# Automated Containment of Rootkits Attacks

Arati Baliga, Liviu Iftode
Department of Computer Science
110 Frelinghuysen Road, Piscataway, NJ
Email: {aratib,iftode}@cs.rutgers.edu
Xiaoxin Chen
VMware Inc
3145 Porter Drive, Palo Alto, CA
Email: mchen@vmware.com

*Abstract*— **Rootkit attacks are a serious threat to computer systems. Packaged with other malware such as worms, viruses and spyware, rootkits pose a more potent threat than ever before by allowing malware to evade detection. In the absence of appropriate tools to counter such attacks, compromised machines stay undetected for extended periods of time. Leveraging virtual machine technology, we propose a solution for real-time automated detection and containment of rootkit attacks. We have developed a prototype using VMware Workstation to illustrate the solution. Our analysis and experimental results indicate that this approach can very successfully detect and contain the effects of a large percentage of rootkits found for Linux today. We also demonstrate with an example, how this approach is particularly effective against malware that use rootkits to hide.**

*Index Terms*— **rootkits, stealth malware, intrusion, containment,virtual machines**

## I. Introduction

Despite efforts for decades, software continues to be buggy and vulnerabilities are frequently found [1]. Exploiting a vulnerability gives the attacker access to the system. A root-level exploit provides the attacker with the capability of modifying the operating system kernel and critical system utilities. Since all applications obtain services from the operating system, the data from the applications cannot be trusted once the operating system is compromised. Detecting a compromised system is in itself a challenge as attackers often hide their presence from regular users of the system. Tools used by attackers to hide and spy on the remote system are bundled in the form of kits known as rootkits. In the absence of appropriate detection mechanisms, a rootkit infested system can stay undetected for extended periods of time. Since their first appearance, the level of sophistication of rootkits has increased considerably.

Recent attack trends have begun to bundle rootkits with viruses and worms to help the malware evade detection from anti-virus software running on the machine [2], [3], [4], [5]. Rootkits are also bundled with spyware and adware programs to hide their presence from the user [6], [7]. Networks of such compromised systems controlled by the attacker, are known as botnets. Past few years have witnessed a surge in botnets [9], [10], [11]. Emerging schemes such as pay-per-click and Google Adsense, have added an economic incentive to attackers to hijack and control more machines on the Internet. As a result, writing malware has transitioned from criminal mischief to criminal profiteering [8]. Botnets are used for profitability and criminal purposes, such as installing adware,stealing identities, sending spam mails or launching distributed denial-of-service(DDoS) attacks on other websites, to name a few.

Recent works [12], [13], [14] have proposed methods to automatically detect rootkits. However, simple detection forces the administrator to take the system offline, making the system unavailable for service. With the growing attack trends, this will lead to frequent service outages. We argue that future systems will need to be built with automated mechanisms to counter such attacks. Countering attacks automatically consists of building efficient detection, containment and recovery mechanisms.

In this paper, we propose an approach to automatically detect and contain rootkits by leveraging the virtual machine technology. This approach also efficiently contains the effects of other malware such as viruses, worms and spyware that use rootkits to hide. We focus on countering automated attacks, where the attacker probes the Internet for machines that are vulnerable and has no particular bias towards attacking any specific machine. Towards this end, we assume that the machines are physically secure from tampering. We do not consider insider attacks, where an attacker may have local access to the machine or can exploit any of the DMA devices attached to the system. We assume that the only threat of attack to the system is from the network.

The main contributions of this paper are as follows:

- We propose an approach for automated detection and containment of user level as well as kernel level rootkit attacks and other malware that use rootkits to hide.
- We present a proof of concept prototype, *Paladin*, implemented as part of *VMware Workstation*. Using this prototype, we show that our approach is effective in containing a large percentage of Linux rootkits found today.
- We examine the applicability of our solution to certain Windows specific techniques used by modern rootkits.

The rest of the paper is organized as follows: We describe related work in Section II. In Section III, we provide background information. In Section IV, we discuss our idea of automated detection and containment. In Section V, we describe the design and implementation of the Paladin prototype. Section VI covers evaluation of our prototype. In Section VII, we describe counter attacks, limitations of our approach and facilitate a discussion about Windows rootkits, we discuss future work in section VIII and finally conclude in section IX.

## II. Related Work

We examine related work from the point of view of methodologies used as well as the objectives of the work. We cover work done in the virtual machine context as well as on stand-alone systems.

### *A. Different Approaches, Similar Objectives*

Copilot [12] monitors the operating system to detect kernel rootkit installation by periodically polling kernel memory. Since it resides on a PCI add-in card and is independently connected to a monitoring station, Copilot reliably detects kernel rootkits and cannot easily be compromised by an attacker with root on the system. Petroni et al designed a constraint specification architecture [14] that can check for violations of manually specified constraints in kernel dynamic data. This also is limited to detection alone. Molina et al [29] proposed the approach of using an independent auditor device to detect malicious changes to the file system. This only works when the PCI card used as an auditor and the disk being monitored are connected to the same PCI bus. Tripwire [21] also detects modifications to the file system but can be easily defeated by a kernel rootkit attack as it is a user level program. While these approaches can detect a compromised kernel, they cannot perform any form of containment.

Strider Ghostbuster [13] can detect all hidden files and processes. It uses a *cross view-diff* based approach, which compares the user level view with the kernel level view on the same system and the inside the box view with a clean outside the box view to check for hiding processes and files. The process is offline and time-consuming. This is a good tool for efficient detection but cannot contain attacks in progress. Tools currently used to detect rootkits such as *chkrootkit* and *Samhain*, reside in the kernel and are, therefore, themselves vulnerable to attacks. Grizzard et al [30] address the issue of recovering from rootkits that modify the system call table by replacing the infected copy with a clean copy. This is an offline recovery procedure, which works only for rootkits that modify the system call table.

Recent research in rootkits has proposed a new type of rootkit called Virtual Machine Based Rootkits(VMBR) [31]. Having a secure VMM model for defense, such as the architecture we propose, gives the user an advantage over such rootkits. Since our policies can protect boot scripts from being overwritten, the VMBR inside the Guest OS cannot insert itself below the Guest OS.

### *B. Similar Approaches, Different Objectives*

Logging system calls from the guest operating system to track dependencies between operating system objects has been used before in RFS, Backtracker and Taser [32], [33], [34]. These systems differ in several ways from our system. RFS and Taser [32], [34] perform recovery on the file system that may have been damaged as a result of intrusion or human errors. Both systems perform off-line recovery and aid the administrator by reducing the time to recover. They also rely on external host-based intrusion detection systems (HIDS) or on the users themselves to detect anomalies that indicate that the system is compromised or damaged. Backtracker [33] relies on the administrator to find a suspicious file or process as a detection point. The goal of Backtracker is to show all dependencies, tracking backwards from the point of detection to the first process that was created on the system. This helps the administrator in analyzing how the intruder could have gained access to the system. The analysis is performed offline. Though the tool is helpful in fixing the system, most of this process is largely manual. All the above three systems also perform append only logging, which generates about 1.2-1.9GB of data per day [34], [33].

