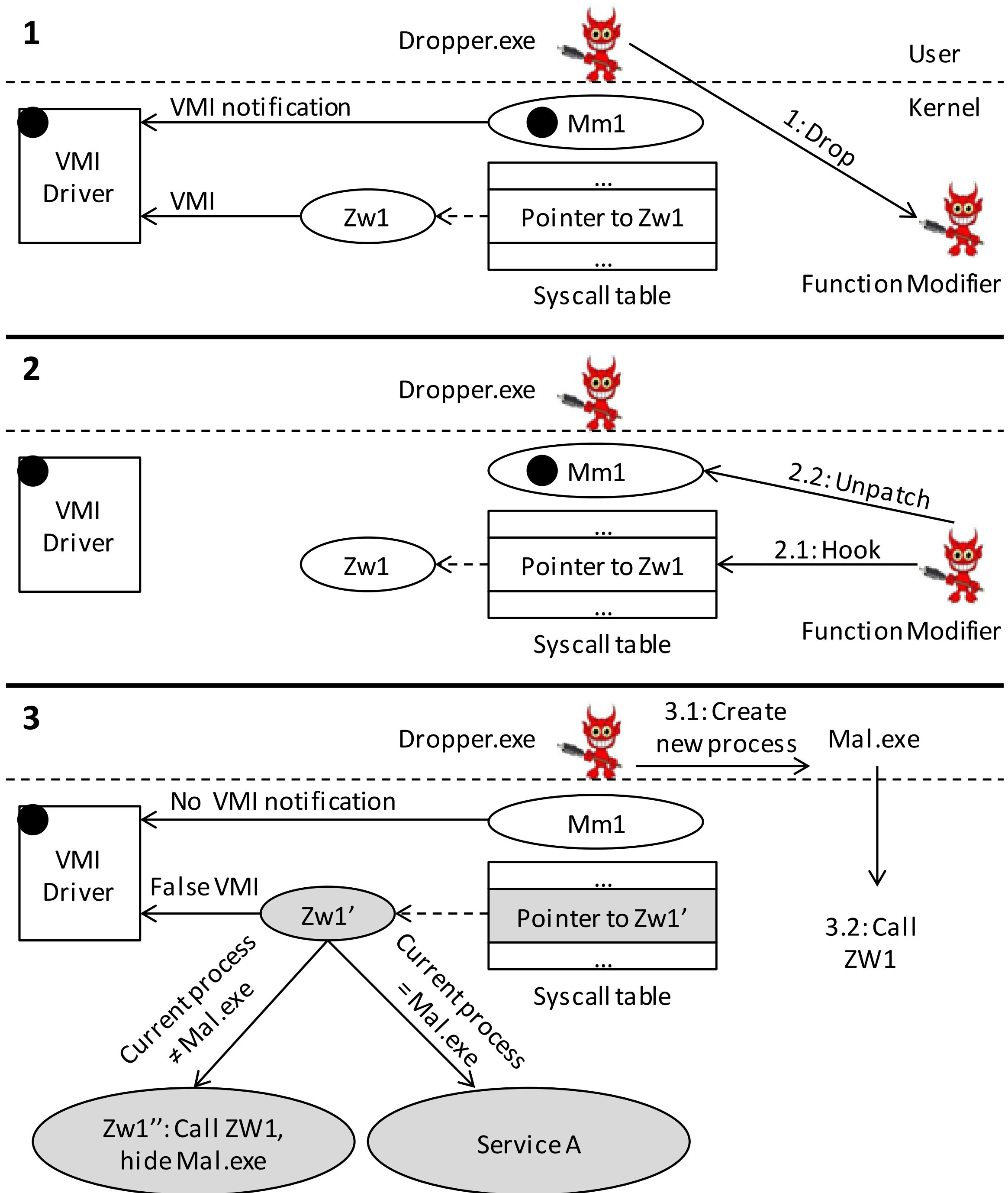

Figure 7.3. Example malware payload (calling A) that misleads TEMU; box = data; oval = function; dashed arrow = pointer; black dot = VMI entity; gray = corrupted.
.

CHAPTER 8

SEMU APPROACH

This chapter describes SEMU's high-level design concepts. SEMU follows earlier approaches such as TEMU in that it uses virtualization. However SEMU places all analysis components outside the guest OS and its analysis covers both user-mode and kernel-mode malware codes. SEMU's concepts can be implemented in various ways. To evaluate SEMU I implemented it on top of QEMU. However the concepts are general and could be re-implemented using other virtualization techniques, such as various software-based virtual machines or via hardware virtualization extensions.

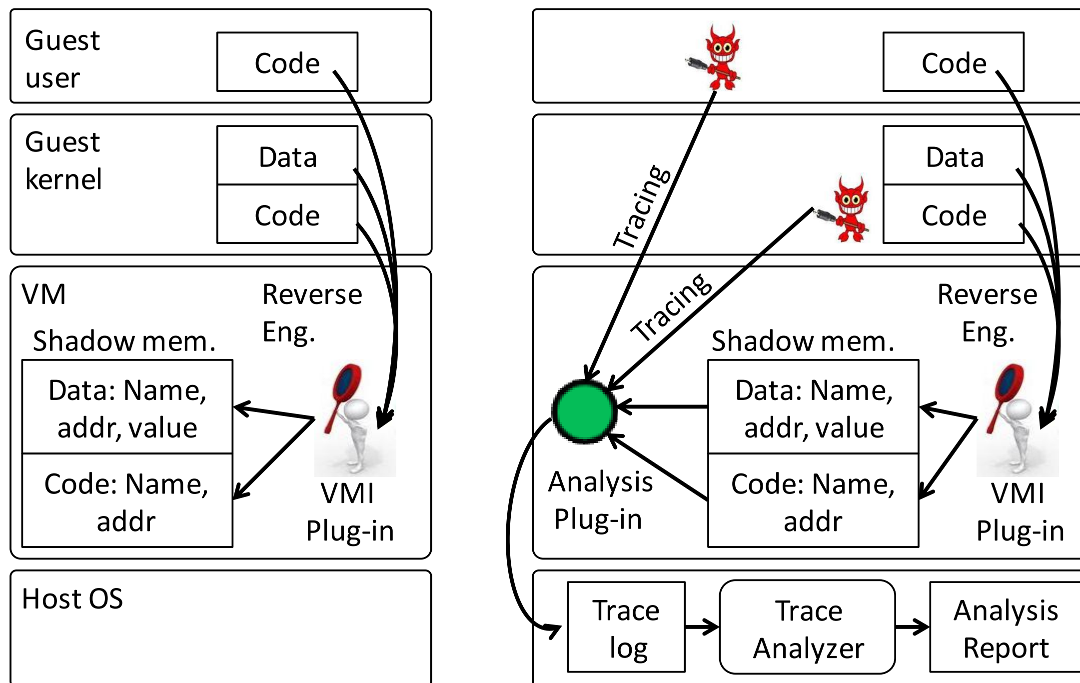

Figure 8.1. SEMU architecture and main execution phases: Pre-malware-execution phase (left) and the malware execution and post-execution log-analysis phases (right).
.

Figure 8.1 gives a high-level overview of SEMU's architecture and main execution phases. SEMU consists of the following four major components. (1) First, SEMU has a reverse-engineered model of the guest OS. (2) Second, SEMU's pre-execution phase copies key OS level elements from the guest OS into SEMU's *shadow memory* (Figure 8.1 left). The right part of Figure 8.1 shows the remaining components. (3) Malware execution and monitoring keeps SEMU's shadow memory in sync with the guest OS and creates a precise log of malware activities. (4) The last component is a post-execution log-analysis.

SEMU logs and keeps track of both user and kernel events. These events include system calls, I/O control (IOCTL), calls to functions exported by dynamic-link libraries (DLLs), and kernel-mode function calls. Such a comprehensive log captures information flow across the kernel-user divide, such as user-mode code calling a system call and kernel code branching into user-space code.

Before the VM executes a guest instruction, SEMU decides whether to log the instruction. When making this decision, SEMU consults a precise and comprehensive memory model of the guest OS. This model is SEMU's shadow memory, which includes address ranges of code and data as well as individual data values.

SEMU builds and maintains its shadow memory from outside the guest OS and thus outside malware's reach. SEMU builds its initial shadow memory version by traversing the guest OS memory before malware execution. SEMU interprets the guest OS memory by consulting its *reverse-engineered model of the Windows OS* that includes symbol information (e.g., the layout of all kernel data structures) and the addresses of kernel functions. During analysis SEMU keeps its shadow memory updated by keeping track of events in the guest OS that change the kernel data. Such events include the creation and deletion of kernel objects and writes of kernel object fields.

8.1 Reverse-engineered OS Model: PDB

SEMU performs VMI by interpreting the guest OS memory from outside the guest. To parse the current guest OS memory, SEMU leverages the memory layout information provided in PDB (program database) files [78]. SEMU uses this approach because PDB data cover a wide range of kernel data structures in a wide range of Windows operating systems. While there has been promising recent work on synthesizing outside-the-guest VMI tools automatically [41, 62, 79], these synthesized VMI tools do not cover all the kernel data structures that are needed for malware analysis [41, Section 3A]. In future work I plan to explore integrating such synthesized VMI tools into the PDB-based approach.

To extract the precise behavior of the malware, SEMU needs to know where important member fields of OS objects and data structures are. For instance, to detect the Figure 7.1 malware attempt of changing the system call table, SEMU has to detect a manipulation of the system call table pointer within the `KTHREAD` object. SEMU thus tracks the manipulation of `KTHREAD` objects including field writes. These data structure layouts are documented as PDB symbols, together with the name and offset address of internal kernel-mode functions of the OS and drivers.

The format of data structures and other symbols differs across Windows versions. To build an accurate model, the VMI plug-in first resolves the guest OS and device driver version numbers. Then SEMU downloads the corresponding PDB symbols from Microsoft servers.

SEMU differs from the OS reverse engineering of current tools such as Volatility [1] and Virt-ICE [80]. Volatility is an off-line forensic analysis tool that does not monitor or log malware actions in kernel- and user-mode. Virt-ICE is an interac-

[1]http://code.google.com/p/volatility/

tive debugger but does not monitor kernel manipulations and thus it is not effective against mixed-mode malware.

