5-1-2009

Autoscopy: Detecting Pattern-Searching Rootkits via Control Flow Tracing

Ashwin Ramaswamy
Dartmouth College

Autoscopy: Detecting Pattern-Searching Rootkits via Control Flow Tracing

Dartmouth Computer Science Technical Report TR2009-644

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Master of Science

in

Computer Science

by

Ashwin Ramaswamy

DARTMOUTH COLLEGE

Hanover, New Hampshire

May, 2009

Examining Committee:

(chair) Sean W. Smith, Ph.D.

Andrew T. Campbell, Ph.D.

Sergey Bratus, Ph.D.

Michael E. Locasto, Ph.D.

Brian Pogue, Ph.D.
Dean of Graduate Studies

Abstract

Traditional approaches to rootkit detection [24] [45] [25] [34] assume the execution of code at a privilege level below that of the operating system kernel, with the use of virtual machine technologies to enable the detection system itself to be immune from the virus or rootkit code. In this thesis, we approach the problem of rootkit detection from the standpoint of tracing and instrumentation techniques, which work from within the kernel and also modify the kernel's run–time state to detect aberrant control flows. We wish to investigate the role of emerging tracing frameworks (Kprobes, DTrace etc.) in enforcing operating system security without the reliance on a full-blown virtual machine just for the purposes of such policing. We first build a novel rootkit prototype that uses pattern-searching techniques to hijack hooks embedded in dynamically allocated memory, which we present as a showcase of emerging attack techniques. We then build an intrusion detection system — autoscopy, atop kprobes, that detects anomalous control flow patterns typically exhibited by rootkits within a running kernel. Furthermore, to validate our approach, we show that we were able to successfully detect 15 existing Linux rootkits. We also conduct performance analyses, which show the overhead of our system to range from 2% to 5% on a wide range of standard benchmarks. Thus by leveraging tracing frameworks within operating systems, we show that it is possible to introduce real-world security in devices where performance and resource constraints are tantamount to security considerations.

Acknowledgments

I would like to sincerely thank Prof. Sean W. Smith for all his support and encouragement, for guiding me through the unfamiliar world of computer security, and for his supervision and unrelenting sense of humor. He showed confidence and faith in me ever since the first time we met, and I'm greatly indebted to him for that.

I would also like to thank Prof. Sergey Bratus and Prof. Michael E. Locasto (now at George Mason University) for their constant interaction and encouragement. Sergey, with his eclectic talents, introduced me to anti-rootkit technologies and their existing drawbacks, prompting me to investigate this field further. Regular discussions with Sean, Sergey and Michael led to the development of Autoscopy.

I also thank all members of the PKI lab for their consistent chattering, deeply ethical and philosophical discussions, and for always being eager and available to give advice.

Lastly, I wouldn't even be at Dartmouth were it not for my loving parents, who through hard times, put me through an envious education and constantly supported my goals and ambitions. Words cannot describe their selflessness and my gratitude, and I shan't try.

This work was supported in part by the National Science Foundation, under grant CNS-0524695 and the U.S. Department of Homeland Security under Grant Award Number 2006-CS-001-000001. The views and conclusions do not necessarily represent those of the sponsors.

Parts of Chapters 1 to 7, and 9 are based on material from my thesis proposal [38]. Parts of Chapters 7 and 8 are also based on a paper draft currently in submission.

Contents

Chapter 1

Introduction

In this thesis, we focus on the problem of mitigating the proliferation of kernel-based rootkits within operating systems. We do this by leveraging modern tracing frameworks within the OS itself. This enables us to reliably and efficiently monitor required state and examine the system for anomalies using OS-provided "triggers" or probes. Our approach of detecting anomalies focuses on identifying generic kernel control flow paths and then dynamically evaluating them at runtime to detect the presence of rootkits. As we show, this approach has both the *flexibility* of operating within harsh performance constraints, and is also extensive enough to cover most working areas of a kernel.

Charlie Miller, the principal security analyst for Independent Security Evaluators in a recent email exchange [9] with the British website *The Register* notes: "The gap between a script kiddie and a hacker just got a little smaller." He was referring to Immunity Sec's latest educational offering: an x86 "debug register" rootkit dubbed DR [3] that ostensibly blurred the already receding divide between sophomoric "script kiddies" (relegated to the superficial tasks of executing conveniently coded scripts and tools that accomplish the necessary goals, while barely understanding their own actions or consequences) and established hackers. The effects of this diminishing gap have already hit us — a seemingly unstoppable, constantly evolving force of malware that threatens every corner of our lives, arising from the pervasiveness of computers within our homes, offices, governments and mobile environments. Each passing week brings with it reports of malware attacks across the Internet, corporate environments, educational institutions and even cellular phones, these events unmistakably providing a stark analogue to our world's daily corpus of front-page calamities.

These attacks are *real*, they topple our economies, our industries, our politics, and our privately-held lives, and as always, there exist guards, defenses, counter measures and prevention policies against such impending threats at every level — but their reliability remains a dominant question. Do these measures protect us from *every* possible threat? The answer is almost never in the affirmative, the reason being that preventing unknown forms of attack is justifiably hard, perhaps impossible. Software flaws arise out of the manufacturing (i.e. software development) process itself instead of the system acquiring such defects through repeated use, and detecting these crevices is itself a challenge, let alone plugging them with some metaphorical "duct-tape".

Malware, on the other hand, has been progressing rapidly both in terms of technological innovations (packers, VM/Debugger detection, "blue pill" escape techniques etc.) and deployment strategies (botnets, covert network and storage channels, and so on). Whole segments of the software industry have now cropped up dedicated to the task of preventing or at least detecting the spread of malicious code across the network. While this is certainly an admirable endeavor, a large portion of anti-malware techniques in popular deployment today are enervated through their signature-based approaches, which build identifiable software signatures out of known attacks in order to detect or prevent their actions. This is largely a defeatist attitude since our defenders are now always a step behind the seemingly impervious attackers. Deploying systems that do not rely on these signature calculations, but rather attempt to characterize the behavior of normal or malicious code and use this behavior to identify malware, remains a challenging task due to the performance burden imposed by these systems, and their possibility of reporting a significant number of false positives/negatives.

While the term "malware" is pejoratively used to refer to a variety of malicious software — worms, viruses, rootkits, trojans, time-bombs *etc.*, an attacker typically staggers his infiltration of the target [41] using just a subset of these avatars, much like ancient battlefield tactics that staged their armies with the swordsmen in front, archers in the middle and catapults forming the rear. A typical staged malware attack would involve the following stages:

1. **Penetration:** This phase involves exploiting the target through buffer overflows, SQL injection, format string vulnerabilities *etc.*

2. **Installation:** The malware then installs the necessary files or devices, and sets up the malware environment.

3. **Cloaking:** Malware components are then cloaked within the operating system and its environment.

4. **Diffusion:** Finally, the malware spreads across communicable machines within the target network and/or leaks information as desired by the malware author.

Rootkits are a class of malware that are responsible for camouflaging portions of the malware within the target environment while retaining illusions of normalcy to an external observer. The term *rootkit* arises out of the keywords "root," referring to the highest privilege of access on a Unix machine, and "kit", a collection of collaborative programs meant for gaining and retaining root privilege on remote hosts. The threat model of rootkits is significantly different from the threat models of say, viruses, worms or trojans. While the latter category of malware are primarily concerned with exploiting vulnerabilities in Internet-facing services or with rapid diffusion across the network, rootkits take these exploits for granted. They presume that the vulnerabilities (if any) have already been exploited in a previous leg of malware deployment, and as such the attacker now has deifying *root* privileges on the remote system.

Still, the attacker's job is only half done — his priorities now shift from infiltration to covertness — how to effectively and efficiently *hide* the malware and its activities from the remote user, system administrator, and intrusion detection agents. This is at least as important an activity as the "breaking-in" process, because any triggered alert would presumably cut off the attacker and cease his advances. The purposes of a rootkit are not just to camouflage its activities, but to also ensure persistence across reboots, and might further include creation of covert channels for communication with a remote entity. Given these diverse functionalities, rootkits have evolved over the past decade to embed themselves ever deeper into the entrails of the operating system in order to evade detection. Considering the behemoths that are modern operating systems, it should then come as no surprise that such rootkits have been vastly successful in their evasive tactics.

Broadly, rootkits can be divided into two categories:

1. **User-level rootkits:** These rootkits are essentially just user-space binaries or libraries mimicking the functionalities and visibility patterns of normal programs. They replace the standard application binaries and libraries, and sometimes even configuration files, in order to assist cloaking of malware. Examples include illicit replacement binaries for `ls, ps, tcpdump, ifconfig` *etc.*, the focus mostly being user-level monitoring and reporting facilities, since these are typically used by system administrators to monitor the state of the system. On the brighter side, detecting user-level rootkits is mostly a solved problem, with filesystem monitoring solutions such as Tripwire [10] and AIDE [1] able

to detect a large percentage of these rootkits. These systems usually build and maintain regular checksums (usually cryptographic hashes) of the filesystem and install event-handling facilities in order to monitor and validate changes, along with policies that dictate the allowed types and levels of accesses. Finally, as a counter-attack against these defensive measures, rootkit authors started progressing into the kernel space with rootkits that install themselves into the operating system itself rather than its environment.

2. **Kernel-level rootkits:** These rootkits are more intricate and evolving than their user-space counterparts. With techniques ranging from naïve redirection of system-calls and altering interrupt handlers to modifying scheduler lists and VFS redirection, kernel rootkits are gaining tremendous momentum as rootkit authors aspire to sustain their attacks and elude detection even from paranoid system adminstrators. On Unix systems, there exist multiple avenues of attack, from malicious kernel module insertion to overwriting raw memory via `/dev/mem` and `/dev/kmem` interfaces, while Windows rootkits exploit a technique known as Direct Kernel Object Manipulation (DKOM) to seize control over critical kernel objects and modify them to suit their purposes. In this thesis, we focus only on Linux rootkits, but the underlying techniques remain universal and hence can be utilized for similar detection mechanisms in other operating system kernels such as Solaris, BSD, Windows etc.

While not yet as popular as user or kernel rootkits, a recent breed of rootkits termed hypervisor rootkits [44] threatens to cause greater damage and provide a higher guarantee of covertness by installing themselves *below* the operating system (in the traditional system stack), thus acting as a service-provider for the OS itself instead of being a consumer of OS services. While the implication here is that such rootkits would go virtually undetected by any kind of sophisticated detection system running within the virtual machine, the rootkits are also expected to be faithful to the underlying hardware, with dedicated duplication of hardware being their primary criterion for survival. However, such replication is believed to be hard and error-prone [23] primarily due to the various anomalies and discrepancies that can be caught at the level of CPU, MMU, TLB, timer chips etc. This is troublesome for the rootkit since it has to not only duplicate the functionalities of the hardware, but also all its faults and anomalies, just to avoid detection from the adversary running within a virtual machine.

There exist other criteria through which rootkits are categorized. Rutkowska presents a Malware Taxonomy [43] consisting of four classes of malware ranging from Type `0` to Type `III`. Type `0` malware are applications that are harmful to the system but do not subvert the OS interfaces in any way. As such, they cause damage while working within the confines of the system boundaries. Since technically there is no violation of OS interfaces, detection of such "malware" is futile. A typical example of Type `0` malware is a program that deletes all files on disk, or corrupts only audio and video files. Type `I` malware modifies static environment variables such as the system call table, standard system binaries and libraries, kernel code regions etc. Type `II` malware advances the modification technique to dynamic regions of the kernel memory including function pointers, data structures etc. Finally, the impending Type `III` malware are hypervisor rootkits that are present underneath the operating system or system BIOS.

Another division of rootkits is based on the specific techniques that kernel rootkits employ to accomplish the necessary "hiding" goals. The Baliga et. al classification [15] categorizes kernel rootkits as:

1. **Control Hijacking:** These rootkits subvert the kernel control flow without ever returning control towards its original intended path.

2. **Control Interception:** Such rootkits temporarily *divert* kernel control flow, while still maintaining the original flow. Typically, some tampering of the kernel state occurs when the flow is diverted.

3. **Control Tapping:** These rootkits are essentially the same as *Control Interception* rootkits, but the kernel state is untampered when the control flow is diverted.

4. **Data Value Manipulation:** These are kernel rootkits that tamper only with the kernel state through modification of data members or variables, without intervening with the control flow.

5. **Inconsistent Data Structures:** Such rootkits cause inconsistency among data structures (usually linked lists) in order to disrupt the kernel's reporting mechanisms.

In this thesis, our focus is only on Type `II` rootkits that employ *Control Interception* and *Control Tapping* techniques, since these by far comprise the majority of rootkits found in deployment today.

We now give the motivation behind developing autoscopy in Chapter 2, and then give some background of the various tools and techniques involved in Chapter 3. Chapter 4 gives an overview of our requirements and contributions, while Chapter 5 describes the implementation of our pattern-searching rootkit. Chapter 6 gives a detailed design of our system, and Chapter 7 shows how we implemented autoscopy and the various tradeoffs involved. Chapter 8 describes our evaluation criteria, details our benchmarking results and also enumerates the various threats to autoscopy. Finally, Chapter 9 highlights some of the related work in this area, and we conclude in Chapter 10. Appendix A gives a couple of code fragments from the implementation of autoscopy.

Chapter 2

Motivating Scenarios

As we explain in Chapter 3, intrusion detection systems (IDSes) are frequently classified as either signature-based or anomaly-based systems, the latter sacrificing speed for better detection capabilities. However, the performance impact of anomaly IDSes is frequently ignored or underestimated, which limits their feasibility of deployment to only high-end server machines with multi-core processors and several gigabytes of RAM. But the ubiquitousness of light-weight systems with feeble processing power and just a few hundreds of megabytes of RAM, executing off flash chips, is undeniable — mobile phones, PDAs, biometric sensors, SCADA RTUs, barcode scanners, home routers, etc. Security in these devices has largely been ignored, both for technical and economic reasons, as evidenced in the recent work done by our colleagues in analyzing networked set-top boxes widely deployed across campus [13].