Our system performs automated detection and does not rely on the administrator or an external HIDS system. Our logging mechanism generates a near-online view of the files and processes on the system and therefore only requires a modest amount of storage. One of the reasons for this is also a difference in our objectives. While the goal of our system is to track relationships between objects, all the above mentioned systems need to store the timeline of events. Janus [35] and kernel hypervisors [36] implement security policies on application by trapping system calls. This is similar to our approach of observing illegal access to files. Introvirt [37] proposes writing vulnerability-specific predicates to provide response to an intrusion. Introvirt requires the vulnerability to be discovered first and a predicate written to provide the fine-grained response. Our approach does not require such human intervention. Previous research has attempted to automatically generate worm signatures [38] and contain worms by sharing attack signatures [39]. These can be considered as having similar goals to our idea of providing automated response but this work does not address rootkit attacks.

## III. Background

### A. Virtual Machine Technology

Today, virtual machines are widely used in servers as well as in desktop environments for running multiple operating systems. In server environments - be it enterprise services, Internet services or data centers - virtualization results in substantial cost savings and ease and efficiency of management, due to server consolidation. With increasingly adversarial Internet traffic, security is a major concern for enterprises as well as lay users. Virtualization provides a secure and robust computing environment (See Section IV). A virtual machine monitor (VMM) is a thin layer of software that runs on the bare hardware or on an existing OS. VMM emulates the underlying hardware in such a way that operating systems can run on top of it without any or little change. VMMs also allow multiple operating systems to run on top of it by virtualizing all resources and efficiently multiplexes them between multiple operating systems. Virtual machine monitor provides isolation and gives good performance guarantees for different operating systems running on the same machine. The performance of the VMM software has improved further due to hardware support built into processors for virtualization [15], [16].

Virtual machines can be classified into two categories: Type I and Type II as shown in Figure 1. Type I virtualization software such as *VMware ESX server* [17] and *Xen* [18], run on the bare machine controlling the physical resources. They provide a virtual interface to operating systems running inside the VMs. Type II virtualization software such as the *VMware Workstation*, uses the hosted architecture [19]. In this case, VMware

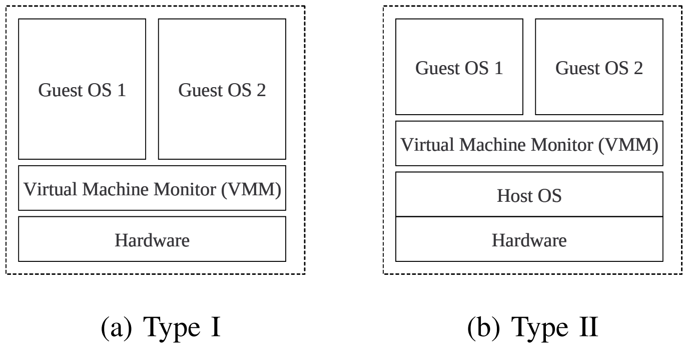

Fig. 1. Virtual Machine Types

is first installed as an application on an existing OS, called the Host OS. The operating system running inside the virtual machine is known as the Guest OS. VMware runs as a process on the Host OS and relies on it to fulfill Guest OS I/O requests. Since the Guest OS runs at a less privileged level on the physical processor, compared to the VMM, the virtual machine monitor has the capability of intercepting events from the Guest OS. *While our prototype is implemented with a Type II virtual machine, it can just easily be implemented using a Type I virtual machine.*

### B. Security Model

Having the Intrusion Detection System (IDS) on the Host OS to monitor the Guest OS is known as virtual machine introspection [20]. From the time this model was first proposed, it has been widely accepted as a secure model for monitoring and intercepting events from the Guest OS. It is extremely difficult for the attacker to compromise the IDS despite having complete control of the Guest OS, as the Guest OS runs at a lower privilege level compared to the Host OS on the physical processor.

Our design is derived from this model and has the following properties.

- **Encapsulation:** The VMM presents a virtual hardware interface to the OS inside the virtual machine. It is nearly impossible for an attacker in the Guest OS to inject instruction stream into the virtualization layer or access resources outside of the emulated virtual hardware.
- **Introspection:** The VMM can inspect the virtual machine's state at instruction level without possible detection by the code running inside the VM. Any host-based IDS suffers from the problem of sharing resources with the OS and can be compromised. On the other hand, network based IDS has to rely on network stream and is required to reassemble fragments of evidence to detect possible intrusion.

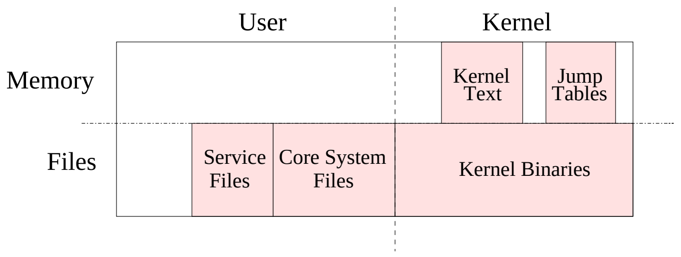

Fig. 2. Protected zones shows files and memory locations protected by access control policies

| File access control policies | |
|---|---|
| /bin | RD_X |
| /sbin | RD_X |
| /boot | RD_X |
| /usr/bin | RD_X |
| /usr/sbin | RD_X |
| /etc/passwd | RD_ONLY (!/usr/bin/passwd) |
| **Memory access control policies** | |
| KERNEL_TEXT | RD_X |
| SYS_CALL_TABLE | RD_X |
| IDT | RD_X |

Fig. 3. Sample Paladin policies

Such approaches are typically less accurate and have limited visibility of the host operating system.

- **Tamper proof:** The VMM runs at a higher privilege on the physical processor compared to the Guest OS. This makes the VMM code inaccessible from the Guest OS except through well-defined interfaces.

The Encapsulation and Tamper proof properties above can be compromised if there is an exploitable flaw in the VMM. However we contend that this is extremely rare, since the VMM is a very thin layer of software with very well-defined interfaces. This makes the VMM a well-tested component where bugs are easily identified and fixed.

## IV. Our Approach

Our system identifies and counters hiding behavior of rootkits using the combination of the following three mechanisms: *Prevention & Detection, Tracking and Containment*.

