DETECTING AND MITIGATING MALWARE

IN VIRTUAL APPLIANCES

by

Prashanth Nayak

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

December 2014

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Prashanth Nayak**

has been approved by the following supervisory committee members:

Eric Eide	, Chair	**07/16/2014** Date Approved
John Regehr	, Member	**07/16/2014** Date Approved
Jacobus Van der Merwe	, Member	**07/16/2014** Date Approved

and by **Ross Whitaker** , Chair of

the School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

System administrators use application-level knowledge to identify anomalies in virtual appliances (VAs) and to recover from them. This process can be automated through an anomaly detection and recovery system. In this thesis, we claim that application-level policies defined over kernel-level application state can be effective for automatically detecting and mitigating the effects of malicious software in VAs.

By combining user-defined application-level policies, virtual machine introspection (VMI), expert systems, and kernel-based state management techniques for anomaly detection and recovery, we are able to provide a favorable environment for the execution of applications in VAs. We use policies to specify the desired state of the VA based on an administrator's application-level knowledge. By using VMI we are able to generate a snapshot that represents the true internal state of the VA. An expert system evaluates the snapshot and identifies any violations. Potential violations include the execution of an irrelevant application, an unauthorized process, or an unfavorable environment configuration. The expert system also reasons about appropriate recovery strategies for each of the violations detected. The recovery strategy decided by the expert system is carried out by recovery tools so that the VA can be restored to an acceptable state.

We evaluate the effectiveness of this approach for anomaly detection and repair by using it to detect and recover from the actions of different types malicious software targeting a web server VA. The system is shown to be effective in guarding the VA against the actions of a kernel-exploit kit, a kernel rootkit, a user-space rootkit, and an application malware. For each of these attacks, the recovery component was able to restore the VA to an acceptable state. Although, the recovery actions carried out did not remove the malicious software, they substantially mitigated the harmful effects of the malicious software.

This thesis is dedicated to my parents for their love, support, and encouragement.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my thesis advisor, Dr. Eric Eide, for the constant guidance, motivation, and support. During the course of this thesis, he has spent endless hours providing me valuable insight into problems and guiding me in the right research direction. Working under his mentorship has truly been a rewarding experience.

I thank my committee members, Kobus van der Merwe, and John Regehr, for their suggestions and feedback, which has helped me improve this work.

I am grateful to David Johnson, who has been extremely kind and has selflessly helped me with all the problems I faced during my research.

Last but not least, I am thankful to Robert Ricci, Mike Hibler, and all other members of the Flux research group for their suggestions and feedback.

This work was supported by the Air Force Research Laboratory and DARPA under Contract No. FA8750–10–C–0242.

CHAPTER 1

INTRODUCTION

Virtual machines have significantly affected present day computing environments. The ease with which a virtual machine can be deployed and the intrinsic properties of virtual machines such as isolation, security, and mobility have moved many computing environments from multitasking to multi-OS computing. Due to the benefits of virtualization such as reduced cost, ease of deployment, and ease of management, more and more applications are being deployed on virtual machines. The current trend is to have all related and mutually dependent applications deployed within a single virtual machine, sharing an execution environment. The resulting lightweight, self-sufficient application containers are known as virtual appliances [30].

A virtual appliance is created by installing a software application on a virtual machine and packaging that into a disk image. Linux-Apache-MySQL-PHP (LAMP) is an example of an application bundle generally deployed on a virtual appliance. In this controlled environment, the Linux operating system, an Apache web server, a MySQL database server, and PHP, Perl, or Python processes would be considered as legitimate. No other application is expected to run in this environment using a noticeable share of the computing resources.

Operating systems and applications are designed to be highly reliable, flexible, and secure. However, bugs in operating system and application code are inevitable. Studies show that there exist six to sixteen bugs per thousand lines of code [24, 25], which makes applications and operating systems vulnerable to attacks that can compromise their integrity. A computer system can be subjected to denial-of-service attacks preventing it from providing its intended service [27]. A system can be subjected to attacks that violate the integrity of the service that it provides [22]. Further, once a system is compromised, an attacker might gain and continue to retain complete control of the application or OS.

The amount of damage caused by an attack is limited when a single application is compromised, and such compromises are often easy to identify and recover from. However, attacks exploit application bugs to gain access to the underling OS. Once the OS is compromised, the scope of the damage is amplified because the attack can now affect all the applications running on top of the OS. Detecting such attacks becomes difficult because the attacker has all the privileges to hide his actions from the user. Further, seamless recovery from these attacks is a challenge because of the lack of available recovery tools and because of the extent of the damage already caused.

The execution environment provided by a virtual appliance may not be always favorable for the execution of the applications that are intended to run within the appliance. There always exists a chance that an irrelevant application may be using a considerable share of the system resources, thereby depriving the intended applications of system resources. The intended applications may crash, become unresponsive or otherwise not provide the intended service. Detecting such conditions requires constant monitoring and validating of the execution environment.

An intrusion detection system (IDS) is software designed to aid in deterring or mitigating the damage that can be caused by an attack. An IDS can detect attempts to compromise the confidentiality, integrity, or availability of a computer or network. Most existing IDSs rely on specification languages, predicates, or contextual inspection to detect anomalies [20, 11, 14]. Once an anomaly is detected, the problem lies in identifying the set of actions to be taken to restore the compromised system to an acceptable state. Generally, IDSs are not equipped to initiate repair actions to recover from an attack [20, 16, 11]. At best they are capable of suspending the execution of an affected application or just shutting down the OS so as to prevent further damage. A human administrator is then responsible to decide the course of repair.

The most simple and widely used solution to recover a computer system from an attack is to reinitialize the system to its starting state. This is generally done through an application or OS restart; the worst case would involve reinstalling the affected application or the OS itself. The drawbacks associated with this are the downtime and loss of state it incurs. In most production environments, this downtime is not appreciated. If the repair requires a complete reinstall, then the state and information gathered by the application may be completely lost. In either of these cases, the abrupt discontinuity in service is directly exposed to the end user.

This thesis investigates an alternative recovery strategy: online repair. We identified the need for a system that can monitor a virtual appliance, detect anomalies, and recover the virtual appliance to an acceptable state. We claim that an administrator's application-level knowledge can be used by an anomaly detection system to detect anomalies. Once an anomaly is detected, generic recovery actions can be performed to restore the virtual appliance to an acceptable state. An acceptable state for a VA is decided by the administrator of that VA and approximated through an application-level policy. Ideally, the policy should be able to capture all the possible acceptable states of the VA and should not include any nonacceptable state.

There are two approaches to restore the virtual appliance to an acceptable state. We term these approaches *application-based state management* and *kernel-based state management*. These two approaches address abstractions at different levels of the software stack and each has its own set of advantages and drawbacks. We decided to base our solution on the kernel-based state management approach because of the advantages associated with that approach.

1.1 Application-Based State Management

This approach allows an administrator to encode knowledge over application-level objects. Facts about applications are expressed as statements over application-level objects and state. For example, for the Apache web server, knowledge is encoded over objects such as requests, responses, configuration files, and enabled modules. This approach relies on a set of tools built specifically for each application. These tools run along with the target applications with full access to the application's resources and data structures. For example, consider the DarkLeech [22] malware that compromises the Apache web server and injects malicious iframes into the web pages that Apache serves. In this case, an application-based recovery strategy could be implemented as an Apache repair module that sits in the Apache process pipeline and detects and removes iframes with a particular signature.

An advantage of application-based state management is that it provides fine-grain control over the repair actions needed to support the continuous execution of applications. Further, since the repairs are application specific, the risk associated with a repair is contained to that particular application. For example, an Apache repair module would not have access to the state and resources of other processes in the VA, and hence it cannot affect their execution.

The main disadvantage of this approach is that application-specific knowledge is required to develop repair tools. Since a different set of repair tools is needed for each and every application, this approach cannot scale well.

1.2 Kernel-Based State Management

This approach allows an administrator to express knowledge over kernel-level objects such as processes, files, loaded objects, and sockets. Facts about applications are expressed as statements over kernel-level objects and state. Any violation of policies would result in a recovery action on appropriate kernel data structures or the kernel's view of the applications. Thus the recovery action relies on tools that run in the kernel space with privileged access to the data structures that the higher-layer applications rely on. Again, consider the Darkleech [22] malware that compromises the Apache web server and injects malicious iframes into web pages. A possible kernel-based recovery strategy would be to overwrite the instructions in the loaded object of the Darkleech module. This overwriting of instructions can be done on the fly without having to stop and restart the Apache process.

The main advantage of kernel-based state management is its generality, which makes it possible to build repair tools that support a wide range of applications. Further, developing repair tools based on this approach does not require application-specific knowledge. These repair tools can affect data structures that are not within the direct control of applications.

The disadvantage associated with this approach is that the scope of the repair may be limited as the repair tools are not application specific. For example, if a configuration file of an application is compromised by malware, then restoring the configuration file to its original state would not be possible using kernel-based state management. Further, the risk associated with the repair may not be restricted only to the affected application.

1.3 Selected Approach

Recent developments in system intrusion techniques, the use of highly crafted attack vectors, and the scope of these attacks demand a powerful yet generic approach for recovery actions. The repair for a detected anomaly may scale beyond the scope of repair tools associated with particular applications. Furthermore, the application-based state management approach would not scale well simply because of the diverse range of available applications. We therefore decided to base our recovery actions on kernel-based state management.

The policy-driven anomaly detection and recovery system we have designed consists of state gathering, anomaly detection, and recovery components. The state-gathering component periodically captures a snapshot of the virtual appliance state. The anomaly detection component validates the point-in-time state information presented by the snapshot against the administrator-specified policies and flags any deviations as anomalies. The recovery component reasons about and enforces an appropriate recovery action that would restore the virtual appliance to an acceptable state.

Our anomaly detection system relies on kernel-level state information of the target virtual appliance, application-level knowledge, and expert systems to detect intrusions. We make use of the Stackdb [17] programming libraries to build tools that capture point-in-time snapshots of the virtual appliance being monitored. These snapshots have kernel-level information of the executing processes, their credentials, memory layout, CPU utilizations, open sockets, and open files. It also has information about the system-call table and the Linux modules loaded by the kernel. This information about the virtual appliance is used as an input to the anomaly detection system.

Our anomaly detection and recovery system is designed to target virtual appliances. Since virtual appliances are generally meant to deploy only one application, the normal behavior of a virtual appliance is quite stable and can be easily defined. The normal behavior of the system can be defined in terms of policies that capture an administrator's application-level knowledge. The expert system of our anomaly detection system is built using CLIPS [28] and the application-level knowledge is represented as facts and rules. These facts and rules are used to examine the information contained in the snapshot and detect anomalies.

When an anomaly is discovered, our system is be able to take appropriate measures to restore an affected virtual appliance to an acceptable state. The recovery component of our system relies on the administrator's application-level knowledge, expert systems, and loadable Linux kernel modules. The expert system uses the application-level knowledge to decide on appropriate recovery strategy for each anomaly that is detected. Recovery actions are then carried out by a collection of repair tools that are implemented as loadable Linux kernel modules.

The rationale behind using guest kernel-based tools is that it makes it much easier to instate certain repairs from within the kernel that would be nontrivial tasks using VMI. For example, to start a user space application, it would be much easier use the usermode-helper APIs provided by the Linux kernel than using VMI libraries. We acknowledge the fact that the guest kernel-based tools could themselves be targeted by malicious programs. Protecting the guest kernel-based tools is a significant research undertaking by itself, which we consider to be beyond the scope of this thesis.

1.4 Thesis Statement

Our thesis is: an application-level policy expressed over kernel-level application state can be effective for automatically detecting and mitigating the effects of malicious software in virtual appliances.

We designed and developed a policy-driven anomaly detection and recovery system to test our thesis. This system uses: (i) Stackdb libraries to monitor the state of the virtual appliance; (ii) a policy defined by the administrator that represents his or her application-level knowledge; (iii) an expert system to detect anomalies and reason about appropriate recovery strategy; and (iv) recovery tools implemented as loadable kernel modules to repair the affected virtual appliance.

We evaluated the effectiveness of our policy-driven anomaly detection and expert system against different types of malicious software: (i) Darkleech, an application-level malware; (ii) Azazel, a user-space rootkit; (iii) Suterusu, a kernel rootkit; and (iv) an exploit kit for CVE-2014-038. Our policy-driven anomaly detection and recovery system was successfully able to detect and mitigate the actions of all of these malicious software packages.

1.5 Contributions

The contributions of this thesis work are:

- We have designed a policy-driven anomaly detection and recovery system targeted at virtual appliances.
- We have developed a prototype system and demonstrated its utility to monitor a virtual appliance hosting an Apache web server.

- We have demonstrated the effectiveness of our system in identifying and mitigating real-world malicious software such as Darkleech, Azazel, Suterusu, and an exploit kit for CVE-2014-0038.
- We have explored the use of virtual machine introspection technologies to monitor virtual appliances.
- We have explored the use of an expert system to detect anomalies and reason about appropriate recovery strategies.
- We have explored the use of loadable Linux kernel modules as repair tools.

CHAPTER 2

BACKGROUND AND RELATED WORK

The policy-driven anomaly detection and recovery system we have designed relies on technologies such as virtual machine introspection, expert systems, and kernel-based state management. The following sections give a brief introduction to intrusion detection systems, virtual machine introspection, expert systems, and other work in areas related to this thesis.

2.1 Intrusion Detection Systems

Intrusion detection is the process of monitoring a system to identify any abnormal activities that can compromise the integrity of the system [29]. These abnormal activities may be caused by malicious software, unauthorized system access, or authorized users trying to exploit their privileges to gain higher privileges. An intrusion detection system (IDS) is software that is responsible for automatically detecting an intrusion. When an intrusion is detected, the intrusion detection system reports the event to the system administrator, who may initiate some recovery action to minimize the damage caused by the attack.

Intrusion detection systems can be broadly be classified into three types based on the layer at which they operate: (i) host-based, (ii) network-based, and (iii) virtual machine monitor (VMM) based [31]. Host-based intrusion detection systems run on the system being monitored. These intrusion detection systems monitor the activities of a single host to detect malicious events. Since a host-based intrusion detection system runs in the target system itself, it has a complete view of all the system events. This view helps the intrusion detection system detect a wide range of malicious activity, but at the same time, host-based intrusion detection systems have less isolation from an attacker and risk being compromised by an attack. Network-based intrusion detection systems monitor the target system at the network level. They are usually deployed at network entry or exit points and have full visibility of incoming or outgoing network traffic. These intrusion detection systems can have the ability to monitor multiple computer systems on the network or the entire network itself. Compared to host-based intrusion detection systems, a network-based intrusion detection system has lower visibility of the internal state of the systems it monitors. Hence it may be limited in its ability to monitor a wide range of events. Network-based intrusion detection systems have the advantage of better isolation from an attacker. A VMM-based intrusion detection system operates on the same physical machine as the virtual machine it monitors, but is separated from the virtual machine by the VMM. The VMM allows the intrusion detection system to operate in a separate hardware protection domain, thus providing better isolation from an attacker. The VMM also provides the intrusion detection system with access to the hardware and software state of the virtual machine being monitored. Thus, a VMM-based intrusion detection system can have the advantages of both host-based and network-based intrusion detection systems.

Intrusion detection systems can also be classified into two types based on the approach used to detect intrusions: (i) misuse detection and (ii) anomaly detection [31]. In the misuse-detection approach, the abnormal system behavior is first defined, and then any other behavior is categorized as normal system behavior. Thus, intrusion detection systems based on misuse detection require predefined patterns or signatures of malicious events. An important advantage of the misuse-detection approach is that it allows intrusion detection systems to be very effective at detecting attacks whose signatures are known without having high false-alarm rates. The obvious drawback of the misuse-detection approach is the need to have predefined attack signatures. This also makes it difficult for intrusion detection systems based on misuse detection to detect new attacks.

The anomaly-detection approach requires the normal behavior (baseline) of the system to be first defined, and then any other behavior of the system is categorized as an anomaly. Thus anomaly detection relies on the baseline information that is collected over a period of normal operation. An important advantage of this approach is its ability to detect novel attacks that exploit previously unknown vulnerabilities in a system. Since the expected behavior of the system is not expected to change often, constant updating of the baseline information is not required. Since the system depends on the baseline information, it would categorize any small deviation as an anomaly, resulting in the disadvantage of high false positive rate.

The system described in this thesis is an IDS paired with automatic repair capabilities. Our system is VMM-based and implements anomaly detection.

2.2 Virtual Machine Introspection

Virtual machine introspection (VMI) is a technique for externally monitoring the state of a virtual machine at run time. Using VMI, the memory of one virtual machine, called the target, can be viewed from another privileged virtual machine. Thus, VMI allows monitoring and controlling the target from an isolated and protected location. Virtual machine introspection allows:

- Reading and writing data from and to memory.
- Accessing memory using physical address, virtual address or kernel symbols.

- Translating kernel symbols to virtual addresses or translating virtual addresses to physical addresses.
- Controlling the execution state of the target virtual machine.

The biggest challenge for VMI is the semantic gap [6] between the monitoring and the target virtual machines. The semantic gap arises mainly because of the independent design of the target virtual machine and the privileged virtual machine used for introspection. The VMM and the privileged VM can observe the low-level operations of the target but generally lack knowledge of OS-level semantics of the target.

XenAccess [26] and LibVMI [3] are examples of powerful application programming interfaces that facilitate VMI. LibVMI is an evolved version of XenAccess with additional support for 64-bit guest operating systems and the KVM platform [21].

Stackdb [17] is a VMI-based debugging library that supports the analysis of software systems at multiple levels of the software stack. The library allows users to write programs that can analyze live, whole-system executions. Programs can use the library to pause, single-step, and resume target executions. It also allows querying of symbol data, modifying memory and CPU state, and inserting breakpoints and watchpoints.

The system described in this thesis uses VMI to inspect targets. It is implemented using Stackdb, which has features that allow our system to overcome the semantic gap between the target's OS and our monitoring and repair system.

2.3 Expert Systems

Expert systems are artificial-intelligence programs that use human-like logic for problem solving [12]. An expert system consists of a knowledge store and an inference engine. The knowledge store is a collection of information known as "facts." The inference engine uses facts to make intelligent decisions.

Expert systems are generally built using the rule-based programming paradigm. In this paradigm, program logic is expressed using a collection of "rules." These rules resemble the *if* conditional construct of C programming language A rule consists of an *if* part and a *then* part. If all the conditions in the *if* part evaluate to true, the instructions specified in the *then* part are executed. This evaluation and execution process of the inference engine is carried out every time a new fact is entered into the knowledge store.

C Language Integrated Production System (CLIPS) is a tool for building an expert system. We use CLIPS to implement an expert system that can detect anomalies in the target system and reason about appropriate recovery strategies.

2.4 Related Work

The major areas of work related to this thesis are kernel integrity measurement and monitoring, virtual machine introspection, and automatic data structure repair. The following subsections describe prior work in these areas.

2.4.1 Integrity Measurement and Monitoring

Linux kernel integrity measurement (LKIM) [23] is a tool that uses contextual inspection for measuring the integrity of the Linux kernel. Contextual inspection is a technique that uses layout information of kernel data structures to examine all data structures associated with running processes. It produces detailed records about the state of those structures within the kernel. Records are generated during boot time, during major system events, and also on demand. These records are compared to a baseline measurement to verify the integrity of the kernel. LKIM also cryptographically hashes the static code and data in the Linux kernel so that their integrity can be verified.

The authors of LKIM modified the Linux kernel so that it notifies LKIM during module loading. This was needed to support the integrity measurement of loadable Linux kernel modules.

