Stony Brook University

Run-Time Deep Virtual Machine Introspection and Its Applications

A Dissertation Presented

by

Jennia Hizver

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

July 2013

Stony Brook University

The Graduate School

Jennia Hizver

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation.

Tzi-cker Chiueh – Dissertation Advisor
Professor, Department of Computer Science

Jie Gao - Chairperson of Defense
Associate Professor, Department of Computer Science

Scott Stoller
Professor, Department of Computer Science

Steven Murdoch
Senior Research Associate, Computer Laboratory, University of Cambridge

This dissertation is accepted by the Graduate School

Charles Taber
Interim Dean of the Graduate School

Abstract of the Dissertation

Run-Time Deep Virtual Machine Introspection and Its Applications

by

Jennia Hizver

Doctor of Philosophy

in

Computer Science

Stony Brook University

2013

Virtual Machine Introspection (VMI) is a new and important technique developed specifically for virtualized environments. VMI provides the ability to perform virtual machine (VM) monitoring by gathering VM run-time states from the hypervisor and analyzing those states to obtain information about a running operating system (OS) without installing an agent inside the VM. The agentless VMI approach has enabled the development of applications that combine the best of two worlds: efficient centralization and effective monitoring.

VMI's primary drawback is the semantic gap problem. The semantic gap refers to the difficulty in interpreting low level run-time OS states obtained through VMI into a high level model of the OS's state. We approached the problem through the creation of the real-time kernel data structure monitoring (RTKDSM) system. The RTKDSM system leverages the rich OS analysis capabilities of Volatility, an open source forensics framework, to simplify and automate analysis of VM run-time states of Windows and Linux OSes. The RTKDSM system is designed as an extensible software framework, which can be extended by writing Volatility plugins to perform new VM analysis tasks. In addition, the RTKDSM system is built to perform real-time monitoring of the extracted OS states in guest VMs to detect changes made to these states. This feature is especially important for effective security monitoring of VMs. To improve the efficiency of the RTKDSM framework, we reduce the overhead of monitoring changes to guest OS states.

The RTKDSM system is capable of supporting a wide range of VMI applications due to the RTKDSM framework's flexibility and extensibility. Leveraging the RTKDSM framework, VMI developers can easily create new VMI applications. To demonstrate the practicality and effectiveness of the RTKDSM framework, we built three novel applications on top of the

framework: (1) an inter-VM data flow tracking tool, (2) a VM lock down tool to restrict the execution environment to running only approved user applications, and (3) a tool for detection of malicious attacks that manipulate privileges of running processes. These systems are expected to contribute to enhanced system monitoring in virtual machine environments.

Table of Contents

1 Introduction 1

1.1 Motivation and Challenges 1

1.2 Dissertation Contributions 3

1.2.1 Real-Time Kernel Data Structure Monitoring System 3

1.2.2 Payment Card Data Flow Tracking Tool 4

1.2.3 Cloud-Based Application Whitelisting 5

1.2.4 Access Token Manipulation Detection Tool 5

2 Real-Time Kernel Data Structure Monitoring System 7

2.1 Introduction 7

2.2 Related Work 7

2.2.1 Semantically Aware Systems 7

2.2.2 Semantically Unaware Systems 8

2.2.3 VMI Frameworks For Semantically Aware Systems 9

2.2.4 Real-Time Data Structure Monitoring Systems 11

2.3 Background 13

2.3.1 Xen Hypervisor 13

2.3.2 Dirty Page Tracking 14

2.3.2.1 Shadow Paging Technique 14

2.3.2.2 Log Dirty Mode 15

2.3.3 Forensic Memory Analysis 16

2.3.3.1 Volatility Framework 17

2.3.3.2 Data Structure Classification 22

2.4 Design and Implementation 23

2.4.1 Assumptions and Requirements 23

2.4.2 Design 24

2.4.3 Implementation 27
2.4.4 Limitations 33
2.5 Evaluation 35
2.5.1 Experimental Setup 35
2.5.2 Spurious Page Fault Experiments 35
2.5.3 Performance Experiments 41
2.5.3.1 “Always-On” Mode 42
2.5.3.2 “Periodic Polling” Mode 47
2.6 Summary 52
3 Automated Discovery of Credit Card Data Flow for PCI DSS Compliance 53
3.1 Introduction 53
3.2 Related Work 55
3.2.1 Dynamic Binary Instrumentation Systems 56
3.2.2 Emulator-Based Systems 57
3.3 Design and Implementation 58
3.3.1 Payment Card Processing System 58
3.3.2 Assumptions 59
3.3.3 Requirements 59
3.3.4 System Overview 59
3.3.5 Main Components 61
3.3.5.1 Tracing of Inter-VM Communications 62
3.3.5.2 Searching the Process Memory 64
3.3.5.3 Card Data Flow Reconstruction 65
3.4 Evaluation 66
3.4.1 Card Data Flow Tracking Across Multiple VMs Hosted on the Same Physical Host 68
3.4.1.1 Experimental Setup 68

3.4.1.2 Experiments 68
3.4.2 Card Data Flow Tracking Across Multiple VMs Hosted on Multiple Physical Hosts 71
3.4.2.1 Experimental Setup 72
3.4.2.2 Experiments 72
3.5 Limitations 73
3.6 Summary 74
4 Cloud-Based Application Whitelisting 75
4.1 Introduction 75
4.2 Background 76
4.2.1 Code Regions 77
4.2.1.1 Code in File-Backed Address Space Regions 78
4.2.1.2 Code in Private Address Space Regions 80
4.2.1.3 Other 82
4.2.2 Relevant Kernel Data Structures 82
4.2.2.1 Windows OS 82
4.2.2.2 Linux OS 86
4.2.2.3 System Call Table Structures 88
4.2.3 System Call Interception in Xen Hypervisor 89
4.3 Related Work 90
4.3.1 Code Verification Systems 90
4.3.1.1 Periodic Code Verification 91
4.3.1.2 Continuous Run-Time Code Verification 92
4.3.1.3 On-Demand Code Verification 94
4.3.2 System Call Interception Systems 95
4.3.2.1 Hardware Emulators 96
4.3.2.2 Para-Virtualized Systems 97

4.3.2.3 Fully Virtualized Systems 98
4.4 System Architecture 98
4.4.1 Overview 98
4.4.2 Design and Implementation 101
4.4.2.1 Verification of Code in File-Backed Space 101
4.4.2.2 Verification of Code in Private Space 103
4.4.3 Key Data Structures Monitored by CLAW 106
4.5 Evaluation 107
4.5.1 Experimental Setup 107
4.5.2 Experiments 108
4.5.2.1 Effectiveness 108
4.5.2.2 Performance 108
4.6 Limitations 112
4.7 Summary 113
5 Access Token Manipulation Attack Detection Tool 114
5.1 Introduction 114
5.2 Background 116
5.2.1 Access Token Data Structure 116
5.2.2 Token Manipulation Attacks 121
5.2.2.1 Access Token Patching 121
5.2.2.2 Access Token Stealing 125
5.3 Related Work 127
5.3.1 Control Data Manipulating Rootkits 127
5.3.1.1 Static Control Data Monitoring 127
5.3.1.2 Execution Path Monitoring 129
5.3.2 Non-Control Dynamic Data Manipulating Rootkits 131
5.3.2.1 Periodic Checks 131

5.3.2.2 Continuous Monitoring 132
5.3.3 Summary of Methods 134
5.4 System Architecture 136
5.4.1 Overview 136
5.4.2 System Design and Implementation 137
5.4.2.1 Creation of a New Process 137
5.4.2.2 Access Token Analysis 138
5.4.2.3 Access Token Monitoring 140
5.4.3 “Always-on” and “Periodic Polling” Monitoring Modes 142
5.4.4 ATOM Implementation for Linux OS 143
5.4.4.1 Background 143
5.4.4.2 Implementation 145
5.4.5 Summary of Data Structures Monitored by ATOM 147
5.5 Evaluations 147
5.5.1 Experimental Setup 147
5.5.2 Experiments 148
5.5.2.1 Effectiveness 148
5.5.2.2 Performance Assessment 150
5.6 Summary 150
6 Conclusions and Future Work 152
6.1 Conclusions 152
6.1.1 RTKDSM 153
6.1.2 vCardTrek 153
6.1.3 CLAW and ATOM 154
6.2 Future Work 155
6.2.1 RTKDSM 155
6.2.2 ATOM and CLAW 156

6.2.3 vCardTrek 156

Bibliography 157

List of Tables

Table 2.1 VMI frameworks summary. 11

Table 2.2 Real-time data structure monitoring systems summary. 12

Table 2.3 Examples of a Volatility profile, a Volatility object, and a Volatility plugin. 21

Table 2.4 Data structures used in the experiments. 37

Table 2.5 Page faults on pages containing the PsActiveProcessHead, TCBTable, and init_task structures in the idle Windows VM #1 and Linux VM # 1 recorded during 1 minute. 37

Table 2.6 Page faults on pages containing EPROCESS structures for the 50 calc.exe test processes in the idle Windows VM #1 recorded during 1 minute. 38

Table 2.7 Page faults on pages containing ETHREAD structures for the 50 calc.exe test processes in the idle Windows VM #1 recorded during 1 minute. 39

Table 2.8 Page faults on pages containing TOKEN structures for the 50 calc.exe test processes in the idle Windows VM #1 recorded during 1 minute. 39

Table 2.9 Page faults on pages containing PEB_LDR_DATA structures for the 50 calc.exe test processes in the idle Windows VM #1 recorded during 1 minute. 40

Table 2.10 Page faults on pages containing task_struct structures for the 50 gcalctool test processes in the idle Linux VM #1 recorded during 1 minute. 40

Table 2.11 Page faults on pages containing files_struct structures for the 50 gcalctool test processes in the idle Linux VM #1 recorded during 1 minute. 41

Table 2.12 Performance in the “always-on” mode using the PCMark05 benchmark in Windows OS. 44

Table 2.13 Performance in the “always-on” mode using the Apache benchmark in Windows OS. 45

Table 2.14 Performance in the “always-on” mode using the NBench & gzip benchmarks in Linux OS. 46

Table 2.15 Performance in the “always-on” mode using the Apache benchmark in Linux OS… 47

Table 2.16 Performance in the “periodic polling” mode for the PsActiveProcessHead, TCBTable, and init_task data structures. 49

Table 2.17 Performance in the “periodic polling” mode for the EPROCESS data structure. 49

Table 2.18 Performance in the “periodic polling” mode for the ETHREAD data structure. 50

Table 2.19 Performance in the “periodic polling” mode for the TOKEN data structure. 50

Table 2.20 Performance in the “periodic polling” mode for the task_struct data structure. 51

Table 2.21 Performance in the “periodic polling” mode for the files_struct data structure. 51

Table 3.1 The data structures accessed by vCardTrek. 63

Table 3.2 Evaluation suites and testing results. 67

Table 4.1 Code source types in memory. 77

Table 4.2 Windows system calls. 79

Table 4.3 Linux system calls. 80

Table 4.4 Code verification systems. 91

Table 4.5 Comparison of system call monitoring systems. 96

Table 4.6 Summary of the data structures monitored by CLAW. 107

Table 4.7 Run-time performance of CLAW. 109

Table 4.8 Startup performance of CLAW 110

Table 4.9 Run-time performance of NtCreateSection and NtMapViewOfSection system call interception. 111

Table 4.10 Startup performance of CLAW using NtCreateSection and NtMapViewOfSection system call interception 112

Table 5.1 Non-control data manipulating rootkit detection systems. 136

Table 5.2 Summary of the data structures monitored by ATOM. 148

Table 6.1 Summary of the data structures used by vCardTrek, CLAW, and ATOM. 154

List of Figures

Figure 1.1 The RTKDSM system provides the underlying interfaces for the development of vCardTrek, CLAW, and ATOM. 4

Figure 2.1 Logical layout and workflow of the system. 24

Figure 2.2 System implementation. 28

Figure 2.3 Windows OS test environment 35

Figure 2.4 A sample Windows OS command script to invoke 10 processes. 36

Figure 3.1 (1) Inter-VM network communications are tracked by vCardTrek, and (2) the memory of the interacting processes is inspected for card data. 60

Figure 3.2 Card data flow concatenation from multiple physical hosts. 61

Figure 3.3 (1-2-3) Network connections are intercepted, and the processes participating in the network connections are determined; (4-5) the memory of the identified processes is searched for card data, and the card data flow is reconstructed. 62

Figure 3.4 (A) 4 possible states of two inter-VM communicating processes (grey rectangle - the card number found in process memory, white rectangle - no card number found in process memory. The arrow indicates the direction of connection initiation, not traffic flow); (B) 4 possible states of processes within a VM at packet receiving time and at packet sending (the same process may serve as the receiving and sending process). 66

Figure 3.5 Processes involved in card data flow (CreditLine flow at the top, osCommerce flow in the middle, and AbleCommerce flow at the bottom). 69

Figure 3.6 AbleCommerce Card Data Flow (machines found to participate in the card data flow are shown in grey) (left) using vCardTrek; (right) using a packet sniffer. 69

Figure 3.7 osCommerce Card Data Flow (machines found to participate in the card data flow are shown in grey) (left) using vCardTrek; (right) using a packet sniffer. 69

Figure 3.8 CreditLine Card Data Flow (machines found to participate in the card data flow are shown in grey) (left) using vCardTrek; (right) using a packet sniffer. 70

Figure 3.9 Detailed information uncovered about a test card, including the card number (4556156372833798), the card expiration date (0412), the CVV number (354), and the cardholder's name (Jon Jones) were identified within the process memory. 70

Figure 3.10 Card data flow across multiple VMs hosted on multiple physical hosts. 73

Figure 4.1 Windows code and memory management data structures. 83

Figure 4.2 The PEB data structure. 84

Figure 4.3 Linux code and memory management data structures. 87

Figure 4.4 System call dispatching. 89

Figure 4.5 The CLAW architecture. 99

Figure 4.6 Windows OS process creation flow. 101

Figure 4.7 The CLAW system call interception steps – we enable/disable the present bit on system call entry/return. 105

Figure 4.8 Combination of the CLAW and the MSR-register based system call interception... 106

Figure 5.1 _EPROCESS data structure. 116

Figure 5.2 EX_FAST_REF data structure. 117

Figure 5.3 TOKEN data structure in Windows XP. 117

Figure 5.4 Static and variable parts of the token in Windows XP [70]. 118

Figure 5.5 SID_AND_ATTRIBUTES and SID data structures. 119

Figure 5.6 LUID_AND_ATTRIBUTES data structure. 120

Figure 5.7 SIDs and Privileges contained in the process's access token using Sysinternals' Process Explorer tool [71]. 120

Figure 5.8 TOKEN data structure in Windows Vista. 124

Figure 5.9 _SID_AND_ATTRIBUTES_HASH data structure. 124

Figure 5.10 Impersonation levels. 125

Figure 5.11 The ATOM architecture. 138

Figure 5.12 Windows OS process creation flow. 138

Figure 5.13 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList key hive 139

Figure 5.14 _SEP_TOKEN_PRIVILEGES structure in Windows Vista and later Windows versions 141

Figure 5.15 The task_struct data structure. 143

Figure 5.16 The task_struct and cred data structures in Linux kernel versions >= 2.6.29. 144

Figure 5.17 Rootkit attacks on process credentials in Linux OS. 145

Acknowledgements

First and foremost, I want to express my deepest gratitude to my advisor, Dr. Chiueh, for his patience, thoughtfulness, and a sharp eye for detail. His guidance and mentorship were of the highest quality. Without his support, this dissertation would have never come into being. I strive to achieve the same brilliance of mind and the ability for great visions.

I am indebted and thankful to my committee members, Dr. Gao, Dr. Murdoch, and Dr. Stoller for their support, advice, and helpful insights. Their time and contributions are much appreciated.

This achievement was only possible because of my parents, Maria and Roman, whose love, support, and understanding have carried me through life. Thank you for believing in me!

To my family, Genevieve and Ilia, who have supported me through good times and bad.

Publications

[1] J. Hizver and T. Chiueh. Cloud-based application whitelisting. In *Proceedings of the 6th IEEE International Conference on Cloud Computing (CLOUD 2013)*, pages 636-643, July 2013.

[2] J. Hizver and T. Chiueh. Tracking payment card data flow using virtual machine state introspection. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC 2011)*, pages 277-285, December 2011. ISBN: 978-1-4503-0672-0. doi:10.1145/2076732.2076771. http://dl.acm.org/citation.cfm?id=2076771.

[3] J. Hizver and T. Chiueh. Automated discovery of credit card data flow for PCI DSS compliance. In *Proceedings of the 2011 IEEE 30th International Symposium on Reliable Distributed Systems (SRDS 2011)*, pages 51-58, July 2011. ISBN: 978-0-7695-4450-2. doi:10.1109/SRDS.2011.15. http://dl.acm.org/citation.cfm?id=2085039.2085350.

[4] J. Hizver and T. Chiueh. An introspection-based memory scraper attack against virtualized point of sale systems. In *Proceedings of the 2011 International Conference on Financial Cryptography and Data Security (FC 2011)*, Lecture Notes in Computer Science, vol. 7126, pages 55-69, March 2011. ISBN:978-3-642-29888-2.doi:10.1007/978-3-642-29889-9_6.http://dl.acm.org/citation.cfm?id=2341444.2341451.

1 Introduction

1.1 Motivation and Challenges

Cloud computing ushers in an era of consolidated information technology infrastructure that is elastic, available, and scalable. Virtualization is a critical building block in this evolution enabling multiplexing of the underlying computing resources. With the growth of virtualization, re-design of traditional agent-based monitoring technologies is underway by moving monitoring functionalities out of virtual machines (VMs) to delegate responsibilities to automated services in the cloud using the virtual machine introspection (VMI) technology. The cloud computing industry has witnessed a growing adoption of the VMI technology for building a wide range of agentless tools including intrusion detection systems, virtual firewalls, malware analysis, and live memory forensics [1-3]. In the agentless approach, users can focus on using their VMs without the burden of monitoring VM operations. Furthermore, such approach de-couples the monitoring system from the monolithic OS and eliminates the need for homogeneous environments where every VM runs a common monitoring suite.

VMI was first introduced to describe the operation of the Livewire intrusion detection system [4], which was placed in a special management VM isolated from the other VMs to observe their execution. Using the VMI approach, the management VM reconstructs the internal state of the monitored VMs through low level information, such as memory pages. Access to this information is possible because the hypervisor on which the management VM runs has complete access to all memory in the monitored VMs and can read it as needed. Given a VM's entire physical memory, it is possible for a VMI application in the management VM to access the

contents of the monitored VM's kernel and user-space memory and to extract the memory-resident critical OS data structures. From these data structures, the VMI application can then infer exactly what the OS is doing.

While enabling the implementation of centralized agentless monitoring architectures, VMI has to overcome the so-called semantic gap to providing efficient monitoring of VMs. Since native OS application programming interfaces (APIs) are not available to VMI, the low semantic level in which data are captured by the hypervisor makes it difficult to render the OS high-level semantic views needed to make decisions. Given the low level VM views, the first step in overcoming the semantic gap is to gather information about the state of the OS by locating and examining the internal data structures that the in-guest APIs use. This step generally requires tedious, prolonged, and error-prone efforts to accurately translate the acquired low level views to the OS structures in the VM. The process is particularly challenging in closed source OSes such as Windows, where details of data structures must be obtained using reverse engineering. Even for systems where the OS source code is available, reconstructing data structures can be an overwhelming task. Moreover, the time and efforts spent reverse-engineering the internals of one OS version may not be applicable to future versions. The lack of automated VMI frameworks that aggregate the underlying data structure knowledge of multiple OS flavors and versions to eliminate reverse-engineering efforts presently poses a significant challenge for developers of VMI applications.

This work contributes toward the goal of providing automated frameworks for development of VMI applications.

1.2 Dissertation Contributions

1.2.1 Real-Time Kernel Data Structure Monitoring System

In Chapter 2, we present the real-time kernel data structure monitoring (RTKDSM) system that allows developers of VMI applications to perform real-time analysis and monitoring of OS data structures in a VM memory. We demonstrate how applying the vast data structure knowledge in an existing open source computer forensics platform enables the development of VMI tools to proceed more rapidly and with significant reduction in effort. Our system does not require VMI application developers to know the version of the guest OS in advance, since it is determined on the fly by the framework, nor does it require access to the OS source code, making it also suitable for real-world production execution environments.

The RTKDSM system is able to identify at run-time data structures of interest in memory of monitored VMs and to react to changes in those data structures. Responding to changes effectively in real-time requires the system to react to a potentially large volume of events impacting system performance. As VMI developers may need to track changes to rapidly changing data structures or to a large number of data structures, we introduce a performance optimization technique to reduce the monitoring overhead.

To demonstrate the applicability of the RTKDSM system, we developed three agentless monitoring systems: payment card data flow tracking tool (vCardTrek), cloud-based application whitelisting solution (CLAW), and access token manipulation detection tool (ATOM) (Figure 1.1). These systems are expected to contribute to enhanced monitoring in cloud computing centers.

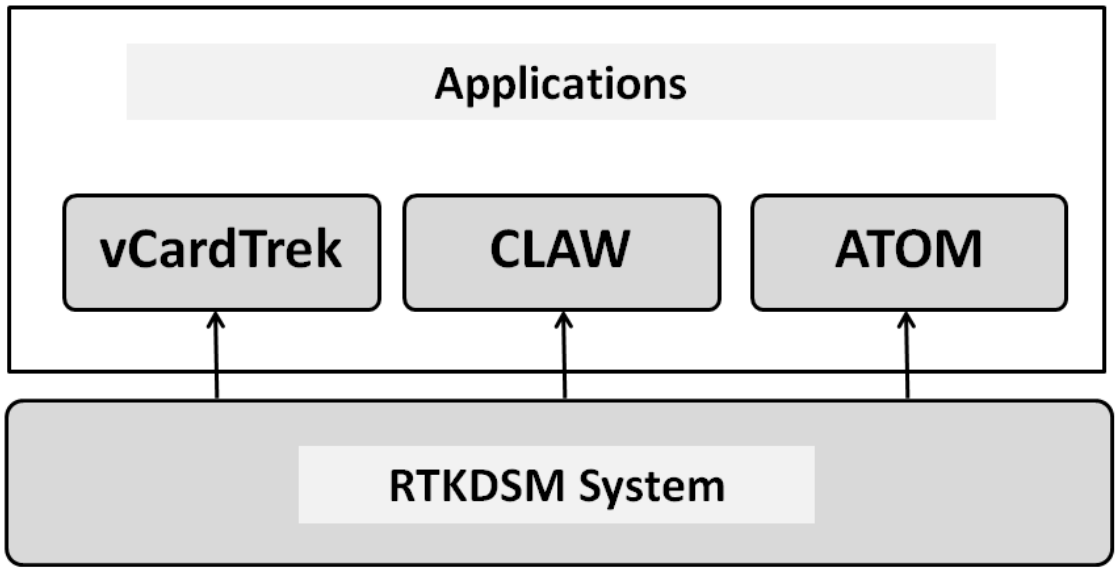

Figure 1.1 The RTKDSM system provides the underlying interfaces for the development of vCardTrek, CLAW, and ATOM.

1.2.2 Payment Card Data Flow Tracking Tool

Credit and debit card payment processing systems are key elements in financial transactions. Negligence in securing these systems makes them vulnerable to hacking attacks, which may lead to significant monetary losses for both merchants and the financial organizations. To reduce this risk, mandatory security compliance regulations, such as the Payment Card Industry Data Security Standard (PCI DSS), were developed and adopted by the industry. A key pre-requisite of the PCI DSS compliance process is the ability to identify the components of the payment systems directly involved with the card data (i.e. process, transmit, or store). However, existing data flow tracking tools cannot fully automate the process of identifying system components that interact with card data, because they either can not examine encrypted communications or they use an instrumentation-based approach and thus require a

priori detailed knowledge of the payment card processing systems.

In Chapter 3, we describe the implementation and evaluation of a novel tool called vCardTrek to identify the card data flow in commercial payment card processing systems running on virtualized servers. vCardTrek performs real-time monitoring of network communications between virtual machines and inspects the memory of the communicating processes for unencrypted card data. Our implementation can accurately identify the system components involved in card data flow even when the communications among system components are encrypted. Effectiveness of this tool is demonstrated through its successful discovery of the card data flow of several open- and closed-source payment card processing applications.

1.2.3 Cloud-Based Application Whitelisting

In Chapter 4, we present a cloud-based application whitelisting system called CLAW, which leverages the centralized monitoring capability of the VMI technology to guarantee that only application binaries in a pre-approved set are allowed to run in each VM under its management. By applying the RTKDSM system, CLAW performs its security policy enforcement without installing any agents inside the monitored VMs. We describe the key techniques in the design and implementation of CLAW and compare them with previous hypervisor-based application whitelisting systems.

1.2.4 Access Token Manipulation Detection Tool

The direct kernel object manipulation (DKOM) technique is used by hackers to manipulate OS-critical data structures without the use of application programming interfaces (APIs). Rootkits often use this technique to hide their presence by manipulating data structures

of running processes. In a similar DKOM attack, called access token manipulation, rootkits escalate process privileges by overwriting the malicious process's privileges with those of a more privileged account.

In Chapter 5, we present the design, implementation, and evaluation of an access token manipulation detection tool called ATOM. ATOM performs real-time monitoring of the running processes' access tokens storing process privileges and is able to detect attacks on access tokens for privilege escalation purposes. A key design decision of ATOM was to apply the RTKDSM system to monitor the access tokens' states. Effectiveness of the tool was demonstrated through its successful discovery of real world access token manipulation attacks.

2 Real-Time Kernel Data Structure Monitoring System

2.1 Introduction

VMI systems fall into one of the two categories: those that are semantically aware and those that are semantically unaware. Semantic awareness capability indicates whether a VMI system seeks to extract different OS characteristics to carry out its operations. For instance, a semantically aware VMI system may parse VM memory to build a list of running processes and to obtain process-specific information. Semantically unaware VMI systems are largely unaware of the OS semantics associated with the VMs they manage.

In this study, we present a real-time kernel data structure monitoring (RTKDSM) system for use by semantically aware VMI applications. The RTKDSM system that has the ability to automatically identify OS kernel data structures and continuously track all changes that occur to the data structures marked as structures of interest by semantically aware VMI systems. The RTKDSM system is designed as a modular component suitable for integration into VMI tools to ensure continuous monitoring of critical data. We have implemented a working prototype of the RTKDSM system for the Xen hypervisor.

2.2 Related Work

2.2.1 Semantically Aware Systems

The availability of VMI firstly triggered the development of security monitoring systems, which were mainly divided into passive and active monitoring systems. Passive monitoring systems were only able to inspect a VM and report an attack instead of preventing it [3-6]. Conversely, active monitoring systems interposed on events of interest within the monitored VM

and prevented malicious acts instead of relying on mere detection [2, 7, 8]. While some of these VMI systems were entirely agentless, others bridged part of the semantic gap by placing components inside the monitored VM.

Livewire, the first host-based intrusion detection system, monitored VMs to gather information and detect attacks from within the monitored VM by acquiring semantic awareness through analysis of kernel dumps [4].

Another semantically aware system, Lares, inserted internal "hooks" into the monitored VM that activated an external monitoring control upon execution [2]. The monitor interrupted execution and passed control to a security mechanism to deliver understanding of the OS's semantics.

VMwatcher demonstrated how VMI could be used for passive out-of-VM anti-virus monitoring [3]. VMwatcher reconstructed OS states from a snapshot a VM memory. The authors presented a detailed description of how the OS states were reconstructed that clearly highlighted both the need for expert knowledge of the OS to implement a VMI system and the fragility of the approach to changes of the OS.

VMwall application-level firewall executed outside of the VM and intercepted network connections to and from the hosted VMs [1]. It applied VMI to correlate each flow to sending/receiving processes through extraction of process and socket data structures, and used predefined policies to decide whether a connection should be allowed.

2.2.2 Semantically Unaware Systems

AntFarm was specifically designed to monitor a VM's memory management unit (MMU)

to infer information about the VM's processes and OS [9]. AntFarm was semantically unaware of the monitored system but built up such awareness over time.

LycosID system used cross-view validation techniques to compare running processes visible from high and low abstraction layers [8]. The system then patched running code to enable reliable identification of hidden processes. No detailed implementation information about the monitored OS, such as versions and patch levels of the target OS, was required.

Manitou, a VMI system designed to detect malware, compared known instruction-page hashes with memory-page hashes at runtime [7]. If no match was found, the instruction page was considered corrupted and marked as non-executable. Similarly, Patagonix, a system that ensured no binary code could be covertly executed on the monitored system, used the processor MMU to receive notifications whenever binary code was executed and identified the code using the binary format specification [10]. Unrecognized code, whether malicious or in a form that could not be understood, was reported by the system. The Patagonix approach was OS-agnostic so long as an executable file format could be understood by the monitor and the executed code could be identified.

2.2.3 VMI Frameworks For Semantically Aware Systems

Several research studies have attempted to developed frameworks to make it easier for researchers to experiment with the many uses of VMI without focusing on low-level details.

XenAccess framework was developed as a monitoring library for the Xen hypervisor [6]. The purpose of this library was to provide memory and disk monitoring capabilities for both open source and closed source OSes. XenAccess library required the kernel symbol and address information associated with the guest OS to perform memory mapping and conversions. The

symbol information was sensitive to the guest OS and was not very portable. XenAccess was only able to generate a list of running processes and loaded modules. XenAccess was later extended to create the LibVMI library to provide introspection functions for reading and writing memory in multiple virtualization platforms [11].

Hay and Nance created the VIX tools to perform forensic analysis of VMs running on Xen [12, 13]. The VIX tools were designed to allow a forensic investigator to perform live analysis of a VM system. VIX consisted of a library of common functions and a suite of tools which mimicked the behavior of common Unix command line utilities, such as ps, lsmod, netstat, lsof, who, and top. Using VIX, unobtrusive live system analysis was performed on the target VM without changing the system state during the data acquisition process.

A whole-system binary code extractor, called Virtuoso, generated out-of-box code for use in VMI [14]. Using Virtuoso, developers could create VMI programs to monitor VMs running a variety of different OSes.

In another study, a novel technique called process implanting was proposed to narrow the semantic gap by implanting a process into the monitored VM and executing it under the cover of an existing running process to bridge the semantic gap between the VMI application and the monitored VM [15]. With the protection and coordination from the hypervisor, the implanted process ran with a degree of stealthiness and exited gracefully without leaving negative impact on the VM. The downside of this approach was that any reliance on functionality on the monitored VM ran the risk of deception by malware present in that VM, as if the implanted process were running as a process on the VM itself.

Table 2.1 summarizes the existing VMI frameworks and compares the RTKDSM system

with the described VMI frameworks.

Table 2.1 VMI frameworks summary.

System	Detection of Changes	Exposed to the Monitored OS	Built on the Existing Forensic Framework
RTKDSM System	Synchronous	No	Yes
XenAccess	Asynchronous	No	No
VIX	Asynchronous	No	No
Virtuoso	Asynchronous	No	No
Process Implanting	Synchronous	Yes	No

2.2.4 Real-Time Data Structure Monitoring Systems

A number of studies have developed out-of-VM real-time data structure monitors to detect integrity violations. Table 2.2 compares the RTKDSM system to these monitors.

Petroni et al. [16] proposed a framework for detecting attacks against dynamic kernel data structures using a coprocessor-based external monitor. The monitoring system periodically compared actual observed dynamic kernel data structures in the snapshots of kernel memory with specifications of correct kernel data structures and reported any semantic integrity violations against the kernel's dynamic data. The data structure extractions were performed asynchronously with the monitored system's execution. The asynchronous nature of this processing rendered this approach vulnerable to dynamic data attacks launched and withdrawn between snapshot periods. On the contrary, the system developed in this study is able to extract and analyze the data structures synchronously, overcoming the limitation of the coprocessor-based approach.

Table 2.2 Real-time data structure monitoring systems summary.

System	Detection Mode	Monitoring	Exposed to monitored OS	Supports closed source OSes	Supports HVM
RTKDSM	Synchronous	Passive	No	Yes	Yes
Petroni et al.	Asynchronous	Passive	No	Yes	No
Sentry	Synchronous	Active	Yes	No	No
Rhee et al.	Synchronous	Active	No	Yes	No

In another related study, Srivastava et al. [18] developed Sentry, a VM-based system that prevented illegitimate changes to critical kernel data structures. Sentry's memory protection required modifications to the monitored OS to identify locations of dynamically-allocated kernel data objects. Code instrumentations were introduced within the monitored OS's kernel to activate and deactivate protections on kernel object construction and destruction. The instrumentation passed the physical page frame number (PFN) of the newly allocated memory page holding a kernel data structure requiring protection to the hypervisor. When the memory protection module in the hypervisor received a request to add protection for the monitored VM's page, it added the PFN to a list of protected pages and removed the page's write permission causing page faults on all attempted kernel object alterations. Sentry allowed only those alterations invoked by legitimate kernel functionality. Sentry assumed that existing techniques could protect the core kernel code's integrity, so an attacker would not be able to remove the instrumentation. The system required the OS source code in order to partition a structure into secure and insecure parts. This kind of protection was difficult to design for a closed source operating system such as Windows. Compared Sentry, the RTKDSM system offers an advantage of not requiring modifications to the monitored OS.