**a)Prevention and Detection:** This mechanism relies on specification of access control policies for rootkit detection. The policies are tailored to protect memory areas and system files that are a target of rootkit attacks. These are categorized into file access control and memory access control policies as depicted in Figure 2. File access control policies protect the system utilities from being replaced by their trojaned counterparts. Memory access control policies protect the kernel code and data structures from being overwritten in memory, which is a common method utilized by kernel rootkits.

Figure 3 shows a sample policy file used by our system. Several directories are made non-writable. These policies can be tailored as per the system requirement. For example, Figure 3 shows that the program /usr/bin/-passwd is allowed to write into /etc/passwd, which is readonly to other programs. Such policies are easy to specify and are commonly used with other tools like Tripwire [21] and AIDE [22]. Memory access control policies for rootkit detection currently include protecting the kernel system call table, the interrupt descriptor table and the kernel text. These are the three most common hooking places in memory for rootkits. As rootkit authors find newer data structures to manipulate, we can extend the access control policies to protect the newer data regions. Certain legitimate applications may need to write into kernel memory or hook to certain protected areas. Only these applications can be allowed exclusive access to these areas. These applications, in turn, need to be protected on disk by using appropriate access control policies to prevent rootkits from modifying these.

The access control policy file resides outside the Guest OS that is being monitored. It is not visible to the attacker who gains control of the Guest OS. Since attacking the Guest OS does not give the attacker access to the VMM, the VMM can enforce the access control policies without itself being compromised.

**b)Tracking:** The tracking mechanism generates a dependency tree by maintaining parent-child relationships between processes by keeping track of files created by processes. This information is updated from the system call events. The dependency tree is used by the containment algorithm to identify possible malicious processes. Figure 8 shows a sample representation of a dependency tree. Processes are represented by ellipses and files by rectangles. In a process → process relationship, a directed edge from one process to another represents a parent-child relationship. A process→file dependency, shows that the file is created by the process.

We use the following dependency rules for updating the dependency tree:

- Upon process creation, a link is created between the parent and the newly created child process. This is represented as a directed edge from the parent to the child in the dependency tree.
- When a process image is overlayed for execution,

we store the filename from where the process was executed. This filename, shown within the ellipse, represents the process name in the dependency tree.

- When a process exits, if it has created other files or child processes, the process is not deleted from the dependency tree but simply marked for deletion. If the process has not spawned any child processes nor created any files, the process is deleted.
- When a file is created, a link is created from the process to the file.
- When a file is deleted, any process that becomes childless and has been previously marked for deletion is also deleted.

**c)Containment:** Containment is required to stop immediate ongoing damage as soon as a violation of the access control policies is detected. Rootkits are usually bundled with other programs such as keyloggers and backdoors. With the current trend of rootkits being shipped with worms, viruses and spyware, these programs can consist of almost any kind of malware capable of doing immense damage stealthily. When a rootkit accompanies a virus or a worm, it can (and often does) easily disable anti-virus software. This makes even already known worms and viruses effective all over again, as the anti-virus software is stealthily disabled by the rootkit. Containment can be effective in stopping such damage. For example, containment can stop a virus from formatting the hard disk, an attacker from stealing confidential data, or a worm from spreading.

A violation of an access control policy triggers the containment mechanism. The containment algorithm tracks possible malicious processes by referring to the information in the dependency tree. We define a **Process Resident Set (PRS)** as the set of processes that need to be always running in the system. These processes include processes such as *init*, *login* and other system daemons that are usually always present on the system. These programs are listed by the administrator in the form of full pathnames.

Figure 4 shows process P2 performing malicious access. P2 is identified as the offending process. The system automatically identifies the subtree that is considered malicious using the information in the dependency tree. A path is traced back from the malicious process P2 to the root of the dependency tree till a process in the PRS is encountered, all of whose ancestors are also in the PRS. The previous node visited becomes the root of the subtree. In this case, P0 belongs to the PRS and hence, P1 becomes the root of the subtree identified as malicious. All processes identified in this subtree are considered malicious and will be killed by our system.

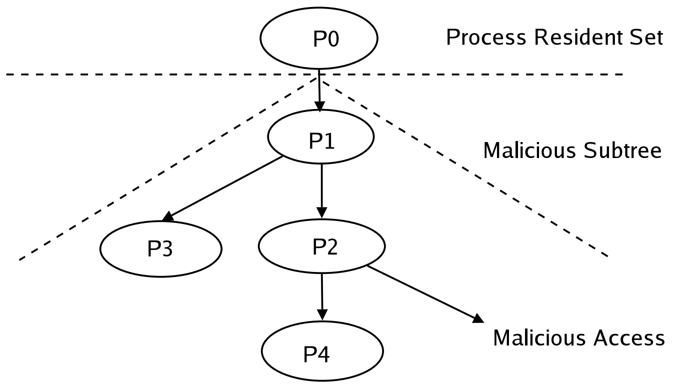

Fig. 4. Automated Containment in Paladin

```
current_node = offending_process;
prev_node = offending_process;
root->parent = NULL;

while (current_node!=NULL) {
    if(!(in_PRS(current_node)))
    {
        /* Current node does not belong to PRS */
        prev_node = current_node;
        current_node = current_node->parent;
    }
    else
    {
        /* Current node belongs to PRS */
        if(all_ancestors_in_PRS(current_node))
        {
            /* Set prev_node as root of
                malicious tree */
            set_root_malicious_subtree(prev_node);

            /* Kills all processes in the
               subtree rooted at prev_node */
            kill_all_processes_in_tree(prev_node);
            break;
        }
        else
        {
            /* Ignore this node and continue
               traversing up to the root */
            prev_node = current_node;
            current_node = current_node->parent;
        }
    }
}
```

Fig. 5. Containment algorithm pseudocode

The pseudo code for the containment algorithm is shown in Figure 5.

## V. Prototype Design and Implementation

In this section, we describe the design approach and the prototype Paladin as shown in Figure 6. Paladin comprises of several components: The modified form of *VMware Workstation*, PaladinApp, the driver and the database. VMApp and the VMM are a part of the *VMware Workstation* software. We added hooks into these to enable communication with PaladinApp,

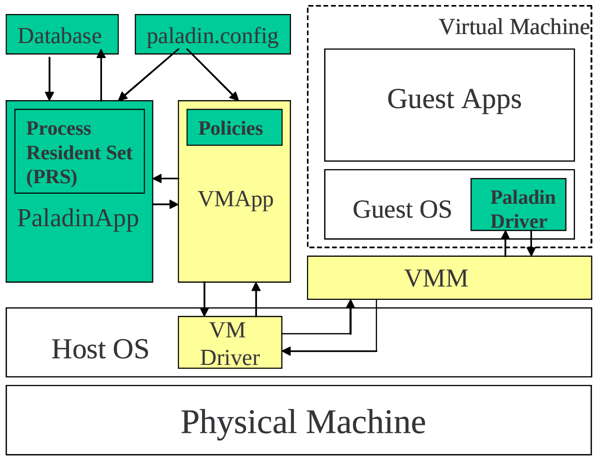

Fig. 6. Paladin Architecture: The components shown in gray are the ones added by us

which is an application process in the host OS. Arrows indicate the communication paths between the different components of the system. The dashed box in the figure represents a virtual machine.