LKIM executes in a separate virtual machine and it uses the memory mapping functionality of Xen [5] to access the memory of the target virtual machine. This design isolates LKIM from the possibly affected target virtual machine.

The goal of LKIM is to measure and attest the dynamic data structures within the Linux kernel. The system administrator can use this to detect rootkits and malware that affect the integrity of the system. The system we have designed, on the other hand, tries to maintain the target virtual appliance in an acceptable state. We use virtual machine introspection to monitor the target system and kernel-based state management techniques to restore a compromised target to an acceptable state.

OSck [14] is another system designed to ensure operating system kernel integrity. It builds on hypervisor-based monitoring techniques to detect rootkits based on the violation of operating system invariants. OSck verifies control-flow integrity to detect rootkits that operate by modifying control flow. Static and persistent control transfers are made immutable by write-protecting kernel text, read-only data, and the values of specific machine registers. Dynamic control transfers are protected by making sure the function call target is known and the type signatures match. The integrity of heap data structures is verified by making sure that any dereferences of pointers point to safe functions. OSck also provides APIs to verify the integrity of noncontrol-flow data.

Unlike OSck, the research described in this thesis does not rely on write-protecting static data or on kernel invariants. Our system uses policies that capture the expected state of the system at the application level. These policies define invariants at the application level, which indirectly reflect an expected state of the kernel in the target virtual appliance.

2.4.2 Virtual Machine Introspection for Intrusion Detection Systems

Several intrusion detection systems have used virtual machine introspection to monitor virtual machines. These include Livewire, IntroVirt, and Lycosid.

The Livewire [11] IDS consists of three main components: a VMM interface, an OS interface library, and a policy engine. The VMM interface sends management, inspection, and monitoring commands to the virtual machine monitor. The OS interface library interprets the guest OS state based on OS-specific knowledge. The policy engine consists of a policy framework that allows interactions with other components and policy modules that implement the security policies. Livewire is capable of detecting hidden process and modules, scanning file systems for signatures to detect malicious programs, detecting the presence of raw sockets, and detecting NICs entering promiscuous mode. It is highly effective because it uses prior knowledge of the system state and reliable state measurements to determine the current system software state.

IntroVirt [20] is designed for checking if a vulnerability was exploited in the past before it was publicly known and also preventing the vulnerability from being exploited until the system is patched. InroVirt uses vulnerability-specific predicates to test the state of the target system. These predicates are executed outside the target virtual machine. An important aspect of IntroVirt is that it makes sure that the target system is not perturbed during the execution of the predicates. IntroVirt uses virtual machine introspection to examine the state of the operating system or application running in the virtual machine. The semantic gap between the virtual machine introspection abstractions and the application or operating system abstraction is bridged by reusing the code present within the application or operating system itself. IntroVirt uses vulnerability-specific predicates and the replay functionality of ReVirt [9] to determine if the vulnerability was exploited in the past. Further, by using the predicates with some response strategy, IntroVirt can detect attempts to exploit the vulnerability in the future until the operating system or application is patched.

Lycosid [19] is a VMM-based system that can detect hidden processes. Lycosid is based on a cross-view validation technique. This technique detects inconsistencies between objects by looking at multiple views of the same object and noting any difference between them. Lycosid detects hidden processes by comparing an untrusted view of the process data-structure list with a trusted view of that list. Lycosid passively generates a trusted view of the system by using measurable quantities associated with processes. An untrusted view is generated from the information returned by OS utilities. Lycosid is different from other systems because it does not directly rely on low-level OS-specific details. Because Lycosid does not depend on the consistency of OS data structures, hidden process detection cannot be avoided by common evasion techniques. Further, since it is not dependent on the implementation of any OS-level abstractions, it is not tied to any particular platform. Lycosid consists of detection and identification phases. Hidden processes are detected by comparing the lengths of the process lists from the trusted and untrusted views. Any difference in views indicates the existence of a hidden process. To identify the hidden process, Lycosid relies on measurable quantities associated with the processes. Lycosid uses Antfarm [18] to get the execution times of each process. These execution times are used to identify the hidden process.

The IDS presented in this thesis is complementary to previous work on VMI-based intrusion detection systems. Unlike IntroVirt or Lycosid, our work concentrates on ensuring an acceptable state of the virtual appliance. In addition to anomaly detection, our system also uses kernel-based state management to restore the affected virtual appliance to an acceptable state. Further, we do not rely on predicates for kernel data structures or cross-view validation to detect intrusions or to validate the execution environment. Our system uses application-level policies to define an expected state of the system. These application-level policies indirectly reflect an expected state of the kernel in the target virtual appliance. Our anomaly detection and recovery system can identify and recover from a wider range of problems that include intrusions, unfavorable executions, and unfavorable system configurations.

2.4.3 Automatic Data Structure Repair

Exterior [10] and the work by Brian Demsky and Martin Rinard [8] are examples of systems that are capable of data structure repair similar to the work done in this thesis. Exterior is a dual virtual machine system that can be used for introspection, configuration, and recovery. Exterior uses a secure virtual machine that runs the same operating system as the monitored target virtual machine which Exterior calls the "guest" VM. This dual VM architecture defines a new program execution model: secure code from a trusted virtual machine is used to operate on the data in the guest virtual machine.

The three main components of Exterior are kernel system call identification, kernel data identification, and guest virtual machine (GVM) memory mapping and address resolution. These three components work together to produce the effect of executing programs in a guest virtual machine while using trusted code from a secure virtual machine. Thus the OS utilities in the guest virtual machine can be used in a reliable way to act as introspection, configuration, and recovery tools. Exterior provides automation that uses the OS utilities and detects intrusions through cross-view validation. Exterior also provides a repair tool called "MakeUp" that runs in the secure virtual machine and is capable of fixing malicious entries in the system call table. This tools demonstrated the feasibility of repair tools that are capable of fixing the guest virtual machine.

Brian Demsky and Martin Rinard developed a system for automatic detection and repair of data structures [8]. It presents an approach to ensure application data structure consistency, thereby enabling continued execution of applications in case of errors. This system had two main components:

a specification language and an inconsistency detection and repair component. The specification language was used to ensure consistency of data structures used by the application. The consistency-checking algorithm evaluated all the application data structures against the stated specifications. If any discrepancy was detected, then a repair action would be initiated that would fix the data structures according to defined specifications. Thus, even though they were not able to restore the data structures to the original state, they were able to recover data structures to a state that satisfies the consistency definitions. An important feature of this system is that it provided the ability to enforce data structure repair on the fly, thereby eliminating the downtime caused by system crashes. Since this system was designed to mitigate the problems caused by bugs in applications, it could not handle data structure inconsistencies intentionally injected by rootkits and malware.

As mentioned in earlier sections, our system relies on application-level knowledge encoded as policies. Any deviation from these policies indicates an anomaly and an appropriate repair action can be triggered to recover the system to an acceptable state. In contrast to the work by Brian Demsky et al., since our application-level policies do not define application or kernel data structures, our repair does not prevent application misbehavior resulting from bugs in the application. For example, our recovery tools cannot prevent application crashes caused by `NULL` pointer dereferencing, but are capable of restarting crashed applications. Exterior can be considered to be similar to our work; both have the same end objectives but use different approaches. Unlike Exterior, our introspection and recovery actions do not rely on trusted code from another virtual machine, but instead makes use of application-level knowledge and OS-level abstractions. This gives us the flexibility to develop customized tools and not be restricted by the utilities provided by the operating system.

2.4.4 Summary

Our anomaly detection and recovery system differs from previous work in integrity monitoring, intrusion detection, and repair systems in the following aspects.

- It does not rely on a specification language or predicates for kernel data structures, cross-view validation, or contextual inspection, but instead is driven by application-level policies specifically defined for virtual appliances.
- Policies are not only used to detect and recover from intrusions but also help to ensure legitimate use of computing resources.
- Validating the conditions specified by the policies does not rely on an operating system's inbuilt utilities. This eliminates the possibility of execution of compromised system code providing false data.
- Our system uses both guest kernel-based and VMI-based tools to instantiate recovery to achieve better scope and flexibility.

CHAPTER 3

DESIGN

To test the feasibility of our thesis, we designed and implemented a system that uses application-level knowledge to detect anomalies in a virtual appliance and to restore the virtual appliance to an acceptable state. The anomaly detection system is VMM-based and it uses kernel-based state management techniques.

A high-level overview of our anomaly detection and recovery system is shown in Figure 3.1. The three main design components of this system are (i) the state-gathering component, (ii) the anomaly-detection component, and (iii) the recovery component. The state-gathering component consists of a snapshot engine that captures point-in-time snapshots of the virtual appliance. The anomaly-detection component consists of an application-level knowledge store and a policy engine. The policy engine uses the information in the snapshots and the knowledge store to detect anomalies. The recovery component consists of a repair engine, a repair driver, and a set of repair tools. The repair engine uses the information in the knowledge store to reason about an appropriate repair strategy and conveys it to the repair driver. The repair driver invokes an appropriate repair tool to carry out the repair as directed. The above-mentioned components collectively operate to periodically monitor the virtual appliance, detect anomalies, and recover from them. Furthermore, these components are designed in a modular fashion so that one can be enhanced with little or no modification to another.

3.1 State Gathering Component

An anomaly detection system should have complete view of the state of the system it monitors. The system being monitored is called the "target." Once the anomaly detection system has access to the target system state, it can validate it to detect anomalies.

The state gathering component is responsible for providing the anomaly detection system with the state of the target system. The state gathering component we have designed consists of a snapshot engine. This snapshot engine is a VMI application that executes in the user space of a control virtual machine. The snapshot engine executes at regular intervals of time and captures snapshots of the state of the target virtual appliance. It makes use of VMI libraries to probe and read values stored in the

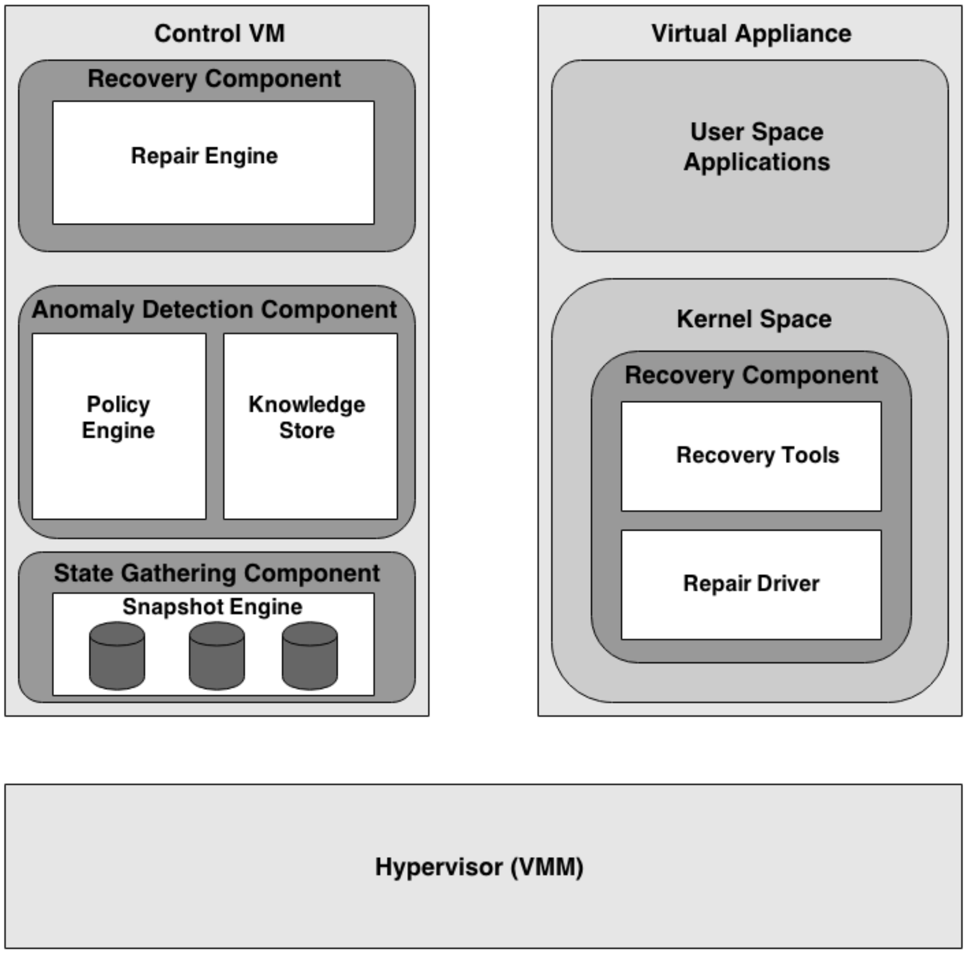


Figure 3.1. High Level Design

kernel data structures of the target virtual appliance.

For each process executing in the target virtual appliance, the snapshot contains information about the process credentials, the process priorities, the environment for the process, the CPU utilization of the process, the files opened by the process, the network sockets opened by the process, and the objects loaded for that process. In addition, for the virtual appliance as a whole, the snapshot engine finds information about the CPU load, the system call table, and the loaded kernel modules.

This state information is encoded as a collection of "base facts." Listing 3.1 is an example of a base fact that represents an executing process in the virtual appliance.

This fact indicates that a process `mysqld` is executing in the virtual appliance. It also captures other information about the process such as its credentials, priorities, and its parent process.

The state information gathered by the snapshot engine is reliable because the snapshot engine operates outside of the target virtual appliance. State information could also be gathered from OS utilities that run within the target, but there is always a chance of the target's OS code being compromised, which could result in an inaccurate view of the state of the VA. For example, a malicious process may prevent itself from being listed by the `ps` utility by hooking methods in the `/proc` file system. Therefore, we use VMI-based tools to gather the state information that is required by the anomaly-detection component.

Listing 3.1: Base Fact

```
(task-struct
   (comm "mysqld")        ; name of the process
   (pid 663)              ; process ID
   (tgid 663)             ; thread group ID
   (is_vcpu 0)            ; process is a vcpu
   (is_wq_worker 0)       ; work queue worker thread
   (used_superpriv 1)     ; process has super-user privileges
   (is_kswapd 0)          ; process is a kswap daemon
   (is_kthread 0)         ; kernel thread
   (prio 120)             ; process priority
   (static_prio 120)      ; static priority
   (normal_prio 120)      ; normal priority
   (rt_priority 0)        ; real-time priority
   (nice 0)               ; process nice value
   (uid 108)              ; real user ID
   (euid 108)             ; effective user ID
   (suid 108)             ; saved user ID
   (fsuid 108)            ; UID for VFS
   (gid 120)              ; real group ID
   (egid 120)             ; effective group ID
   (sgid 120)             ; saved group ID
   (fsgid 120)            ; GID for VFS
   (parent_pid 1)         ; parent process ID
   (parent_name "init")   ; parent process name
)
```

3.2 Anomaly Detection Component

The anomaly detection component is the brain of the system. It responsible for detecting anomalies in the target virtual appliance. The design of the anomaly detection component is shown in Figure 3.2. It consists of a knowledge store and a policy engine. It takes as input the base facts captured by the snapshot engine. It generates a new set of facts that represent anomalies in the target virtual appliance.

The knowledge store contains information provided by the administrator of the virtual appliance. This information reflects the administrator's application-level knowledge about the virtual appliance. It contains two types of information: application-level facts that are necessary to define the execution environment, and a set of rules that are used to validate the execution environment. The application-level facts represent the expected state of the target virtual appliance in terms of kernel-visible abstractions such as processes, files, system users, loaded objects, kernel modules, and sockets. For example, one could have a fact that identifies all the high-priority processes of an execution environment. One could also define a rule that states that when CPU utilization is beyond a certain threshold value, only high-priority process should be allowed to execute. In a web server execution environment, one can have a rule that only allows Apache, MySQL, and other dependent

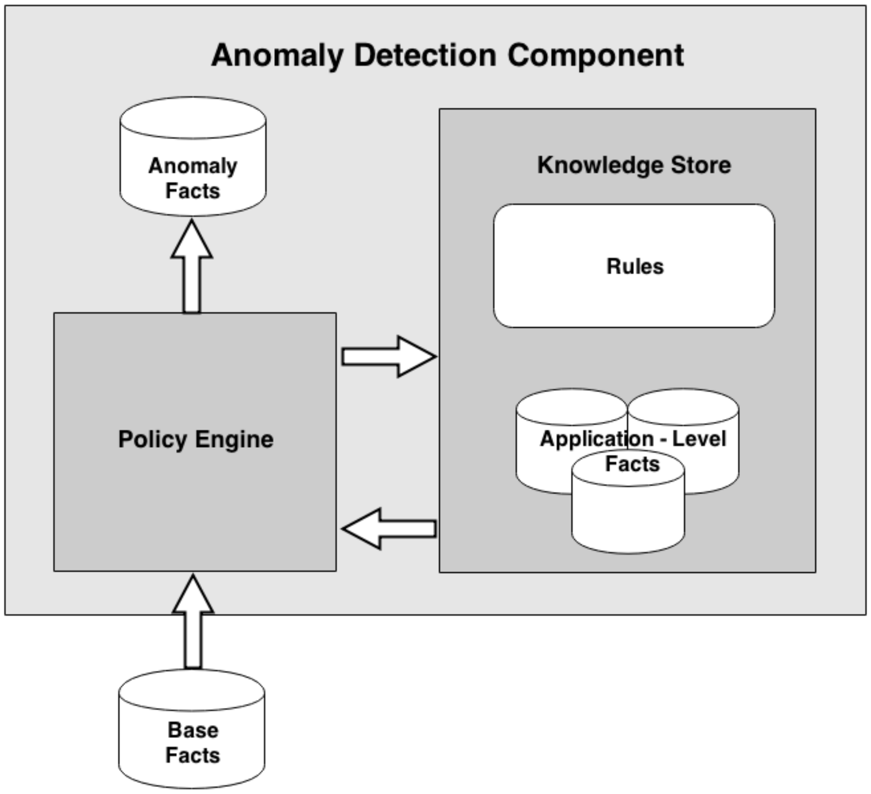


Figure 3.2. Anomaly Detection Component

Linux process to be running. Any other running processes found to be running would be considered as a policy violation. By means of a comprehensive set of facts and rules, the creator or deployer of a VA can build a knowledge store that represents the administrator's view of the expected system behavior. For example, Listing 3.2 is a fact that captures the administrator's knowledge of a process expected to be executing on a web-server virtual appliance. For each process expected to be executing in the VA, one can have facts that capture information about the process credentials, process lineage, loaded objects, and other related information.

The information that is represented as facts includes:

- Processes that are expected to be executing in the virtual appliance.
- Credentials for each process (uid, gid, fsgid).
- Process hierarchies, i.e., parent-child relationships.
- Processes allowed to have network connectivity.
- Processes permitted file IO.
- Valid objects for each process.
- Linux kernel modules that are allowed to be loaded.
- Valid state of the system call table.

The policy engine is an expert system that executes in the user space of the control virtual machine along with the snapshot engine, isolated from the target virtual appliance. The policy engine uses information from the knowledge store and the base facts captured by the system snapshot as input. It outputs a new set of facts called "anomaly facts."

The rules in the knowledge store use the application-level facts and the base facts gathered by the snapshot engine to validate the state of the system. These rules capture an administrator's domain expertise in anomaly detection. During each iteration of the system, the policy engine takes as input the system snapshot and information from the knowledge store. It passes all the base facts through the rules looking for all possible matches. Each of these matches results in a new anomaly fact being asserted. Some of the anomalies detected by these rules are:

- Unknown executing process.
- Process with wrong credentials.
- Root shell.
- Invalid process hierarchy, i.e., wrong parent-child relationship.
- Unauthorized network access.
- Unauthorized file IO.
- Corrupted system call table entry.
- Hooked system call.

