

ABSTRACT OF THE DISSERTATION

Automated Detection and Containment of Stealth Attacks on the Operating System Kernel

By ARATI BALIGA

Dissertation Director:
Professor Liviu Iftode

The operating system kernel serves as the root of trust for all applications running on the computer system. A compromised system can be exploited by remote attackers stealthily, such as exfiltration of sensitive information, wasteful usage of the system's resources, or involving the system in malicious activities without the user's knowledge or permission. The lack of appropriate detection tools allows such systems to stealthily lie within the attackers realm for indefinite periods of time.

Stealth attacks on the kernel are carried out by malware commonly known as rootkits. The goal of the rootkit is to conceal the presence of the attacker on the victim system. Conventionally, kernel rootkits modified the kernel to achieve stealth, while most functionality was provided by accompanying user space programs. The newer kernel rootkits achieve the malice and stealth solely by modifying kernel data. This dissertation explores the threat posed by both types of kernel rootkits and proposes novel automated techniques for their detection and containment.

Our first contribution is an automated containment technique built using the virtualization architecture. This technique counters the ongoing damage done to the system by the conventional kernel

rootkits. It is well suited for attacks that employ kernel or user mode stealth but provide most of the malicious functionality as user space programs.

Our second contribution is to identify a new class of stealth attacks on the kernel, which do not exhibit explicit hiding behavior but are stealthy by design. They achieve their malicious objectives by solely modifying data within the kernel. These attacks demonstrate that the threat posed to kernel data is systemic requiring comprehensive protection.

Our final contribution is a novel automated technique that can be used for detection of such stealth data-centric attacks. The key idea behind this technique is to automatically identify and extract invariants exhibited by kernel data structures during a training phase. These invariants are used as specifications of data structure integrity and are enforced during runtime. Our technique could successfully detect all rootkits that were publicly available. It could also detect more recent stealth attacks developed by us or proposed by other recent research literature.

Table of Contents

Abstract	ii
Acknowledgements	iv
Dedication	vi
List of Tables	x
List of Figures	xi
1. Introduction	1
1.1. Thesis	1
1.2. What is a Rootkit?	1
1.3. Attack Injection Vectors	3
1.4. Evolutionary Trends	3
1.5. Rootkit Attack Techniques	7
1.6. Commercial Rootkit Detectors	10
1.7. Rootkits Research	13
1.8. Monitoring Kernel Data Integrity	19
1.9. Contributions of this Dissertation	21
1.10. Contributors to this Dissertation	22
1.11. Organization of this Dissertation	22

2. Attack Containment	23
2.1. Virtual Machine Technology	24
2.2. Security Model	25
2.3. Approach	26
2.4. Design and Implementation	31
2.5. Evaluation	35
2.6. Discussion	42
2.7. Summary	45
3. Stealth Attacks on Kernel Data	46
3.1. Problem Statement	46
3.2. Disable Firewall	48
3.3. Resource Wastage Attack	50
3.4. Entropy Pool Contamination	53
3.5. Disable Pseudo-Random Number Generator (PRNG)	55
3.6. Intrinsic Denial of Service	57
3.7. Altering Real Time Clock Behavior	58
3.8. Routing Cache Pollution	60
3.9. Defeating In Memory Signature Scans	61
3.10. Attack Categorization	62
3.11. Summary	64
4. Attack Detection via Invariant Inference	66
4.1. Problem Statement	66

4.2. Approach	67
4.3. Invariants and Stealth Attacks	68
4.4. Design and Implementation	76
4.5. Experimental Results	86
4.6. Summary	93
5. Conclusions and Future Work	94
5.1. Concluding Remarks	94
5.2. Future Work	96
References	99
Curriculum Vita	104

List of Tables

2.1. Per system call performance	41
2.2. Performance overhead	41
3.1. Watermark values and free page count before and after the resource wastage attack	52
3.2. Performance degradation in applications after the resource wastage attack	52
3.3. Results of running the Diehard battery of tests after contamination of the entropy pool	55
4.1. Rootkits that modify control data	87
4.2. Rootkits from research literature.	89
4.3. Number of invariants inferred by Gibraltar.	90
4.4. Gibraltar false positive rate	90
4.5. Results from the Stream microbenchmark averaged over 100 iterations.	92

List of Figures

1.1. Evolution of rootkit attack techniques	4
1.2. Timeline showing evolution of rootkit research	13
1.3. Comparison of kernel data integrity monitors	20
2.1. Types of virtual machines	25
2.2. Protected zones in the file system and system memory	27
2.3. Sample Paladin policies	28
2.4. Automated containment in Paladin	30
2.5. The containment algorithm	31
2.6. The Paladin architecture	32
2.7. Linux rootkits publicly available, categorized by hiding techniques	37
2.8. Dependency tree shows the Lion worm attack	40
2.9. Multiple control processes	43
3.1. Kernel data structure affecting user level views	47
3.2. Hooks provided by the Linux netfilter framework	48
3.3. Firewall rules deny admission to the web server port	49
3.4. Zone balancing logic and the use of zone watermarks	51
3.5. The Linux random number generator	54
3.6. File and device hooks in the Linux virtual file system layer	56

3.7. Attack categorization	65
4.1. Invariants violated by the entropy pool contamination attack	70
4.2. Invariant violated by the hidden process attack	71
4.3. Invariant violated by the adding binary formats attack	72
4.4. Invariants violated by the resource wastage attack	73
4.5. Invariant violated by the disable firewalls attack	74
4.6. Invariant violated by the disable PRNG attack.	75
4.7. The Gibraltar Architecture	77
4.8. Algorithm used by the data structure extractor.	79
4.9. An example showing the CONTAINER annotation	80
4.10. Example of a transient invariant.	85

Chapter 1

Introduction

1.1 Thesis

This dissertation investigates the threat posed by malware (commonly known as rootkits) to the operating system kernel. Conventionally, kernel rootkits modify the kernel to achieve stealth, while most of the malicious functionality is provided by accompanying user space programs. This dissertation proposes an automated containment technique for such rootkits to stop the ongoing malicious activity and thereby minimizing the damage to the system. It further identifies a new class of stealth attacks on the kernel that achieve the malice and stealth solely by altering data structures in the kernel. Finally, it proposes a novel automated technique, based on invariants inferred over kernel data structures, for the detection of both types of kernel rootkit attacks.