Recall that VMware workstation is a type II virtual machine [19]. It is installed on a host operating system and relies on the host to fulfil I/O requests from the Guest OS. The VMApp appears as a process on the host OS and relies on the host to schedule it. VMApp runs the VMM, which in turn schedules the guest operating systems.

## A. Design Overview

Hooks added in the VMM and VMApp establish a two way communication channel between the VMM and PaladinApp. PaladinApp can register events of interest with the VMM. VMM forwards these events to PaladinApp for processing. A similar channel is established between the VMM and the driver inside the Guest OS. In this case however, commands are always sent from the VMM and actions are carried out by the driver.

In our prototype, VMM forwards file and process related system calls to PaladinApp. These are used by PaladinApp to update the dependency tree, stored in the database. If a violation of a given access control policy is intercepted by the VMM, it notifies the PaladinApp, which in turn initiates the containment procedure. The Paladin prototype works in the following three phases: *Initialization, Normal Operation and Containment.*

*1)* ***Initialization****:* In the initialization phase, the PaladinApp registers file and process related system calls with the VMM. The VMApp and PaladinApp processes read the *paladin.config* file provided by the administrator. This file consists of **a)** The access control policy specification (both file and memory access control) and **b)** The filenames of process belonging to the Process Resident Set (PRS). The file access control policies are stored by the VMApp and used to validate system calls. The memory access control policies are used by the VMM to protect the memory regions. The addresses of the memory regions are obtained by issuing commands to the driver, which in turn performs symbol lookup in the Guest kernel. The driver is a kernel module and has knowledge about Guest OS semantics. The entries in the PRS are used only by the containment algorithm. At the end of the initialization phase, file and memory protection checking is active.

*2)* ***Normal Operation****:* During normal operation, the VMM intercepts system calls and forwards registered system call events to PaladinApp. PaladinApp validates these events against access control policies. It generates the dependency tree from this system call information. VMM has to be aware of the system call interface used by the Guest OS. For the given system call intercepted, VMM provides the system call argument values to the application by accessing the guest memory. Dependencies are inferred by matching entries with exits from system calls for processes and file accesses. Since each process is uniquely identified by a page table during its lifetime, we use the page global directory address stored in the *cr3* register of the x86 virtual CPU as the identifier for processes. Multiple threads within the same process are distinguished using the stack pointer register.

*3)* ***Containment****:* This phase is initiated when a specified access control policy is violated. In both cases, the process performing the malicious access is identified by the VMM and the information is passed to the PaladinApp. In case the illegal access is performed

| Category 1 | |
|---|---|
| Rootkit | Test Set |
| 0x333openssh-3.7.1 | - |
| ark 1.0.1 | ✓ |
| balaur 2.0 | ✓ |
| cbr00tkit | ✓ |
| devNull v0.9 | - |
| dica | ✓ |
| fk v0.4 | ✓ |
| flea | ✓ |
| lrk5 and variants | - |
| sm4ck | ✓ |
| tl0gin | ✓ |
| tnet-tools v1.55 | - |
| torn 6.66 | ✓ |
| trNkit v1.0 | ✓ |
| troier v 1.0 | ✓ |

| Category 2 | |
|---|---|
| Rootkit | Test Set |
| adore-0.42 | ✓ |
| all-root | ✓ |
| linspy2 | ✓ |
| kbd v3 | ✓ |
| kis 0.9 | ✓ |
| knark 2.4.3 | ✓ |
| modhide | ✓ |
| maxty | - |
| override | - |
| phalanx-b6 | - |
| phide | ✓ |
| rial | ✓ |
| rkit 1.01 | ✓ |
| synapsys | ✓ |
| taskigt | ✓ |

| Category 3 | |
|---|---|
| Rootkit | Test Set |
| enyelkm v1.1 | - |
| phantasmagoria | ✓ |
| suckit | ✓ |
| suckit2priv | ✓ |
| superkit | ✓ |

| Category 4 | |
|---|---|
| Rootkit | Test Set |
| backdoor-caca | - |

| | |
|---|---|
| ✓ | Included in test set |
| - | Not included in test set |

Fig. 7. Linux rootkits in the wild categorized by hiding techniques
**category 1:** User-level - Install trojaned system binaries
**category 2:** Kernel-level - Modify the system call table
**category 3:** Kernel-level - Modify kernel text
**category 4:** Kernel-level - Modify interrupt descriptor table (IDT)

from a kernel module, the process inserting the module is considered malicious. The PaladinApp refers to the dependency tree and runs the containment algorithm. It relies on the driver loaded in the guest OS to kill malicious processes. The driver code itself is protected by the VMM and cannot be tampered with by the attacker.

**Protection of the Paladin Driver:** An important aspect of this design is to have the driver inside the Guest OS. This is required for two reasons: a) to retain the VMM as an independent layer (without building Guest OS semantics into it) and b) to have a component in the Guest OS capable of performing certain actions that cannot be carried out effectively from the VMM. However, since the driver exists inside the Guest OS, it is important to protect the driver from being tampered by a kernel rootkit. This is achieved by verifying the code signature during load time against a registered signature and protecting the code pages from writes during execution time. Data pages of the Paladin driver can also be write-protected in a similar fashion during execution time. Whenever there is a write into these pages, a fault is generated in the VMM. The VMM can then verify that the instruction pointer where this write originated from is part of the Paladin driver code page. Given the fact that the driver is called upon only during the initialization and the containment phases and the driver executes only a small set of instructions during this brief period of time, the overhead of protecting the driver pages is reasonable. Since there is no way for the rootkit to interpose between the VMM and the Paladin driver, the driver is considered completely secure.

## B. Implementation

Our prototype was developed for *VMware Workstation*. The host machine and the virtual machine were running the 2.4 Linux kernel. The database used was MySql. The driver is a Linux kernel module (LKM). The driver looks up the *System.Map* file. It finds the symbols for kernel text segment, system call table and interrupt descriptor table (IDT) and returns the physical addresses of these symbols[2].