Listing 3.2: Application-Level Fact

```
(deffacts application-facts
   (known-process
      (name "apache2")
      (uid 33)
      (euid 33)
      (suid 33)
      (fsuid 33)
      (gid 33)
      (egid 33)
      (sgid 33)
      (fsgid 33)
      (parent_name "init" "apache2")
      (object-list "httpd" "libnss_files-2.15.so"
      "libnss_nis-2.15.so" "libnsl-2.15.so"
      "libnss_compat-2.15.so" "libm-2.15.so" "mod_uni_mem.so"
      "mod_rewrite.so" "mod_alias.so" "mod_userdir.so"
      "mod_speling.so" "mod_actions.so" "mod_imagemap.so"
      "mod_dir.so" "mod_negotiation.so" "mod_vhost_alias.so"
      "mod_dav_fs.so" "mod_cgi.so" "mod_info.so"
      "mod_asis.so" "mod_autoindex.so" "mod_status.so"
      "mod_dav.so" "mod_mime.so" "mod_proxy_balancer.so"
      "mod_proxy_ajp.so" "mod_proxy_scgi.so" "mod_dumpio.so"
      "mod_proxy_http.so" "mod_proxy_ftp.so"
      "mod_proxy_connect.so" "mod_proxy.so" "mod_version.so"
      "mod_setenvif.so" "mod_unique_id.so" "mod_dbd.so"
      "mod_usertrack.so" "mod_ident.so" "mod_headers.so"
      "mod_expires.so" "mod_cern_meta.so" "mod_env.so"
      "mod_mime_magic.so" "mod_logio.so" "mod_log_forensic.so"
      "mod_log_config.so" "libz.so.1.2.3.4" "mod_deflate.so"
      "mod_substitute.so" "mod_filter.so" "mod_include.so"
      "mod_ext_filter.so" "mod_reqtimeout.so" "libdl-2.15.so"
      "mod_auth_digest.so" "mod_auth_basic.so"
      "mod_authz_default.so" "mod_authz_owner.so"
      "mod_authz_dbm.so" "mod_authz_user.so" "libc-2.15.so"
      "mod_authz_groupfile.so" "mod_authz_host.so"
      "mod_authn_default.so" "mod_authn_dbd.so" "ld-2.15.so"
      "mod_authn_anon.so" "mod_authn_dbm.so" "mod_authn_file.so"
      "libcrypt-2.15.so" "libexpat.so.0.5.0" "libapr-1.so.0.4.5"
      "libapr-1.so.0.4.5" "libaprutil-1.so.0.4.1")
   )
)
```

Listing 3.3 is an example of a rule that identifies all the unknown processes running in the virtual appliance. The lines 2 and 3 are the left-hand side (LHS) of the rule, also called the *if* section, and lines 5 and 6 are the right-hand side (RHS) of the rule, called the *then* section. When all of the conditions on the LHS of a rule are satisfied, the RHS of the rule is executed. The rule `identify-unknown-process` matches all the `task-struct` base facts with the `known-process` application-level fact. If any process represented by the `task-struct` is not present as a `knownprocess` fact, then it is identified as an anomaly and asserted as an `unknown-process` anomaly fact.

The administrator of the VA is expected to define the application-level facts. These facts are specific to applications hosted on a VA and are reusable for the same type of VAs. The rules in the knowledge store are application-agnostic and are reusable for all VAs.

3.3 Recovery Component

Once an anomaly has been identified by the policy engine, the recovery component is responsible for reasoning about an appropriate recovery strategy and restoring the virtual appliance to an acceptable state. The design diagram for the recovery component is shown in Figure 3.3. The recovery component consists of a repair engine, a repair driver, and set of recovery tools. The repair engine is an expert system that executes in the control virtual machine along with the snapshot engine and policy engine. It takes as input the anomaly facts generated by the policy engine and passes these facts through a set of recovery rules. The repair engine outputs a set of “recovery facts” that are used to drive the repair driver and eventually the repair tools.

Listing 3.3: Rule

```
1 (defrule identify-unknown-process
2    (task-struct (comm ?name1) (pid ?pid))
3    (not (exists (known-process (name ?name1))))
4    =>
5    (assert (unknown-process (name ?name1) (pid ?pid)))
6    (printout t "ANOMALY: Unknown process " ?name1 " found" crlf)
7 )
```

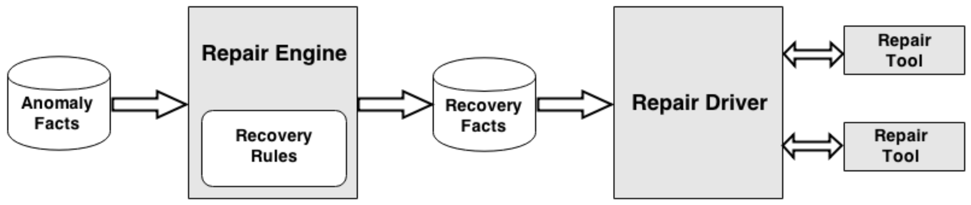


Figure 3.3. Recovery Component

The recovery rules do not just provide one-to-one mappings of anomalies to repairs. Instead, they consider other conditions, such as the history of previous occurrences of the anomaly and the recovery actions already taken, before suggesting the next repair strategy. To be able to provide reasoning, the repair engine maintains information about the recovery component's previous actions. This information is stored as facts. Thus the policy engine and repair engine together provide a mapping between the administrator's application-level knowledge, encoded in the knowledge store, and an appropriate kernel-based repair strategy.

The recovery facts generated by the repair engine identify individual repair tools and their input parameters. Recovery facts are interpreted as invocation commands for the repair tools. Commands are communicated to the repair driver through an interdomain communication channel, which in turn invokes an appropriate repair tool. Listings 3.4, 3.5, and 3.6 show examples of recovery rules to deal with unknown processes executing in the virtual appliance. These rules apply to `unknown-process` facts and map them to appropriate recovery actions. The three rules `kill-unknown-process`, `kill-unknown-process_1`, and `kill-unknown-process_2` deal with recovering from unknown processes found executing in the virtual appliance. The rule `kill-unknown-process`, shown in Listing 3.4, deals with unknown processes found executing in the virtual appliance for the first time. If any unknown process is found, the recovery action is to kill the process. The rule `kill-unknown-processes_1`, shown in Listing 3.5, deals with unknown processes found executing in the virtual appliance for a second time. In this case, the recovery action is to kill the process as well as its parent. The rule `kill-unknown-process_2`, shown in Listing 3.6, deals with unknown processes found executing in the virtual appliance for the third time. In this case, the administrator is notified and the entire recovery strategy is repeated.

Once an appropriate recovery action is identified, the repair driver is responsible for carrying out that repair using the repair tools. The repair-tool invocation commands are received by the repair driver through an interdomain communication channel. The repair driver and repair tools are implemented as loadable kernel modules that execute in the target VA. On receiving an invocation command, the repair driver loads and executes the repair tool as directed by the repair engine. The result of the execution of the tool is communicated back to the repair engine from the repair driver through the interdomain communication channel.

Repair tools are responsible for carrying out repair tasks as directed by the repair engine. The functionality provided by the repair tools includes: (i) terminating processes, (ii) starting processes, (iii) terminating network sockets, (iv) closing open files, (v) correcting process credentials, (vi) sledding the text section of processes, (vii) preventing the loading of objects, (viii) correcting function pointers in system call table, and (ix) unhooking system calls. Additional details regarding the implementation of these tools are provided in Section 4.3. These repair tools provide a generic

Listing 3.4: `kill-unknown-process` Recovery Rule

```
(defrule kill-unknown-process
   (declare (salience 10))
   ?f <- (unknown-process (name ?name1) (pid ?pid))
   (not (exists (unknown-process-recovery-prev-action
                  (prev_action ps_kill | ps_kill_parent)
                  (name ?name1))))
   =>
   (assert (recovery-action
            (function-name kill_process) (arg_list ?name1 ?pid)))
   (assert (unknown-process-recovery-prev-action
            (name ?name1) (prev_action ps_kill)))
   (retract ?f)
   (save-facts "recovery_action.fac" visible recovery-action)
   (save-facts "process_state_info.fac" visible
               unknown-process-recovery-prev-action)
   (printout t "RECOVERY: Killing the unknown process" ?pid crlf)
)
```

Listing 3.5: `kill-unknown-process_1` Recovery Rule

```
(defrule kill-unknown-process_1
   (declare (salience 20))
   ?f <- (unknown-process (name ?name2) (pid ?pid1))
   ?of <- (unknown-process-recovery-prev-action
            (prev_action ps_kill) (name ?name2))
   =>
   (assert (recovery-action
            (function-name kill_parent_process)
            (arg_list ?pid1 ?name2)))
   (retract ?f)
   (retract ?of)
   (assert (unknown-process-recovery-prev-action
            (prev_action ps_kill_parent) (name ?name2)))
   (save-facts "recovery_action.fac" visible recovery-action)
   (save-facts "process_state_info.fac" visible
               unknown-process-recovery-prev-action)
   (printout t "RECOVERY: Killing the process and parent process" crlf)
)
```

Listing 3.6: `kill-unknown-process_2` Recovery Rule

```
(defrule kill-unknown-process_2
   (declare (salience 30))
   ?f <- (unknown-process (name ?name2) (pid ?pid1))
   ?of <- (unknown-process-recovery-prev-action
            (prev_action ps_kill_parent) (name ?name2))
   =>
   (retract ?of)
   (retract ?f)
   (save-facts "process_state_info.fac" visible
               unknown-process-recovery-prev-action)
   (printout t "RECOVERY: Tried killing the unknown
            process and its parent, but the process
            still exists! Repeating the recovery
            cycle now. " crlf)
)
```

framework for recovering from malware and rootkit attacks and restoring the virtual appliance to an acceptable state.

3.4 Operation

The overall control flow within the components of the anomaly detection and recovery system is shown in Figure 3.4. The initialization phase depicts startup of the anomaly detection and recovery system. During this phase, the snapshot engine captures an initial snapshot of the system. It is expected that the virtual appliance was deployed around that time and is believed to be in a reliable state.

The repetition phase represents the control flow during the periodic execution of the anomaly detection and repair system. During each iteration, the main function in the anomaly detection and recovery system first invokes the snapshot engine to capture a snapshot of the virtual appliance. This snapshotting operation requires the virtual appliance to be paused. The main function then transfers control to the policy engine. The policy engine uses the information present in the snapshot and the knowledge store to detect anomalies in the target virtual appliance. If any anomalies are detected, they are represented as anomaly facts. The control is then transfered back to the main function. If anomalies are detected, the main function invokes the repair engine that uses anomaly facts and the recovery rules to generate a set of recovery facts. For each recovery fact generated, control is transfered to the repair driver, which in turn invokes the appropriate repair tool. The repair tool performs the repair action as directed by the repair engine. After completion, the repair tool sends an acknowledgment back to the repair driver, which in turn sends back an acknowledgment to the main program.

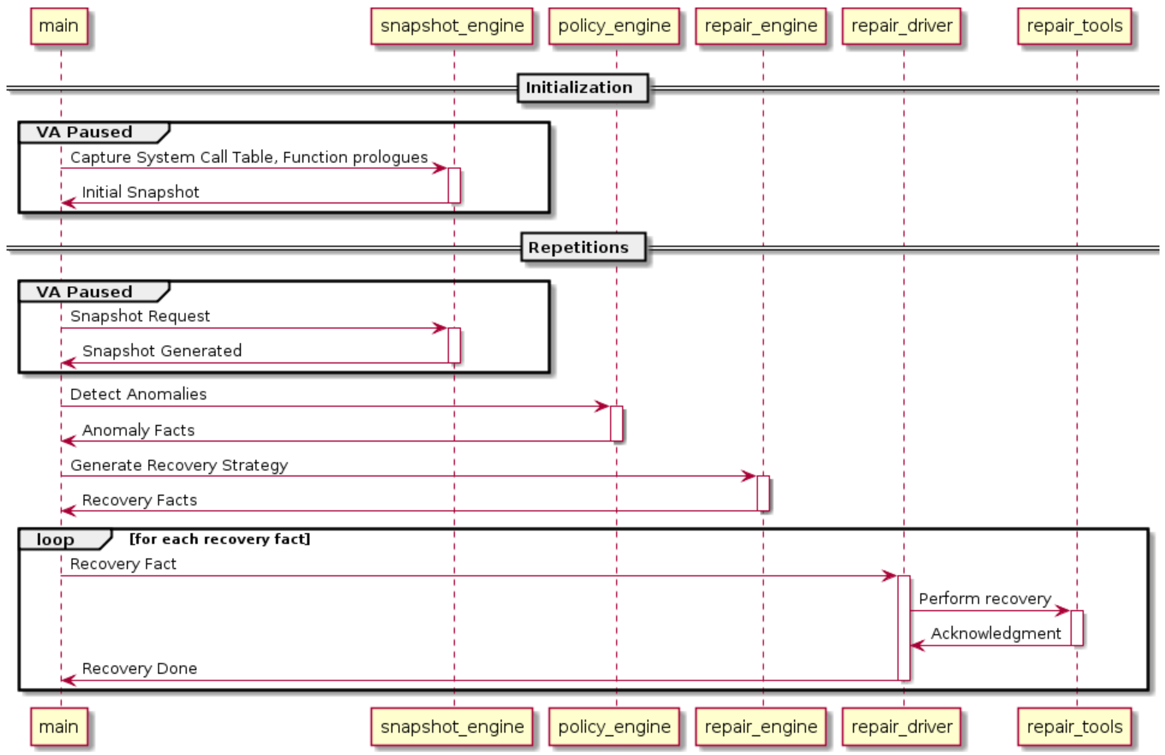


Figure 3.4. Sequence Diagram

CHAPTER 4

IMPLEMENTATION

We implemented the anomaly detection and recovery system for the Xen virtualization framework. The state-gathering component is built using the Stackdb [17] VMI libraries. The anomaly-detection component is implemented as an expert system built using the open source CLIPS [28] framework. The recovery component is implemented as a combination of an expert system and loadable Linux kernel modules. The communication channel between the expert system and the loadable Linux kernel modules is implemented using the Stackdb libraries. Figure 4.1 depicts the overall implementation of our anomaly detection and recovery system. The snapshot engine, policy engine, and repair engine execute within the protected dom0 (control VM). The repair driver and the repair tools execute within domU (target VA).

4.1 State-Gathering Component

The snapshot engine in the state-gathering component is a user-level application built using Stackdb. Stackdb libraries allow the snapshot engine to probe the kernel data structures of the target virtual appliance.

Figure 4.2 shows the working of the snapshot engine. To capture the value stored in a particular kernel data structure, the snapshot engine calls an appropriate Stackdb API. Stackdb reads the value stored in a data structure by making use of either the name or address of the data structure. The snapshot engine traverses the linked list of Linux `task_struct` structures to gather information about all the executing processes in the virtual appliance. Each `task_struct` contains information about a single process's credentials and priorities. Information about the files and network sockets opened by a process is captured from the Linux `file` data structures, which can be reached by traversing the file descriptor table of that particular process. The loaded-object list for a process is obtained by traversing the linked list of `vm_area` structures associated with the process. Similarly, the information about the CPU utilization, system call table, and loaded modules is captured by looking up the corresponding data structures from the kernel in the target virtual appliance.

The CLIPS facts representing the system state are written to plain-text files by the snapshot engine.

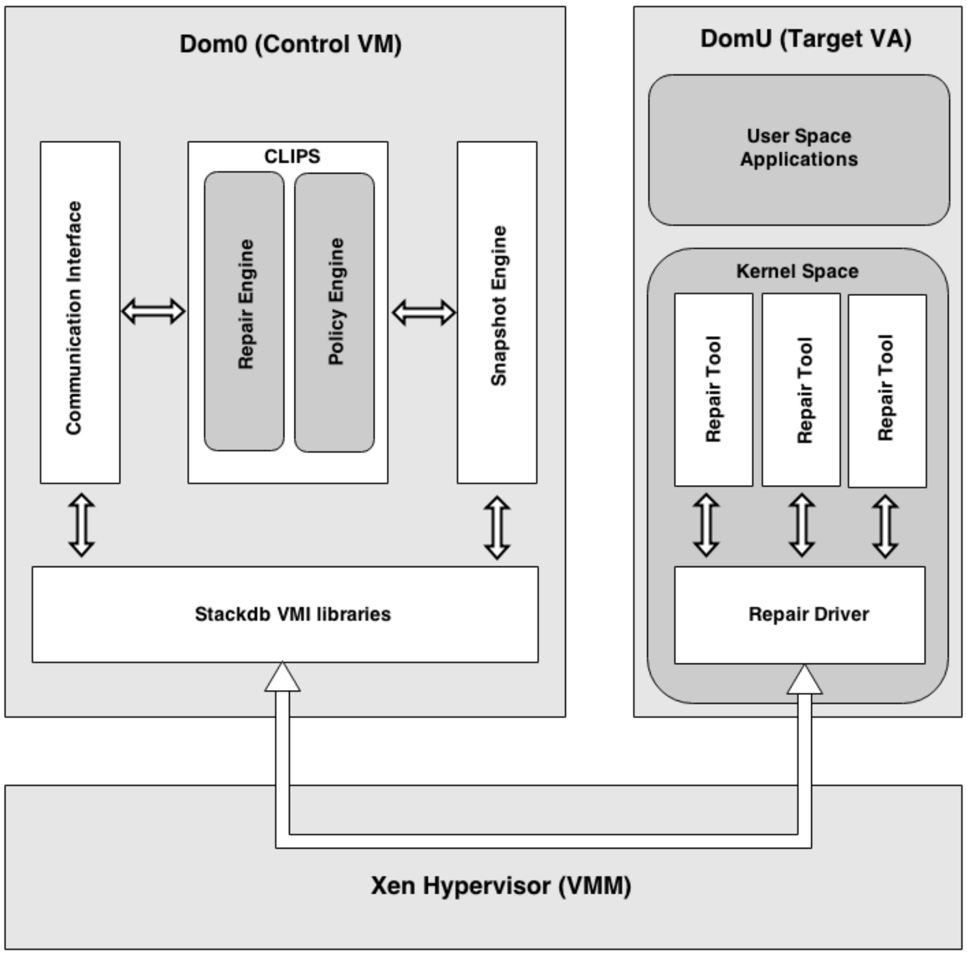


Figure 4.1. Implementation Diagram

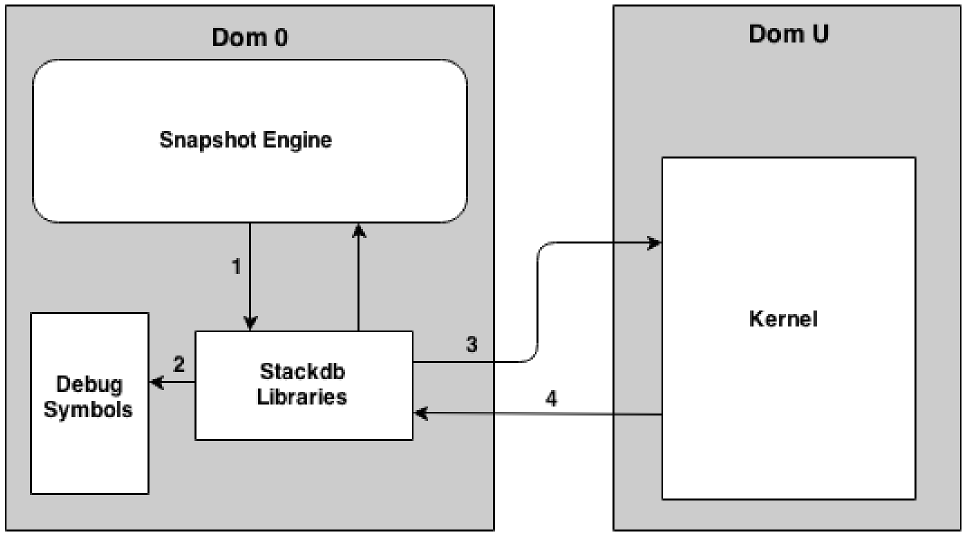


Figure 4.2. Snapshot Engine Internals

These files serve as input to the policy engine. These facts could also be directly loaded into the policy engine using the API provided by CLIPS. Using the CLIPS API directly might improve the performance of the snapshot engine, as the overhead of file IO would be eliminated. However, we chose to use files to store snapshots for ease of implementation and for logging purposes. The snapshot file also serves as a log that the administrator can later examine if need arises.