Existing IDSes rely on the sophistication of *virtual machine* (VM) technology to shield themselves from malicious code, and as such necessitate the presence of a virtual machine monitor (VMM) appropriated for these purposes. In addition to the extra execution code that VMMs confer on the running hardware, guest system performance is limited to a fraction of its original. According to VMware's report, the average performance hit for hardware-assisted fully virtualized guests was about

6% for Vmware and 15% for Xen 3.0.3. We request the reader to consult Vmware's report [46] for detailed analyses on the latency delays for both these hypervisors.

In the embedded domain, it would be unpropitious to deploy systems built on top of VMMs because of limited resources, performance constraints and real-time requirements, specifically within the US Power Grid infrastructure. *Supervisory Control and Data Acquisition* (SCADA) systems in the Power Grid and elsewhere are known for their negligible and antiquated security mechanisms on millions of devices strewn across the country today. When implementing add-on security mechanisms for these devices, one should consider the elongated lifetime of these systems, and accordingly consider both upgrade flexibility and system overheads, wherever applicable. The security agency Riptech (now a part of Symantec) reports a lack of real-time monitoring on SCADA systems [40] due to the incapability of these systems to handle large amounts of data without significantly affecting system performance. We are thus motivated to introduce real-world security in these devices where performance and resource constraints are tantamount to security considerations.

Finally, in terms of coverage, kernel-level intrusion detection needs to be far more exacting and thorough than detection systems at the application-level. On the other hand, performance overheads in the latter are only constrained to the application of choice, whereas any overhead suffered at the kernel impacts system-wide performance, across both processor-bound and I/O-bound paths. These seemingly irreconcilable constraints to be both exhaustive and efficient drive us to implement a detection system that is comparable to the speed of signature-based IDSes while retaining the behavioral detection capabilities of anomaly intrusion detection systems.

Chapter 3

Background

In this chapter, we give a brief overview of the technologies, tools and techniques used to implement our system.

3.1 Intrusion Detection

As with rootkits, intrusion detectors can also be classified on different criteria. Here we focus only on the classification based on the *type* of detection mechanism.

1. **Signature-based:** These detectors proceed by first conducting a source-level or object-level analysis of the malware sample and deriving a set of signatures that uniquely identify the malware. These signatures are then distributed and fed into the detection system that can now detect this malware. Signature-based IDSes are fast since the system only needs to compute and verify against a finite set of signatures. However, even slight variants of the analyzed malware samples can fool the IDS altogether. Also in the specific case of rootkit detectors, signature-based methods [36] [42] [2] [8] [33] must be run manually (or periodically). Such detectors do not provide continuous inline monitoring of the host system.

2. **Anomaly-based:** Anomaly-based IDSes attempt to characterize the normal working behavior of the client application or system, and use this characteristic as the ground truth against which all future measurements are classified. These systems work by observing either behavior patterns of processes over time [22] [26] or call stack monitoring [21] or other patterns of established behavior. Any deviation from the normal order would indicate aberrant behavior and hence a high likelihood of an intrusion. Such systems usually suffer from slow learning times, detection latencies and false positives. In the subdomain of rootkit detection, anomaly detectors [45] [25] employ virtual machine monitors (VMM) to enforce containment and isolation properties on the guest machine.

3.2 Loadable Kernel Modules

Loadable Kernel Modules (abbreviated as LKMs) are modules that can be dynamically inserted and removed from the core kernel without affecting other system services. They provide OS extensibility in monolithic kernels, and are designed by borrowing ideas from traditional micro-kernel models. Linux's `kmod` driver provides load-on-demand functionalities to the kernel modules residing on disk, so that an LKM is injected into the kernel only if its functionalities are required. In other cases, LKMs passively reside on disk. Interfaces in the kernel are exported to the LKMs for module code to hook into the kernel. However, even though inserting LKMs require root privileges on the box, the LKM interface opens up a major avenue of attack for kernel rootkits.

3.3 Kprobes

As system administrators started to grapple with increasing program complexity, heisenbugs [1], and seemingly impenetrable operating systems that offered sparse debugging support, OS engineers came to terms with the growing need for systems that offered monitoring, tracing, and debugging facilities across the system as a whole instead of focusing on just individual applications. Building frameworks to understand system performance and locate bottleneck and throttling conditions became of utmost importance, and Sun responded in January, 2005 with its release of DTrace [18]. With its low-latency observation mechanisms that could be run on *actual* deployed systems and inflicting no overhead when offline, DTrace was an instant hit with administrators everywhere.

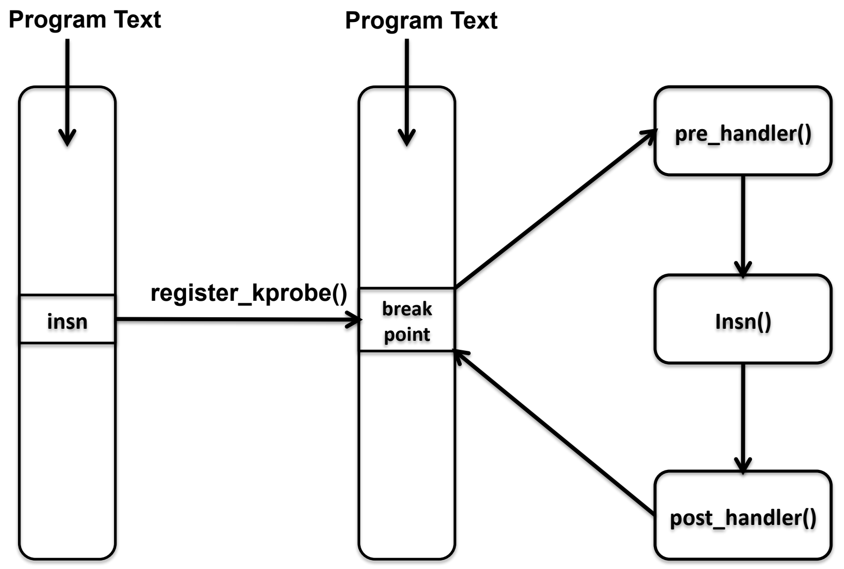


Figure 3.1: Control flow within the Kprobes framework (duplicated from [32])

Linux responded with Kprobes [32], developed by IBM researchers and introduced in kernel version 2.6.9 that allowed the execution of arbitrary handler code at any point during the kernel's execution. Since probe handlers are written within

[1] hard-to-track bugs, whose behavior changes when observed

LKMs, one typically needs root privileges in order to use this facility. Kprobes is also extremely useful for debugging specific parts of the kernel without the need for traditional recompile/reboot/test cycles, thus saving development costs and time. Furthermore, kprobes provides better tracing support for the kernel, facilitating deeper transparency and performance evaluations. Recently, a project called SystemTap [37] was undertaken whose current members include RedHat, IBM, Intel and Hitachi. The aims of the project are to provide a usable abstraction framework atop Kprobes and make the design palatable for users possessing little programming knowledge.

Since our system uses Kprobes for installing probe points across the kernel, it would behoove us to take a deeper look at how Kprobes is internally implemented. Kprobes provides three handlers for the user to implement (all of which are optional) for his choice of instruction in kernel memory. Kprobes copies the instruction at the address indicated, and installs a breakpoint[2]. When the breakpoint is hit, the trap handler verifies if a probe was inserted at that instruction. If so, the trap handler transfers control to Kprobes, which then calls the pre-handler (if registered). The copy of the original instruction is then executed. If the user had installed a post-handler, this function is then called. Finally, normal control resumes from where the trap was initiated. If the OS encounters a fault during any of these processes, a user-specified fault handler is called which is useful mainly for diagnostic purposes. Figure 3.1 is a succint visual representation of the above verbiage.

3.4 Intel x86 assembly calls

Function calls on `x86` can be either direct or indirect — a direct function call is one where the target function is known (and resolved) during compilation or linking time. On the other hand, an indirect function call is one whose call target cannot

[2]On `x86` systems, the opcode of a breakpoint instruction is `0xCC`

be determined at compilation or even program linking time, but can only be resolved at runtime by either the operating system, the system dynamic linker, or sometimes the application's own linker. Hence the address of the target (or callee function) is not known until runtime, when the call is actually decoded and executed by the processor. An assembly `CALL` instruction implements a typical C function call on the `x86` architecture. The `x86` instruction set provides assembly-level counterparts to the high-level direct and indirect function calls present in C.

The operands of the `CALL` instruction encode the call target (or callee function). In the case of a direct function call, the operand is a fixed value that is the address of the target function. This operand is inserted either during the compilation, static linking, or dynamic linking phases. When calling a function through a pointer (indirect call), there are two ways in which the call can be accomplished in assembly:

1. **Memory-indirect:** The operand of the `CALL` is a memory address. When the `CALL` instruction is executed, the contents of that address are first fetched, then treated as the address of the target function, and finally the call is performed. Examples of such calls are:

```
call *(%eax)
call *0x4(%eax)
```

2. **Register-indirect:** The operand of the `CALL` is the index of a suitable register. The address of the callee function is loaded into this register just before the `CALL` instruction. The `CALL` subsequently picks this address from the register and calls the function it points to. Some examples are:

```
call *%eax
call *%ecx
```

3.5 *kmem* and *LKM* rootkits

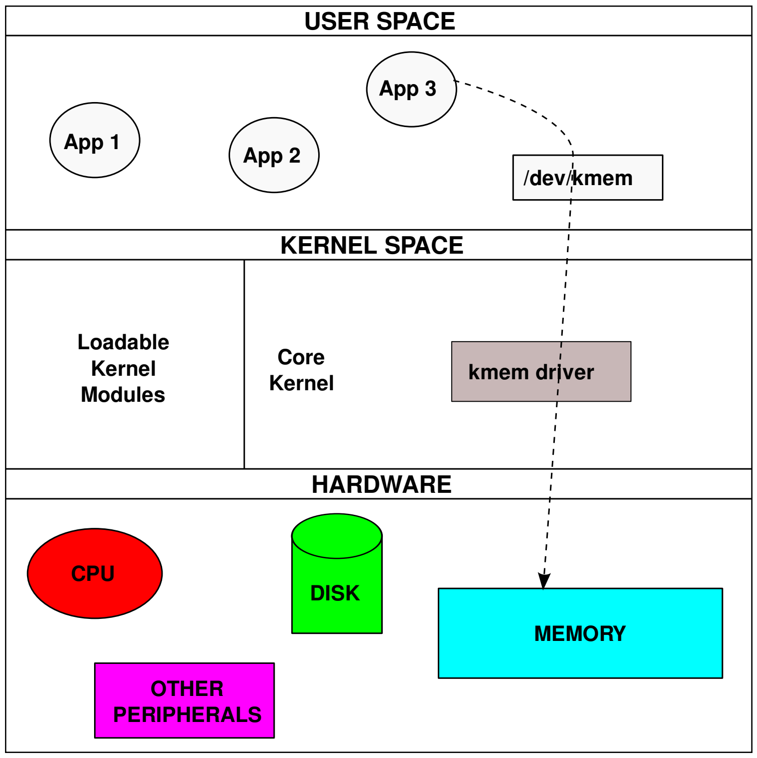


Figure 3.2: System architecture with respect to LKM and kmem interfaces. `App3` is a userspace application that uses `/dev/kmem` to access raw system memory.

The */dev/kmem* driver is used to directly access kernel memory from userspace. This interface is sometimes used to apply bug-fixes or dynamically patch the kernel without incurring the overhead of a system reboot. Also, user-space drivers usually access device memory through either the mem or kmem interfaces. Disabling these interfaces could either break backward compatibility, or necessitate workarounds for applications currently using them. That being said, the kmem and LKM interfaces are the two most widely abused boundaries in the Linux kernel, with rootkits routinely exploiting them for entry into the kernel. In fact, there have been cases of administrators disabling loadable modules in the kernel to completely avoid rootkit attacks. But this still admits */dev/kmem* rootkits. There do exist slight differences in rootkits deployed through these two interfaces. kmem rootkits are arguably weaker than LKM rootkits, mostly due to the fact that the rootkit now has to guess more often, since kernel-exported interfaces are not available. But kmem rootkits can rely on the driver always being available while in the case of LKM rootkits there is the possibility that the administrator might have crippled the interface. There have also been efforts to disable[3] the kmem interface altogether — but on the 2.6.19.7 kernel we tested on, which was running Ubuntu 7.04, */dev/kmem* could still be mmapped[4]. We note that kmem and LKM are just interfaces to access the kernel, and a kernel vulnerability is equally capable of providing such accesses, though its convenience is arguable. Figure 3.2 depicts the system architecture with respect to LKMs and the kmem interface.

[3]http://kerneltrap.org/mailarchive/linux-kernel/2008/2/11/809424

[4]This is also the case with later versions of the kernel

Chapter 4

Approaching the Problem of Detecting Rootkits

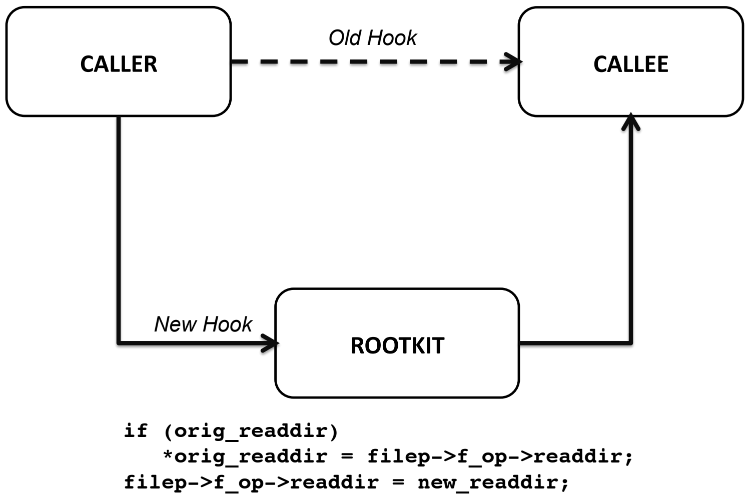
4.1 Overview

We present a radically different approach to rootkit detection that allows us to leverage existing tracing frameworks in modern operating systems and obviate the need for an external VMM. This also has the advantage of reducing system overhead since we are essentially protecting the OS from *within* itself, instead of delegating the implementation to an additional VM layer. But since we are attempting to protect ring `0` code from within ring `0`, there is no guarantee that our system is completely shielded from malware code. As such, we advise the use of other technologies such as W$\oplus$X/NX [7], Copilot [35] to ensure static text integrity.