Rhee et al. [17] proposed a solution to prevent dynamic rootkit attacks on kernel data structures using QEMU emulator as an external monitor. The system monitored the execution of the OS at the instruction level within QEMU. At runtime, the system identified data structures in memory and intercepted all writes to their address ranges. The system relied on writing a policy that described how the monitor should identify the data structure in a raw memory as well as the characteristics of an attack against the data structure. Only limited details were given regarding the data structures extraction mechanisms used by the system. The methodology described in the study was only portable to VM monitors that supported memory interposition to translate guest instructions into host instructions. Unlike in the RTKDSM system, such methodology could not be extended to support commercial hypervisors that did not support memory interposition, such as Xen and VMWare ESX.

2.3 Background

2.3.1 Xen Hypervisor

The RTKDSM architecture is designed and implemented using the popular open-source Xen hypervisor [19, 20] capable of supporting multiple types of guest OSes, including Windows and Linux. This section gives an overview of Xen and describes concepts used in our prototype implementation.

The Xen hypervisor is the lowest and most privileged software layer, which is added to a single physical machine to abstract the underlying hardware by creating multiple interfaces to VMs. To present a VM with the illusion that it is running on the bare hardware, the hypervisor dynamically partitions and shares the available physical resources such as CPU, memory,

network connections, and I/O devices among multiple concurrently running VMs. The operating system and software applications are executed on top of the VMs.

The first VM, which boots automatically after the Xen hypervisor is loaded, is called the Dom0 domain. The Dom0 domain is typically a modified version of UNIX operating system. By default, Dom0 is granted special privileges for managing and controlling other VMs including access to the raw memory of other VMs known as DomU domains. DomUs may either be unmodified closed-source OSes, if the host processor supports x86 virtualization (hardware assisted virtualization) or modified OSes with special drivers that support Xen features (para-virtualization). Hardware assisted virtualization approach uses help from hardware capabilities developed by Intel (VT-x hardware) and AMD (AMD-V hardware). This technology made virtualization of closed-source OSes possible without requiring modifications to the guest OS. Para-virtualization is the technique whereby the hypervisor and the OS running in a VM communicate through hypercalls. This technique requires modifications to the guest OS to introduce the hypercalls.

This study focuses on Hardware Virtual Machines (HVM), which utilize hardware assisted virtualization technology.

2.3.2 Dirty Page Tracking

To perform real-time monitoring of kernel data structures, the RTKDSM system builds on top of the existing log dirty mode technique and the shadow paging technique.

2.3.2.1 Shadow Paging Technique

In the shadow paging technique, Xen maintains two versions of page tables for each VM:

guest OS page tables controlled by the guest OS and shadow page tables controlled by the hypervisor. The guest OS translates virtual addresses into physical addresses of the VM via its guest page tables. The real page tables, exposed to the hardware MMU, are shadow page tables maintained by the hypervisor. The structure of shadow page table is the same as the guest page table. To avoid an extra level of indirection on every memory access, the shadow page tables map directly from the guest virtual addresses into the hardware machine addresses. Each shadow entry is created on-demand according to the guest page table entry. The hypervisor detects all modifications to the guest page tables and ensures that the shadow page table entries being used by the hardware for translations correspond to those of the guest OS environment. This is commonly done by write protecting the guest page tables and trapping any access to the page table pointer by a guest VM. When an entry is added or changed in a guest page table, Xen translates the physical address into its corresponding machine address, performs any necessary adjustments, and then updates the corresponding shadow page table. This process is called page table entry (PTE) propagation.

2.3.2.2 Log Dirty Mode

The Xen's log dirty mode capability was originally designed for live VM migration to track dirty memory pages between consecutive migration rounds. VM live migration employs an iterative copy mechanism to ease performance degradation during migration. In the first iteration, all the VM pages are transferred to the designated host without pausing the VM. Subsequent iterations copy only those pages dirtied during the previous transfer phase. To do so, the hypervisor enables the log dirty mode of the shadow page tables to record dirty pages. The principle of the log dirty mode is as follows. Initially, all the shadow entries are marked as read-only, regardless of the permission of its associated guest entries. When the guest OS attempts to

modify a memory page, a shadow page write-fault occurs and is intercepted by the hypervisor. If the write is permitted by its associated guest entry, the hypervisor grants write permission to the shadow entry and marks the page as a dirty one accordingly. Subsequent write accesses to this page do not incur any shadow page faults in the current round.

2.3.3 Forensic Memory Analysis

The field of memory analysis first became popular within the digital forensics community. Forensic monitoring and analysis occurs after a system is known to have been attacked. Instead of detecting or preventing an attack, the goals in this case are to learn more about what happened during the attack. Memory snapshots of a running system are taken and analyzed post-intrusion to determine details about the activities happening on the machine at the time of the snapshot.

The memory analysis has evolved from a basic technique, such as string matching, to more complex methods, such as list traversal [6, 21, 22] and signature-based scanning [23-26]. The list traversal method works by looking at hard-coded locations and offset values to identify the well-known key data structures and using these data structures to derive other data structures by traversing linked lists. Often, for a given version of an OS or application software these hard-coded locations and offset values are consistent on different machines and at different times. Finding the appropriate values in the first place typically involves reverse engineering, source code analysis, or vendor-provided debugging symbols. Conversely, signature-based scanning involves a linear scan of physical memory looking for a constant pattern of bytes using known signatures. For instance, some Windows data structures are tagged with a four byte ASCII identifier as well as size information and therefore can be easily found in memory using

signature-based scanning.

2.3.3.1 Volatility Framework

The runtime state information accessed using the RTKDSM system is memory as it stores current OS states of the system in OS data structures. Our system utilizes the open-source Python-based Volatility forensic memory analysis framework for extraction and analysis of such data structures in the monitored VM memory [22]. Volatility supports the following operating systems and versions:

- Windows
    - 32-bit Windows XP Service Pack 2 and 3
    - 32-bit Windows 2003 Server Service Pack 0, 1, 2
    - 32-bit Windows Vista Service Pack 0, 1, 2
    - 32-bit Windows 2008 Server Service Pack 1, 2
    - 32-bit Windows 7 Service Pack 0, 1
    - 64-bit Windows XP Service Pack 1 and 2
    - 64-bit Windows 2003 Server Service Pack 1 and 2
    - 64-bit Windows Vista Service Pack 0, 1, 2
    - 64-bit Windows 2008 Server Service Pack 1 and 2
    - 64-bit Windows 2008 R2 Server Service Pack 0 and 1
    - 64-bit Windows 7 Service Pack 0 and 1
    - Image Identification
    - Processes and DLLs
    - Process Memory
    - Kernel Memory and Objects

    - Networking
    - Registry
    - Malware/Rootkits
    - Win32k / GUI Memory
    - File Formats
    - File System
    - Miscellaneous
- Linux
    - 32-bit Linux kernels 2.6.11 to 3.5
    - 64-bit Linux kernels 2.6.11 to 3.5
    - OpenSuSE, Ubuntu, Debian, CentOS, Fedora, Mandriva, etc.
- Mac OSX
    - 32-bit 10.5.x Leopard
    - 32-bit 10.6.x Snow Leopard
    - 64-bit 10.6.x Snow Leopard
    - 32-bit 10.7.x Lion
    - 64-bit 10.7.x Lion
    - 64-bit 10.8.x Mountain Lion

Volatility is a modular framework in which most of the functionality is implemented by *plugins* performing a certain function, such as identifying a list of running processes. Plugins are declared as Python classes by extending base Volatility classes. When using Volatility as a library, it can be extended by new plugins from within one's code without embedding them into the library itself. Volatility currently includes over 100 known plug-ins divided info the following major groups:

- Windows

    - Image Identification
    - Processes and DLLs
    - Process Memory
    - Kernel Memory and Objects
    - Networking
    - Registry
    - Malware/Rootkits
    - Win32k / GUI Memory
    - File Formats
    - File System
    - Miscellaneous
- Linux / Mac OSX / Android
    - Processes
    - Process Memory
    - Kernel Memory and Objects
    - Networking
    - Malware/Rootkits
    - System Information
    - Miscellaneous

Volatility provides support for a variety of processor architectures through the use of *address spaces* (AS) intended to abstract the handling of different memory images and formats and to facilitate random access to a memory image by a plugin. A valid AS for a given memory image is derived by Volatility automatically. The derived AS is used to satisfy a read request by a plugin. Exactly how the read request is satisfied is not important to the plugin code, so long as the read request is satisfied. Volatility supports the following ASes:

- FileAddressSpace - direct file AS

- Legacy Intel x86 AS
    - IA32PagedMemoryPae
    - IA32PagedMemory
- Standard Intel x86 AS
    - JKIA32PagedMemoryPae
    - JKIA32PagedMemory
- AMD64PagedMemory - AMD 64-bit AS
- WindowsCrashDumpSpace32 - this AS supports windows Crash Dump format (x86)
- WindowsCrashDumpSpace64 - this AS supports windows Crash Dump format (x64)
- WindowsHiberFileSpace32 - this AS supports windows hibernation files (x86 and x64)
- EWFAddressSpace - this AS supports expert witness (EWF) files
- FirewireAddressSpace - this AS supports direct memory access over firewire
- LimeAddressSpace - this AS supports LiME (Linux Memory Extractor)
- MachOAddressSpace - this AS supports 32- and 64-bit Mac OSX memory dumps
- ArmAddressSpace - this AS supports memory dumps from 32-bit ARM
- VirtualBoxCoreDumpElf64 - this AS supports memory dumps from VirtualBox virtual machines
- VMware Snapshot - this AS supports VMware saved state and VMware snapshot files

Once an AS is loaded, most plugins begin accessing data structures (*objects*) within the AS. Objects are declared as Python classes by extending the base object classes. Any time that data are needed from an AS, it will usually be accessed through an object. Examples of objects include EPROCESS and ETHREAD objects corresponding to the process and thread in Windows OS. Volatility's object manager parses objects using *profiles*, which are collections of data structure definitions (member fields and offsets) relating to a certain OS.

Concrete examples of an object, a profile, and a plugin are given in Table 2.3. The *profile* defines the _MYOBJECT data structure, which is 0x4 bytes long and has only one field, Id, at the offset 0x0 within the data structure. The corresponding *object* is declared as the _MYOBJECT class. This class has one member function *getID*, which returns the value of the field Id. The MyPlugin *plugin* defines the *calculate* function that carries out the main operation against a memory image being analyzed. This function acquires a valid address space and yields Ids for all _MYOBJECT objects carved from the address space. The plugin assumes there is already a Volatility API imported as myobjects.list to produce all _MYOBJECT objects. These results are processed by the plugin's *render* function rendering the output in a text form. The *render* function accepts the object Ids (data) yielded by the *calculate* function.

Table 2.3 Examples of a Volatility profile, a Volatility object, and a Volatility plugin.

PROFILE	OBJECT	PLUGIN

'_MYOBJECT' : [0x4, {
'Id' : [0x0, ['unsigned long']],
}]
	import volatility.obj as obj
	
	class _MYOBJECT(obj.CType):
	def getID(self):
	return self.Id
		import volatility.plugins.common as common
		import volatility.utils as utils
		import volatility.obj as obj
		import volatility.win32.myobjects as myobjects
		
		class MyPlugin(common.AbstractWindowsCommand):
		def calculate(self):
		address_space = utils.load_as(self._config)
		for myobject in myobjects.list(address_space)
		yield myobject.getID()
		def render_text(self, outfd, data):
		for id in data:
		outfd.write("Id: {0}\n".format(ID))

2.3.3.2 Data Structure Classification

We classify OS kernel data structures into two types:

1) Global data structures created at the system initialization time and located at fixed offsets.

   Typically, there are small numbers of global data structures of particular types per an OS instance. Examples include the System Service Descriptor Table (SSDT), Kernel Debugger Block (KDBG) and Kernel Processor Control Region (KPCR) structures in Windows OS. We further classify these data structures into static and dynamic. Static global data structures do not change at run-time. Field values within dynamic global data structures may be updated by the system during its run-time.

2) Dynamically created data structures generated by the system post-initialization at run-time.

   Numbers of such data structures per OS instance may widely very during the system run-time. Examples include: EPROCESS (process), ETHREAD (thread), TOKEN (process access token), ADDRESS_OBJECT (socket), TCPT_OBJECT (connection), and FILE_OBJECT (file) in Windows OS. Dynamically created data structures may be derived from the global data structures. For instance, the KDBG and KPCR data structures contain the memory addresses of a large number of kernel variables. Examples include PsLoadedModuleList (points to the list of currently loaded kernel modules) and PsActiveProcessHead (pointer to the start of the kernel's list of EPROCESS structures). For those data structures that can not be derived automatically from the global data structures, Volatility scanners may be used to identify unlinked structures at run-time.

2.4 Design and Implementation

2.4.1 Assumptions and Requirements

The development of the RTKDSM system was driven by the following requirements:

1) The system did not require any modifications to the monitored OS and no additional software needed to be installed in the monitored VM.

2) The system imposed minimal performance overhead and operated seamlessly in the background with the monitored VM running at full speed.

The following assumptions were made when developing the system:

1) The Trusted Computing Base (TCB) for our system included the hypervisor and all of the software in the monitoring VM.

2) Kernel data structures of the introspected OS conformed to known semantic and syntactic data structure layouts even in a compromised state.

   This assumption is common to most current VMI-based solutions. It is fairly difficult for an attacker to modify the layout of these data structures as such modifications would require updating all code in the system that uses them or, otherwise, the affected OS would no longer function properly. These updates would also be challenging to perform and to hide. Although Bahram et al. [27] demonstrated the feasibility of semantic and syntactic data structure manipulation attacks to subvert introspection, this type of attacks could be defeated using data structure invariant inference and enforcement tools [28] and by generating robust signatures for kernel data structures [29].

3) Kernel data structures of interest were assumed to be memory-resident at the time of scan and, once identified, were never moved (paged) between physical memory and the page file. While the kernel might keep some data in the paged memory whose contents might be swapped into a file, the most critical and frequently accessed kernel objects, such as those used in this study, were known to be permanently kept in the non-paged memory. So, the rest of this study referred to the non-paged memory and non-paged kernel data structures only.

2.4.2 Design

The RTKDSM system is composed of two agents: the introspection agent and the monitoring agent. The introspection agent gathers and analyzes kernel data structures in the monitored VM. The monitoring agent is hosted in the hypervisor. Its purpose is to detect write attempts to the monitored kernel data structures (Figure 2.1).

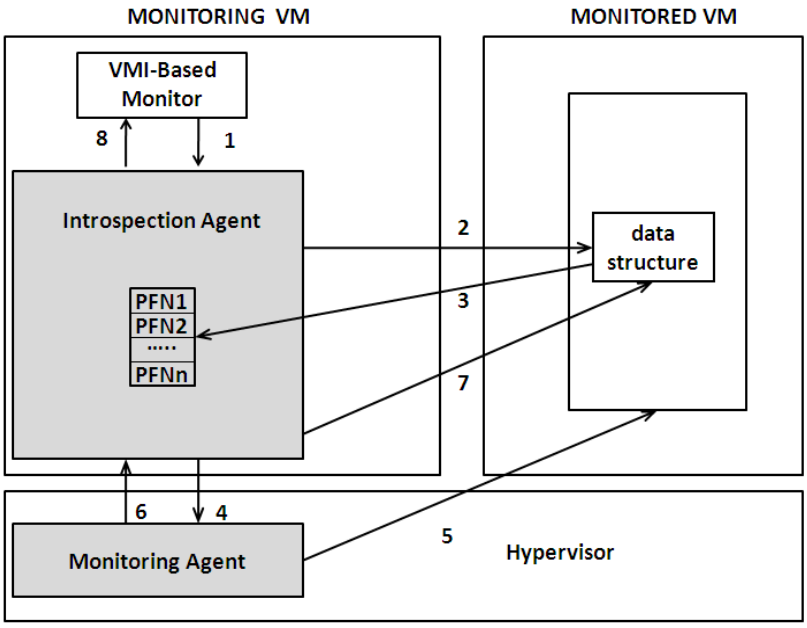

Figure 2.1 Logical layout and workflow of the system.

The RTKDSM system is designed to operate in two modes: (1) data structure identification and analysis and (2) data structure monitoring. The identification and analysis mode may be used by VMI monitors to request the RTKDSM system to identify locations of data structures and to return values of specific fields within data structures. The VMI monitor is responsible for deducing the semantic meaning of the returned values. The monitoring mode is used by VMI monitors to request the RTKDSM system to monitor data structures for changes in real-time. The VMI request has the following format: (*mode, data_structure_type, data_structure_offset, field_name1, field_name2, …, field_nameN*). Examples of data structure types (*data_structure_type*) include: EPROCESS (process), TOKEN (token), and ETHREAD (thread) in Windows OS. Examples of field names (*field_name*) include: ImageFileName (EPROCESS), UserAndGroupCount (TOKEN), and CreateTime (ETHREAD). The RTKDSM system provides VMI application developers with pre-configured lists of supported data structure types and field names for each data structure type. These lists are derived from the Volatility profiles.

The overall algorithmic outline of the RTKDSM comprises the following high-level steps:

1) Upon a request from a VMI monitor (Step 1 of Figure 2.1), the introspection agent searches the physical memory of the monitored VM (Figure 2.1, Step 2) to locate data structures specified in the request. If the identification mode is used, the introspection agent extracts the memory offsets of the identified data structures or values of the requested fields and returns the results to the VMI monitor (Step 8 of Figure 2.1). Examples of VMI requests in the identification mode include:

- (identification, EPROCESS, 0x0, ‘’) request instructs the introspection agent to identify all EPROCESS data structures and returns the memory locations of the identified data structures to the VMI monitor.

- (identification, EPROCESS, 0x0, ‘ImageFileName’) request instructs the introspection agent to identify all EPROCESS data structures and returns the names of the corresponding processes.

- (identification, EPROCESS, 0x000fabcd, ‘ImageFileName’) instructs the introspection agent to return the name of the process whose EPROCESS data structure is located at the 0x000fabcd offset.

If the monitoring mode is requested, the introspection agent extracts the monitored VM’s physical page frame numbers (PFN) of those memory pages where the monitored data fields reside including their address ranges within the page (Step 3 of Figure 2.1). Examples of VMI requests in the monitoring mode include:

- (monitoring, EPROCESS, 0x000fabcd, ‘ImageFileName’) instructs the RTKDSM system to calculate the offset of the ImageFileName field within the EPROCESS data structure located at the 0x000fabcd offset, calculate the corresponding PFN, and to monitor the ImageFileName field for changes in real-time. When a change in the field is detected, the new value is returned to the VMI monitor.

- (monitoring, EPROCESS, 0x000fabcd, ‘’) instructs the RTKDSM system to calculate the PFN (or multiple PFNs if the data structure crosses page boundaries) for the entire EPROCESS data structure located at the 0x000fabcd offset and to

monitor the entire data structure in real-time. When a change in the data structure is detected, the VMI monitor is notified of the change.

2) The introspection agent stores the calculated PFNs and the address ranges in a list, called the monitored list. The monitored list is delivered to the monitoring agent (Step 4 of Figure 2.1). The monitoring agent continuously monitors data structures in real-time by intercepting all memory writes to the pages in the monitored list (Step 5 of Figure 2.1).

3) On intercepting a write on a page, if the write is within one of the monitored address ranges, the monitoring agent allows the write operation to proceed and notifies the introspection agent of the corresponding PFN (Step 6 of Figure 2.1) for real-time analysis of the updated page (Figure 2.1, Step 7). If the memory page hosts a data structure known to cross page boundaries and to reside on multiple pages, the analysis involves the entire set of PFNs comprising the data structure. Subsequently, the VMI monitor is notified of the new state of the data structure (Figure 2.1, Steps 8) and is responsible for deducing the semantic meaning of the returned values. If the write is not within any of the known monitored memory ranges, the monitoring agent allows the write operation to proceed without notifying the introspection agent.

2.4.3 Implementation

We implemented a prototype RTKDSM architecture using the Xen hypervisor and HVM Windows-and Linux-based VMs. In our implementation, the introspection agent is deployed in the Dom0 domain. The monitoring agent is implemented in the Xen hypervisor. The RTKDSM system implementation involves the following steps:

1) *Request from a VMI Monitor:* A VMI monitor requests the RTKDSM system to either

identify data structures (in the identification mode) or to perform real-time monitoring of a data structure (in the monitoring mode) (Step 1 of Figure 2.2).

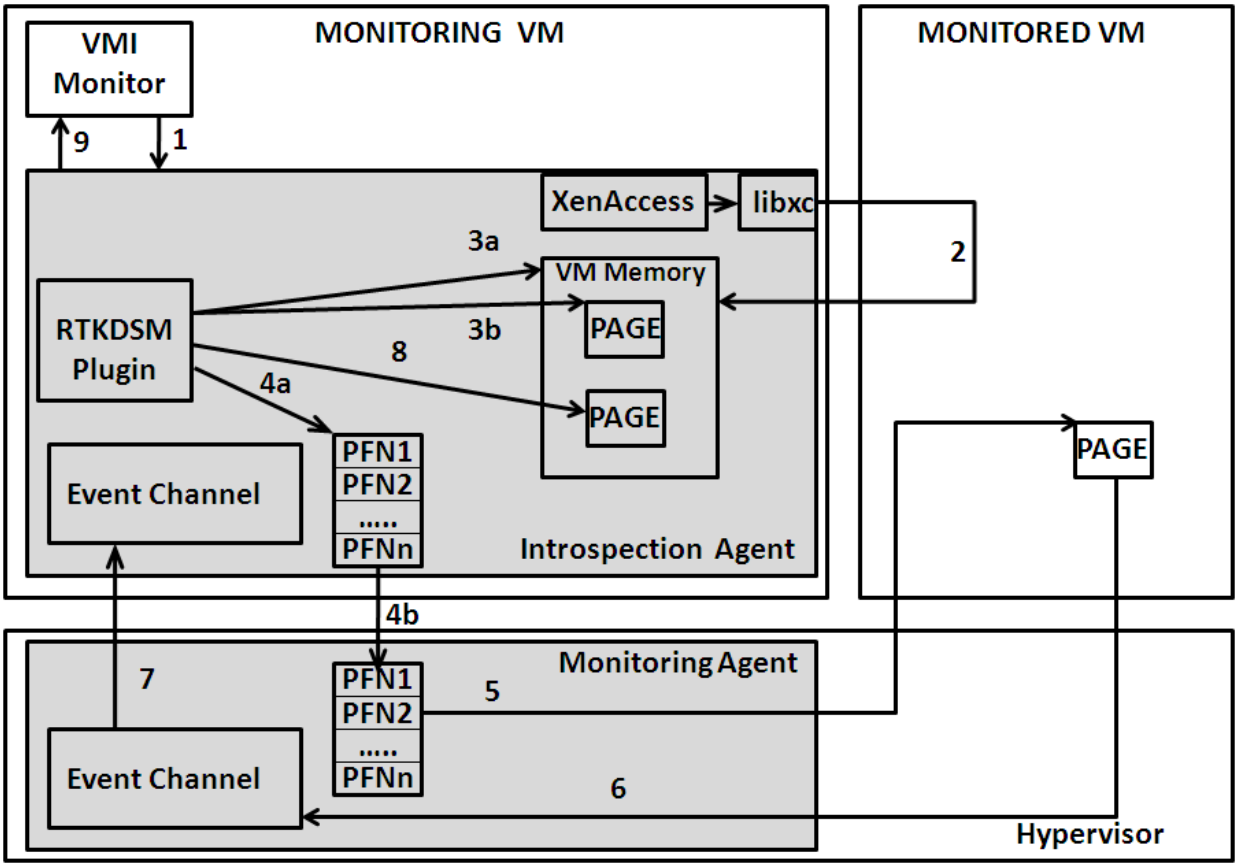

Figure 2.2 System implementation.

2) *Memory mapping*: To analyze the memory of a running VM, we first have to access the VM's memory. As the Volatility framework does not have built-in mechanisms to map the memory of a running VM, we configured the RTKDSM system to access the VM memory using the XenAccess API [6] (Step 2 of Figure 2.2). XenAccess is a Dom0 user-space library built upon the low-level APIs provided by Xen to facilitate VM state introspection. The Xen distribution provides a Xen Control library (libxc) for a Dom0

process to act on the VMs, including pausing a VM, resuming a paused VM, reading a VM's physical memory page, modifying a VM's physical memory page, etc. Specifically, libxc provides a xc_map_foreign_range() function that is designed to map the physical memory space of a target VM into a Dom0 process's virtual address space so that the latter can easily manipulate the target VM's physical memory. XenAccess uses this API function to map the physical memory pages of the VM. Specifically, we leverage the PyXaFS file system, which is part of the XenAccess tool suite, to map physical memory pages of a VM inside Dom0. PyXaFS exposes the memory of a VM as a regular file and allows the introspection agent to read a live VM's memory as if it were a normal file. PyXaFS is designed for integration with the Volatility framework as an address space.

3) *Data Structure Search*: To allow the RTKDSM system perform its data structure searches, we extended the Volatility framework with two new plugins called rtkdsm.py (real time kernel data structure monitoring plugin) for Windows OS and rtkdsm_linux.py for Linux OS. The rtkdsm.py and rtkdsm_linux.py plugins utilize the existing Volatility list traversal and signature-based scanning algorithms for extraction of data structures. In the current implementation, the plugins' functionality is limited to identification and monitoring of only those data structures that are used in the vCardTrek, CLAW, and ATOM studies but can be easily extended to support other data structures documented in the Volatility profiles. The plugins are written in Python, and when used in the monitoring mode, can directly access a memory page and a data structure within the memory page by supplying the data structure type and offset. The rtkdsm plugins are also used to calculate offsets and lengths of data fields that require monitoring. Data fields'

offsets and lengths within the data structure are determined using the Volatility profiles. For instance, a VMI monitor may issue a request to monitor the ImageFileName field storing the process name. This field is defined by a Volatility profile as 16 bytes long and located at the 0x174 offset from the top of the EPROCESS data structure.

Given the VM's physical memory mapped using PyXaFS, the introspection agent searches the mapped pages for target data structures (Step 3a of Figure 2.2) or analyzes a particular data structure at a known offset (Step 3b of Figure 2.2). This live system analysis is unobtrusive to the target VM and does not change the system state during the data acquisition process. In the monitoring mode, the data structure and fields offsets are converted to PFNs (Step 4a of Figure 2.2), which are delivered to the monitoring agent for real-time monitoring.

4) *Monitoring*: The monitored PFN list is mapped for shared access from the hypervisor context between the introspection agent and the monitoring agent (Step 4b of Figure 2.2). We added a new hypercall to the hypervisor to trigger this sharing. The list is stored using a page-level bitmap. The bitmap maintains one bit for each page of physical memory assigned to the monitored VM. The monitoring agent manages the bitmap by setting the appropriate bits for the monitored PFNs.

   All writes to the memory pages corresponding to those in the PFN list are intercepted by the monitoring agent. This is achieved by marking the pages as read-only (Step 5 of Figure 2.2) and configuring the hypervisor to recognize page faults caused by writes to these read-only pages (Step 6 of Figure 2.2). To reduce the amount of code modifications in the hypervisor for implementing this mechanism, we developed an extension to the

Xen's log dirty mode to support continuous tracking of modifications to memory pages. Specifically, we leveraged the shadow paging infrastructure to configure the hypervisor to intercept writes to monitored memory pages. Unlike the log dirty mode where all shadow entries are destroyed on its activation, we destroyed only those shadow page table entries that corresponded to the PFNs of memory pages with the identified data structures. When the monitored VM attempted to access a page without an existing shadow entry, a shadow page fault occurred, and the shadow entry was re-constructed. In Xen, the PTE propagation logic is implemented in the _sh_propagate function (defined in xen/arch/x86/mm/shadow/multi.c) — the "heart" of the shadow paging code, which constructs the shadow PTEs from the corresponding guest entries. In the_sh_propagate function, we intercepted the propagation of entries between the guest page tables and shadow page tables, and then write-protected designated frames of the guest OS's physical memory by setting the shadow PTEs with read-only bit if the physical memory page referenced by the PTE was marked as containing a data structure in the PFN list. The shadow PTE flags were otherwise identical to the original guest PTE flags. By doing so, all the shadow entries corresponding to the monitored pages were effectively marked as read-only.

When set on a page, the read-only bit caused the processor to trap into the hypervisor whenever a write was detected on the page and transfer control to the _sh_page_fault function, the Xen's page fault handling routine. In the log dirty mode, such writes resulted in the page marked dirty and write permissions being granted to the accessed page, so as to avoid traps on subsequent writes. In our implementation, if the write was within a monitored address range on the page, we allowed the write in a three-step

procedure:

i. Marked the page as writable and re-executed the faulting instruction as if no fault occurred.

ii. Set the trap flag, commonly used by single-stepping debuggers for the guest OS, to cause a debug exception after the writing instruction was executed. We trapped this exception in the hypervisor and then re-set the page to read-only restoring the protected state.

iii. If the write was within the monitored range, we notified the introspection agent of the write (Step 7 of Figure 2.2) via an event channel established between the introspection and the monitoring agents at the beginning of the monitoring. Notifications were delivered via two types of memory pages created by the hypervisor and shared with Dom0: a descriptor page of 4 KByte to notify availability of data to the introspection agent and a data page of 4 KByte to share the details of the updated page including the offset of the write and the PFN.

5) *Repeat Analysis*: Upon receiving a notification from the monitoring agent, only the page (or a set of pages if the data structure was known to span multiple pages) where the modification occurred was re-analyzed by the introspection agent (Step 8 of Figure 2.2). The rtkdsm plugins extracted the new value of the field where the change had occurred and returned it to the calling VMI monitor (Step 9 of Figure 2.2).

6) *Modifications to the monitored list*: The monitored PFN list was designed to be modified at run-time by adding new or deleting existing entries. Each time an update was made to the monitored list, the system forced propagation of new PTE mappings in the shadow

page cache.

2.4.4 Limitations

An inherent limitation of the RTKDSM system is its performance penalty in the monitoring mode. While the OS inside the monitored VM accesses and manipulates data at the granularity of machine words, the RTKDSM system intercepts writes only at the page level. This is because the commodity x86 processors do not offer a mechanism for generating faults upon access to specific byte-level memory addresses. Even though the RTKDSM system is able distinguish between monitored and non-monitored addresses within a single page, page faults will still occur and introduce performance cost for writes to all other addresses that do not contain target data on the page.

Consequently, the RTKDSM implementation results in two types of page faults. First, when the shadow entry does not exist, both read and write access generate a shadow page fault. Second, when an attempt is made to modify a page through an existing shadow entry without a write permission, a shadow page fault occurs. The second type is the predominant source of overhead in the RTKDSM system and is likely to cause a significant performance impact on the guest OS by VMI monitors relying on monitoring of a large number of dynamic data structures that are constantly written to. In the worst case, every write to every kernel data structure may be monitored resulting in the costs being extremely high. So it is important to provide a mechanism to reduce the number of page faults of the second type.

We extended the RTKDSM design of the monitoring agent to operate in two modes: 1) the “always-on” mode that continuously monitors the VM kernel data structures; 2) the “periodic polling” mode that performs periodic checks after a pre-defined period of time T. In the

"periodic polling" approach, the monitoring agent intercepts a write to a monitored page and enables the write flag on the page for a specified period of time T. Once T elapses, the introspection agent re-analyzes the page, and the monitoring agent enables the read-only flag on the page. As the next write is intercepted, another detection round comprising the above steps is repeated.

Although the "periodic polling" mode prevents the hypervisor from accounting for potentially unrelated and/or spurious modifications as relevant, reducing the frequency of checks introduces the possibility of evasion when used in VMI security systems. A malicious data structure modification can go undetected if it occurs between two consecutive checks. This is especially possible when the polling interval is predictable. To prevent adversaries from exploiting the periodic nature of the polling mode, we support randomization of the timing parameter T using intervals pulled from a uniform distribution in the interval $(T-\partial t, T+\partial t)$, with $\partial t < T$. As the security provided to a system is closely related to the frequency of checks, the "always-on" mode vs. "periodic polling" mode should be considered in each individual instance with the following consideration in mind: the "always-on" provides increased security, while the "periodic polling" mode reduces performance overhead. The greater the period of time between checks, the more time an attacker has to execute a sophisticated attack and to avoid detection by removing the traces of the intrusion between subsequent checks.

Another limitation of the RTKDSM system is its inability to detect inconsistencies in OS data structures undergoing updating, for instance, a multi-word field might be updated in parts but the system would try to analyze each update before updating of the entire field is completed.

2.5 Evaluation

2.5.1 Experimental Setup

Our testbed consisted of a virtualized server that used Xen version 3.3 as the hypervisor and Ubuntu 9.04 (Linux kernel 2.6.26) as the kernel for Dom0. The host system had Duo CPU P8600 processor running two cores at 2.4GHz and 2GB of system memory. The RTKDSM system was installed in the Dom0 domain. In addition, the virtualized server hosted 2 VMs running a default installation of Windows XP OS with the IIS web server, MSSQL database server, Internet Explorer, and MS Office installed on each of the machines and 2 VMs running a default installation of Ubuntu Jaunty (Linux kernel 2.6.28) with the Apache web server, MySQL database server, and Firefox installed on each of the machines (Figure 2.3). These VMs were configured with 512Mb RAM.