4.2 Anomaly Detection

The anomaly detection component consists of the knowledge store and the policy engine. This component is implemented as an expert system using CLIPS. The knowledge store consists of application-level facts and rules. The facts are application-specific and encode the administrator's application-level knowledge. The application-level facts are written to describe the expected state of the target virtual appliance. Table 4.1 lists the kind of facts that can be used to represent the administrator's expected view of the system. Currently these application-level facts in the knowledge store need to be manually coded. However, this step can automated so that the base facts are generated by probing the virtual appliance when it is known to be in a reliable state.

Table 4.2 lists the rules that are used by the policy engine to detect anomalies in the target VA. Our prototype implementation consist of about 300 lines of CLIPS anomaly detection rules. As described in the table, these rules validate various attributes and actions of all the processes running in the virtual appliance. These rules are CLIPS translation of an administrator's domain expertise in detecting anomalies in a virtual appliance. The rules are application-agnostic and can be reused for different types of virtual appliances. The most challenging part about implementing these rules is learning rule-based programming and effectively using the programming constructs that CLIPS provides.

4.3 Recovery Component

The recovery component consists of a repair engine, a repair driver, and repair tools. The repair engine is implemented as a CLIPS expert system that executes in the protected dom0. The repair engine has about 500 lines of CLIPS rules. It inputs the anomaly facts generated by the policy engine and outputs a set of recovery facts. A recovery fact contains the name of the repair tool to be invoked and the list of parameters to be passed to the tool. Apart from the recovery facts, the repair engine also generates some state information regarding each recovery fact asserted. The state information is represented as facts in the expert system. This state information is used in the subsequent iterations of the recovery component to make smart decisions based on the recovery actions taken in the previous iterations.

The recovery actions dictated by the repair engine are carried out by the repair driver and the repair

Table 4.1. Application-Level Facts

Fact	Description
`known-process`	Name of a process allowed to execute in the virtual appliance.
`known-process-cred`	Credentials for each of the processes running. This includes name, uid, euid, gid, egid, and name of the parent process.
`mandatory-process`	List of processes that are expected to be executing at all times.
`process-run-as-root`	List of processes allowed to have root privileges.
`known-objects`	List of objects that are allowed to be loaded for each process.
`known-open-files`	List of regular files that are known to be accessed by each process.
`process-with-udp`	UDP port numbers for a process allowed network connectivity.
`process-with-tcp`	TCP port numbers for a process allowed network connectivity.
`known-modules`	List of modules that are allowed to be loaded by the kernel in the virtual appliance.
`high-priority-processes`	List of high-priority processes.
`low-priority-processes`	List of low-priority processes.

Table 4.2. Application-Level Rules

Rule	Description
`identify-unknown-process`	Identifies all unauthorized process executing in the virtual appliance.
`identify-unknown-module`	Identifies all unauthorized modules loaded by the kernel.
`identify-open-udp-sockets`	Identifies any unauthorized UDP sockets opened by processes.
`identify-open-tcp-sockets`	Identifies any unauthorized TCP sockets opened by processes.
`identify-high-cpu-utilization`	Identifies all processes resulting in high CPU utilization.
`identify-privilege-escalation`	Scrutinizes process privileges for unauthorized escalations.
`identify-unknown-loaded-objects`	Scrutinizes process memory for unknown loaded objects.
`identify-unknown-open-files`	Makes sure only authorized process perform file IO. Also makes sure only permitted files are accessed.
`identify-wrong-process-hierarchies`	Scrutinizes process lineages.
`identify-missing-processes`	Identifies all mandatory processes that are not currently executing.
`identify-tampered-syscall`	A special rule for kernel integrity check, that scrutinizes function pointers in the system call table.
`identify-hooked-syscall`	A special rule for kernel integrity check, that detects system call hooking.

tools. The repair driver provides a communication channel between the repair engine and the repair tools. One end of this communication channel is implemented as a VMI application that executes in the protected dom0. It takes as input the recovery facts generated by the repair engine, parses them into a command structure and sends it over to the other end of the communication channel. The other end of the communication channel is the repair driver that executes within the target virtual appliance. The repair driver provides a ring buffer of command structures. The VMI application part of the repair engine writes commands into the ring buffer. The repair driver reads those commands and invokes the appropriate repair tools. The repair tools finally carry out the repair action as directed by the repair engine. These repair tools are implemented as loadable Linux kernel modules, one for each kind of repair.

The `psaction` module can terminate a process that is running in the virtual appliance. This module takes as input the process ID of the target process. It traverses the linked list of `task_struct` structures and matches the process ID to identify the target process. Once the process is identified, it is systematically killed using the Linux `force_sig` function to send a SIGKILL signal. When the kernel later schedules the process to execute, it notices that the process has a pending SIGKILL signal and terminates the process.

The `ps_deescalate` module takes care of resetting process credentials. It takes as input the process ID and the new values for the process credentials. Once the `task_struct` is identified, the process's `cred` structure is obtained by invoking the Linux `get_cred` function. The `get_cred` function increments the reference count and returns a pointer to the `cred` structure. After the credential values are reset, the `cred` structure is released using the `put_cred` function.

The `kill_socket` module can shut down all open sockets of a target process. This module takes the process ID of the target process as input. It traverses the `file_descriptor_table` of target process and identifies all the open sockets. Finally, all the open sockets are closed by calling the `shutdown` function on the `socket` structure.

The `close_file` module can close files opened by a process. This module takes the process ID of the target process and a file name as input. It traverses the file descriptor table of the target process and matches the file name to locate the file descriptor of the open file. The file is closed by invoking the `filp_close` function. The `file` pointer in the file descriptor table is nullified to prevent further access to the file.

The `system_map_reset` module provides two functions, one to fix corrupt system call table entries and another to unhook hijacked system calls. It fixes corrupt function pointers in the system call table by overwriting them with the original values that were captured during system initialization. Similarly, it repairs highjacked system calls by rewriting the first 12 bytes of instruction in the function prologue. The snapshot engine captures only the first 12 bytes in the function prologue because this is enough to implement function hijacking on the x86_64 architecture.

The `start_process` module implements functionality to start a user-space process from the kernel space. This module takes the path of the executable, executable parameters, and the environment parameters as input. It invokes the `call_usermodehelper` function provided by the Linux kernel to start a process in user space.

The `trusted_load` and `start_process` modules together implement functionality to start processes in a trusted-boot environment. The *trusted_load* module takes as input the names of blacklisted objects that are not allowed to be loaded by the process. The trusted-boot environment only allows the nonblacklisted objects to load during process startup. The `start_process` module starts a process from user space as mentioned earlier. The `trusted_load` module hooks the `open` and `mmap` system calls. The hooked versions of the `open` and `mmap` system calls return an `ENOENT` error when the process tries to load a blacklisted object. Thus the blacklisted objects are prevented from being loaded into process memory. The system calls are unhooked immediately after the `call_usermodehelper` function returns and the `start_process` module completes execution.

The `sled_object` module is a special repair tool that deals with malicious objects loaded in process memory. This module uses the Linux `get_user_pages` function to load the pages containing the text segment of the object. These pages are mapped to the address space of the repair tool using the `kmap` function. All instructions other than the return instruction in the text segment of the module are then over written by `noop` instructions. Thus, the `noop` sled overrides the actions of the malicious object code.

This repair tool works when the functions in the object being sledded have a `void` return type and are not required to manipulate data. For example, consider the Apache web server. All of Apache's functionality is implemented as a set of modules. When a web request comes in, a `request_rec` data structure and a `response_rec` data structure are created to represent the HTTP request and response, respectively. Each module in the Apache pipeline examines the `request_rec` record and decides if it has to be processed or not. If the request needs to be processed processed by the module, the module makes the necessary changes to the `response_rec` record and forwards it in the pipeline. If the request does not need to be processed by the module, then the module can simply return without modifying the `response_rec`. The next module in the pipeline does not rely on any data from the previous modules for its execution.

Another possible approach for dealing with malicious objects is to terminate the infected process and then restart it under a boot environment provided by the `trusted_load` module.

CHAPTER 5

EVALUATION

Our anomaly detection and recovery system was deployed to monitor a web server virtual appliance. In the following sections we describe our experiment setup and the evaluation of our system in terms of (i) expressiveness of the policies, (ii) run-time cost, and (iii) effectiveness. Expressiveness was qualitatively evaluated in terms of descriptiveness and the ease of writing the policies. The cost of the system was measured in terms of the time required to execute the system, which can reduce the run-time performance of the target VA. The effectiveness of the system was evaluated against different types of malicious software such as an application malware, a user-land rookit, a kernel space rootkit, and a kernel exploit kit.

The evaluation shows that an administrator's application-level knowledge can be used by an anomaly detection system to successfully detect anomalies in virtual appliances. Furthermore, the detection and recovery system can often, but not always, restore an affected virtual appliance to an acceptable state.

5.1 Experiment Setup

Our experiment setup consists of a web-server virtual appliance. This virtual appliance is a virtual machine that runs under the Xen 4.1.2 hypervisor. The virtual appliance runs an Ubuntu 12.04 distribution with a Linux 3.8.0 kernel and provides the necessary execution environment for the Apache 2.2 web server.

Dom0 acts as the control VM for the anomaly detection and recovery system. It also runs an Ubuntu 12.04 distribution with Linux 3.8.0 kernel. This VM has the CLIPS libraries, Stackdb libraries, and the debug symbols installed that are required by the anomaly detection system.

The Xen virtualization environment is deployed on a single d710 node in the Utah Emulab testbed [32]. This d710 node consists of a 64-bit Intel 2.40GHz quad-core Xeon E5530 processor with 4 cores, and 12GB of RAM.

The policy in the knowledge store describe the acceptable states of the web server virtual appliance. It allows the execution of an Apache web server with support from a MySQL database server, NTP daemon, FTP daemon, SSH daemon, and PHP processes. The policy also allows routine kernel processes such as kworker, swapper, rcu_sched, kswap, and xenwatch. For each authorized process executing on the virtual appliance, the policy declares the process's credentials, priorities, CPU utilization, loaded objects, file access, and network access information.

5.2 Expressiveness of Policies

The policy captures an administrator's application-level knowledge for the VA used in our evaluation. Our implementation of the knowledge store consists of 940 lines of CLIPS facts. The majority of these facts describe the processes that are expected to execute in the virtual appliance. These facts capture information about process credentials, process priorities, authority of the process to access files or the network, and process lineage information. Since the administrator is expected to have knowledge of all the processes that are allowed to execute on a virtual appliance, the administrator should be able to express his or her knowledge as facts. The main challenge for an administrator in expressing his or her domain knowledge as facts is to learn and use the programming constructs provided by CLIPS. In the future, the process of expressing an administrator's domain knowledge as facts can be simplified by a tool that captures these facts from the target VA that is known to be in a reliable state.

As mentioned earlier, the set of acceptable states for a virtual appliance is decided by the administrator and is specified through a policy. Ideally, the policy specified by the administrator should capture the complete set of acceptable states and should not include any anomalous states of the VA. For the evaluation of our system, we believe we were able to write a policy that reasonably approximates administrators expectation of a web server VA. As shown in later sections, the policy proves to be sufficient in detecting and mitigating a majority of anomalies caused by different types of malicious software.

5.3 Run-Time Overhead

The execution of the anomaly detection and recovery system impacts the availability of the virtual appliance. The snapshot engine makes use of VMI tools to capture snapshots of the virtual appliance. For the snapshotting operation to be atomic with respect to the execution of the virtual appliance, the anomaly detection system pauses the VA each time the snapshot engine is activated. During the time that the virtual appliance is paused, no work is done by the virtual appliance. Table 5.1 shows the average and standard deviation of the execution times for each of the VMI tools over 10 iterations. The total runtime of the snapshot engine is about 140.3 milliseconds. During that time, the virtual appliance is paused and no work is done by it.

With CLIPS debugging enabled, the average run times of the policy engine and the repair engine over 10 iterations is about 118 microseconds with a standard deviation of 23. Since the execution of

Table 5.1. VMI Tool Execution Times

VMI Tool	Time (msec)	Standard Deviation
Process info	21.4	2.6
File info	16.3	1.9
Module info	62.5	0.5
CPU Load info	1.0	0.0
Object info	27.8	2.1
Command line info	10.3	0.7
System call table info	1.0	0.0
Total Time	**140.3**	**5.4**

the policy engine and repair engine does not require the target VA to be paused, their execution does not affect the applications in the target VA.

The repair driver and the repair tools execute within the kernel space of the target VA. Their execution does not impact the availability of the VA but may indirectly affect the run time of applications.

If one schedules the anomaly detection and recovery system to run once in every 5 seconds, it results in approximately 140.3 milliseconds of downtime for every 5 seconds of execution of the virtual appliance. This equates to 2.79% unavailability for the virtual appliance, which may be too high for some production environments. Based on requirements of the production environment, one can tune the percentage of the time paused by changing the frequency of the snapshotting operation. Higher frequency results in reduced availability of the virtual appliance, while lower frequency increases the latency of anomaly detection and repair.

The anomaly detection and recovery system could be enhanced with a dynamic frequency scaling capability. The frequency of the snapshotting operation could be varied based on the number of anomalies detected in recent snapshots; a large number of anomalies detected results in a high frequency and a small number of anomalies results in a low frequency.

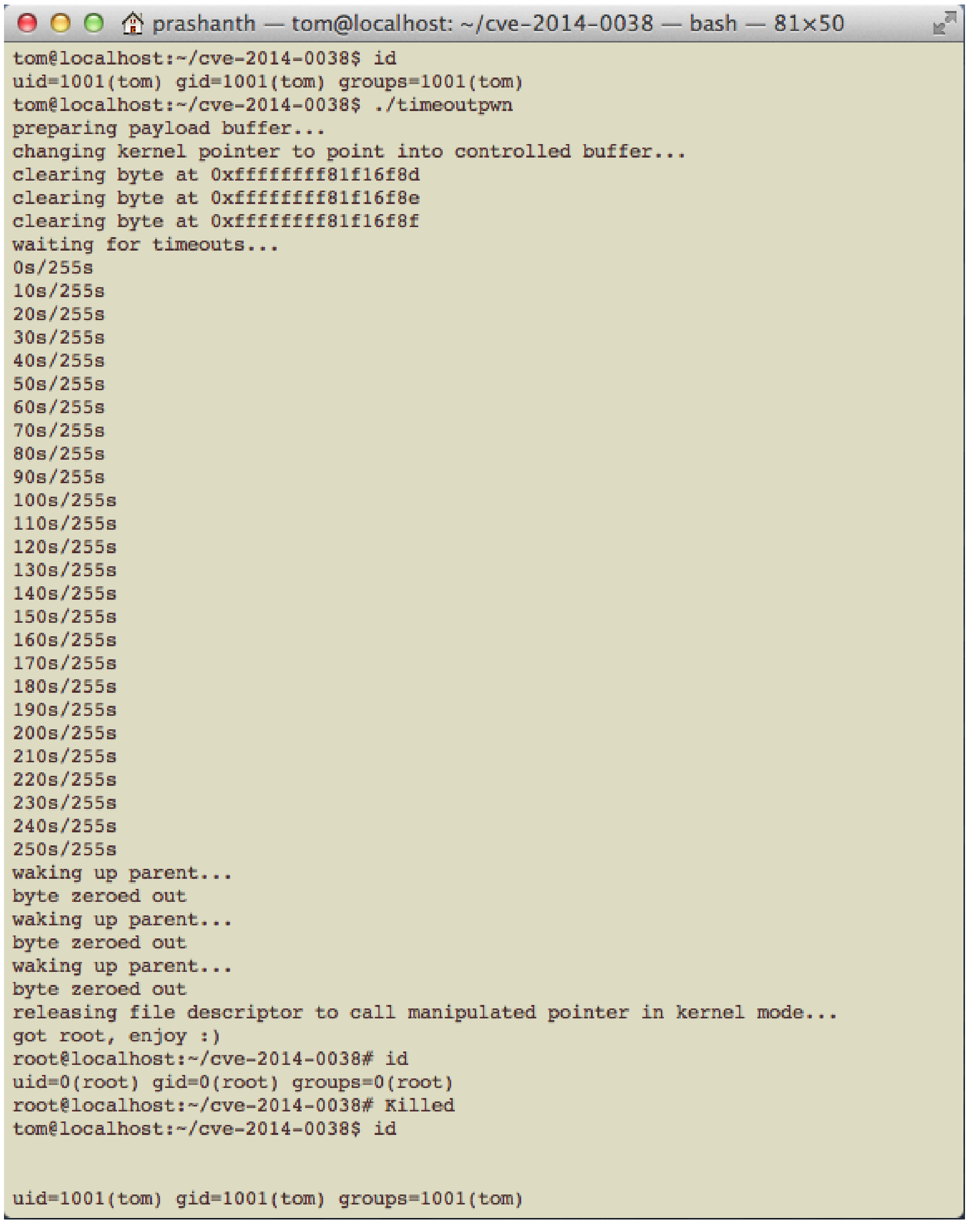
5.4 Effectiveness

To test the effectiveness of our anomaly detection and recovery system, we ran it against different types of malicious software: (i) a kernel exploit kit, (ii) a kernel-space rootkit, (iii) a user-land rootkit, and (iv) an application malware.

For each of the experiments described in the following subsections, we manually start the anomaly detection and recovery system only after the malicious software is installed on the VA. If the anomaly detection and recovery system were active during the installation of the malicious software, it would detect and try to terminate the installation of the malicious software itself.