1.2 What is a Rootkit?

The term "rootkit" was originally used to refer to a toolkit developed by the attacker, which would help conceal his presence on the compromised system. The rootkit is typically installed after the attacker has obtained root level privileges on the compromised system.

Despite efforts for decades, software continues to be buggy and vulnerabilities are frequently

found in applications as well as in operating system code. Exploiting a vulnerability gives the attacker access to the system. A root-level exploit provides the attacker with virtually a free reign over the compromised system. The proliferation of the internet and the ubiquity of computers with always-on internet connections running software with vulnerabilities have provided attackers with easy access to vulnerable systems on the internet. Attack motives have therefore shifted from criminal mischief to criminal profiteering. Attackers are interested in retaining access to a compromised system, which allows them to reuse the resources of the system for committing fraudulent or illegal activities. Rootkits facilitate such stealthy existence of the attacker on the victim system.

A compromised system can be exploited by remote attackers in several ways, such as exfiltration of sensitive information, wasteful usage of the system's resources, adverse effect on the system performance, or system involvement in possibly fraudulent or malicious activities. Further, these compromised systems are often made part of botnets, where the attacker might use the system as part of a larger collection of victim systems (the botnet). These collectively participate in illegal group activities, such as spam relays and distributed denial of service attacks. Because the rootkit conceals the fact that the system is compromised, a rootkit infested system can stay undetected for indefinite periods of time in the absence of appropriate detection tools.

Rootkits pose a serious and growing threat to computer systems today. Recent studies have shown a phenomenal increase in the number of malware that use stealth techniques commonly employed by rootkits. For example, a report by MacAfee Avert Labs [1] observes a 600% increase in the number of rootkits in the three year period from 2004-2006. Indeed, this trend continues even today; over 200 rootkits were discovered in the first quarter of 2008 alone (according to the forum antirootkit.com [2]).

1.3 Attack Injection Vectors

Rootkits are installed on a system that has already been compromised via some other method. The most popular method of gaining entry into the system is by exploiting software vulnerabilities, which usually happens without the user's knowledge or interaction. Most commonly exploited software vulnerabilities exist in the operating system and browser software. Users often lag in updating their systems with released security patches, which plays out to the attacker's advantage. Rootkits might also be injected when the user intentionally downloads software from untrusted web sites. Such software is often bundled with unwanted components, several of which might be malicious. Rootkits can also be externally delivered through spam mail, peer to peer file sharing applications, or through other attacks, such as worms or bots. They might also be installed as part of an insider attack.

While rootkit code can be injected into the system via the methods discussed above, it requires root level control to tamper with the kernel or inject software that runs below the kernel [3–5]. Root level access might be obtained by exploiting bugs in the operating system, such as buffer overflows, race conditions, or privilege escalation attacks. Code can also be injected in the kernel by exploiting bugs within device drivers. Alternatively, the attacker might gain root privileges directly as a virtue of the user being logged on with root permissions.

1.4 Evolutionary Trends

Rootkits attack techniques have matured over the past few years, posing a realistic threat to commodity operating systems. Comprehensive detection of such advanced rootkits is still an open research problem. The new attack techniques used by rootkits have in turn triggered the development of novel techniques to detect their presence. The evolution of rootkits and techniques to detect

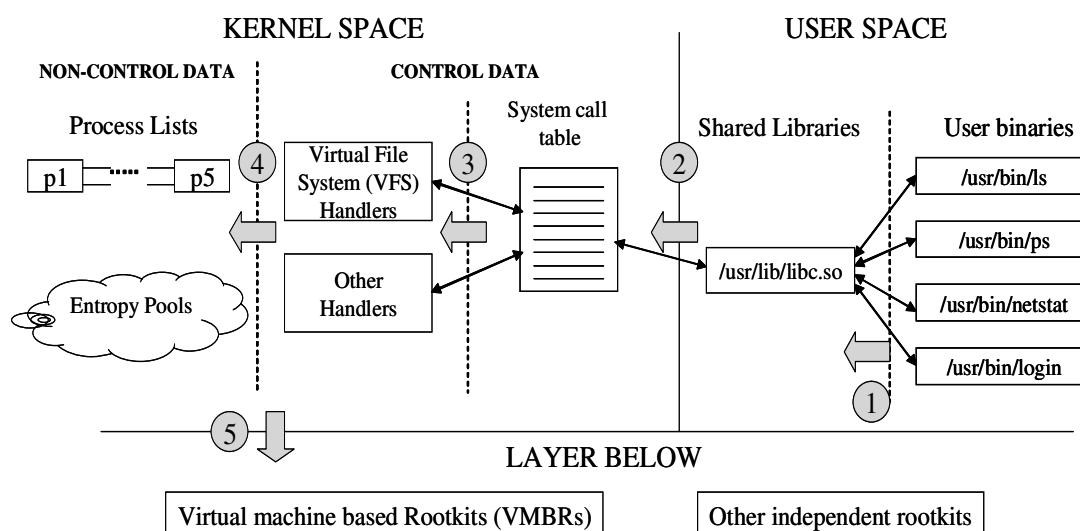


Figure 1.1: Evolution of rootkit attack techniques

them continues to be an arms race between attackers and defenders. Figure 1.1 shows the evolution in rootkit attack techniques. Rootkits have evolved from manipulating user space binaries and shared libraries to modifying control and non-control data in the kernel. The latest rootkits install themselves below the operating system.

