Paladin: Automated Detection and Containment of Rootkit Attacks

Arati Baliga[1], Xiaoxin Chen[2], and Liviu Iftode[1]

[1] Department of Computer Science
Rutgers University
110 Frelinghuysen Road, Piscataway, New Jersey
{aratib,iftode}@cs.rutgers.edu
[2] VMware Inc
3145 Porter Driver, Palo Alto, California
mchen@vmware.com

Abstract. Rootkit attacks are a serious threat to computer systems. Packaged with other malware like worms, viruses and spyware, rootkits pose a more potent threat than ever before by allowing the 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 for a Linux virtual machine using *VMware Workstation* to illustrate the solution. Our analysis and experimental results indicate that we can successfully detect and contain the effects of almost all rootkits found for Linux today. We also demonstrate with an example, how this approach is particularly effective against malware that use rootkits to hide.

Keywords: rootkits, intrusion detection, containment, stealthy malware

1 Introduction

Despite efforts for decades, software today still continues to be buggy and vulnerabilities are frequently found [1]. Exploiting a vulnerability not only gives an attacker access to the system, it can also allow the attacker to modify the operating system kernel and important system utilities compromising the integrity of the system. Since all applications obtain services from the operating system, the data from the applications cannot be trusted once such a compromise takes place. Detecting a compromise is in itself a challenge as attackers have mastered the art of hiding themselves and wiping all tracks of how they got access to the system in the first place. Attackers have conveniently bundled tools to hide and spy on the remote system, into software packages known as rootkits. Rootkits are a group of stealth malware, which in the absence of appropriate detection systems, can remain on a system undetected for extended periods of time. Since their first appearance, the level of sophistication of rootkits has increased considerably. Once the rootkit takes control of the machine, it is extremely hard to determine what is compromised or which files are installed by the attacker. This is because the very core of the system, is modified to hide the attacker's presence, and can no longer be trusted.

Rootkits are now being bundled with viruses or worms to help the malware evade detection [2–5] from anti-virus software running on the machine. Rootkits are also bundled with spyware and adware programs to hide their presence [6, 7] from the user. Schemes like 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]. Past few years have witnessed a surge in botnets [9–11]. Botnets are network of compromised PCs, also known as zombies that can be controlled by the attacker. Botnets are used for profitability purposes like installing adware or employed in criminal activities like identity theft or launching distributed denial-of-service(DDoS) attacks on other websites. According to a recent survey [12] conducted by the FBI across 2066 organizations, 83.7% of the organizations were affected by worms, viruses and trojans in a period of one year (2005), while 79% of them were affected by spyware.

Most rootkit detection systems in practice today are neither complete nor tamper-proof [13–15]. Recent works [16, 17] have proposed methods to automatically detect rootkits. However, simple detection just forces the administrator to take the system offline making the system unavailable for service. The administrator has to then examine the system and find the type of attack by observing trails left behind by the attacker, if any, or detect compromise by comparing the infected view of the compromised system with an uninfected view of the system [17]. Apart from being an arduous task, this process requires several man hours to reestablish faith in the compromised system. With the growing attack trends, this may no longer be a feasible solution. We argue that future systems will need to be built with automated mechanisms to counter such attacks. Countering attacks automatically, consists of building efficient 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 like worms and spyware that use rootkits to hide. This research is aimed towards 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. We assume that the only threat of attack to the system is from the network.

The main contribution of this paper is twofold:
1. 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.

2. We present a proof of concept prototype, *Paladin*,[1] implemented as part of *VMware Workstation.* Using this prototype, we show that our approach is effective in containing almost all rootkits found for Linux in the wild today.

Rest of the paper is organized as follows: Section II provides background information. Section III discusses our idea of automated defense. In Section IV, we describe the design and implementation of the Paladin prototype. Section V covers evaluation of our prototype. In Section VI we describe counter attacks, limitations of our approach and facilitate a discussion about Windows rootkits. We describe related work in Section VII and finally conclude with summary and future work in section VIII.