5.4.1 Kernel Exploit

CVE-2014-0038 [2] describes a bug in the x32 version of the `recvmmsg` system call in the Linux kernel. The `recvmmsg` system call allows multiple messages to be received on a socket with just one system call. The bug results from blindly trusting the contents of the timeout pointer that is passed as input from user space. This bug can be exploited by making the timeout pointer refer to some kernel memory and overwriting the memory location with malicious code. Figure 5.1 is a screenshot that demonstrates how the bug can be exploited to gain root access on our web server VA. As shown in the figure, `timeoutpwn` is the name of the exploit program [7] that spawns a root shell.



```
tom@localhost:~/cve-2014-0038$ id
uid=1001(tom) gid=1001(tom) groups=1001(tom)
tom@localhost:~/cve-2014-0038$ ./timeoutpwn
preparing payload buffer...
changing kernel pointer to point into controlled buffer...
clearing byte at 0xffffffff81f16f8d
clearing byte at 0xffffffff81f16f8e
clearing byte at 0xffffffff81f16f8f
waiting for timeouts...
0s/255s
10s/255s
20s/255s
30s/255s
40s/255s
50s/255s
60s/255s
70s/255s
80s/255s
90s/255s
100s/255s
110s/255s
120s/255s
130s/255s
140s/255s
150s/255s
160s/255s
170s/255s
180s/255s
190s/255s
200s/255s
210s/255s
220s/255s
230s/255s
240s/255s
250s/255s
waking up parent...
byte zeroed out
waking up parent...
byte zeroed out
waking up parent...
byte zeroed out
releasing file descriptor to call manipulated pointer in kernel mode...
got root, enjoy :)
root@localhost:~/cve-2014-0038# id
uid=0(root) gid=0(root) groups=0(root)
root@localhost:~/cve-2014-0038# Killed
tom@localhost:~/cve-2014-0038$ id


uid=1001(tom) gid=1001(tom) groups=1001(tom)
```

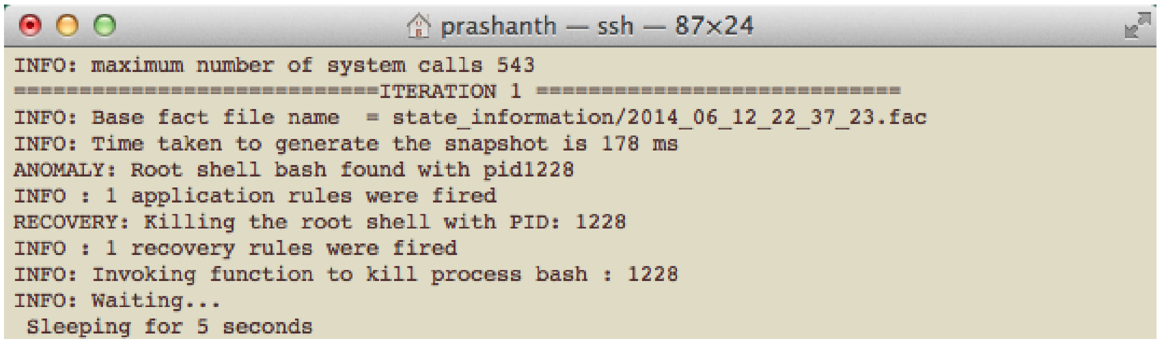
Figure 5.1. CVE-2014-0038 Exploitation

The anomaly detection system detects this root shell by validating the process lineages of each process against the defined policies. Because the root shell is spawned from a process that does not have root privileges, the existence of the root shell is flagged as an anomaly. The root shell is terminated as part of the recovery process. This is shown in Figure 5.2. Thus, although our anomaly detection and recovery system is not able to identify or correct the underlying vulnerability in the Linux kernel, the malicious activity emerging from the exploitation of this vulnerability can be identified and subdued.

As shown in Figure 5.1, the exploit program takes about 255 seconds to complete its execution. If the anomaly detection and recovery component were scheduled to execute at frequent intervals of time, then the anomaly detection component would detect the execution of the exploit program itself. Thus, the recovery component would have taken an appropriate measure to prevent the exploit program from compromising the virtual appliance.

5.4.2 Kernel Rootkit

A kernel rootkit is a malicious program that can be used to provide continuous unauthorized root access to a computer system. A rootkit can also hide the traces of existence of itself as well as that of other malicious programs in a computer system. Suterusu [13] is a kernel rootkit targeting Linux 3.X kernels on x86_64 architectures. This rootkit provides multiple features including (i) root shell access, (ii) process hiding, (iii) network socket hiding, (iv) file/directory hiding, and (v) enabling and disabling module loading. Suterusu implements these features by hooking functions in the Linux kernel, mainly the functions that implement `/proc` file system. Unlike traditional rootkits, Suterusu does not perform system call hooking by modifying the function pointers in the system call table. Instead, it overwrites instructions in the prologues of target functions. The advantage of this hooking approach is that Suterusu can evade detection by most rootkit detectors. It allows not only system calls but also any kernel function to be hooked.



```
INFO: maximum number of system calls 543
============================ITERATION 1 ============================
INFO: Base fact file name  = state_information/2014_06_12_22_37_23.fac
INFO: Time taken to generate the snapshot is 178 ms
ANOMALY: Root shell bash found with pid1228
INFO : 1 application rules were fired
RECOVERY: Killing the root shell with PID: 1228
INFO : 1 recovery rules were fired
INFO: Invoking function to kill process bash : 1228
INFO: Waiting...
 Sleeping for 5 seconds
```

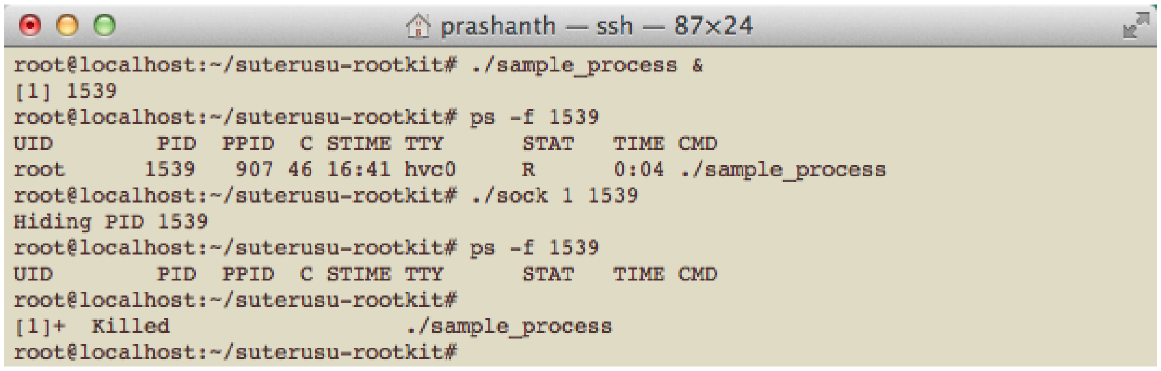
Figure 5.2. Root Shell Detection and Recovery

Suterusu hooks the `proc_root_readdir` function of the `/proc` file system to implement process hiding. Figure 5.3 is a screenshot that demonstrates the use of Suterusu's process-hiding functionality to hide the process `sample_process` with PID 1539. After the execution of the "`sock 1 1539`" command, the process is not listed by the `ps` command.

Figure 5.4 shows how the snapshot engine is able to discover the existence of the `sampl-e_process`. The policy engine identifies the existence of the process as a policy violation and reports it as an anomaly. The repair engine decides to terminate the process as a recovery action so that the virtual appliance can be restored to an acceptable state. The repair action is carried out by the `psaction` repair tool.

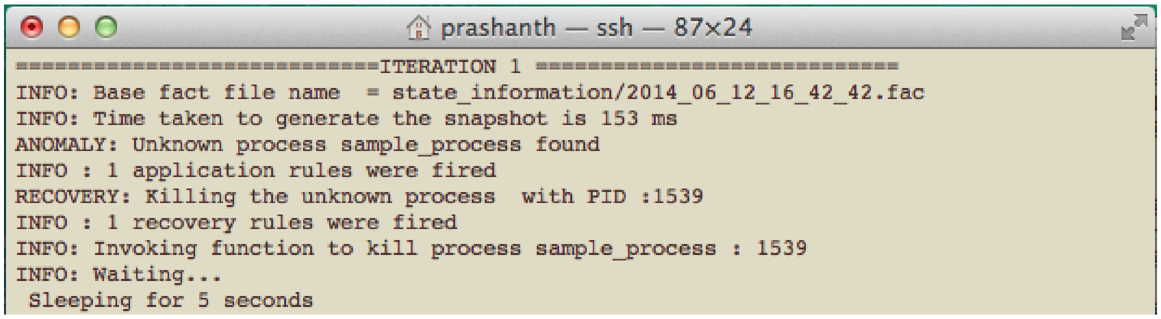
Legitimate programs can be compromised so that they perform certain malicious activities without the knowledge of the administrator. To detect such malicious activities, the policy defined for our web server virtual appliance restricts the actions of authorized processes. For example, for each process, the policy defines the list of files that the process is allowed to access. Figure 5.5 is a screenshot that captures the output of the `lsof` command. Here, `samp_proc` is a process that is authorized to execute in the virtual appliance, but it makes unauthorized access to the file `log.txt`.

Figure 5.6 shows how the anomaly detection system identifies this unauthorized access and closes



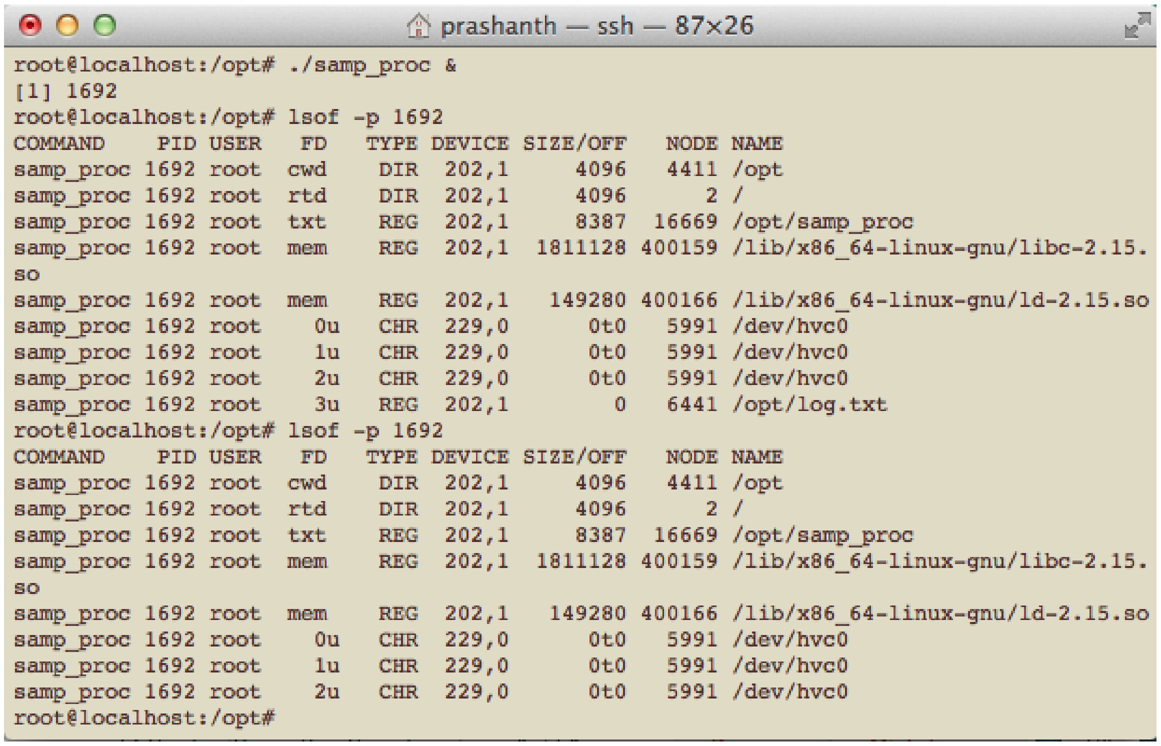
```
root@localhost:~/suterusu-rootkit# ./sample_process &
[1] 1539
root@localhost:~/suterusu-rootkit# ps -f 1539
UID        PID  PPID  C STIME TTY      STAT   TIME CMD
root      1539   907 46 16:41 hvc0     R      0:04 ./sample_process
root@localhost:~/suterusu-rootkit# ./sock 1 1539
Hiding PID 1539
root@localhost:~/suterusu-rootkit# ps -f 1539
UID        PID  PPID  C STIME TTY      STAT   TIME CMD
root@localhost:~/suterusu-rootkit#
[1]+  Killed                  ./sample_process
root@localhost:~/suterusu-rootkit#
```

Figure 5.3. Process Hiding



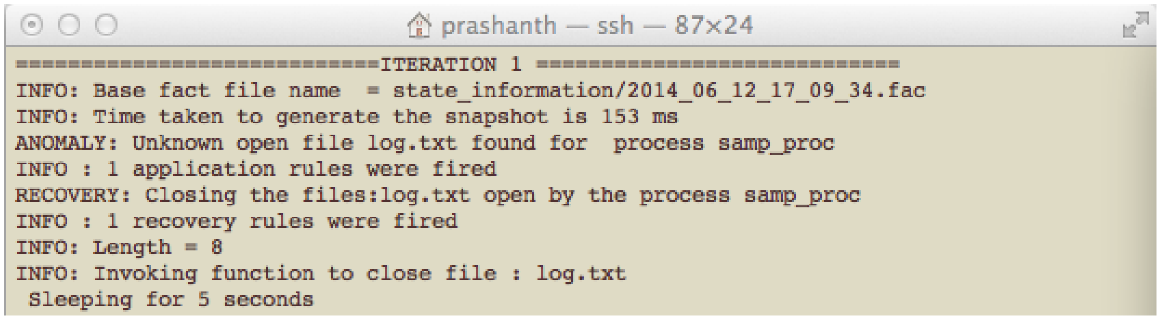
```
============================ITERATION 1 ============================
INFO: Base fact file name  = state_information/2014_06_12_16_42_42.fac
INFO: Time taken to generate the snapshot is 153 ms
ANOMALY: Unknown process sample_process found
INFO : 1 application rules were fired
RECOVERY: Killing the unknown process  with PID :1539
INFO : 1 recovery rules were fired
INFO: Invoking function to kill process sample_process : 1539
INFO: Waiting...
 Sleeping for 5 seconds
```

Figure 5.4. Hidden Process Detection and Repair



```
root@localhost:/opt# ./samp_proc &
[1] 1692
root@localhost:/opt# lsof -p 1692
COMMAND     PID USER   FD   TYPE DEVICE SIZE/OFF   NODE NAME
samp_proc 1692 root  cwd    DIR  202,1     4096   4411 /opt
samp_proc 1692 root  rtd    DIR  202,1     4096      2 /
samp_proc 1692 root  txt    REG  202,1     8387  16669 /opt/samp_proc
samp_proc 1692 root  mem    REG  202,1  1811128 400159 /lib/x86_64-linux-gnu/libc-2.15.
so
samp_proc 1692 root  mem    REG  202,1   149280 400166 /lib/x86_64-linux-gnu/ld-2.15.so
samp_proc 1692 root    0u   CHR  229,0      0t0   5991 /dev/hvc0
samp_proc 1692 root    1u   CHR  229,0      0t0   5991 /dev/hvc0
samp_proc 1692 root    2u   CHR  229,0      0t0   5991 /dev/hvc0
samp_proc 1692 root    3u   REG  202,1        0   6441 /opt/log.txt
root@localhost:/opt# lsof -p 1692
COMMAND     PID USER   FD   TYPE DEVICE SIZE/OFF   NODE NAME
samp_proc 1692 root  cwd    DIR  202,1     4096   4411 /opt
samp_proc 1692 root  rtd    DIR  202,1     4096      2 /
samp_proc 1692 root  txt    REG  202,1     8387  16669 /opt/samp_proc
samp_proc 1692 root  mem    REG  202,1  1811128 400159 /lib/x86_64-linux-gnu/libc-2.15.
so
samp_proc 1692 root  mem    REG  202,1   149280 400166 /lib/x86_64-linux-gnu/ld-2.15.so
samp_proc 1692 root    0u   CHR  229,0      0t0   5991 /dev/hvc0
samp_proc 1692 root    1u   CHR  229,0      0t0   5991 /dev/hvc0
samp_proc 1692 root    2u   CHR  229,0      0t0   5991 /dev/hvc0
root@localhost:/opt#
```

Figure 5.5. Unauthorized File Access



```
============================ITERATION 1 ============================
INFO: Base fact file name  = state_information/2014_06_12_17_09_34.fac
INFO: Time taken to generate the snapshot is 153 ms
ANOMALY: Unknown open file log.txt found for  process samp_proc
INFO : 1 application rules were fired
RECOVERY: Closing the files:log.txt open by the process samp_proc
INFO : 1 recovery rules were fired
INFO: Length = 8
INFO: Invoking function to close file : log.txt
 Sleeping for 5 seconds
```

Figure 5.6. Unauthorized File Access Detection and Repair

the process's file descriptor for `log.txt`, releasing the file descriptor from the process. The second `lsof` command output in Figure 5.5 shows that the `samp_proc` process no longer accesses the file `log.txt`.

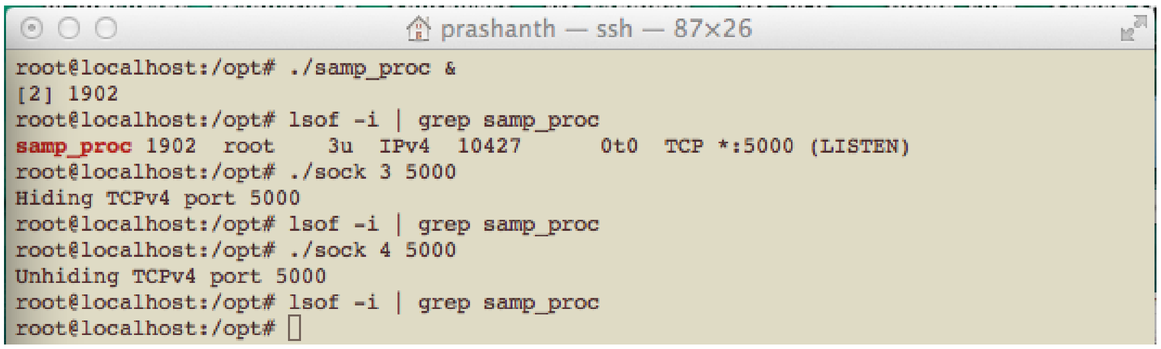
The Suterusu rootkit hooks the `tcp4_seq_show`, `tcp6_seq_show`, `udp4_seq_show`, and the `udp6_seq_show` functions so that it can hide any TCP or UDP socket. Figure 5.7 shows how the Suterusu rootkit can hide network access.

According to the policy for our VA, the `samp_proc` process is authorized to execute in the virtual appliance but it is not allowed to have network connections. The `samp_proc` process opens a TCP listening socket that waits for incoming connections. The rootkit hides the TCP socket so that it is not listed by the `lsof` and `netstat` utilities. The anomaly detection and recovery system is able to detect the socket because of the external view provided by the snapshot engine. The recovery engine decides to terminate the socket as part of the recovery action. This is shown in Figure 5.8.

The Suterusu rootkit implements most of its functionality by hooking functions that implement the `/proc` file system. Our anomaly detection and recovery system is only equipped to detect and recover from corrupted system call table entries and hooked system calls, so it cannot deactivate the rootkit. Although our system is not able to detect and recover from the hooks on generic functions implemented by this rootkit, it was able to detect and recover from the malicious activity enabled by the hooked functions.

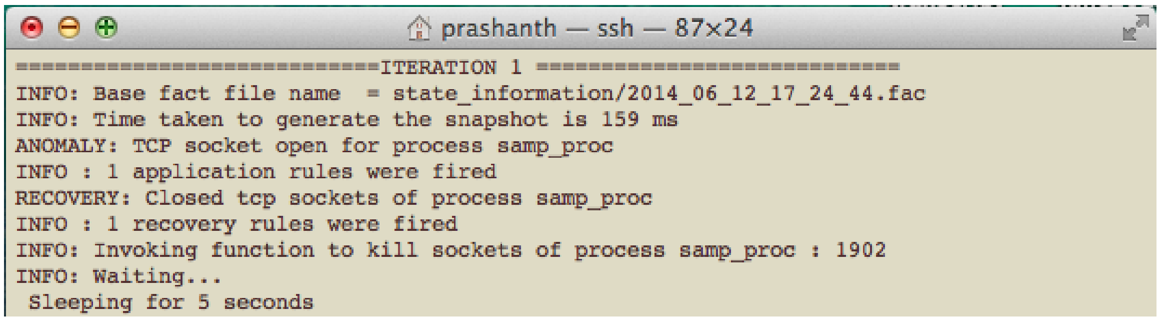
5.4.3 Userland Rootkit

A userland rootkit executes in the user space of a computer system along with other applications. Azazel [1] is a userland rootkit based on the LD_PRELOAD technique. This rootkit hooks individual programs at the time of execution by setting the LD_PRELOAD environment variable to override standard C library functions with malicious ones. It also modifies the `ld.so.preload` file to load malicious shared libraries during program startup.