Early rootkits operate by modifying system binaries and shared libraries replacing them with trojaned versions. The goal of these trojaned binaries is to hide malicious objects or grant privileged access to malicious processes. For example, a trojaned `ps` binary will not list the malicious processes running on the system. A trojaned `login` process can give root privileges to a malicious user. To detect trojaned system binaries and shared libraries, tools such as Tripwire [6] and AIDE [7] were developed. These tools generate checksums of authentic binaries when run on a clean system and store them in a database. A user can examine the system at later points in time, using these tools, and compare the checksums of system binaries with those previously stored in the database. A mismatch in checksum indicates the presence of the trojaned binary. Other detection tools used an anti-virus like approach, where the presence of a rootkit is detected using a database of known signatures, such

as a specific sequence of bytes in memory, or by the presence of certain files on disk. This approach does not protect the system against newer unknown rootkits. Rootkits could thwart such detectors by using polymorphic and metamorphic techniques for code obfuscation, traditionally used by viruses to escape detection from anti-virus programs.

To escape detection from disk based integrity checkers, rootkits have evolved to make modifications to kernel code and certain well known immutable data structures in the kernel, such as the system call table, to achieve the same goals. These rootkits are known as kernel-level rootkits because they modify the kernel. Modifications to the kernel make the rootkit powerful enough to control all application level views. For example, intercepting the file related system calls, allows the rootkit to control all files accesses by all applications on the system. The rootkit can intercept these accesses and perform the necessary filtering to hide its malicious objects. Since the rootkit manipulates the kernel, which is the trusted computing base of the system, it can also manipulate any user level applications on the system. Such applications include the rootkit detection tools that run in user space. Therefore, researchers proposed isolating the rootkit detectors from the operating system by either moving them onto a secure co-processor that does not rely on the operating system [8, 9] or isolating them using the virtualization architecture where the detector is run in a separate virtual machine [10, 11]. The rootkit detectors, built to detect the kernel level rootkits, use a checksum/secure hash based method to detect corruption of the kernel code or other well known immutable data structures in the kernel, such as the system call table. The hashes are pre-computed over the memory locations of a clean system, where the code and data structures are stored. They are periodically recomputed and compared with the stored hashes to detect code or data structure corruption [9, 10].

To further thwart detection tools, rootkit authors have adopted stealthier techniques. Since detection tools solely checked the integrity of the kernel code and some well known data structures,

such as the system call table, rootkits delved deeper into the kernel and altered data structures that were less known. For example, instead of modifying file related system calls in the system call table, rootkits modified hooks in the virtual file system layer instead. For a while, the arms race continued where the rootkit explored a new data structure that it could exploit, while the detector had to incorporate the newly discovered data structure in its verification list. Most of the data that the rootkits modified was immutable control data i.e. function pointers used by various layers in the kernel. An automated approach was later developed to uniformly check for manipulation of all control data in the kernel, by validating every function pointer against a valid kernel function address [12].

Since the integrity of mostly immutable control data can be verified, rootkit authors have advanced another step and have built innovative attacks that work by solely manipulating data structures that are mutable [13]. This defeats the existing integrity checking mechanism of storing checksums and performing periodic comparisons because these data structures are also modified by authentic kernel code. We demonstrated some attacks that worked by modifying relatively immutable non-control data [14]. These attacks modify variable values to alter the behavior of kernel algorithms. They escape detection because they manipulate non-control data within data structures not typically monitored by rootkit detectors. Detection approach was built to detect these advanced attacks using manual specifications, as long as the attack obeys some constraint [15]. This approach is effective as long as a manual security expert is capable of analyzing, anticipating and specifying the constraints on data structures that might become the target of future attacks.

More recent trends have shown rootkits that operate below the operating system layer. Researchers have demonstrated rootkits that use the virtual machine technology to subvert the system [4, 5] and rootkits that work independently of the operating system without requesting its services or affecting its state [3]. While these indicate a new trend in the development of rootkits, they are

likely to be unpopular because they are highly platform specific and depend on specific hardware features for their deployment. The operating system is still an attractive target because kernel level rootkits work independent of the hardware and can therefore be easily ported across different platforms. The kernel also provides a large code base and numerous amount of complex data structures, providing the rootkit authors with several avenues for building stealthy innovative attacks.

1.5 Rootkit Attack Techniques

A careful survey of the attack techniques used by rootkits developed by attackers provides an insight into the kind of solutions that need to be designed to counter them. We classify the attack depending on the layer that they attack namely (a) the application layer and (b) the operating system layer. We discuss attack techniques used by rootkits on the Linux and the Windows operating systems, which covers majority of the rootkits in existence today. Some of these techniques might also apply to other operating systems, such as BSD.

1.5.1 Application layer

Rootkits that reside in the application layer are also known as user level rootkits. In most of these attacks, the attacker does not require any form of kernel level control. Most of the stealth and functionality provided by the rootkit exists in the user space.

On Linux operating systems, user level rootkits have exhibited a solo trend. The purpose of all rootkits that we surveyed is to hide the presence of malicious objects installed by the attacker on the system. This is typically achieved by overwriting system binaries, such as *ps*, *top* and *netstat*, commonly used to inspect the system. More advanced versions achieve system wide control from user space by overwriting common libraries, such as *libc*, used by most applications on the system. They often have other additions that allow the attacker to take advantage of the compromised system, such as backdoors and key loggers. A detailed list of user level rootkits for Linux is available in

Figure 2.7.

Windows rootkits alter user level views of applications by manipulating their in-memory process image rather than overwriting system binaries on disk. Two features provided by the Windows operating system namely, the OpenProcess API and the Auto start extensibility points (ASEPs), enable easy deployment of such techniques.