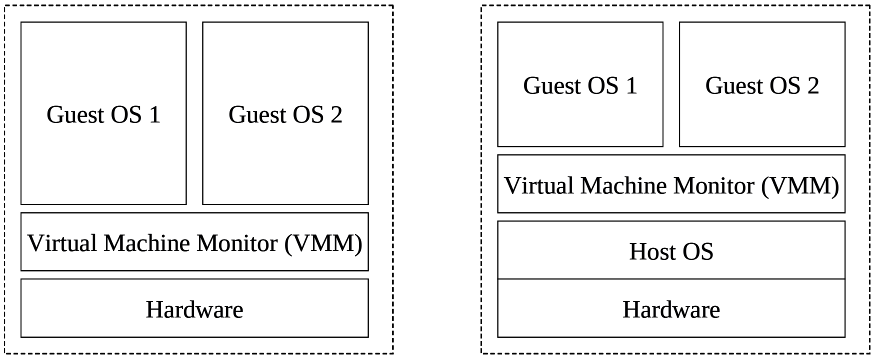
2 Background

2.1 Virtual Machine Technology

Virtual machines today, 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 and virtualization provides a secure and robust computing environment (See section 2.2). 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 then efficiently multiplexing it 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 is expected to improve further in the near future due to hardware support built into processors for virtualization [18, 19].

Virtual machines can be classified into two categories: Type I and Type II as shown in Fig 1. Type I virtualization software like *VMware ESX server* [20] and *Xen* [21] 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 like the *VMware Workstation* uses the hosted architecture [22]. In this case, VMware 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.

[1] A paladin is the prototypical "knight in shining armour," a hero of sterling character and courage, who rights wrongs and defends the weak and oppressed. (http://en.wikipedia.org/wiki/Paladin)



(a) Type I (b) Type II

Fig. 1. Virtual Machine Types

2.2 Security Model

Having the Intrusion Detection System (IDS) on the Host OS to monitor the Guest OS is known as virtual machine introspection [23]. This model, since it was first proposed, has been widely accepted as a secure, tamper-proof model for monitoring and intercepting events from the Guest OS. Our design is derived from this model and has the following properties.

- **Encapsulation:** The VMM presents a virtual hardware interface to the OS in the VM. There's no means to inject instruction stream into the virtualization layer. It's also impossible to 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. Network based IDS, on the other hand has to rely on network stream and is required to reassemble fragments of evidence to detect possible intrusion. Such approaches are typically less accurate and have limited visibility of the host operating system.
- **Tamper proof:** Although the VMM can infer a lot of the system's state by inspecting the runtime events at the hardware level, it is still desirable to use driver in the guest OS to gather information and perform remediation. The VMM can verify the integrity of this driver by comparing the code signature at runtime with a registered signature when the driver is installed into the guest.

3 Automated Defense

Our automated defense 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: We define the notion of *protected zones*. These zones are guarded and illegal access to them is *prevented*. Fig. 2 shows the protected zones as defined by our approach. Protected zones can be subdivided into *Memory Protected Zone* (**MPZ**) and *File Protected Zone* (**FPZ**). The memory image of the kernel and the various jump tables are a part of the MPZ. Some user space processes that are resident in the system referred to as the *Process Resident Set (PRS)* can also be made part of the MPZ. The MPZ is set non-writable. Any attempt to write into the MPZ will trigger an alarm with our detection mechanism and the write is prevented. The FPZ consists of the system files that need to be protected from being modified by an attacker. These can consist of several types of files - system executables, shared libraries, configuration files, log files, data files and service specific files. System utilities are frequent targets of rootkits that try to hide by installing trojaned system binaries. The FPZ is protected by allowing the administrator to create customized access control policies for the files and directories in the FPZ. Tools like Tripwire and AIDE, which are widely used, require similar customization at the file and directory level. Fig. 3 shows the sample policies used with our prototype. The policies specify directories that are non-writable (read & execute only) and the */etc/passwd* file as read-only. Any process trying to perform illegal access into the FPZ is also considered malicious and the actual operation to the file is prevented. Detection mechanism raises an alarm when such an illegal access occurs.