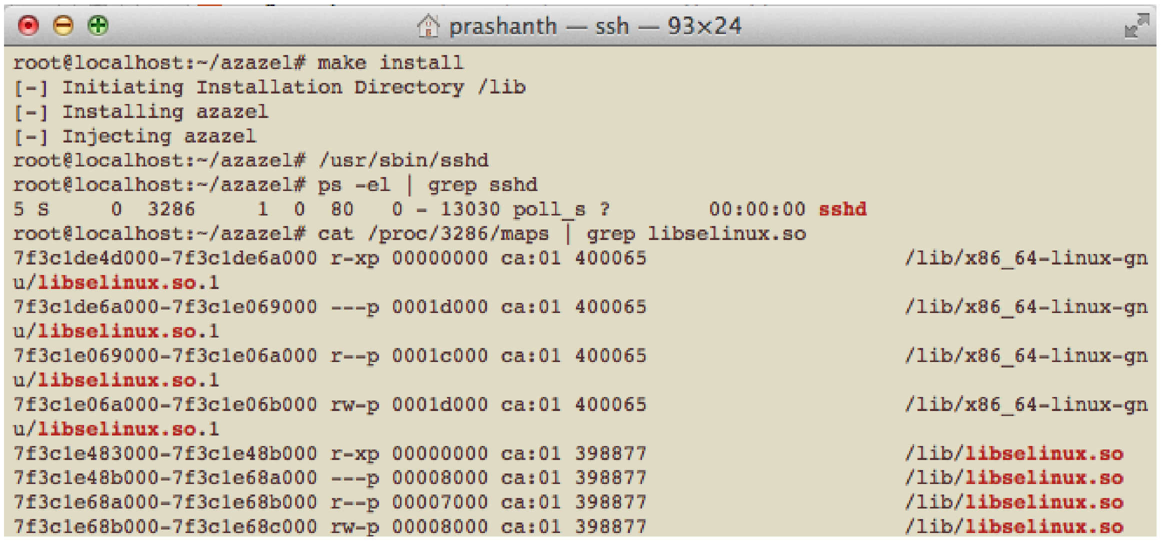
```
root@localhost:/opt# ./samp_proc &
[2] 1902
root@localhost:/opt# lsof -i | grep samp_proc
samp_proc 1902  root    3u  IPv4  10427      0t0  TCP *:5000 (LISTEN)
root@localhost:/opt# ./sock 3 5000
Hiding TCPv4 port 5000
root@localhost:/opt# lsof -i | grep samp_proc
root@localhost:/opt# ./sock 4 5000
Unhiding TCPv4 port 5000
root@localhost:/opt# lsof -i | grep samp_proc
root@localhost:/opt# 
```

Figure 5.7. Unauthorized Network Access



```
============================ITERATION 1 ============================
INFO: Base fact file name  = state_information/2014_06_12_17_24_44.fac
INFO: Time taken to generate the snapshot is 159 ms
ANOMALY: TCP socket open for process samp_proc
INFO : 1 application rules were fired
RECOVERY: Closed tcp sockets of process samp_proc
INFO : 1 recovery rules were fired
INFO: Invoking function to kill sockets of process samp_proc : 1902
INFO: Waiting...
 Sleeping for 5 seconds
```

Figure 5.8. Unauthorized Network Access Detection and Repair

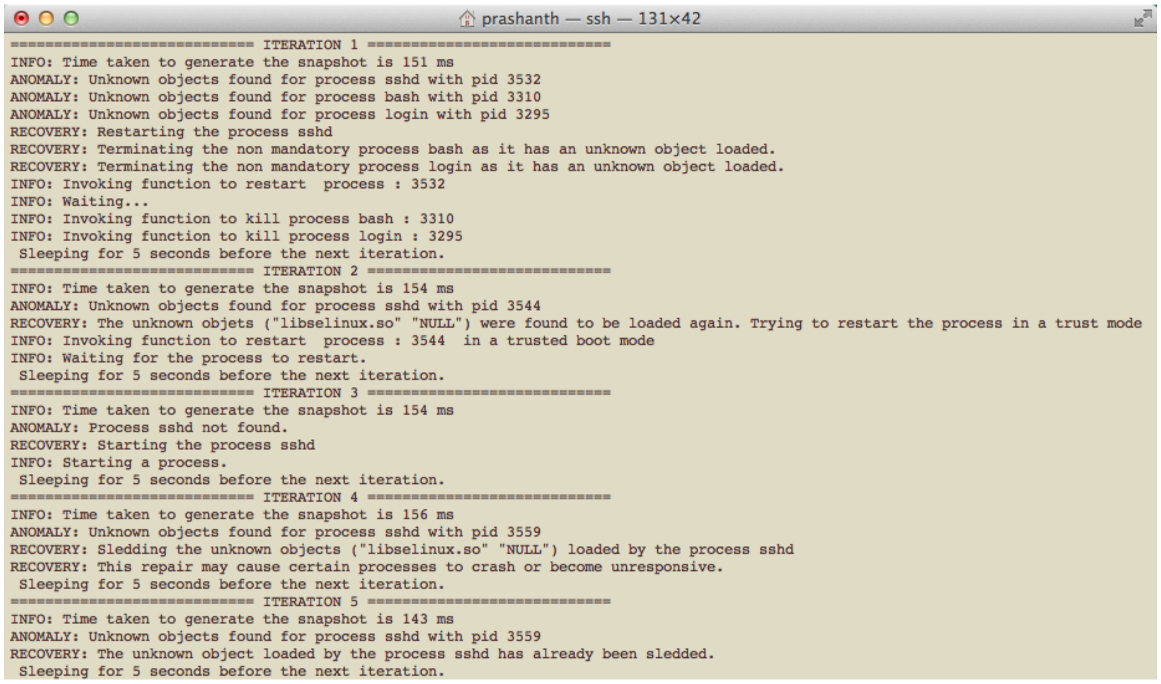


```
root@localhost:~/azazel# make install
[-] Initiating Installation Directory /lib
[-] Installing azazel
[-] Injecting azazel
root@localhost:~/azazel# /usr/sbin/sshd
root@localhost:~/azazel# ps -el | grep sshd
5 S     0  3286     1  0  80   0 - 13030 poll_s ?        00:00:00 sshd
root@localhost:~/azazel# cat /proc/3286/maps | grep libselinux.so
7f3c1de4d000-7f3c1de6a000 r-xp 00000000 ca:01 400065                     /lib/x86_64-linux-gn
u/libselinux.so.1
7f3c1de6a000-7f3c1e069000 ---p 0001d000 ca:01 400065                     /lib/x86_64-linux-gn
u/libselinux.so.1
7f3c1e069000-7f3c1e06a000 r--p 0001c000 ca:01 400065                     /lib/x86_64-linux-gn
u/libselinux.so.1
7f3c1e06a000-7f3c1e06b000 rw-p 0001d000 ca:01 400065                     /lib/x86_64-linux-gn
u/libselinux.so.1
7f3c1e483000-7f3c1e48b000 r-xp 00000000 ca:01 398877                     /lib/libselinux.so
7f3c1e48b000-7f3c1e68a000 ---p 00008000 ca:01 398877                     /lib/libselinux.so
7f3c1e68a000-7f3c1e68b000 r--p 00007000 ca:01 398877                     /lib/libselinux.so
7f3c1e68b000-7f3c1e68c000 rw-p 00008000 ca:01 398877                     /lib/libselinux.so
```

Figure 5.9. Azazel Installation

A program infected by Azazel has a malicious object `libselinux.so` loaded into its memory. This object provides functions to spawn unauthorized root shells, hide processes, and deploy backdoors. Figure 5.9 shows the installation of the Azazel rootkit. This installation causes it to load the `libselinux.so` malicious object into the `sshd`, `login`, and `bash` processes. This loaded object is hidden from the `ldd` and `lsof` commands.

Figure 5.10 shows that the anomaly detection component is able to identify the loaded shared object as an anomaly in the `sshd`, `login`, and `bash` processes. As a recovery measure, the recovery component terminates and restarts the `sshd` process. Since the policy does not specify the `bash` and `login` process to be mandatory, they are terminated (iteration 1 in Figure 5.10). In the second iteration, the anomaly detection system notices that the unknown object is loaded again in the `sshd` process memory. Hence this time, the `sshd` process is terminated and restarted in a trusted boot environment (iteration 2 in Figure 5.10). The `sshd` process does not start correctly in the trusted



prashanth — ssh — 131×42

```
============================ ITERATION 1 ============================
INFO: Time taken to generate the snapshot is 151 ms
ANOMALY: Unknown objects found for process sshd with pid 3532
ANOMALY: Unknown objects found for process bash with pid 3310
ANOMALY: Unknown objects found for process login with pid 3295
RECOVERY: Restarting the process sshd
RECOVERY: Terminating the non mandatory process bash as it has an unknown object loaded.
RECOVERY: Terminating the non mandatory process login as it has an unknown object loaded.
INFO: Invoking function to restart  process : 3532
INFO: Waiting...
INFO: Invoking function to kill process bash : 3310
INFO: Invoking function to kill process login : 3295
 Sleeping for 5 seconds before the next iteration.
============================ ITERATION 2 ============================
INFO: Time taken to generate the snapshot is 154 ms
ANOMALY: Unknown objects found for process sshd with pid 3544
RECOVERY: The unknown objets ("libselinux.so" "NULL") were found to be loaded again. Trying to restart the process in a trust mode
INFO: Invoking function to restart  process : 3544  in a trusted boot mode
INFO: Waiting for the process to restart.
 Sleeping for 5 seconds before the next iteration.
============================ ITERATION 3 ============================
INFO: Time taken to generate the snapshot is 154 ms
ANOMALY: Process sshd not found.
RECOVERY: Starting the process sshd
INFO: Starting a process.
 Sleeping for 5 seconds before the next iteration.
============================ ITERATION 4 ============================
INFO: Time taken to generate the snapshot is 156 ms
ANOMALY: Unknown objects found for process sshd with pid 3559
RECOVERY: Sledding the unknown objects ("libselinux.so" "NULL") loaded by the process sshd
RECOVERY: This repair may cause certain processes to crash or become unresponsive.
 Sleeping for 5 seconds before the next iteration.
============================ ITERATION 5 ============================
INFO: Time taken to generate the snapshot is 143 ms
ANOMALY: Unknown objects found for process sshd with pid 3559
RECOVERY: The unknown object loaded by the process sshd has already been sledded.
 Sleeping for 5 seconds before the next iteration.
```

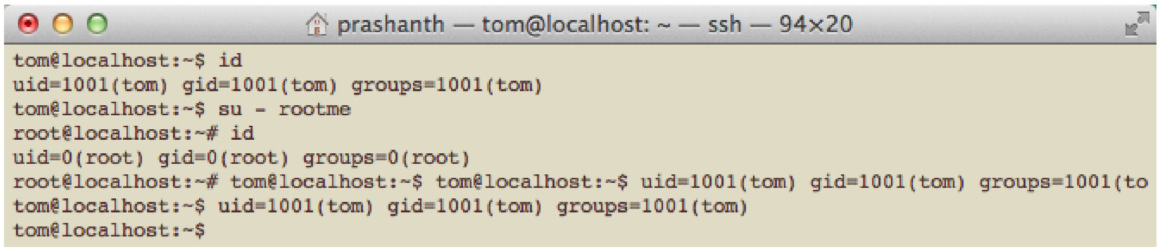
Figure 5.10. Detecting Malicious Libraries

boot environment as it is not allowed to load the `libselinux.so` object. If any processes that is defined as mandatory for the virtual appliance is missing, then the absence of that process in the virtual appliance is considered to be an anomaly and the process is restarted as part of recovery. In this experiment, since `sshd` is an important service for the virtual appliance it is restarted in the next iteration (iteration 3). However, since Azazel has overwritten the `ld.so.preload` file, the malicious library is loaded again in the `sshd` process memory. In the next iteration, the recovery component decides to sled all the instructions in the unknown object (iteration 4). In all subsequent iterations no recovery action is initiated if the unknown object is detected in the memory of previously observed `sshd` processes. However, if the unknown object is detected in new `sshd` processes, then the unknown object is directly sledded.

The anomaly detection and recovery system is able to detect the Azazel compromise and restore the virtual appliance to a state that is acceptable according to the policy defined. By restarting the `sshd` process the recovery system is able to close the backdoor setup by Azazel. We noticed that the `sshd` process becomes unresponsive to SSH-connection requests after the unknown object was sledded. The nonresponsiveness of the `sshd` process is an anomaly that is not detected by our anomaly detection system. The VA is in a state that is acceptable according to our policy, but still has anomalous behavior. Since the backdoor is closed and the operation of the Apache web server is not affected, we claim that the anomaly detection and recovery system is successful in mitigating the malicious effects of Azazel.

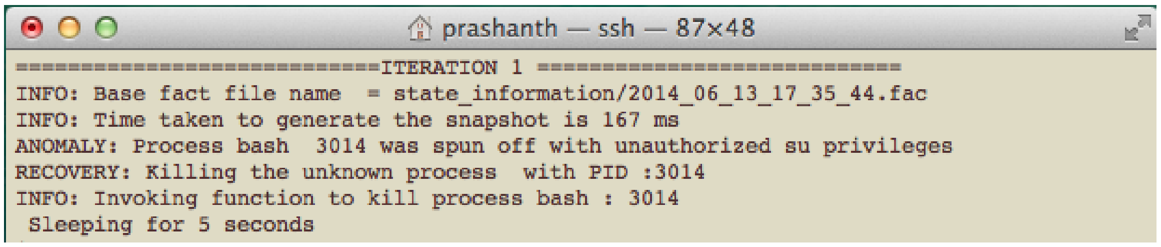
The nonresponsiveness of `sshd` could potentially be eliminated by developing a recovery tool that could rewrite the instruction in the unknown objects in a smart manner. Our anomaly detection rules could also be enhanced to be able to detect unresponsive processes based on some process statistics such as CPU utilization.

Figure 5.11 shows how Azazel can be used to exploit the substitute user (`su`) command to spawn a root shell. Figure 5.12 shows that the anomaly detection system is able to detect the root shell and terminate it. The anomaly detection system detects such privilege escalations by validating the process lineage information in the snapshot against the process lineage specified in the policy.



```
tom@localhost:~$ id
uid=1001(tom) gid=1001(tom) groups=1001(tom)
tom@localhost:~$ su - rootme
root@localhost:~# id
uid=0(root) gid=0(root) groups=0(root)
root@localhost:~# tom@localhost:~$ tom@localhost:~$ uid=1001(tom) gid=1001(tom) groups=1001(to
tom@localhost:~$ uid=1001(tom) gid=1001(tom) groups=1001(tom)
tom@localhost:~$
```

Figure 5.11. `su` Command Exploitation



```
============================ITERATION 1 ============================
INFO: Base fact file name  = state_information/2014_06_13_17_35_44.fac
INFO: Time taken to generate the snapshot is 167 ms
ANOMALY: Process bash  3014 was spun off with unauthorized su privileges
RECOVERY: Killing the unknown process  with PID :3014
INFO: Invoking function to kill process bash : 3014
 Sleeping for 5 seconds
```

Figure 5.12. Detection and Repair of `su` Exploit

Azazel sets up an SSH backdoor by preloading malicious objects during `sshd` process startup. Figure 5.13 shows how a connection is established from a remote machine to the backdoor to get access to a shell on our web server VA. The anomaly detection and recovery system detects that the `sshd` process has been compromised and restarts it, resulting in the `sshd` session being terminated.

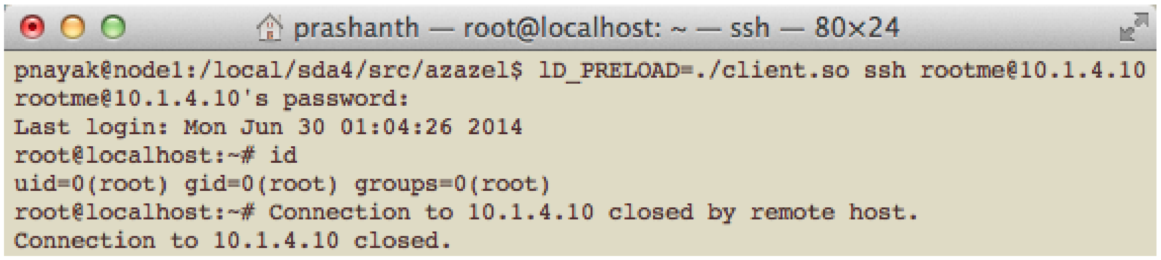
The detection and repair of the compromise is shown in Figure 5.14. In the case of Azazel, a legitimate process is compromised to deploy a backdoor and hence the anomaly detection system detects it as a corrupted process rather than a backdoor itself. However, termination of the corrupted process also results in terminating the backdoor.

Thus, by validating the shared libraries loaded by processes, the anomaly detection and recovery system is able to detect the anomalous actions of authorized processes.

5.4.4 Application Malware

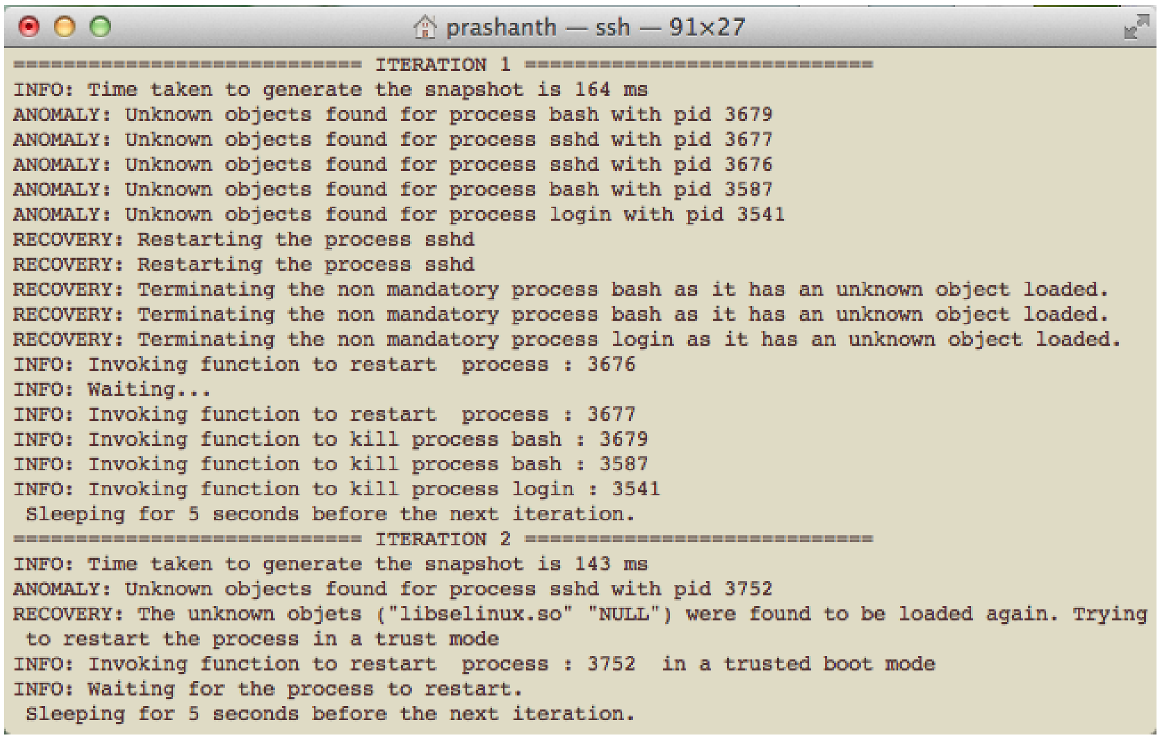
Darkleech [22] is a malicious Apache web server module. Darkleech analyzes HTTP traffic and injects malicious iframes into legitimate web server responses. These iframes redirect the HTTP clients to other malicious sites. The Darkleech module is loaded every time the Apache web server is started though a `LoadModule` command, which is added to one of the Apache configuration files (generally `httpd.conf`).