Control flow anomaly detection techniques have been used before [34] to detect kernel rootkits, but these approaches use static code analysis techniques (by tracking global kernel variables) to determine control flow graphs, and a VMM at runtime to verify the control paths. But static analysis techniques are incapable of detecting hooks present in dynamically allocated memory. Also, performing such detection by frequently interrupting the guest control flow (i) first needs to address the VM semantic gap [20], (ii) can cause performance issues due to rapid context-switching between the host and the guest and, (iii) might still allow ephemeral rootkits to persist on the monitored guest. Basically, we intend to operate where more traditional methods of interposition and program analysis tend to be either insufficient, expensive, or both. Our self-monitoring system, autoscopy, attempts to learn a model of the kernel's function pointers using kprobes, and classify all future flows in accordance to the learned model.

To understand how our system works, we first give a brief explanation of how existing rootkits hijack kernel function pointers (analogously hooks or pointer hooks) in order to alter control flow. Control-flow altering rootkits comprise by far the majority of rootkits deployed today. Recall from Chaper 1 that the *raison d'etre* for rootkits is to expertly camouflage hostile behavior without raising the suspicions of the system administrator or host IDSes. In order to achieve this, the system should seem to be functioning as normal even though it may not actually be. To illustrate, if one is hijacking hooks at the `vfs` layer, then all reads/writes/stats to files and directories other than the one the rootkit is interfering with, should not be tampered or inhibited in any way. Otherwise, an alert system admin or monitoring IDS might detect changes in the filesystem that should not have manifested in the first place.

Interfering with control flow is possibly one of the easiest ways a rootkit can inject itself into a working kernel without significantly altering system state. To conduct a typical injection attack, the rootkit author first identifies the kernel components to carry out the deception — the network stack, filesystem layer, scheduler routines *etc.* Identifying the target gives the attacker a set of possible hooks to work with. These hooks may be present either in static or dynamic memory, either within the core kernel or in kernel modules. The attacker then proceeds to select the subset of hooks



```
if (orig_readdir)
   *orig_readdir = filep->f_op->readdir;
filep->f_op->readdir = new_readdir;
```

Figure 4.1: Interposing kernel control flow by a rootkit through hook hijacking.

that usually give him greater flexibility and control to conduct the attack, after which the rootkit is coded to interpose itself using these hooks. Figure 4.1 is an illustration of how this works. The *caller* and *callee* blocks signify their respective functions. The caller calls the callee using a function pointer (hook). By overwriting this hook to point to the its own function, a rootkit performs a proverbial *man-in-the-middle* attack. Note that the rootkit still tranfers control to the original callee function, since it wishes to retain the services provided by this layer, while still carrying out its designated tasks.

Thus rootkits hijack kernel hooks, but eventually call the kernel's native functions to do the actual work (thereby providing the illusion of uninterrupted service). An unreliable or unavailable service increases the risk of discovery — precisely the event that rootkit authors strive to avoid. This tendency also makes good software engineering sense because rootkit authors, like regular software engineers, are under pressure to reuse specialized known-to-work code.

To understand and further motivate the case for our intrusion detection system that is capable of detecting advanced rootkits, we first set out developing a rootkit prototype of our own that does not conform to existing methods of attack (eg: hijacking hooks in static memory regions, or using `System.map`/`kallsyms`/LKM interface to resolve symbols). While a majority of rootkits available for Linux hijack only system-calls (with exceptions like *adore-ng*), we wish to hijack hooks at a layer much below that of a functioning system-call interface, preferably somewhere near the device-driver structures that are omnipresent within the kernel. Hijacking hooks deep down in the bowels of operating systems can be relatively hard to detect, primarily because it is difficult to track down all hooks used by low-level kernel code (especially if the hooks are dynamically allocated), and also monitoring the OS in time-critical sections can be a challenging task.

We further constrain ourselves to working from within the "weaker" */dev/kmem* interface instead of using LKMs. One reason for doing this is to show that the `kmem` (and equivalently, the `mem`) interface is just as enabling a platform for rootkit injections as the LKM interface is. However, we need to adapt ourselves to the lack of symbol resolution and limited visibility when working through `kmem`, and we come up with a pattern-searching technique that allows us to leverage known patterns within kernel structures to identify the hooks, which we then hijack.

4.2 Our Requirements

To better understand the space in which our system lies, and effectively outline our contributions, we come up with a set of requirements that autoscopy must satisfy. These are described below:

1. We aim to dynamically *discover* the kernel hooks to protect, by querying running system state as opposed to static tools or verification processes. This allows autoscopy to be more exhaustive and thorough than existing IDSes. We should

also not consider the source code of the operating system, which might not be available to us in all cases.

2. Autoscopy needs to perform in-place or *synchronous* intrusion detection, i.e. to line up our detector with the control flow so we do not open a time-slot during which one can deploy an ephemeral rootkit. Almost all signature-based systems and a few anomaly-based systems existing today are unable to detect transient rootkits that are installed and removed within a short period. Autoscopy needs to be able to detect even these kinds of rootkits.

3. We aim to not use virtual machines, tainting or other high-cost observation techniques.

4. Autoscopy also needs to be *performance-aware* and practical in deployment. We accomplish this using existing kernel frameworks such as kprobes. While performance awareness might mean adapting the detection system itself to cope with varying system performance, for example by reducing the number of hooks that are monitored, we do not consider it in this thesis. Instead, we aim for low performance overheads at all times.

5. We need to support detection both when the rootkit is already present on the system when autoscopy is deployed, or when the rootkit is installed at some later point in time.

4.3 Contributions of Autoscopy

Our contributions in this work both involve newer rootkit development techniques that require minimal context information and no symbol resolution, and the implementation and evaluation of a mature intrusion detection system that leverages existing OS frameworks to waylay and validate kernel control paths. The primary contributions of our work are as follows:

1. **Development of a pattern-matching rootkit technique** – Existing rootkits implemented as kernel modules use kernel support for symbol resolution in order to hijack hooks. `kmem` rootkits on the other hand require hardcoded addresses, or read symbols off `/proc/kallsyms` or the local `System.map` in order to resolve them — neither approaches being feasible for deployment in the real world. We propose a technique that uses basic semantic pattern-matching expressions to identify hooks in memory using just a minimal specification of the memory regions surrounding the hooks, that we (as an attacker) would be interested in.

2. **An anomaly-based system for detecting control-flow hijacking kernel rootkits** – Our detection system is aimed at providing continuous *inline* protection for kernel hooks whose locations are initially learned by exercising the kernel. Our detection system works either in the presence of an already installed rootkit, or if a rootkit is installed at a later point in time. We use the kernel's tracing framework to insert probes along control flow paths, disassemble the appropriate `x86` instructions on the fly, construct function-level *control-flow graphs* (CFGs), and finally determine the validity of the control flow. Appropriate alarms and notices are raised in case a rootkit is detected within the system.

3. **A detection architecture with low performance impact** – Our system does not rely on static analysis, VMMs, tainted information flow, or other high-cost observation mechanisms. Instead we take advantage of the recent surge in OS-embedded tracing technologies to implement an observation layer on top. As such, we hope to significantly reduce performance overhead and work within

strict resource constraints. In our experiments, autoscopy imposes an overhead ranging from 2% to 5%.

4.4 Caveats of our Approach

Our system is meant to detect a rootkit's presence, detecting deviations in control flow and providing a framework where one can analyze the system state and repair the corruption. We do not seek to understand the high-level behavior (e.g., the protocol of its control channel, or the files it spies upon) of an installed rootkit [29]. Since we work at the same privilege level as potential malicious code, complete isolation is not a privilege we can afford. As such, we require a static integrity monitor [35] [7] to enforce kernel and module text integrity from the most basic rootkits that attempt to alter kernel code.

Without such a system or other kernel text integrity monitor in place, there is a possibility that our system can be subverted. We do not see this assumption as a limitation; rather, our system is aimed at detecting and defeating the more advanced threats that have evolved (and were in response to systems like CoPilot) to not rely on modifying kernel text. In addition, we do not currently deal with the case where a rootkit can copy the hooked function and then execute it (we note that no existing rootkits do this, but it still is a possibility). This scenario can be handled by enabling the processor's NX bit, which prevents execution from any memory page that has been written. We elaborate on the threats to autoscopy and our mitigation strategies in Section 8.6.

We emphasize that, by themselves, static integrity enforcement techniques cannot detect sophisticated rootkits that employ some form of dynamic hook modification or control-flow redirection. We see these tools as building blocks to help build systems such as autoscopy that are directed towards more devious rootkits than the ones that just modify static text or data.

Chapter 5

A Network Driver Rootkit

We undertake the development of a rootkit prototype that advances the field of existing offensive techniques by integrating basic pattern-searching to alleviate the requirement for kernel symbol resolution. As explained in Chapter 3, rootkits can be installed into the operating system through either the LKM interface or the *mem* and *kmem* interfaces. In terms of generated object code, Linux kernel modules are just ELF (Executable and Linking format) objects [5], with dynamic relocation sections indicating the regions that require symbol resolution. The kernel's module insertion code maps the target module into memory and resolves the symbols with their appropriate locations. However, for symbol resolution to succeed in this way, the symbol must first be "exported" from the kernel during compilation time, usually through an identifier such as EXPORT_SYMBOL or EXPORT_SYMBOL_GPL, which are kernel macros.

Rootkits deployed through the *mem* and *kmem* interfaces have no such luxury; and if `/proc/kallsyms` and `System.map` are absent on the target system, then no symbol resolution is possible. To counter this limitation, certain rootkits hardcode the addresses of the system-call table or target hook, but this makes the rootkit unportable and impractical for mass deployment. Also, hijacking of dynamically allocated hooks is hard. Other rootkits use either the IDT handler routines (eg: `enyelkm`) or use naïve search techniques to locate the system call table (eg: `override`). Both techniques only try to identify the system call table, and hijacking hooks at this layer is relatively trivial to detect because of the static nature of the syscall table.

We first see how rootkits do not even need LKM functionality in order to subvert the system. The administrator of a system can choose to disable the loadable module interface, in which case many of the attacks through this interface are defeated. But the */dev/kmem* interface that allows one to read and write into kernel memory is still a dangerous foothold for integrating malicious code into the kernel. Here the lack of kernel-exported interfaces means that a rootkit deployed through the *kmem* driver can no longer be agnostic about where in memory the structure actually resides and has to resort to searching for this structure.

What we first propose is to search for relevant structures in kernel memory using only a small amount of semantic information. Our aim is to make this process generic enough to work on multiple kernel versions and across varying architectures. Examples of such semantic information are the MAC address of the target network driver, the major and minor numbers of the character or block driver, address range patterns in the binary-format structure *etc.* However, naïvely searching for such patterns leads to a large number of positive matches, because the search query is small enough to warrant such hits.

To limit the query results within acceptable bounds, we require some form of specification that describes the data in close proximity to the desired address. The observation that we leverage in writing our rootkit relates to the general layout of kernel data structures in memory. Linked lists, and in particular doubly-linked lists (D-LL), constitute the backbone of a majority of kernel data structures. The Linux kernel provides a framework for efficiently embedding these lists into any given structure. This is how most structures are chained together. Examples of such lists are lists of character devices and block devices, lists of network devices, lists of processes and LKMs, a scheduler list of processes, a list of binary-format handlers, lists of file-systems, inodes, dnodes, timer lists and so on. In order to support OS extensibility and modularity, these structures have function pointers embedded within them that the necessary modules modify to point to their own functions. Thus, doubly-linked lists are an excellent distinguisher for hooks that lie within such structures. We note that these hooks are just a subset of function pointer hooks that are vulnerable to rootkits.

The aim of our rootkit is to hijack the network driver's `send` function, for a device of our choice. This gives us total control over all packets sent into the network, and is particularly dangerous since the packets we modify cannot be revealed by a local sniffer on the host such as tcpdump, ethereal etc. Our semantic specification consists of just the MAC address of the network card. We then search for the MAC string over the entire kernel memory, which might potentially result in a large number of matches (when we performed this experiment, we got about 93 matches). But since we know that all network devices are chained together, we narrow down the search results by pattern-searching for doubly-linked lists within the vicinity of the previous hits (the range of the vicinity is approximated by the size of the kernel's `net_device` structure). The pattern for these lists is a familiar one: the next and previous pointers should point to the appropriate nodes in the list, and the previous and next pointers of adjacent nodes should be identical. In case of a circular list, the last node should point back to the first, otherwise should contain a null next pointer. If all these patterns are detected, then we have successfully identified a doubly linked list at the given memory location.