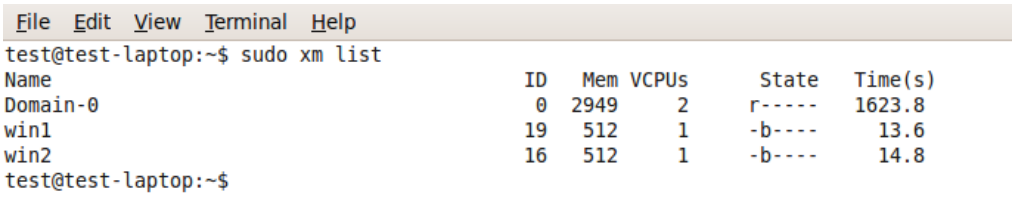

Figure 2.3 Windows OS test environment

2.5.2 Spurious Page Fault Experiments

We conducted experiments to estimate the probability of spurious updates, i.e. updates that might occur outside of monitored kernel data structures. Specifically, we recorded page faults caused by real-time monitoring of the data structures listed in Table 2.4 over the period of one minute in idle Windows and Linux VMs. In the Windows VM, the experiments included: (1)

monitoring of the PsActiveProcessHead structure, (2) monitoring of the TCBTable structure, (3) monitoring of 50 EPROCESS structures, (4) monitoring of 50 ETHREAD structures, (5) monitoring of 50 TOKEN structures, and (6) monitoring of 50 PEB_LDR_DATA structures. In the Linux VM, the experiments included: (1) monitoring of the init_task structure, (2) monitoring of 50 task_struct structures, and (3) monitoring of 50 files_struct structures. Prior to each experiment, the test VMs were rebooted bringing the environment into a known and reproducible state. The script shown in Figure 2.4 was then executed to invoke 50 processes inside the test VM.

```
@echo off
for /L %%i in (1,1,10) do (
start C:\WINDOWS\system32\calc.exe
)
```

Figure 2.4 A sample Windows OS command script to invoke 10 processes.

In the Windows VM, the RTKDSM system located the PsActiveProcessHead structure, the TCBTable structure, and enumerated all EPROCESS, ETHREAD, TOKEN, and PEB_LDR_DATA data structures corresponding to the processes invoked by the script. The PsActiveProcessHead, TCBTable, EPROCESS, ETHREAD, TOKEN, and PEB_LDR_DATA data structures were then monitored for updates in real-time using the RTKDSM system. In the Linux VM, the RTKDSM system located the init_task structure and enumerated all task_struct and files_struct data structures corresponding to the processes invoked by the script. The init_task, task_struct, and files_struct data structures were then monitored for updates in real-time using the RTKDSM system. Table 2.5, Table 2.6, Table 2.7, Table 2.8, Table 2.9, Table 2.10, and Table 2.11 show the results of these experiments.

Table 2.4 Data structures used in the experiments.

OS	Data Structure	Description
Windows OS	PsActiveProcessHead	Points to the first and the last EPROCESS (see below) data structure.
	TCBTable	Transmission Control Block Table lists network connections.
	EPROCESS	Represents a running process.
	ETHREAD	Represents a running thread.
	TOKEN	Represents authorization information for a running process.
	PEB_LDR_DATA	Represent a list of loaded modules.
Linux OS	init_task	Points to the first and the last task_struct (see below) data structure.
	task_struct	Represents a running task. This structure also stores the process authorization information similar to TOKEN in Windows OS and thread related information similar to ETHREAD in Windows OS.
	files_struct	Represents a list of files used by a process.

Table 2.5 Page faults on pages containing the PsActiveProcessHead, TCBTable, and init_task structures in the idle Windows VM #1 and Linux VM # 1 recorded during 1 minute.

Data Structure	Inside the structure	Outside the structure
PsActiveProcessHead	0	11258
TCBTable	0	1812
init_task	0	5634

Table 2.6 Page faults on pages containing EPROCESS structures for the 50 calc.exe test processes in the idle Windows VM #1 recorded during 1 minute.

Process number	Number of page faults	
	Inside the EPROCESS structure	Outside the EPROCESS structure
6	0	26
7	0	18
14	0	54
15	0	1828
20	0	28
23	36	290
25	0	149
31	0	49
32	0	37
34	0	65
35	0	6
37	0	6
38	0	91
41	0	59
42	0	76
46	0	34
47	0	11
50	0	51
All other processes	0	0

Table 2.7 Page faults on pages containing ETHREAD structures for the 50 calc.exe test processes in the idle Windows VM #1 recorded during 1 minute.

Process number	Number of page faults	
	Inside the ETHREAD structure	Outside the ETHREAD structure
3	5,304	0
11	5,698	5
14	0	214
17	0	43
19	5,465	0
26	5,338	0
27	5,347	0
31	5,317	4
39	5,569	0
47	0	1028
All other processes	0	0

Table 2.8 Page faults on pages containing TOKEN structures for the 50 calc.exe test processes in the idle Windows VM #1 recorded during 1 minute.

Process number	Number of page faults	
	Inside the TOKEN structure	Outside the TOKEN structure
4	0	16
11	0	34
49	0	28
All other processes	0	0

Table 2.9 Page faults on pages containing PEB_LDR_DATA structures for the 50 calc.exe test processes in the idle Windows VM #1 recorded during 1 minute.

Process number	Number of page faults	
	Inside the PEB_LDR_DATA structure	Outside the PEB_LDR_DATA structure
All processes	0	0

Table 2.10 Page faults on pages containing task_struct structures for the 50 gcalctool test processes in the idle Linux VM #1 recorded during 1 minute.

Process number	Number of page faults	
	Inside the task_stuct structure	Outside the task_struct structure
15	0	178
16	0	24
18	0	226
30	0	75
All other processes	0	0

Table 2.11 Page faults on pages containing files_struct structures for the 50 gcalctool test processes in the idle Linux VM #1 recorded during 1 minute.

Process number	Number of page faults	
	Inside the files_stuct structure	Outside the files_struct structure
1	0	16
4	0	44
21	0	186
34	0	52
37	0	242
46	0	36
All other processes	0	0

Although updates to the PsActiveProcessHead, TCBTable, init_task, EPROCESS, task_struct, and files_struct data structures were infrequent, the pages hosting these structures contained varieties of other unrelated data structures, which experienced frequent updates. Several ETHREAD data structures changed quite rapidly leading to a large number of page faults on the corresponding pages. Updates outside of the ETHREAD data structures were infrequent. Updates to the pages containing the TOKEN and PEB_LDR_DATA data structures were rare.

2.5.3 Performance Experiments

We used a combination of micro/synthetic and application benchmarks to understand the direct computational overhead introduced by the RTKDSM system on the test VMs. In Windows OS, we used the PCMark05 benchmark [30] to measure the impact of the running RTKDSM

system on the VM's CPU, memory, and hard drive. In Linux OS, we used the NBench benchmark [31] to measure the impact of the running RTKDSM system on the VM's CPU, FPU, and system memory speed. We also ran the Apache HTTP performance benchmark as an application benchmark for both the Windows OS and Linux OS [32]. This benchmark heavily relied on both threading and I/O operations. Additionally, we ran the file compressing application (gzip) in Linux OS to evaluate the performance incurred by extensive I/O operations based on the time required to compress a 20 MB file.

2.5.3.1 "Always-On" Mode

We assessed the performance of the RTKDSM system in the "always-on" monitoring mode. In Windows VMs, the experiments included: (1) monitoring of the PsActiveProcessHead structure, (2) monitoring of the TCBTable structure, (3) monitoring of EPROCESS structures of 10, 25, and 50 processes, (4) monitoring of ETHREAD structures of 10, 25, and 50 threads, (5) monitoring of TOKEN structures of 10, 25, and 50 processes, and (6) monitoring of PEB_LDR_DATA structures of 10, 25, and 50 processes. In Linux VMs, the experiments included: (1) monitoring of the init_task structure, (2) monitoring of task_struct structures of 10, 25, and 50 processes, and (3) monitoring of files_struct structures of 10, 25, and 50 processes. Prior to each experiment, the test VMs were rebooted bringing the environment into a known and reproducible state. The script shown in Figure 2.4 was then executed to invoke a required number of processes inside a test VM.

The performance overhead was first measured with only 1 running VM and then with 2 VMs running concurrently for each OS. Each benchmark was run 3 times against one test VM for each OS. Table 2.12 and Table 2.13 show the average results of running the PCMark05 and

Apache benchmarks in Windows OS. Table 2.14 and Table 2.15 show the average results of running the NBench, gzip, and Apache benchmarks in Linux OS. In the Apache benchmark, the average process time per request was used for comparison. The results shown have been calculated with respect to the speed of the Xen system with the RTKDSM system enabled with zero pages monitored.

The performance results demonstrated the performance overhead generally increased as the number of monitored structures increased. Additionally, the performance overhead also increased as the number of monitored VMs grew within the host. The performance was also affected by the type of a benchmark used in the experiments. Particularly, the Apache benchmark had a significant impact on the performance due to spurious page faults resulting from running this benchmark. However, the outputs generated by the RTKDSM system would be sufficiently fast for use in systems that either monitored data structures in memory regions, which did not incur many page faults, such as those hosting TOKEN and PEB_LDR_DATA data structures or systems that could tolerate reduced performance, for instance, in a VM replay for live forensic analysis of running VMs [33].

Table 2.12 Performance in the “always-on” mode using the PCMark05 benchmark in Windows OS.

Benchmark	# of VMs	Monitoring of PsActiveProcessHead	Monitoring of TCBTable	# of EPROCESS structures monitored per VM			# of TOKEN structures monitored per VM			# of ETHREAD structures monitored per VM			# of PEB_LDR_DATA structures monitored per VM		
				10	25	50	10	25	50	10	25	50	10	25	50
CPU	1	1.1%	0.9%	1.2%	1.1%	1.6%	0.7%	0.9%	1.0%	1.2%	1.3%	1.7%	<0.2%	<0.2%	<0.2%
	2	2.0%	1.7%	2.3%	2.7%	3.1%	1.3%	2.1%	2.5%	2.2%	2.6%	3.3%	<0.2%	<0.2%	<0.2%
Memory	1	0.1%	0.1%	0.2%	0.2%	0.3%	0.2%	0.2%	0.3%	0.2%	0.3%	0.3%	<0.2%	<0.2%	<0.2%
	2	0.6%	0.5%	1.0%	1.2%	1.5%	0.8%	1.0%	1.3%	0.9%	1.0%	1.5%	<0.2%	<0.2%	<0.2%
HDD	1	3.5%	2.8%	6.7%	6.8%	8.7%	3.6%	4.3%	4.9%	8.7%	9.1%	11.9%	<0.2%	<0.2%	<0.2%
	2	5.3%	4.5%	7.2%	13.1%	13.6%	5.7%	8.5%	9.2%	9.7%	14.7%	15.1%	<0.2%	<0.2%	<0.2%

Table 2.13 Performance in the “always-on” mode using the Apache benchmark in Windows OS.

Number of requests / concurrency	# of VMs	Monitoring of PsActiveProcessHead	Monitoring of TCBTable	# of EPROCESS structures monitored per VM			# of TOKEN structures monitored per VM			# of ETHREAD structures monitored per VM			# of PEB_LDR_DATA structures monitored per VM		
				10	25	50	10	25	50	10	25	50	10	25	50
1000/5	1	8.7%	3.3%	<0.5%	45.1%	50.2%	<0.5%	0.4%	0.6%	13.7%	46.2%	106.1%	<0.5%	<0.5%	<0.5%
	2	12.2%	5.9%	<0.5%	56.9%	78.9%	<0.5%	2.1%	2.4%	15.9%	56.3%	131.8%	<0.5%	<0.5%	<0.5%
1000/10	1	8.0%	2.8%	<0.5%	41.6%	48.3%	<0.5%	0.3%	0.5%	12.3%	42.2%	99.8%	<0.5%	<0.5%	<0.5%
	2	11.5%	5.2%	<0.5%	51.6%	66.7%	<0.5%	1.7%	2.1%	15.3%	53.4%	124.6%	<0.5%	<0.5%	<0.5%
5000/5	1	10.3%	4.4%	<0.5%	62.5%	78.4%	<0.5%	0.4%	0.6%	32.0%	63.1%	121.3%	<0.5%	<0.5%	<0.5%
	2	13.9%	6.9%	<0.5%	64.8%	75.3%	<0.5%	2.3%	2.6%	37.5%	76.7%	141.4%	<0.5%	<0.5%	<0.5%
5000/10	1	10.1%	4.1%	<0.5%	61.4%	71.6%	<0.5%	0.4%	0.6%	28.1%	60.4%	116.8%	<0.5%	<0.5%	<0.5%
	2	13.6%	6.5%	<0.5%	63.8%	75.1%	<0.5%	2.1%	2.3%	33.9%	72.6%	132.7%	<0.5%	<0.5%	<0.5%

Table 2.14 Performance in the “always-on” mode using the NBench & gzip benchmarks in Linux OS.

Benchmark	# of VMs	Monitoring of init_task	# of task_struct structures monitored per VM			#of files_struct structures monitored per VM		
			10	25	50	10	25	50
NBench Memory Index	1	0.2%	0.1%	0.2%	0.2%	0.1%	0.1%	0.2%
	2	0.4%	0.3%	0.5%	0.5%	0.2%	0.3%	0.5%
NBench Integer Index	1	0.7%	0.5%	0.9%	1.1%	0.5%	0.6%	0.9%
	2	1.2%	0.9%	1.3%	1.6%	1.0%	1.1%	1.4%
NBench Floating-Point	1	0.7%	0.4%	0.5%	0.5%	0.3%	0.5%	0.6%
	2	1.1%	0.7%	0.7%	0.8%	0.5%	0.8%	0.8%
gzip	1	2.7%	1.9%	2.3%	2.6%	1.8%	2.4%	2.8%
	2	3.8%	2.9%	3.7%	3.9%	3.1%	3.5%	3.5%

Table 2.15 Performance in the “always-on” mode using the Apache benchmark in Linux OS.

# of requests / concurrency	# of VMs	Monitoring of init_task	# of task_struct structures monitored per VM			# of files_struct structures monitored per VM		
			10	25	50	10	25	50
1000/5	1	6.5%	<0.5%	23.2%	25.4%	<0.5%	14.4%	22.1%
	2	8.3%	<0.5%	28.3%	30.7%	<0.5%	17.3%	26.7%
1000/10	1	5.9%	<0.5%	21.7%	24.2%	<0.5%	13.6.7%	20.7%
	2	7.9%	<0.5%	26.6%	29.8%	<0.5%	17.1%	24.2%
5000/5	1	7.4%	<0.5%	27.4%	30.6%	<0.5%	21.5%	32.4%
	2	8.9%	<0.5%	34.1%	38.2%	<0.5%	29.1%	39.2%
5000/10	1	7.0%	<0.5%	27.2%	29.2%	<0.5%	20.8%	31.3%
	2	8.7%	<0.5%	33.9%	36.2%	<0.5%	28.9%	38.8%

2.5.3.2 “Periodic Polling” Mode

We assessed the performance of the RTKDSM system in the “periodic polling” monitoring mode using the Apache HTTP benchmark only. As this benchmark was shown to cause significant performance deteriorations in the “always-on” monitoring mode, switching to the “periodic polling” monitoring mode was expected to improve the performance.

In the Windows VMs, the experiments included: (1) monitoring of the PsActiveProcessHead structure, (2) monitoring of the TCBTable structure, (3) monitoring of EPROCESS structures of 10, 25, and 50 processes, (4) monitoring of ETHREAD structures of 10, 25, and 50 threads, and (5) monitoring of TOKEN structures of 10, 25, and 50 processes. In

the Linux VMs, the experiments included: (1) monitoring of the init_task structure; (2) monitoring of task_struct structures of 10, 25, and 50 processes, and (3) monitoring of files_struct structures of 10, 25, and 50 processes. Prior to each experiment, the test VMs were rebooted bringing the environment into a known and reproducible state. The script shown in Figure 2.4 was then executed to invoke a required number of processes inside a test VM.

In each experiment, the benchmark was run 3 times against one test VM per each OS. The average process time per request was used for comparison. Table 2.16, Table 2.17, Table 2.18, Table 2.19, Table 2.20, and Table 2.21 show the average results of running the Apache benchmarks in Windows OS and Linux OS with the timing parameter T set to 50 msec, 10 msec, and 5 msec. The results shown have been calculated with respect to the speed of the Xen system with the RTKDSM system enabled with zero pages monitored.

The performance results demonstrated the “periodic polling” approach significantly decreased the performance overhead observed in the “always-on” mode. The recorded write bursts involving spurious updates caused by the Apache benchmark to the monitored pages lasted in the 1 to 15 msec range. Hence, the improvement in the performance was due to a significantly reduced number of page fault interceptions that excluded page faults caused by such write bursts.

Table 2.16 Performance in the “periodic polling” mode for the PsActiveProcessHead, TCBTable, and init_task data structures.

Number of requests / concurrency	PsActiveProcessHead			TCBTable			init_task		
	T=50 msec	T=10 msec	T=5 msec	T=50 msec	T=10 msec	T=5 msec	T=50 msec	T=10 msec	T=5 msec
1000/5	<0.5%	1.3%	2.7%	<0.5%	<0.5%	<1.0%	<0.5%	1.1%	2.2%
1000/10	<0.5%	1.2%	2.5%	<0.5%	<0.5%	<1.0%	<0.5%	0.9%	2.1%
5000/5	<0.5%	1.8%	3.5%	<0.5%	<0.5%	<1.0%	<0.5%	1.5%	3.2%
5000/10	<0.5%	1.5%	3.3%	<0.5%	<0.5%	<1.0%	<0.5%	1.3%	3.0%

Table 2.17 Performance in the “periodic polling” mode for the EPROCESS data structure.

Number of requests / concurrency	# of EPROCESS structures, T=50 msec			# of EPROCESS structures, T=10 msec			# of EPROCESS structures, T=5 msec		
	10	25	50	10	25	50	10	25	50
1000/5	<0.5%	3.1%	3.8%	<0.5%	4.1%	5.2%	<0.5%	7.3%	9.7%
1000/10	<0.5%	2.5%	2.9%	<0.5%	2.9%	3.4%	<0.5%	6.4%	8.1%
5000/5	<0.5%	3.4%	4.2%	<0.5%	4.4%	5.3%	<0.5%	7.9%	10.1%
5000/10	<0.5%	2.8%	3.3%	<0.5%	3.1%	3.8%	<0.5%	6.5%	8.8%

Table 2.18 Performance in the “periodic polling” mode for the ETHREAD data structure.

Number of requests / concurrency	# of ETHREAD structures, T=50 msec			# of ETHREAD structures, T=10 msec			# of ETHREAD structures, T=5 msec		
	10	25	50	10	25	50	10	25	50
1000/5	<0.5%	3.5%	5.1%	1.5%	5.2%	8.1%	2.8%	8.1%	15.6%
1000/10	<0.5%	3.2%	4.8%	1.5%	4.9%	7.7%	2.7%	7.9%	15.2%
5000/5	<0.5%	3.8%	5.7%	2.1%	5.9%	8.4%	3.9%	9.3%	17.3%
5000/10	<0.5%	3.4%	5.6%	1.9%	5.8%	8.1%	3.6%	8.8%	16.9%

Table 2.19 Performance in the “periodic polling” mode for the TOKEN data structure.

Number of requests / concurrency	# of TOKEN structures, T=50 msec			# of TOKEN structures, T=10 msec			# of TOKEN structures, T=5 msec		
	10	25	50	10	25	50	10	25	50
1000/5	<0.5%	<1%	<1%	<0.5%	<1%	<1%	<0.5%	<1%	<1%
1000/10	<0.5%	<1%	<1%	<0.5%	<1%	<1%	<0.5%	<1%	<1%
5000/5	<0.5%	<1%	<1%	<0.5%	<1%	<1%	<0.5%	<1%	<1%
5000/10	<0.5%	<1%	<1%	<0.5%	<1%	<1%	<0.5%	<1%	<1%

Table 2.20 Performance in the “periodic polling” mode for the task_struct data structure.

Number of requests / concurrency	# of task_struct structures, T=50 msec			# of task_struct structures, T=10 msec			# of task_struct structures, T=5 msec		
	10	25	50	10	25	50	10	25	50
1000/5	<0.5%	1.3%	1.5%	<0.5%	3.2%	3.9%	<0.5%	5.8%	6.9%
1000/10	<0.5%	1.3%	1.3%	<0.5%	3.1%	3.4%	<0.5%	5.7%	6.7%
5000/5	<0.5%	1.5%	2.1%	<0.5%	3.9%	4.4%	<0.5%	6.3%	8.8%
5000/10	<0.5%	1.3%	1.9%	<0.5%	3.5%	4.3%	<0.5%	6.0%	8.5%

Table 2.21 Performance in the “periodic polling” mode for the files_struct data structure.

Number of requests / concurrency	# of files_struct structures, T=50 msec			# of files_struct structures, T=10 msec			# of files_struct structures, T=5 msec		
	10	25	50	10	25	50	10	25	50
1000/5	<0.5%	1.1%	1.3%	<0.5%	2.8%	3.4%	<0.5%	4.7%	6.4%
1000/10	<0.5%	0.9%	1.2%	<0.5%	2.7%	3.3%	<0.5%	4.7%	6.3%
5000/5	<0.5%	1.1%	1.6%	<0.5%	3.6%	4.2%	<0.5%	5.2%	7.9%
5000/10	<0.5%	1.0%	1.4%	<0.5%	3.5%	3.9%	<0.5%	4.8%	7.5%

2.6 Summary

We presented the design and implementation of RTKDSM, a real-time kernel data structure monitoring system, capable of automatically identifying OS data structures supported by the open source Volatility forensic framework in memory of a running VM and tracking updates to the identified data structures in real-time. To demonstrate the applicability of the RTKDSM system under real-life conditions, we built three systems described in Chapters 3, 4, and 5 correspondingly: (1) payment card data flow tracking tool (vCardTrek), (2) cloud-based application whitelisting system (CLAW), and (3) access token manipulation attack detection tool (ATOM). By demonstrating the applications of the RTKDSM system, we hoped to promote the creation of new VMI tools through the techniques described in the following chapters.

3 Automated Discovery of Credit Card Data Flow for PCI DSS Compliance

3.1 Introduction

Among organizations increasingly targeted by ongoing cyber security attacks are retail businesses. These businesses make high-value targets for financially motivated cyber attackers because of the valuable credit and debit card data used in payment transactions. In the recent years, hackers have exploited weaknesses in payment card processing systems to steal sensitive customer card data [34].

To reduce security vulnerabilities in payment card processing systems, the Payment Card Industry Security Standards Council developed and released the Payment Card Industry Data Security Standard (PCI-DSS) [35]. All merchants that store, process, or transmit card data are required to comply with the PCI-DSS security requirements to ensure that not only the payment processing infrastructure, but the data it carries are better protected from unauthorized exposure. Noncompliant entities receive monthly fines and eventually may lose their ability to process card payments.

The key pre-requisite for PCI DSS compliance is the construction of the card data flow diagram for a payment processing network that accepts card charges and provides card processing service. Merchants must determine precisely how card data flow through their payment processing systems from their inception, what systems they traverse, and where they reside. A card data flow could start from a card swipe at a store, or a card number input by a user

into an E-commerce web site, and consists of all intermediate stops in a merchant's IT network at which the card information is examined or processed. This discovery process and the resulting card data flow diagram help merchants understand which IT equipments in the organization interact with the card data so as to implement the security of these IT equipments according to the PCI DSS compliance requirements.

In practice, this pre-requisite poses a challenge to merchants. As the payment card processing infrastructure is implemented and later maintained, it often deviates from the originally documented design. Without consistent tracking and auditing of changes, such deviations in many cases remain undocumented. Today, no known tool exists that could automatically discover the card data flow of a distributed payment card processing system in heterogeneous computing environments. The only available solution to this problem today is manual card data flow reconstruction based on outputs from data loss prevention (DLP) tools and system design documents. DLP tools work by searching network packets and data stored on disk for clear text card numbers. Although highly effective when dealing with unencrypted data, the DLP tools are largely powerless when card data are encrypted in transit and on storage. Likewise, manual review of system design documents is an extremely labor-intensive and time-consuming effort. The required information is often difficult to extract because it is spread across a variety of IT elements and applications. Therefore, building the card data flow for a given payment card processing infrastructure is considered a daunting task that at this point requires significant manual efforts.

We developed an automated tool called vCardTrek capable of building the card data flow in a distributed payment card processing system hosted on virtualized physical servers. We focus on virtualized servers because virtualization technology is quickly rising to predominate in

merchants' data centers, and many payment card processing systems start to run inside virtual machines [36-38]. A key design decision of the vCardTrek tool is to apply the RTKDSM system to track card data flows.

To the best of our knowledge, vCardTrek is the first known tool to leverage VMI to automatically discover the card data flow of distributed applications running in virtualized environments. We have implemented a working prototype for the Xen hypervisors. Our implementation does not require modifications to the hypervisor, VMs, guest OS, or payment card processing system components themselves. We have demonstrated the effectiveness of vCardTrek by applying it to 3 commercial payment card processing systems and successfully building the card data flow path for each of them. We expect the availability of vCardTrek could significantly decrease the efforts and costs in meeting the security regulations stipulated in the PCI DSS standard.

3.2 Related Work

Previous research efforts approached the automated data flow tracking problem from different angles, including a process-wide flow tracking, cross-process flow tracking, and cross-host flow tracking using fine-grained dynamic taint analysis (DTA). In DTA, data of interest are marked as tainted, and the taint propagation is monitored along with the data. The DTA data flow tracking mechanisms lead to increased level of detail, but either require a priori knowledge of the applications and hosts participating in information exchange so they can be properly instrumented or incur significant performance overheads that make such approaches unsuitable for interactive distributed network applications in production environments. Although our approach is more coarse-grained than the DTA methods and thus leads to a reduced level of

detail in the produced data flow, it removes the need for application-specific instrumentations and the associated performance penalties.

Several studies have explored the problem of data flow tracking in cross-host distributed systems. These can be roughly divided into dynamic binary instrumentation (DBI) and emulator-based implementations both using the DTA technique. Unlike these studies, we consider the most generic black-box approach that can be easily integrated into production environments, where no previous knowledge of the components participating in the data flow is provided, and only passive non-intrusive (i.e. require no modification of the monitored system) monitoring instruments with low performance impact are used.

3.2.1 Dynamic Binary Instrumentation Systems

The data flow tracking tool described in [39] is built upon a DBI framework and is designed to track information flow between processes which may be located in different host systems. In this implementation, hosts and processes participating in the information flow are manually identified, and a DBI tool is then attached to each of the identified processes to track information flow within the process boundary. Additionally, a flow manager is placed in each participating host to relay taint information between interacting processes and to handle cross-host communications and data flow concatenations.

In another related study [40], a single process DBI framework is extended to perform cross-process and cross-host transfer of taint information by intercepting and instrumenting the system calls used for cross-process as well as for cross-host communication. As these implementations require prior knowledge of the hosts and the processes involved in the information flow, these tools can not be utilized for data flow tracking where systems and

processes participating in the data flow are unknown a priori.

3.2.2 Emulator-Based Systems

Several DTA studies explored the use of emulators to perform fine-grained taint propagation and tracking between processes and hosts in virtualized networks. In these implementations, the typical approach is to instrument hardware emulators, such as QEMU [41], with taint tracking instructions and monitor the taint propagation at the hardware level [42]. Taint tracking data structures are used to keep taint status flags of every byte in the system including physical memory, CPU registers, and device state. The emulator propagates taint flags whenever their corresponding values in hardware are involved in an operation.

In a related study, Data Flow Tomography [43] built on QEMU emulator implements fine-grained data flow analysis system to track and visualize data flow on a networked set of virtual machines each running on a separate physical host. The Data Flow Tomography method uses full instruction emulation and is inherently heavy weight both in memory and time. Hardware emulation is extremely slow and incurs significant performance overheads making this approach unsuitable for interactive network applications in production environment. To be a useful tool in the life cycle of a system, methods will be needed to speed up the analysis. While data flow tracking within a single machine is rarely problematic, the scalability of the approach as the number of nodes increases beyond two is certainly a question. This method also requires QEMU installation on every machine involved in the data flow.

Some research has been done to explore more efficient means for dynamic taint analysis. Zhang *et. al.* [44] implemented Neon, an extension of the [42] approach developed to prevent data leaks. Neon focuses on taint propagation across applications, systems, and networks. Neon

implementation is based on the Xen hypervisor combined with demand emulation via QEMU, in which a running system dynamically switches between virtualized and emulated execution, and emulation is only used when tainted data is being processed by the CPU. This implementation leads to increased performance compared to using a processor emulator alone [43]. However, because propagating taint requires the invocation of QEMU, the Neon implementation incurs significant execution time overhead due to tag processing from the emulator and thus does not significantly improve performance.

3.3 Design and Implementation

3.3.1 Payment Card Processing System

vCardTrek is designed for a payment card processing system consisting of multiple distributed application components all running on distinct VMs as separate processes and communicating with one another using synchronous requests. A payment request using a credit or debit card number is sent to the entry component in the system, e.g. a card swipe at a point-of-sale terminal at a store. Each application component forwards the request to the next component along the card processing path and blocks until the corresponding response is received. Once the payment card processing system verifies that an input request's card information is accurate and sufficient funds are available in the account, the request is granted permission to proceed with the purchase. Additional processing steps within the merchant's network may be triggered after a payment request is authorized, such as submission of payment data to storage, marketing data collection, payment reconciliation and settlement etc.

3.3.2 Assumptions

Two assumptions were made when developing the tool:

1) Each card data handling component processes each request in a synchronous fashion, i.e., it reacts to an input request immediately and does not queue it for later processing.

2) When applying vCardTrek to a network to discover the card data flow, the network is in a "quiescent" state in the sense that only one test payment transaction is running through the payment system and a false positive caused by multiple concurrent requests is unlikely.

3.3.3 Requirements

The vCardTrek development is driven by the following requirements, which are derived from analyzing card data flows in real-world production environments:

1) The tool does not require any modifications to the guest OS or the application components of the target payment card processing system, and no additional software needs to be installed on the VMs on which the payment system runs.

2) The tool does not make any assumptions on the internal operations of the target payment application system being tracked other than the following: (a) the target application runs on a virtualized environment, and (b) credit and debit card numbers are transiently stored in memory in a particular form.

3.3.4 System Overview

To identify the trajectory of the card data flow, a payment request is sent to the entry

point of a payment card processing system, and vCardTrek is employed to determine the set of VMs and the corresponding processes exchanging network packets as a result of this request (Step 1 of Figure 3.1).

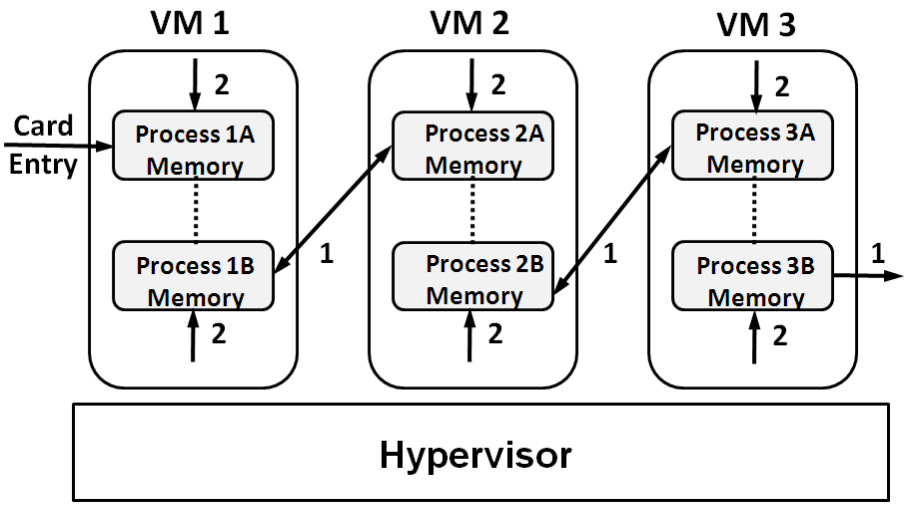

Figure 3.1 (1) Inter-VM network communications are tracked by vCardTrek, and (2) the memory of the interacting processes is inspected for card data.

Because network communications among payment system processes may be encrypted, it is not always possible to detect card data from intercepted network packets. Therefore, vCardTrek searches the memory spaces of the communicating processes for the card data as they travel from the entry-point process to other card data handling processes along the way (Step 2 of Figure 3.1). Even though card data may be encrypted during their IPC transmissions, they are decrypted and operated on during their processing, and therefore the clear text version of card data can be traced in the interacting processes' memory. Once the processes whose memory contains card data are found the machines involved in the card data flow are readily identified.

The card data flow trajectories from multiple VMs spread over several physical hosts can

be further concatenated to determine how card data flow among networked hosts within the organization (Figure 3.2).