The OpenProcess API on Windows, allows an application to retrieve a handle to another process running with the same privilege level. This allows a process to create a remote thread within another process. Windows also has a DLL injection feature where a process can inject a DLL in the process space of another process. Code injection into another processes's memory image is usually used for process monitoring and debugging in Windows applications. Rootkits use this feature to inject their own code into a remote process.

The other feature used by rootkits is ASEPs. There are four types of ASEPs, which allow different types of auto startup functions. ASEPs that start a new process allow processes to be started automatically on system startup. ASEPs that hook system processes allow a DLL to be loaded into a system process. ASEPs that load drivers allow loading of drivers and ASEPs that hook multiple processes allow a DLL to be loaded into every process that links with a particular DLL [16]. Rootkits use these ASEPs to carry out in-memory attacks that persist across system reboots.

Several Windows Enumeration API's are available as dynamically linked libraries (DLL). These dll's are linked by all user programs that use the Windows API. By intercepting calls to these dlls, either by changing the per-process Import Address Table (IAT) or by directly changing the in-memory API's [17] or the Export Address Table (EAT), the attacker can hide her own files and processes successfully. Some rootkits introduce an API detour. They modify the return address on the stack in such a way that they get called on the return path from the API call and can alter the

results.

1.5.2 Operating system layer

Rootkits that alter the operating system kernel or the device drivers are known as kernel level rootkits. Kernel level rootkits are dependent on the operating system and might change across different versions of the OS.

Linux kernel rootkits comprise of a range of rootkits that alter several data structures in the kernel. The simplest rootkits exhibit hooking behavior at different layers in the kernel. The most popular target of these rootkits is the system call table. Hooking into this table allows the rootkits to intercept all application level system calls and alter their view of the system by filtering requests or responses. The goal is still to hide malicious objects belonging to the attacker on the system. Some rootkits bypass the regular system call table and install a new system call table of their own. The system call handler is modified to jump to its newly created system call table. Other rootkits modify the interrupt descriptor table. The trend exhibited by these rootkits is to modify data structure hooks that exists along the system call path, the ultimate goal being to hide user level objects in the system.

The Windows kernel rootkits primarily use three techniques - (a) Modifying the System Service Table (SST) or the IDT, (b) modifying the interrupt request packet function table, or (c) Direct kernel object manipulation (DKOM). The first category of rootkits, which modify the SST or IDT, is analogous to those modifying the system call table and the interrupt descriptor table on Linux. The second technique is to hook into the function table installed by a device driver. This table can be replaced with rootkit addresses to perform filtering, such as hiding network ports. The final technique used by Windows kernel rootkits is called direct kernel object manipulation, which involves modifying non-control data for hiding purposes. One such technique used by the rootkit *fu* [13] is to hide a process by removing it from the process list maintained by Windows. This hides the process from tools, such as Task Manager, which refer to this list to enumerate processes running on

the system. The rootkit process still gets scheduled because it present in the list of processes used by the Windows scheduler.

1.6 Commercial Rootkit Detectors

Commercial rootkit detectors in existence today run on the same system as the system being monitored, much like the anti-virus running on the system. Though many of them use techniques to hide themselves from rootkits, they basically operate at the same privilege as the rootkit itself and therefore, have an equal footing. They use different techniques to detect rootkits, which are discussed in detail below.

1.6.1 Signature based detectors

Signature based checking is the most primitive technique used mostly by anti-virus software. The idea behind signature based checkers is to identify the rootkit based on its unique signature, which might be a sequence of bytes in memory or existence of some unique files on disk. Several tools implement the signature based checking technique. Chkrootkit [18] checks for modified versions of system binaries and loadable kernel modules by looking for known strings within the binary. It also checks for other signs for rootkit such as log file modifications. Rootkit Hunter [19] uses some amount of signature based checking by scanning binaries and kernel modules for suspect strings.

1.6.2 Integrity based detectors

Tools such as Tripwire [6] and AIDE [7] check the integrity of system binaries on disk. These tools, in the initialization phase, are run on a clean system. They create checksums of clean system files and store them in a database. At later points in time, the integrity of system binaries can be verified by calculating their checksums and comparing the checksums against those stored in the database. Integrity based checkers have an advantage over signature based checkers as they can detect rootkits not previously known. Detection is based on the change that is made to a protected

binary.

System Virginty Verifier [20] is a memory based integrity checker that verifies the commonly used Windows components against a known good state. It checks the integrity of commonly compromised data structures in the kernel, such as the SSDT, IDT and IRP function tables. It also uses some heuristics to counter the number of false positives generated by authentic hooking performed by Windows applications, such as anti-virus software running on the system.

1.6.3 Cross view based detectors

Cross view based checkers can detect rootkits that are hidden in different layers between the user space and the kernel space. This idea was first proposed by the Microsoft Strider team in their prototype Ghostbuster [17]. Cross view detection is based on the concept of querying the system from multiple view points and then comparing the generated answers. Any discrepancies between the two views indicates the presence of a rootkit. Cross view based detection uses two types of scans (a) Inside the box scan and (b) Outside the box scan. In the "inside the box" scan, a user level view is obtained by querying the OS enumeration APIs. Another view is generated by traversing the low level kernel data structures. A difference between these two views reveals rootkits that are hidden between the user and kernel space. An "outside the box" scan compares the low level view of the system to a low level view of a clean system, which can be obtained by rebooting the same system from a clean media. A difference in these views indicates the presence of a kernel rootkit. The efficacy of this detection technique largely depends on how the tool is implemented.

Microsoft Rootkit Revealer uses the "inside the box" scanning technique for rootkit detection. Therefore, it can only detect rootkits that hide between user and kernel space. F-secure Blacklight uses a similar cross view based approach to detected hidden processes and hidden files [21]. Klister is another detector that detects rootkits that hide by direct manipulation of kernel objects [22]. It consists of a set of utilities for Windows, specifically aimed at detecting hiding processes.