8.2 Pre-Execution: Create Shadow Memory

Directly before malware execution starts, SEMU initializes its shadow memory by copying guest OS code information and data into its shadow memory (Figure 8.1 left). SEMU performs this pre-execution phase before every execution of the malware, as the guest OS state may change between subsequent malware executions.

Algorithm 1: Main steps of the pre-execution phase.

```
1 On_init_event()
2 begin
3   trusted_code = phyAddr(kBase, kPE, drvBase, drvPE);
4   fMap = resolve(PDB, kBase, kPE, drvBase, drvPE);
5   dMap = resolve(kpcr_pointer);
6   current_proc = get_cr3_from_kprocess(kpcr_pointer);
7 end
```

Algorithm 1 summarizes the pre-execution phase. The algorithm basically initializes the following four key data structures. First, SEMU infers the address range of each trusted kernel code component and stores it in *trusted_code*. This includes ntoskrnl.exe, Win32k.sys, and other basic device drivers such as tcpip.sys and disk.sys. Subsequent phases use these address ranges to distinguish trusted from non-trusted kernel code. This step is important as monolithic operating systems such as Windows and Linux operate large amounts of code and drivers in kernel-mode, without address-space separation to isolate the kernel from possibly malicious code.

Second, SEMU creates *fMap*, a detailed list of functions within each trusted code component. For each function, fMap contains its name and its entry point address and name of the trusted code it belongs to.

Third, SEMU creates *dMap*, a detailed structure of key OS objects. dMap contains the name, address range and field values of many OS-level objects. Finally, SEMU stores the set of process objects also in *current_proc*.

SEMU retrieves these guest OS data by traversing the large number of OS objects that are reachable through the x86/x64 segmentation registers `FS` (x86), `GS` (x64). SEMU interprets the guest OS memory using its OS model reverse-engineered from PDB symbols. When in kernel-mode, the `FS/GS` register points to a kernel-mode data structure called kernel processor control region (`KPCR`). `KPCR` gives access to base addresses and PE[2] information of both kernel components (kBase, kPE) and drivers (drvBase, drvPE).

Via `KPCR`, SEMU retrieves information about both dynamic and static kernel addresses. A static address does not change during normal OS execution. Example static addresses include the interrupt table `IDT`, the system call table `SSDT`, and the global descriptor table `GDT`. Dynamic addresses may change during normal OS execution. Examples include OS process objects in the OS heap managed by the kernel's object manager such as `ETHREAD` and `EPROCESS` [78].

A special case of dynamic objects is the list of current processes current_proc. Via `KPCR`, SEMU extracts the base address of the page directory for each running process from the `KPROCESS` object's `DirectoryTableBase` field. Subsequent phases use this process list to track malware processes.

When executing an instruction in user-mode, the `FS/GS` register points to a thread environment block (`TEB`) user-mode data structure. A `TEB` contains informa-

[2]http://msdn.microsoft.com/en-us/magazine/cc301805.aspx

tion about the currently running thread and points to a process environment block (PEB). SEMU uses the PEB to resolve user-level information about the current process.

PEB also indirectly points to the InloadOrderModuleList list of the loaded dynamic-link libraries (DLLs) within a process memory. SEMU identifies each of these DLLs by the LDR_MODULE, which includes information about its corresponding DLL such as the base (start) address of a DLL in the process address space. By using the base address of a DLL and extracting the offsets of its functions from the export table in the PE header, SEMU finds the entry point addresses of library functions in the process memory.

8.3 Whole-System Malware Analysis

When the malware executes, SEMU monitors and logs key events in both user- and kernel-mode. That is, unlike the user-only tracing common in current tools such as TEMU, Anubis, CWsandbox, and Ether, SEMU monitors and logs control flow in the kernel whenever the processor switches to the kernel to serve a malware's request. SEMU thereby discovers system call swapping attacks such as Figure 7.1.

To perform whole-system malware analysis the analysis plug-in distinguishes user-mode from kernel-mode malware code. Address-space separation in user-mode makes it easy to identify the user-mode malware instructions. But in kernel-mode I need to monitor and log instruction execution in the following two cases: (1) First, whenever the processor switches to kernel-mode to execute the request of user-mode malware (e.g., to perform a system call); and (2) Second, when an untrusted kernel instruction (injected kernel code or dropped driver module) executes.

Algorithm 2 summarizes how the analysis plug-in enables whole-system monitoring and logging. In line 2 the analysis plug-in checks the processor mode and the value of the CR3 register. (The CR3 value is the base address of the page directory

Algorithm 2: Whole-system tracing of malware operations.

```
1  begin
2      if CS ∈ User and CR3 ∈ malware_proc_CR3 then
3          return trace_user;
4      end
5      if CS ∈ Kernel then
6          if phy_addr (current_instruction) ∉ trusted_code then
7              return trace_kernel;
8          else
9              kthread = current_user_thread (kpcr_pointer);
10             if kthread ∈ malware_proc then
11                 return trace_kernel;
12             end
13         end
14     end
15     return dont_trace;
16 end
```

for the currently running process.) If the processor performs an instruction from user space and the `CR3` value belongs to a stored list of page directory base addresses of the malware process and new processes that the malware has created, the analysis plug-in enables user-mode tracing. Otherwise, if the processor works in kernel-mode, the analysis plug-in enables tracing if the current instruction is untrusted (line 6) or the current `KTHREAD` object belongs to a user-mode malware thread (line 10).

When tracing is enabled, SEMU provides two main logging options. The low-level logging option logs each instruction. The high-level option uses the data stored in the shadow memory to provide a high-level summary in terms of the names and addresses of both the invoked functions and the accessed data objects. In user-mode, the resulting log includes library calls, system calls, and IOCTLs. In kernel mode, the resulting log includes, besides others, the kernel code and data manipulation whenever an untrusted code attempts to modify a memory location that is included in the

shadow memory. SEMU thereby detects and logs function pointer hooking attacks and kernel data manipulations such as DKOM (Direct Kernel Object Manipulation).

For the Figure 7.1 example, SEMU logs any malware system calls from user-mode in step 1, operations performed by the rootkit in steps 1 and 2, the malware's service A system call in step 3, and the rootkit's execution of service B in step 6.

During malware execution, SEMU keeps its shadow memory in sync with the guest OS. This is done by the VMI plug-in, which tracks the execution of kernel functions that load new code or create, modify, or delete objects. SEMU then reflects such operations in its shadow memory. The VMI plug-in updates the shadow memory by adding the addresses of newly created objects and removing the addresses of deleted objects from the dMap. For this purpose the VMI plug-in monitors changes in the `OBJECT_DIRECTORY` structure after execution of several OS functions that allocate and deallocate memory in the kernel, such as `RtlAllocateHeap`, `ExAllocatePoolWithTag`, `ExFreePoolWithTag`, and `RtlFreeHeap`. When malware overwrites trusted code, SEMU similarly removes the overwritten code range from trusted_code.

8.3.1 Malware Logging

Table 8.1. Kernel execution trace format; EP = entry point; Addr = address; C = caller; T = target; M = module; D = data; F = member field; Inst = current instruction.

	Description
E	EP addr+name, M name
C	C addr+name, T addr+name, T M name, M name
W	Inst addr, D addr+name, F name, M name
R	Inst addr, D addr+name, F name, M name

Table 8.1 shows the format of the trace file the analysis plug-in collects from the kernel during malware execution. SEMU creates this log with information from its shadow memory, such as addresses and names of functions and objects. **E** represents the execution of a function. Whenever a function executes, the analysis plug-in logs its entry point address and its name (if the VMI plug-in has resolved the name). Since there is no resolved name for the functions that execute within kernel-mode malware, the analysis plug-in only writes the address and the name of the malware's kernel-mode module or U (untrusted code) for dropped drivers and injected codes.

For each executed control transfer (**C**) to an address I have in the shadow memory, the analysis plug-in writes the caller's address and the module name as well as the target's address and the module name. The analysis plug-in traces accesses (**W**=write and **R**=read) within the address range of a kernel object as follows. SEMU logs all direct writes of kernel data performed by untrusted code. SEMU logs the name of kernel data and its overwritten field members. SEMU also monitors writes of kernel data by the memory management functions that malware invokes.

The analysis plug-in also tags manipulated data structures. Whenever a read operation occurs within a manipulated kernel data structure, SEMU logs which kernel functions are affected by manipulated kernel data.

8.4 Post-Execution: Log Analysis

In the final step of mixed-mode analysis SEMU's trace analyzer produces a human readable report. The report contains name and address of modified kernel data as well as the internal OS functions that execute after user-mode requests and the functions that referred to manipulated OS objects.

SEMU contains a trace analyzer application that performs post-execution operations. The post-execution aggregates the collected log, for example, matching

system calls from user-mode with operations invoked by kernel-mode malware. For instance, in the Figure 7.1 example, SEMU matches the A call with the invocation of B, which reveals the malware's redirection of the system call A to the service B.

To extract the effect of kernel data manipulations in malware behavior, the trace analyzer compares the traces of malware operations both in presence and absence of kernel data protection. For this purpose, whenever a malware sample starts execution, SEMU takes a snapshot of the VM at the original entry point (OEP) of the program. Then it uses this snapshot to run the sample twice. For the first run, the analysis plug-in protects kernel data from manipulations of untrusted codes. In the second round, the plug-in allows the write operations of untrusted code within the kernel data addresses. Then, it compares the two execution logs and reports the differences.

8.5 Implementation in QEMU

At a high-level, SEMU uses a plug-in architecture. SEMU's functionality is packaged in components that plug into a VM such as QEMU. This approach decouples SEMU's analysis from the underlying virtual machine, which provides two main advantages. First, SEMU plug-ins can be loaded and unloaded dynamically at runtime to suite the analyst's needs. Second, all the analysis code that is specific to the guest OS or specific to a certain OS version is encapsulated within plug-ins. This architecture makes it relatively easy to support additional versions of the guest OS or a different guest OS such as Linux.