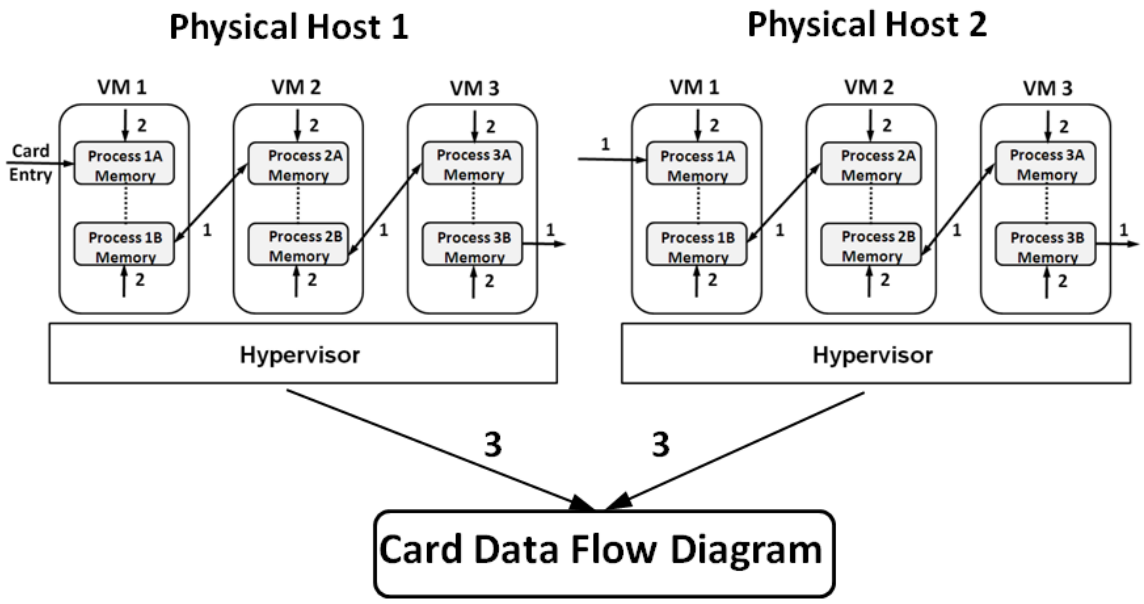

Figure 3.2 Card data flow concatenation from multiple physical hosts.

3.3.5 Main Components

We implemented the card data flow tracking tool for the Xen hypervisor and fully-virtualized (HVM) Windows-based VMs (payment card processing systems predominantly run Windows OS). In our implementation, we deploy vCardTrek in Dom0 and run the components of the payment card processing system in DomUs (Figure 3.3). The vCardTrek algorithmic outline comprises the following high-level steps:

(1) vCardTrek traces inter-VM TCP connections starting from the entry-point VM that receives the test input request;

(2) vCardTrek searches the memory space of communicating processes bound to the

intercepted TCP connections for the card number used as a test input at the entry-point VM;

(3) The card data flow path is reconstructed based on the results from (1) and (2).

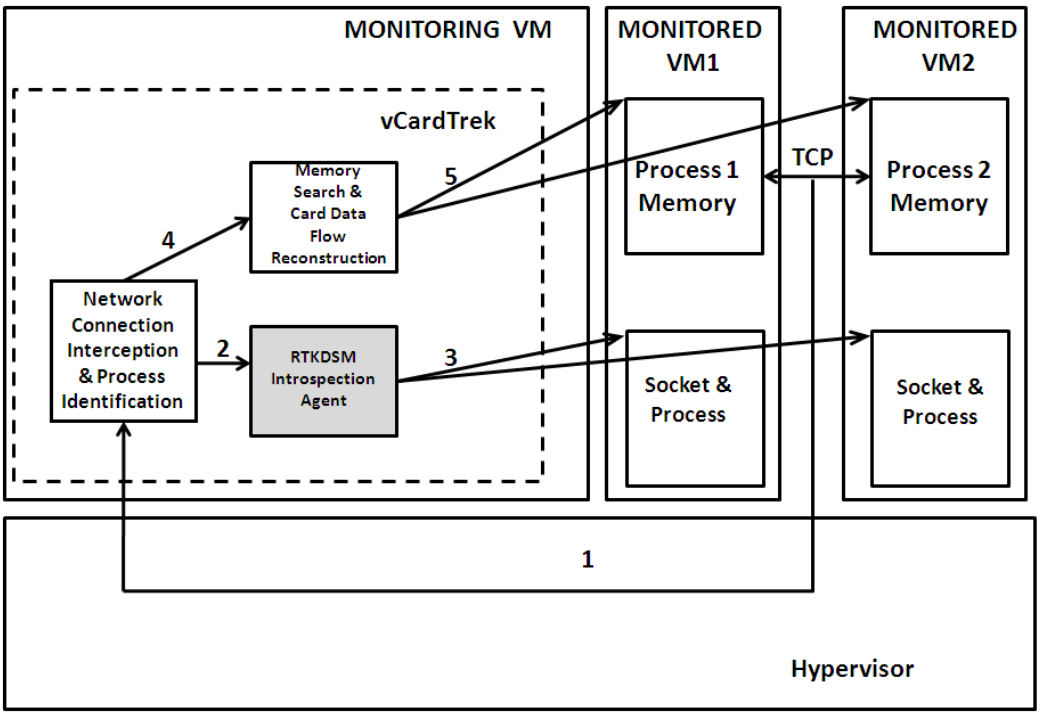

Figure 3.3 (1-2-3) Network connections are intercepted, and the processes participating in the network connections are determined; (4-5) the memory of the identified processes is searched for card data, and the card data flow is reconstructed.

3.3.5.1 Tracing of Inter-VM Communications

vCardTrek makes use of the packet filtering tool *ebtables* to intercept all packets sent to or from VMs. Ebtables is an open source utility that filters packets at an Ethernet bridge [45]. As of the Linux kernel 2.6, the ability to perform bridge mode filtering using ebtables is natively included in the kernel and supported by default. Through command line arguments, ebtables is instructed to pass intercepted packets to vCardTrek using netlink sockets. Tracing of inter-VM

communications begins with the entry-point process and continues recursively on each intercepted network connection.

When vCardTrek receives a packet from ebtables (Step 1 of Figure 3.3), it parses the packet to extract its source and destination MAC addresses and port numbers (src MAC, src port, dst MAC, dst port) from the packet header. The src and dst MACs are then resolved to the VM IDs using XenStore. In Xen, XenStore stores information about each VM during its execution including the VM IDs and the corresponding MAC addresses. vCardTrek initiates a VMI request to the RTKDSM introspection agent (Step 2 of Figure 3.3) to extract all open sockets for the source and destination VMs so it can identify the processes bound to the source and destination sockets (Step 3 of Figure 3.3). vCardTrek invokes VMI requests in a multi-threaded fashion and never blocks on these requests allowing the RTKDSM introspection agent to perform the VM analysis in parallel using separate threads. The summary of the data structures accessed by vCardTrek is provided in Table 3.1.

Table 3.1 The data structures accessed by vCardTrek.

Operating System Version	Data Structures
Windows XP Windows 2003	_ADDRESS_OBJECT' - socket _TCPT_OBJECT – TCP connection _EPROCESS - process
Windows Vista Windows 2008 Windows 7	_TCP_LISTENER - socket _TCP_ENDPOINT – TCP connection _EPROCESS - process

vCardTrek maintains a table of all the (src MAC, src port, dst MAC, dst port) connections being currently analyzed to avoid issuing redundant requests while a VMI request processing is in progress. Upon completion of the VMI request, the corresponding connection record is removed from this connection table.

3.3.5.2 Searching the Process Memory

vCardTrek identifies the portions of the VMs' memory space that belong to the identified processes, so that it can focus on those portions only, and searches these memory portions for the test card number used in the test transaction (Step 5 of Figure 3.3). vCardTrek starts with the entry-point process and continues recursively on each intercepted network connection.

The memory search is conducted using the following patterns. Payment card numbers are sequences of 13 to 16 digits. The card issuer is identified by a few digits at the start of these sequences. For instance, Visa card numbers have a length of 16 and a prefix value of 4. MasterCard numbers have a length of 16 and a prefix value of 51-55. Discover card numbers have a length of 16 and a prefix value of 6011. Finally, American Express numbers have a length of 15 and a prefix value of 34 or 37. Therefore, finding these card numbers in memory can be accomplished by searching for ASCII strings that match the following regular expression: ((4\d{3})|(5[1-5]\d{2})|(6011))-?\d{4}-?\d{4}-?\d{4}|3[4,7]\d{13}. However, sequences of 13 to 16 digits with proper prefix values are not always card numbers. Each potential card number obtained by the above search procedure has to be further verified using the Luhn algorithm [46], which is a simple checksum formula that is commonly used to validate the integrity of a wide variety of identification numbers.

When vCardTrek does not find the test card number in a process's memory, there are

three possible explanations. First, the process does not receive the test card data at all. Second, the process receives an encrypted version of the test card data, but does not decrypt it. Third, the process receives the test card data either in clear text or in encrypted form, but vCardTrek scans the process at an inopportune time, e.g. before the decryption of an encrypted card number or after the clear text card number is overwritten.

To increase the probability of card detection, vCardTrek scans each communicating process multiple times. The first scan examines every memory page in the process. If the card number is not found in the first scan, vCardTrek re-scans the memory. Each subsequent scan only inspects those memory pages that are modified since the last scan. We exploit the Xen's dirty page tracking capability to identify modified pages between consecutive scans. This incremental scanning approach significantly decreases the card number search overhead in subsequent scans. If no card number is found after a specified number of scans of a given process, vCardTrek assumes the process is not in the card data flow.

Just because no card number is found in a process does not mean that the process cannot be part of a card data flow. For example, the process can receive an encrypted card number and pass it on to the next process without decrypting it. Therefore, vCardTrek has to scan all communicating processes regardless of whether the sending process contains the test card number.

3.3.5.3 Card Data Flow Reconstruction

To build the card data flow, the processes whose memory contains the test card data and the communication connectivity among them are combined into a graph. When two processes of a payment card processing system communicate, there are three possible state combinations after

searching their memory pages, as shown in Figure 3.4(A): (1) The test card data found in the memory of both processes, (2-3) the test card data found in the memory of either process but not both, and (4) the test card data is found in the memory of neither process.

Similarly, when vCardTrek scans a process's memory in a VM that serves as a card receiver and as a card sender, there are three possible state combinations, as shown in Figure 3.4(B).

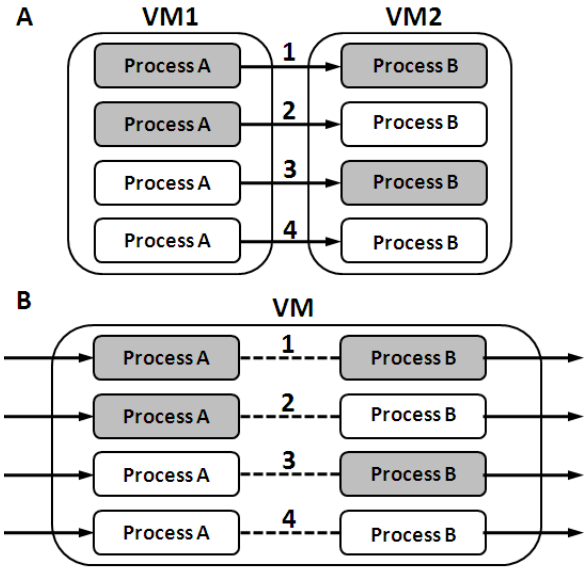

Figure 3.4 (A) 4 possible states of two inter-VM communicating processes (grey rectangle - the card number found in process memory, white rectangle - no card number found in process memory. The arrow indicates the direction of connection initiation, not traffic flow); (B) 4 possible states of processes within a VM at packet receiving time and at packet sending (the same process may serve as the receiving and sending process).

3.4 Evaluation

In this section, we describe experiments demonstrating distributed card data flow tracking using vCardTrek. We tested the tool on three payment card processing systems: two e-commerce

shopping carts and a point-of-sale system (Table 3.2).

Table 3.2 Evaluation suites and testing results.

System Name	AbleCommerce System	osCommerce System	CreditLine System
Software Description	Commercial shopping cart system used by > 10,000 stores worldwide [47]	E-commerce management software program [48] used by >12,000 online shops	Client-server application designed as point-of-sale system [49]
Language/Platform	ASP.NET/MSSQL	PHP/MySQL	Windows executable
DomU Client	Internet Explorer browser	Internet Explorer browser	Client application
DomU Server	IIS 5.1 web server with .NET framework v3.5	IIS 5.1 web server running PHP v5.3.3	Server application
DomU DB	MSSQL'05 Express Server	MySQL 5.1.52	N/A
Encryption in Transit	SSL	SSL	N/A
Results	The test card number was found in Client and Server DomUs.	The test card number was found in Client, Server, and DB DomUs.	The test card number was found Client and Server DomUs.
Average Time to Identify the Flow, sec	9	7	8

3.4.1 Card Data Flow Tracking Across Multiple VMs Hosted on the Same Physical Host

3.4.1.1 Experimental Setup

Our testbed consisted of a virtualized server that used Xen version 3.3 as the hypervisor and Ubuntu 9.04 (Linux kernel 2.6.26) as the kernel for Dom0. vCardTrek was installed in the Dom0. In addition, the virtualized server hosted three DomU domains (DomU Client, DomU Server, DomU DB) running Windows XP. The payment card processing systems were installed in these three domains as outlined in Table 3.2 and were running simultaneously to mimic the real-world production environments with multiple services running on the communicating hosts.

3.4.1.2 Experiments

When conducting our experiment, we selected several items for purchase and submitted credit card information at checkout. Following the payment card processing requests, vCardTrek determined the set of machines exchanging packets, identified the processes involved in these communications, and inspected the processes' memory for the card number used in the transaction, while allowing the applications to run throughout the analysis. The testing results are presented in Figure 3.5.

Additionally, we also captured network packets exchanged between machines to determine if an accurate card data flow could be built by only inspecting the contents of the sniffed packets without applying vCardTrek. As expected, we could not detect the test card number in the sniffed packets due to the SSL encryption configured on these communications (Figure 3.6, Figure 3.7, and Figure 3.8).

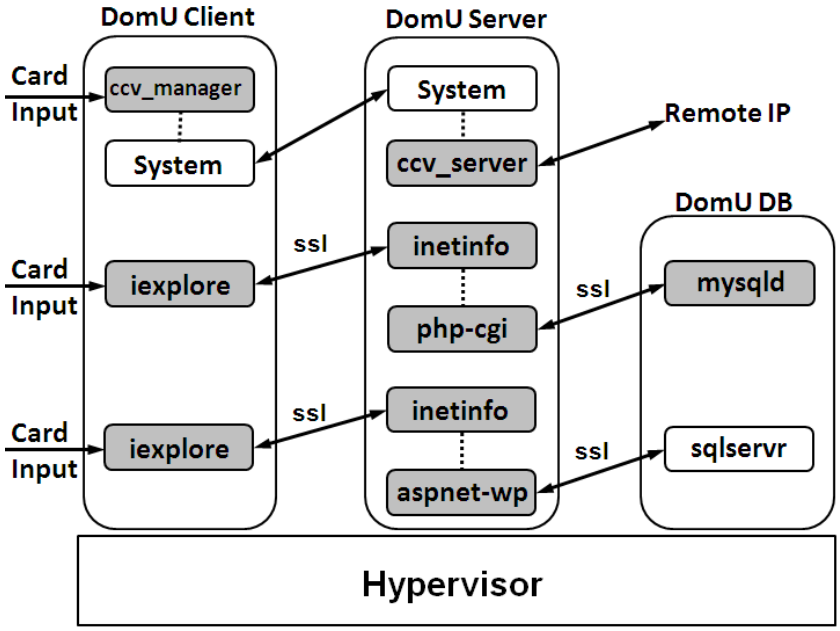

Figure 3.5 Processes involved in card data flow (CreditLine flow at the top, osCommerce flow in the middle, and AbleCommerce flow at the bottom).

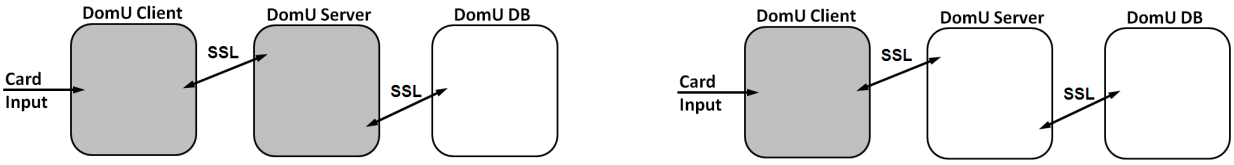

Figure 3.6 AbleCommerce Card Data Flow (machines found to participate in the card data flow are shown in grey) (left) using vCardTrek; (right) using a packet sniffer.

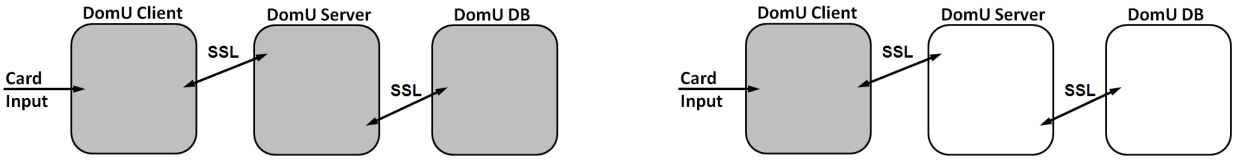

Figure 3.7 osCommerce Card Data Flow (machines found to participate in the card data flow are shown in grey) (left) using vCardTrek; (right) using a packet sniffer.

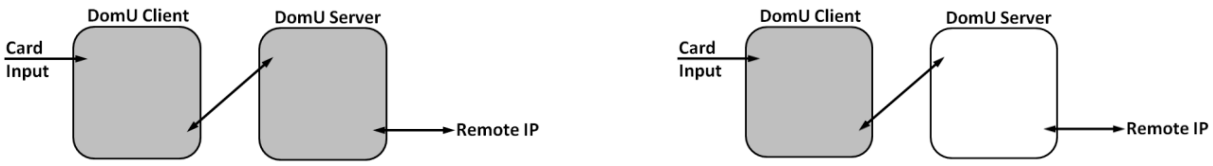

Figure 3.8 CreditLine Card Data Flow (machines found to participate in the card data flow are shown in grey) (left) using vCardTrek; (right) using a packet sniffer.