System call information consists of the system call number, arguments and the virtual CPU registers. When a fork is encountered, in a process, a relationship is created between the parent and the child process. The child process has a different page table root (*cr3*) than the parent, but the same stack and instruction pointer upon exit from the fork system call. Thus a fork system call return with the same stack pointer and instruction pointer but a different *cr3* indicates a return to a child process.

[2]In Linux 2.6, the system call table is not exported. But the driver can still find the address of this table using similar techniques used by rootkits to hook on to this table. Hence this approach works equally well with the 2.6 kernel

Moreover, when comparing stack pointer values, we use the physical address converted from the virtual address to avoid ambiguity due to two identical processes making fork system calls at the same point in the program. This assumes that the OS implementation of process creation uses copy-on-write for the user stack pages.

We have about 2000 lines of code in PaladinApp, about 150 lines of code in the driver and about 300 lines of code added to the *VMware Workstation* software.

## VI. Evaluation

In this section, we describe how we evaluated Paladin, our experimental results and performance measurements.

### A. *Linux Rootkits*

We analyzed 36 significant rootkits available for Linux. Figure 7 lists these rootkits categorized according to the hiding mechanisms used. Rootkits classified as Category 1 use trojaned system binaries to hide from the users. These rootkits are easily portable and can be quickly installed. Category 2-4 rootkits change the kernel to hide themselves and are highly sophisticated compared to their user-level counterparts. Many of the rootkits use the Linux kernel loadable module interface to load their code in the kernel. More sophisticated rootkits write directly in the kernel memory using the /dev/kmem and /dev/mem interfaces and are effective even when the module support is disabled. Category 2 rootkits hook to the system call table, still a widely used technique. Category 3 rootkits change the kernel text and Category 4 rootkits hook to the interrupt descriptor table (IDT). Often, kernel rootkits use user-space programs to perform the actual malicious job of installing backdoors, sniffers and keyloggers. Rootkits bundled with other malware such as viruses, worms and spyware have almost no limits on the damage that they can stealthily do to the system.

### B. *Experimental Methodology*

In all the experiments, we first run Paladin with Prevention & Detection enabled and Containment disabled (**PD mode**) and then with Prevention, Detection & Containment, all enabled (**PDC mode**). This is done to demonstrate experimentally why containment is critical.

As shown in Figure 7, we could successfully test 27 rootkits (indicated by a check mark in column *Test Set*) against Paladin, and Paladin detected and contained all of them. We were unable to get a functional version for the nine others and hence, they were excluded from our test set. An examination of their source code revealed that they use similar techniques as the others within the same category. Therefore, we contend that Paladin will be able to counter these rootkits as well. We use simple policies shown in Figure 3 for our experiments.

We pick one sample rootkit from each category to describe the effects of our mechanism in detail. Additionally, to demonstrate the strength of the containment mechanism in case of fast spreading automated attacks, we evaluate it with a worm called *Lion* that carried a rootkit [23].

### C. *Experimental Results*

*1) Category 1:* The *Tornkit* rootkit trojans the following system binaries : *du, find, ifconfig, in.fingerd, login, ls, netstat, pg, ps, pstree, sz* and *top*. It also installs a log cleaner (*t0rnsb*), a standard linux sniffer (*torns*) and a sniffer log parser (*t0rnp*). The kit creates a hidden directory called */usr/src/.puta*, where it stores all the hidden information.

**PD mode:** In this mode, the system binaries mentioned above are prevented from being overwritten since they violate the specified access control policies. However, this mode cannot prevent running of the sniffer and the log eraser processes.

**PDC mode:** In this mode, all the files are protected from being trojaned. Additionally, sniffers and log erasers are unable to start as the *t0rn* process is killed before starting the sniffer process.

*2) Category 2:* The *Adore* rootkit replaces 14 system call entries in the system call table and redirects them to its own versions. It has a user-space program called *ava*, which it uses to hide files and processes and to execute commands as root. Adore is loaded in the kernel as a linux kernel module *adore.o*

**PD mode:** Prevents the corruption of the system call table. The module continues to be loaded in memory but none of the functions in this module can be invoked from user-space. The program *ava* is still active.

**PDC mode:** User-space program *ava* gets instantly killed preventing process and file hiding.

*3) Category 3:* The *Suckit* rootkit changes machine code in the IDT handler *system_call* and redirects requests to its own private system call table. It works even when LKM support is disabled for the kernel. It uses the /dev/kmem interface to write into the kernel. SuckIt is a user process that exploits this interface to find addresses of kernel symbols. It installs versions of its doctored system calls, which get executed instead of the original system calls. It very effectively hides itself using this method. It also installs a backdoor on the system that listens to connections.

**PD mode:** In this mode, the kernel text is prevented from being corrupted. This does not prevent the backdoor from running.

**PDC mode:** In this mode, the backdoor process is killed as well, preventing remote access to the system.

*4) Category 4:* Since we did not find code for a rootkit in this category, we wrote a simple kernel module that tries to overwrite the IDT entry. This illegal access is successfully detected and prevented by Paladin. Hence, we contend that Paladin will be able to successfully counter rootkits using these hiding mechanisms as well.

*5) Stealth Worm that uses rootkit:* *Lion* is a notorious stealth worm [23] that spread very quickly and evaded detection for a long time due to the presence of a rootkit. It installs the *t0rn* rootkit to hide its files. Lion worm has several variants including some without the rookit. Since we could not find the variant of the Lion worm with the rootkit, we created a home-grown version of this worm. We combined the variant available on the Internet that did not contain the rootkit and modified it to include the rootkit.

Lion affects DNS servers that have the BIND TSIG vulnerability [24]. Figure 8 shows the Lion worm in action. Once the worm has gained access to a machine, it scans for vulnerable hosts with class B Internet addresses. The program *pscan* is used to scan the network while *randb* generates random class B network addresses. If the worm finds a machine with a given IP address, it checks if the machine is susceptible to the BIND attack. If the target machine has a vulnerable version of BIND running, it uses the vulnerability to get root privileges on the system and continue propagating. It installs the *t0rn* rootkit to hide the compromise. The files *scan.sh* and *hack.sh* execute the worm algorithm. *getip.sh* tries to find more victims while *1i0n.sh* and *star.sh* are the controlling processes [25]. The worm finds all the files named *index.html* present on the system and defaces the pages by replacing it with its own version.

**PD-mode:** In this mode, only the system binaries are prevented from being corrupted. Malicious access is detected when the rootkit is activated and tries to overwrite the system binary */bin/ps*. Since the prevention part stops the corruption of the binaries, the hiding behavior is effectively disabled. The worm activities are visible to the administrator. However, these activities continue to occur. Lion defaces all the *index.html* files found on the machine. It also tries to propagate to other hosts running the vulnerable version of the BIND service.