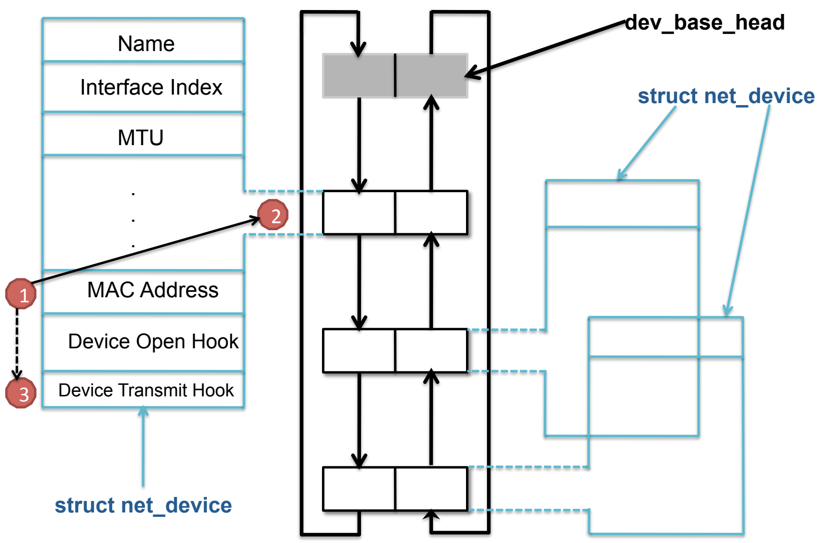


Figure 5.1: Pattern-searching in our rootkit. The numbers indicate the order of search: from (1) the MAC address to (2) D-LL verification to (3) the network driver's transmit hook.

If the final set of matches obtained are not within acceptable boundaries, then we would need to append a stronger specification. In our case, this could be the possible range of values at addresses close to the match (for example the MTU value, interface name etc). Finally, we determine the `send` function-pointer by a fixed offset from the linked-list as determined from the `net_device` structure. We also verify that the pointer points to a valid function prelude and then modify it to point to our rootkit function, which modifies the packets as necessary, finally calling the original `send` function to deliver the packets into the network. The pattern-searching process of this rootkit is shown in Figure 5.1

What this implementation aims to show is that rootkits can completely bypass standard kernel interfaces and instead rely on just the kernel memory to discover their desired hooks, thus making them possibly even more difficult to detect. We stress that these hooks live at unpredictable memory locations and so their values cannot be targeted beforehand. Our rootkit thus adapts to the environment it is injected into.

Chapter 6

The Design of Autoscopy

6.1 Overview

Autoscopy is designed to operate in two phases. An initial *learning* phase attempts to characterize the hooks used within the kernel by identifying the dynamic caller-callee function context (i.e. the function addresses themselves) in which the hooks are used. Since this phase is decoupled from the succeeding detection phase, we could implement our learning using a VMM (and we have experimented with this using QEMU), but in order to provide a homogenous development platform and to better reuse code across both phases, we resorted to kprobes as our final implementation framework. Although not strictly necessary for our detection, we also "beautify" the results in order to facilitate tabulation of our results and possibly aid a manual analysis of hook behavior. The hooks learned through this process form the input for our *detection* phase.

The detection phase uses kprobes as the primary platform for conducting analyses, gathers the previously learned hooks, implements a rigorous detection mechanism that ensures *in-line* flow integrity, accounts for variable module addresses, and is also capable of detecting an already-present rootkit (since we scan for anomalies instead of signatures). Finally, we also conduct an evaluation of our system and benchmark it against a wide range of standard benchmarks.

6.2 Revealing Kernel Hooks (Learning Phase)

Having established that rootkits consistently hijack hooks within the kernel, our approach to detecting rootkits is to first focus on enumerating these hooks that are frequently abused by malware. SBCFI [34] relies on a similar technique, although their technique of discovering hooks is to traverse the kernel source code, mark all global variables that lead to function pointers, and at runtime, traverse these variables to identify the hooks. This approach works, but the limitation imposed by static code analysis along with the VMM implementation prohibits SBCFI from performing *inline* detection thus requiring the system to be run at frequent intervals. Moreover, hooks that are not necessarily reachable from global variables would go unrevealed, and considering the C language's loose type checking, some hooks might be camouflaged as just `void*` pointers and completely evade source code analysis.

The broad task of revealing hooks within the kernel can be approached in a number of ways. One could design a system working from the inside (the target that is being analyzed), or from the outside (a monitoring environment such as a VMM). Even after choosing a particular design, one has to contend with what kind of hooks to detect, since hooks are spread across the entire breadth of the Linux kernel. For example, Hookmap [47], an independent research project, chooses to focus only on hooks that lie on the kernel control flow paths executed by a designated user space application. Typically this application is some kind of a monitoring tool or intrusion-detection system.

We cast a wider net, and our system aims to identify as many hooks as possible in a running Linux kernel. The obvious advantage of this approach is that we can expose more hooks that might be compromised by kernel rootkits, even hooks that might not lie on the control path of a monitoring application (examples of such hooks are the ones present in asynchronous kernel threads or interrupt handlers). A disadvantage of our approach arising due to the fact that we use no “filter” during our process of capturing hooks, is that our approach might lead to an unmanageable number of them and hence thwart manual inspection. We remedy this by providing a convincing analysis that links the hooks we find with the functions that use them, and also link them with the kernel source-code (when possible) to aid better inspection. We note that this tabulation of hooks is just an intermediate glossy process to better illustrate our findings and is not strictly necessary for our second phase.

Moreover, the system we build in this phase is not just meant to feed our particular IDS, but other existing and future intrusion detection systems. We hope to fuel both automatic and manual inspection of kernel hooks and as such, we cannot restrict ourselves to only a subset of them (since an IDS should be capable of handling and protecting hundreds, if not thousands of pointers). Our design leverages the kprobes framework, which we use primarily for validating the hooks we examine, and collecting the necessary information needed for further analysis.

We begin by first collecting plausible hooks from kernel memory. Any pointer into the kernel code-space has the likelihood of being a hook, and we do not (yet) know whether this is the case or not. Since pointers are word-aligned (and the word size on a 32-bit machine is 4 bytes), we attempt to search the kernel memory in increments of 4 bytes. For each word encountered, if its value is a valid text address, we insert a probe at that address. Note that the probes are not inserted on the hooks themselves, but rather on the functions that are called using these hooks. To enumerate memory addresses, we first give a brief description of how Linux handles virtual addresses.

The Linux kernel is statically mapped during bootup at virtual address `0xc0000000` + `0x100000`. The virtual-to-physical mapping is just a linear transformation from this base address, i.e. virtual address `0xc0000000` corresponds to physical address `0x0`. If the amount of physical memory installed on the system is above 896M, then by default, the first 896M is statically mapped, and the rest is considered as highmem region, which is dynamically allocated as necessary. In our case, the amount of physical memory was 710M.

To search for function-pointer hooks, we only need to walk the kernel's data section and not its text, so we skip the initial text region (demarcated by the kernel symbols `_stext` and `_etext`), align the start pointer on the next word boundary and traverse the memory contiguously. Our search pertains only to the physical memory mapped directly into the kernel and we do not attempt to traverse the kernel high memory. We do not see this as a problem since embedded systems rarely have more than a gigabyte of memory, and if needed, we can easily adapt autoscopy to traverse kernel high memory areas. Our algorithm for traversing memory and filtering out this initial estimate of hooks is shown below:

```
1: kdata_start ← _etext {adjusted to a word boundary}
2: kdata_end ← kdata_start + Size-of-physical-memory
3: for $w$ = kdata_start to kdata_end do
4:    $v \leftarrow *w$. {Dereference $w$}
5:    if $v$ is a valid address in the kernel text then
6:       if $*v$ is a valid function prelude then
7:          Insert a kprobe at $v$.
8:       end if
9:    end if
```

```
10:     $w \leftarrow w + 4$.
11: end for
```

A valid x86 function prelude consists of the following assembly statements:

```
push %ebp
mov %ebp, %esp
```

Since we built the kernel with frame pointers enabled, every function will have this signature at the start. Hence a pointer is a function-pointer iff it points to a valid prelude.

When the above algorithm ends, we would have at our disposal an initial estimate of kernel/module functions that *might* be called using hooks. To filter out the false reports, we need to actually verify that these functions were called using function pointers, and to do that we use kprobes to insert our own handling routines when any of these functions are called, and then determine the manner of the call.

For all the "potential" targets obtained through the above memory traversal algorithm, we insert a probe at the indicated location. Note that at this point we have thousands of probes sitting on potential targets. We still have to exercise the kernel so that these probes are fired, and we can determine whether the targets were indeed called using a hook. To fully exercise the kernel, we rely on the extensive Linux Test Suite or LTP [6], which is a project started by SGI and currently being maintained by IBM that consists of a wide range of test cases meant to validate every aspect of the Linux kernel. A complete run of the test suite takes anywhere from 2-4 hours depending on system load. This duration might be further amplified by the fact that we have inserted a few thousand probes across the kernel.

Basically, the idea here is to hit as many of the probes as we can. Hence choosing an appropriate suite (in our case, LTP) is paramount, since we want to exercise the kernel as much as possible in order to maximize our chances of hitting the probes. Note that it might be possible to never hit some of the probes, since the corresponding functions might never get called (such as those in unused drivers) or only get called during system bootup.

We now turn our attention to the probe handlers. Since their task is to determine indirect calls, we first retrieve the return address from the stack, and disassemble it in reverse. The last instruction of the caller function *has* to be a `CALL`. Disassembling this instruction reveals the type of call. If the call was direct, we skip it. Else, we have successfully identified the hook's caller/callee context, but the identity of the hook itself remains elusive. To obtain the address of the hook, we diagnose the operands of the indirect call as follows:

1. **Memory-indirect CALL:** Here the address of the hook is already present in the operand. Since we've just disassembled the call instruction, we can obtain the hook address quite easily.

2. **Register-indirect CALL:** In this type of a call, the operand only has an index of some register. In order to obtain the hook address, we would need to disassemble in reverse and look at the last point in code where this register was written into from memory, which would give us the required hook address. Traditional forward disassembly of code disassembles instructions starting from the base address and moving upwards to higher memory locations, while reverse disassembly typically needs to disassemble instructions at memory locations that are lower than a base. But such reverse disassembly on `x86` is never an easy task, mostly due to the fact that there is no knowledge of where the previous instruction begins or what its length is. This makes reverse disassembly a clumsy and inaccurate process. But we still somehow need to know what the previous instructions are.

We get at them through traditional forward disassembly, starting the disassembly from the point where the function begins, and constructing a *control flow graph* (CFG) of the current function. We can do this because we know that every function has to start with a prelude; so we first search back (from the return EIP) till the point where we find a valid prelude, then disassemble forward, while creating a CFG of the complete function.

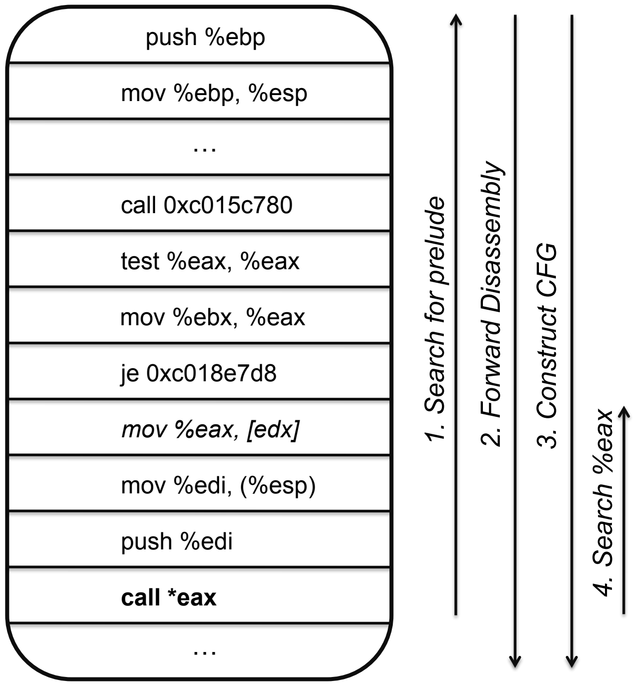


Figure 6.1: Steps undertaken by our system to find the hook address in case of a register-indirect call.

After constructing the CFG, we locate the node in which the `CALL` lies, mark the desired register that makes up the call, and search backwards from this node for the instruction that writes into the required register. In case the `CALL` contains multiple registers (such as base and indexed registers), we repeat this procedure for each individual register. To search backwards in the CFG, we simply reverse the edges of the graph and follow the paths. Within each node, we traverse its instructions in reverse. After locating this instruction(s), we determine the address of the hook by analyzing the contents of processor registers when this instruction(s) was executed by the kernel. The above process assumes that function text is organized as a sequence of instructions laid out consecutively in memory, which is the case with most Unix kernels. This technique is shown in Figure 6.1. Consider the following fictitious example of a single node in the CFG to illustrate our analysis of a register-indirect call:

```
1. mov ebx, 0xc0454f70

2. add edx, ecx

3. mov eax, [ebx+4]

4. cmp esi, edx

5. jne 0x300

6. call *eax

7. jmp 0x100

8. nop
```

The register-indirect call (using register `eax`) is at instruction 6, but just analyzing the contents of this register gives us the target function (callee) address, and not the hook address. Searching backwards within this node for the instruction that wrote into `eax` lands us at position 3. After determining this, if we analyze the register contents at that instruction, and add 4 to the value of `ebx`, this gives us the hook address. Note that this example only illustrates how the search is conducted within a node; if the desired instruction cannot be found in the same node, we traverse backwards to the nodes leading up to the current one in the CFG, and repeat the procedure shown here.

6.3 Detecting the Hijacking of Hooks (Detection Phase)

To implement an anomaly-based system, we first have to characterize the behavior of a majority of existing rootkits that use pointer hooks. Recall that pointer hooks are essentially function pointers that are present in the kernel's data sections. The addresses of these hooks can be either static or dynamic, depending on where the hooks are located. We henceforth refer to the function that is called using a pointer hook as a *hook function*. A *rootkit hook function* is a function that lies within a rootkit, while a *kernel hook function* is one that lies within the kernel or in non-rootkit LKMs. In his book on rootkits [27], Joseph Kong describes hooking as a programming technique that employs handler functions to modify control flow. Typically, a hook function will call the original function at some point in order to preserve the intended behavior.