1.6.4 Behavior based detectors

VICE is a rootkit detector for Windows that checks for hooks in user space processes as well as within the kernel [23]. It is a standalone program that installs a driver that scans the kernel for hooks. It uses a policy that every function pointer should resolve to a code address within the kernel or the process, when looking for process hooks. In the kernel, it checks for IDT and SSDT and ensures that the function pointers point to kernel code. It checks the IRP function tables in the drivers and verifies that they point to code within the driver. It also verifies IATs and EATs within processes to check for hooking. However, VICE has a high false positive rate as several Windows applications use hooking for genuine purposes.

Patchfinder uses a method called runtime execution path profiling [24]. It is built upon the observation that a rootkit must add code to a given execution path to perform additional tasks, such as filtering. It uses the single step feature of the x86 processor to count the number of instructions executed. This technique has several limitations. First, the processor has to execute in the single step mode, in which it halts execution after each instruction and calls an interrupt service routine where the instruction count is updated. This is a huge performance overhead. Secondly, on complex operating systems such as Windows, this method leads to non-deterministic behavior because of interleaving of kernel control paths, consequently leading to false positives.

1.6.5 Generic intrusion detection systems

Some generic intrusion detection systems have the capability of detecting certain kind of rootkits. The St. Jude IDS is a kernel level intrusion detection and response system that can detect improper privilege transitions occurring due to vulnerability exploitation in the kernel [25]. Rootkits often exploit such vulnerabilities to get root-level access on a system and therefore, are successfully detected by St. Jude. St. Jude is based on a rule based anomaly detector that uses normal system

Year	Attack Techniques	Defense Techniques
Pre-2003		Tripwire [6], Secure coprocessor [8]
2003		VMI [10]
2004		Copilot [9]
2005	Shadow Walker [27]	Ghostbuster [17]
2006	Non-control data attacks [15], Subvirt [4]	Specification based detection [15]
2007	Stealth Attacks [14]	SBCFI [12]
2008	Cloaker [3]	Paladin [28], Lares [11], Gibraltar [29]

Figure 1.2: Timeline showing evolution of rootkit research

behavior in the training phase. The Linux Intrusion Detection System (LIDS) provides a general intrusion detection framework equipped with mandatory access control, file protection, process protection and port scan detection [26].

1.7 Rootkits Research

Most research in rootkit attack and defense techniques has been carried out over the past five years. Figure 1.2 shows the timeline of work done in this area. Researchers have explored different attack techniques, including proposing new types of advanced attacks that manipulate kernel data and those that install themselves below the operating system. These attacks have motivated the development of new tools and techniques to defend against futuristic rootkits. Research has also explored different architectures for secure placement of the rootkit detector. This Section provides a summary of the contributions of the latest rootkit attacks and defense architectures, tools and techniques.

1.7.1 Attacks

Researchers have explored novel attack techniques by which the operating system can be subverted [14,15,27,30]. These techniques include subverting kernel data to launch novel attacks and creating rootkits that reside below the operating system either independently [3] or by installing a virtual machine monitor [4,5].

This dissertation demonstrates a new class of stealth attacks that do not make an explicit attempt

to conceal their presence. The attacks operate solely by altering kernel data to achieve stealth and carry out their malicious goals, such as disabling the firewall or weakening the pseudo random number generator. These attacks are described in detail in Chapter 3.

Sparks *et al.* demonstrated a rootkit called Shadow Walker that subverts the memory management subsystem by exploiting hardware features [27]. Shadow Walker alters the application's view of a memory page, thereby, subverting rootkit detectors that scan physical memory pages to detect the presence of rootkits. The subversion is achieved by installing a new page fault handler. The page fault handler exploits the existence of two different TLBs in the Pentium architecture for instructions and data. To begin with, the pages belonging to the rootkit are marked as "not present" in the page table entries and any corresponding TLB entries are flushed. The next time a request is made for the rootkit page, it is trapped by the page fault handler, which populates the TLBs. If the request is for code execution, it passes this to the original page fault handler and the rootkit code gets executed. The corresponding physical address is stored in the instruction TLB. If a detector tries to read this page, it is redirected to a fake page by populating the data TLB with a different physical address, consequently hiding the rootkit code.

King *et al.* demonstrated a virtual machine based rootkit (VMBR) that installs itself below the operating system [4]. The VMBR changes the boot sequence on the compromised system, providing the virtual machine monitor full control over the physical hardware after the system reboots. The original operating system is hoisted inside a virtual machine. Since the VMBR runs below the operating system, it can use virtual machine introspection to spy on the guest operating system for sensitive data or interested events such as system calls. The VMBR is also capable of creating a new virtual machine and hosting malicious services, effectively using the compromised system for malicious activities, such as botnets and spam relays.

Rutkowska *et al.* demonstrated a VMBR called Blue Pill [5] that is much stealthier than Subvirt,

by using the AMD SVM technology [31]. Blue Pill does not need to modify the BIOS, the boot sector or system files. It comprises of a thin layer of hypervisor that installs itself by overwriting device driver functions within the Windows swap file, and then requesting the kernel to execute the function. The function then installs blue pill, which takes control of the operating system.

David *et al.* demonstrated a rootkit called Cloaker that installs itself beneath the operating system and hides itself solely by exploiting hardware features [3]. Cloaker does not alter any operating system state or rely on its services. It exploits two main features provided by the ARM processors deployed on mobile phones. The first feature it uses is the ability to establish a trampoline interrupt vector by flipping a single bit in the processor's control register. This allows Cloaker to intercept all interrupts and perform the necessary filtering before it invokes the original interrupt handler. The second feature is the ability to lock down TLB entries, preventing them from being flushed. These locked down entries correspond to Cloaker payload hidden in unused memory pages. Cloaker also provides malicious services, such as keylogging, information theft and distributed denial of service attacks. All of these services are implemented by Cloaker using low level device driver code, which is built as part of the rootkit.