**PDC-mode:** In this mode, as soon as the malicious access is detected, Paladin kills all the other processes identified as part of the malicious subtree as shown in the figure. This stops the worm from defacing the *index.html* pages and propagating to other hosts. Before the rootkit springs into action and tries to overwrite the *ps* binary, the *lion.sh* defaces couple of *index.html* files on the system that Paladin cannot stop. However it is useful in preventing further damage to the system.

**False Positives and False Negatives:** There were no false negatives during our experiments. All 27 rootkits and the Lion worm were effectively detected and contained by Paladin. We used the virtual machine as a regular workstation with Paladin enabled for a duration of one week. We also installed and uninstalled several device drivers. We did not encounter any false positives during these tests.

***LKM handling:*** LKMs are handled in a different fashion. On module insertion, the VMM verifies the integrity of the LKM by comparing the code signature at runtime with a registered signature. If the LKM needs to hook into protected memory areas, the memory protection is automatically disabled by the VMM and re-enabled after the LKM is loaded, provided the module is an authorized module. One such LKM is anti-virus software that usually hooks to the open system call to scan for virus patterns on every file open call.

***Applications Mapping Kernel Memory:*** Genuine applications mapping kernel memory do not change kernel code or crucial data structures such as the system call table. Hence, mapping other parts of kernel memory does not pose any problems to Paladin, as those regions are not protected. If a malicious application tries to modify the protected regions, Paladin will prevent the modification.

## *D. Performance*

To test the overhead of Paladin, we performed two system call intensive tasks. First, we copied a large set of files from one directory to another. The second task was to compile the Linux kernel. We measured the time taken for both these tasks with and without Paladin support. Table II shows the performance overhead incurred by applications that run with and without Paladin prototype. Paladin adds about 12% overhead to the execution time for applications in the Guest OS.

Table I shows the per system call overhead for the four most common file and process related system calls. Paladin incurs relatively larger overhead for open and fork system calls. This is due to the fact that the application performs bookkeeping and matching of parent/child system calls respectively.

## *E. Dependency Tree Size*

The dependency tree stores information about processes, files and relationships between these objects.

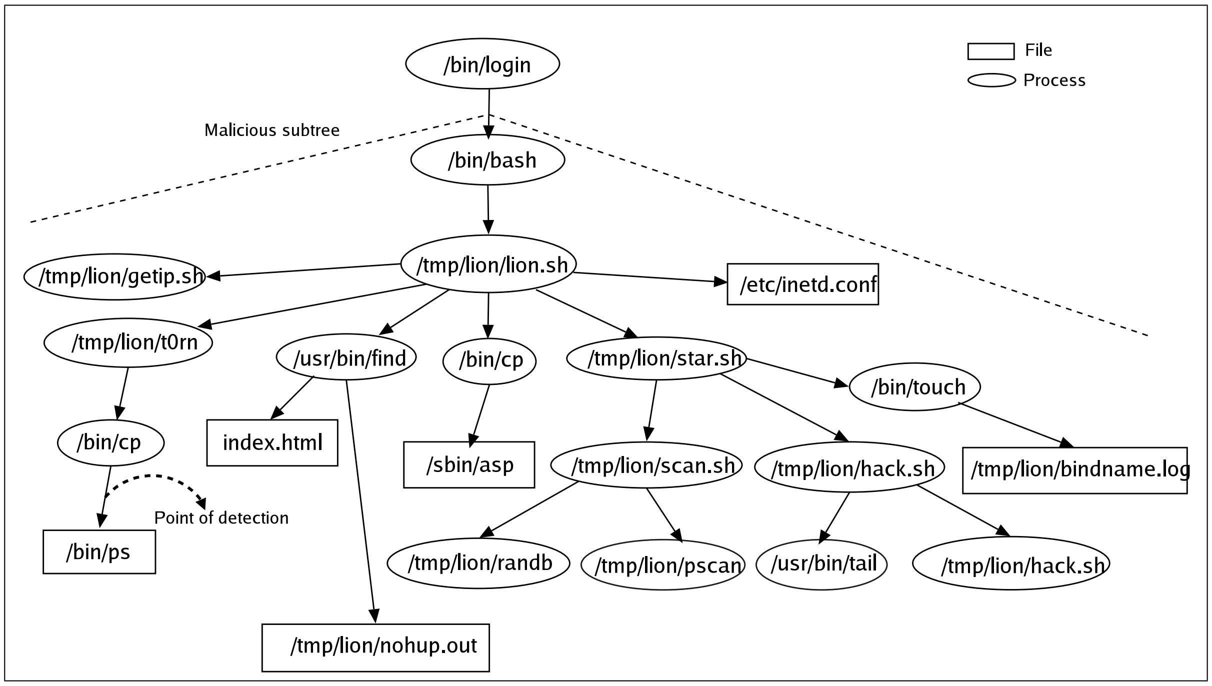

Fig. 8. Dependency tree showing the Lion worm attack

When a new process is created, a new entry is made into the database. When the process exits, if it has not created a new file, it is deleted from the database. A similar approach is employed for files. When a file is created, an entry is made for the file and when it is deleted, the cleanup procedure deletes entries for the file and all the other processes that do not have a role to play in the dependency tree are deleted as well. This pruning procedure ensures that the number of objects in the database is small and storage requirements are modest.

## VII. Discussion

In this section, we discuss counter attacks on Paladin, limitations of our solution and techniques used by Windows rootkits.

| | Paladin | |
|---|---|---|
| | Disabled | Enabled |
| fork | 1.5 $\mu$s | 3.5 $\mu$s |
| exit | 1.5 $\mu$s | 1.6 $\mu$s |
| open | 0.8 $\mu$s | 1.5 $\mu$s |
| close | 0.5 $\mu$s | 0.7 $\mu$s |

TABLE I
Performance of individual system calls

| | Paladin | |
|---|---|---|
| | Disabled | Enabled |
| File Copy | 7m 29s | 8m 30s |
| Kernel Compilation | 53m 3s | 56m 7s |

TABLE II
Paladin Performance Measurements

### A. *Counter Attacks*

Counter attacks are discussed assuming that the attacker has complete knowledge of the defense techniques used by our system.

*1) Multiple Control Processes:* An immediate attack that comes to mind is the use of multiple control processes to carry out the attack. Here, the controlling process for the hiding part is separated from the controlling process that performs other malware activities (non-hiding). This is shown in Figure 9. In this figure, P1 is the controlling process that performs the hiding. P1 may spawn other processes or write into the kernel directly. P3 is the controlling process for carrying out other activities, such as installing keyloggers, scanning network packets, sending passwords etc. Here, since P1 and P3 share a parent P0 (this process could be *sshd* for example), which resides in the PRS, our containment mechanism cannot automatically link the process P3 and its children to P1. The hiding processes, P1 and its children will be killed, while P3 and its children will continue to run.