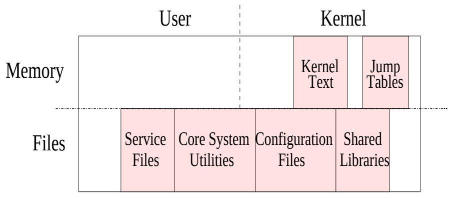


Fig. 2. Protected Zones

```
/bin          RD_X
/sbin         RD_X
/boot         RD_X
/usr/bin      RD_X
/usr/sbin     RD_X
/etc/passwd  RD_ONLY
```

Fig. 3. Sample Paladin policies

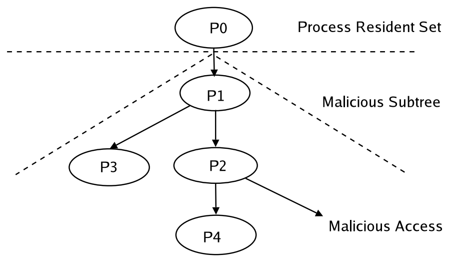


Fig. 4. Automated Containment in Paladin

b)Tracking: The tracking mechanism is responsible for maintaining relationships between different operating system objects. It tracks system calls in the Guest OS and builds a dependency tree that denotes relationships between files and processes. The goal here is to track continually the parent-child relationship between processes and file creation by processes. This is used later by the containment phase to deduce information about the malicious subtree.

c)Containment: The containment mechanism is designed to minimize the damage caused to the system once a malicious process is detected. Rootkits often bundle keyloggers and backdoors that run on the system. This involves a group of processes that perform these actions. This is especially the case when other malware like worms or spyware use rootkits or rootkit like techniques to hide. In such a case, the processes also belong to the worm or spyware code that can do much damage to the system. Containment is triggered when an alarm is raised by the detection mechanism. The containment mechanism identifies the malicious processes using the dependency tree and kills them preventing further damage to the system. We also assume a Process Resident Set (PRS), which consists of processes that need to be always running in the system. These include processes like *init*, *login* and system daemons. These processes are not killed by the containment mechanism.

Fig. 4 shows process P2 performing malicious access. This may involve an illegal access to FPZ or MPZ. Our system is capable of intercepting and identifying this access as illegal and preventing it from being fulfilled. P2 is identified as the malicious 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. The previous node visited then becomes the root of the subtree. In this case, P0 belongs to the PRS and hence P1 becomes the root of the malicious subtree. All processes identified in this subtree are considered malicious and will be killed by our system.

Actions such as installation or upgrade of software inside the Guest OS, which results in modification of protected binaries or addition of files to protected directories, has to be preceded with changing the access control policy temporarily. Otherwise, our system will treat it like an attack. This policy change is done through a special controlled interface in the Host OS. Some drivers may genuinely want to hook to the system call table. They require disabling the write-protection on the system call table before they are loaded and unloaded. This also has to be done through the controlled interface. 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.

4 Design and Implementation

In this section, we describe the design approach and the prototype Paladin as shown in Fig. 5. Paladin comprises of several components: Modified form of *VMware Workstation*, PaladinApp, the driver and the database. VMApp and the VMM are a part of the *VMware Workstation* software. Arrows indicate the communication paths between the different components of the system. The dashed box in the figure represents a virtual machine. VMware Workstation uses a hosted architecture [22]. It is installed on a host operating system and relies on the host to fulfil I/O requests from the Guest operating systems. 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.

4.1 Design Overview

The VMM intercepts system calls from the guest operating system and forwards them to VMApp. Only file and process related system calls are forwarded to the VMApp. Upon receiving a system call event, the VMApp process consults the policies specified and determines if the given system call violates any access control policies. If the event violates any specified policy, an intrusion is detected and it signals the PaladinApp to trigger the containment mechanism. If the call does not violate any specified policy, VMApp forwards this information to PaladinApp to update dependency information. The PaladinApp processes the system call and updates any related dependencies. The dependency information is stored in the database. The database may reside on the same machine or on a separate machine on the network.

The application communicates with the VMApp process using two channels. These two channels of communication are established when the virtual machine is powered on. The low priority channel is for logging events. The high priority channel is used by the application to send commands to VMApp. These commands are issued during initialization, containment and the special control interface for changing access control policies. The Paladin system works in three phases: Initialization, Tracking and Containment.