1.7.2 Architectures

Since kernel level rootkits render the kernel untrustworthy, runtime detection tools must execute on an entity that is outside the control of the kernel. Researchers have proposed both virtual machine-based (*e.g.*, [10, 32]) and hardware co-processor based infrastructures [9]), which allow rootkit detection tools to securely observe the runtime execution state of the kernel.

Zhang *et al.* first proposed the use of a secure coprocessor to verify the integrity of kernel code and data [8]. The idea was to isolate the detector from the operating system that it monitors. The coprocessor boots and executes independently and serves as an external entity that can verify the operating system state and at the same time, secures itself from the rootkit. Their prototype was a

proof of concept that was implemented using a kernel module.

Petroni *et al.* built a detection system on a real secure coprocessor [9]. Their prototype, Copilot, works on a secure coprocessor plugged into the PCI bus. It does not rely on the operating system for any services. It can access the system main memory via PCI DMA. It can monitor the integrity of kernel code and some well known immutable data structures in the kernel, such as the system call table. It does this by periodically fetching these well known data structures at fixed locations from main memory and comparing a cryptographic hash of the memory area against pre-computed values, previously stored from a clean kernel. Copilot was tested against real-world rootkits and was found to be effective in detection of almost all of them that modified either kernel code or well-known immutable data structures in the kernel. It could detect the attack within a matter of a few seconds.

Garfinkel *et al.* proposed virtual machine introspection (VMI) architecture for rootkit detection [10]. This architecture isolates the system being monitored from the detector by running them in different virtual machines. The detector runs in a privileged VM, and is therefore able to inspect the target's physical memory and the events, such as system calls. The VMM intercepts interesting events and has the capability to forward it to the detector. The detector in this architecture has an advantage that it is able to provide a suitable response to events over the coprocessor based architecture.

Payne *et al.* proposed a virtual machine based architecture for active monitoring of the operating system inside the guest VM in a secure fashion [11]. Lares installs event hooking and forwarding components within the untrusted guest operating system. The event handling component installs the hooks within the guest OS that is responsible for trapping the event. The hook forwards the event to the trampoline code also running within the guest OS. The trampoline forwards the event along with the context information to the security application running in the monitoring VM. Lares

protects the integrity of the components within the untrusted VM by implementing a write protector within the hypervisor. Any write performed to these components is trapped by the hypervisor and disallowed. The security application is responsible for handling the event and implementing the policy. The response is delegated to the trampoline code through the hypervisor. Lares provides a systematic framework for introspection applications to securely monitor and interpose on events of interest within the guest OS.

Petroni *et al.* proposed a specification-based architecture for detection of rootkits that modify dynamic data in the kernel [15]. In this architecture, data structures in kernel memory are periodically checked against integrity specifications. These specifications describe key semantic properties of data structures, which must hold during normal execution of the kernel; violation of any of these specifications indicates the presence of a rootkit. The architecture allows the specification writer to develop these data structure integrity specifications in a higher level language. They developed a specification compiler that translates these specifications to a low level code. While this technique has the advantage of being able to detect rootkits that modify both static and dynamic data, it requires the integrity specifications to be developed manually.

1.7.3 Tools and techniques

Microsoft Strider Ghostbuster proposed a technique to detect rootkits that exhibited hiding behavior [17]. Their technique is based on a cross view based approach. The idea is to compare multiple views obtained in different ways and check the resulting answers for "lies". Any discrepancy between the answer indicates the presence of a rootkit that tries to hide its objects. It uses an "inside the box" scan that compares results from querying the OS enumeration APIs and a low level query of the OS data structures. These reveal rootkits that are hiding between user and kernel space. An "outside the box" scan compares the low level kernel scan of the system with that of a clean system, typically obtained by rebooting the same system from a clean media. This scan detects

rootkits that alter the kernel state to hide. The efficacy of this technique largely depends on how it is implemented.

Petroni *et al.* proposed state based control flow integrity [12], a technique to automatically verify all dynamic control data in the kernel. The technique traverses all objects in the kernel heap beginning from a set of static roots i.e. pointers to objects in the kernel static area. Traversal is guided by the use of type definitions that is extracted statically from the kernel source code. Upon encountering function pointers, it checks the integrity of the function pointer based on a certain policy. A simple policy that they implemented is to ensure that each function pointer points to an exported function within the kernel code. The policy can be made more accurate by substituting it with a result from a points-to analysis of the source code.

An alternative to the above runtime techniques are tools that pro actively scan kernel modules and device drivers to determine whether they are malicious. These include both signature-matching techniques as employed by most commercial malware detection tools, and symbolic execution tools [33, 34], which statically approximate the behavior of a kernel module to determine whether it likely affects key kernel data structures. Several attestation based approaches have been proposed to verify the integrity of the running code [35–41]. These approaches use a secure chip as the trusted computing base to bootstrap trust. The verification procedure is based on comparing hashes with known good values or timing calculations. These approaches work well for checking the integrity of code, but they are ill-suited to check the integrity of data.

Grizzard *et al.* [42] address the issue of recovering from rootkits that modify the system call table by replacing the infected copy with a clean copy. This is an offline recovery procedure, which works only for rootkits that modify the system call table.

1.8 Monitoring Kernel Data Integrity

The increase in number and complexity of kernel rootkits can be attributed to the large and complex attack surface that the kernel presents. The kernel manages several hundred heterogeneous data structures, most of which are critical to its correct operation. A rootkit can subvert the kernel integrity by subtly modifying any of these data structures.

Kernel data structures that hold control data, such as the system call table and other jump tables, have long been a popular target for attack by rootkits. However, recent rootkits achieve a variety of malicious goals by modifying non-control data in the kernel. For example, the *fu* rootkit hides a malicious user space process by manipulating linked lists used by the kernel for bookkeeping. Similarly, in Chapter 3, we demonstrate rootkits that degrade application performance by modifying memory management meta data and those that affect the output of the pseudo random number generator by contaminating entropy pools.