The two main SEMU plug-ins are the VMI plug-in and the analysis plug-in. Execution reaches these plug-ins via callback functions. The VM calls these callback functions before processing certain events such as guest OS system calls, switching from user-mode to kernel-mode, context switches, and kernel heap accesses. In QEMU, memory access operations can be monitored by analyzing the semantics of

x86 instructions. For example, I monitor `mov` instructions as they can change the value of a memory region. The x86 language has a vast number of read and write operations. However I can express the analysis very compactly in terms of QEMU's built-in write operation abstractions. QEMU then maps the analysis to all concrete x86 write operations.

Virtual addresses are easy to infer from physical addresses. SEMU therefore stores all addresses as physical addresses, which makes it easy to detect cases in which malware exploits the fact that two different virtual addresses may map to the same physical address. SEMU currently utilizes QEMU's built-in functions for converting virtual addresses into physical addresses.

To monitor read and write operations I customize softmmu codes in QEMU. QEMU uses softmmu in order to convert the physical addresses of the guest system to virtual addresses of the host system. This conversion is needed for each read and write. By hooking into QEMU's softmmu functions SEMU extracts the guest OS address being read or written by the current instruction.

To store the kernel data in the shadow memory, the current SEMU implementation uses the kernel data structure layout definitions of ReactOS[3]. ReactOS is an open-source re-implementation of Windows. But the SEMU code uses these ReactOS layouts only for ease of implementation. That is, I could easily generate these layouts from the PDB files and will do so for future SEMU versions.

[3]http://www.reactos.org

CHAPTER 9

MIXED-MODE MALWARE SAMPLES

This chapter describes several samples of mixed-mode malware. These samples serve to evaluate existing and future malware analysis tools.

9.1 Misleading User-Only VMI

This sample implements the Figure 7.1 motivating example attack that evades analysis by Ether. The sample misleads Ether-style user-only VMI by modifying the semantics of the system calls invoked by the malware, which leads tools such as Ether to log system calls that are different from the system calls actually executed by the malware.

The sample has a user-mode component and a kernel-mode component. The user-mode component Mal.exe is based on the SDBOT malware. I customized SDBOT source code to install our kernel-mode component (Figure 7.1, step 1) and to make system calls (Figure 7.1, step 3) after the kernel-mode component has modified the kernel.

Our kernel-mode component (also called rootkit) is a kernel-mode driver packaged as a resource file that changes the semantics of kernel system services. Our rootkit changes the semantics of the `DeleteFile` and `TerminateProcess` system calls.However a different implementation of our rootkit could easily change other system calls.

When the current user-mode thread such as Mal.exe requests a service, the OS follows the `ServiceTable` pointer of the current thread's user-mode `KTHREAD` object

to find the address of the requested service (Figure 7.1, step 4). To change system call semantics, our rootkit manipulates the `ServiceTable` pointer in the `KTHREAD` object of the user-mode Mal.exe process.

For console applications, `ServiceTable` points to the system calls exposed by OS image `ntoskrnl.exe`, via the system call table `SSDT`. That is, the OS initializes the `ServiceTable` pointer once via the internal (non-exported) `KeInitializeThread` function to point to the `SSDT` table.

Our kernel-mode component is somewhat similar to earlier rootkits that operate in isolation, without a cooperating user-mode component. These earlier rootkits set the `ServiceTable` pointer of various threads to the address of a fake `SSDT` table to hide the presence of malware processes [81]. Beyond hiding processes, I manipulate system call semantics to redirect subsequent system calls of Mal.exe in a way that evades current malware analyses.

9.2 MDL System Call Semantic Modification

This sample differs from Figure 7.1 in that it does not manipulate kernel objects directly. Instead this sample uses OS memory management functions to access and modify the system call table.

Figure 9.1 illustrates the kernel-mode component of of this sample. By using standard functions for memory operations, the rootkit creates a Memory Descriptor List (MDL)[1]. A MDL enables the rootkit to map the addresses of the `SSDT` table and overwrite its system call pointers.

In the example of Figure 9.1 the rootkit has overwritten the original pointers for system services A and B with the addresses of the A' and B' functions. A' and B' are provided by the rootkit. These rootkit functions check the currently running

[1] `http://msdn.microsoft.com/en-us/library/windows/hardware/ff565421(v=vs.85).aspx`

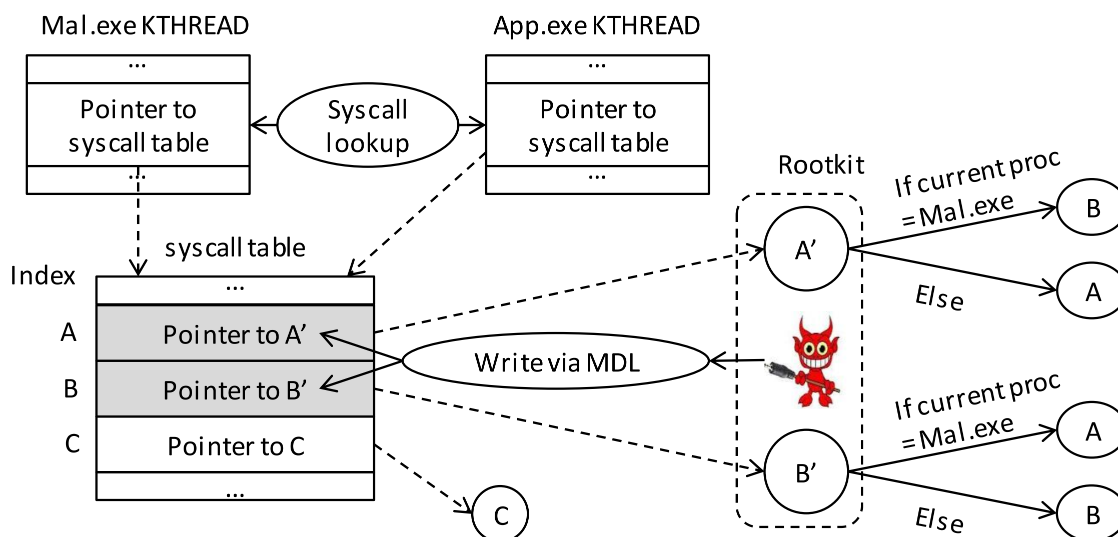

Figure 9.1. MDL system call semantic modification. .

process and swap system calls A and B only if the current process is the malware process Mal.exe.

Similar to Figure 7.1 this malware can mislead system call tracing. Similar to the sample of Section 9.1 we swap the system calls for `DeleteFile` with `CreateFile` and `TerminateProcess` with `CreateProcess`. The rootkit adjusts input parameters (and return values) in functions A’ and B’.

9.3 User-Level Acts on DKOM Attack

This mixed-mode malware sample consists of a user-mode component and a kernel-mode component. The kernel-mode component is based on the kernel-mode component of the FU rootkit, which uses Direct Kernel Object Manipulation (DKOM).

The FU rootkit also has a user-mode component but I replace this component as it only acts as a UI that sends commands to the kernel-mode FU component. Similar to FU, our user-mode component first installs the kernel-mode component,

i.e., a kernel-mode driver. Our user-mode component then waits for the kernel-mode component to use DKOM to hide both malware components. The user-mode component then checks if its own process has been hidden successfully and then adapts its subsequent behavior accordingly.

Our kernel-mode component performs DKOM by attempting to hide both malware components by unlinking the corresponding `EPROCESS` and `DRIVER_OBJECT` kernel-level objects in the list of running processes and drivers. Then the user-mode malware component enumerates the current running processes in the victim systems to check if the object hiding attempt succeeded. If the user-mode process is still in the list of running processes the user-mode component injects its payload as a shellcode into the SVCHOST program and terminates. Otherwise the user-mode component continues the execution of its malicious payload.

9.4 User-Level Acts on DKSM Attack

This sample differs from the one in Section 9.3 by replacing DKOM with DKSM [82]. This sample uses Direct Kernel Structure Manipulation (DKSM) to swap the image name and process id of the malware process with one of the running processes.

In the Windows operating system, the process id `PID` and the image name `ImagFileName` are member fields of the `EPROCESS` process object. To swap the `PID` and `ImageFileName` of the malware process with a running process, the kernel-mode rootkit accesses the list of process objects by calling the `PsGetCurrentProcess` function. The kernel-mode component can traverse the process list using the `flink` field. Since a kernel-mode driver such as our kernel-mode component can write all kernel memory it is then easy to swap the process id and process name of the malware process with another process.

Similar to the Section 9.3 DKOM sample, if process manipulation does not succeed, malware injects its payload into a victim process in user-mode. Otherwise, the user-mode malware continues execution as a standalone executable.

9.5 Stuxnet's Kernel Exploit

We used the Stuxnet-based example exploit (CVE-2010-2743) of Figure 7.2 and added a shell-code that performs a privilege escalation attack. The shell-code escalates the privilege level of the malware process, by swapping the `token` fields of the SYSTEM and malware process.

Specifically, the shell-code traverses the list of `EPROCESS` objects of the current running processes, searches for the SYSTEM process, and stores the SYSTEM `EPROCESS token` field. It then swaps this token field with the token field of the malware process.

Such a modification makes the malware process execute with administrator privileges. The malware can thus freely invoke a range of Windows APIs that are not allowed for non-privileged users. SEMU effectively detects the execution of the shell-code as an untrusted code running in kernel-mode and logs the modification of the `token` field.

9.6 User-Level Malware Acts on User-Mode Unhooking of Mapped Kernel SSDT

This malware is similar to the Stuxnet sample of Section 9.5 in that it also does not have a kernel-level component. Instead this malware sample has two user-mode components. The first user-mode component performs the tasks of a kernel-mode component, by writing directly into the physical memory pages of the OS kernel. Such memory mapping techniques are commonly used by malware [83]. Although all

malware components reside in user-mode, to analyze this malware sample, a malware analysis tool has to keep track of both kernel-space and user-space memory.

For this sample I assume that the system call table (`SSDT`) has been hooked by a malware analysis tool. Our malware sample thus writes into kernel memory to perform DKOM and unhook the `SSDT`.

Our first user-mode malware component writes to kernel memory by calling the Windows Native API cite[84], which is implemented in *ntdll.dll*. The Native API gives access to the physical pages of the `SSDT` table in the kernel, i.e., via the `NtOpenSection` and `NtMapViewOfSection` functions. Our second user-mode malware component then operates based on the success of the attempted `SSDT` DKOM manipulation.