A hook function calling another hook function (diverting control away from the core kernel) is exactly the kind of behavior we wish to detect. Since the hook functions in both cases are called via indirect calls, and the arguments passed to these functions are somewhat identical (we elaborate more in Chapter 7), we should be able to detect such a scenario and report the presence of a rootkit. We insert probes on the final list of callee functions identified in the learning phase, and when these probes are hit, we scan for the possibility of the above attack vector. We also make the assumption that the kernel text is unmodified by the rootkit — this is not a limiting factor of our system since text modifications are easily detectable by existing systems [31] [42] [30], and so is a dangerous avenue for any rootkit trying to avoid detection. Moreover, having the entire region of kernel memory at its disposal, a rootkit can perform subtler attacks than simply overwriting kernel text, which can be detected almost instantaneously by such detectors. Also, maintaining text integrity is an orthogonal problem which we do not tackle here.

Our method works as follows. During the learning phase, if the hook function (or callee function) is within the core kernel text, then we insert a probe on this function. Else, if the hook function is within an LKM, then we first insert a probe on the caller's `CALL` instruction. When this probe is hit, we disassemble the call, obtain the address of the callee, and insert a probe on it. After inserting probes this way, our system now enters a passive state, where the actions are triggered only when these probes are hit. Hence if a particular hook function is never called, then our system imposes no overhead on it. These initial probe handlers (also called Type I probe handlers) first determine whether the function was called through a pointer hook. To do this, we repeat the same procedure as in the learning phase, i.e. look at the return address on the stack, disassemble the last CALL instruction and verify that it is an indirect call. In case of a direct call, we return. Note that we still need to repeat this step, since the same function can be called both directly and indirectly from different parts of the kernel.

Our next step is to check for the presence of a rootkit. Let h be a hook, and F be the corresponding function we've inserted a probe on. We have two cases to handle:

1. *When a rootkit is installed while autoscopy is functioning:* In this case, the rootkit would modify h to point to its own hook function F' and since the original functionality must be unhindered, F' still calls F. The call chain now looks like $F' \rightarrow A \rightarrow B \rightarrow ... \rightarrow F$, where $A, B...$ might be intermediate rootkit functions. Since our probe is on F, we look up the call stack from F, and for each return EIP, disassemble the previous `CALL` and check if it is an indirect one. If so, then we additionally verify the arguments passed to that function

(in this case, F', since $A, B, ...$ would be direct calls), with the arguments that F got, and report the presence of a rootkit above the current stack frame.

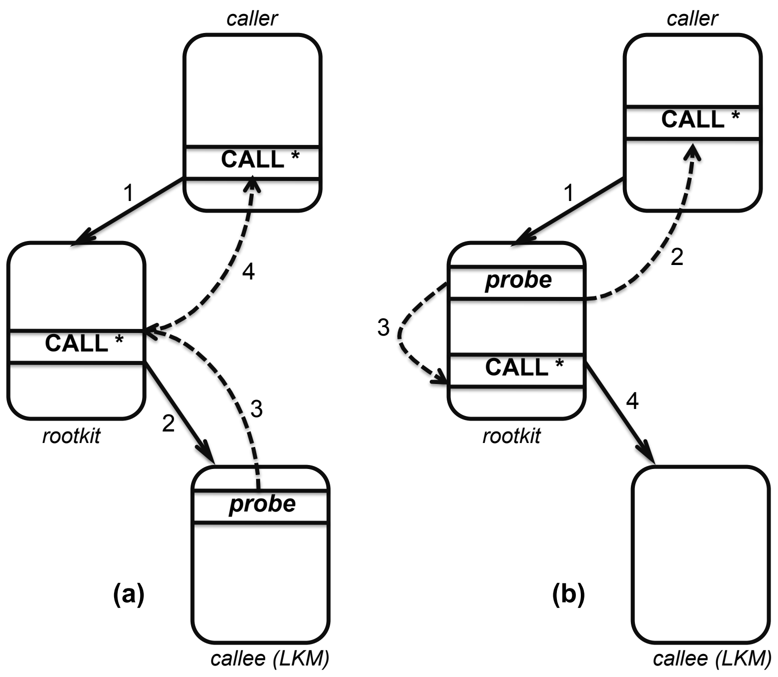


Figure 6.2: *Our Detection Process.* Autoscopy inserts probes on hooks to determine if control flow has been diverted to a new rootkit (a), or if control flow is returning from an already-present rootkit (b). The dotted lines indicate how the probe handlers detect the indirect calls. The numbering shows the order of control flow. In case (a), our probe resides in the original callee. In case (b), our probes land within the rootkit itself.

2. *When a rootkit is already present on the system:* In this case, the function that we put a probe on is the rootkit function itself. But we don't know that yet. The call-chain still looks like $F' \rightarrow A \rightarrow B \rightarrow ... \rightarrow F$, but the probe is now on F'. Since the probe is at the beginning of the function, we first disassemble F' in memory. Then we insert probes (also called Type II probes) on all functions that F' calls (both directly and indirectly). When these probes are hit, we repeat the same process i.e. disassemble the function and insert probes on all function calls. If F' (or $A, B, ...$) makes a direct kernel function call, we need not insert a probe on it since we assume the kernel text is unmodified. Hence we insert probes only on indirect calls and on direct calls whose target is not

within the core kernel.

We pass context information across all Type II probes, so that when they're "hit", they look up their context to determine what to do. The context supplied to direct function-call probes indicates the probe handler should continue putting probes on functions underneath it, while the context for indirect call probes indicates that the handler should check its arguments against those of F', while also continue inserting probes if the function is not within the core kernel.

So, in the above call-chain, the probe-handler for F' puts a probe on A, A's handler puts a probe on B, B's handler puts a probe on F, and F's handler checks its arguments against F' (since F is an indirect call from B), and detects the rootkit. Figure 6.2 details this process.

All Type I and Type II probes are also cached so we do not keep inserting and removing probes across these modules, which might diminish performance.

To insert probes on all calls within a function, we first construct a CFG of the function. Briefly, a CFG consists of basic-blocks that are delimited only by jump or return instructions. Each node in the CFG represents a contiguous sequence of instructions that has a single jump or return instruction at the end. An edge exists between two nodes n_1 and n_2 if the target of n_1's (conditional or unconditional) jump is some instruction in n_2, or if n_2 contains the successive instruction that follows n_1's conditional jump. The second case is necessary because in case of a conditional jump, the program can either take the jump, or ignore it and continue with the next successive instruction.

Chapter 7

Autoscopy's Implementation

7.1 Overview

We now detail our implementation of the learning and detection phases. For both these phases, the probe handlers that autoscopy installs need to be as fast and efficient as possible. Any latency incurred at this level might get exaggerated if the underlying functions are frequently called within the kernel. For the detection phase, though latency is not particularly an issue, since there are thousands of probes inserted by our system, we need to make sure the OS does not get significantly slow. For the learning phase, latency becomes a critical issue since it is the determining factor in measuring the performance and overhead imposed by autoscopy.

We also need an implementation of a disassembler within the kernel. We use the excellent *udis86* disassembler [11], which was modified and compiled to work within an LKM. Also, since we perform reverse disassembly (when disassembling the `call` instruction within the caller function), *udis86* was modified to support this mode of operation. The query interfaces of *udis86* and other library accessor functions were used to query the current instruction and the operands that were disassembled, and appropriate actions were taken based on the result.

7.2 Learning Phase

The possibility of thousands (or tens of thousands) of probes to drastically slow down or crash the system is not an unrealistic one, especially if the probe handlers are computation intensive. Moreover, since our probe insertion algorithm is unaware of the *location* of these probes, some of them may (and usually do) land in highly sensitive areas of the kernel (such as top halves, interrupt timers, core scheduling routines etc). Such areas of the kernel usually cannot suffer even a moderate delay in execution time, and the kernel is prone to crashes when instrumented uncouthly in these parts. Even "simple" operations such as tracing up a stack, or disassembling a single instruction, or removing an existing probe are relatively expensive since they might acquire locks, conduct exhaustive bounds checking, require initializations etc.

Hence, in order to reliably insert probes all across the kernel, our probe handlers in this phase are highly optimized and efficient. In order to avoid expensive computation within the handlers, we only collect the *context* of each probe, and delegate the actual workload to a user-space process. Collecting probe context (we explain what we mean by context shortly) is extremely fast, and thus allows us to insert a large number of probes without drastically affecting system performance. While performance is not a major concern yet, after inserting all our probes, we run LTP to exercise all parts of the kernel, which takes a few hours. Unoptimized probe handlers can stretch LTP's execution time exponentially. Hence we need to ensure that overall system response is still manageable.

Autoscopy's implementation of the learning phase is split into two modules. One module, which we call the Hook Analyzer (or *Analyzer* for short), walks the kernel memory (using */dev/kmem*), identifies all potential hooks, and subsequently feeds them into a kernel module (which we call the Hook Collector or simply *Collector*) through a driver interface. The *Collector* then reads this incoming stream of addresses and inserts probes on all of them. As mentioned before, since we do not really know in advance which functions in the kernel we're inserting probes on, and certain parts of the Linux kernel are extremely time-sensitive, we use this split-module approach to analyzing kernel hooks.

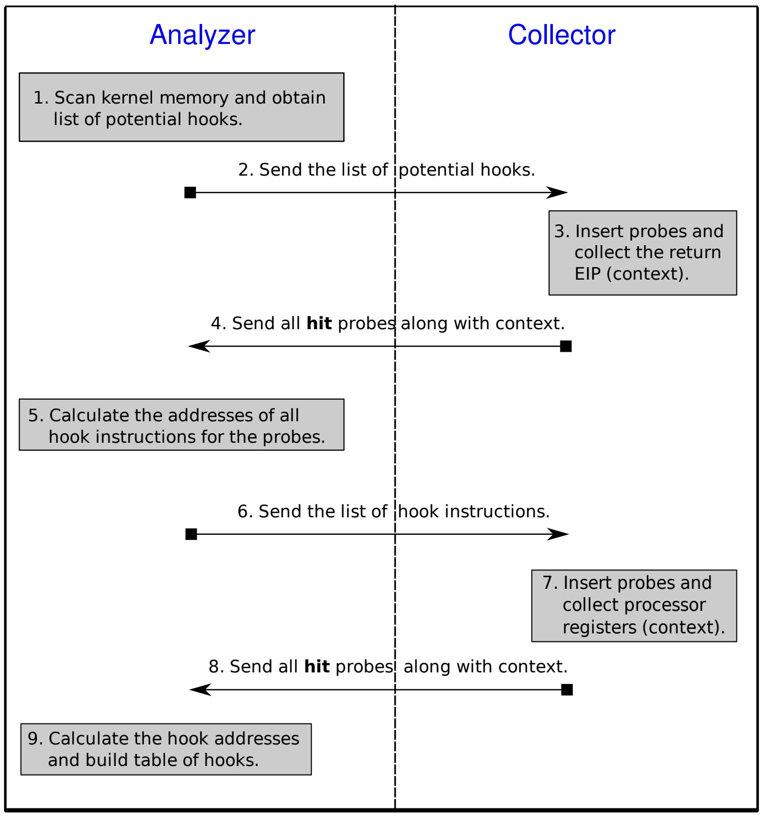

Analyzer
Collector
1. Scan kernel memory and obtain list of potential hooks.
2. Send the list of potential hooks.
3. Insert probes and collect the return EIP (context).
4. Send all **hit** probes along with context.
5. Calculate the addresses of all hook instructions for the probes.
6. Send the list of hook instructions.
7. Insert probes and collect processor registers (context).
8. Send all **hit** probes along with context.
9. Calculate the hook addresses and build table of hooks.

Figure 7.1: *Interaction between the Analyzer and Collector during our learning phase*

Figure 7.1 depicts the different steps in our learning phase and the basic interaction between the *Analyzer* and *Collector* modules. The *Collector* is implemented as an LKM and exports a driver interface that is used by both the modules for protocol interaction and data transmission. We now go through all the steps from Figure 7.1 in detail:

1. The *Analyzer* scans kernel memory through the *kmem* interface and uses the algorithm from section 6.2 to obtain the list of possible kernel hooks and callee

functions.

2. The *Analyzer* then sends this list to the *Collector*.

3. The *Collector* inserts probes on all the callee functions. The context of a probe at this point is the return instruction pointer (or return EIP) that identifies the caller function.

4. After the LTP tests complete, the *Collector* returns back the list of all probes that were *hit*. Before proceeding further, we first define a **hook instruction** as the instruction that contains the *hook address* in the form of processor registers and optionally an offset. In case of a memory-indirect call, the hook instruction is the CALL instruction itself, while for a register-indirect call, it is some instruction in the caller that is present *before* the call in the caller's control flow graph.

5. The *Analyzer* now has a list of possible caller/callee pairs that *might* use function pointers to perform the call. To verify whether a hook is indeed involved, the *Analyzer* uses the *kmem* interface to access the kernel function, disassembles the call, and if indirect, then disassembles the caller function, builds a control flow graph (CFG), analyzes the operands of the `call` instruction, and finally identifies the set of hook instruction(s) that contain the hook address.

6. Since we still need to obtain the processor registers when the hook instruction was executed, the *Analyzer* feeds its list of hook instructions (along with the caller-callee information) back to the *Collector*.

7. The *Collector* again inserts probes on all hook instructions and callee functions. The context of a probe now is the processor registers (instead of the return EIP). Hence, all probe handlers of the hook instructions collect this context, while all probe handlers on the callee functions associate themselves with the

hook instructions (to make sure that the registers are collected only when an indirect call is made, and not otherwise).

8. After a second round of LTP tests, the *Collector* returns the set of all *hit* probes along with their context to the *Analyzer*.
9. Finally, the *Analyzer* uses the context information to build a table of hook addresses, resolve all addresses into symbols, and also calculate function argument information (explained in section 7.3).