In some cases, vCardTrek was also able to identify other card related information including the card expiration date, CVV number, and the cardholder's name within the same memory segment as the corresponding card number as shown in Figure 3.9.

```
00CA7910  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00CA7920  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00CA7930  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00CA7940  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00CA7950  34 35 35 36 31 35 36 33  37 32 38 33 33 37 39 38  4556156372833798
00CA7960  00 00 00 00 30 34 31 32  00 00 00 00 34 35 30 2E  ....0412....450.
00CA7970  30 30 00 00 00 00 00 00  00 00 00 00 00 00 00 00  00..............
00CA7980  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00CA7990  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00CA79A0  00 43 56 56 32 20 47 4F  4F 44 20 4D 41 54 43 48  .CVV2 GOOD MATCH
00CA79B0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00CA79C0  33 35 34 00 00 00 00 00  00 00 00 00 00 00 00 00  354.............
00CA79D0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00CA79E0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00CA79F0  00 00 00 4A 6F 6E 20 4A  6F 6E 65 73 00 00 00 00  ...Jon Jones....
00CA7A00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

Figure 3.9 Detailed information uncovered about a test card, including the card number (4556156372833798), the card expiration date (0412), the CVV number (354), and the cardholder's name (Jon Jones) were identified within the process memory.

When running the tests, we observed the timings and the portions of memory from which card data were extracted and classified the card data extraction instances into four categories:

1) Transient/Stack: The card data were uncovered from a stack region while the associated transaction was being processed.

2) Persistent/Stack: The card data were uncovered from a stack region after the associated transaction was completed.

3) Transient/Heap: The card data were uncovered from a heap region while the associated transaction was being processed.

4) Persistent/Heap: The card data were uncovered from a heap region after the associated transaction was completed.

The successful card data extractions vCardTrek was able to perform against the test payment card processing systems fell into category (1), (3) and (4). Category (2) was rare because memory words allocated on the stack were automatically freed and possibly overwritten when they were no longer needed. In contrast, memory words allocated from the global heap had a much longer life time, because application programs needed to explicitly free them when they were no longer needed, but application programs rarely did so. As a result, card data stored on the heap existed for at least the duration of the associated transaction, which typically took up a few seconds to complete, and in many cases continued to exist even after the associated transaction is completed.

3.4.2 Card Data Flow Tracking Across Multiple VMs Hosted on Multiple Physical Hosts

All communications in the first experiment occur between VMs running on top of the same hypervisor, while in the real world the processes in a payment card processing system are more likely to reside in multiple VMs spread over multiple physical hosts. In the following experiment, we demonstrate the capability of vCardTrek to work equally effective in a multi-physical-host setting.

3.4.2.1 Experimental Setup

Our testbed consisted of three virtualized servers that used Xen version 3.3 as the hypervisor and Ubuntu 9.04 (Linux kernel 2.6.26) as the kernel for Dom0. vCardTrek was installed in the Dom0 on each physical host. Each physical host was running one DomU domain. The first virtualized server hosted DomU Client, the second virtualized server hosted DomU Server, and the third virtualized server hosted DomU DB all running Windows XP. The payment card processing systems were installed in these three domains as outlined in Table 3.2 and were running simultaneously to mimic the real-world production environment with multiple services running on the communicating hosts.

3.4.2.2 Experiments

When conducting our experiment, we selected several items for purchase and submitted credit card information at checkout. Following the payment card processing requests, vCardTrek determined the set of machines exchanging packets, identified the processes involved in these communications, and inspected the processes' memory for the card number used in the transaction, while allowing the applications to run throughout the analysis. The card data flow trajectories from the three VMs spread over three physical hosts were then concatenated to build the card data flow. The testing results are presented in Figure 3.10.

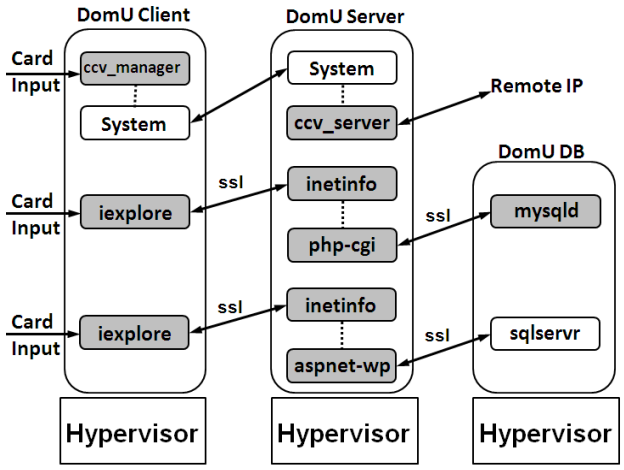

Figure 3.10 Card data flow across multiple VMs hosted on multiple physical hosts.

3.5 Limitations

The test environments used to date have been useful in demonstrating the vCardTrek effectiveness but they are rather simplistic and do not display many of the characteristics of large-scale deployments. Although we tested three different settings, they all involved just four processes distributed across three VMs interacting in almost identical fashion. Unlike the simple test scenarios described in this work where the number of factors influencing the correctness of the data flow reconstruction is minimal, the task of the card flow identification becomes increasingly more complex in real-world production setups. For instance, if two VMs communicate for reasons not related to payment data flow, such as periodic updates, heartbeats, replications, backups, other services running on the communicating hosts, and so on, then these connections may be mixed up with those for card data processing. These additional communications could significantly increase the workload of the card data flow tracking tool. Moreover, network delays may also critically affect the ability to track a card number within a process.

More complex environments may introduce a race condition that may affect the effectiveness of the system: the target process (which may be running on a different physical machine) may have completed its processing (all of it, or just the part that involves the card data in its unencrypted form) before vCardTrek on that physical machine manages to analyze the process' memory. Additionally, we have also assumed that the system is in a “quiescent” state where only one simulated transaction takes place at a time. This assumption is quite restricting from a practical point of view, given that in a production setting it would be quite difficult to ensure that there are no other ongoing transactions.

Finally, it is possible that a card number can be handled by processes in an encrypted form and is never decrypted during its processing, as revealed by some of our experiments. This issue will affect the accuracy of the derived data flow diagram vCardTrek produces.

3.6 Summary

This study presented the vCardTrek tool that automatically tracked card data flow of payment card processing applications running in a virtualized environment and identified the system components involved in card data processing. The primary use of this tool is to ensure compliance with Payment Card Industry Data Security Standard (PCI DSS) that has been widely adopted by commercial and financial institutions. The key features of vCardTrek include: 1) the ability to discover the card data flow of a distributed payment card processing system; 2) independence of applications and platforms; and 3) the ability to deal with communication protocols that encrypt messages. We have demonstrated the vCardTrek effectiveness by testing it with three different commercial applications, and vCardTrek successfully identified the correct card data flow for each tested application.

4 Cloud-Based Application Whitelisting

4.1 Introduction

A cloud service that has proven commercially significant, especially in the private cloud space, is virtual desktop infrastructure (VDI), which gives each end user a dedicated virtual machine (VM) as her desktop computer and manages these VMs in a centralized manner. By virtue of the centralized management architecture, VDI makes more efficient use of the underlying computing resources and enforces high-level security policies on these desktop VMs consistently and persistently.

As desktop computing is being virtualized, protection of desktop VMs also evolves from an agent-based approach, which installs the security agent inside every VM to be protected, to an agentless approach, which deploys the security agent on every physical machine on which the VMs to be protected run. The agentless approach not only greatly simplifies security agent maintenance and upgrade, but also effectively shields the agents from being attacked if the VMs are compromised [4].

A standard way for an attacker to take control of a victim user machine is to (1) hijack an existing application running on the machine, and (2) then download and execute additional malicious helper programs to actually perform damaging acts, such as stealing information or mounting attacks against other machines. Attackers perform the hijacking step by taking advantage of vulnerabilities in applications, e.g., buffer or integer overflow. Many solutions [50-52] have been proposed to deter such vulnerability-exploiting hijack attacks, but see limited commercial adoption. In contrast, mainstream anti-virus (AV) products are designed to stop the

"download and execute" step by creating a blacklist of known malicious programs and preventing an unknown program module from being loaded into an active address space if it matches any entry in the blacklist. This blacklisting approach is losing steam because new malware samples are programmatically generated from existing ones and as a result it is difficult if not impossible for AV companies to keep their blacklists up to date. Blacklisting is effective when there are fewer malicious programs than benign programs. Today, because the number of malicious programs is much larger than the number of benign programs, and the gap is widening, whitelisting, which prevents an unknown program module from being loaded into an active address space if it is not in a whitelist of known good programs, seems to be a more promising approach to keeping malicious helper programs out. In addition to defending against malware, an application whitelisting system could also be used to prevent illegal, pirate or personal software from running on corporate VMs assigned to employees.

This study describes the design and implementation of a cloud-based application whitelisting system called CLAW, which checks an executable file or a library module against a whitelist before it is loaded into the address space of a user process, and aborts the program load operation if the executable file or library module is not in the whitelist. Moreover, CLAW runs outside the VM on which the user process runs, and performs this check without installing any agent inside the VM. We have successfully implemented a CLAW prototype on the Xen hypervisor and targeted it at Windows and Linux VMs. The run-time performance overhead of out-of-VM application whitelisting is shown to be under 10% in this prototype.

4.2 Background

The design goal of CLAW is to detect when an executable file or library module is to be

loaded into a user process in a VM and check if the executable file or library module is in a white list. To motivate the design of CLAW, we start with a description of how the Windows OS and Linux OS load code into a user process's address space. We also describe Windows and Linux data structures that are relevant to the design and implementation of CLAW. It is mandatory to reconstruct these data structures in order to extract and analyze code regions in the process address space without having access to the APIs inside the VM.

4.2.1 Code Regions

Code regions in the address space of running processes are classified into the following three categories, summarized in Table 4.1, according to the type of sources used to populate them:

Table 4.1 Code source types in memory.

Source	Code Introduced Using
Binary File	1) a benign on-disk binary 2) a malicious on-disk binary
Private Allocation	1) native system calls 2) system call hooking techniques to prevent binary registration 3) remote thread injection
Other	1) hot-patching of the existing code 2) function-pointer hooking 3) modifying the return address on the stack

We discuss the three code source types for both Windows and Linux OS in more details below.

4.2.1.1 Code in File-Backed Address Space Regions

4.2.1.1.1 Windows OS

The application is typically made up of a base executable that loads library components containing additional functionality. The executable and the library components are represented by on-disk files that are mapped into a process's address space when the application is launched or during run-time. File-backed address space regions contain data from on-disk files. Windows OS differentiates between files that are mapped as data, and files that are mapped as executable images. The code that loads a file into memory has to specify whether the file is loaded as a data file or an image file. Data files have arbitrary content and structure and are simply mapped one to one to their address space regions. Image files, on the other hand, must be stored in the portable executable (PE) format and may contain data as well as executable code. A PE file contains several sections each with its own read/write permission characteristics. Data sections may be read-only or writable. Code sections in general are executable and read-only. Image files can be mapped as data files. However, if a data file is not stored in the PE format, the system loader will refuse to load it as an image. When a new process is started or a library component is loaded, the *NtMapViewOfSection* native system call is used to map a code section into a process's address space in memory. The description of the *NtMapViewOfSection* system call and its parameters is given in Table 4.2.

4.2.1.1.2 Linux OS

The binary loader maps the executable file along with the loadable segments of any required libraries into memory using the *mmap* system call. The *mmap* system call exposes an interface that allows for associating a memory range with a file descriptor. The description of the *mmap* system call and its parameters is given in Table 4.3.

Table 4.2 Windows system calls.

System Call	Parameters and their description
NtMapViewOfSection	(IN SectionHandle, IN ProcessHandle, IN OUT BaseAddress, IN ZeroBits, CommitSize, IN OUT SectionOffset, IN OUT PULONG ViewSize, IN InheritDeposition, IN AllocationType, IN Protect) SectionHandle – a handle to Section object, successfully created by a call to NtCreateSection or NtOpenSection ProcessHandle – a handle to the process that the view should be mapped into BaseAddress – a pointer to the variable receiving virtual address of mapped memory Protect – specifies the type of protection for the region, such as PAGE_EXECUTE_READWRITE
NtProtectVirtualMemory	(IN ProcessHandle, IN BaseAddress, IN NumberOfBytesToProtect, IN NewAccessProtection, OUT OldAccessProtection) ProcessHandle – a handle to the process that the protection should be set for BaseAddress – a pointer to base address to protect NumberOfBytesToProtect – a pointer to size of region to protect NewAccessProtection – specifies the type of protection for the region, such as PAGE_EXECUTE_READWRITE
NtAllocateVirtualMemory	(IN ProcessHandle, IN OUT BaseAddress, IN ZeroBits, IN OUT RegionSize, IN AllocationType, IN Protect) ProcessHandle – a handle to the process to allocate memory in BaseAddress – a pointer to a variable that will receive the base address of the allocated region of pages. RegionSize – a pointer to a variable that will receive the actual size of the allocated region of pages Protect - specifies the type of protection for the region, such as PAGE_EXECUTE_READWRITEs

Table 4.3 Linux system calls.

System Call	Parameters and their description
mmap	(void *start, size_t length, int prot, int flags, int fd , off_t offset) mmap function asks the system to map *length* bytes starting at *offset* from the file specified by the file descriptor *fd* into memory, preferably at address *start*. If *start* is 0, mmap returns the actual place where the object is mapped. *prot* describes the desired memory protection and can be a bitwise-or of the values PROT_NONE / PROT_READ / PROT_WRITE / PROT_EXEC
mprotect	(const void *addr , size_t len, int prot) mprotect changes protection for the calling process's memory page(s) containing any part of the address range in the interval [*addr*, *addr+len*-1]. *prot* describes the desired memory protection and can be a bitwise-or of the values PROT_NONE / PROT_READ / PROT_WRITE / PROT_EXEC

4.2.1.2 Code in Private Address Space Regions

4.2.1.2.1 Windows OS

Private address space regions are created through dynamic memory allocation calls and contain volatile data, which only exist when the hosting process is alive. Two types of code exist in private address space regions: dynamically generated code and injected code. Dynamically generated code is created by the process itself at run time while injected code is forcibly loaded into a process' address space by another process. Examples of applications that may generate dynamic code include just-in-time (JIT) compilers, interpreters, and executable unpackers.

To create code in private address space regions, Windows applications first allocate new address space regions by calling the *NtAllocateVirtualMemory* system call with proper read/write/execute permission setting, and later follow by writing code into the allocated regions. Setting these regions to be writable is necessary because the code may modify itself as it is being executed. In addition, applications could use the *NtProtectVirtualMemory* system call to modify

the permissions of private address space regions later on. The usage of these two system calls is given in Table 4.2.

While dynamic code generation sees a great deal of legitimate usage, code injection is almost exclusively used by malicious programs. A common code injection attack is to inject a user space rootkit code into the address space of a system daemon process. There are three common code injection attacks:

1) *Hooking the Native Loader:* Assume a user space shell code is placed on a victim host via an initial exploitation. The shell code then hooks the file loading system call to trick the dynamic loader into loading a malicious binary in memory rather than from an intended file on disk [53]. Because the malicious binary is registered with the victim process, a query for modules loaded into the victim process allows for detection of the injected code.

2) *Reflective Library Injection:* The code loaded through a remote exploitation contains a minimal PE loader that can load additional code without relying on the native loader [54]. Because the native loader is not involved, the loaded code is largely undetectable to the operating system and the hosting process. The only indicator that the loaded code exists is a chunk of private address space region is allocated with read/write/execute permissions.

3) *Remote Thread Creation:* The Windows API *CreateRemoteThread* allows a process to start a thread in another process. Common use cases of this system call include injecting a thread into a remote process being debugged to issue a breakpoint or injecting a thread into a process to query heap or other process information. Using

this API, a malicious process starts a new thread in a victim process by passing it the address of a piece of code already injected into the victim process.

4.2.1.2.2 Linux OS

To create code in private address space regions, Linux applications first allocate new address space regions by calling the *mmap* system call with read, write, and execute memory protection flags and the anonymous flag not tired to a file descriptor, and later follow by writing code into the allocated regions. In addition, applications could use the *mprotect* system call to modify the permissions of private address space regions later on. The usage of these two system calls is given in Table 4.3.

4.2.1.3 Other

Code may also be introduced via run-time overflow attacks that alter the execution path through hot-patching of existing code or control-sensitive data structures, e.g., changing a return address or a function pointer by overflowing a buffer. CLAW does not provide protection against attacks using these types of code.

4.2.2 Relevant Kernel Data Structures

4.2.2.1 Windows OS

EPROCESS: The kernel creates an EPROCESS data structure for each running process to hold a variety of information about the process. EPROCESS structures for all active processes are linked in a doubly linked list (Figure 4.1). The PsActiveProcessHead kernel symbol points to the doubly-linked list of EPROCESS structures. The PsActiveProcessHead pointer includes two pointers, a forward (Flink) pointer and a backward (Blink) pointer. The Flink pointer points to

the active process links of the first EPROCESS. The Blink pointer points to the active process links of the last EPROCESS structure in the active process list.

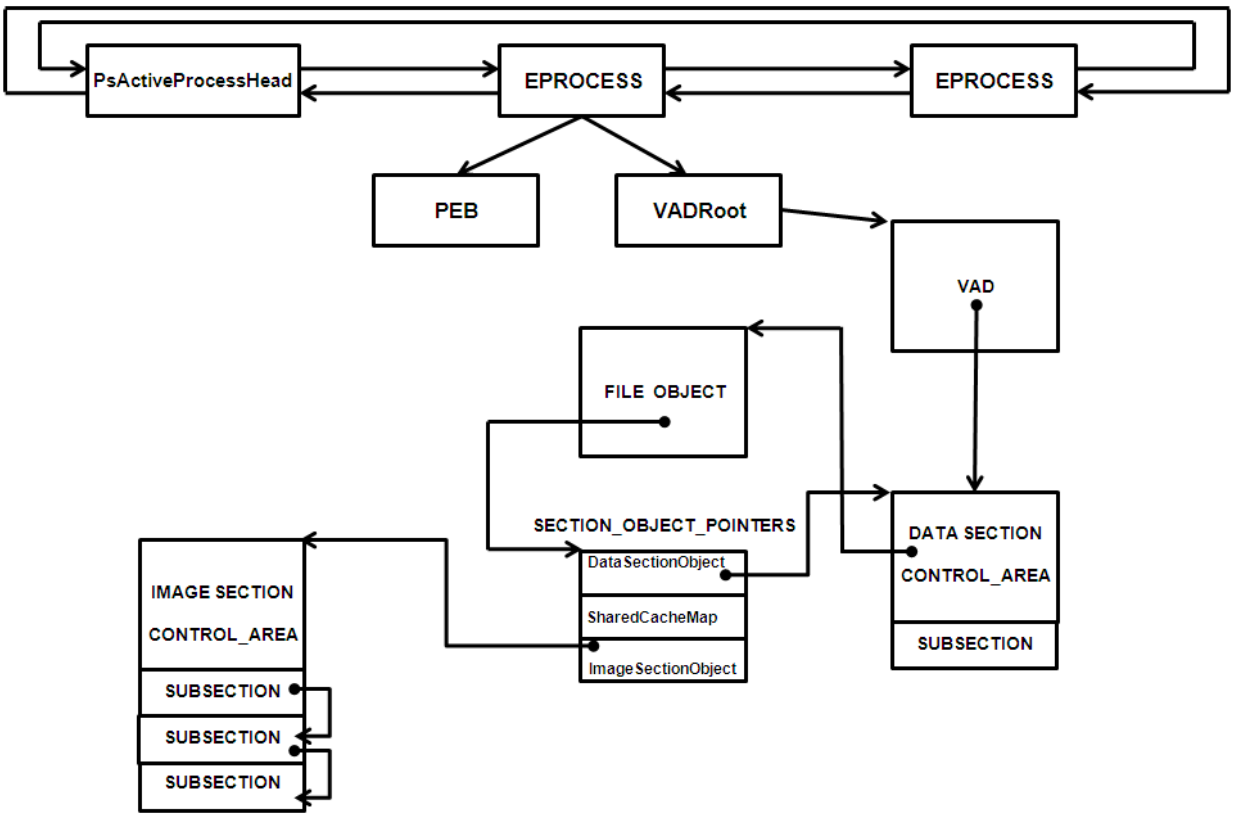

Figure 4.1 Windows code and memory management data structures.

PEB: The process environment block (PEB) component in a process's EPROCESS data structure contains a pointer to the virtual address of the memory-mapped PE image of the program loaded into the process, and a pointer to the virtual address location of the PEB_LDR_DATA object that maintains information about all DLLs loaded into the process (Figure 4.2) [55]. PEB is actually stored in a process's user address space rather than in the kernel because it needs to be modified in user space. PEB_LDR_DATA contains pointers to doubly-linked lists of loaded modules that are sorted in load order (InLoadOrderLinks), in

memory order (InMemoryOrderLinks), and in initialization order (InInitializationOrderLinks). PEB_LDR_DATA is modified as modules are loaded or unloaded. Each loaded module is represented as a LDR_DATA_TABLE_ENTRY structure, which is an element of a doubly-linked list of loaded modules, and contains details about the module name, base address and size.

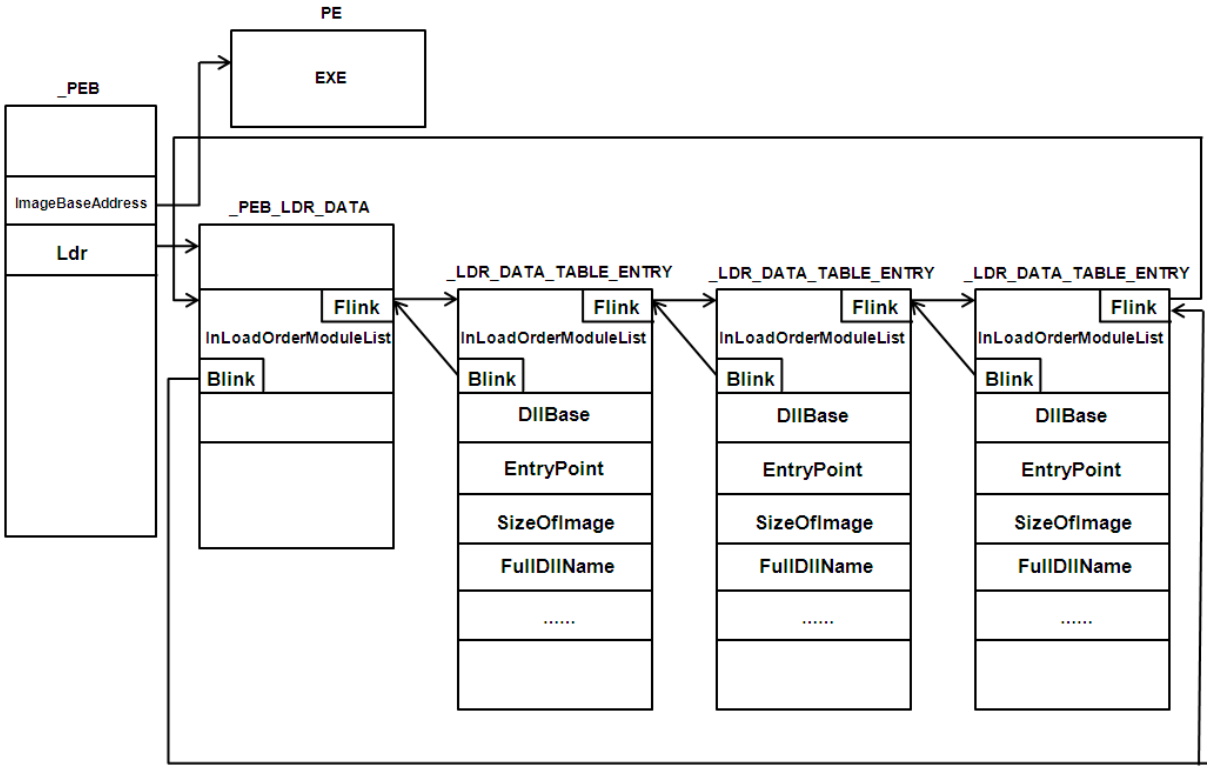

Figure 4.2 The PEB data structure.

VAD: For each block of consecutive memory addresses that share the same memory-related settings, Windows maintains a virtual address descriptor (VAD) entry storing the following information: start and end addresses, protection settings (read-only, writable, executable), data source type (file-backed memory or private address space), information about the associated file (if file-backed memory). All entries for a process are aggregated in a VAD tree (VadRoot) (Figure 4.1).

CONTROL_AREA: A VAD object points to a CONTROL_AREA object that stores detailed information about different subsections of a file.

SUBSECTION: For each mapped file, there are one or more SUBSECTION objects which store important mapping data. For data files, there is normally only one subsection, since the complete address range has the same characteristics, but for image files, multiple subsections may exist: one for each PE section plus one for the PE header. This is due to different characteristics of PE sections, e.g. some may be read-only while others are writable or executable. Each subsection contains a pointer to the next subsection.

FILE: FILE object is used by Windows to track a single open instance of a file. The file object contains a pointer to the Unicode name of the file. Another most important pointer is the SECTION_OBJECT_POINTERS field described next.

SECTION_OBJECT_POINTERS: Due to the different mapping and usage characteristics of data files and image files, different control areas are used. If, for example, a file is first mapped as a data file, a corresponding data section control area is created. If then the same file is mapped as an image, an image section control area is created as well. Both objects are of the same type except that for data files normally only one subsection is created, while for image files the number of subsections equals the number of PE sections in the related file plus one for the PE header. In fact, Windows internally maps each executable, which is about to be loaded first as a data file and then in a second step as an image. This results in the creation of two different control areas, from which either is used depending on the type of the created view. To maintain these different control areas per file, in the file object Windows stores one unique array for each opened file that contains pointers to the related data and image control areas. Either of

these two pointers may be zero, but not both of them. This array is called SECTION_OBJECT_POINTERS and is pointed to by each file object. The SECTION_OBJECT_POINTERS structure contains three pointers as seen in Figure 4.2. The first is called the DataSectionObject, the next is called the SharedCacheMap, and the final pointer is called the ImageSectionObject. DataSectionObject and ImageSectionObject are related and are actually pointers to the data and image control areas correspondingly. The SharedCacheMap is a pointer to the SHARED_CACHE_MAP structure, which is used by the operating system to maintain the cache.

4.2.2.2 Linux OS

TASK_STRUCT: The kernel creates a task_struct for every process running on a Linux system. The task_struct structure holds information about the current state of the process (Figure 4.3). task_struct structures for all active processes are linked in a doubly linked circular list. The global variable init_task is of type task_struct and represents the head to the doubly-linked list of task_struct structures. The init_task includes forward and backward pointers. The forward pointer points to the active process links of the first task_struct. The backward pointer points to the active process links of the last task_struct structure in the active process list.

VM_AREA_STRUCT: The vm_area_struct descriptor (similar to VAD in Windows OS) represents a memory region owned by the process and contains the start and the end addresses of the region. All vm_area_struct structures are linked together in an address-ordered singly linked list. Each vm_area_struct points to the associated mm_struct structure (similar to VadRoot in Windows OS) that describes a process' address space. There is only one mm_struct per process shared by all user-space threads. The vm_file field of each memory region descriptor

contains the address of a file object for the mapped file; if that field is null, the memory region is not associated with a file.

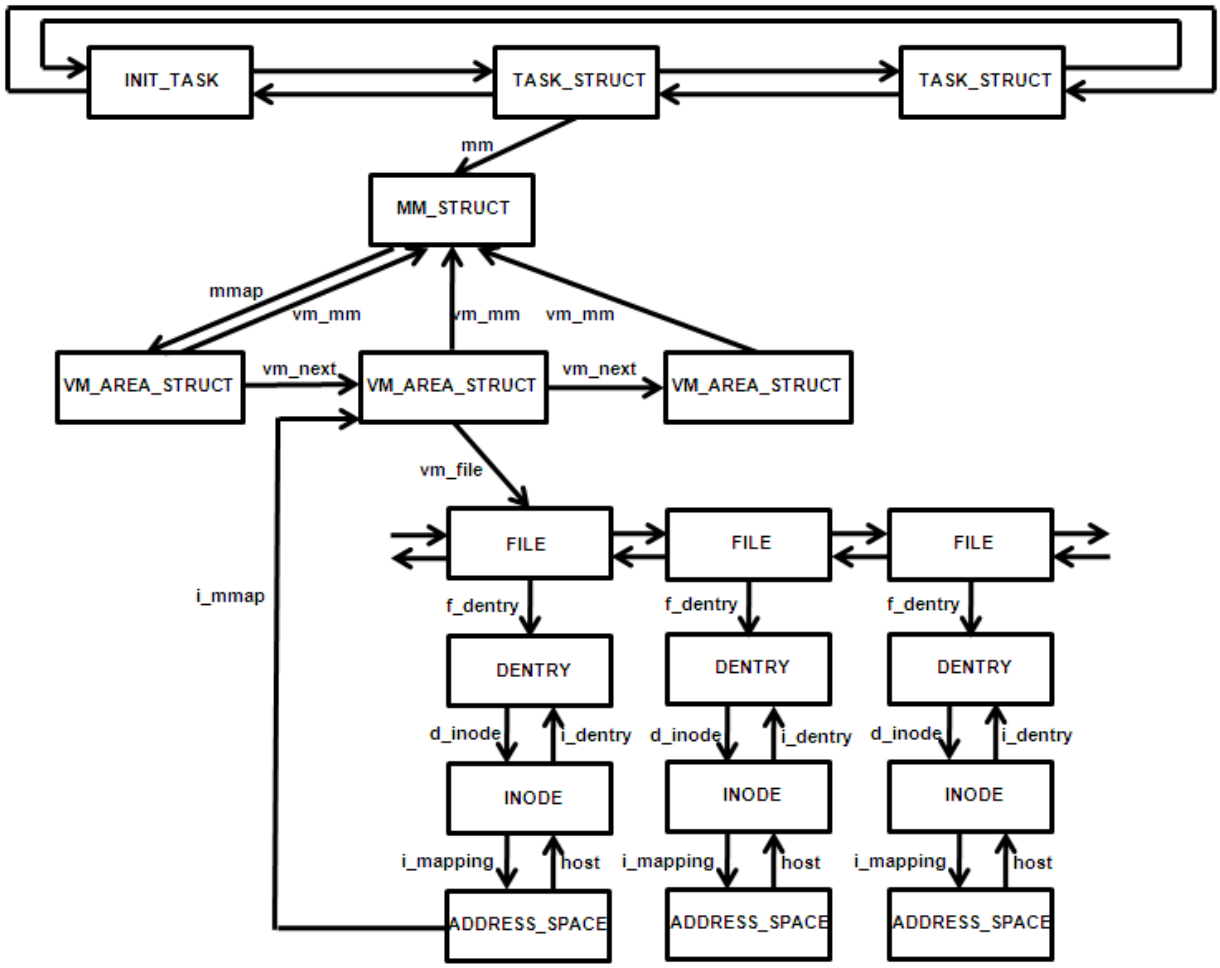

Figure 4.3 Linux code and memory management data structures.

FILE: The file object contains fields that allow the kernel to identify both the process that owns the memory mapping and the file being mapped. The file structure includes a pointer to the dentry data structure.

DENTRY: Dentry structures are created by the virtual file system to represent a directory

entry (directory or file). The dentry structure contains the name of the directory entry and a pointer to the inode structure.

INODE: The mapped file is identified by the inode data structure, which is an in-memory representation of a disk inode. The i_mapping field of each inode object points to the address_space object of the file.

ADDRESS_SPACE: The address_space structure represents the virtual memory image of the file and holds the search tree of pages for a file. The address_space structure allows for ordered enumeration of all physical pages pertaining to an inode. In turn, the i_mmap field of each address_space object point to a vm_area_struct data structure. While a single file may be represented by multiple vm_area_struct structures corresponding to the file portions mapped by multiple processes into their address space, there is only one address_space structure for the file no matter how many processes have mapped a particular file.

4.2.2.3 System Call Table Structures

The function pointers (addresses) of individual system calls exported by the kernel are stored in the system call table (Figure 4.4). In Windows OS, the system call table is represented by the system service dispatch table (SSDT) data structure. In Linux OS, the system call table is represented by the sys_call_table data structure. When an application makes a system call, it places the associated system call number in the EAX register, which is used as an index into the system call table. Each system call pointer in the table is four byte long. Thus, to get a system call offset into the system call table, the system call number in the EAX register is multiplied by 4. The address stored at the calculated offset points to the actual system call function in the kernel address space in memory.

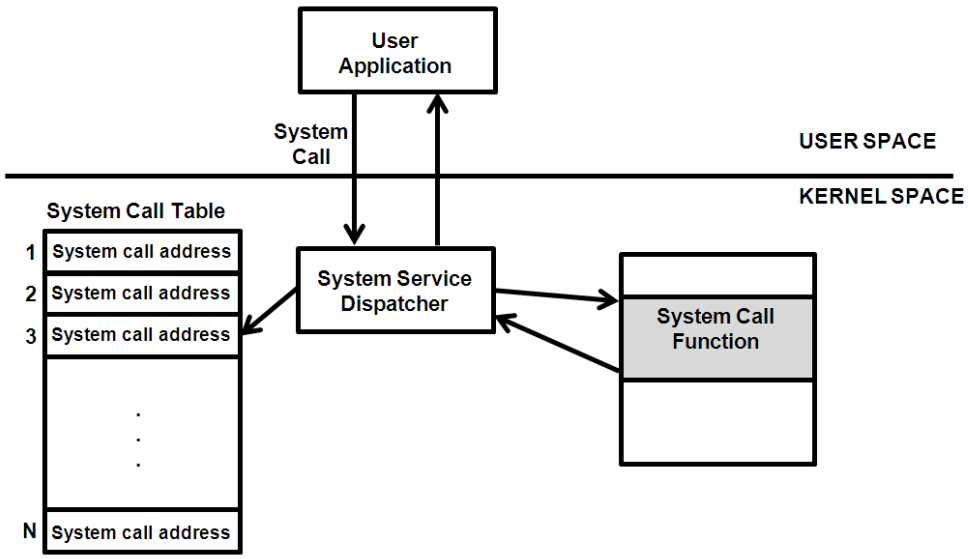

Figure 4.4 System call dispatching.

4.2.3 System Call Interception in Xen Hypervisor

User applications invoke system call requests by either executing software interrupts (INT 0x2E in Windows OS and INT 0x80 in Linux OS) or by the fast system call entry mechanism using the Intel SYSENTER/SYSEXIT or AMD SYSCALL/SYSRET instruction pairs. The fast system call entry mechanism was introduced due to performance issues on Pentium processors with the software interrupt method. All Windows versions starting with XP and Linux kernels starting with 2.6 use the fast system call entry method.

On a Xen para-virtualized platform, capturing system calls and their arguments is straightforward. Each trap from a DomU transfers control to the hypervisor, which forwards the trap to the Dom0 domain. However, the situation is more complex on an HVM platform. On such a platform, traps are directly forwarded to the kernel of the HVM by the hardware without

the involvement of the hypervisor. Fortunately, it is still possible to capture system calls on the HVM platform, although differently on AMD and Intel hardware. When an application needs to execute a system call, it normally specifies the requested system call number in the EAX register and a pointer to the user stack in the EDX register and then issues the SYSENTER instruction. The SYSENTER instruction passes control to the address specified in the model specific register (MSR) containing the entry point of the system call handler. Execution of this instruction results in transition into kernel mode. Once in kernel mode, the system call number is read from the EAX register and is looked up in the system call table. On the Intel platform, interception of system calls can be achieved by guaranteeing that the MSR points to an unmapped memory address, causing a trap to the hypervisor by a page fault. Conversely, AMD supports control flags that can be set to trigger transfers to the hypervisor on system calls. The hypervisor then forwards the relevant information, such as the values of the registers containing the system call number and parameters, to the Dom0 domain for system call processing. While these methods of interception are certainly effective, they introduce performance overhead because they require that every system call trigger an exit to the hypervisor.

4.3 Related Work

4.3.1 Code Verification Systems

Several studies have explored the problem of code verification in memory of running processes using the out-of-VM approach (Table 4.4). These can be roughly divided into (1) periodic code verification methods that periodically check the static code portions of the running program to detect if the program has been tampered with; (2) continuous run-time code integrity verification methods to detect code tampering attempts; (3) on-demand code verification

methods to ensure only approved code is allowed to be loaded by a program into the process' address space at load-time. All these methods work by calculating hashes of sections of memory, such as kernel text or user program memory during a known good state that are then used as a comparison baseline at the time of code verification.

Table 4.4 Code verification systems.

System Name	User Space (U) / Kernel Space (K) Monitoring	Code Verification Type	Virtualization Type
CLAW	U	On-demand	Full virtualization
Livewire	U	Periodic	Software-based (Type 2)
Copilot	K	Periodic	Coprocessor-based
SBCF	K	Periodic	Full virtualization
NICKLE	K	Continuous run-time	Emulator, Software-based (Type 2)
Secvisor	K	Continuous run-time	Custom-made hypervisor
Manitou	U/K	Continuous run-time	Full virtualization
Patagonix	U/K	Continuous run-time	Full virtualization
HIMA	U	On-demand	Para-virtualization
X-Spy	K	On-demand	Para-virtualization

4.3.1.1 Periodic Code Verification

The Livewire intrusion detection system used an integrity checker to detect if a running user-level program had been tampered with by periodically computing a hash of the immutable sections (.text) of a running program, and comparing it to a known good hash [4].

The Copilot integrity monitor implemented a detection strategy based on MD5 hashes of the host kernel's text, the text of any loaded kernel modules, and the contents of some of the host

kernel's critical data structures [16]. Copilot calculated "known good" hashes for these items when they were believed to be in a correct, non compromised state. The Copilot monitor then periodically recalculated these hashes throughout host kernel run-time and watched for results that differed from the known good values to detect cases where a rootkit had modified some of the kernel's existing executable instructions.

Similarly to the Copilot approach, state-based control flow integrity (SBCFI) monitor kept a copy of the kernel code's hash, and at each control flow integrity check, it made sure the kernel's code had not been modified by comparing it against the "known good" hashes [56].

The periodic nature of this group of methods introduces the possibility of evasion. An attacker can modify the code and revert back to the original code between two consecutive checks without the security monitor detecting the code tampering.

4.3.1.2 Continuous Run-Time Code Verification

A hypervisor-based NICKLE was developed to transparently prevent unauthorized kernel code execution [57]. NICKLE computed a priori off-line cryptographic hash of the kernel's code and on each VM startup performed the authentication of the loaded kernel code by comparing it with the known correct value. The authenticated kernel code was copied into a shadow physical memory of the target VM that was not accessible from within the VM. If the hash values did not match, the kernel module's code was not copied into the shadow memory. At run-time each kernel instruction fetch was verified by comparing the shadow memory maintained by the hypervisor with the actual physical memory at that location. Any differences indicated the presence of a rootkit, and thus the code was prevented from executing on the guest system. Linux kernel modules (LKMs) also required authentication before their insertion since NICKLE could

not distinguish between a valid and a malicious kernel module. A disadvantage of this kind of authentication scheme was that it needed to be manually performed every time a module was inserted into the kernel, and in-depth analysis was necessary to ensure that the LKM did not invalidate the kernel.

A small hypervisor system SecVisor was proposed to enforce the write+execute property of memory pages of the VM with the goal of preventing unauthorized code from running with kernel-level privileges [58]. The write+execute property stated that the pages of kernel memory could be either writable or executable, but never both. SecVisor used a white-list based approval policy containing “known good” SHA-1 hashes of all kernel runtime code to allow loading of kernel code at runtime. All code that was attempted to be loaded into kernel memory from the time the kernel was started was checked against the whitelist approval policy. SecVisor required modifications of the kernel code and thus did not support closed-source OSes. Moreover, SecVisor was not able to function if the OS kernel had mixed pages that contained both code and data.

Litty and Lie proposed a hypervisor-based system, called Manitou, for validating the executing code of both user applications and the kernel within a guest VM [7]. The hypervisor maintained a list of cryptographic hashes of the in-memory representations of application and kernel-level code pages that might be run within the VM. Manitou authenticated executing code by taking a cryptographic hash of the content of a page right before executing code contained on that page. Only pages that matched those in the trusted list were allowed to execute.

A hypervisor-based Patagonix system based on Manitou was designed to detect rootkits that avoided tampering with files on disk by injecting malicious code into binaries as they ran.

Patagonix identified covertly executing binaries by inspecting the code as it executed in memory and verifying the integrity of the executing binaries [10]. The executing code was identified using a trusted external database that contained cryptographic hashes of binaries. Patagonix compared the executing binaries reported by the OS with the good known binaries it identified and reported any discrepancies to the administrator. Patagonix did not handle the on-demand loading of running programs to measure them in their entirety.

As the continuous run-time code verification methods employ the VM executable memory protection, this approach may lead to spurious page faults impacting the performance of the system.

4.3.1.3 On-Demand Code Verification

The goal of CLAW is to track on-demand code loading events and to perform verification of the loaded code prior to its first execution. This objective is related to the group of methods that focus on providing code integrity measurements by actively monitoring system events.

A hypervisor-based HIMA was developed to measure the integrity of VMs running on top of the hypervisor by measuring user-level programs to be loaded into the guest VM and validating the integrity of the measured programs throughout the program execution [59]. HIMA monitored all the system calls that changed the VM's program memory layout, including loading and removing kernel modules, creation and termination of user processes, and loading and unloading of libraries. On intercepting the appropriate event, HIMA computed the SHA1 hash of the program code and initial data segment as they got loaded into the memory. HIMA completed all its measurements before the control jumped to the loaded program to guarantee that no instruction ran inside the system before being measured. After measuring the program, HIMA

added a new entry to the measurement list, and ensured consistency of the integrity measurement of user programs by capturing any attempt to modify measured programs throughout their execution. HIMA measured para-virtualized Linux systems only.

A hypervisor-based X-Spy system was implemented as an intrusion detection and protection framework [60]. One of the X-Spy's functions was to monitor system calls within a Linux OS for the purpose of protecting the integrity of the kernel. System calls were traced using the INT 0x80 instruction interception. X-Spy used a whitelisting technique by which all kernel modules allowed to be loaded were explicitly specified along with their respective SHA-1 hash values. If the module or binary to be loaded at run-time was not specified in the whitelist or if it had an incorrect hash value, X-Spy prevented it from being loaded by preventing the system call from reaching the VM kernel space. The memory scanning technique was used to computer the hash of the binary that involved loading the complete .text and .data sections of a binary into memory by setting the program counter to the next page and asking the VM kernel to load the page, and then hashing it while handling the page fault. If the hash could not be verified, the hypervisor invalidated all of the memory and returned the control back to the guest domain. Because of the invalid .text section to which the VM pointed, the process crashed.

The scope of the above tools was limited to para-virtualized VMs only whereas the CLAW was specifically designed for fully-virtualized VMs.

4.3.2 System Call Interception Systems

A number of systems have been developed for detection of malicious processes by analyzing system calls (Table 4.5). We cover the related work including hardware emulators, para-virtualized systems, and fully virtualized systems.

Table 4.5 Comparison of system call monitoring systems.

System Name	Virtualization Type	System Call Interception Mechanism
CLAW	full virtualization	System call table + MSR invalidation
Ether	full virtualization	MSR invalidation
VMScope	emulator	Instruction tracking
TTAnalyze	emulator	Instruction tracking
XView	emulator	Instruction tracking
Onoue et al. [61]	para-virtualization	Guest OS binary code patching & Native system call trapping chain
Xenini	para-virtualization	Native system call trapping chain
HIMA	para-virtualization	Native system call trapping chain
X-Spy	para-virtualization	Native system call trapping chain

4.3.2.1 Hardware Emulators

Out-of-VM system call tracing has been employed in emulator-based environments for malware analysis to identify malware startup mechanisms, command and control channels, and access to sensitive information. Examples of such systems include VMScope [62], TTAnalyze [63], and XView [64], which are based on dynamic binary translation technique of QEMU [41].

TTAnalyze automated the process of analyzing a malware process where the malware under analysis was executed inside an emulator environment, and relevant Windows API and native system calls were tracked and logged. The instruction pointer value of the virtual processor was compared to the start addresses of all operating system functions to determine the exact system function invoked by the malware process. TTAnalyze monitored the CR3 register value to determine whether or not the system call invoked by current instruction belonged to the malware process.

An emulator based system VMScope allowed viewing of the system call events of a VM-based honeypot by intercepting and interpreting the parameters and return values of various internal system calls invoked inside the VM.

XView used a dynamic cross-view based approach to detect processes hidden by rootkits. In order to identify a rootkit process, XView dynamically maintained a list of active processes built outside the monitored VM and comparing it with the list reported by the guest system. The outside view containing active processes was constructed by intercepting low-level system calls used to create and terminate processes and interpreting system call arguments and the return values of these system calls.

4.3.2.2 Para-Virtualized Systems

The code verification systems, HIMA and X-Spy, described earlier also made use of system call monitoring to detect code loading events.

Onoue et al. [61] proposed a security system that controlled the system call execution of processes using the para-virtualization version of Xen to intercept events related to system calls. The hypervisor intercepted system calls invoked by processes in the monitored VM and restricted their execution based on the security policy defined by a user. When a system call invoked by a process matched with an allow-rule in the security policy, it was allowed to execute. Otherwise, a system call violating the security policy was forced to fail.

Xenini was developed as a system for detecting intrusions in the para-virtualized XEN hypervisor by intercepting and analyzing system call traces [65]. Xenini disabled the fast system calls facility and used the 0x80 software interrupt to intercept system calls.

4.3.2.3 Fully Virtualized Systems

The Ether analyzer was developed for malware analysis on fully virtualized hardware platforms utilizing hardware virtualization extensions [66]. Ether was able to trace all system call executed by the target OS by exploiting the x86 fast system call entry mechanism. The performance evaluation of the system showed that tracing added extra latency to system calls, however, the majority of this latency was due to notifications of the Ether user space component and a full in-hypervisor implementation would have had much lower latencies.

4.4 System Architecture

4.4.1 Overview

CLAW assumes that the administrator has determined a set of approved permitted executable files and library modules and then prepared a whitelist that consists of the SHA-1 cryptographic hash values of these executable files and library modules (Step 1 of Figure 4.5). At run time, CLAW intercepts every program load operation in the VMs that it protects, applies the SHA-1 function to the executable file or library module being loaded, and uses the resulting SHA-1 value to look up the whitelist. Creation and maintenance of a whitelist according to its list of allowed programs is actually non-trivial, especially in the face of constant software patches and upgrades, and growing sophistication of software installation. But this issue is outside the scope of this study.

The design of CLAW should attain the following functionalities outside the monitored VMs:

1) Detecting new loaded programs in monitored VMs before they are executed in

monitored VMs,

2) Checking the hash values of loaded programs against the whitelist, and

3) Aborting the processes holding loaded programs if the whitelisting checks do not go through.

The Issue 1 could be addressed by intercepting system calls associated with specific program loading operations. However, because the performance overhead of system call interception may be substantial, monitoring kernel or processor data structures as a result of program loading operations may be more efficient.

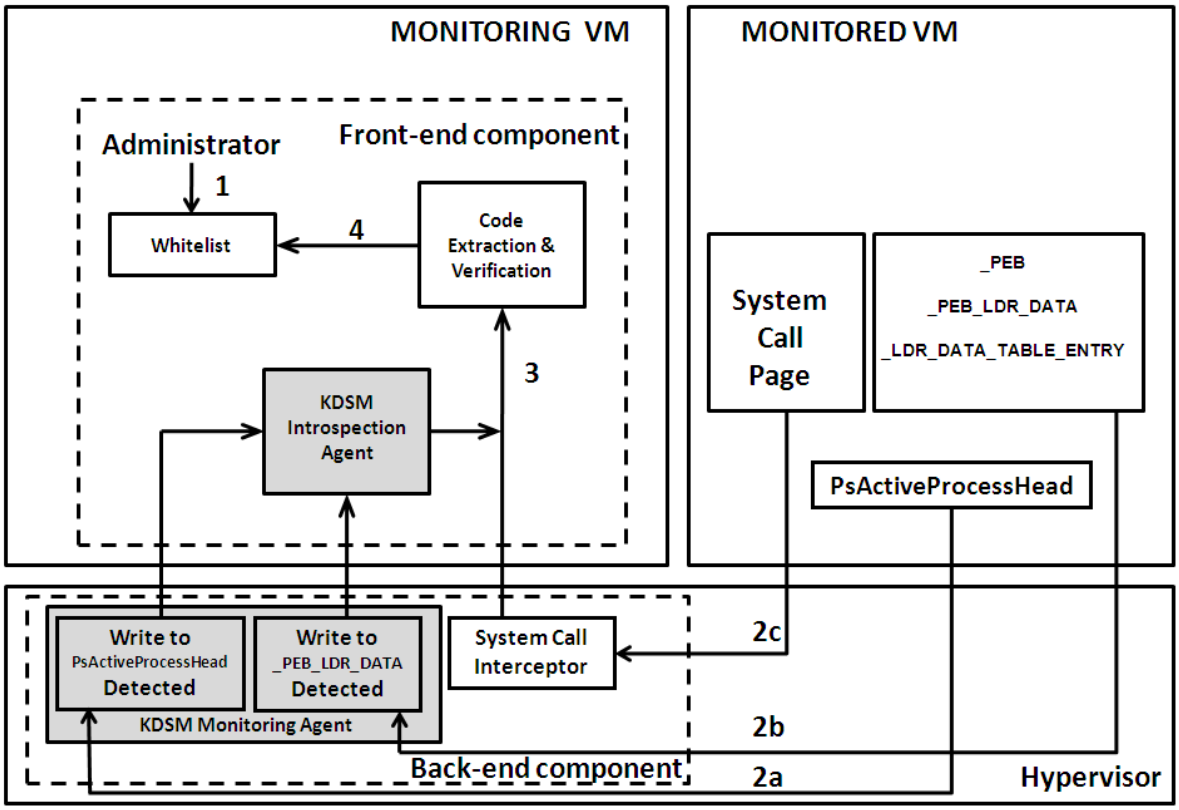

Figure 4.5 The CLAW architecture.

For Issue 2, CLAW computes a loaded program's hash value by applying the SHA-1 hash function to the in-memory PE/ELF image rather than the on-disk PE/ELF file of the loaded program. It would have been very difficult to access the on-disk files of loaded programs without installing any agent in the monitored VMs.

For Issue 3, to simplify the interaction between CLAW and the monitored VMs, CLAW aborts a process holding an illegitimate loaded program by zeroing out the address space region holding the loaded program. This approach is simple and effective, and does not require any cooperation from monitored VMs.

As shown in Figure 4.5, CLAW is composed of a front-end component running in a monitoring VM and a back-end component running inside the hypervisor. The back-end component of CLAW suspends a monitored VM when detecting a new loaded program in a user process running in the VM. After suspending a VM, the back-end component notifies the front-end component to extract the detected loaded program and verify if the associated hash value is in the whitelist. CLAW's front-end component is able to access the address space of each monitored VM and make sense of the kernel data structures of monitored VMs using the real-time kernel data structure monitoring system. This architecture enables active monitoring of the protected VMs without requiring installation of any agents inside them.

The current CLAW implementation is built on the Intel VT hardware and the Xen hypervisor and is designed to support guest VMs running both Linux Ubuntu Jaunty and Windows XP.

4.4.2 Design and Implementation

4.4.2.1 Verification of Code in File-Backed Space

4.4.2.1.1 Creation of a New Process

CLAW continuously watches for newly created processes in each monitored VM so that it can verify the executable files being loaded before they are executed. To intercept process creation operations in Windows OS, CLAW keeps track of the Flink and Blink pointers in the structure pointed to by PsActiveProcessHead. If CLAW observes a write to either Blink or Flink on the page containing this structure (Step 2a of Figure 4.5), it traverse the processes lists to determine if a new EPROCESS structure has been created or if an existing process has been terminated. The process creation steps in Windows OS, as shown in Figure 4.6, up to the image mapping into the process' address space have already been done. As soon as the back-end component of CLAW detects a new process in a VM, it suspends the VM, and notifies CLAW's front-end component to take over. The front-end component uses the process's EPROCESS data structure to track down the new process's PEB data structure, and eventually the address space region mapping of the PE file used in the newly created process (Step 3 of Figure 4.5).

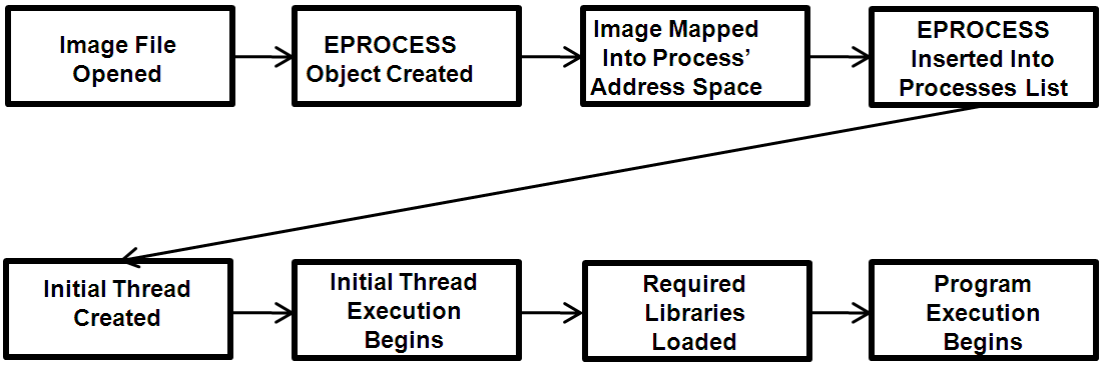

Figure 4.6 Windows OS process creation flow.

To intercept process creation operations in Linux OS, CLAW keeps track of the forward and backward pointers in the init_task structure. If CLAW observes a write to the forward or the backward pointer on the page containing the init_task structure, it traverse the processes list to determine if a new task_struct structure has been created or if an existing process has been terminated. As soon as the back-end component of CLAW detects a new process in a VM, it suspends the VM, and notifies CLAW's front-end component to take over. The front-end component uses the process's task_struct to track down the start and end addresses of the .text section of the ELF file used in the newly created process.

The front-end component verifies the legitimacy of the binary file by applying the SHA-1 hash function to the file's .text immutable code section and comparing the resulting hash value against all cryptographic hash values in the whitelist (Step 4 of Figure 4.5). If there is a match, CLAW allows the new process to run as usual by returning control to the monitored VM. If no match is found, the front-end component of CLAW zeros out the address space region holding the executable file and effectively prevents the process from continuing.

4.4.2.1.2 Loading of a Library Into an Existing Process

To detect new library modules loaded into an existing process in a Windows VM, the back-end component of CLAW monitors writes to the pages that contain the backward (Blink) pointer to the InLoadOrderLinks module list of all user processes in that VM. When CLAW's front-end component detects a write to the Blink (Step 2b of Figure 4.5) field on any of these pages, CLAW's back-end component analyzes the last LDR_DATA_TABLE_ENTRY member appended to the corresponding list to verify the newly loaded module (Step 3 of Figure 4.5). More concretely, using the DllBase field, CLAW locates the in-memory PE image of the library module, computes a SHA-1 hash value for the PE image's .text section, and checks the resulting

hash value against the whitelist (Step 4 of Figure 4.5). If no match is found, CLAW's front-end component zeros out the address space region for the library module and returns control to the VM. This design works for program loading operations for library modules that are either on-disk or in-memory, and therefore covers the type of code injection attacks that eventually use the native loader.

To detect new library modules loaded into an existing process in a Linux VM, the back-end component of CLAW intercepts the *mmap* system call. This call is used (1) in creating and associating a memory range with contents of a library component and (2) for creating memory ranges not tired to file descriptors such as those used in code injection attacks. We apply the system call interception mechanism described in the next section to verify both file-backed address space mappings and private address space regions.

4.4.2.2 Verification of Code in Private Space

In Windows OS, code in private address space regions is created by a *NtAllocateVirtualMemory* system call possibly followed by a *NtProtectVirtualMemory* system call. In Linux OS, code in private space regions is created by a *mmap* system call possibly followed by a *mprotect* system call. CLAW's back-end component supports a system call interception mechanism that captures these two system calls, and notifies the front-end component to analyze the captured system call's target address space. If the write and execute permissions of the target address space region are turned on, CLAW's front-end component sets the target address space region as non-executable, so that when the target address space region is first executed later on, a page fault occurs. At that instant, CLAW's front-end component computes a SHA-1 hash value of the target address space region and looks up the whitelist with

the resulting hash value. The above design works effectively against all code injection attacks described in the Background section. Unfortunately, it also tends to fail dynamically generated code, i.e., those produced by JIT compilers, interpreters and executable unpackers, because it is unlikely for the whitelist to include dynamically generated code. To address this false positive problem, CLAW offers the option to disable whitelisting checks for processes that run JIT compilers, interpreters and executable unpackers.

CLAW's system call interception mechanism works as follows:

1) System Call Table Extraction: In Windows OS, every executing thread stores a pointer to the SSDT at a known offset inside its ETHREAD data structure. CLAW locates the SSDT data structure in memory through the ETHREAD of the executing threads. In Linux OS kernel versions 2.6 and above, the System.map file holds the kernel address for the system_call_table array. The CLAW system uses this file to locate the sys_call_table data structure in memory.

2) System Call Capturing: CLAW turns off the present (P) bit on the memory pages pointed to by the system call table entries associated with the system calls that are to be intercepted, for example, *NtAllocateVirtualMemory* / *NtProtectVirtualMemory* and *mmap/mprotect* (Figure 4.7). Turning off the present bit of a page containing system call routines causes a page fault whenever the monitored VM invokes a systems call in that page and transfers control to the hypervisor (Step 2c of Figure 4.5). CLAW's back-end component then turns on the present bit of the page causing the page fault, and turns off the present bit of the page containing the return address of the invoked system call, and resumes the system call. When the invoked system call returns, another page fault occurs

because the present bit on the return address-containing page is off. At this point, CLAW's front-end component is notified to analyze the system call's input and output arguments (Step 3 of Figure 4.5).

After the analysis of the arguments is complete, CLAW's back-end component turns on the present bit of the page causing the page fault, and continues the system call's return to user mode.

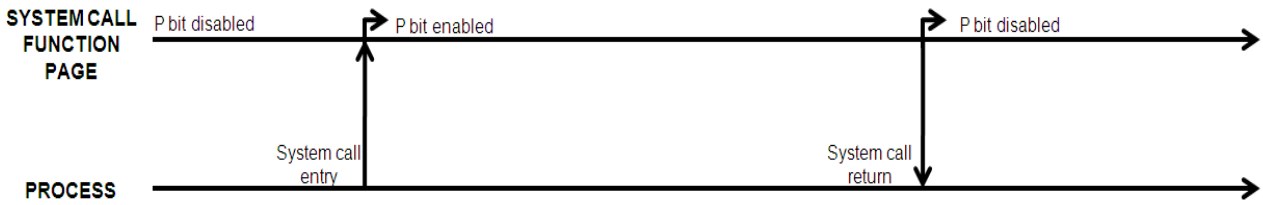

Figure 4.7 The CLAW system call interception steps – we enable/disable the present bit on system call entry/return.

3) Handling of Concurrent Identical System Calls: Modifying the permission of a kernel space page affects all user processes running on top of the kernel because the kernel address space is shared by all processes. Therefore, when the present bit of a page containing system call routines is turned on because one of the system calls in it is invoked, it is not possible to intercept other system calls in the same page. To solve this problem, we modify the SYSENTER_EIP_MSR register to point to an invalid page whenever there is at least one system call in execution (Figure 4.8). With this mechanism, system call interception works correctly even when some system calls are being executed, because every SYSENTER system call will trigger a page fault due to the setting of the SYSENTER_EIP_MSR register.

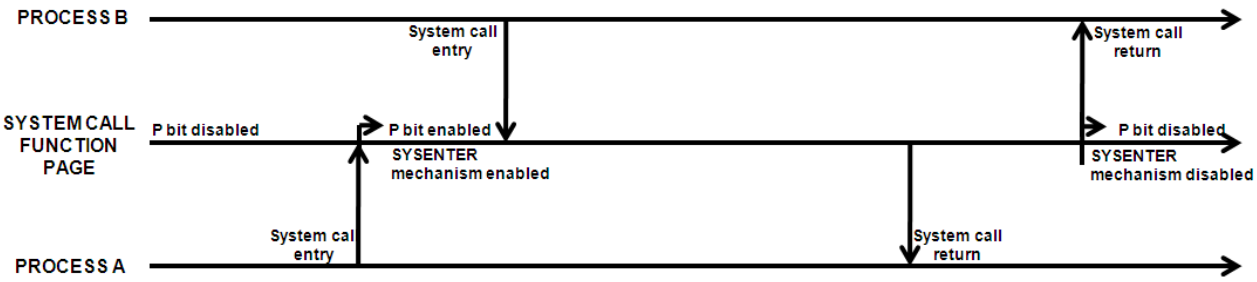

Figure 4.8 Combination of the CLAW and the MSR-register based system call interception.

In summary, CLAW features two system call interception mechanisms. When no system call is in execution, it uses a fine-grained interception mechanism that traps only for pages containing system calls that are to be intercepted. However, as soon as one or more system calls are invoked and being executed, it switches to a coarse-grained interception mechanism that stops all system calls. Note that as soon as the coarse-grained interception mechanism is enabled, the fine-grained one is disabled.

Because a page could contain multiple system call routines, when system calls that co-reside with a system call to be intercepted are called, they also trigger a page fault. When such page faults arise, CLAW simply ignores them and moves on.

4.4.3 Key Data Structures Monitored by CLAW

Table 4.6 provides the summary of the key data structures actively monitored by CLAW.

Table 4.6 Summary of the data structures monitored by CLAW.

OS	Data Structures (Fields)	Actions Taken
Windows	PsActiveProcessHead (Flink,Blink)	On write, traverse the processes lists to determine if a new EPROCESS structure has been created or if an existing process has been terminated.
	_PEB_LDR_DATA (InLoadOrderModuleList)	On write, analyze the last _LDR_DATA_TABLE_ENTRY member appended to the list to verify the newly loaded module.
Linux	init_task (next, prev)	On write, traverse the processes lists to determine if a new task_struct structure has been created or if an existing process has been terminated.

4.5 Evaluation

In this section, we describe the experiments conducted to evaluate the effectiveness and performance impact of the CLAW system.

4.5.1 Experimental Setup

The test machine consisted of a virtualized server that used Xen version 3.3 as the hypervisor and Ubuntu 9.04 (Linux kernel 2.6.26) as the Dom0 kernel. The host system used a Duo CPU P8600 processor containing two CPU cores at 2.4GHz and 2GB of system memory. The CLAW prototype was installed in the Dom0 domain. In addition, the virtualized server hosted a DomU domain running a default installation of Windows XP and configured with 512MB RAM.

4.5.2 Experiments

4.5.2.1 Effectiveness

After initializing the CLAW prototype, we ran the Internet Explorer application as the benign sample. CLAW was successful in identifying Internet Explorer as a trusted application, which was allowed to execute.

In the next test, we used the Metasploit Framework to exploit a buffer overflow in the Microsoft Server Service (MS08-067) [67, 68]. We then ran a payload introduced via the buffer overflow vulnerability that injected malicious code into the running Internet Explorer process via a remote thread injection attack. Execution of the injected code was prevented. Next, we configured the Metasploit framework to use a reflective library injection payload that allowed the library to load itself into the target address space without using the native loader (e.g., the library did not appear in the list of loaded modules in the PEB). When we executed the exploit, CLAW detected a call to allocate a private virtual memory in the process with the write/execute permissions and blocked execution of the injected code because it was not in the whitelist.

4.5.2.2 Performance

VM performance is impacted by the following CLAW monitoring components: (1) data structure monitoring (the PsActiveProcessHead + _PEB_LDR_DATA monitoring); (2) system call interception (present bit-based system call interception + MSR-based system call interception).

To measure the run-time CLAW overhead, we selected the PCMark industry standard benchmarking application [30] to run several benchmarks for the data structure monitoring and

the system call interception components. The results of testing appear in Table 4.7. Each of the benchmarks was first run without CLAW to obtain the baseline performance and then re-run with (1) the PsActiveProcessHead monitoring enabled and the _PEB_LDR_DATA and CLAW system call interception disabled; (2) the PsActiveProcessHead and _PEB_LDR_DATA monitoring enabled and the CLAW system call interception disabled; (3) the PsActiveProcessHead and _PEB_LDR_DATA monitoring disabled and the CLAW system call interception enabled; (4) all system call interception using the MSR-based system call interception based approach (continuous interception of all system calls) enabled.

Table 4.7 Run-time performance of CLAW.

Benchmark	PsActiveProcessHead Monitoring	PsActiveProcessHead + _PEB_LDR_DATA Monitoring	CLAW's System Call Interception	MSR System Call Interception of all system calls
CPU	2.4%	2.6%	0.8%	7.9%
Memory	1.3%	1.3%	4.6%	64.3%
HDD	3.7%	3.8%	1.1%	29.5%

Among the three schemes used in CLAW to detect program loading, the CLAW system call interception incurs the least overhead because it is targeted at specific pages containing system call routines of interest. The MSR-based system call interception incurs the most overhead. The overhead incurred by monitoring of the PsActiveProcessHead structure is somewhat higher than expected, because there are modifications to the same page holding the process list that trigger spurious write protection faults. The overhead incurred by

_PEB_LDR_DATA monitoring is negligible as components of this structure are located in user address space pages and are rarely modified.

Because it is difficult to measure the run-time performance degradation of interaction with applications, we focus on their startup time instead. We measured the startup time of three interactive applications: MS Office Word, Mozilla Firefox, and Adobe Acrobat. We ran these applications and used the PassMark AppTimer [69] tool to measure the time between when an application was started and when its main window for use input appeared, with and without the CLAW. These measurements also included the code verification times while the VM was suspended, and appear in Table 4.8. Even though the percentage overheads are more substantial than batched programs, the start-up overhead time of all three interactive applications are less than one second, which are reasonable and acceptable user experiences.

Table 4.8 Startup performance of CLAW.

Applications	Total Startup Time, msec	Overhead
MS Office Word	1,764	37%
Mozilla Firefox	366	43%
Adobe Acrobat	1,487	17%

To evaluate the performance advantage of the PsActiveProcessHead and _PEB_LDR_DATA data structure monitoring over interception of system calls involved in new process creation and code mapping using libraries, we extended the CLAW system call interception mechanism to include monitoring of the NtCreateSection and NtMapViewOfSection system calls. NtCreateSection is always invoked when a new process is started to create a section object. NtMapViewOfSection is used to map views of section objects created using

NtCreateSection into a process address space. Invocation of NtMapViewOfSection with the request of mapping a view of a section combined with the page protection flag argument set to allow executions indicates that the process is mapping a new executable region. To extract the code region used in mapping a view, we look up the NtMapViewOfSection's section handle argument among the handles owned by the requesting process to locate the address of the corresponding section object. The list of handles owned by the process can be found using the corresponding process EPROCESS structure. We use the identifies section object to traverse the related memory structures in the kernel memory of the VM to extract the subsection corresponding to the .text section of the file and verify the identity of the region using SHA-1 hashing. The performance benchmark results of the NtCreateSection and NtMapViewOfSection system call interception are provided in Table 4.9. Startup performance results are provided in Table 4.10.

Table 4.9 Run-time performance of NtCreateSection and NtMapViewOfSection system call interception.

Benchmark	PsActiveProcessHead + _PEB_LDR_DATA Monitoring	CLAW's System Call Interception of NtCreateSection & NtMapViewOfSection
CPU	2.6%	1.1%
Memory Latency	1.3%	3.2%
HDD	3.8%	1.3%

Table 4.10 Startup performance of CLAW using NtCreateSection and NtMapViewOfSection system call interception.

Applications	Total Startup Time, msec	Overhead
MS Office Word	1,996	52%
Mozilla Firefox	538	110%
Adobe Acrobat	1,623	28%

Interception of the NtCreateSection and NtMapViewOfSection system calls has a minor run-time performance advantage while the start-up overhead time using these system calls has significantly increased. The increase in the start-up time is due to parsing of the system call arguments that requires traversing and parsing of series of data structures in the kernel address space. Although our experiments show that direct interception of the NtCreateSection and NtMapViewOfSection system calls has a better run-time performance, the PsActiveProcessHead and _PEB_LDR_DATA data structure monitoring may be beneficial in addressing scenarios where the NtCreateSection and NtMapViewOfSection system calls are hooked by malicious user-space code to bypass invocations of the actual NtCreateSection and NtMapViewOfSection.

4.6 Limitations

Code verifications performed by the CLAW at load-time include the binary code of the executable file and libraries in its initially loaded state. However, a process may be exploited over the course of its execution through an application vulnerability, such as a buffer overflow, and new unverified code may be introduced by manipulating the existing code and thus, bypassing the CLAW load-time defense mechanisms. Our current system does not specifically defend against these attacks.

The CLAW does not detect malicious activity that does not introduce any unapproved code into the system but rather uses the approved code. For instance, a malicious process could attempt to tamper with the non-control data of a process that may be used by the application while carrying out its computations and interactions and indirectly modify its operations without injecting additional code. Examples of such attacks and a proposed defense mechanism are described in Chapter 5.

Finally, false positives may also arise from processing code dynamically generated by JIT compilers, interpreters, and packed executables. In our future work, we will investigate how these special cases can be addressed by CLAW.

4.7 Summary

We presented the CLAW, a system that verified the code identity in the VM execution environment. The CLAW verified binary code in user processes by computing a cryptographic hash over the executable file and its dependencies (library components) at their load-time mapping. These verifications were taken when a process and libraries were loaded in memory but before their first execution. The CLAW also tracked and analyzed code in executable memory regions allocated at run-time. We developed a prototype of our approach for the Windows and Linux operating systems. The results showed that the system was able to reliably identify whitelisted codes in applications while blocking unapproved codes. Successful identification of the malicious code introduced through code injection attacks further demonstrated CLAW's effectiveness in dealing with sophisticated attacks designed to hide the code's presence. The concepts and techniques discussed in this study could be applied to other operating systems and hypervisors.

5 Access Token Manipulation Attack Detection Tool

5.1 Introduction

Many real-world software applications are susceptible to attacks that alter the target program's control data (e.g., return addresses and function pointers) in order to execute injected malicious code. Because control-data attacks have been pre-dominant, many defensive techniques have been developed to protect program control flow integrity to prevent such attacks. With the advancement of control flow protection techniques, attackers have devised a new group of attacks to bypass the defenses. These attacks target non-control data and are less straightforward to construct than control data attacks because they require in-depth semantic knowledge of the target data. The current range of defensive techniques against non-control data attacks is limited. This is because data structures frequently targeted by non-control attacks change rapidly making it difficult to differentiate between normal and abnormal states.

The stealthiest of non-control data attacks is the direct kernel object manipulation (DKOM) attack, which directly accesses and writes to kernel data structures stored in memory without using any APIs. A unique example using the DKOM technique is the hidden process attack in which the attacker manipulates the doubly linked list of running processes to unlink a malicious process and hide it from the OS view [70]. Other examples of hidden object attacks include driver and network data hiding to create false views of loaded drivers and network usage.

In this study, we focus on DKOM-based access token manipulation attacks that target authorization and authentication data assigned to a running process. The access token manipulation is a post-exploitation technique allowing the attacker to escalate privileges on an

already compromised Windows host. The access token data structure determines the access privileges associated with a running Windows process and is derived from the user's log-on session. When a process attempts to perform various actions, the privileges in the access token are compared to the required privileges to determine if access should be granted or denied. Privilege escalation is achieved either by altering (token patching) or copying (token stealing) the access token of a target process.

In the token patching attack, the attacker alters the access token of a target process to raise the process's privileges to the maximum level on the local system. Rootkits are known to make use of the token patching attack by directly overwriting portions of the kernel memory storing the process's access token with new privileges. In the token stealing attack, the attacker copies an already existing token of a user who has previously logged into the compromised machine and swaps the target process's access token with the copied token to assume the user's privileges. Because tokens of logged-in users may have domain-wide privileges, the token stealing attack magnifies the dangers of the token patching attack by allowing the attacker to compromise additional machines on the network domain.

We describe the design and implementation of a novel defensive tool called ATOM that watches access tokens of running processes to detect access token manipulation attacks. ATOM has an agentless architecture built on top of the RTKDSM system. We have successfully implemented an ATOM prototype on the Xen hypervisor and targeted it at Windows and Linux VMs.

5.2 Background

5.2.1 Access Token Data Structure

Every Windows process has an associated EPROCESS data structure (Figure 5.1). The EPROCESS keeps track of various process-specific data including a pointer to its own access token in the Token member of the _EX_FAST_REF type (Figure 5.2). The pointer points to the TOKEN data structure (Figure 5.3). The exact memory address of the TOKEN structure is calculated from the Token member by XORing the Token value with 0xFFFFFFF8. The XOR operation is required because the last 3 bits of the Token value are used to keep a reference count for optimization purposes. Thus, token addresses always end with the last three bits equal to zero.