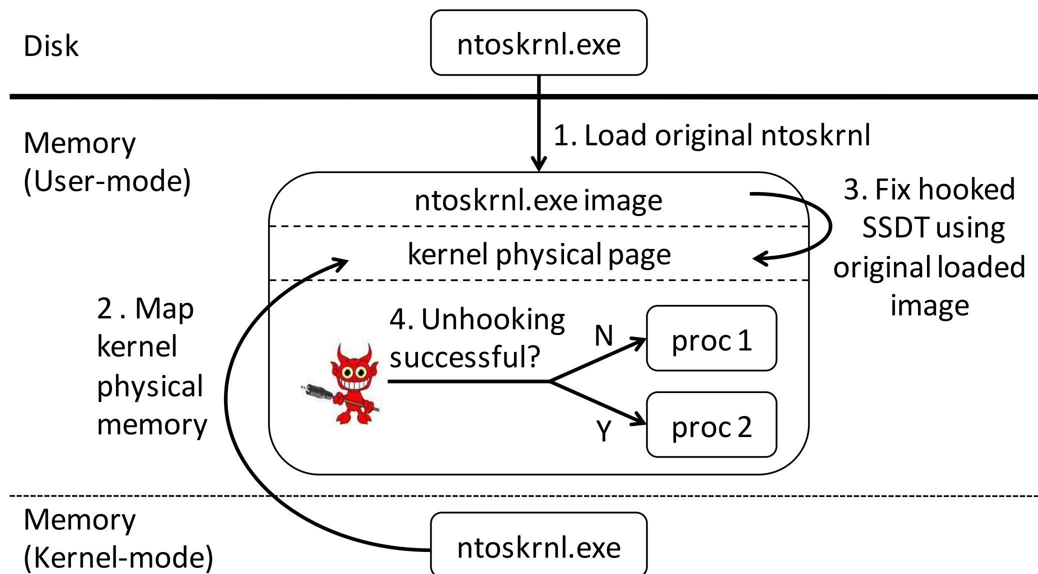

Figure 9.2. Unhooking system call table by a user-mode malware..

Figure 9.2 summarizes the attack procedure. As a first step, (1) the malware loads the file of the OS (ntoskrnl.exe) from disk into its user-level memory. This enables the malware to obtain the system service indices from the original SSDT

table, which has not yet been hooked by other programs such as a malware analysis or anti-virus software. In step (2), using Native API functions, the malware maps kernel memory to its own address space. In step (3) the malware compares the current SSDT with the original unhooked `SSDT` and fixes the current SSDT in the kernel memory based on the original unhooked SSDT loaded from disk. Based on the success of unhooking the system call table, the malware executes either “proc 1” or “proc 2”.

If a malware analysis tool such as a sandbox for user-mode malware analysis depends on `SSDT` hooking for malware analysis then the malware exposes two different behaviors, depending on the success of the attempted kernel manipulation. First, if the malware analysis tool protects the `SSDT` table from being overwritten by malware it can only analyze “proc 1”. Second, if the user-mode malware unhooks the `SSDT` successfully then the analysis tool is ineffective for monitoring the rest of the malware execution.

When tracing such a sample with SEMU, I run the sample twice. In the first run I allow the malware to access and overwrite the kernel. In the second run, I prevent the kernel from being overwritten, either directly by malware or indirectly by OS functions invoked by malware. This enables SEMU to analyze both malware behaviors (proc1 and proc2).

CHAPTER 10

SEMU EVALUATION

To evaluate the SEMU approach of analyzing mixed-mode malware I ask the following two research questions.

- RQ1: Can SEMU analyze mixed-mode malware that cannot be fully analyzed by current state-of-the-art approaches?
- RQ2: Is the SEMU execution time competitive with current state-of-the-art approaches?

To answer these two research questions I first implemented SEMU on top of QEMU as described in Section 8.5. I then compared the SEMU implementation with the two tools that are both closely related to SEMU and are fully available for experimentation, i.e., tools that provide access to their source code. These two tools were TEMU version 1.0 and the latest available Ether release (version 0.1).

10.1 Analyzing Mixed-Mode Malware (RQ1)

For RQ1, I implemented variations of the motivating examples and analyzed them on TEMU, Ether, and SEMU. I conducted these malware analysis experiments on a Debian Wheezy host system running on a 2.9 G.Hz Intel Core i7-3520M processor machine. The guest OS was Windows XP SP3 with 1 GB RAM 32 bit.

The six malware samples are written in C/C++. Table 10.1 lists the size of each malware sample in lines of code (LOC). The slowdown numbers in the last column are the overhead SEMU imposes for monitoring the system and writing the system

Table 10.1. Results of analyzing six mixed-mode malware samples. Via OS functions denotes if the malware manipulates kernel entities directly or by calling OS functions; # = section describing the malware.

#	Description	Affected Object	Via OS functions	Kernel LOC	User LOC	Slow-down
9.1	Modify system calls	KTHREAD	no	370	1,684	35.3
9.2	Modify system calls (MDL)	SSDT	yes	417	1,684	38.7
9.3	DKOM object hiding	EPROCESS, DRIVER_OBJECT	no	96	451	28.2
9.4	DKSM renaming	EPROCESS	no	111	451	20.6
9.5	Privilege escalation	EPROCESS	no	0	149	25.2
9.6	User-mode unhook	SSDT	yes	0	710	29.1

log during malware sample execution. I compare SEMU's overhead with the overhead of competing malware analysis approaches in the following Section 10.2.

The samples perform attacks including DKOM, DKSM [82], and hooking, by manipulating OS objects or data structures such as SSDT, KTHREAD, EPROCESS, and DRIVER_OBJECT. In this comparison SEMU was the only tool that could log all the events that are necessary for analyzing these attacks.

10.2 Malware Analysis Execution Time (RQ2)

For RQ2, I compared the total execution times of TEMU and SEMU in Table 10.2 and of Ether and SEMU in Table 10.3. To summarize, SEMU was faster than TEMU but slower than Ether. The TEMU/SEMU difference can be explained by the newer QEMU version used by SEMU. The Ether/SEMU difference is due to Ether using hardware extensions. However SEMU also works if these extensions are not available.

10.2.1 SEMU vs. TEMU's Inside-the-Guest VMI

This section compares the performance of SEMU with the closely related TEMU. SEMU and TEMU are built on the same QEMU VM architecture. TEMU uses QEMU v0.9 whereas SEMU uses the newer QEMU v0.14. While the QEMU versions

differ slightly, I do not expect a big performance difference from these different QEMU versions.

The main conceptual difference between SEMU and TEMU is the placement of VMI components (partially in-guest in TEMU vs. outside-the-guest in SEMU). The main goal of the evaluation in this section is thus to determine if this change of VMI architecture has a large negative impact on the malware analysis overhead. A malware analyst may be concerned that the switch from in-guest VMI in TEMU to outside-the-guest VMI in SEMU incurs a prohibitive performance penalty.

Table 10.2. Performance comparison of TEMU's (T) inside-the-guest VMI vs. SEMU's (S) outside-the-guest VMI using a typical, coarse-grained analysis task (symbol extraction); O/H = Overhead; ListDLLs = ListDLLs -d ntdll.dll.

Subject	w/o VMI [s]		Coarse [s]		% O/H	
	T	S	T	S	T	S
PsGetsid	1.68	0.56	3.44	1.09	105	95
Pslist -t	3.19	1.03	4.69	1.31	47	27
Psinfo -s	5.76	2.88	9.79	4.78	70	66
Coreinfo	1.70	0.65	3.75	1.07	121	63
ListDLLs	3.20	2.58	5.01	3.75	57	45

To compare the performance of in-guest with outside-the-guest VMI, I picked a typical, coarse-grained analysis task (symbol extraction), and applied it on five standard programs. Table 10.2 lists both the programs and the analysis times of TEMU and SEMU. The performance numbers show that SEMU did not incur an additional overhead over TEMU.

To make this comparison, I re-implemented the TEMU symbol extraction feature in SEMU, but placed it outside-the-guest with the rest of SEMU. TEMU extracts the names of processes, modules, and exported symbols from a running Windows sys-

tem. In other words, it keeps track of which processes have which modules loaded at which address, and it enumerates the exported symbols from each module.

To perform the comparison, I execute a Windows batch file in the guest OS that automatically executes and terminates the benchmark applications. The batch file records the application start and termination time stamps.

The guest system in the experiment is Windows XP SP 3 with 512MB allocated RAM. The first column of Table 10.2 lists benchmark applications from the Sysinternals utilities[1]. The second and third columns are average run-time of the applications in TEMU and SEMU when the guest system runs in its normal state—without VMI. The next two columns are the average application run-times when VMI is active and extracts symbols. The last two columns show the overhead of in-guest VMI in TEMU against the outside-the-guest VMI in SEMU. SEMU exhibited both an overall lower runtime and a lower relative VMI overhead.

10.2.2 SEMU vs. Ether's Single-Domain Analysis

This section compares SEMU's performance with the closely related Ether. Both tools place their analysis components (such as VMI) outside the guest OS.

SEMU differs from Ether in two key aspects. First, while SEMU is implemented on a software virtual machine, Ether leverages hardware extensions. I expect this difference to lead to higher performance in Ether. Second, Ether focuses on a single analysis domain, whereas SEMU covers both kernel-mode and user-mode. I expect this difference to further favor Ether over SEMU in terms of performance.

I conducted this experiment to determine if SEMU's performance remains within the same order of magnitude as the hardware-accelerated Ether. While hardware acceleration is often useful, not every hardware platform offers such acceleration. So it

[1] `http://technet.microsoft.com/en-us/sysinternals`

is important to have a an alternative such as SEMU that does not have the hardware constraints of Ether but still provides reasonable performance.

Table 10.3. Fine-grained VMI: Instruction tracing in Ether and SEMU (S). Timezone is Timezone /g.