Initialization: In the initialization phase, the PaladinApp registers file and process related system calls with the VMM. The VMM forwards only these system calls to the PaladinApp. The PaladinApp also registers Guest OS kernel addresses to be protected. The PaladinApp obtains the guest kernel addresses from the driver in the Guest OS, which does a symbol lookup. Since the driver resides in the Guest OS, it has knowledge of the Guest OS semantics. The VMApp process reads the *paladin.config* file provided by the administrator and stores the list of policies to validate system calls. The *paladin.config* file also consists of a second list : the PRS list. This list is stored by the PaladinApp and used during the containment phase. At the end of this phase, prevention and detection mechanisms are initialized.

Tracking: The VMM intercepts system calls and forwards registered system calls to VMApp. VMApp forwards this information to PaladinApp. The goal here is to build relationships between files and processes by inferring dependencies. PaladinApp generates the dependency tree from the system call information provided by VMM. 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 it's lifetime, we use the page global directory address stored in the *cr3* register of

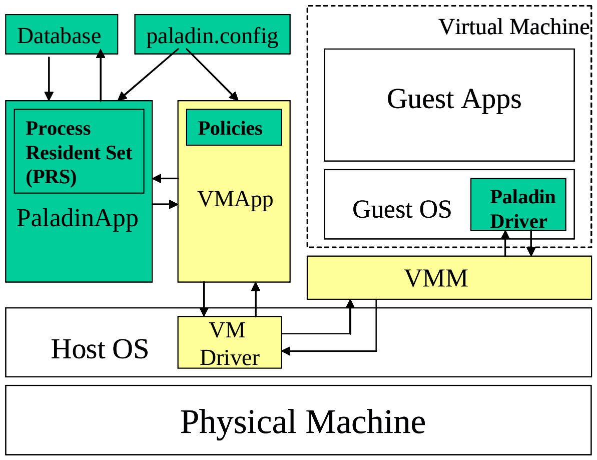


Fig. 5. Paladin Architecture

the x86 virtual CPU as the identifier for processes. Multiple threads within the same process are distinguished using the stack pointer register. For files, the full pathname is used as an identifier. Tracking phase maintains the dependency tree using simple dependency rules.

Dependency tree and dependency rules: In the dependency tree (See Fig. 7), 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. Before updating dependencies, we check the system call return value to make sure that the call actually succeeded. In case a malicious access is detected, the system initiates the containment phase before waiting for the return value.

Paladin uses the following dependency rules:

– 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, represent the process name in the dependency tree.
– When a process exits, if the process 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.

The dependency rules allow our system to track the processes and files related to a malicious process. At the same time, our goal is to keep the dependency tree size small enough to be able to provide quick response. This is achieved by continuously pruning the dependency tree according to the dependency rules. Though our containment approach only uses process dependency information, we do track file dependencies for exploring approaches to automated recovery, in the future.

Containment: The goal of the containment phase is to prevent any further damage to the system. This is triggered when the detection mechanism raises an alert. An alert implies one of the two possibilities: (a) There is a illegal access to the MPZ or (b) An illegal access is made to the FPZ. 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 from a kernel module, the process inserting the module is considered malicious. The PaladinApp immediately issues kills the offending process and invokes the containment algorithm to locate and kill processes in the malicious subtree. 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.

4.2 Implementation

Our prototype was developed for *VMware Workstation*. The host machine was running the 2.6 kernel. The virtual machines were running the 2.2 and the 2.4 Linux kernels. The database used was MySql. We installed several hooks in the VMM to enable it to run commands specified by the application. Upon initialization, the PaladinApp requests for addresses of various kernel symbols from the driver. In our prototype, 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 location of these areas to the PaladinApp.

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. 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 the OS implementation of process creation uses copy-on-write for the user stack pages.

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

5 Evaluation

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

5.1 Linux Rootkits

We analyzed 37 significant rootkits available in the wild for Linux. Fig. 6 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 like viruses, worms and spyware have pretty much no limits on the damage that they can do to the system, stealthily.