Darkleech injects iframes into legitimate HTTP responses. However, it takes care so that is it not easily detected. We received a sample of a version of the Darkleech binary from Sucuri Security [4]. A sample of the malicious code injected by the Darkleech malware is shown in Listing 5.1.



```
pnayak@node1:/local/sda4/src/azazel$ lD_PRELOAD=./client.so ssh rootme@10.1.4.10
rootme@10.1.4.10's password:
Last login: Mon Jun 30 01:04:26 2014
root@localhost:~# id
uid=0(root) gid=0(root) groups=0(root)
root@localhost:~# Connection to 10.1.4.10 closed by remote host.
Connection to 10.1.4.10 closed.
```

Figure 5.13. Azazel Backdoor



```
============================ ITERATION 1 ============================
INFO: Time taken to generate the snapshot is 164 ms
ANOMALY: Unknown objects found for process bash with pid 3679
ANOMALY: Unknown objects found for process sshd with pid 3677
ANOMALY: Unknown objects found for process sshd with pid 3676
ANOMALY: Unknown objects found for process bash with pid 3587
ANOMALY: Unknown objects found for process login with pid 3541
RECOVERY: Restarting the process sshd
RECOVERY: Restarting the process sshd
RECOVERY: Terminating the non mandatory process bash as it has an unknown object loaded.
RECOVERY: Terminating the non mandatory process bash as it has an unknown object loaded.
RECOVERY: Terminating the non mandatory process login as it has an unknown object loaded.
INFO: Invoking function to restart  process : 3676
INFO: Waiting...
INFO: Invoking function to restart  process : 3677
INFO: Invoking function to kill process bash : 3679
INFO: Invoking function to kill process bash : 3587
INFO: Invoking function to kill process login : 3541
 Sleeping for 5 seconds before the next iteration.
============================ ITERATION 2 ============================
INFO: Time taken to generate the snapshot is 143 ms
ANOMALY: Unknown objects found for process sshd with pid 3752
RECOVERY: The unknown objets ("libselinux.so" "NULL") were found to be loaded again. Trying
 to restart the process in a trust mode
INFO: Invoking function to restart  process : 3752  in a trusted boot mode
INFO: Waiting for the process to restart.
 Sleeping for 5 seconds before the next iteration.
```

Figure 5.14. Backdoor Detection and Repair

Listing 5.1: Iframe Sample

```
<style>
.loqxfd2c { position:absolute; left:-1004px; top:-1566px}
</style>
<div class="loqxfd2c">
<iframe
<src="http://xacirko.myftp.biza/a61f23b8bb0285742fc03463ec2aa0e10-
      o0o02606900881" width="416" height="343">
</iframe>
</div>
```

The URLs in the iframes are obtained from an external server and they change periodically. These URLs point to blackhole sites. Blackhole sites [15] host malicious programs that exploit the vulnerabilities of machines trying to access them.

To test the effectiveness of our anomaly detection and recovery system against Darkleech, we built an experiment setup consisting of three clients and a web server virtual appliance in Emulab. The web server and the clients in the experiment are as shown in Figure 5.15.

Node1 hosts the web server VA. *Node2*, *Node3*, and *Node4* are the clients that make HTTP requests to the web server. The four nodes are configured to be in different networks and are connected by a router. The web server virtual appliance was monitored using the policy-driven anomaly detection and recovery system. The three client nodes make HTTP request to the web server

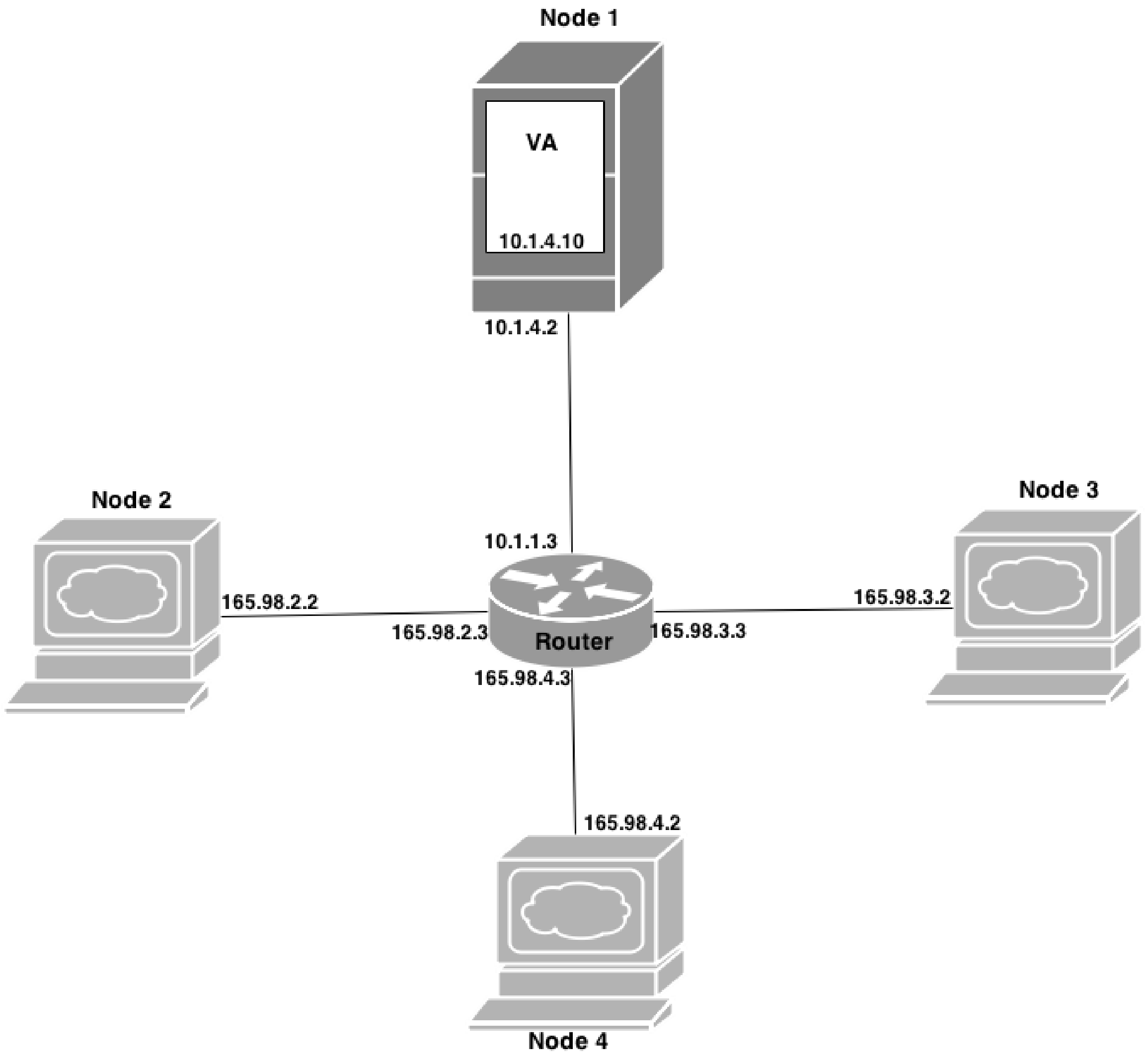
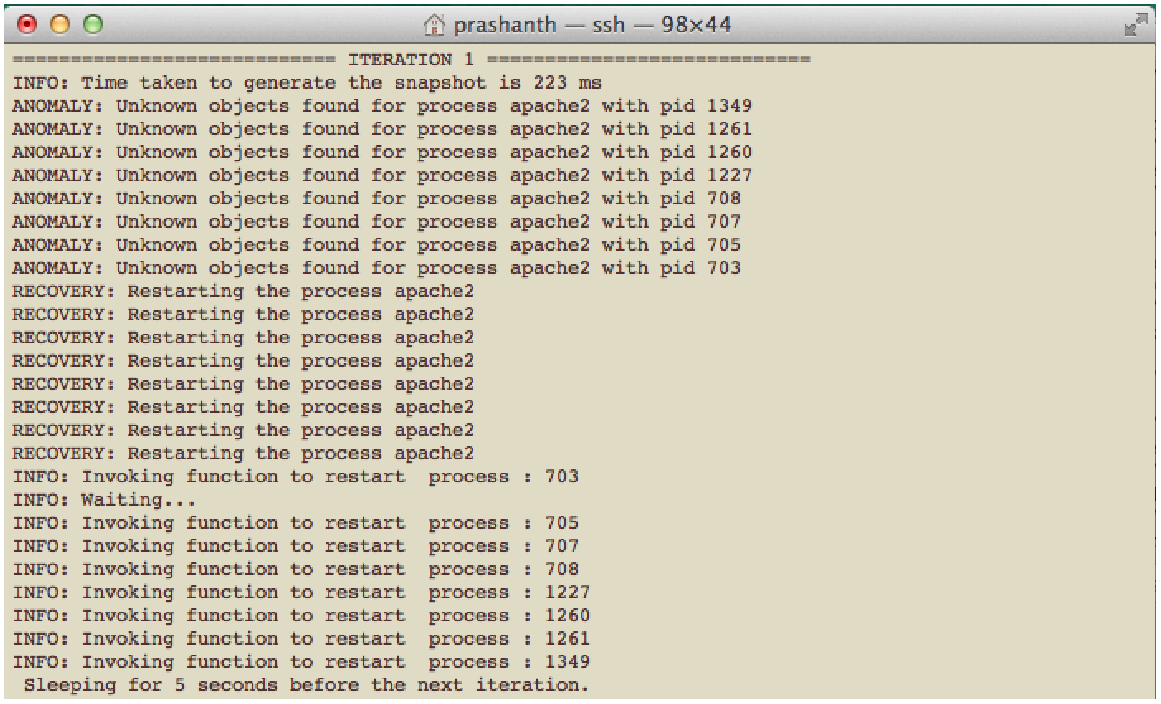


Figure 5.15. Darkleech Test Setup

virtual appliance. Each response received by the clients is checked for malicious iframes injected by the Darkleech module. Since Darkleech does not respond to requests coming from the same IP more than once, we built a script that would change the client's IP and then set the routes on the router appropriately before sending a new request to the web server. The user-agent string in the HTTP request is that of an actual browser (Internet Explorer). Using such a configuration, we were able to achieve an average request-response rate of approximately 45 HTTP request-responses per second. Our anomaly detection and recover tool was configured to run once every 5 seconds.

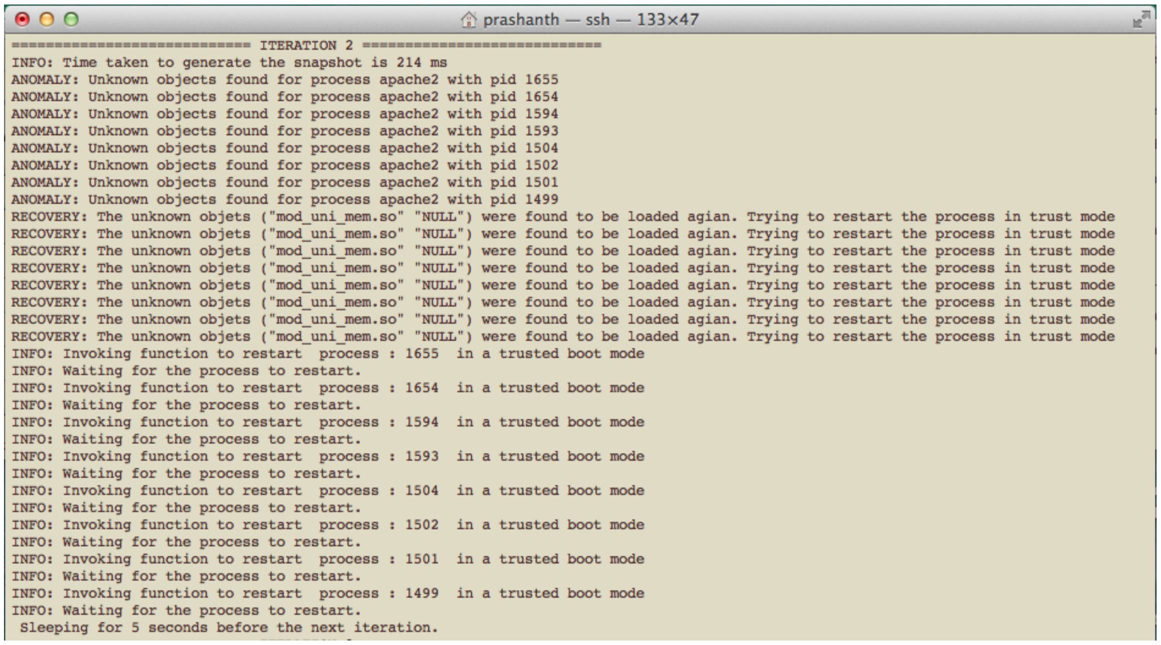
Figures 5.16, 5.17, 5.18, and 5.19 show four iterations of the anomaly detection and recovery system. The graph in Figure 5.20 shows the pattern of HTTP responses from the Darkleech-infected web server over a period of 60 seconds. At time 0, the web server starts and the three hosts start making HTTP requests. Since the Darkleech malware module is loaded at start time, the web server is infected and the web server injects malicious iframes into the responses. This is shown by the initially overlapping dotted and solid lines in the graph.

At time 5, the anomaly detection component is triggered and it detects an unknown object loaded into the Apache process memory. This is reported as an anomaly. The repair action decided is to restart the Apache processes. This is shown in Figure 5.16. The number of HTTP responses drop at time 5 because the Apache processes are terminated and restarted. Since Darkleech modifies the `httpd_conf` file to load the Darkleech module at Apache start time, the unknown module is



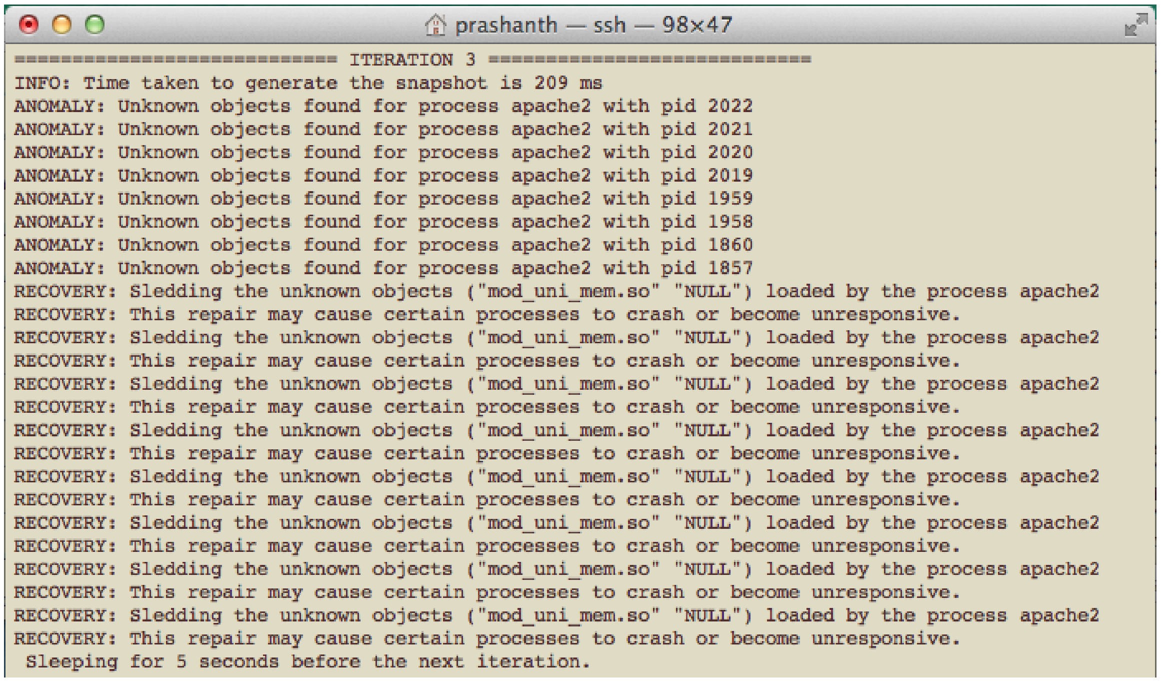
```
============================ ITERATION 1 ============================
INFO: Time taken to generate the snapshot is 223 ms
ANOMALY: Unknown objects found for process apache2 with pid 1349
ANOMALY: Unknown objects found for process apache2 with pid 1261
ANOMALY: Unknown objects found for process apache2 with pid 1260
ANOMALY: Unknown objects found for process apache2 with pid 1227
ANOMALY: Unknown objects found for process apache2 with pid 708
ANOMALY: Unknown objects found for process apache2 with pid 707
ANOMALY: Unknown objects found for process apache2 with pid 705
ANOMALY: Unknown objects found for process apache2 with pid 703
RECOVERY: Restarting the process apache2
RECOVERY: Restarting the process apache2
RECOVERY: Restarting the process apache2
RECOVERY: Restarting the process apache2
RECOVERY: Restarting the process apache2
RECOVERY: Restarting the process apache2
RECOVERY: Restarting the process apache2
RECOVERY: Restarting the process apache2
INFO: Invoking function to restart  process : 703
INFO: Waiting...
INFO: Invoking function to restart  process : 705
INFO: Invoking function to restart  process : 707
INFO: Invoking function to restart  process : 708
INFO: Invoking function to restart  process : 1227
INFO: Invoking function to restart  process : 1260
INFO: Invoking function to restart  process : 1261
INFO: Invoking function to restart  process : 1349
 Sleeping for 5 seconds before the next iteration.
```