Subject	w/o VMI [s]		fine VMI [s]		Slowdown	
	Ether	S	Ether	S	Ether	S
Efsinfo	0.63	2.42	20.54	21.39	32	8
Timezone	0.05	0.79	4.41	13.03	87	16
Whoami	0.03	0.72	4.49	19.83	149	27
UPX	0.32	9.00	45.58	322.60	141	35
RAR a	0.15	3.07	45.16	302.93	300	98

To compare the performance of SEMU to the hardware-accelerated Ether, I picked a typical fine-grained VMI task, i.e., logging each instruction, and applied it on the five standard programs listed in Table 10.3. The programs are Windows XP tools and the command-line version of the popular packing and archiving tools UPX and RAR. This experiment was conducted on a Debian Lenny domain-0 OS running on a 2.33 GHz Xeon machine with 32 GB RAM with a 1 GB RAM 32 bit Windows XP SP2 guest OS.

Table 10.3 also lists the analysis times of SEMU and Ether. SEMU maintains a reasonable performance when compared to the hardware-accelerated Ether.

I expect SEMU's performance to improve in future versions, as I have not yet optimized SEMU for speed. For example, SEMU does not yet leverage QEMU accelerators such as KQEMU [85] or KVM [86].

CHAPTER 11

SEMU LIMITATIONS AND DISCUSSION

In the following, I discuss the points of improvement for SEMU.

11.1 Run-time Patching Support

In the experiments I mainly focused on analysis of the mixed-mode malware samples that attack kernel data. However, in known rootkit samples, attackers employ run-time patching technique that is overwriting (hooking) internal OS functions which may not be accessible by function pointers within kernel data. The target in most of such attacks is a function that reveals some information about the presence of malicious codes in victim systems. The patched kernel function may not be executed during analysis of a sample but it runs within kernel context to serve other programs requests (i.e., antivirus) by providing inaccurate information about system state (i.e., hiding an existence of a thread). SEMU is able to detect and log traditional run-time patching attacks by monitoring write operation of untrusted code within the addresses in the shadow memory. However, in case of a mixed-mode malware attack, a patched function may execute during malware execution. Thus, for tracking a patched code execution, I need to update the shadow memory and remove the addresses which malware have modified by malware from the function map. I study the usage of run-time patching technique in mixed-mode malware and its analysis as the future work.

11.2 Taint Analysis Capability

In current implementation, for understanding malware behavior, I used combination of fine-grained (tracking write operations) and coarse-grained (function calls monitoring) to analyze the mixed-mode samples. For in-dept analysis I need to equip SEMU with taint analysis capability which enable us to track effects of sensitive information such as network input or function parameters in whole system information flow and mixed-mode malware execution. For instance, a malware sample can be written in a way that different input parameters for an API lead to execution of two different kernel-mode functions. Therefore, for in-depth analysis of the behavior adjustment feature in mixed-mode malware, I aim to apply taint analysis techniques.

11.3 Security Applications

In this dissertation, I used SEMU for malware analysis. As a future work, my goal is to customize current framework as a suite for other security applications such as rootkit detection, prevention and removal. For such purposes, in order to build an accurate reverse-engineered model of the guest OS, I may need to develop a technique that detects the presence of a malicious program in the guest system an also distinguish the clean state of a guest OS from the state that has been manipulated by an unwanted program.

11.4 Analysis Speedup.

In building the current prototype, I did not employ any technique to speed up the binary translation overhead in QEMU. However, since the analysis approach does not require any specific configuration from the virtualization software, by usage of accelerators (i.e., KQEMU [87]) or hardware virtualization technology such as

KVM [86] or Xen, I will be able reduce the analysis overhead. By building the prototype on top on QEMU, I were able to evaluate the effectiveness of the outside-the-guest VMI over in-guest VMI used in TEMU. At time of the experiment, TEMU was the only available tool that provided more semantic information from the guest OS.

11.5 Handling Anti-emulation Attacks

Malware authors attempt to evade analysis in virtual machines by writing the codes that crash or behave differently in presence of virtual hardware. These attacks mainly rely on inherent differences between virtualized hardware and real machines [88]. SEMU is not effective against such attacks. However, there are several approaches in the literature that have been previously proposed some techniques to mitigate such attacks [43, 88].

11.6 Execution Paths Coverage

Although the dynamic analysis techniques are powerful to extract malware behavior, they suffer from some limitations. One of the main limitations of the dynamic analysis is code coverage. That is, not all the code paths may execute when a piece of malware runs. Moreover, malware writers can embed a large loops of junk instructions into their codes in order to slowdown or foil the analysis. Thus, regarding such limitations, getting a complete log of all malicious activities of a mixed-mode malware still remains an open problem for this study. However, researchers in previous works provided some techniques to get better code coverage [56, 89].

11.7 Defending Against Obfuscated Malware.

Malware writers take advantage of obfuscation techniques such as packers or self-modifying codes to defeat malware analysis. Several countermeasures have been discussed in prior studies [90, 33, 91]. Current version of SEMU does not support obfuscated code analysis.

CHAPTER 12

CONCLUSION

In this dissertation, I explore the limitations of current malware detection and analysis techniques. In order to address the problems of the current techniques, I propose TDOIM and SEMU.

TDetecting rootkits in legacy applications poses several challenges in practice. Existing anti-malware techniques do not fully meet all these challenges. To bridge this gap, this paper introduced a Tiny Distributed On-demand Integrity Monitor for legacy applications (TDOIM). With a tiny footprint and therefore attack surface on the monitored machines TDOIM enables instant rootkit detection for legacy applications. Unlike existing techniques, TDOIM also also does not require virtualization stacks or special hardware. To evalute TDOIM, I implemented TDOIM for recent versions of Linux. In our experiments on several user and kernel mode rootkits, TDOIM achieved with moderate overhead and a relatively low false positive rate a 100% rootkit detection rate.

SEMU, is a mixed-mode malware analysis tool that resides outside the operating system and thereby outside the domain of user-kernel level malware. By monitoring of critical operations such as kernel data access and control flow of kernel during the analysis of all types of malware samples, SEMU is effective against the sophisticated mixed-mode malware which articulates actions in both user and kernel modes. The dissertation compares both analysis capabilities and overhead of SEMU with TEMU on Ether malware analysis techniques by performing several experiments on proof-of-concept implementations of mixed-mode malware as well as real-work rootkis.

CHAPTER 13

RELATED WORK TO TDOIM

Most closely related to TDOIM are cloud-based antivirus approaches, pioneered by CloudAV [13, 15, 16, 17]. These approaches reduce the attack surface for malware on the monitored host, by shifting much of the detection functionality from the host to a cloud-based server. Cloud-based antivirus is also useful for resource constrained (e.g., mobile) devices [15, 16, 17]. However cloud-based antivirus approaches still rely on curated blacklists (challenge 2).

Google's Camp uses whitelists, blacklists, and a reputation system to determine if a file downloaded by the web browser is safe [92]. However this approach does not meet all challenges of legacy applications, as it has a relatively big attack surface (challenge 1) and is not effective against an ongoing malware infection (challenge 3).

Similar to TDOIM, a rootkit detector built on Pioneer also periodically computes hashes of the kernel code and read-only data and sends these hashes to a server component. Pioneer does not rely on virtualized OS stacks or special hardware [93, 94, 95, 96, 97]. Instead, Pioneer times its execution and thereby detects rootkits. However, Pioneer is not well-suited for legacy applications because it requires prior knowledge of the installed software (challenge 4) and makes strong assumptions about machine and communication speed (challenge 5). While these assumptions have been partially relaxed, they do not support legacy applications communicating over public networks [94, 95].

Much progress has been made in end-to-end verification of the entire software stack [98]. While such approaches are promising, they currently do not scale to legacy operating systems such as Linux or Windows.

Many other approaches have been proposed for application and OS security monitoring. These approaches can be broadly classified into three categories—detecting kernel-level attacks, leveraging virtualized OS stacks or special hardware, and comparing memory with disk contents.

In the following, I provide more information about some of the related works to TDOIM.

13.1 Detecting Attacks on the Kernel

Traditional malware detection approaches focus exclusively on the integrity of the kernel [99, 21, 20]. As an example approach, when the OS loads a kernel-level device driver, earlier work performs static symbolic execution on the driver binaries, to check if the driver matches given patterns of malicious behavior [99]. Another well-known approach, Nickle, obtains the hash of kernel code and does not allow execution of any code that does not match this trusted hash [20]. Poker uses Nickle to detect rootkit execution at runtime and then captures a trace of the rootkit execution [21].

In some sense these approaches generalize the model followed by antivirus tools, as they compare relevant data against existing white-lists or black-lists. While these approaches are more general, as they may check for an entire class of attacks [99], they do not fully address challenge (4).

While necessary, detecting kernel-level attacks is not sufficient for end-to-end system reliability. For example, by employing user-mode rootkit techniques, attackers can compromise systems, run malicious payloads, and remain undetected from this category of integrity checkers. By monitoring both user-mode and kernel-mode ap-

plications and the whole operating system TDOIM is effective against common user- and kernel-mode rootkits.

13.2 Leveraging Virtualized OS Stacks or Special Hardware

Several integrity checking techniques such as Tripwire, Nickle, and Vigilare rely on virtualization (e.g., based on a hypervisor or software-based virtualization) [19, 20, 21, 22, 23] or specific hardware such as TPM or PCI add-in cards [25, 26, 27, 28, 30, 76, 100].

Livewire periodically checks the hashes of important programs and detects attacks by comparing these memory hashes in one system [42]. Such approaches require prior knowledge of the original file signatures or installation on the clean state of machines (challenges 3 and 4). Livewire also relies on virtual machine introspection (VMI). Similarly, ModChecker runs in a privileged VM and inspects the kernel modules of all other local VMs via VMI [101]. Nickle extends a VM monitor with a memory shadow copy for each monitored VM and code authentication features [20]. Nickle also has to be installed when a user system is clean of malware and rootkits.

SecVisor is a small hypervisor that protects kernel integrity by monitoring the execution of the kernel and avoids the execution of untrusted code [19]. SecVisor operates based on customizing the access rights to kernel memory pages at the harware-level. But using SecVisor requires modifying OS code. Several approaches rely on multiple VMs with the same OS configuration to detect rootkits [62, 102, 103]. For example, VMST and Virtuso use VMI to detect rootkit attacks, by comparing the machine states of a product VM with a security VM.