For symbol resolution, parsing addresses, and building a table of hooks, we use ruby scripts and a gdb back-end that uses debugging symbols from the kernel's build image (if available).

7.3 Detection Phase

We first detail the argument correlation process in autoscopy. Since the learning phase has the capability to determine the prototype of the callee function, we leverage this potential to calculate the number of arguments that a callee function expects. For functions within an LKM, since there is no possible way for us to determine this information, we use a heuristic and set the number of arguments as four for such functions. Argument *similarity* is defined as the number of arguments that are equivalent between two function calls. If this similarity metric exceeds half the total number of arguments for a callee function, and the attack vector we're searching for is satisfied, then we report the presence of a rootkit.

Broadly, our detection inserts probes on all input callee addresses from the learning phase, if they're present within the core kernel. But during the learning phase, if the callee was determined as being in an LKM, then the detection system first inserts a probe on the caller (present within the core kernel) and when the probe is hit, moves the probe from the caller to the callee (present within an LKM). Whenever a callee probe handler is called, both scenarios from section 6.3 are considered i.e. both when a rootkit might be installed after autoscopy (detectable by moving up the stack), and when a rootkit might already be present on the system (detectable by disassembling the callee and inserting Type II probes recursively). If either scenario reports a hit, then a rootkit is detected. Since we also make the assumption that the kernel text is unmodified, we make some performance enhancements this way.

We now describe the various components present within autoscopy's detection system. To reiterate, Type I handlers are installed for probes detected during the learning phase; all other handlers make up Type II. Figure 7.2 shows the various components within our system.

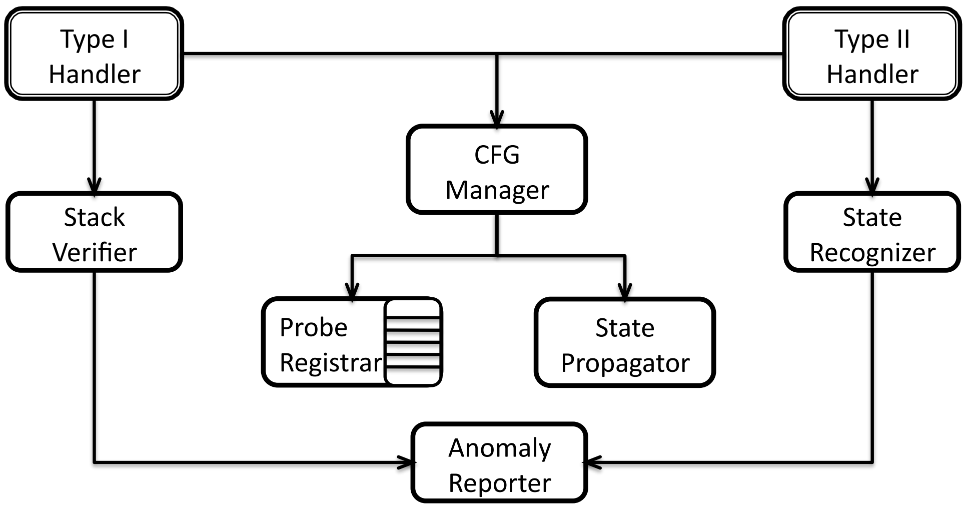


Figure 7.2: *Component interaction within autoscopy for the detection phase.* The two control flows originate from Type I and Type II handlers respectively.

1. **Stack Verifier:** The *Stack Verifier* is reponsible for ensuring that the current stack is untampered and also determining the origin and type of the function call. Reports an anomaly if it detects either an unusual stack or a control flow anomaly, such as the presence of two indirect calls within a single control flow that diverts flow from the core kernel to within an LKM.

2. **CFG Manager:** This component constructs the CFG for the desired function object, by using an inbuilt `x86` disassembler. It is called from both Type I and Type II handlers, and is responsible for recognizing both direct and indirect function calls within the current function, and inserting Type II probes, if so desired by the caller. It consults the **Probe Registrar** to manage the hierarchy of probes, and the **State Propagator** to propagate state information across Type II handlers.

   Maintaining state information for probes is necessary since Type II handlers are inserted for both direct and indirect function calls within LKMs; the former type of handlers need to continue inserting probes recursively on their own functions, while the latter need to verify their parameter list with the stack above.

3. **State Recognizer:** This component is only present within Type II handlers. Acquires and validates the state information handed down by either Type I handlers or other Type II handlers. Also checks the current control flow and raises an alert in case of an anomaly.

4. **Probe Registrar:** The *Probe Registrar* is responsible for the insertion and deletion of probes, and also for maintaining a record of active probes within the system. This record is necessary for when autoscopy is uninstalled, and the system needs to be cleaned of all probes inserted for the purposes of assisting detection. The registrar also caches all probes, instead of continuously inserting and removing them, which would have drastically affected system performance.

5. **State Propagator:** This component propagates the state information from either Type I or Type II handlers. We note that state information is maintained as a part of the probe structure within the kprobes framework. This is an optimization (and can also be considered as a tiny hack) that allows us to quickly get the context information associated with a given probe. This relieves

us from the task of building a hash table of probe/context, which would have been complicated, unnecessary, and expensive.

6. **Anomaly Reporter:** The *Anomaly Reporter* is responsible for reporting and logging any detected anomaly, along with the present state of the stack and any argument values associated with the function where the anomaly was detected.

7.4 Constructing a CFG

Constructing a CFG reliably and efficiently is critical for our system to be functional. As mentioned before, we use udis86 as our disassembler of choice. We maintain a queue of outstanding addresses that is initially empty. The construction of control flow graphs is undertaken only for functions, hence the starting address of a function is taken as input. Given this address, the CFG construction proceeds by disassembling each succeeding instruction one by one and adding it to the current node in the graph. If the disassembled instruction is a `jmp`, `ret` or `call`, then a new node is created.

If the instruction is an **unconditional jump**, then we take the jump (if not already taken before) and follow the disassembly from the jump target. Else if the jump target has already been disassembled before, we keep removing addresses from the head of the queue until we find one that has been untraversed. We quit if the queue becomes empty.

If the instruction is a **conditional jump**, we add the jump target to the tail of the queue, and resume from the succeeding instruction. Else if the instruction is a **return**, we again keep removing addresses from the queue, until we can proceed with the disassembly. The algorithm ends when the queue has become empty.

Thus, our implementation of autoscopy is both modular and optimized for efficiency. While there is the additional overhead of kprobes, as we show in the coming chapter, the overall performance overhead is still within agreeable limits.

Chapter 8

Evaluation

Our evaluation focuses on illustrating both the efficacy of autoscopy as a detection technique and its performance impact on a running kernel. We first give coverage details for all hooks identified through our learning process, and also illustrate our findings by giving a partial table of descriptions for all *resolved* hooks (hooks determined during our learning phase). Ideally, our full table serves as a catalogue for kernel hooks that could be used by other detection agencies aiming to protect the kernel. While it is not possible for us to claim that our kernel coverage is 100%, assuming the LTP tests are extensive in breadth, these hooks should cover most of the operating kernel.

Also, our learning and detection phases need not be limited by our existing implementations. Our underlying technique is general enough to be accommodated by a VMM, if so desired, isolation being the only advantage that can be reaped from such an implementation. While we have not thought hard about the VMM implementation, our experience with QEMU makes us believe that it should be feasible (and simpler to implement than our current code), especially because of QEMU and VirtualBox's technique of dynamically translating [16] entire basic-blocks into host instructions. Such techniques allow us to easily install handlers for indirect call and jump instructions, greatly reducing code complexity.

To accommodate embedded platforms that have little native support for installing external systems such as CoPilot (which uses a PCI interface), we advocate the use of architecture-specific functionalities that prevent the execution of code from a memory page that has been specifically tagged as non-executable. The modern x86 architecture provides the NX bit, while the ARM architecture (ubiquitous on embedded systems) has the “Execute Never”/XN bits within TLB entries.

If available, the Trusted Platform Module (TPM) is also an effective platform to detect violations in code integrity. A TPM is a passive hardware module that provides secure storage and cryptographic computation outside the system processor. The TPM offers a set of platform configuration registers (PCRs) that store hash values from any system measurement. The process of saving a measurement hash to a PCR is called “extending” a PCR, wherein the TPM concatenates the existing hash with the new one, calculates the hash of this concatenated value, and finally saves it onto the PCR.

We propose extending the appropriate PCR registers with the contents of the text, and verifying the register contents either periodically, or when a probe is hit. This would help detect modifications in static code or data.

Finally, if the MMU itself can be modified, then our proposal for a better policy interface at the MMU, via interaction with a customized FPGA [17] efficiently guarantees text integrity, along with other complex policy enforcements.

Hook Address	*Hook Symbol*	*Caller Function*	*Callee Function*
c035b64c	sync_cmos_timer + 12	run_timer_softirq	sync_cmos_clock
c035be0c	cpuinfo_op +8	seq_read	c_next
c035e158	modules_op +8	seq_read +493	m_next
c0363c38	key_type_keyring +24	key_cleanup	keyring_destroy
c0314390	memory_fops +48	chrdev_open	memory_open
dfe9e564	-	do_tty_hangup	pty_flush_buffer
c0374ba0	ipv4_specific	tcp_transmit_skb	ip_queue_xmit
c0374c14	tcp_prot +20	inet_create	raw_init
c031a790	unix_dgram_ops +16	sys_connect	unix_dgram_connect
c035fdc0	elf_format +8	search_binary_ handler	load_elf_binary
c03602f8	proc_root_operations +24	vfs_readdir	proc_root_readdir
c0372eb0	socket_file_ops +80	file_send_actor	sock_sendpage
c035f4bc	read_pipe_fops+28	do_poll	pipe_poll

Table 8.1: *A partial listing of hook addresses and symbols with associated function pairs, on Linux kernel 2.6.19-7.*

8.1 Discovering Kernel Hooks

A brute-force search for hooks by scanning */dev/kmem* identified 65,213 possibilities. However, a large portion of these hooks were duplicates (they referred to locations with the same value). Our next step was to filter out redundant hooks - after this, the number of hook possibilities drastically reduced to 9155 (a whopping 86% reduction). The next step, as described in section 6.2, was to insert probes on these locations, run the LTP suite, and gather all *hit* probes. This phase collected 28.7% of all inserted probes, or 2629 possible hooks. Finally, filtering these locations based on whether they were actually called using indirect calls led us to identify 634 hooks (of these about 160-odd were system calls. Table 8.1 shows a partial listing of kernel hooks discovered by autoscopy. A complete listing of hooks and their associated caller/callee function pairs is available on our website[1]

[1]`http://www.cs.dartmouth.edu/~pkilab/autoscopy`

8.2 Detection Ability

We tested autoscopy's ability to detect seven different kernel rootkits (in addition to our proof-of-concept pattern searching rootkit, and a SCADA Modbus rootkit developed for purposes of demonstration). For eight other rootkits that were impractical to compile and install into 2.6 kernels, we chose to conduct a source-level analysis in order to demonstrate our detection abilities. We note that a majority of publicly available rootkits are released to just demonstrate a proof-of-concept, and accordingly our analysis targets rootkit techniques and not specific implementations. The rootkits we tested against are shown in Table 8.2.

For the rootkits we tested, we modified them to properly resolve the kernel's system call table (since these rootkits modified the table and it is no longer exported in Linux 2.6.x). As expected, we were able to successfully detect and report all the rootkits, both when they were pre-installed before our system began operation and when they were installed after our system was present. We point out that our approach is agnostic to the particular content or feature set of a rootkit. Rather, we detect the behavior a rootkit must undertake to invade a system. In some sense, our detection can be considered rootkit-agnostic (our detection approach is analogous to efforts that focus on vulnerabilities rather than exploit content).

8.3 Detailed Rootkit Evaluations

We now evaluate our system against source-code analyses of two rootkits: `adore-ng` and `enyelkm`. Our initial aim is to expose the hooks used by these rootkits so that autoscopy can detect the presence of both these instances of malware.

Adore-ng is used to hide files, processes and network sockets from an IDS. It works by hijacking hooks at the *virtual filesystem* (VFS) layer. The hook that is hijacked is

Rootkit	Type of Analysis	Technique	Placement
kbdv3	autoscopy	syscall hooking	*LKM*
override	autoscopy	syscall hooking	*LKM*
enyelkm v1.0	autoscopy	syscall hooking using IDT	*LKM*
Rial	autoscopy	syscall hooking	*LKM*
Synapsys v0.4	autoscopy	syscall hooking	*LKM*
Adore–ng 2.6	source-code	*VFS* and *proc* hooking	*LKM*
DR v0.1	source-code	syscall hooking using DR	*LKM*
knark–2.4.3	source-code	syscall hooking	**Both**
linspy2beta2	source-code	syscall hooking	*LKM*
phalanx–b6	source-code	syscall hooking	*/dev/mem*
Rkit–1.01	source-code	syscall hooking	*LKM*
superkit	source-code	syscall table duplication	*/dev/kmem*
modhide1	source-code	syscall hooking	*LKM*

Table 8.2: *Rootkits detected by our system.* We distinguish between autoscopy and manual source-code analysis.

the `readdir` pointer contained within the `file` structure for all files under a particular filesystem. The hook is modified to point to `adore_proc_readdir`, adore's handler, which then calls the original function with modified arguments. The filesystems that adore-ng considers are the root filesystem and the proc filesystem. The following code sample from the rootkit illustrates how it hijacks these hooks:

```
if (orig_readdir) *orig_readdir = filep->f_op->readdir;
filep->f_op->readdir = new_readdir;
```

The rootkit first saves the original pointer in `orig_readdir`, and then overwrites it to point to `new_readdir`. Typically, `new_readdir` is the rootkit's own function. We now show that our system exposes these hooks that are used by adore. Table 8.3 is a partial capture of our complete list that gives details of just the pertinent hooks. In this case, there are only two: the readdir hooks of the proc and ext3 filesystems. The actual structures used are `proc_root_operations` and `ext3_dir_operations` - both of type *file_operations*. Note that the hook offset within the structure (+24) is the same in both cases since it is essentially the same hook, just present within two different objects. Thus we see that we're able to successfully expose both hooks.