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

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

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

Category 4	
Rootkit	Test Set
backdoor-caca	-

✓	Included in test set
-	Not included in test set

Fig. 6. 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)

5.2 Experimental Results

As shown in Fig. 6, 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 10 others and hence they were excluded from our test set. We however contend that since these rootkits use similar techniques to hide as the ones included in our test set, Paladin will be able to counter these rootkits as well. We use simple policies shown in Fig. 3 for our experiments. These policies are directly borrowed from commonly used policies with Tripwire [15] and AIDE [24]. To test for false positives, we chose a few applications like *vi*, *emacs*, *mozilla* and services like *sshd, sendmail, apache* to run with Paladin enabled. We did not encounter any false positives while running these applications. However, installation and upgrades to the system if not done through the controlled interface will yield false positives. There were no false negatives since all 27 rootkits were detected.

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 [25]. 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.

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. These are a part of the protected zones as specified by the policy file. 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.

Category 2: The *Adore* rootkit replaces 14 system call entries in the system call table and redirects them to it's own versions. It has a user-space program called *ava* that 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.

Category 3: The *Suckit* rootkit changes machine code in the IDT handler *system_call* and redirects requests to it's 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 it's doctored system calls, which get executed instead of the original system calls. It very effectively hides itself using such a 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.

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.

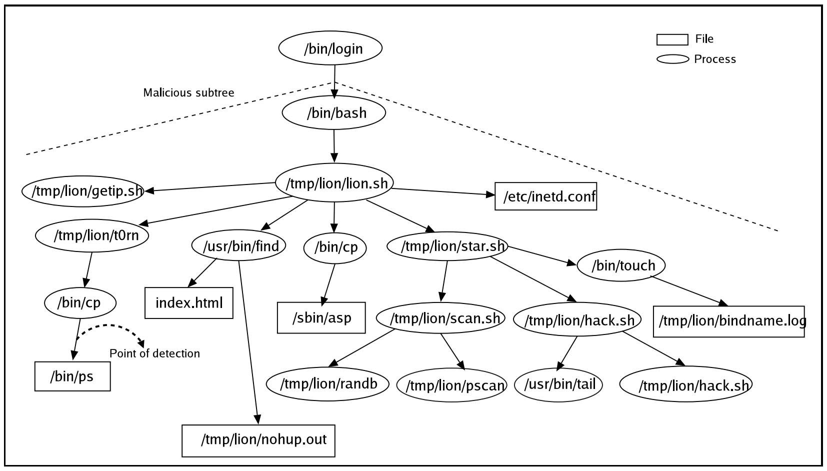


Fig. 7. Dependency tree showing the Lion worm attack

Stealth Worm that uses rootkit: *Lion* is a notorious stealth worm [25] 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 it's 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 which did not contain a rootkit and modified it to install the *t0rn* rootkit.

Lion affects DNS servers that have the BIND TSIG vulnerability [26]. Fig. 7 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 [27]. The worm finds all the files present on the system called *index.html* and defaces the pages by replacing it with it's 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. It is however useful in preventing further damage to the system.

5.3 Performance

Fig. 8 shows the per system call overhead for the four most common file and process related system calls. Paladin incurs larger overhead for open and fork as the application performs bookkeeping and matching of parent/child system calls respectively. Fig. 9 shows the performance overhead incurred by applications that run with and without Paladin prototype. 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. Paladin adds about 12% overhead to the execution time for applications in the Guest OS. VMware provides *fast path system call optimization*, which incurs about 50% less overhead compared to the slow path system calls used by the Paladin prototype. We plan to support this as part of our future work, which will substantially reduce the Paladin overhead.

5.4 Dependency Tree Size

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

	Paladin	
	Disabled	Enabled
fork	1.5 μs	3.5 μs
exit	1.5 μs	1.6 μs
open	0.8 μs	1.5 μs
close	0.5 μs	0.7 μs

Fig. 8. Performance of individual system calls

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

Fig. 9. Paladin Performance Measurements

from the database. 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.

6 Discussion

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

6.1 Counter Attacks

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

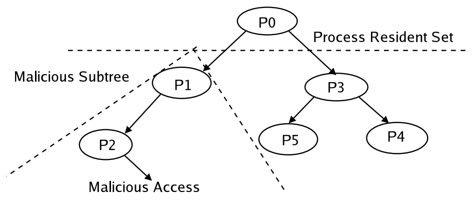


Fig. 10. Multiple control processes

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 Fig. 10. 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 like 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 it's children to P1. The hiding processes, P1 and it's children will be killed, while P3 and it's children will continue to run.