1.8.1 Current research

Current research has proposed techniques to monitor the integrity of kernel data. These approaches are summarized and compared in Table 1.3.

Copilot [9] can monitor the integrity of kernel static immutable data structures in the kernel. These data structures have to be manually specified and are often derived from known rootkit behaviors. It has no provision to detect the integrity of dynamic data in the kernel.

Specification based detection technique [15] can monitor the integrity of data on the kernel heap but requires elaborate specifications of kernel data structure integrity. These specifications are supplied by an expert who has a detailed understanding of kernel data structure semantics. Kernel data structures are continuously monitored during runtime against these specifications, and violations are used as indicators of rootkit behavior. While this approach has the advantage of

	Location of data		Type of data		Specifications
	Static	Dynamic	Control	Non-control	Automatic
Copilot [9]	✓	x	✓	✓	x
Specification based detection [15]	✓	✓	✓	✓	x
SBCFI [12]	✓	✓	✓	x	✓
Gibraltar [29]	✓	✓	✓	✓	✓

Figure 1.3: Comparison of kernel data integrity monitors

detecting sophisticated rootkits, developing specifications is currently a manual procedure. Because the kernel maintains several hundred data structures, the specification writer could either fail to supply certain integrity specifications, *e.g.*, because he is unaware that they exist, or may fail to realize how a rootkit could exploit them.

SBCFI is an automated integrity checking technique that verifies the integrity of all control data in the kernel. A simple policy that SBCFI uses is to check if each function pointer encountered in the kernel points to a valid kernel function address. One shortcoming of this technique is that it can only verify the integrity of control data in the kernel.

In this dissertation, we propose Gibraltar, discussed in detail in Chapter 4. Gibraltar addresses the shortcomings of the above techniques by using a learning based approach. It can automatically generate integrity specifications for all control and non-control data in the kernel based on the invariants that it observes during a training period. However, Gibraltar does suffer from some limitations. It can only detect rootkits that violate invariants on data structures. It is possible that a rootkit might manipulate a data structure such that it does not violate an invariant or manipulate data structures in ways that violate invariants that Gibraltar does not mine for.

1.8.2 The future landscape

Current rootkits make persistent changes to kernel data structures. Persistent modifications give rootkits long-term control over the system. However, persistent changes to kernel memory are also

likely to be detected by systems, such as SBFi [12] and Gibraltar [29]. Thus, with maturing detection techniques, rootkits will increasingly begin to employ transient techniques to achieve their malicious goals. Existing infrastructures, such as the PCI-hardware based infrastructure and the VMI-based architecture described previously, do not suffice to detect such rootkits because they operate by capturing and observing data structures in kernel memory snapshots. Capturing and analyzing memory snapshots is time-consuming and does not operate in real-time. Consequently, these detection techniques may miss observing the state of the kernel when the rootkit has compromised the integrity of a data structure. Alternately, a rootkit could time its modifications so as to evade these detectors.

1.9 Contributions of this Dissertation

This dissertation explores the threat posed to computer systems from advanced rootkit attacks from both perspectives; offense and defense. It makes the following contributions:

- *A virtualization based architecture for automatic containment of conventional rootkit attacks.*
The goal of this work is to minimize the damage caused to the system by executing a containment procedure upon detection of a rootkit. This work also successfully contains other malware, such as worms and viruses that use rootkits to hide. We designed and implemented a prototype Paladin, using the VMware Workstation software. Paladin could effectively contain rootkit attacks and rootkit carrying malware that we used in our experiments. This work is published in the Elsevier Computers and Security Journal Nov, 2008 [28].
- *Identification of a novel class of stealth attacks that manipulate the compromised system by solely manipulating data structures in the kernel, without making an explicit attempt to hide.*
None of these attacks were detectable using the state of the art detection tools. The goal of this work was to show that the threat to kernel data is systemic, requiring a comprehensive solution

that accounts for all data structures in the kernel. It also highlights the shortcomings of the state of the art tools used for rootkit detection. This work is published in the Proceedings of the IEEE Symposium of Security and Privacy (IEEE S&P 2007) [14].

- *A novel rootkit detection technique that detects advanced data-centric stealth attacks on the kernel.* It is based on the hypothesis that data structures exhibit invariants and attacks violate some of the invariants exhibited by them. Our technique automatically and uniformly infers invariants on control as well as non-control data structures in the kernel. These invariants are used as specifications of data integrity. At run-time, attacks that violate these invariants are automatically detected. This work is published in the Proceedings of the Annual Computer Security Applications Conference (ACSAC 2008, to appear) [29].

1.10 Contributors to this Dissertation

Following are two of my colleagues who co-authored papers from which I used material in this dissertation, along with their contributions: Xiaoxin Chen helped me in building the Paladin prototype during my internship at VMware Inc. Pandurang Kamat helped in developing some of the attack ideas in this dissertation. Tzvika Chumash wrote parts of the code for the Paladin driver.

1.11 Organization of this Dissertation

This dissertation is organized as follows. Chapter 2 describes our containment technique designed using the virtualization architecture, to contain conventional rootkit attacks. Chapter 3 describes the novel class of stealth attacks against kernel data that we identified. In Chapter 4, we describe our automated rootkit detection technique based on invariant inference over kernel data structures. Finally, Chapter 5 concludes this dissertation with some directions for future work.

Chapter 2

Attack Containment

Conventionally, kernel rootkits provide most of the malicious functionality as user space programs. The actual stealth is achieved by altering the code or data in the kernel, particularly control data that provides the rootkit with a vantage point of intercepting and influencing application level views. The malicious code injected into the kernel is designed to hide user space objects, such as files, processes and network connections, belonging to the attacker. The most popular data structure altered for this purpose is the system call table. Other function pointers in the virtual file system layer also have been targets of kernel rootkits.