While this attack cannot be contained by Paladin completely, it exposes the attack to other anti-malware programs running on the system. This defeats the attacker's incentive of carrying out such an attack as it is reduced to launching an attack without the use of a rootkit. Hence, our prototype works in a complementary fashion with the existing anti-virus tools.

*2) Overwriting Disk Blocks:* The attacker can directly attempt to change system binaries by overwriting disk blocks. Our approach tracks processes using the system call interface to access files. While this is generally true for most processes, it is possible for an attacker to perform a write directly to disk blocks. We can handle this by disabling raw writes to disks by augmenting the access control policies. A more effective solution is to use other approaches such as storing data in a separate data VM [26] to force file access through a well-defined interface.

*3) In-memory Corruption of Resident Processes:* Rookits may be able to backdoor resident processes by overwriting code in memory. This attack is very hard to carry out on Linux due to the absence of APIs to perform such actions. We can however, defend against this attack by simply protecting the code pages of these processes from the VMM.

## B. Limitations

Though our system manages to detect, prevent and contain several rootkit attacks, it does suffer from some limitations.

*1) Killing Legitimate Processes:* Our containment algorithm kills all processes inside the malicious subtree. This may involve other genuine applications run by the user, may be undesirable in certain circumstances. We however contend that this is a better option than letting the malware progress with damaging the system, which can cause the owner financial or legal distress.

*2) Accidental Modifications:* It is possible that access-control policies may be accidentally violated by the user. This will result in killing the user's processes including the login shell. Considering the fact that users seldom accidentally overwrite binaries in the system directories, this is a minor issue.

*3) System Upgrades:* Actions such as installation or upgrade of software inside the Guest OS, which may result in the modification of protected files or addition of files to protected directories, have to be preceded with changing the access control policy temporarily. Otherwise, our system will treat them like an attack. While this exposes a small window of vulnerability to the attacker, it is hard to exploit because the attacker has no way of determining this from within the Guest OS. The policy file resides on the Host OS and is not accessible to an attacker who has gained control of the Guest OS.

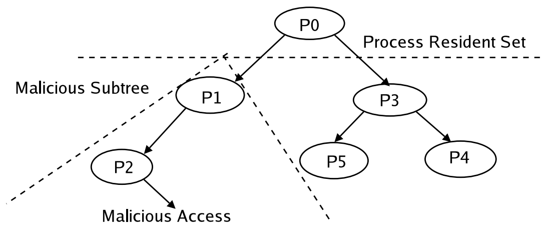

Fig. 9. Multiple control processes

*4) Other Types of Stealthy Behavior:* Other types of stealthy behavior may include modifying the user's environment variables, so that the user executes corrupted binaries without his knowledge. Though this is stealthy behavior, it does not strictly fall under the umbrella of the hiding characteristics unique to rootkits. In this case, the corrupted binaries are visible and can be detected by the user by installing other tools like Tripwire and AIDE.

## C. Windows Rootkits

While our prototype was designed for the Linux OS, it can just as easily work against Windows rootkits. However, there exist some rootkits for Windows that exploit Windows specific techniques. In this section, we examine the challenges posed by Windows rootkits and the applicability of our solution to them.

The Windows operating system is designed to be very modular and extensible. This is an advantage for application developers but at the same time, rootkit writers can exploit these features to their advantage. Windows provides the OpenProcess API, where an application can retrieve a handle to another process running with the same privilege level. This API also allows a process to create a remote thread in another process. These features are usually used for process monitoring and debugging in Windows applications. Rootkits use these features to inject their own code into a remote process. Windows has a DLL injection feature where a process can inject a DLL in the process space of another process. Windows also provides Auto-Start Extensibility Points (ASEPs), which applications can hook on to. Most of the ASEPs reside in the registry. As classified in [27], ASEPs are of four types . (i) *ASEPs that start a new process* - allows processes to be started automatically on system startup. (ii) *ASEPs that hook system processes* - allows a DLL to be loaded into a system process. (iii) *ASEPs that load drivers* - allows loading of drivers and (iv) *ASEPs that* *hook multiple processes* - allows a DLL to be loaded into every process that links with a particular DLL. Rootkits use these hooks to automatically be loaded into process memory and auto-startup after a system reboot.

Windows rootkits can be classified into user and kernel level rootkits. We discuss the different techniques used by these rootkits in the sections below.

*1) Windows User-Level Rootkits:* Windows user-level rootkits take advantage of the ASEPs and code injection techniques instead of overwriting trojaned binaries on disk. Several Windows Enumeration API's are available as dynamically linked libraries (DLL). These dll's are linked by all user programs that use the Windows API. By intercepting calls to these dlls, either by changing the per-process Import Address Table (IAT) (*Urbin and Mestring rootkits*) or by directly changing the in-memory API's (*Vanquish rootkit*) [13] or the Export Address Table (EAT), the attacker can hide her own files and processes successfully. The more powerful rootkits like *Aphex* and *Hacker Defender* introduce an API detour. They modify the return address on the stack in such a way that they get called on the return path from the API call and can alter the result set. Detecting these rootkits in real-time is extremely hard due to the fact that their hooking behavior is identical to the behavior of legitimate Windows applications. We believe that our prototype can be extended to tackle these attacks by adding hooks to those system calls that attach DLLs. We would also need a set of policies that specify the list of programs that are allowed to load the specific DLLs.

*2) Windows Kernel-Level Rootkits:* The kernel rootkits primarily use two techniques - Modifying the System Service Table (SST) or the IDT and Direct Kernel Object Manipulation (DKOM). The first category of rootkits, which modify the SST or IDT, is analogous to those modifying the system call table and the interrupt descriptor table on Linux. We can counter this class of rootkits effectively using our approach. The second category is very hard to detect as it modifies kernel objects directly. The FU rootkit [28] is a good example of this type of rootkit. FU removes its process entry from the process list maintained by Windows. This hides the process from tools like Task Manager, which refer to this list to enumerate processes running on the system. The rootkit process still gets scheduled, as the scheduler refers to a different list for scheduling. Our approach can contain this type of attack if we extend our detection mechanism to specify constraints to observe discrepancies in the two lists as discussed in [14].

## VIII. Future Work

As future work, we plan to implement a prototype of Paladin for Windows. Since Windows rootkits use a variation of hiding techniques, which are not exactly identical to the ones used on Linux, it would be interesting to adapt Paladin to this new platform. We also plan to a VM based honeypot infrastructure to study the techniques used by latest rootkit attacks and test the efficacy of our system in automatically dealing with such attacks.