While this attack, cannot be contained by Paladin completely, it exposes the attack to other anti-malware programs. 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.

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. Our prototype cannot currently handle this. However, we can consider other approaches like storing data in a separate data VM [28] to force file access through a well-defined interface.

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, just like we currently protect the kernel code pages. This is not present in the current prototype as we do not have a Linux rootkit to test against.

6.2 Limitations

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

Killing Legitimate Processes: Our containment algorithm, kills all processes inside the malicious subtree. This may involve other genuine applications run by the user. The algorithm kills all user applications including the user's shell as a precautionary measure. This may be undesirable in certain circumstances. This is however, a side-effect of our algorithm. We believe this can be resolved by adding more context and we strive to improve this further by delving more deeply into the access control policies used by our system.

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.

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. Stealthy software may also try to hide in video RAM, BIOS, inside unused disk blocks or by using steganographic techniques. These are issues we are investigating as part of our future work.

6.3 Windows Rootkits

In terms of number of rootkits present in the wild, rootkits pose a huge problem to the Windows platform as well. It is worth discussing the techniques used by these rootkits, which are somewhat different from those used by their Linux counterparts. These differences are mainly due to the different designs of the two operating systems. The goal here is to examine the challenges posed by these 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 [29], 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. Most rootkits found in existence today are user-level rootkits as they are simply more portable and easier to write. They are however more sophisticated than the Linux user-level rootkits that replace system binaries by trojaned versions. We discuss the different techniques used by these rootkits in the sections below.

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, which 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*) [17] 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 we can extend our work to tackle these attacks by adding hooks to those system calls that attach DLLs, along with a set of policies detailing which programs are allowed to do it legally. This is part of our future work. These types of attacks are almost impossible to perform on Linux or other UNIX like operating systems.

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 that 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 [30] is a good example of this type of rootkit. FU removes it's process entry from the process list maintained by Windows. This hides the process from tools like Task Manager that 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 cannot automatically counter this kind of compromise, where data structures are manipulated. In fact none of the available tools today can counter such attacks without previously known signatures.

7 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.

7.1 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 [31–33]. However these systems differ in several ways from our system. RFS and Taser [31, 33] perform recovery on the file system which may have been damaged as a result of intrusion or human errors. Both systems perform offline recovery and aid the administrator by reducing the time to recover. Both systems also rely on external host-based intrusion detection systems (HIDS) or the users themselves to detect anomalies that indicate that the system is compromised or damaged. Backtracker [32] 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. This analysis is again performed offline and 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 [33, 32].

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 hence does only require 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 [34] and kernel hypervisors [35] implement security policies on application by trapping system calls. This is similar to our approach of observing illegal access to files.

Our model of placing the IDS in the host OS and intercepting events from the Guest OS, was derived from VMI [23]. However, we differ in a very significant way. While VMI performs detection of kernel rootkits, we detect kernel as well as user-level rootkits. More significantly, our framework performs automatic containment of the attack as well. Our approach of automatic containment is novel to the best of our knowledge. Introvirt [36] proposes writing vulnerability-specific predicates to react to an intrusion. This can be considered similar to our approach in providing fine-grained reaction to an intrusion when an intrusion is detected. Introvirt requires the vulnerability to be discovered first and a patch written to specifically fix the vulnerability.

7.2 Different Approaches, Similar Objectives

Copilot [16] 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, it reliably detects kernel rootkits and cannot easily be compromised by an attacker with root on the system. While this approach can perform very efficient detection, it cannot perform any containment as it is restricted to polling based approach. Another disadvantage of Copilot is that it requires the firmware on the card to have intimate knowledge of the kernel layout like the module structure. This approach is not very easily portable to different versions of the same operating system. Since our implementation uses the virtual machine technology, we can perform efficient containment due to the fact that we can trap events in the Guest OS. Our approach is also more portable across different versions of the OS because the system call interface provided by the OS does not change much across different versions (changes are usually additions of new calls). The driver that Paladin uses requires knowledge of Guest OS data structures. This is the only software component that needs to change with different versions. Molina et al [37] 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 [15] also detects modifications to the file system but can be easily defeated by a kernel rootkit attack as it depends on the kernel being uncompromised.