```
typedef struct _EPROCESS
{
     KPROCESS Pcb;
     EX_PUSH_LOCK ProcessLock;
     LARGE_INTEGER CreateTime;
     LARGE_INTEGER ExitTime;
     EX_RUNDOWN_REF RundownProtect;
     PVOID UniqueProcessId;
     LIST_ENTRY ActiveProcessLinks;
     ULONG QuotaUsage[3];
     ULONG QuotaPeak[3];
     ULONG CommitCharge;
     ULONG PeakVirtualSize;
     ULONG VirtualSize;
     LIST_ENTRY SessionProcessLinks;
     PVOID DebugPort;
     union
     {
          PVOID ExceptionPortData;
          ULONG ExceptionPortValue;
          ULONG ExceptionPortState: 3;
     };
     PHANDLE_TABLE ObjectTable;
     EX_FAST_REF Token;

     [...]

} EPROCESS, *PEPROCESS;
```

Figure 5.1 _EPROCESS data structure.

.

```
typedef struct _EX_FAST_REF
{
     union
     {
          PVOID Object;
          ULONG RefCnt: 3;
          ULONG Value;
     };
} EX_FAST_REF, *PEX_FAST_REF;
```

Figure 5.2 EX_FAST_REF data structure.

```
typedef struct _TOKEN
{
     TOKEN_SOURCE TokenSource;
     LUID TokenId;
     LUID AuthenticationId;
     LUID ParentTokenId;
     LARGE_INTEGER ExpirationTime;
     PERESOURCE TokenLock;
     LUID ModifiedId;
     ULONG SessionId;
     ULONG UserAndGroupCount;
     ULONG RestrictedSidCount;
     ULONG PrivilegeCount;
     ULONG VariableLength;
     ULONG DynamicCharged;
     ULONG DynamicAvailable;
     ULONG DefaultOwnerIndex;
     PSID_AND_ATTRIBUTES UserAndGroups;
     PSID_AND_ATTRIBUTES RestrictedSids;
     PVOID PrimaryGroup;
     PUID_AND_ATTRIBUTES Privileges;
     ULONG * DynamicPart;
     PACL DefaultDacl;
     TOKEN_TYPE TokenType;
     SECURITY_IMPERSONATION_LEVEL ImpersonationLevel;
     ULONG TokenFlags;
     UCHAR TokenInUse;
     PSECURITY_TOKEN_PROXY_DATA ProxyData;
     PSECURITY_TOKEN_AUDIT_DATA AuditData;
     SEP_AUDIT_POLICY AuditPolicy;
     ULONG VariablePart;
} TOKEN, *PTOKEN;
```

Figure 5.3 TOKEN data structure in Windows XP.

The TOKEN data structure is composed of static and dynamic parts (Figure 5.4). The static part has a well-defined structure and does not change in size. It stores the count of privileges in the PrivilegeCount field and the count of the security identifiers (SIDs) in the UserAndGroupCount field. The dynamic part contains all the user privileges and SIDs. The exact number of these varies depending on the credentials of the user who created the process.

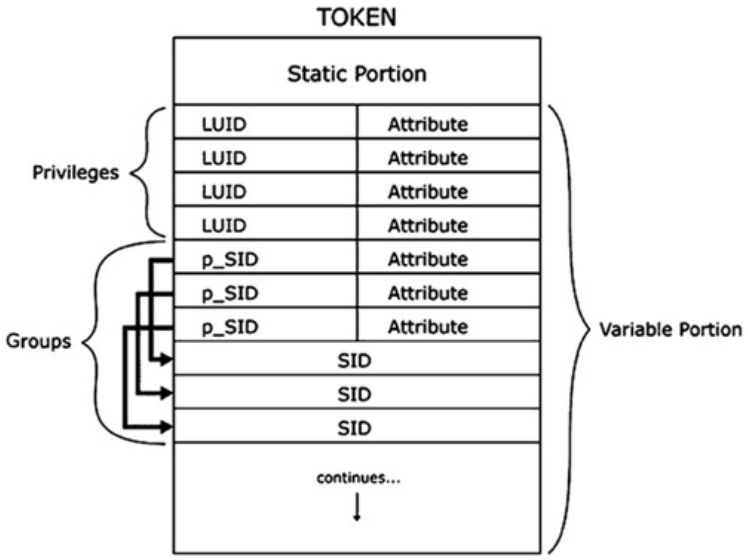

Figure 5.4 Static and variable parts of the token in Windows XP [70].

The UserandGroups field stores a pointer to a dynamically allocated array of PSID_AND_ATTRIBUTES structures storing security identifiers (SIDs) including a SID for the user and all of the SIDs for the groups to which the user belongs (Figure 5.5).

```
typedef struct _SID_AND_ATTRIBUTES {
  PSID  Sid;
  ULONG Attributes;
} SID_AND_ATTRIBUTES, *PSID_AND_ATTRIBUTES;

typedef struct _SID {
  UCHAR  Revision;
  UCHAR  SubAuthorityCount;
  SID_IDENTIFIER_AUTHORITY  IdentifierAuthority;
  ULONG  SubAuthority[ANYSIZE_ARRAY];
} SID, *PISID;
```

Figure 5.5 SID_AND_ATTRIBUTES and SID data structures.

Each PSID_AND_ATTRIBUTES structure is composed of two fields: Sid, which is a pointer to the SID structure holding SID information, and Attributes, which stores a series of binary flags that hold the SID attributes. When a SID is added to the token, the UserAndGroupCount value is incremented. The Security Descriptor Definition Language form of a SID can be illustrated using the following example: “S-1-5-21-2833009033-2652595096-1975694352-1012”, where “1” is the revision, “5” is the identifier authority that created the SID, “21-2833009033-2652595096-1975694352” is the computer identifier, and “1012” is the account or group identifier.

The Privileges field of the TOKEN data structure stores a pointer to a dynamically allocated array of LUID_AND_ATTRIBUTES structures (Figure 5.6). Each LUID_AND_ATTRIBUTES structure is composed of two fields: Luid storing the privilege ID and Attributes storing a series of binary flags that define whether a privilege associated with a given LUID is enabled or disabled. In the Privileges list, some of the privileges are disabled by default (Figure 5.7).