Hook	*Caller*	*Callee*
proc_root_operations+24	vfs_readdir	proc_root_readdir
ext3_dir_operations+24	vfs_readdir	ext3_readdir

Table 8.3: *Hooks used by the adore rootkit.* Autoscopy was able to successfully discover both hooks.

Hook	*Caller*	*Callee*
sys_call_table	sysenter_past_esp + 79	sys_kill
sys_call_table	sysenter_past_esp + 79	sys_getdents64
sys_call_table	sysenter_past_esp + 79	sys_read

Table 8.4: *Hooks used by enyelkm.* Autoscopy was able to discover these hooks and detect the rootkit.

Enyelkm is a kernel rootkit that modifies the kernel functions `system_call` and `sysenter_entry` to redirect control to its own handlers while also maintaining a partial copy of the system-call table so as to not modify the original (since it could be integrity-checked by IDSes). Enyelkm redirects system-calls `sys_kill`, `sys_getdents64` and `sys_read` to its own functions that can then give root access to arbitrary processes, hide modules, files, directories, processes or even hide chunks within a single file. Our system exposes all three hooks modified by enyelkm and Table 8.4 gives specific details on these hooks. Since this particular rootkit does not modify the original hooks, simple integrity checkers on the `sys_call_table` structure would fail to detect this rootkit. However, since autoscopy uses control flow anomaly detection, we were able to successfully detect the presence of enyelkm within our system.

8.4 Evaluating Performance Impacts

As explained in Section 7, our detection system inserts probes on hooks identified during the discovery phase. With these probes in place, we ran a series of experiments to determine the overhead of running our system versus an uninstrumented version of the kernel. We first conducted three non-synthetic benchmarks: compiling the Apache 2.2.10 Web server, creating a file from */dev/urandom*, and compiling the

Linux kernel 2.6.19.7. Since these tasks are largely I/O-bound, we expect them to exercise the kernel significantly more than a CPU-bound process.

We have also determined the impact of two standard benchmarks: the SPEC CPU2000 benchmark suite and lmbench. Both these benchmarks offer a wide range of micro-benchmarking utilities that are useful for stressing particular components of the system and for streamlining results. In the tables that follow, when reporting the performance overhead, a minus sign before the value indicates that our autoscoped system performed better than a native system. In these cases, we assume that the particular scenario did not heavily exercise our autoscoped hook locations. For these results, we also assume that our measurement values were within the statistical error. A plus sign before the value indicates that our system performed worse than the native and the percentage value quantifies this overhead.

Benchmark Name	**Native (s)**	**Autoscoped (s)**	**Overhead**
164.gzip	458.851	461.66	+0.609%
168.wupwise	420.882	419.282	-0.382%
176.gcc	211.464	209.825	-0.781%
256.bzip2	458.536	457.16	-0.303%
254.perlbmk	344.356	346.046	+0.489%
255.vortex	461.006	467.283	+1.343%
177.mesa	431.273	439.97	+1.977%

Table 8.5: *SPEC CPU2000 Results.* We rebooted the machine between the tests on native and autoscopy. Each SPEC experiment was run three times: we report the median of those runs. In the last column, a plus sign indicates that our system performed worse than the native, and a minus sign means autoscopy imposed no measurable overhead.

We conducted seven different components of the SPEC benchmark; our results are shown in Table 8.5. The first five benchmarks were picked for their capability to stress parts of the filesystem and I/O, while the last two were mostly CPU intensive. As our results indicate, autoscopy performed on par with many benchmarks. Our "worst" performance was a 1.9% overhead for the OpenGL `177.mesa` benchmark.

lmbench [19] is a micro-benchmark tool for conducting performance analysis,

Latency Measurements	**Native**	**Autoscoped**	**Overhead**
Simple syscall (μs)	0.1230	0.1228	-0.163%
Simple read (μs)	0.2299	0.2332	+1.415%
Simple write (μs)	0.1897	0.1853	-2.375%
Simple fstat (μs)	0.2867	0.2880	+0.451%
Simple open/close (μs)	7.1809	8.0293	+10.566%
Bandwidth Measurements	**Native**	**Autoscoped**	**Overhead**
Mmap Read (Mbps)	6622.19	6612.64	+0.144%
File Read (Mbps)	2528.72	1994.18	+21.139%
libc bcopy unaligned (Mbps)	6514.82	6505.84	+0.138%
Memory Read (Mbps)	6579.30	6589.08	-0.149%
Memory Write (Mbps)	6369.95	6353.28	+0.262%

Table 8.6: *lmbench Results.* Reported values are medians obtained from multiple runs.

and we use it here to measure the latency of system calls, and also for measuring bandwidth. For the bandwidth measurements, lmbench repeats each test for varying amounts of data that is transferred, from about 512 bytes to 536MB. We report the values in Table 8.6 for a median transfer size of 0.524MB. From the results, we again notice that the performance of autoscopy is almost head to head with the native system. But there seems to be an anomalous result for the "File Read" bandwidth measurement. To discover the cause for this 21% overhead, we compare the overall number and frequency of probes that were hit when this particular measurement was conducted against the "Mmap Read" bandwidth measurement (which showed only 0.144% overhead). The only major difference between the "Mmap Read" and "File Read" measurements are that the former maps a file into memory and works on the mmapped version while the latter reads from disk every time.

From our measurements, we observed that 32% of probes were hit in both cases and the probes were hit 298,632 times in the case of "File Read", and 223,341 times in the case of "Mmap Read". Thus we can conclude that the overhead of the probe handlers is relatively consistent across these two measurements, and the actual cause of the overhead is related to the massive amounts of I/O being performed in the case of “File Read”. Our hypothesis is that either the kernel is preempting the I/O path or interfering with disk caching when probed, thus causing the identified overhead.

Benchmark Name	Native *(s)*	Autoscopy *(s)*	Overhead
Apache httpd 2.2.10 Compilation	184.090	187.664	+1.904%
Random 256M File Creation	141.788	147.78	+4.055%
Linux kernel 2.6.19.7 Compilation	5687.716	5981.036	+4.904%

Table 8.7: *Non-synthetic Benchmarks.*

Our first non-synthetic test was to compile Apache under our system and compare this with an Apache compile using a pristine kernel. We repeated the experiment thrice under each environment, and have reported the median values in Table 8.7. Our results show that we suffer only 1.9% of overhead. For the file creation experiment, we ran `dd` specifying a block size of 1KB to create a 256MB file from */dev/urandom*. Again, we report the median values and in this case, we note that our performance drops a little, but not considerably, to about 4%.

Finally, we compiled the Linux 2.6.19.7 kernel, and the median numbers in this case indicate just less than 5% overhead. Since the kernel compilation process is thorough and rigorous, stressing both the computation and I/O performance of the system, we take this to be our worst possible case.

8.5 False Positives and False Negatives

A false negative scenario arises whenever autoscopy *fails* to detect a control-flow altering rootkit when one is actually present on the system. But such a case might occur only if the underlying hook that the rootkit modifies is not being “protected” by autoscopy. This is not a likely scenario since our discovery system is comprehensive enough to cover all of the kernel hooks, as we rely on LTP to hit every area of the kernel.

On the other hand, a false positive arises if autoscopy reports a kernel rootkit when none actually exists on the system. With our design from Section 6.3, it is possible to obtain false positives under the following scenario. We first define a *hook type* for a hook as the combo `structure_offset`, where `structure` refers to the type of structure (C types as identified in debugging symbols) under which the hook lies (for global hooks under no structure, the type is `global`), and `offset` is the offset of the hook within the structure.

Now, a normal LKM installs a handler (say h_1) for a hook of type t_1 and another LKM installs a handler (h_2) for a hook of a different type t_2. When h_1 is called by the kernel, it in turn calls (a direct call) some other kernel function which eventually calls h_2. Hence the control flow is: $kernel \rightarrow h_1 \rightarrow kernel \rightarrow h_2$. Since h_1 and h_2 are valid LKM functions, this is a valid control flow, but the presence of the two indirect calls (to h_1 and h_2) would be detected by autoscopy and flagged as a rootkit (assuming the argument lists are similar). To accommodate this scenario, we enhance our implementation with a *type checker*.

Notice that the underlying hooks for h_1 and h_2 are of different types t_1 and t_2. In fact, these types shouldn't ever be of the same type, because that would violate the very concept of hooks — to provide an abstract implementation layer, where each hook accomplishes a specific functionality. For example, one would never see a `read` hook function turn around and call another `read` hook function — it shouldn't since it is itself supposed to accomplish the "reading" functionalities. Thus, hook functions of the same type should never call each other, but can call hook functions of a different type. This is what we're trying to distinguish, although in practice, we've never seen hook functions calling other valid hook functions before.

We first build hook types for all the hooks we detect. For hook addresses that can be resolved into symbols, we implement a DWARF [4] parser on the kernel's debug image to track down the required symbols, identify the type of structure it's enclosed under, and the offset of the hook symbol within that structure. We feed this along with the list of hooks to the detection system. For hooks that cannot be resolved since they lie within dynamic memory regions, we use the caller's function name for `structure` and the offset of the call from the start of the function for `offset`.

Our detection system then associates each probe with the corresponding type of hook, and whenever the system identifies a rootkit, it also verifies whether the types of hooks used within both indirect calls are identical or not. If identical, or one of the types is missing, then the rootkit is reported, otherwise it constitutes a false positive and is logged as such without raising any other critical alarms.

8.6 Threats against Autoscopy

We now consider a few of the possible attacks that a rootkit might conduct against autoscopy and also detail our countermeasures. We reiterate that since autoscopy operates at the same privilege level as the attacker, by definition, we cannot prevent the attacker from overwriting kernel text or any self-modification of code. We thus rely on other approaches to guard the kernel text (taking breakpoint instructions into account): AMD/Intel's No-Execute (NX) bit, CoPilot [35], SWATT [12] and similar systems. We emphasize that, by themselves, these other approaches cannot detect even moderately sophisticated rootkits that employ some form of dynamic hook modification or control-flow redirection. Rather, we see these tools as building blocks to help systems such as autoscopy that are directed towards more devious rootkits than the ones that just modify static text or data. The following are some scenarios of how a rootkit might attempt to subvert autoscopy:

1. **After getting control, the rootkit modifies the hook to point to the original callee, then calls a kernel function that uses this hook to**

redirect flow to the orginal callee, and finally reverts the hook to point back to the rootkit: This attempt is easily foiled by autoscopy, which recognizes the two indirect calls even though they might potentially be situated far apart on the stack.

2. **The rootkit constructs CALL instructions manually, i.e. it saves registers and flags, pushes <args, return IP, current FP> on the stack, and executes a JMP to the target:** We detect this in the probe handler by verifying that the last executed instruction within every function on the stack is indeed a CALL instruction. If necessary, we can also impose that hook functions can only be called indirectly if the CALL originates from an LKM.

3. **The rootkit modifies its stack to remove the rootkit's call frame, then executes a JMP to the target:**. By analyzing the operand of each `call` instruction that we disassemble, and verifying that the call operand corresponds to the actual function that was called, we can detect such an attack. Since the rootkit's hook function is now basically "missing" on the stack, but the caller's `call` operand would still contain the address of the rootkit hook, autoscopy would be able to detect this attack.

4. **The rootkit creates an alternate stack that omits the rootkit's call frame, modifies the stack pointer to point to the new stack, and then executes a jump to the original callee:** This is just an elaborate version of the previous attack, and can be detected in the same way as outlined before.

Chapter 9

Related Work

We now discuss some of the related work in the area of rootkit detection. We consider both static and dynamic analyses techniques, and evaluate them against autoscopy.

9.1 SBCFI

Control-flow integrity (CFI) is a security technique to ensure that the execution paths taken by a program at runtime conform to a CFG that is compiled statically by analyzing the program source code. Petroni and Hicks present an imitation of CFI called state-based CFI or SBCFI [34] that validates the data structures of the kernel and the kernel state itself as a *whole* instead of tracking the individual execution branches as they occur. This has the advantage that the introspecting process can be external to the system (in a VMM), but opens up a *window* during which an ephemeral (or non-persistent) rootkit might be deployed.

Their learning process is done statically by analyzing the kernel code for global variables and tracking their usage across the kernel, and validating them at runtime. The static nature of their system and the inability to adapt their learning process to the rapidly evolving Linux kernel are some shortcomings of this approach. The unoptimized performance of SBCFI is also shown to incur close to 40% overhead on a typical machine running on Xen, which makes it quite unrealistic for embedded systems.

9.2 Static Binary Analysis

Kruegel, Robertson and Vigna [28] propose a method to prevent rootkits from entering the kernel by statically analyzing all the LKMs *before* they are loaded into the kernel. Using a symbolic execution technique that “simulates the program execution using variable names rather than actual values for input data”, the LKM’s behavior is determined to be either conforming or surreptitious. However, a static white-list of valid memory regions and kernel symbols are necessary in order to distinguish a malicious LKM from an benign one. Building such comprehensive lists is of course a huge problem for such large and constantly evolving systems. Another drawback of this approach is that it only monitors LKM rootkits leaving the door open for *kmem* rootkits, that as we’ve shown can be just as devastating to the kernel.