Strider Ghostbuster [17] can detect all hidden files and processes. It uses a *cross view-diff* based approach that compares the user level view with the kernel level view on the same system and then the inside the box view with a clean outside the box view to check for hiding processes and files. The whole process however is time-consuming as it scans the entire disk and requires rebooting the system from a clean OS. Tools currently used to detect rootkits like chkrootkit and Samhain reside in the kernel and hence are vulnerable to compromise by a kernel rootkit. 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 defense but do not address rootkit attacks.

8 Conclusion & Future Work

Rootkits attacks pose a serious threat to computer systems. In this paper, we have designed a framework to automatically detect and contain rootkit attacks. We could successfully detect and contain the effects of almost all rootkits that we could find in the wild for Linux. Our containment mechanism was found especially useful in the case where other malware uses rootkits to hide.

As part of future work, we plan to investigate the extensibility of our system to detect malware that exhibits other types of stealthy behavior like hiding in the BIOS, video RAM, unused disk blocks or hiding using steganographic techniques. Tackling Windows rootkits is also a part of our future work landscape. Additionally, we plan to experiment with different policies that can be specified with Paladin, look at generating rootkit attack signatures and also explore the feasibility of automated recovery.

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://searchsecurity.techtarget.com/originalContent/0,289142,sid14_gci1176494,00.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. "2005 fbi computer crime survey," http://www.fbi.gov/publications/ccs2005.pdf.
13. "chkrootkit," http://www.chkrootkit.org/.
14. "Samhain," http://la-samhna.de/samhain/index.html.
15. Gene H. Kim and Eugene H. Spafford, "The design and implementation of tripwire: a file system integrity checker," in *Proceedings of the 2nd ACM Conference on Computer and communications security*, 1994.
16. Nick L. Petroni Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh, "Copilot - a coprocessor-based kernel runtime integrity monitor.," in *USENIX Security Symposium*, 2004.

17. 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)*.
18. "Intel virtualization technology specification for the ia-32 intel architecture," ftp://download.intel.com/technology/computing/vptech/C97063-002.pdf.
19. "Amd pacifica virtualization technology," http://enterprise.amd.com/Downloads/Pacifica_en.pdf.
20. Carl A. Waldspurger, "Memory resource management in vmware esx server," *SIGOPS Operating Systems Review*, 2002.
21. 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 ACM Symposium on Operating Systems Principles*, 2003.
22. Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim, "Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor," in *USENIX*, 2001.
23. Tal Garfinkel and Mendel Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. Network and Distributed Systems Security Symposium*, 2003.
24. "Advanced intrusion detection environment," http://sourceforge.net/projects/aide.
25. "Lion worm attack advisory," http://www.ciac.org/ciac/bulletins/l-064.shtml.
26. "Bind tsig vulnerability," http://www.sans.org/resources/idfaq/tsig.php.
27. Dan Ellis, "Worm anatomy and model," in *Proceedings of the 2003 ACM workshop on Rapid Malcode*, 2003.
28. Xin Zhao, Kevin Borders, and Atul Prakash, "Towards protecting sensitive files in a compromised system," in *IEEE Security in Storage Workshop*, 2005.
29. 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.," 2004.
30. "Fu rootkit," http://www.rootkit.com/project.php?id=12.
31. Ningning Zhu and Tzi cker Chiueh, "Design, implementation, and evaluation of repairable file service," in *DSN*, 2003.
32. Samuel T. King and Peter M. Chen, "Backtracking intrusions," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.
33. Kamran Farhadi Zheng Li Ashvin Goel, Kenneth Po and Eyalde Lara, "The taser intrusion recovery system," in *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.
34. Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer, "A secure environment for untrusted helper applications," in *Proceedings of the 6th Usenix Security Symposium*, 1996.
35. T. Mitchem, R. Lu, and R. O'Brian, "Using kernel hypervisors to secure applications," in *ACSAC*, 1997.
36. 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 twentieth ACM symposium on Operating systems principles*, 2005.
37. 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*, 2002.
38. Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage, "Automated worm fingerprinting," in *OSDI*, 2004.
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.