```
typedef struct _LUID_AND_ATTRIBUTES {
  LUID  Luid;
  DWORD Attributes;
} LUID_AND_ATTRIBUTES, *PLUID_AND_ATTRIBUTES;
```

Figure 5.6 LUID_AND_ATTRIBUTES data structure.

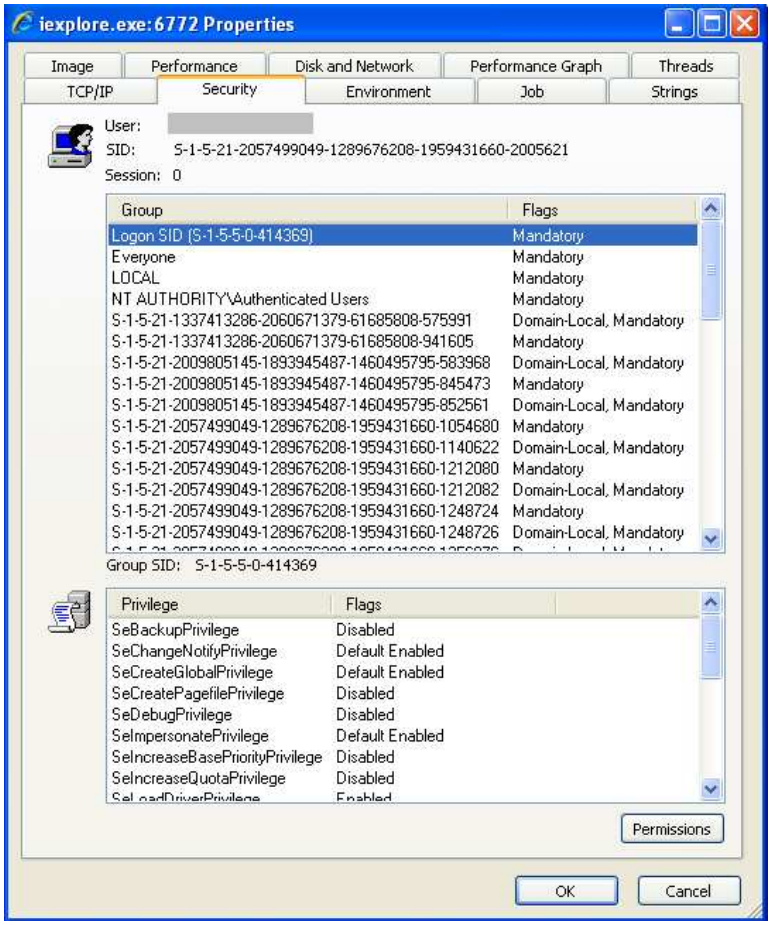

Figure 5.7 SIDs and Privileges contained in the process's access token using Sysinternals' Process Explorer tool [71].

5.2.2 Token Manipulation Attacks

This section describes the post-exploitation process using token manipulation attacks.

5.2.2.1 Access Token Patching

Access token patching attacks are commonly launched by kernel level rootkits. There are two main rootkit families: control-data manipulating rootkits and non-control data manipulating rootkits.

Control data manipulating rootkits, known as hooking rootkits, change the kernel control flow path in such a way that control first flows to the attack code. The original code is either never invoked or executed after the attack code is executed. Hooking may come in several variations including import/export table hooking, system service dispatch table hooking, interrupt descriptor table hooking, and inline function hooking. These methods allow an attacker to gain control of the execution path by patching function pointers in a table through which a set of calls or events are routed or by modifying the binary code of a target function.

Non-control data manipulating rootkits do not change the control flow directly but manipulate values of critical variables, which in turn directly or indirectly influence the algorithms used by the kernel. Such rootkits often target kernel data with dynamic characteristics without injecting any code into the kernel memory space. These rootkits use DKOM techniques to dynamically change certain kernel data structure, such as the access token data structure.

Non-control data manipulating rootkits launch access token manipulation attacks to raise privileges of a malicious process without making a single call to any of the process or token related APIs. This can be accomplished by modifying data contained in the TOKEN data

structure directly in memory. When modifying the TOKEN data structure, rootkits patch the SID list and the Privileges values. The FU rootkit is one example of an access token patching rootkit [70]. The FU rootkit operates using the following steps:

1) Finds the EPROCESS data structure for the target process using the process PID;

```
DWORD FindProcessEPROC (int terminate_PID)
{
        DWORD eproc       = 0x00000000;
        int   current_PID = 0;
        int   start_PID   = 0;
        int   i_count     = 0;
        PLIST_ENTRY plist_active_procs;
        if (terminate_PID == 0)
                return terminate_PID;
        eproc = (DWORD) PsGetCurrentProcess();
        start_PID = *((DWORD*)(eproc+PIDOFFSET));
        current_PID = start_PID;
        while(1)
        {
                if(terminate_PID == current_PID)
                        return eproc;
                else if((i_count >= 1) && (start_PID == current_PID))
                        return 0x00000000;
                else {
                        plist_active_procs = (LIST_ENTRY *) (eproc+FLINKOFFSET);
                        eproc = (DWORD) plist_active_procs->Flink;
                        eproc = eproc - FLINKOFFSET;
                        current_PID = *((int *)(eproc+PIDOFFSET));
                        i_count++;
                }
        }
}
```

2) Finds the TOKEN data structure associated with the EPROCESS data structure;

```
DWORD FindProcessToken (DWORD eproc)
{
        DWORD token;
        __asm {
                mov eax, eproc;
                add eax, TOKENOFFSET;
                mov eax, [eax];
                and eax, 0xfffffff8;
                mov token, eax;
        }
        return token;
}
```

3) Finds the privileges in the token and adds new privileges;

The fact that many privileges are disabled by default when a token is created proves to be useful for an attacker in order to add privileges and groups to a process token. If a desired

privilege already exists in the token but is disabled, the rootkit enables the privilege. If a desired privilege does not exist in the token, the rootkit finds a disabled privilege and re-uses its space by overwriting it with the new privilege. By enabling or overwriting disabled privileges already contained in the token, the attacker can avoid increasing the token's size and overwriting memory regions adjacent to the process's token some of which may be invalid.

4) Finds the SIDs in the token and adds new SIDs;

   Disabled privileges may also be overwritten to make room for new SIDs.

5) Finally, modifies the PrivilegeCount and UserAndGroupCount counts.

In Windows versions prior to Windows Vista, there were no integrity checks on the UserAndGroup list of SIDs and therefore, it was possible to add SIDs by finding dead space in the token structure to overwrite it with. In the recent versions of Windows starting with Vista, new fields SidHash and RestrictedSidHash have been added in the access token structure (Figure 5.8 and Figure 5.9). These two fields contain the hashes of the SIDs stored in the dynamic part of the token in order to prevent accidental or intended modification of this part of the access token. The hashes are checked every time the token is used. Despite the added integrity checks, access token manipulation attacks are still possible with three main alternatives to bypass these defense measures:

1) Applying the hash algorithm after modifying the SID lists;

2) Avoiding SID list patching and acting only on the Privileges;

3) Directly swapping the TOKEN value of the attacker’s process with the value in the

EPROCESS structure of a victim process using the token stealing attack as described in the next section.

```
typedef struct _TOKEN
{
     TOKEN_SOURCE TokenSource;
     LUID TokenId;
     LUID AuthenticationId;
     LUID ParentTokenId;
     LARGE_INTEGER ExpirationTime;
     PERESOURCE TokenLock;
     LUID ModifiedId;
     SEP_TOKEN_PRIVILEGES Privileges;
     SEP_AUDIT_POLICY AuditPolicy;
     ULONG SessionId;
     ULONG UserAndGroupCount;
     ULONG RestrictedSidCount;
     ULONG VariableLength;
     ULONG DynamicCharged;
     ULONG DynamicAvailable;
     ULONG DefaultOwnerIndex;
     PSID_AND_ATTRIBUTES UserAndGroups;
     PSID_AND_ATTRIBUTES RestrictedSids;
     PVOID PrimaryGroup;
     ULONG * DynamicPart;
     PACL DefaultDacl;
     TOKEN_TYPE TokenType;
     SECURITY_IMPERSONATION_LEVEL ImpersonationLevel;
     ULONG TokenFlags;
     UCHAR TokenInUse;
     ULONG IntegrityLevelIndex;
     ULONG MandatoryPolicy;
     PSECURITY_TOKEN_PROXY_DATA ProxyData;
     PSECURITY_TOKEN_AUDIT_DATA AuditData;
     PSEP_LOGON_SESSION_REFERENCES LogonSession;
     LUID OriginatingLogonSession;
     SID_AND_ATTRIBUTES_HASH SidHash;
     SID_AND_ATTRIBUTES_HASH RestrictedSidHash;
     ULONG VariablePart;
} TOKEN, *PTOKEN;
```

Figure 5.8 TOKEN data structure in Windows Vista.

```
typedef struct _SID_AND_ATTRIBUTES_HASH
{
     ULONG SidCount;
     PSID_AND_ATTRIBUTES SidAttr;
     ULONG Hash[32];
} SID_AND_ATTRIBUTES_HASH, *PSID_AND_ATTRIBUTES_HASH;
```

Figure 5.9 _SID_AND_ATTRIBUTES_HASH data structure.

5.2.2.2 Access Token Stealing

During normal operations of a system, there are tokens of some variety present depending on the system's function and its usage environment. If the system is compromised, these tokens can be used by the attacker in token stealing attacks to achieve privilege escalation. The token stealing attack involves the exchange of a malicious process's token with an access token of another process running on the same system.

There are two main types of access tokens useful for this attack: primary tokens and impersonation tokens. Every process has a primary token that describes the security context of the user account associated with the process. Impersonation is the ability of a process to temporarily impersonate a security context different from the context of the process by starting a thread using a different access token. The main reason for impersonation is to enable a service running under a certain security context act on behalf of connecting clients by executing threads under the clients' own security context. There are four impersonation levels: Anonymous, Identification, Impersonation, and Delegation, of which the Impersonation level and the Delegation level have the most significant security implications (Figure 5.10).

```
typedef enum _SECURITY_IMPERSONATION_LEVEL
{
        SecurityAnonymous = 0,
        SecurityIdentification = 1,
        SecurityImpersonation = 2,
        SecurityDelegation = 3
} SECURITY_IMPERSONATION_LEVEL;
```

Figure 5.10 Impersonation levels.

The Impersonation level tokens, which are normally created as a result of a non-

interactive login, allow a thread to impersonate the security context on the local system but do not allow access to external systems. A common example would be an FTP server impersonating client requests. The Delegation level tokens, which are normally created as a result of an interactive login, allow a thread to impersonate the security context on any system. Examples include logging in using remote access services and solutions.

By hijacking Delegation level tokens, an attacker can gain domain level privileges to access systems that are otherwise secure from direct remote exploits. This is possible because Delegation tokens contain authentication credentials and so can be used to access external systems for which those credentials are valid.

A token stealing attack normally involves the following steps:

1) Enumeration of tokens present on the compromised system;

2) Selection of a Delegation or Impersonation token;

3) Starting a new process and swapping the process's token with the token selected in the previous step.

The swapping in the last step can be accomplished by calling an existing API, such as ImpersonateLoggedOnUser in Windows OS. In this study, however, we only consider DKOM-based token stealing attacks that directly overwrite the value of the Token member of the EPROCESS structure in memory to point to a different access token [72].

5.3 Related Work

5.3.1 Control Data Manipulating Rootkits

Methods for detection of control data manipulating rootkits can be roughly divided into static control data monitoring methods and execution path monitoring methods. Static control data monitoring methods detect signs of a rootkit intrusion by checking known invariant data regions in memory for suspicious entries. Violations of such invariants suggest the kernel has been compromised. Execution path monitoring methods identify known program execution paths in advance and monitor run-time execution paths to ensure they conform to the known paths. Deviations from know execution paths are suggestive of rootkit presence.

5.3.1.1 Static Control Data Monitoring

5.3.1.1.1 Periodic Checks

Co-processor based Copilot was designed to detect kernel rootkits overwriting the addresses of the kernel's system call handling functions in the system call table with the addresses of their own doctored system call handling functions as well as modifying the host kernel's text or the text or any loaded LKMs [16]. Copilot extracted the memory addresses of the system call table and the kernel text from the host kernel and its System.map file at configuration time, calculated known "good hashes" for these items, and monitored the related memory regions throughout the host kernel run-time using periodic checks to detect changes to these kernel memory regions. The fundamental limitation of Copilot was its inherent inability to detect modifications as they occurred. A clever rootkit might conceivably modify and rapidly repair the host kernel between checks as a means of avoiding detection.

5.3.1.1.2 Continuous Monitoring

The hypervisor-based intrusion detection system Livewire employed similar methods to detect signs of malicious rootkit activity [4]. To detect modifications to sensitive portions of the kernel memory continuously in real-time, Livewire marked the code sections and system call table derived from the debugging information of the kernel binary as read-only. If a program tried to modify these sections of memory, the monitor was notified about the malicious attempt, and the VM was halted.

In another related study, Paladin leveraged the virtual machine technology to propose a solution for real-time detection and containment of rootkit attacks. Paladin relied on specification of access control policies tailored to protect memory areas and system files that could be a target of rootkit attacks [73]. The memory access control policies included policies to protect the kernel system call table, the interrupt table, and the kernel code from being overwritten in memory by defining legitimate applications that could write into kernel memory. To obtain the knowledge about the guest OS semantics, Paladin ran a driver inside the host OS to facilitate symbol lookups in the System.map file for kernel text segment, system call table, and interrupt descriptor table. Given the specifications of the access control policies and the physical addresses of the protected memory regions, Paladin used the hypervisor to monitor write accesses across the system for validity. Any time an illegal access was detected, the process attempting modifications was killed.

Static control data monitoring systems make themselves vulnerable to rootkits that take this type of discovery method into account and evade the security monitors, for instance, by manipulating the system call table dispatch handler and redirecting the system call to a completely fabricated table filled with pointers to malicious system call handlers. A static control

data monitor would continue to monitor the original, unchanged system call table, which would no longer used by the kernel. Furthermore, monitoring for writes in known static data locations would not prevent rootkits from hijacking function hooks within data structures that were meant to be overwritten.

5.3.1.2 Execution Path Monitoring

5.3.1.2.1 Periodic Checks

State-based control flow integrity (SBCFI) performed a static analysis of the kernel's source code and compiled binary for global variables and function pointers reachable from the global variables and built an approximation of kernel control-flow graph that would be followed at run-time by a legitimate kernel [56]. Function pointers were tracked and validated periodically at run-time to determine consistency with the control-flow graph using a monitor placed in a separate security VM. The monitor process traversed the target kernel's memory in parallel with the target VM's execution. Because the monitoring was done periodically, the SBCFI monitor could only be used to reliably discover persistent changes: if an attacker modified the kernel for a short period, but undid the modifications in time less then the next check period, then monitor might fail to discover the change. Additionally, due to the lack of dynamic run-time information, SBCFI was only able to achieve an approximation of kernel control-flow graph. The static nature of the SBCFI system and learning inextensibility (due to the rapidly changing nature of the Linux kernel) were some shortcomings of this approach. The performance of SBCFI was also shown to incur close to 40% overhead on a typical machine running on Xen.

5.3.1.2.2 Continuous Monitoring

HookSafe, a hypervisor-based system, was designed to detect control flow modifying rootkits [74]. On initialization, HookSafe used an in-guest kernel module to allocate memory

pages from the non-paged pool and copy protected kernel hooks from their original locations to the newly allocated memory pages. It then loaded the indirection layer code in the guest OS to regulate accesses to these memory pages. The hypervisor was notified through a hypercall about the allocated memory pages to detour all accesses to protected hooks to the hook indirection layer. For read accesses, the indirection layer simply read from the shadow hooks and returned to the hook site. For write accesses, the indirection layer issued a hypercall and transferred the control to the hypervisor to validate the write request according to values seen in the offline normal operation profiling phase. By re-locating hooks to dedicated memory pages, HookSafe avoided the unnecessary page faults caused by trapping writes to irrelevant data that might be co-located with hooks on the same page. Hooks allocated at run-time were identified by instrumenting the guest OS memory allocation functions and utilizing the run-time context information to infer whether a particular kernel object of interest containing an embedded hook was being allocated. If one such kernel object containing a kernel hook was being allocated, a hypercall was issued to HookSafe to create a shadow copy of the hook. The HookSafe implementation required modifications to the monitored OS and therefore could not be extended to support closed source OSes.

5.3.1.2.3 Offline Analysis

From another perspective, HookFinder [75] based on a whole system emulator was developed to automatically analyze an unknown potentially malicious binary and identify if this code installed any hooks into the system. HookFinder was designed for malware analysis rather than on-line detection. By instrumenting CPU instructions with taint propagation capabilities, HookFinder considered any changes made by the malware as tainted and tracked taint propagation throughout the system. HookFinder recognized a specific change as a hooking point

if the control flow was affected by some tainted value. Though effective in identifying specific hooks registered in the malware code, HookFinder could not discover other hooks that did not lie in the execution paths of any of these programs, and therefore would go undetected.

The HookMap implementation collected a list of sequentially executing kernel instructions when handling a system call and identified the control-flow transfer instructions that could potentially be exploited by rootkit for hiding purposes [76].

5.3.2 Non-Control Dynamic Data Manipulating Rootkits

The control data manipulating rootkit detection methods that detect violations based on changes to static kernel content, control flow, or the executing binaries can not be applied to detection of rootkit attacks on non-control data structures as they often include data and functions pointers that are meant to be overwritten. Additionally, attacks against such data may be performed by using already approved kernel code which satisfies kernel code integrity. Therefore, a number of specialized methods have been developed to combat such attacks.

5.3.2.1 Periodic Checks

Petroni et al. [77] extended the capability of Copilot [16] for detection of attacks against dynamically allocated constantly-changing kernel objects using a co-processor. The monitor relied on an expert to describe the correct operation of the system via specifications of security-relevant data structures and constraints on how these data structures interoperated. The monitor periodically compared actual observed dynamic kernel data values in the snapshots of kernel memory with the specifications of constraints on kernel dynamic data values and reported any semantic integrity violations.

The cross view detection method in a hypervisor-based VMwatcher implementation leveraged the self-hiding nature of rootkits to infer rootkit presence by detecting discrepancies between process lists from different points of detection [3]. VMwatcher approach used an introspection-based method to obtain a view of the processes running in the system and invoked a standard API function from within the OS to get the API view of the processes running in the system. The two results were compared, and the difference in the results revealed hidden rootkit processes.

The above systems use a periodic sampling approach that may be exploited by the malware to remain undetected in between two consecutive snapshot periods making this approach far less attractive due to its lack of immediacy. Conversely, ATOM is able to extract and analyze the data structures continuously, overcoming the limitations of the periodic checks approach.

5.3.2.2 Continuous Monitoring

Srivastava et al. [18] developed Sentry, a VM-based system that prevented illegitimate changes to dynamically allocated kernel data objects from occurring by mediating access to these objects. Sentry introduced modifications to the monitored OS kernel to identify locations of newly constructed dynamically-allocated kernel data object. The need for mediated access to a newly constructed data object was communicated by the kernel to the hypervisor at the time that it constructed the object. Similar to page protections manipulation approach used in the RTKDSM system, the OS passed the physical page frame number (PFN) of the newly allocated memory page holding kernel data object requiring protection to the hypervisor. When the memory protection module in the hypervisor received a request to add protection for the

monitored VM's page, it added the PFN to a list of protected pages and removed the page's write permission causing page faults on all attempted kernel object alterations. Sentry only allowed alterations invoked by legitimate kernel functionality. Sentry implementation also made alterations to the memory layout of kernel data structures to separate security critical and non-critical fields for increased performance and therefore required access the OS source code.

Rhee et al. [17] proposed the KG system that prevented rootkit attacks targeting dynamic data by detecting changes to monitored kernel data structures. KG monitored the execution of the OS at the instruction level using QEMU emulator as an external monitor. For each kernel data structure requiring protection, a policy was written describing how the data structure should be identified in a raw view of memory as well as the characteristics of an attack against that data structure. The policies were derived using the kernel source code and the analysis of functions used to access given kernel data structures. At runtime, the system identified data structures of interest in memory and intercepted all writes to their address ranges. The methodology described in the study was only portable to VM monitors that supported memory interposition to translate guest instructions into host instructions and therefore, it could not be extended to support commercial hypervisors that did not provide memory interposition, such as Xen and VMWare ESX, unlike in ATOM developed in this study.

A hypervisor-based VMhuko was designed to provide real-time protection for static and dynamic kernel data by mediating access to these data using access control policies [78]. VMhuko relied on the static analysis of the OS source code to extract information about data structures as well as their related normal kernel object access patterns and to build access control policies for the extracted kernel objects. Locations of all static kernel objects were identified at run-time using the kernel debug symbols and system map information in Linux. Dynamic kernel

objects were located at run-time using the assumption that all dynamic data were accessible from global kernel data structures residing at well-known locations. For instance, the init_task global data structure was assumed to be first accessed by rootkits to locate and traverse the task linked list and then manipulate the dynamically allocated task_struct structures. Therefore, init_task was monitored for read accesses. Memory pages containing static objects and pointer-valued fields of global kernel data structures were marked as protected using not-writeable or not-present fields and monitored for abnormal read and write kernel access patterns by comparing function call traces to known good ones obtained from the static analysis of the sources code. All rootkits used in the evaluation were system call table modifying rootkits running as self-hiding processes. Although the average VMhuko performance overhead was reported as 17%, no details were provided regarding the number of static and dynamic objects monitored in their experiments. The VMhuko protection would be difficult to design for a closed source OS such as Windows where the source code could be unavailable. Furthermore, a disadvantage of this kind of implementation was that the source code analysis needed to be manually performed every time a module was inserted into the kernel to ensure that valid accesses by the module were not invalidate by VMhuko. Additionally, due to the lack of dynamic run-time information, VMhuko was only able to achieve an approximation of normal kernel access activity.

5.3.3 Summary of Methods

Static control data monitoring methods including Copilot, Paladin, and Livewire, are intended for protection of a small number of invariant data structures positioned at fixed locations and known at compile time. These methods are not suited well for advanced attacks targeting dynamic non-control kernel data where locations of data and the number of instances are not known in advance. While the Copilot architecture was later extended to support detection

of such attacks [77], the new extension was based on an asynchronous approach and suffered from inherent inability to detect attacks launched and withdrawn between two subsequent periodic snapshots as a means of avoiding detection. Paladin served as a good detection and prevention mechanism but the specifications of memory access control policies for protected memory regions were static and a comprehensive survey of them was infeasible especially when dealing with dynamically allocated objects. The Sentry, KG, and VMhuko architectures might be considered extensions of the Paladin approach for protection of dynamically allocated objects. However, these extensions either required OS modifications to detect object allocations or the availability of the kernel source code to construct access control policies that might be difficult to obtain for a closed source OS such as Windows. In our implementation, we extended the static control data monitoring approach to protecting against attacks on dynamic non-control data.

Execution path monitoring methods are a subset of the general concept of protecting invariant data known at compile time or enumerable data derivable from the invariant data. Although these methods may be extended to support dynamically allocated control data, such extensions may not be applicable to guard data structures unreachable from the global variables and lacking semantic relationships with others [65].

Table 5.1 Non-control data manipulating rootkit detection systems.

Name	Monitor	OS Semantics Acquisition	Detection/ Prevention (D/P)	Continuous/ Periodic (C/P)	Requires OS Modifications (Y/N)	Requires the OS source code (Y/N)
ATOM	hypervisor	VMI	D	C	N	N
VMwatcher	hypervisor	VMI	D	P	N	N
Copilot extension	co-processor	Manual specifications	D	P	N	N
Sentry	hypervisor	OS instrumentation	P	C	Y	Y
KG	hardware emulator	OS source code, kernel debug symbols	P	C	N	Y
VMhuko	hypervisor	OS source code, kernel debug symbols	P	C	N	Y

5.4 System Architecture

5.4.1 Overview

As shown in Figure 5.11, ATOM is composed of a front-end component running in the monitoring VM and a back-end component running inside the hypervisor. The back-end component suspends the monitored VM on detecting a new process. After suspending the VM, the back-end component notifies the front-end component to extract the new process's access token and analyze the privileges and the SIDs in the token using the real-time kernel data structure monitoring system. Following the token analysis, the front-end component asks the back-end component to resume the VM execution and to initiate the monitoring of the memory portion containing the extracted access token. The back-end component tracks all attempts to

overwrite the privileges and the SIDs in the access token. When a write is detected, the back-end component notifies the front-end component to analyze the altered access token and to alert the administrator if an access token manipulation attack is detected.

5.4.2 System Design and Implementation

The ATOM architecture is built on the Intel VT hardware and the Xen hypervisor, and is designed to support VMs running both Linux Ubuntu Jaunty and Windows XP.

5.4.2.1 Creation of a New Process

We assume the hypervisor component is started before any malicious process is running. ATOM continuously watches for newly created processes in each monitored VM so that it can extract its access token. To intercept process creation operations, ATOM keeps track of the Flink and Blink pointers in the structure pointed to by PsActiveProcessHead. If ATOM observes a write to either Blink or Flink on the page containing this structure (Step 1 of Figure 5.11), it traverse the processes lists to determine if a new EPROCESS structure has been created or if an existing process has been terminated. The process creation steps in Windows OS, as shown in Figure 5.12, up to the access token set up have already been done. As soon as the back-end component of ATOM detects a new process in a VM, it suspends the VM, and notifies ATOM's front-end component to take over. The front-end component uses the process's EPROCESS data structure to track down the new process's TOKEN data structure so it can analyze the privileges and SIDs.

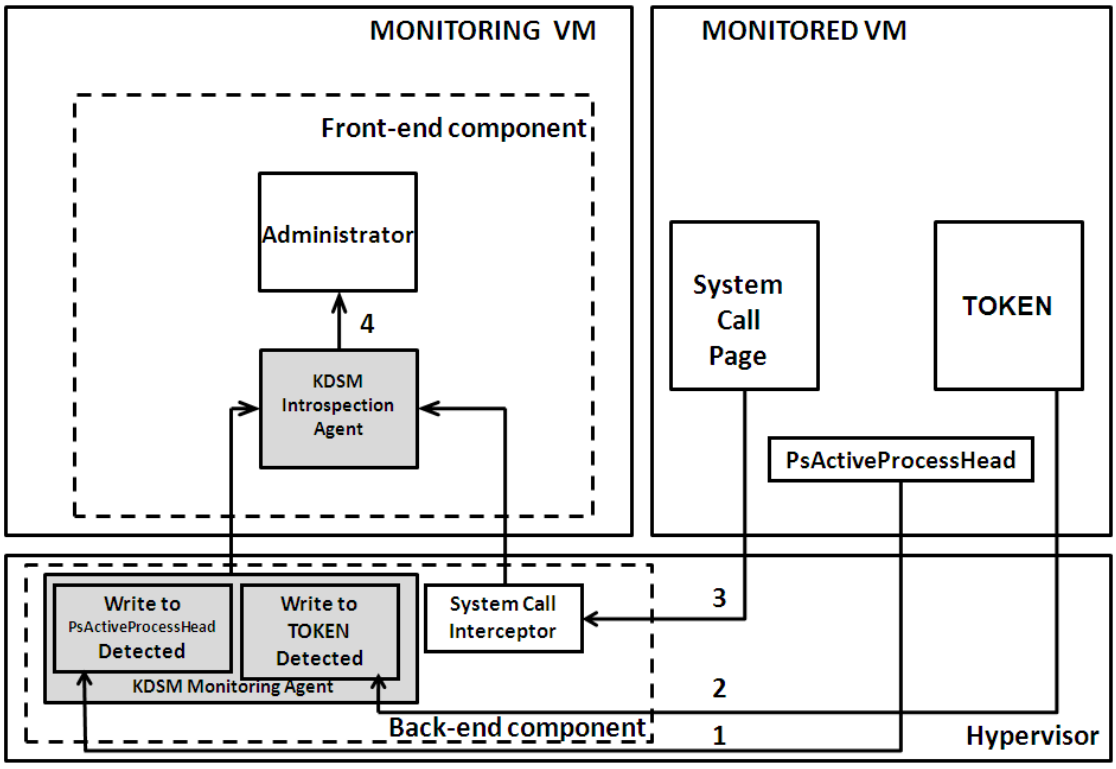

Figure 5.11 The ATOM architecture.

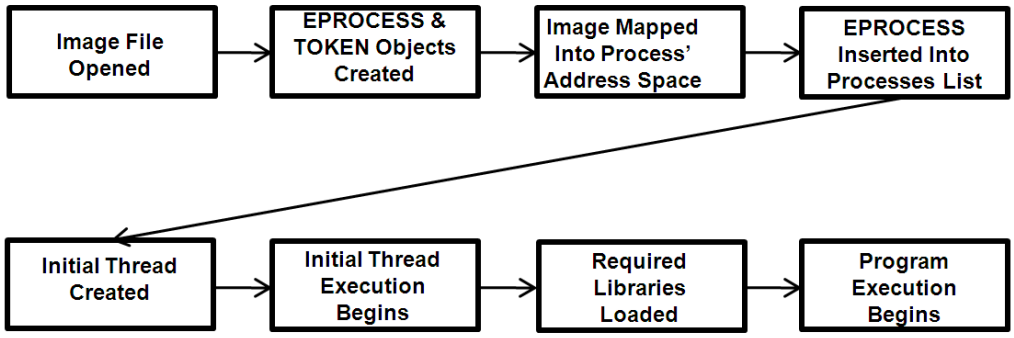

Figure 5.12 Windows OS process creation flow.

5.4.2.2 Access Token Analysis

The process's TOKEN data structure is identified in the physical memory using the value

of the Token field in the corresponding EPROCESS data structure. The rtkdsm.py plugin implemented in the RTKDSM system links up a process to a particular user account by extracting all the SIDs contained in the process's access token and mapping the SIDs' values to their corresponding usernames and user groups. While some of the SIDs have well-known values and can be easily mapped to their associated user or group name, other SIDs require additional processing to determine the username associated with each SID. This additional processing involves extraction of information from the machine's registry. Specifically, we process the *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList* registry key hive shown in Figure 5.13 to extract the list of all local user account SIDs on the machine.

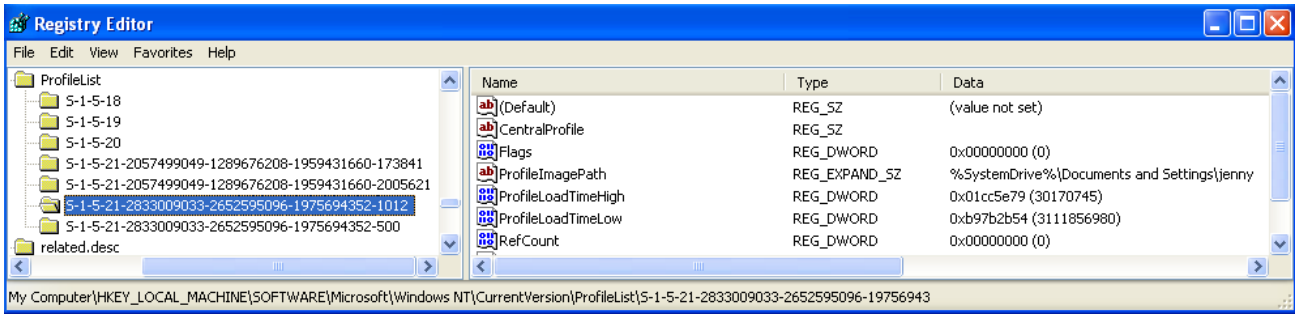

Figure 5.13 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList key hive

The username for each SID can be inferred by looking at the *ProfileImagePath* string value inside the *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList\<SID>* key. Using the Volatility's registry-related APIs, we extract all the local user account SIDs on the machine contained as subkeys in the "*HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList*"

registry key hive. The *ProfileImagePath* string value is usually of the *%SystemDrive%\Documents and Settings\<username>* form, where the value of the <username> string is the actual username. We use the Volatility's registry-related APIs to extract the SIDs in the ProfileList key hive and map them to their usernames. Apart from the individual account SIDs, we also extract well-known group SIDs. These SIDs have been set aside for specific purposes and are the same on any Windows machine.

5.4.2.3 Access Token Monitoring

ATOM performs real-time monitoring of the extracted token using the following steps:

1) Using the rtkdsm.py plugin, ATOM accesses and saves the current values of the SIDs, privileges, and the counts of the privileges and SIDs;

2) The front-end component requests the RTKDSM system to calculate memory ranges containing the Token member of the EPROCESS, the SIDs, privileges, and their counts. The back-end component is notified to monitor the calculated ranges for writes.

   For Windows XP and older Windows version, the following formula is used to calculate the variable memory region size containing the SIDs and the privileges: (Size of _LUID_AND_ATTRIBUTES structure) * PrivilegeCount + (Size of _SID_AND_ATTRIBUTES structure) * UserAndGroupCount + (Size of _SID structure) * UserAndGroupCount. For Windows Vista and later Windows versions, the following formula is used to calculate the variable memory region size: (Size of _SEP_TOKEN_PRIVILEGES structure) + (Size of _SID_AND_ATTRIBUTES structure) * UserAndGroupCount + (Size of _SID structure) * UserAndGroupCount.

The two formulas differ because in Windows Vista and later, the privileges are stored in a bitmap form inside an SEP_TOKEN_PRIVILEGES structure as shown in Figure 5.14. Each field (Present, Enabled, and EnabledByDefault), being of type UINT64, has the potential of holding up to 64 distinct privileges, each identified by an index within the bitmap; the Present field holds the active privileges bitmap, while the Enabled and EnabledByDefault fields keep track of the status of the privileges similar to the Attributes field in older Windows implementations.