The problem of slight differences in hashed binaries (yielding different hashes) has been discussed in the scope of Bind [104]. Bind takes a proactive approach, by allowing each protected software to be associated with many hashes. However

such a proactive association has to be managed, for example, via an assumed trusted authority. TDOIM takes a reactive approach and manages different hash values via its server-based voting technique. Bind relies on a secure kernel and TPM.

Fides protects individual software modules. While Fides is very promising for newly developed software, Fides requires TPM and a custom compiler and thus Fides does not address legacy applications [105]. Haven would take this a step further and utilizes new hardware extensions (Intel SGX) to prevent not just malware but also cloud providers from manipulating user code [106]. While such hardware extensions may address much of the malware problem in the future they do not cover the large number of legacy machines.

Copilot runs completely in a PCI-add-in card and checks OS integrity at runtime [25]. These approaches are thus only effective when a particular hardware or a virtualization technology is available (challenge 5).

Flicker can execute analysis code securely, even on an infected system, by running the code in the hardware-protected secure VMM mode of modern machines that have a TPM crypto co-processor [76]. The integrity measurement architecture (IMA) is a TPM-based approach to detect if files on a system have been modified accidentally or maliciously [29]. To check the integrity of running applications, it is required to install IMA on a clean machine with access to original files on disk. In addition to checking the integrity of static memory contents, ReDAS also checks dynamic memory areas such as the stack [107]. On the other hand, ReDAS relies on TPM. I plan to add dynamic memory monitoring in future work.

13.3 Comparing Memory with Disk Contents

File integrity checkers such as Tripwire rely on an accurate comparison of runtime memory contents with original on-disk binaries [29, 24]. Such a comparison

requires either that the security tool maintains the original signatures of such files or the tool is installed on a clean state of the system to ensure that no binary is patched by malware. In other words, such security tools are not able to detect malware when they are installed in a compromised machine whose binaries have been modified on disk (challenge 3).

The system virginity verifier (SVV) is a cross-view based Windows rootkit detection approach that checks if code sections of important system DLLs and system drivers are the same in memory and in the corresponding executables on disk [60]. A related cross-view approach to detect rootkits uses outside-the-guest VMI introspection and compares different system views (e.g., inside vs. outside VMI) [51].

CHAPTER 14

RELATED WORK TO SEMU

I classify known malware techniques for kernel data manipulation into three categories. (1) First, in direct manipulation of kernel data an untrusted code directly modifies kernel data. (2) Second, malware can use memory management OS functions to map the physical addresses of the kernel data into newly allocated virtual addresses and manipulate kernel data through OS standard functions (using trusted code for kernel data manipulation). (3) Third, malware may not inject malicious code into the kernel. It instead may apply return-oriented programming (ROP) [72] and reuse OS codes to manipulate kernel data [73].

Tracking function calls in both the kernel and in untrusted code enables SEMU to analyze the first two types of kernel data attacks. Handling the last category, return-oriented attacks, is outside the scope of this article. However, since the analysis plug-in monitors execution of internal OS functions, SEMU can be customized to flag suspicious kernel data accesses that take place through trusted code execution but not by invoking standard OS functions that are responsible for initializing, modifying, or deleting OS objects. In other words, detecting return-oriented rootkits can be implemented easily in SEMU, by equipping SEMU with kernel control flow integrity checks and policies for accessing kernel data.

In the following, I discuss some related works to SEMU that I classified them into different categories.

14.1 Attacks on Trusted Kernel Code.

A common attack on trusted kernel code is a rootkit that patches or hooks kernel code at run-time. In this attack the attacker overwrites internal OS functions that may not be accessible by function pointers within kernel data. The target in most of these attacks are functions that reveal some information about the presence of malicious codes in victim systems.

The patched kernel functions may not be executed while analyzing a malware sample. But the patched function may run in the kernel context to serve other program requests, for example, to provide inaccurate information about the system state to antivirus tools [39, 108].

SEMU detects and logs run-time patching by monitoring how untrusted code writes within shadow memory addresses. After such a write a formerly trusted kernel function becomes untrusted. Hence SEMU removes the updated function from the list of trusted kernel functions.

14.2 Evading Virtual Machines.

There is a long history of "arms-races" between malware writers and malware analysts [109, 110, 88]. Evasive mixed-mode malware is one entry in this arms-race. However evasive mixed-mode malware differs from prior attempts of beating malware analysis. Previous evasion techniques mainly exploit properties of the infrastructure underlying malware analysis tools. That is, earlier techniques exploit the inherent differences between real hardware and virtual machines. Following are two examples. (1) Virtual machines such as QEMU typically run slower than real hardware. Malware can measure its execution time and adjust its behavior accordingly. (2) Several virtual machines such as QEMU have bugs or implementation shortcomings. For example,

QEMU currently does not support floating point instructions. By executing such non-supported instructions, malware can crash inside a virtual machine and therefore prevent analysis.

However virtual machines are becoming more common and widely used outside of malware analysis. So evading virtual machines completely may become less attractive for malware. By not running in virtual machines, a malware may prevent itself from infecting many systems that do not perform malware analysis.

14.3 Rootkit Analysis Techniques.

There are several recent pieces of work on profiling different behaviors of kernel-mode malicious codes. K-tracer [55], dAnubis [44], HookFinder [57], PoKeR [21] and Panorama [54] are example rootkit analysis tools that leverage various techniques such as taint analysis and system-wide information control flow tracking. These approaches have kernel-mode components or only focus on kernel-mode malware and thus are not effective for mixed-mode malware analysis.

14.4 Automatically Generating Outside-the-Guest VMI Tools.

Recent work including Virtuoso and VMST has made progress toward bridging the semantic gap outside-the-guest automatically. That is, these recent techniques can automatically generate outside-the-guest VMI tools for a variety of operating systems. Virtuoso analyzes the sequence of low-level commands executed by a OS-level introspection program to infer how to gather such data [41]. VMST forwards kernel data to the original copy of the OS [62, 79]. SEMU currently uses VMI tools that can be applied to several OS versions by supplying the corresponding PDB symbols. I plan to adapt these VMI automation ideas in future version of SEMU.

However both VMST and Virtuoso rely on the integrity of kernel data. These techniques are therefore currently not sufficient for extracting an accurate semantic view when the guest OS has been compromised by attacks on kernel data (e.g., via DKOM) [41, Section 6C].

For in-depth malware analysis VMST and Virtuoso also can only solve a part of the VMI problem. That is, VMST and Virtuoso do not cover many of the kernel data that are needed for malware analysis, such as member fields of kernel data [41, Section 3A]. However such detailed memory data is required, for example, to monitor the syscall table pointer within the `KTHREAD` object in Figure 7.1.

14.5 Protecting Against Kernel Exploits.

There have been several recent kernel exploit protection schemes [19, 20, 53, 111]. SEMU uses such a kernel data protection scheme in one round of its analysis to collect the difference between the compromised and the original kernel state. However kernel data protection is just one aspect of SEMU and SEMU has several analysis components that are not found in kernel protection schemes.

SecVisor is a kernel exploit protection approach that uses hardware virtualization to prevent unauthorized code from running in the guest OS [19]. SecVisor requires modifying kernel code and does not support closed-source operating systems.

Nickel monitors the execution of kernel-level instructions in the guest OS and prevents the execution of unauthorized code such as rootkits [20]. Similar to SEMU, Nickel maintains a hash map of trusted kernel codes.

Sentry is a hypervisor-based kernel data protection system that monitors the execution of untrusted kernel-mode applications [53]. Sentry prevents writes of untrusted kernel code within the address space of kernel data.

DeepSafe is a hardware assistant product by McAfee and Intel that resides beyond the operating system [111]. DeepSafe monitors sensitive memory regions and CPU registers to prevent advanced stealthy rootkit attacks and detect APTs (Advanced Persistence Threats).

14.6 Offline and Interactive Analysis Tools.

In-depth binary analysis in virtual machines requires an OS-aware technique that builds an accurate semantic view of the guest OS. This requirement makes reverse-engineering of a closed-source OS such as Windows a prominent VMI task. There are several tools for offline forensic analysis, which construct a reverse-engineered model of the Windows OS [112, 113, 80]. These tools derive from a given snapshots of OS memory and data.

An example forensic framework is Volatility [113]. Volatility has been used to perform VMI on the memory snapshots of virtual machines to detect the presence of rootkits [114, 115].

These approaches perform VMI after a security incident has occurred. SEMU's VMI reverse-engineers the guest system online and records the interactions of the mixed-mode malware components. Although we presented SEMU as an online analysis framework, SEMU could also be customized to detect malware footprints offline in memory dumps of guests systems.

Virt-ICE [80] is an interactive virtualized debugger that provides complete isolation between debugger and the guest OS. Similar to the current SEMU implementation, Virt-ICE uses the ReactOS source code for outside-the-guest VMI. In the Virt-ICE architecture the virtual machine communicates with the Virt-ICE client through a TCP connection. For example, a user sends commands such as `ps` to return the list of processes using the Virt-ICE client to the virtualized debugger. While

Virt-ICE requires user interactions, SEMU is a fully automated malware analysis tool.

APPENDIX A

Linux Implementation of Client-Side Agent

Since TDOIM's client-side agent has a deliberately small feature set, it is relatively straight-forward to adapt the prototype implementation to older versions of Linux and other operating systems such as Windows. Following is the detailed list of APIs and macros the TDOIM client-side agent calls in the prototype implementation for recent versions of Linux.

A.1 Memory Access

kmalloc, kfree, kmap, kunmap, memcpy, get_free_page, free_page, page_cache_release, get_user_page, get_kernel_page, down_read, get_kernel_page, phys_to_virt, read_cr3, pgd_present, pmd_present, pte_present, pmd_large, pte_pfn, pte_offset_kernel.

A.1.0.1 Hashing

crypto_alloc_hash, crypto_hash_init, crypto_hash_update, crypto_hash_final and crypto_free_hash,

A.1.0.2 String Manipulation

Standard C library calls for string manipulations such as strncat, strncpy, strlen.

A.1.0.3 Network Access

sock_create, bind, sock_release, sock_sendmsg.