Figure 5.16. Darkleech Detection and Repair - Iteration 1



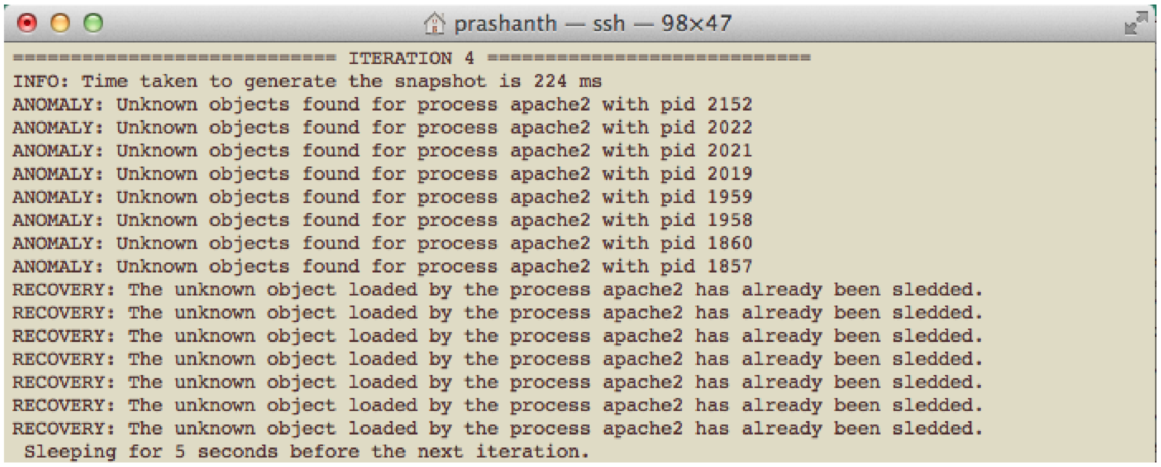
```
============================ ITERATION 2 ============================
INFO: Time taken to generate the snapshot is 214 ms
ANOMALY: Unknown objects found for process apache2 with pid 1655
ANOMALY: Unknown objects found for process apache2 with pid 1654
ANOMALY: Unknown objects found for process apache2 with pid 1594
ANOMALY: Unknown objects found for process apache2 with pid 1593
ANOMALY: Unknown objects found for process apache2 with pid 1504
ANOMALY: Unknown objects found for process apache2 with pid 1502
ANOMALY: Unknown objects found for process apache2 with pid 1501
ANOMALY: Unknown objects found for process apache2 with pid 1499
RECOVERY: The unknown objets ("mod_uni_mem.so" "NULL") were found to be loaded agian. Trying to restart the process in trust mode
RECOVERY: The unknown objets ("mod_uni_mem.so" "NULL") were found to be loaded agian. Trying to restart the process in trust mode
RECOVERY: The unknown objets ("mod_uni_mem.so" "NULL") were found to be loaded agian. Trying to restart the process in trust mode
RECOVERY: The unknown objets ("mod_uni_mem.so" "NULL") were found to be loaded agian. Trying to restart the process in trust mode
RECOVERY: The unknown objets ("mod_uni_mem.so" "NULL") were found to be loaded agian. Trying to restart the process in trust mode
RECOVERY: The unknown objets ("mod_uni_mem.so" "NULL") were found to be loaded agian. Trying to restart the process in trust mode
RECOVERY: The unknown objets ("mod_uni_mem.so" "NULL") were found to be loaded agian. Trying to restart the process in trust mode
RECOVERY: The unknown objets ("mod_uni_mem.so" "NULL") were found to be loaded agian. Trying to restart the process in trust mode
INFO: Invoking function to restart  process : 1655  in a trusted boot mode
INFO: Waiting for the process to restart.
INFO: Invoking function to restart  process : 1654  in a trusted boot mode
INFO: Waiting for the process to restart.
INFO: Invoking function to restart  process : 1594  in a trusted boot mode
INFO: Waiting for the process to restart.
INFO: Invoking function to restart  process : 1593  in a trusted boot mode
INFO: Waiting for the process to restart.
INFO: Invoking function to restart  process : 1504  in a trusted boot mode
INFO: Waiting for the process to restart.
INFO: Invoking function to restart  process : 1502  in a trusted boot mode
INFO: Waiting for the process to restart.
INFO: Invoking function to restart  process : 1501  in a trusted boot mode
INFO: Waiting for the process to restart.
INFO: Invoking function to restart  process : 1499  in a trusted boot mode
INFO: Waiting for the process to restart.
 Sleeping for 5 seconds before the next iteration.
```

Figure 5.17. Darkleech Detection and Repair - Iteration 2



```
============================ ITERATION 3 ============================
INFO: Time taken to generate the snapshot is 209 ms
ANOMALY: Unknown objects found for process apache2 with pid 2022
ANOMALY: Unknown objects found for process apache2 with pid 2021
ANOMALY: Unknown objects found for process apache2 with pid 2020
ANOMALY: Unknown objects found for process apache2 with pid 2019
ANOMALY: Unknown objects found for process apache2 with pid 1959
ANOMALY: Unknown objects found for process apache2 with pid 1958
ANOMALY: Unknown objects found for process apache2 with pid 1860
ANOMALY: Unknown objects found for process apache2 with pid 1857
RECOVERY: Sledding the unknown objects ("mod_uni_mem.so" "NULL") loaded by the process apache2
RECOVERY: This repair may cause certain processes to crash or become unresponsive.
RECOVERY: Sledding the unknown objects ("mod_uni_mem.so" "NULL") loaded by the process apache2
RECOVERY: This repair may cause certain processes to crash or become unresponsive.
RECOVERY: Sledding the unknown objects ("mod_uni_mem.so" "NULL") loaded by the process apache2
RECOVERY: This repair may cause certain processes to crash or become unresponsive.
RECOVERY: Sledding the unknown objects ("mod_uni_mem.so" "NULL") loaded by the process apache2
RECOVERY: This repair may cause certain processes to crash or become unresponsive.
RECOVERY: Sledding the unknown objects ("mod_uni_mem.so" "NULL") loaded by the process apache2
RECOVERY: This repair may cause certain processes to crash or become unresponsive.
RECOVERY: Sledding the unknown objects ("mod_uni_mem.so" "NULL") loaded by the process apache2
RECOVERY: This repair may cause certain processes to crash or become unresponsive.
RECOVERY: Sledding the unknown objects ("mod_uni_mem.so" "NULL") loaded by the process apache2
RECOVERY: This repair may cause certain processes to crash or become unresponsive.
RECOVERY: Sledding the unknown objects ("mod_uni_mem.so" "NULL") loaded by the process apache2
RECOVERY: This repair may cause certain processes to crash or become unresponsive.
 Sleeping for 5 seconds before the next iteration.
```

Figure 5.18. Darkleech Detection and Repair -Iteration 3



```
============================ ITERATION 4 ============================
INFO: Time taken to generate the snapshot is 224 ms
ANOMALY: Unknown objects found for process apache2 with pid 2152
ANOMALY: Unknown objects found for process apache2 with pid 2022
ANOMALY: Unknown objects found for process apache2 with pid 2021
ANOMALY: Unknown objects found for process apache2 with pid 2019
ANOMALY: Unknown objects found for process apache2 with pid 1959
ANOMALY: Unknown objects found for process apache2 with pid 1958
ANOMALY: Unknown objects found for process apache2 with pid 1860
ANOMALY: Unknown objects found for process apache2 with pid 1857
RECOVERY: The unknown object loaded by the process apache2 has already been sledded.
RECOVERY: The unknown object loaded by the process apache2 has already been sledded.
RECOVERY: The unknown object loaded by the process apache2 has already been sledded.
RECOVERY: The unknown object loaded by the process apache2 has already been sledded.
RECOVERY: The unknown object loaded by the process apache2 has already been sledded.
RECOVERY: The unknown object loaded by the process apache2 has already been sledded.
RECOVERY: The unknown object loaded by the process apache2 has already been sledded.
 Sleeping for 5 seconds before the next iteration.
```

Figure 5.19. Darkleech Detection and Repair - Iteration 4

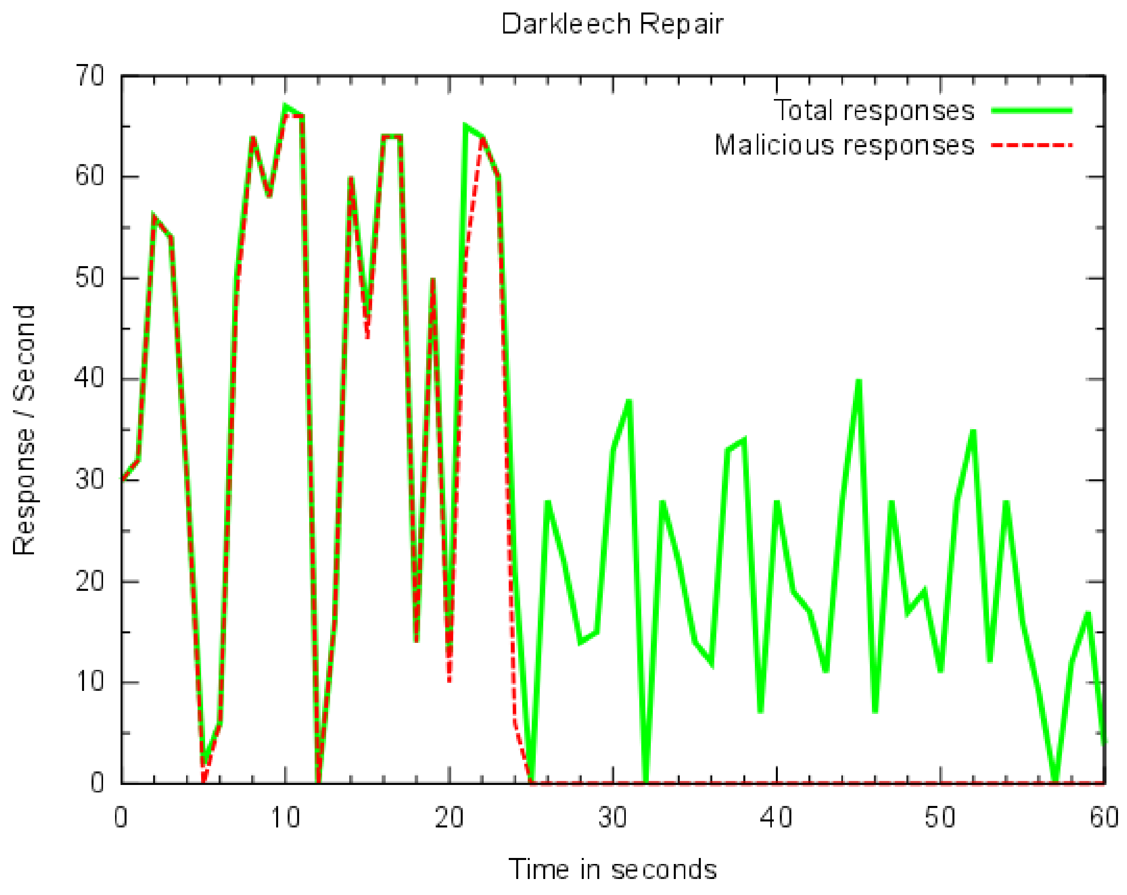


Figure 5.20. HTTP Responses Over Time

reloaded into the Apache process memory. In other words, simply restarting Apache is an ineffective repair.

At time 11, during the second iteration of the anomaly detection system, the unknown module is again detected in the Apache process memory. The recovery component does not repeat the same repair action instead tries a different repair strategy. The recovery action decided this time is to restart the Apache process in a trusted boot environment. This is shown in Figure 5.17 and corresponds to the reduced responses in the graph between time 11 and 17. As the Apache threads were being killed by the repair tool new threads were spawned by the main Apache process. These threads started outside the trusted boot environment created by the repair tool. Hence the unknown objects were found in the next iteration.

At time 22, during the third iteration, the recovery component decides to sled the text segment of the unknown object with `noop` instructions. After the instructions are sledded at time 24, the responses do not have any malicious iframes in them. This is depicted by the dashed line, which stays at zero along the y-axis after time 24. The solid line periodically drops to zero at intervals of about 5 seconds. This is because the anomaly detection system pauses the VA every 5 seconds to capture a snapshot.

The `noop` sled repair is effective because the Apache web server has a modular design. When a web request comes in, it is processed by a set of modules. For each module that is enabled, its handler method looks at the `request_rec` record and decides if the web request has to be processed by it or not. If a web request is processed by a module, the module updates the `response_rec` and forwards it to the next module in the pipeline. The handler has a `void` return type and is not required to manipulate `response_rec`. Thus, each module in the pipeline is independent from every other module. When the Darkleech module is sledded by the repair tool, all the instructions in Darkleech's handler method are replaced by the `noop` instruction. The Darkleech handler executes those `noop` instructions, does not invoke any other functions and simply returns. Hence, no iframes are injected in the responses once the Darkleech module is sledded.

By validating the shared objects loaded in memory of the Apache processes, our anomaly detection system was able to detect the malicious Apache module loaded by Darkleech. By dynamically overwriting the instructions in the text segment of the malicious module, it was able to nullify the malicious actions of Darkleech.

5.5 Summary of Experiments

Table 5.2 summarizes the results of the experiments described in this chapter. We evaluated the anomaly detection and repair system against a kernel exploit kit, a kernel-space rootkit, a user-land rootkit, and an application malware. The anomaly detection component was able to detect all the

Table 5.2. Summary of Experiments

Experiment		Mitigated	Acceptable	Notes
Kernel Exploit Kit for `recvmmsg`		✓	✓	Root shell identified and terminated.
Suterusu	Process Hiding	✓	✓	Process detected and terminated.
	File Access	✓	✓	Detected and stopped.
	Network Access	✓	✓	Detected and stopped.
Azazel	Unauthorized Objects	✓	?	Detected,mitigated but not cleaned.
	`su` exploitation	✓	✓	Detected and privileges restored.
	SSH backdoor	✓	✓	Detected and terminated.
Darkleech application malware		✓	✓	Unknown objected detected and sledded.

anomalies introduced by the malicious software. The recovery component was not able to clear the malicious object loaded into the process memory by Azazel. However, the malicious actions performed by the processes compromised by Azazel were detected and terminated. The recovery component was also successful in mitigating all actions of the other malicious software. Thus, the anomaly detection and recovery system proved to be effective in detecting anomalies in the VA and restoring the VA to an acceptable state.

CHAPTER 6

CONCLUSION AND FUTURE WORK

We have presented the design and a prototype implementation of a system capable of detecting anomalies in a virtual appliance and restoring the virtual appliance to an acceptable state. By making use of virtual machine introspection and expert systems, we demonstrated how an administrator's application-level knowledge can be used to detect anomalies in virtual appliances. Furthermore, by combining expert systems and kernel-based state management techniques, we were able to reason about and apply appropriate recovery actions for each anomaly detected. We demonstrated the effectiveness of our policy-driven anomaly detection and recovery system by testing it against different classes of malicious software: (i) application malware, (ii) kernel rootkits, (iii) userland rootkits, and (iv) kernel exploits. Based on our design and prototype implementation, we have proved the feasibility of a policy-driven anomaly detection and recovery system. Further, based on the results of evaluation of the system against real-world malicious software, we have established the correctness of our thesis statement.

The results of this thesis can be used as the basis for future work in anomaly detection and automated repair of virtual appliances. We have provided a detailed design and prototype implementation of a policy-driven anomaly detection and recovery system. The prototype implementation provides a generic framework that integrates the state gathering, anomaly detection, and recovery components. Each of these could be further individually enhanced to by adding more functionality. For example, new VMI-based state-gathering tools could be added to the snapshot engine, and the recovery component could be enhanced by the addition of new recovery tools. A particularly interesting area of further work is to make the `noop` sledding tool more generic so that it can be used to sled malicious code in applications other than the Apache web server.

In our evaluation we implemented a knowledge store for the anomaly detection component that was tailored to a web server virtual appliance. New knowledge stores can be built and plugged into the framework so that the system supports a wider range of virtual appliances. The process of building a knowledge store is not automated, and it requires the administrator to manually encode the expected state of the system as facts in the knowledge store. This process could be partially automated so that the application-level facts of the system are automatically generated when the system is known to be in a consistent state.

When the anomaly detection and recovery system is scheduled to operate once in every 5 seconds, it causes the VA to be unavailable about 2.79% of the time. This unavailability may be unacceptable for some production environments. The anomaly detection system can be optimized by making use of a dynamic frequency scaling technique. In this technique, the frequency of the snapshotting operation can be varied to tune the overall fraction of time that the VA is unavailable. A large number of anomalies detected in the recent snapshots would result in a high frequency and a small number of anomalies would result in a low frequency.

REFERENCES

[1] Azazel Userland Rootkit. http://blackhatlibrary.net/Azazel. Last Accessed June 9, 2014.

[2] CVE-2014-0038. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0038. Last Accessed June 9, 2014.

[3] LibVMI. https://code.google.com/p/vmitools/. Last Accessed June 9, 2014.

[4] Sucuri Security. http://sucuri.net. Last Accessed June 9, 2014.

[5] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (2003), pp. 164–177.

[6] Chen, P., and Noble, B. When virtual is better than real. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems* (2001), pp. 133–138.

[7] Coppola, M. Suterusu Rootkit. https://github.com/mncoppola/suterusu. Last Accessed June 9, 2014.

[8] Demsky, B., and Rinard, M. Automatic detection and repair of errors in data structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications* (2003), pp. 78–95.

[9] Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A., and Chen, P. M. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (2002), pp. 211–224.

[10] Fu, Y., and Lin, Z. Exterior: using a dual-VM based external shell for guest-OS introspection, configuration, and recovery. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2013), pp. 97–110.

[11] Garfinkel, T., and Rosenblum, M. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of Network and Distributed System Security Symposium* (2003), pp. 191–206.

[12] Giarratano, J. C., and Riley, G. *Expert Systems*, 3rd ed. PWS Publishing Co., 1998.

[13] Grob, S. Timeoutpwn Exploit Kit. https://github.com/saelo/cve-2014-0038/blob/master/timeoutpwn.c. Last Accessed June 9, 2014.

[14] Hofmann, O. S., Dunn, A. M., Kim, S., Roy, I., and Witchel, E. Ensuring operating system kernel integrity with OScK. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), pp. 279–290.

[15] HOWARD, F. Exploring the Blackhole Exploit Kit. http://nakedsecurity.sophos.com/exploring-the-blackhole-exploit-kit/. Last Accessed June 24, 2014.

[16] JIANG, X., WANG, X., AND XU, D. 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* (2007), pp. 128–138.

[17] JOHNSON, D., HIBLER, M., AND EIDE, E. Composable multi-level debugging with Stackdb. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2014), pp. 213–226.

[18] JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Antfarm: tracking processes in a virtual machine environment. In *Proceedings of the 2007 USENIX Annual Technical Conference* (2006), pp. 1–14.

[19] JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. VMM-based hidden process detection and identification using Lycosid. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2008), pp. 91–100.

[20] JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (2005), pp. 91–104.

[21] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. KVM: the linux virtual machine monitor. In *Proceedings of the Linux Symposium* (2007), vol. 1, pp. 225–230.

[22] LANDESMAN, M. Apache Darkleech Compromise. http://blogs.cisco.com/security/apache-darkleech-compromises. Last Accessed June 9, 2014.

[23] LOSCOCCO, P. A., WILSON, P. W., PENDERGRASS, J. A., AND MCDONELL, C. D. Linux kernel integrity measurement using contextual inspection. In *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing* (2007), pp. 21–29.

[24] OSTRAND, T. J., AND WEYUKER, E. J. The distribution of faults in a large industrial software system. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* (2002), pp. 55–64.

[25] OSTRAND, T. J., WEYUKER, E. J., AND BELL, R. M. Where the bugs are. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis* (2004), pp. 86–96.

[26] PAYNE, B., DE CARBONE, M., AND LEE, W. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference* (2007), pp. 385–397.

[27] PTACEK, T. H., AND NEWSHAM, T. N. Insertion, evasion, and denial of service: Eluding network intrusion detection. Tech. rep., DTIC Document, 1998.

[28] RILEY, G. C Language Integrated Production System. http://clipsrules.sourceforge.net. Last Accessed June 9 2014.

[29] SCARFONE, K., AND MELL, P. Guide to intrusion detection and prevention systems. *NIST Special Publication 800* (2007), 94.

[30] Sun, C., He, L., Wang, Q., and Willenborg, R. Simplifying service deployment with virtual appliances. In *Proceedings of the IEEE International Conference on Services Computing* (2008), pp. 265–272.

[31] Verwoerd, T., and Hunt, R. Intrusion detection techniques and approaches. *Computer Communications 25* (2002), 1356 – 1365.

[32] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. An integrated experimental environment for distributed systems and networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (2002), USENIX Association, pp. 255–270.