```
typedef struct _SEP_TOKEN_PRIVILEGES
{
  UINT64 Present;
  UINT64 Enabled;
  UINT64 EnabledByDefault;
} SEP_TOKEN_PRIVILEGES, *PSEP_TOKEN_PRIVILEGES;
```

Figure 5.14 _SEP_TOKEN_PRIVILEGES structure in Windows Vista and later Windows versions

3) If a write is detected at a monitored memory region (Step 2 of Figure 5.11), the front-end component is notified by the back-end component so it can repeat the token analysis. Depending on the memory region where the write is detected, we classify write instances into the 4 categories:

   i. *False token stealing attack* - a write is detected to the Token field of the EPROCESS data structure. The new address is different from the previous address, and it points to an invalid token. For instance, following the process termination, the Token field of the EPROCESS structure is overwritten as a result of the EPROCESS data structure de-allocation.

ii. *True token stealing attack* - a write is detected to the Token field of the EPROCESS data structure. The new value points to a valid token. ATOM extracts the new values of the privileges/SIDs, their counts and compares them to the previously saved values, and alerts the administrator about the changes (Step 4 of Figure 5.11).

iii. *False token patching attack* – a write is detected to the privileges/SIDs following a system call. To modify a process token, Windows provides the *NtAdjustPrivilegesToken* and *NtAdjustGroupsToken* system calls. We intercept the *NtAdjustPrivilegesToken* and *NtAdjustGroupsToken* system calls using the CLAW system call interception technique (Step 3 of Figure 5.11). If the write to the token is caused by a system call, we consider it a false token patching attack and thus, do not notify the administrator. In our implementation, we do not consider adversarial attempts to evade detection by invoking a token-modifying system call concurrently with a DKOM attack.

iv. *True token patching attack* – a write is detected to the privileges/SIDs and their counts, and it is not a result of the *NtAdjustPrivilegesToken* and *NtAdjustGroupsToken* system calls. Using the rtkdsm.py plugin, ATOM extracts the new values of the privileges/SIDs/their counts, compares them to the previously saved values, and alerts the administrator about the changes (Step 4 of Figure 5.11).

5.4.3 "Always-on" and "Periodic Polling" Monitoring Modes

The ATOM implementation supports both the "always-on" and "periodic polling"

monitoring modes. The system operations and the related security implications were described in the RTKDSM system.

5.4.4 ATOM Implementation for Linux OS

5.4.4.1 Background

5.4.4.1.1 Process Credentials

In the kernel versions < 2.6.29, user privileges are stored in the uid, euid, gid, and egid fields of the task_struct data structure (Figure 5.15). In the kernel versions >= 2.6.29, the task_struct was changed along with the logic of how access to the process credentials (Figure 5.16). The cred data structure was introduced and contains the uid, euid, gid, and egid fields.

```
struct task_struct {
      ...
      /* pointers to next and previous task_struct in the task vector */
      struct task_struct *next_task, *prev_task;
      ...
      int pid;
      ...
      /* pointers to (original) parent process, youngest child, younger sibling,
         older sibling, respectively. */
      struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
      ...
      /* process credentials */
      unsigned short uid,euid,suid,fsuid;
      unsigned short gid,egid,sgid,fsgid;
      ...
      struct mm_struct *mm;
};
```

Figure 5.15 The task_struct data structure.

```
struct task_struct {
      ...
      const struct cred *cred;
      ...
};

struct cred {
      ...
      uid_t       uid;        /* real UID of the task */
      gid_t       gid;        /* real GID of the task */
      uid_t       suid;       /* saved UID of the task */
      gid_t       sgid;       /* saved GID of the task */
      uid_t       euid;       /* effective UID of the task */
      gid_t       egid;       /* effective GID of the task */
      uid_t       fsuid;      /* UID for VFS ops */
      gid_t       fsgid;      /* GID for VFS ops */
      ...
};
```

Figure 5.16 The task_struct and cred data structures in Linux kernel versions >= 2.6.29.

5.4.4.1.2 Rootkit Attacks on Process Credentials

To alter the process's credentials in the kernel versions < 2.6.29, Linux rootkits overwrite the credentials fields with 0 as shown in Figure 5.17. Later 2.6 Linux versions adopted a cred structure to hold all information related to the privileges of a process. To alter the process's credentials in the kernel versions >= 2.6.29, rootkits overwrite the credentials as shown in Figure 5.17. The prepare_creds function first prepares a new set of credentials by allocating and constructing a duplicate of the process's credentials. The commit_creds function commits the new credentials to the current process. To simplify the privilege escalation path, a number of rootkits simply find another process that has the privileges of root and that never exits, usually PID 1, and set the cred pointer of the target process to that of PID 1's. This effectively gives the attacker's process full control, and the rootkit does not have to attempt the non-trivial task of allocating its own cred structure.

```
#if LINUX_VERSION_CODE < KERNEL_VERSION(2, 6, 29)
      current->uid   = current->gid   = 0;
      current->euid  = current->egid  = 0;
      current->suid  = current->sgid  = 0;
      current->fsuid = current->fsgid = 0;

#else
      new = prepare_creds();
      if ( new != NULL ) {
      new->uid   = new->gid   = 0;
      new->euid  = new->egid  = 0;
      new->suid  = new->sgid  = 0;
      new->fsuid = new->fsgid = 0;
      commit_creds(new);
      }
#endif
```

Figure 5.17 Rootkit attacks on process credentials in Linux OS.

5.4.4.2 Implementation

5.4.4.2.1 New Process Detection

To intercept process creation operations, ATOM keeps track of the forward and backward pointers in the init_task structure. If ATOM observes a write to the forward or the backward pointer on the page containing the init_task structure, it traverse the processes list to determine if a new task_struct structure has been created or if an existing process has been terminated. As soon as the back-end component of ATOM detects a new process in a VM, it suspends the VM, and notifies ATOM's front-end component to take over.

The rtkdsm_linux.py plugin implemented in the RTKDSM system operates similarly to the rtkdsm.py plugin in Windows OS. The process's credentials are found in the physical memory using the corresponding task_struct data structure. The rtkdsm_linux.py plugin reads the values of the uid, euid, gid, and egid contained within this data structure.

5.4.4.2.2 Credentials Monitoring

ATOM performs continuous monitoring of the process's credentials. The monitoring involves the following steps:

1) Using the rtkdsm_linux.py plugin, ATOM accesses and saves the values of the uid, euid, gid, and egid;

2) The front-end component requests the RTKDSM system to calculate memory ranges containing the uid, euid, gid, and egid fields. The back-end component is notified to monitor the calculated ranges for writes.

3) If a write is detected at a monitored memory region, the front-end component is notified by the back-end component to repeat the analysis of the credentials. Depending on the memory region for which the write has been detected, we classify write instances into the 4 categories:

    i. *True credentials stealing attack* - a write is detected at the cred field of the task_struct data structure. The new value points to a valid cred data structure. ATOM extracts the new values of the credentials, compares them to the previously saved values, and alerts the administrator about the changes. This credentials stealing attack is specific to the Linux versions >= 2.6.29 only.

    ii. *False credentials stealing attack* - a write is detected at the cred field of the task_struct data structure. The new address is different from the previous address, and it points to an invalid cred. For instance, following the process termination, the cred field of the task_struct structure is overwritten as a result of the task_struct data structure de-allocation.

iii. *False credentials patching attack* – a write is detected to the uid, euid, gid, and egid following a system call. We intercept the setuid and setgid system calls using the CLAW system call interception technique. If the write is caused by a system call, we consider it a false credentials patching attack and thus, do not notify the administrator.

iv. *True credentials patching attack* – a write is detected to the uid, euid, gid, and egid, and it is not caused by a system call. Using the rtkdsm_linux.py plugin, ATOM extracts the new values of the uid, euid, gid, and egid, compares them to the previously saved values, and alerts the administrator about the changes. The credentials patching attack may occur in all Linux 2.6 versions.

5.4.5 Summary of Data Structures Monitored by ATOM

Table 5.2 provides a summary of the key data structures actively monitored by ATOM.

5.5 Evaluations

5.5.1 Experimental Setup

Our testbed consisted of a virtualized server that used Xen version 3.3 as the hypervisor and Ubuntu 9.04 (Linux kernel 2.6.26) as the kernel for Dom0. The host system had Duo CPU P8600 processor running two cores at 2.4GHz and 2GB of system memory. The ATOM system was installed in the Dom0 domain. In addition, the virtualized server hosted a DomU domain running a default installation of Windows XP OS with the IIS web server. This domain was configured with 512Mb RAM.

Table 5.2 Summary of the data structures monitored by ATOM.

OS	Data Structures (Fields)	Actions Taken
Windows	PsActiveProcessHead (Flink,Blink)	On write, traverse the processes lists to determine if a new EPROCESS structure has been created or if an existing process has been terminated.
	_EPROCESS (Token)	On write, the front-end component is notified by the back-end component to repeat the analysis of the token.
	_TOKEN (UserAndGroupCount, UserAndGroups, Privileges)	On write, the front-end component is notified by the back-end component to repeat the analysis of the token.
Linux	init_task (next, prev)	On write, traverse the processes lists to determine if a new task_struct structure has been created or if an existing process has been terminated.
	task_struct (uid, euid, gid, egid)	On write, the front-end component is notified by the back-end component to repeat the analysis of the credentials.

5.5.2 Experiments

5.5.2.1 Effectiveness

5.5.2.1.1 Token Patching Attack

To demonstrate the effectiveness of the ATOM in detecting token patching attacks, we performed an attack using the Fu rootkit [70, 79]. Fu allows the intruder to hide information from user-mode applications and kernel-mode modules by directly modifying kernel data structures used by the operating system, such as, removing entries from the process and loaded modules

linked lists. In addition, Fu is capable of modifying a process's token to change the process's privileges and replacing the process's owner SID.

Prior to the attack, the Fu rootkit was loaded in the test VM. The malicious process was then started in the VM. The ATOM system running in the "always-on" mode detected the new process and began monitoring its token. The Fu rootkit was directed to modify the malicious process's privileges and SIDs contained in the token. The ATOM system detected the writes to the token and alerted the administrator about the attack.

We further performed a token patching attach with the system running in the "periodic polling" mode with the timing parameter T set to 50 msec. We modified privileges in the malicious process's token and immediately restored them to their original values to avoid detection. Although the first write was detected by ATOM, the overall attack involving overwriting of multiple privileges was not. Despite an improved performance in the "periodic polling" approach as was shown in Chapter 2, the "periodic polling" mode reduced the ATOM effectiveness and provided a lesser degree of assurance. The experiment illustrated that system execution in the "periodic polling" mode introduced a window of vulnerability between two consecutive checks on the monitored data structures. However, by setting the timing parameter T to 5 msec, the ATOM system was routinely able to detect the token patching attempts.

5.5.2.1.2 Token Stealing Attack

To demonstrate the effectiveness of the ATOM system in detecting token stealing attacks, we performed a token stealing attack using the attack code presented in [72]. We started two processes in the test VM – a victim process running with the privileges of the "Administrators" user group and a malicious process running with the privileges of the "Users"

user group. The ATOM system running in the “always-on” mode detected the new processes and began monitoring their tokens. The token stealing code copied the desired access token from the victim process and exchanged the original value of the Token field in the malicious process with the address of the copied token. Following this operation, the malicious process had the same access rights as the victim process. The ATOM system detected the write to the Token field of the malicious process and alerted the administrator about the attack.

5.5.2.2 Performance Assessment

The VM performance is impacted by the following ATOM monitoring components: (1) monitoring of the PsActiveProcessHead structure; (2) monitoring of the EPROCESS data structures; (3) monitoring of the TOKEN data structures; (4) CLAW system call interception. The performance impact of the PsActiveProcessHead, EPROCESS, and TOKEN data structure monitoring using the RTKDSM system in “always-on” and “periodic polling” mode was shown in Chapter 2. The performance impact of the CLAW system call interception was shown in Chapter 4.

5.6 Summary

We presented a detection system called ATOM that used the RTKDSM system to intercept DKOM-based access token manipulation attacks targeting non-control data. This class of attacks is difficult to detect using the existing defensive methods for control-data manipulating attacks. The ATOM defensive approach consisted of monitoring all write accesses to memory pages containing access tokens of running processes and real-time analysis of tokens when updates targeting privileges in a token were detected. To avoid false positives caused by the OS

supported token-modifying system calls, we installed a system call interception mechanism enabling ATOM to differentiate between writes caused by system calls vs. DKOM writes. Our evaluation of ATOM showed that the system was able to successfully detect DKOM-based token manipulation attacks using the presented techniques.

The semantic knowledge and memory locations of data structures targeted by DKOM attacks were the key data required by our implementation. Both of these data could be obtained through the Volatility framework for any of its supported data structures and provided as an input into the ATOM system making the ATOM approach directly applicable for protection of other critical data structures that might be targeted by DKOM attacks.

6 Conclusions and Future Work

6.1 Conclusions

Over the last few years, VMI technology has evolved to monitor VM behavior in an agentless fashion. VMI provides a constellation of information about the states of all running VMs making the agentless approach superior to the traditional in-host agent-based monitoring. The contribution of VMI is especially prominent in security tools, such as virus scanners and intrusion detection systems. By de-coupling security tools from the internal OS execution environment, VMI makes them resilient to malicious attacks.

However, the VMI approach comes at a cost - VMI applications must deal with the semantic gap issues requiring extensive knowledge and reconstruction of the guest OS data structures. Reconstruction is commonly done from scratch leading to correctness challenges, increasing the likelihood of buggy introspection, and limiting flexibility and extensibility of VMI tools. As a result, generality of manual reconstructions is poor since the VMI tool is tied to the guest OS. This problem is exacerbated if the guest OS is closed-source.

As forensic analysis tools aim to tackle many of the same issues that plague VMI tools, the forensic community has already done much of the work bridging the semantic gap to support multiple operating systems and a large number of kernel data structures. Several VMI studies have previously proposed the use of forensic methods and tools for rapid data structure reconstruction. However, existing forensic analysis tools are designed for an offline analysis and thus, lack capabilities required by VMI tools to implement active monitoring techniques capable of analyzing and detecting events as they occur.

6.1.1 RTKDSM

This research focuses on describing the RTKDSM framework designed to *automatically* reconstruct kernel data structures of interest and to *continuously* monitor states of the reconstructed data structures in real-time to support active monitoring. The RTKDSM system is the first VMI framework leveraging a forensic framework to track changes in the reconstructed data structures in real-time. By building on top of the forensic tool acumen, the RTKDSM system reduces the complexity of developing VMI applications associated with data structure reconstruction and by extension the likelihood of buggy introspection. Leveraging the Volatility framework, the RTKDSM system eliminates effort duplications supporting the common modular motif in computer science. These ideas have been previously proposed but not developed to be practically usable. This objective has been achieved in this study. The RTKDSM system is capable of supporting a wide range of VMI applications due to the RTKDSM framework's flexibility and extensibility, which has been lacking until now. This research has demonstrated effectiveness and practicality of the RTKDSM framework by building three novel system prototypes, vCardTrek, CLAW, and ATOM, which can be easily adapted for data flow tracking and security monitoring in industrial settings.

6.1.2 vCardTrek

vCardTrek is the first published example of a VMI system used for the development of a VMI tool for data flow tracking, thus moving the concept of VMI-based monitoring beyond the usual virus and intrusion detection applications. Moreover, the main difference between vCardTrek and other tools with a similar goal is that by applying VMI, it does not rely on machine or application instrumentation when dealing with multiple machines. The conceptual

framework devised in this work could be applied to designing similar tools for real-world payment card processing applications running on real-world computing environments.

Table 6.1 Summary of the data structures used by vCardTrek, CLAW, and ATOM.

System	Data Structures
vCardTrek	ADDRESS_OBJECT TCPT_OBJECT EPROCESS
CLAW	PsActiveProcessHead PEB_LDR_DATA init_task
ATOM	PsActiveProcessHead EPROCESS TOKEN init_task task_struct

6.1.3 CLAW and ATOM

The problems addressed by CLAW and ATOM systems are challenging because of the four restrictive requirements: (1) acting in a preventive mode, that is, the ability to detect events as they occur, (2) OS-independence, that is, no modifications to the monitored OS or installation of agents inside the OS, (3) direct applicability of the approach to HVM machines, which are the main stream in virtualization, and (4) finally, the ability to intercept system calls selectively. The main difference between these tools and other previously published tools with similar goals is the ability to address the four requirements in one system made possible due to the novelty of the

RTKDSM and CLAW system call interception techniques developed in this dissertation.

6.2 Future Work

6.2.1 RTKDSM

The RTKDSM system currently provides a solid foundation for active monitoring in a virtualized environment. Yet, our experience working with the RTKDSM system highlighted some areas that would benefit from additional research. An important problem that needs to be addressed by the future research is how to enable the RTKDSM system to automatically and dynamically choose between the “always-on” and the “periodic polling” mode without affecting VMI applications’ performance and the timeliness of detection. Our research has shown that some data structures are consistently allocated on memory pages that experience frequent spurious updates unrelated to the data structure itself making the “periodic polling” mode more suitable for monitoring of such data structures. For this group of data structures, the next step is to quantify the number of kernel data structures changes that may be missed as a result of different polling frequencies in the “periodic polling” mode to help determine the optimal polling interval to ensure timeliness of detection. Our research has also shown that some data structures are allocated in memory pages that are rarely updated, thereby monitoring of such data structures can be done in the “always on” mode without impacting the performance. Hence, the next stage of our work is to investigate memory locations common to various data structure types and to add capabilities to the RTKDSM system to dynamically choose the appropriate monitoring mode depending on the data structure type. Furthermore, machine learning techniques may be applied to efficiently train the RTKDSM system to choose between the “always-on” and “periodic polling” mode.

6.2.2 ATOM and CLAW

From our experience developing the ATOM system, the next stage of this work is to generalize the ATOM approach by applying it to any Volatility-supported data structure type. This step will require updating the CLAW system call interception mechanism with system calls lists relevant to various data structure types enabling the extensibility of the ATOM approach to detection of DKOM attacks on any kernel data structure. Because the RTKDSM approach is able to detect changes in general, rather than focusing on specific symptoms of known DKOM attacks, the future ATOM system will be able to detect both known as well as unseen previously DKOM attacks.

6.2.3 vCardTrek

The next stage of our vCardTrek work is to develop support for persistent TCP connections and intra-host cross-process communications. Specifically, in our implementation, we monitor TCP connections to track card data flow across multiple VMs. vCardTrek initiates a search of the memory of a VM only when it is involved in a newly established TCP connection. In the future, we plan to support persistent TCP connections, which may stay open for a long time and service multiple transactions. Also, our coarse-grained data flow tracking mechanism does not currently handle data flow tracking of cross-process communications within the same VM. More research is required to determine the extent to which data flow tracking can be implemented via cross-process intra-host TCP connections, pipes, and shared memory. This is an important direction of future research. Finally, we would like to conduct additional evaluations of vCardTrek on testbeds that mimic production environments to identify actual limitations of the tool's current design or implementation.

Bibliography

[1] A. Srivastava and J. Giffin. Tamper-resistant, application-aware blocking of malicious network connections. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID 2008)*, pages 39-58, September 2008. ISBN:978-3-540-87402-7.doi:10.1007/978-3-540-87403-4_3. http://dl.acm.org/citation.cfm?id=1433011.

[2] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE SP 2008)*, pages 233-247, May 2008. ISBN:978-0-7695-3168-7.doi:10.1109/SP.2008.24. http://dl.acm.org/citation.cfm?id=1398072.

[3] X. Jiang, A. Wang, and D. Xu. Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, pages 128-138, October 2007. ISBN:978-1-59593-703-2.doi:10.1145/1315245.1315262. http://dl.acm.org/citation.cfm?id=1315262.

[4] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS 2003)*, pages 191-206, February 2003.

[5] B. D. Payne, M. Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, pages 385-397, December 2007. ISBN:978-0-7695-3060-4. doi:10.1109/ACSAC.2007.10. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4413005.

[6] B. D. Payne. XenAccess Library. http://code.google.com/p/xenaccess/.

[7] L. Litty and D. Lie. Manitou: A layer-below approach to fighting malware. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID 2006)*, pages 6-11, October 2006. ISBN:1-59593-576-2.

doi:10.1145/1181309.1181311. http://dl.acm.org/citation.cfm?id=1181311.

[8] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. VMM-based hidden process detection and identification using Lycosid. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2008)*, pages 91-100, March 2008. ISBN:978-1-59593-796-4. doi:10.1145/1346256.1346269. http://dl.acm.org/citation.cfm?id=1346269.

[9] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proceedings of the 2006 USENIX Annual Technical Conference (USENIX ATEC 2006)*, pages 1-14, June 2006. http://dl.acm.org/citation.cfm?id=1267360.

[10] L. Litty, H.A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *Proceedings of the 17th USENIX Security Symposium (USENIX SS 2008)*, pages 243-258, July 2008.doi:10.1.1.145.2378. http://dl.acm.org/citation.cfm?id=1496728.

[11] B. D. Payne. Simplifying virtual machine introspection using LibVMI. http://prod.sandia.gov/techlib/access-control.cgi/2012/127818.pdf.

[12] B. Hay and K. Nance. Forensics examination of volatile system data using virtual introspection. *ACM SIGOPS Operating Systems Review*, vol. 42, issue 3, pages 75-83, April 2008. doi:10.1145/1368506.1368517. http://dl.acm.org/citation.cfm?id=1368517.

[13] K. Nance, M. Bishop, and B. Hay. Investigating the implications of virtual machine introspection for digital forensics. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES 2009)*, pages 1024-1029, March 2009. ISBN:978-1-4244-3572-2.doi:10.1109/ARES.2009.173. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5066605.

[14] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (IEEE SP 2011)*, pages 297-312, May 2011. ISBN:978-1-4577-0147-4.

[15] Z. Gu, Z. Deng, D. Xu, and X. Jiang. Process implanting: A new active introspection

framework for virtualization. In *Proceedings of the 30th IEEE Symposium on Reliable Distributed Systems (SRDS 2011)*, pages 147-156, July 2011. ISBN:978-0-7695-4450-2. doi:10.1109/SRDS.2011.26. http://dl.acm.org/citation.cfm?id=2085362.

[16] N. Petroni, T. Fraser, J. Molina, and W. Arbaugh. Copilot—a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium (USENIX SS 2004)*, pages 179-194, August 2004. doi:10.1.1.93.5047. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.93.5047.

[17] J. Rhee, R. Riley, D. Xu, and X. Jiang. Defeating dynamic data kernel rootkit attacks via VMM-based guest-transparent monitoring. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES 2009)*, pages 74-81, March 2009.ISBN:978-1-4244-3572-2.doi:10.1109/ARES.2009.116. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5066457.

[18] A. Srivastava, I. Erete, and J. Giffin. Kernel data integrity protection via memory access control. *Georgia Institute of Technology*, 2009. http://hdl.handle.net/1853/30785.

[19] Xen Project. http://www.xenproject.org/.

[20] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 164–177, December 2003.ISBN:1-58113-757-5.doi:10.1145/945445.945462. http://dl.acm.org/citation.cfm?id=945462.

[21] C. Betz. DFRWS 2005 Forensics Challenge: Memparser Analysis Tool. http://www.dfrws.org/2005/challenge/memparser.shtml.

[22] Volatile Systems, LLC. The Volatility framework: Volatile memory artifacts extraction utility framework. https://www.volatilesystems.com/default/volatility.

[23] A. Schuster. Pool allocations as an information source in Windows memory forensics. In *Proceedings of the International Conference on IT-Incidents Management & IT-Forensics (IMF 2006)*, pages 104-115, October 2006.

[24] A. Schuster. Searching for processes and threads in Microsoft Windows memory dumps.

The International Journal of Digital Forensics and Incident Response, vol. 3, pages 10-16, September 2006. doi:10.1016/j.diin.2006.06.010. http://dl.acm.org/citation.cfm?id=2296386.

[25] Bugcheck. Grepexec: Grepping executive objects from pool memory. Uninformed Journal, vol. 4, June 2006. http://www.uninformed.org/?v=4&a=2.

[26] A. Schuster. Ptfinder. http://computer.forensikblog.de/en/2006/03/ptfinder_0_2_00.html.

[27] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, and D. Xu. DKSM: Subverting virtual machine introspection for fun and profit. In *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems (SRDS 2010)*, pages 82-91, November 2010. ISBN:978-0-7695-4250-8.doi:10.1109/SRDS.2010.39. http://doi.ieeecomputersociety.org/10.1109/SRDS.2010.39.

[28] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC 2008)*, pages 77-86, December 2008. ISBN:978-0-7695-3447-3. doi:10.1109/ACSAC.2008.29. http://dl.acm.org/citation.cfm?id=1468197.

[29] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)*, pages 566-577, November 2009. ISBN:978-1-60558-894-0. doi:10.1145/1653662.1653730. http://dl.acm.org/citation.cfm?id=1653730.

[30] Futuremark. PCMark05. http://www.futuremark.com/benchmarks/pcmark05/.

[31] Linux/Unix nbench. http://www.tux.org/~mayer/linux/bmark.html.

[32] ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.2/programs/ab.html.

[33] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pages 211-224, 2002. doi:10.1145/844128.844148. http://dl.acm.org/citation.cfm?id=844148.

[34] Privacy Rights Clearinghouse. Chronology of Data Breaches.

https://www.privacyrights.org/data-breach-header-top.

[35] PCI Security Standards Council. https://www.pcisecuritystandards.org/.

[36] Pippard, Inc. Bringing virtualization and thin computing technology to POS. http://www.retailsolutionsonline.com/doc/Brining-Virtualization-And-Thin-Computing-0001.

[37] Microsoft Corporation. Restaurant chain upgrades systems and cuts 2,000 servers using virtual machines. http://download.microsoft.com/documents/customerevidence/7146_jack__in_the_box_cs.doc.

[38] Micros Systems, Inc. Micros Systems announces deployment of micros 9700 HMS at M Resort Spa Casino in Las Vegas. http://www.micros.com/NR/rdonlyres/3E357BE8-70DB-468D-B9AB-68F0E784527F/2296/MResort.pdf.

[39] H.C. Kim, A.D. Keromytis, M. Covington, and R. Sahita. Capturing information flow with concatenated dynamic taint analysis. In *Proceedings of the 4th International Conference on Availability, Reliability and Security (ARES 2009)*, pages 355-362, March 2009. ISBN:978-1-4244-3572-2.

[40] A. Zavou, G. Portokalidis, and A.D. Keromytis. Taint-Exchange: A generic system for cross-process and cross-host taint tracking. In *Proceedings of the 6th International Workshop on Security (IWSEC 2011)*, pages 113-128, November 2011. ISBN:978-3-642-25140-5. http://dl.acm.org/citation.cfm?id=2075670.

[41] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC 2005)*, pages 41-46, June 2005. http://dl.acm.org/citation.cfm?id=1247401.

[42] A. Ho, M. Fetterman, C. Clark, A Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys 2006)*, pages 29-41, October 2006. ISBN:1-59593-322-0. doi:10.1145/1217935.1217939. http://dl.acm.org/citation.cfm?id=1217939.

[43] B. Mazloom, S. Mysore, B. Agrawal, and T. Sherwood. Understanding and visualizing

full systems with data flow tomography. In *Proceedings of the 13 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008)*, pages 211-221, March 2008. ISBN:978-1-59593-958-6. doi:10.1145/1346281.1346308. http://dl.acm.org/citation.cfm?doid=1346281.1346308.

[44] Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. Vahdat, A. C. Snoeren, G. M. Voelker, and S. Savage. Neon: System support for derived data management. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2010)*, pages 63-74, July 2010. ISBN:978-1-60558-910-7. doi:10.1145/1735997.1736008. http://dl.acm.org/citation.cfm?id=1736008.

[45] Ebtables. http://ebtables.sourceforge.net/.

[46] H.P. Luhn. Computer for verifying numbers, U. S. P. Office, 1954.

[47] Able Solutions Corporation. AbleCommerce: Featured clients. http://www.ablecommerce.com/Featured-Clients-C49.aspx.

[48] osCommerce Corporation. osCcommerce: Open source e-commerce solutions. http://www.oscommerce.com/.

[49] 911 Software Corporation. Payment processing software. http://www.911software.com/.

[50] T. Chiueh. Program semantics-aware intrusion detection. http://www.ecsl.cs.sunysb.edu/PAID/index.html.

[51] L. Lam and T. Chiueh. Checking array bound violation using segmentation hardware. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN 2005)*, pages 388-397, June 2005. ISBN:0-7695-2282-3. doi:10.1109/DSN.2005.25. http://dl.acm.org/citation.cfm?id=1078297.

[52] Wikipedia. Address space layout randomization. http://en.wikipedia.org/wiki/Address_Layout_Randomization.

[53] Nologin.org. Remote library injection. http://www.nologin.org/Downloads/Papers/remote-library-injection.pdf.

[54] S. Fewer. Reflective dll injection, October 2008. http://www.harmonysecurity.com/files/HS-P005_ReflectiveDllInjection.pdf.

[55] A. Walters. Fatkit: Detecting malicious library injection and upping the "anti". http://www.4tphi.net/fatkit/papers/fatkit_dll_rc3.pdf.

[56] N. L. Petroni and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, pages 103-115, November 2007. ISBN:978-1-59593-703-2. doi:10.1145/1315245.1315260. http://dl.acm.org/citation.cfm?id=1315260.

[57] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID 2008)*, pages 1-20, September 2008. ISBN:978-3-540-87402-7.doi:10.1007/978-3-540-87403-4_1. http://dl.acm.org/citation.cfm?id=1433008.

[58] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP 2007)*, pages 335-350, December 2007.ISBN:978-1-59593-591-5.doi:10.1145/1294261.1294294. http://dl.acm.org/citation.cfm?id=1294294.

[59] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang. HIMA: A hypervisor-based integrity measurement agent. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC 2008)*, pages 461-470, December 2009. ISBN:978-0-7695-3919-5.doi:10.1109/ACSAC.2009.50. http://dl.acm.org/citation.cfm?id=1723256.

[60] B. Jansen, H.V. Ramasamy, M. Schunter, and A. Tanner. Architecting dependable and secure systems using virtualization. In *Architecting Dependable Systems*. Lecture Notes in Computer Science, vol. 5135, pages 124-149, Springer-Verlag, Berlin, Heidelberg (2008).ISBN:978-3-540-85570-5.doi:10.1007/978-3-540-85571-2_6. http://dl.acm.org/citation.cfm?id=1428281.

[61] K. Onoue, Y. Oyama, and A. Yonezawa. Control of system calls from outside of virtual machines. In *Proceedings of the 2008 ACM Symposium on Applied computing (SAC 2008)*, pages 2116-2121, March 2008. ISBN:978-1-59593-753-7. doi:10.1145/1363686.1364196. http://dl.acm.org/citation.cfm?id=1364196.

[62] X. Jiang and X. Wang. "Out-of-the-box" monitoring of VM-based high-interaction honeypots. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection (RAID 2007)*, pages 198-218, September 2007. ISBN:3-540-74319-7. http://dl.acm.org/citation.cfm?id=1776450.

[63] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A tool for analyzing malware. In Proceedings of the 15th European Institute for Computer Antivirus Research Annual Conference (EICAR 2006), pages 180-192, April 2006.

[64] L. Xu and Z. Su. Dynamic detection of process-hiding kernel rootkits. Technical Report CSE-2009-24, University of California at Davis, 2009. http://leo.cs.ucdavis.edu/techrep/CSE-2009-24.pdf.

[65] C. Maiero and M. Miculan. Unobservable intrusion detection based on call traces in paravirtualized systems. In *Proceedings of International Conference on Security and Cryptography (SECRYPT 2011)*, SciTePress, July 2011.

[66] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS 2008)*, pages 51-62, October 2008. ISBN:978-1-59593-810-7.doi:10.1145/1455770.1455779. http://dl.acm.org/citation.cfm?id=1455770.1455779.

[67] Rapid7. Metasploit penetration testing software. http://www.metasploit.com.

[68] Microsoft. MS08-067: Vulnerability in server service could allow remote code execution. http://support.microsoft.com/kb/958644.

[69] Passmark Software. AppTimer. http://www.passmark.com/products/apptimer.htm.

[70] G. Hoglund and J. Butler. Rootkits: Subverting the Windows kernel. Addison-Wesley Professional, 2005. ISBN:0321294319. http://dl.acm.org/citation.cfm?id=1076346.

[71] Microsoft. Sysinternals Process Utilities. http://technet.microsoft.com/en-us/sysinternals/bb795533.aspx.

[72] C. Barta. Token Stealing. http://www.ntdsxtract.com/downloads/Token_stealing.pdf.

[73] A. Baliga, X. Chen, and L. Iftode. Paladin: Automated detection and containment of

rootkit attacks. Rutgers University Department of Computer Science, 2006.

[74] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)*, pages 545-554, November 2009. ISBN:978-1-60558-894-0. doi:10.1145/1653662.1653728. http://dl.acm.org/citation.cfm?id=1653728.

[75] H. Yin, Z. Liang, and D. Song. Hookfinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS 2008)*, February 2008.

[76] Z. Wang, X. Jiang, W. Cui, and X. Wang. Countering persistent kernel rootkits through systematic hook discovery. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID 2008)*, pages 21-38, September 2008. ISBN:978-3-540-87402-7.doi:10.1007/978-3-540-87403-4_2. http://dl.acm.org/citation.cfm?id=1433009.

[77] N. L. Petroni, T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the 15th USENIX Security Symposium (USENIX SS 2006)*, pages 289-304, August 2006. http://dl.acm.org/citation.cfm?id=1267356.

[78] D. Tian, D. Kong, H. Changzhen, and P. Liu. Protecting kernel data through virtualization technology. In *Proceedings of the 4th International Conference on Emerging Security Information Systems and Technologies (SECURWARE 2010)*, pages 5-10, July 2010. ISBN:978-0-7695-4095-5.doi:10.1109/SECURWARE.2010.9. http://dl.acm.org/citation.cfm?id=1916038.

[79] J. Butler, J. Undercoffer, and J. Pinkston. Hidden processes: the implication for intrusion detection. In *Proceedings of the 2003 IEEE Workshop on Information Assurance*, pages 116-121, June 2003.