A.1.0.4 Synchronization

spin_lock_irqsave, spin_unlock_irqrestore, create_singlethread_workqueue, flush_workqueue, destroy_workqueue, mutex_lock, mutex_unlock, task_lock, task_unlock.

REFERENCES

[1] Consumer Reports, “Online exposure,” *Consumer Reports Magazine*, June 2011.

[2] A. Matrosov, E. Rodionov, D. Harley, and J. Malcho, “Stuxnet under the microscope,” *ESET LLC (September 2010)*, 2010.

[3] L. of Cryptography and S. S. (CrySyS), “Duqu: A stuxnet-like malware found in the wild,” Budapest University of Technology and Economics, Tech. Rep., Oct. 2011, accessed Sept. 2015. [Online]. Available: www.crysys.hu/publications/files/bencsathPBF11duqu.pdf

[4] sKyWIper Analysis Team, “sKyWIper (a.k.a. flame a.k.a. flamer): A complex malware for targeted attacks,” Budapest University of Technology and Economics, Tech. Rep., May 2012, accessed Sept. 2015. [Online]. Available: http://www.crysys.hu/skywiper/skywiper.pdf

[5] cert.pl, “More human than human: Flame’s code injection techniques,” Aug. 2012, accessed Sept. 2015. [Online]. Available: http://www.cert.pl/news/5874

[6] Ponemon Institute, “2014 cost of data breach study: Global analysis,” May 2014.

[7] S. K. Cha, I. Moraru, J. Jang, J. Truelove, D. Brumley, and D. G. Andersen, “SplitScreen: Enabling efficient, distributed malware detection,” in *Proc. 7th USENIX NSDI*, San Jose, CA, Apr. 2010.

[8] L. Harvey, “Secure your legacy apps and servers: Safeguard end-of-life operating systems, unsupported applications, and classic applications,”

accessed Sept. 2015. [Online]. Available: http://www.mcafee.com/us/resources/technology-blueprints/tb-securing-legacy-systems.pdf

[9] F. Xue, "Attacking antivirus," in *Black Hat Europe*, Mar. 2008.

[10] B. Min, V. Varadharajan, U. K. Tupakula, and M. Hitchens, "Antivirus security: Naked during updates," *Software: Practice and Experience*, vol. 44, no. 10, pp. 1201–1222, Oct. 2014.

[11] B. Min and V. Varadharajan, "Design, implementation and evaluation of a novel anti-virus parasitic malware," in *Proc. 30th Annual ACM Symposium on Applied Computing (SAC)*. ACM, Apr. 2015, pp. 2127–2133.

[12] ——, "A novel malware for subversion of self-protection in anti-virus," *Software: Practice and Experience*, 2015, to appear.

[13] J. Oberheide, E. Cooke, and F. Jahanian, "CloudAV: N-version antivirus in the network cloud," in *Proc. 17th USENIX Security Symposium*. USENIX, July 2008, pp. 91–106.

[14] Damballa, "State of infections report: Q4 2014," accessed Sept. 2015. [Online]. Available: http://landing.damballa.com/state-infections-report-q4-2014.html

[15] J. Oberheide, K. Veeraraghavan, E. Cooke, J. Flinn, and F. Jahanian, "Virtualized in-cloud security services for mobile devices," in *Proceedings of the First Workshop on Virtualization in Mobile Computing*, ser. MobiVirt '08. ACM, 2008, pp. 31–35.

[16] S. Zonouz, A. Houmansadr, R. Berthier, N. Borisov, and W. Sanders, "Secloud: A cloud-based comprehensive and lightweight security solution for smartphones," *Computers & Security*, vol. 37, pp. 215–227, 2013.

[17] C. Jarabek, D. Barrera, and J. Aycock, "Thinav: Truly lightweight mobile cloud-based anti-malware," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 209–218.

[18] "Mcafee global threat intelligence technology," accessed Sept. 2015. [Online]. Available: http://www.mcafee.com/us/threat-center/technology/global-threat-intelligence-technology.aspx

[19] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Oct. 2007, pp. 335–350.

[20] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing," in *Proc. 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*. Springer, Sept. 2008, pp. 1–20.

[21] ——, "Multi-aspect profiling of kernel rootkit behavior," in *Proc. 4th ACM European Conference on Computer Systems (EuroSys)*. ACM, Apr. 2009, pp. 47–60.

[22] M. Conover and T.-c. Chiueh, "Code injection from the hypervisor: Removing the need for in-guest agents," in *Black Hat USA*, July 2009.

[23] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with OSck," in *Proc. 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Mar. 2011, pp. 279–290.

[24] G. H. Kim and E. H. Spafford, "The design and implementation of Tripwire: A file system integrity checker," in *Proc. 2nd ACM Conference on Computer and Communications Security*. ACM, 1994, pp. 18–29.

[25] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot-a coprocessor-based kernel runtime integrity monitor." in *Proc. USENIX Security Symposium*. San Diego, USA, 2004, pp. 179–194.

[26] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell, "Linux kernel integrity measurement using contextual inspection," in *Proc. ACM Workshop on Scalable Trusted Computing*. ACM, Nov. 2007, pp. 21–29.

[27] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, "Vigilare: toward snoop-based kernel integrity monitor," in *Proc. ACM Conference on Computer and Communications Security*. ACM, 2012, pp. 28–37.

[28] L. Duflot, D. Etiemble, and O. Grumelard, "Using cpu system management mode to circumvent operating system security functions," *CanSecWest/core06*, 2006.

[29] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn, "Design and implementation of a tcg-based integrity measurement architecture." in *Proc. USENIX Security Symposium*, vol. 13, 2004, pp. 223–238.

[30] T. Jaeger, R. Sailer, and U. Shankar, "Prima: Policy-reduced integrity measurement architecture," in *Proc. 11th ACM Symposium on Access Control Models and Technologies*. ACM, 2006, pp. 19–28.

[31] D. X. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *Proc. 4th International Conference on Information Systems Security (ICISS)*. Springer, Dec. 2008, pp. 1–25.

[32] U. Bayer, A. Moser, C. Krügel, and E. Kirda, "Dynamic analysis of malicious code," *Journal in Computer Virology*, vol. 2, no. 1, pp. 67–77, Aug. 2006.

[33] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proc. 15th CCS*. ACM, Oct. 2008, pp. 51–62.

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

[35] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. C. Freiling, "Measurements and mitigation of peer-to-peer-based botnets: A case study on storm worm." in *Proc. 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats (LEET)*. USENIX, Apr. 2008.

[36] C. Xuan, J. Copeland, and R. Beyah, "Toward revealing kernel malware behavior in virtual execution environments," in *Proc. International Symposium on Recent Advances in Intrusion Detection (RAID)*. Springer, Sept. 2009, pp. 304–325.

[37] A. Lineberry, "Malicious code injection via /dev/mem." Black Hat Europe, Mar. 2009.

[38] S. S. R. Team, "Rootkits," accessed Sept. 2015. [Online]. Available: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/rootkits.pdf

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

[40] U. Bayer, C. Krügel, and E. Kirda, "TTAnalyze: A tool for analyzing malware," in *15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*. EICAR, Apr. 2006.

[41] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proc. 32th IEEE Symposium on Security and Privacy (Oakland)*. IEEE, May 2011, pp. 297–312.

[42] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. Network and Distributed System Security Symposium (NDSS)*. The Internet Society, Feb. 2003.

[43] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Krügel, and G. Vigna, "Efficient detection of split personalities in malware," in *Proc. Network and Distributed System Security Symposium (NDSS)*. The Internet Society, Feb. 2010.

[44] M. Neugschwandtner, C. Platzer, P. Comparetti, and U. Bayer, "dAnubis - dynamic device driver analysis based on virtual machine introspection," in *Proc. 7th DIMVA*. Springer, July 2010, pp. 41–60.

[45] D. Chisnall, *The definitive guide to the Xen hypervisor*. Pearson Education, Nov. 2007.

[46] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin, "V2E: Combining hardware virtualization and software emulation for transparent and extensible malware analysis," in *Proc. 8th Annual International Conference on Virtual Execution Environments (VEE)*. ACM, Mar. 2012, pp. 227–237.

[47] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 555–565.

[48] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-vm monitoring using hardware virtualization," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 477–487.

[49] H. Yin, P. Poosankam, S. Hanna, and D. X. Song, "Hookscout: Proactive binary-centric hook detection," in *Proc. 7th International Conference on De-*

tection of Intrusions and Malware, and Vulnerability Assessment (DIMVA). Springer, July 2010, pp. 1–20.

[50] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proc. 16th ACM Conference on Computer and Communications Security (CCS).* ACM, Nov. 2009, pp. 545–554.

[51] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction," in *Proc. 14th ACM Conference on Computer and Communications Security (CCS).* ACM, Oct. 2007, pp. 128–138.

[52] X. Jiang and X. Wang, ""Out-of-the-box" monitoring of VM-based high-interaction honeypots," in *Proc. 10th International Symposium on Recent Advances in Intrusion Detection (RAID).* Springer, Sept. 2007, pp. 198–218.

[53] A. Srivastava and J. Giffin, "Efficient protection of kernel data structures via object partitioning," in *Proc. 28th Annual Computer Security Applications Conference (ACSAC).* ACM, Dec. 2012, pp. 429–438.

[54] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proc. 14th Conference on Computer and Communications Security (CCS).* ACM, Oct. 2007, pp. 116–127.

[55] A. Lanzi, M. Sharif, and W. Lee, "K-tracer: A system for extracting kernel malware behavior," in *Proc. Network and Distributed System Security Symposium (NDSS).* The Internet Society, Feb. 2009.

[56] A. Moser, C. Krügel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Proc. 28th IEEE Symposium on Security and Privacy (Oakland).* IEEE, May 2007, pp. 231–245.

[57] H. Yin, Z. Liang, and D. Song, “Hookfinder: Identifying and understanding malware hooking behaviors,” in *Proc. Network and Distributed System Security Symposium (NDSS)*. The Internet Society, Feb. 2008.

[58] C. Kolbitsch, E. Kirda, and C. Kruegel, “The power of procrastination: detection and mitigation of execution-stalling malicious code,” in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 285–296.

[59] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An updated performance comparison of virtual machines and Linux containers,” IBM Research, Tech. Rep. RC25482, July 2014.