9.3 Paladin

Paladin [14] divides the system into two *protected zones*, which are safeguarded from illegal access from rootkits, or other kinds of malware. The *Memory Protected Zone* consists of the kernel hook tables and other static regions of the kernel that need to be protected from rootkits, while the *File Protected Zone* consists of system binaries and libraries that need to be prevented against modifications. Given the specifications of these zones, Paladin uses a VMM to monitor write accesses across the system for validity. Any time an invalid access into any of the above zones is detected, a process-tracker component then identifies the entire process subtree for the process that elicited the write and kills all processes within this tree. A Paladin driver resides on the guest machine to facilitate this process and interacts with the monitor to aid detection. This serves as an excellent prevention mechanism too, except that the specifications of *MPZ* and *FPZ* are again static and a comprehensive survey of them is infeasible. Also, just monitoring for writes to specified locations and preventing them does not prevent rootkits from hijacking function hooks within data structures since these locations are *meant* to be overwritten.

9.4 HookMap

Wang et. al [47] propose a technique to uncover hooks in the kernel by tracking the kernel-level control flow for an interested application using a QEMU-based virtual machine monitor. Their basic idea is quite similar to ours — a rootkit wishing to hide itself — in this case, from certain user-level applications such as ls, ps, netstat or explicit monitoring agencies such as Tripwire [10] or AIDE [1]. To do this, a rootkit would have to hook locations that lie in the execution paths of these programs. Hence tracking the kernel control flow for these applications would reveal those hooks (which, as we know, are just indirect function calls). Cataloguing the hooks would serve an IDS that leverages these findings to protect such pointers. HookMap has the distinct advantage of narrowing its playing field to present a concise description of just the necessary hooks that need to be protected within the kernel. Moreover, hooks that do not lie in the execution paths of any of these programs would still go undetected. Our aim is broader and we cover most of the kernel hooks in the data region that have a reasonable chance of getting manipulated by any rootkit.

9.5 NICKLE

The NICKLE system by Riley et. al [39] implements a shadow memory controlled by a VMM for the entire region of kernel memory in the guest machine. Upon guest bootup, all known authenticated kernel instructions are copied into the shadow memory, and at runtime each kernel instruction fetch is verified by comparing the shadow memory maintained by the VMM with the actual physical memory at that location. Any differences here would indicate the presence of a rootkit, and thus prevent it from executing on the guest system. LKMs also need to get authenticated before their insertion since NICKLE cannot distinguish between a valid and a malicious kernel module. A disadvantage of this kind of authentication scheme is that it needs to be manually performed every time a module is inserted into the kernel, and in-depth analysis is necessary to ensure that the LKM does not invalidate the kernel.

Chapter 10

Discussion and Concluding Remarks

Detecting kernel rootkits invites a wide range of approaches — from static binary analysis and data protection schemes to memory shadowing techniques. Most require either a static specification of vulnerable code and data (and hence are *fast*), or use VM technology to identify and shield the vulnerable portions from malicious code (comparatively slow). Autoscopy is at the crossroads of both these approaches and presents a *manageable* interface, which requires no VM complications and hence is insusceptible to VM overhead, while at the same time is extensive enough to capture and track the anomalous nature of misbehaving malcode.

Looking back to the period following the dawn of Unix, operating systems have been designed to be opaque, focusing more on providing a clean interface for kernel-user communication, rather than opening up the OS internals for any kind of meaningful analysis. This has recently changed, with Solaris' DTrace having probably forever changed the way we look at operating systems today, and other similar designs such as Kprobes cropping up in Linux and other OSes. These tools make it possible to peer deeper into the OS than was previously possible, to gather and accumulate state, and allow us to *reason* about these states in a meaningful manner.

Given this new kind of “tracing” power, the question that then needs to be answered is: How can we use such power responsibly? In this thesis, we attempt to provide just such an answer — by using OS tracing frameworks to protect the OS itself — hence the name autoscopy. Building a security layer within the OS to monitor itself also means we can leverage this intimacy to gather as much of kernel context as needed without the hassle of trying to bridge the VM semantic gap [20]. Finally, through this thesis, we hope that we can fuel more thoughts and ideas about using tracing technologies for better policy enforcement and privilege management at the OS level.

10.1 Future Work

As the implementation of autoscopy is not restricted to just using kprobes, future work would involve implementing our detection system using a VMM for systems that *can* afford to handle the extra overhead and slowdown in performance. One possible implementation we are considering is using a processor emulator such as QEMU [16] that also does dynamic instruction translation. This technique would allow us to quickly trap indirect calls as they're being executed on the guest machine, and insert our control logic to check for the presence of anomalies.

We could also make autoscopy more judicious and adaptive in its handling of kernel probes — for example, if the system performance drops below a certain level due to autoscopy, we could temporarily disable certain probes along critical paths in order to boost performance — or insert more probes if the system performance is exceeding anticipated thresholds. Basically, probe insertion/deletion can be made more intelligent depending on the runtime characteristics of the system.

Other future work would involve developing hardware-based security for our detection system. We can achieve rigid security by building hardware primitives that complement our system by providing better memory guarantees for both the kernel code and user level applications. We have done some preliminary work in this direction in our proposal for a Better Mousetrap [17].

10.2 Conclusion

In this thesis, we have shown the emerging capabilities of contemporary rootkits, and also developed an analysis engine for the kernel data, where we automatically recover function pointer hooks used within the kernel. We then leverage our findings by building a low-overhead, rigorous detection system for rootkits intercepting kernel control flow. We have also evaluated our system by showing its capability to detect real-world rootkits, and estimated its performance impact to below 5%. Thus we believe autoscopy to be a feasible implementation ready for deployment in today's resource-constrained environments.

Parts of Chapters 1 to 7, and 9 are based on material from my thesis proposal [38]. Parts of Chapters 7 and 8 are also based on a paper draft currently in submission.

Appendix A

The following two sections are code fragments from autoscopy's implementation. Section A.1 gives the code for installing probes on the caller, and subsequently deactivating them if the callee function is determined to be within the core kernel, or leaving the caller probes enabled if the callee is within an LKM. Section A.2 shows how we handle conditional branches while constructing a CFG.

A.1 Installing probes on the caller

```
/* if there isn't already a probe at the caller */
if(f_getkp(ptd->from) == NULL)
{
  kp = (struct kprobe*)kmalloc(sizeof(struct kprobe), GFP_KERNEL);
  tctx = (t_context*)kmalloc(sizeof(t_context), GFP_KERNEL);

  if(!kp)
  {
    EPRINT(``kmalloc failure!!\n'');
    return -1;
  }
  memset(kp,0, sizeof(struct kprobe));
  /* if the callee is within a module, we handle it accordingly
  within caller_handler_pre */
  kp->pre_handler = caller_handler_pre;
  kp->post_handler = NULL;
  kp->fault_handler = handler_fault;
  kp->addr = (kprobe_opcode_t*)(ptd->from);

  if(register_kprobe(kp) == 0)
  {
    tctx->stack_ctx = 0;
    tctx->type = ptd->type;
    tctx->args = ptd->args;
    kp->symbol_name = (char*)(tctx);
    probe_list[i].probe = ptd->from;
    probe_list[i].carpet[0] = kp;
    i++;
    /* if the callee is in the core kernel, disarm the caller probe,
    else leave it enabled */
    if(core_kernel_text(ptd->to))
    f_disarm(kp);
  }
  else
  return -1;
}
```

A.2 Handling a conditional branch during CFG construction

```
while(ud_disassemble(udp)) /* disassemble from function-start */
{
  mmnc = udp->mnemonic;
  if(mmnc != UD_Icall && !is_jmp(mmnc) && !is_ret(mmnc))
  continue;
  if(is_conditional_jmp(mmnc))
  {
    if(q_tail >= qnum)
    {
      q_tmp = (unsigned long**)kmalloc(qnum * 4 * sizeof(unsigned long*),
      GFP_KERNEL);
      memcpy(q_tmp, address_queue, qnum * sizeof(unsigned long*));
      kfree(address_queue);
      address_queue = q_tmp;
      qnum *= 4;
    }
    address_queue[q_tail++] = udp->pc + get_branch_target(udp);

    if(blk_count >= bnum)
    {
      blk_tmp = (struct block*)kmalloc(bnum * 4 * sizeof(struct block),
      GFP_KERNEL);
      memcpy(blk_tmp, basic_blks, bnum * sizeof(struct block));
      kfree(basic_blks);
      basic_blks = blk_tmp;
      bnum *= 4;
    }
    basic_blks[blk_count].begin = bstart;
    basic_blks[blk_count].end = ud_insn_off(udp);
    bstart = udp->pc;
    blk_count++;
  }
}
```

Bibliography

[1] AIDE. *http://sourceforge.net/projects/aide*.

[2] chkrootkit. *http://www.chkrootkit.org*.

[3] DR Rootkit. http://seclists.org/dailydave/2008/q3/0215.html.

[4] Dwarf Specification. http://dwarfstd.org.

[5] Elf Specification. http://refspecs.freestandards.org/elf/elf.pdf.

[6] Linux Test Project. *http://ltp.sourceforge.net*.

[7] NX Bit. http://lkml.indiana.edu/hypermail/linux/kernel/0406.0/0497.html.

[8] Rootkit Hunter. *http://rkhunter.sourceforge.net*.

[9] The Register. http://www.theregister.co.uk/2008/09/04/linux_rootkit_released.

[10] Tripwire. *http://www.tripwire.com*.

[11] Udis86. http://udis86.sourceforge.net.

[12] Arvind Seshadri and Adrian Perrig and Leendert Van Doorn and Pradeep Khosla. SWAtt: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2004.

[13] K.H. Baek, Sergey Bratus, Sara Sinclair, and Sean W Smith. Attacking and Defending Networked Embedded Devices. In *WESS '07*.

[14] Arati Baliga, Xiaoxin Chen, and Liviu Iftode. Paladin: Automated Detection and Containment of Rootkit Attacks. In *Technical Report DCS-TR-593, Rutgers University, Department of Computer Science*, 2006.

[15] Arati Baliga, Pandurang Kamat, and Liviu Iftode. Lurking in the Shadows: Identifying Systemic Threats to Kernel Data. *Security and Privacy, 2007. SP '07. IEEE Symposium on*, pages 246–251, May 2007.

[16] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Usenix ATC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.

[17] Sergey Bratus, Michael E. Locasto, Ashwin Ramaswamy, and Sean W. Smith. New Directions for Hardware-assisted Trusted Computing Policies. In *FTC '08: Conference on the Future of Trust in Computing.*

[18] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, page 2. USENIX Association, 2004.

[19] Carl Staelin and Hewlett-packard Laboratories. lmbench: Portable Tools for Performance Analysis. In *In USENIX Annual Technical Conference*, pages 279–294, 1996.

[20] P.M. Chen and B.D. Noble. When virtual is better than real [operating system relocation to virtual machines]. *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 133–138, May 2001.

[21] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly Detection Using Call Stack Information. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 62, Washington, DC, USA, 2003. IEEE Computer Society.

[22] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A Sense of Self for Unix Processes. *sp*, 00:0120, 1996.

[23] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is not Transparency: VMM Detection Myths and Realities. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[24] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[25] Julian Grizzard. *Towards Self-Healing Systems: Re-establishing Trust in Compromised Systems*. PhD thesis, Georgia Institute of Technology, 2006.

[26] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using Sequences of System Calls. *J. Comput. Secur.*, 6(3):151–180, 1998.

[27] Joseph Kong. *Designing BSD Rootkits*. No Starch Press, 2007.

[28] Christopher Kruegel, William Robertson, and Giovanni Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference*, pages 91–100, Washington, DC, USA, 2004. IEEE Computer Society.

[29] Andrea Lanzi, Monirul Sharif, and Wenke Lee. K-Tracer: A System for Extracting Kernel Malware Behavior. In *Proceedings of the* 16th *Annual Network and Distributed System Security Symposium (NDSS)*, 2009.

[30] Tim Lawless. St.Michael and St.Jude. *http://sourceforge.net/projects/stjude.*

[31] Peter A. Loscocco, Perry W. Wilson, J. Aaron Pendergrass, and C. Durward McDonell. Linux kernel Integrity Measurement using Contextual Inspection. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable trusted computing*, pages 21–29, New York, NY, USA, 2007. ACM.

[32] Ananth Mavinakayanahalli, Prasanna Panchamukhi, Jim Keniston, Anil Keshavamurthy, and Masami Hiramatsu. Probing the Guts of Kprobes. In *OLS '06*.

[33] Microsoft. Rootkit Revealer.

[34] Jr. Nick L. Petroni and Michael Hicks. Automated Detection of Persistent Kernel Control-flow Attacks. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 103–115, New York, NY, USA, 2007. ACM.

[35] Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the* 13th *USENIX Security Symposium*, pages 179–194.

[36] pouik and yperite. Zeppoo. *http://sourceforge.net/projects/zeppoo.*

[37] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Brad Chen. Locating System Problems Using Dynamic Instrumentation. In *OLS '05*.

[38] Ashwin Ramaswamy. Detecting kernel rootkits. Technical Report TR2008-627, Dartmouth College, Computer Science, Hanover, NH, September 2008.

[39] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In *RAID*, 2008.

[40] Riptech. Understanding SCADA system security vulnerabilities. *http://www.iwar.org.uk/cip/resources/utilities/SCADAWhitepaperfinal1.pdf.*

[41] Dragos Ruiu. Cautionary Tales: Stealth Coordinated Attack HOWTO.

[42] Joanna Rutkowska. System Virginity Verifier.

[43] Joanna Rutkowska. Introducing Stealth Malware Taxonomy. COSEINC Advanced Malware Labs, 2006.

[44] Joanna Rutkowska. Subverting Vista Kernel For Fun And Profit. In *SyScan*, 2006.

[45] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, volume 39, pages 1–16. ACM Press, December 2005.

[46] Vmware. A Performance Comparison of Hypervisors. `http://www.vmware.com/pdf/hypervisor_performance.pdf`.

[47] Zhi Wang, Xuxian Jiang, Weidong Cui, and Xinyuan Wang. Countering Persistent Kernel Rootkits Through Systematic Hook Discovery. In *RAID*, 2008.