## IX. Conclusion

Today, rootkits and rootkit carrying malware pose a serious threat to computer systems. We have designed a framework to automatically detect and contain such attacks. Our experimental results prove that this framework can successfully detect and contain the effects of almost all rootkits for Linux, that we could find in the real world. Since our solution leverages virtual machine technology, which is used on desktop machines and hosting centers for server consolidation, this framework can be used to secure a large variety of real systems.

## References

[1] "US-Cert vulnerability notes database," http://www.kb.cert.org/vuls/.

[2] Alexey Monastyrsky, Konstantin Sapronov, and Yury Mashevsky, "Rootkits and how to combat them," http://www.viruslist.com/en/analysis?pubid=168740859.

[3] "Rootkit-armed worm attacking aim," http://www.techweb.com/wire/security/172901356.

[4] "Santa im worm installs rootkit payload," http://www.eweek.com/article2/0,1895,1904112,00.asp.

[5] "Fresh bagels offer baked-in rootkits," http://www.malwarehelp.org/news/article-3134.html.

[6] "Spyware danger meets rootkit stealth," http://www.eweek.com/article2/0,1895,1829744,00.asp.

[7] "Elitebar-a," http://www.f-secure.com/v-descs/elitebar.shtml.

[8] "Hackers write spyware for cash, not fame," http://www.informationweek.com/story/showArticle.jhtml?articleID=160403715.

[9] "California man charged with botnet offenses," http://www.eweek.com/article2/0,1895,1881621,00.asp.

[10] "Doj indicts hacker for hospital botnet attack," http://www.eweek.com/article2/0,1895,1925456,00.asp.

[11] "Alleged botnet crimes trigger arrests on two continents," http://abcnews.go.com/Technology/PCWorld/story?id=1314632.

[12] Nick L. Petroni Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh, "Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor.," in *Proceedings of the 13th USENIX Security Symposium (Security'04)*, 2004, Pages 179-194.

[13] Doug Beck, Binh Vo, and Chad Verbowski, "Detecting Stealth Software with Strider Ghostbuster," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, 2005, Pages 368-377.

[14] Jr. Nick L. Petroni and William A. Arbaugh and Timothy Fraser and Aaron Walters, "An Architecture for Specification-based Detection of Semantic Integrity Violations in Kernel Dynamic Data," in *Proceedings of the 15th USENIX Security Symposium (Security'06)*, 2006,Page 20.

[15] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C.M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith, "Intel Virtualization Technology," *IEEE Computer*, 2005.

[16] "AMD Pacifica Virtualization Technology," http://enterprise.amd.com/Downloads/Pacifica_en.pdf.

[17] Carl A. Waldspurger, "Memory Resource Management in VMware ESX Server," *SIGOPS Operating Systems Review*, 2002, Pages 181-194.

[18] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, "Xen and the Art of Virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003, Pages 164-177.

[19] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim, "Virtualizing I/O Devices on VMware Workstation's hosted Virtual Machine Monitor," in *Proceedings of the 2001 USENIX Symposium (USENIX'01)*, 2001, Pages 1-14.

[20] Tal Garfinkel and Mendel Rosenblum, "A Virtual Machine Introspection based Architecture for Intrusion Detection," in *Proceedings of the 10th Network and Distributed Systems Security Symposium (NDSS'03)*, 2003.

[21] Gene H. Kim and Eugene H. Spafford, "The Design and Implementation of Tripwire: a File System Integrity Checker," in *Proceedings of the 4th ACM Conference on Computer and Communications Security (CCS'94)*, 1994, Pages 18-29.

[22] "Advanced intrusion detection environment," http://sourceforge.net/projects/aide.

[23] "Lion worm attack advisory," http://www.ciac.org/ciac/bulletins/l-064.shtml.

[24] "Bind tsig vulnerability," http://www.sans.org/resources/idfaq/tsig.php.

[25] Dan Ellis, "Worm anatomy and model," in *Proceedings of the 2003 ACM Workshop on Rapid Malcode (WORM'03)*, 2003, Pages 42-50.

[26] Xin Zhao, Kevin Borders, and Atul Prakash, "Towards Protecting Sensitive Files in a Compromised System," in *Proceedings of the 3rd International IEEE Security in Storage Workshop (SiSW'05)*, 2005, Pages 21-28.

[27] Yi-Min Wang, Roussi Roussev, Chad Verbowski, Aaron Johnson, Ming-Wei Wu, Yennun Huang, and Sy-Yen Kuo, "Gatekeeper: Monitoring Auto-Start Extensibility Points (aseps) for Spyware Management.," in *Proceedings of the 18th USENIX Large Installation System Administration Conference (LISA'04)*, 2004, Pages 33-46.

[28] "Fu rootkit," http://www.rootkit.com/project.php?id=12.

[29] Jesus Molina and William A. Arbaugh, "Using Independent Auditors as Intrusion Detection Systems," in *Proceedings of the 4th International Conference on Information and Communications Security (ICICS'02)*, 2002, Pages 291-302.

[30] Julian B. Grizzard, John G. Levine, and Henry L. Owen, "Reestablishing Trust in Compromised Systems: Recovering from Rootkits that trojan the System Call Table.," in *Proceedings of the 9th European Symposium On Research In Computer Security (ESORICS'04)*, 2004, Pages 369-384.

[31] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch, "Subvirt: Implementing Malware with Virtual Machines," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, 2006, Pages 314-327.

[32] Ningning Zhu and Tzi-cker Chiueh, "Design, Implementation, and Evaluation of Repairable File Service," in *Proceedings of 2003 Dependable Systems and Networks (DSN'03)*, 2003, Pages 217-226.

[33] Samuel T. King and Peter M. Chen, "Backtracking Intrusions," in *Proceedings of the 19th ACM symposium on Operating Systems Principles (SOSP'03)*, 2003, Pages 51-76.

[34] Kamran Farhadi Zheng Li Ashvin Goel, Kenneth Po and Eyalde Lara, "The Taser Intrusion Recovery System," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, 2005, Pages 163-176.

[35] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer, "A Secure Environment for Untrusted Helper Applications," in *Proceedings of the Usenix Security Symposium (Security'96)*, 1996.

[36] T. Mitchem, R. Lu, and R. O'Brian, "Using Kernel Hypervisors to Secure Applications," in *Proceedings of 13th Annual Computer Security Applications Conference (ACSAC'97)*, 1997, Page 175.

[37] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen, "Detecting Past and Present Intrusions through Vulnerability-specific Predicates," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, 2005, Pages 91-104.

[38] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage, "Automated Worm Fingerprinting," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*, 2004, Page 4.

[39] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham, "Vigilante: End-to-End Containment of Internet Worms," *SIGOPS Operating Systems Review*, 2005, Pages 133-147.