[60] J. Rutkowska, “System virginity verifier,” in *Hack in the Box security Conference*, 2005, pp. 2–25.

[61] D. P. Bovet and M. Cesati, *Understanding the Linux kernel.* ” O’Reilly Media, Inc.”, 2005.

[62] Y. Fu and Z. Lin, “Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection,” in *Proc. 33rd IEEE Symposium on Security and Privacy (Oakland)*. IEEE, 2012, pp. 586–600.

[63] J. Edge, “Kernel address space layout randomization,” in *Linux Security Summit*, Oct. 2013, accessed Sept. 2015. [Online]. Available: https://lwn.net/Articles/569635

[64] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, “On the effectiveness of address-space randomization,” in *Proc. 11th ACM Conference on Computer and Communications Security*. ACM, 2004, pp. 298–307.

[65] J. Sylve, “Lime: Linux memory extractor,” september 2015. [Online]. Available: https://github.com/504ensicslabs/lime

[66] G. Dabah, "Distorm: Powerful disassembler library for x86/amd64," accessed September 2015". [Online]. Available: http://code.google.com/p/distorm

[67] J. Black, M. Cochran, and T. Highland, "A study of the MD5 attacks: Insights and improvements," in *Proc. 13th International Workshop on Fast Software Encryption (FSE).* Springer, Mar. 2006, pp. 262–277.

[68] N. Bareil, "ld-linux.so ELF hooker," accessed September 2015. [Online]. Available: http://justanothergeek.chdir.org/2011/11/ld-linuxso-elf-hooker/

[69] S. Clowes, "Injectso: Modifying and spying on running processes under linux and solaris," in *Blach Hat Europe*, Nov. 2001, accessed Sept. 2015. [Online]. Available: http://www.blackhat.com/presentations/bh-europe-01/shaun-clowes/bh-europe-01-clowes.ppt

[70] E. Bradbury, "Syscall hooker," accessed Sept. 2015. [Online]. Available: https://github.com/ebradbury/linux-syscall-hooker

[71] M. Coppola, "Suterusu rootkit: Inline kernel function hooking on x86 and ARM," accessed Sept. 2015. [Online]. Available: http://poppopret.org/2013/01/07/suterusu-rootkit-inline-kernel-function-hooking-on-x86-and-arm

[72] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, pp. 2:1–2:34, Mar. 2012.

[73] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *Proc. 18th USENIX Security Symposium.* USENIX, Aug. 2009, pp. 383–398.

[74] L. Davi, A.-R. Sadeghi, and M. Winandy, "Ropdefender: A detection tool to defend against return-oriented programming attacks," in *Proceedings of the 6th*

ACM Symposium on Information, Computer and Communications Security. ACM, 2011, pp. 40–51.

[75] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-free: defeating return-oriented programming through gadget-less binaries," in *Proceedings of the 26th Annual Computer Security Applications Conference.* ACM, 2010, pp. 49–58.

[76] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *Proc. EuroSys.* ACM, Apr. 2008, pp. 315–328.

[77] M. Neugschwandtner, C. Platzer, P. Comparetti, and U. Bayer, "dAnubis - dynamic device driver analysis based on virtual machine introspection," in *Proc. 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA).* Springer, July 2010, pp. 41–60.

[78] S. B. Schreiber, *Undocumented Windows 2000 secrets: A programmer's cookbook.* Addison-Wesley, May 2001.

[79] Y. Fu and Z. Lin, "Bridging the semantic gap in virtual machine introspection via online kernel data redirection," *ACM Transactions on Information and System Security (TISSEC)*, vol. 16, no. 2, pp. 7:1–7:29, Sept. 2013.

[80] N. A. Quynh and K. Suzaki, "Virt-ICE: Next-generation debugger for malware analysis," in *Black Hat Briefings USA*, July 2010.

[81] A. Kapoor and R. Mathur, "Predicting the future of stealth attacks," in *Virus Bulletin Conference*, Oct. 2011.

[82] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu, "DKSM: Subverting virtual machine introspection for fun and profit," in *Proc. 29th IEEE Symposium on Reliable Distributed Systems (SRDS).* IEEE, Oct. 2010.

[83] E. Florio, "When malware meets rootkits," in *Virus Bulletin.* Virus Bulletin, Dec. 2005.

[84] M. E. Russinovich and D. A. Solomon, *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*, 4th ed. Microsoft Press, Jan. 2005.

[85] D. Bartholomew, "QEMU: A multihost, multitarget emulator," *Linux Journal*, no. 145, May 2006.

[86] I. Habib, "Virtualization with KVM," *Linux Journal*, no. 166, Feb. 2008.

[87] "Kqemu," http://wiki.qemu.org/KQemu/Doc. Accessed Dec. 2013.

[88] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song, "Emulating emulation-resistant malware," in *Proc. 1st ACM Workshop on Virtual Machine Security (VMSec).* ACM, Nov. 2009, pp. 11–22.

[89] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero, "Identifying dormant functionality in malware programs," in *Security and Privacy (SP), 2010 IEEE Symposium on.* IEEE, 2010, pp. 61–76.

[90] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," in *Proc. 5th ACM Workshop on Recurring Malcode (WORM)*, Oct. 2007.

[91] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *Security and Privacy, 2009 30th IEEE Symposium on.* IEEE, 2009, pp. 94–109.

[92] M. A. Rajab, L. Ballard, N. Lutz, P. Mavrommatis, and N. Provos, "Camp: Content-agnostic malware protection." in *NDSS*, 2013.

[93] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. K. Khosla, "Pioneer: Verifying code integrity and enforcing untampered code execution on

legacy systems," in *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP).* ACM, Oct. 2005, pp. 1–16.

[94] L. Martignoni, R. Paleari, and D. Bruschi, "Conqueror: Tamper-proof code execution on legacy systems," in *Proc. 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA).* Springer, July 2010, pp. 21–40.

[95] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth, "New results for timing-based attestation," in *Proc. IEEE Symposium on Security and Privacy (Oakland).* IEEE, May 2012, pp. 239–253.

[96] F. Armknecht, A. Sadeghi, S. Schulz, and C. Wachsmann, "A security framework for the analysis and design of software attestation," in *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS).* ACM, Nov. 2013, pp. 1–12.

[97] A. Ghosh, A. Sapello, A. Poylisher, C. J. Chiang, A. Kubota, and T. Matsunaka, "On the feasibility of deploying software attestation in cloud environments," in *Proc. 7th IEEE International Conference on Cloud Computing (CLOUD).* IEEE, June 2014, pp. 128–135.

[98] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, "Ironclad apps: End-to-end security via automated full-system verification," in *Proc. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI).* USENIX, Oct. 2014, pp. 165–181.

[99] C. Kruegel, W. Robertson, and G. Vigna, "Detecting kernel-level rootkits through binary analysis," in *Proc. 20th Annual Computer Security Applications Conference.* IEEE, 2004, pp. 91–100.

[100] D. Schellekens, B. Wyseur, and B. Preneel, "Remote attestation on legacy operating systems with trusted platform modules," *Science of Computer Programming*, vol. 74, no. 1–2, pp. 13–22, Dec. 2008.

[101] I. Ahmed, A. Zoranic, S. Javaid, and G. G. R. III, "Modchecker: Kernel module integrity checking in the cloud environment," in *Proc. 41st International Conference on Parallel Processing Workshops (ICPPW)*. IEEE, Sept. 2012, pp. 306–313.

[102] Y. Fu and Z. Lin, "Exterior: Using a dual-vm based external shell for guest-os introspection, configuration, and recovery," in *ACM SIGPLAN Notices*, vol. 48, no. 7. ACM, 2013, pp. 97–110.

[103] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proc. 32th IEEE Symposium on Security and Privacy (Oakland)*. IEEE, May 2011, pp. 297–312.

[104] E. Shi, A. Perrig, and L. van Doorn, "BIND: A fine-grained attestation service for secure distributed systems," in *Proc. IEEE Symposium on Security and Privacy (Oakland)*. IEEE, May 2005, pp. 154–168.

[105] R. Strackx and F. Piessens, "Fides: Selectively hardening software application components against kernel-level or process-level malware," in *Proc. ACM Conference on Computer and Communications Security (CS)*. ACM, Oct. 2012, pp. 2–13.

[106] A. Baumann, M. Peinado, and G. C. Hunt, "Shielding applications from an untrusted cloud with haven," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, Oct. 2014, pp. 267–283.

[107] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang, "Remote attestation to dynamic system properties: Towards providing complete system integrity evi-

dence," in *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, June 2009, pp. 115–124.

[108] B. Blunden, *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Jones & Bartlett, May 2009.

[109] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti, "Detecting environment-sensitive malware," in *Proc. 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*. Springer, Sept. 2011, pp. 388–357.

[110] N. Falliere, "Windows anti-debug reference," http://www.symantec.com/connect/articles/windows-anti-debug-reference. Accessed Dec. 2013.

[111] "Deepsafe," http://www.mcafee.com/us/solutions/mcafee-deepsafe.aspx. Accessed Dec. 2013.

[112] "Responder pro," http://hbgary.com/products. Accessed Dec. 2013.

[113] "Volatility foundation," accessed Sept. 2015. [Online]. Available: https://github.com/volatilityfoundation

[114] A. Bianchi, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Blacksheep: Detecting compromised hosts in homogeneous crowds," in *Proc. ACM Conference on Computer and Communications Security (CCS)*. ACM, Oct. 2012, pp. 341–352.

[115] B. Dolan-Gavitt, B. Payne, and W. Lee, "Leveraging forensic tools for virtual machine introspection," Georgia Institute of Technology, Tech. Rep. GT-CS-11-05, May 2011.

BIOGRAPHICAL STATEMENT

Shabnam Aboughadareh obtained her PhD in computer science from University of Texas at Arlington. She earned a bachelor degree in computer engineering from Amirkabir University of Technology.

Shabnam is working on problems in system security. Her main research interests include operating system, virtualization technologies and malware analysis. She received a best paper award in 4th ACM Program Protection and Reverse Engineering Workshop (PPREW). She worked at Symantec Research Labs (SRL) as a research intern and received the first prize in Symantec's 7th Annual Intern Show Case Competition.