Kernel integrity monitors, which check the integrity of well-known immutable data structures in the kernel, are able to detect the presence of the rootkit as soon as the data structure is corrupted. However, sheer detection does not suffice to stop the malicious activity in progress, which might involve irreversible tasks, such as exfiltration of sensitive information or system involvement in fraudulent or criminal activities. An effective containment technique must stop or minimize the ongoing damage to the system.

In this chapter, we propose a technique to contain rootkits by leveraging the virtual machine technology. We use virtual machine introspection [10] for rootkit detection, and augment it with a containment algorithm, which is executed upon detection of the rootkit. The key idea behind our

technique is to maintain process relationships as the processes are launched on the system, in the form of a dependency tree. When a rootkit is detected and the offending process is known, the algorithm refers to this dependency tree and identifies other possible malicious processes and kills them. This approach also efficiently contains other malware, such as viruses, worms and spyware that use rootkits to hide.

We built a proof of concept prototype, *Paladin*, implemented as part of VMware Workstation. It detects and contains rootkits that use either user mode stealth (e.g. by installing trojaned binaries) or kernel mode stealth (e.g. by corrupting the system call table) to hide objects. Using this prototype, we show that our approach is effective in containing a large percentage of Linux rootkits found in the real world. *Paladin* also effectively contains the stealthy Lion worm (bundled with a rootkit), which we used in our experiments.

2.1 Virtual Machine Technology

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

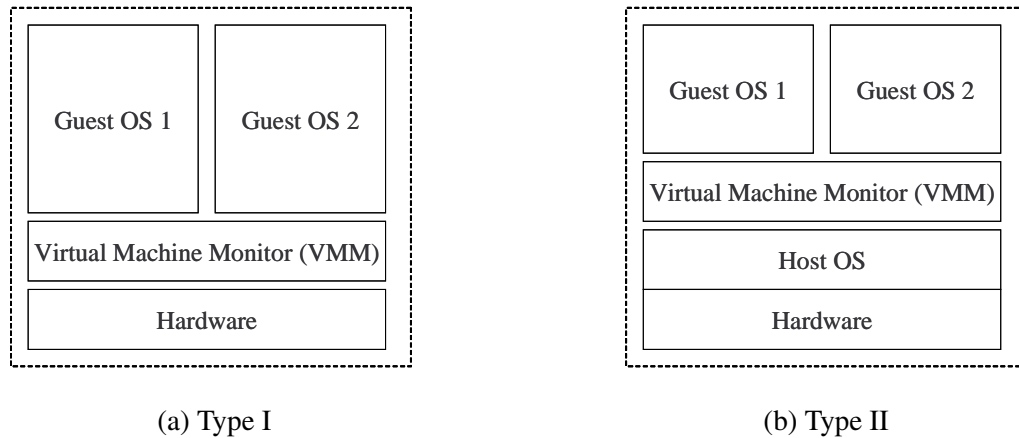


Figure 2.1: Types of virtual machines

[31,43].

Virtual machines can be classified into two categories: Type I and Type II VMs. As shown in Figure 2.1, Type I virtualization software, such as VMware ESX server [44] and Xen [45], run on the bare machine and control the physical resources. It provides a virtual interface to operating systems running inside the VMs. Figure 2.1 also shows Type II virtualization software, such as the VMware Workstation, which uses the hosted architecture [46]. In this case, VMware Workstation is first installed as an application on an existing OS, called the Host OS. The operating system running inside the virtual machine is known as the Guest OS. VMware runs as a process on the Host OS and relies on it to fulfill Guest OS I/O requests. Since the Guest OS runs at a less privileged level on the physical processor, compared to the VMM, the virtual machine monitor has the capability of intercepting events from the Guest OS. While our prototype is implemented with a Type II virtual machine, it can just easily be implemented using a Type I virtual machine.

2.2 Security Model

Having the Intrusion Detection System (IDS) on the Host OS to monitor the Guest OS is known as virtual machine introspection [10]. From the time this model was first proposed, it has been widely accepted as a secure model for monitoring and intercepting events from the Guest OS. It is

extremely difficult for the attacker to compromise the IDS despite having complete control of the Guest OS, as the Guest OS runs at a lower privilege level compared to the Host OS on the physical processor.

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

- **Encapsulation.** The VMM presents a virtual hardware interface to the OS inside the virtual machine. It is nearly impossible for an attacker in the Guest OS to inject instruction stream into the virtualization layer or access resources outside of the emulated virtual hardware.
- **Introspection.** The VMM can inspect the virtual machine's state at instruction level without possible detection by the code running inside the VM. Any host-based IDS suffers from the problem of sharing resources with the OS and can be compromised. On the other hand, network based IDS has to rely on network stream and is required to reassemble fragments of evidence to detect possible intrusion. Such approaches are typically less accurate and have limited visibility of the host operating system.
- **Tamper proof.** The VMM runs at a higher privilege level on the physical processor compared to the Guest OS. This makes the VMM code inaccessible from the Guest OS except through well-defined interfaces.

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

2.3 Approach

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

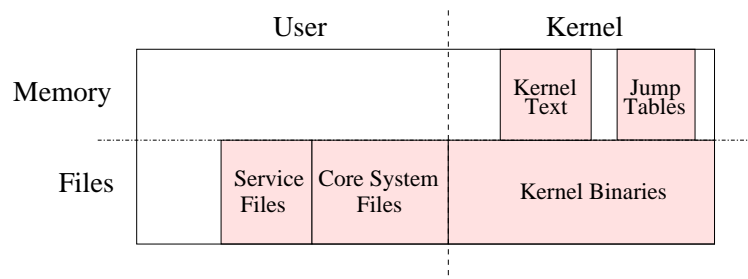


Figure 2.2: Protected zones in the file system and system memory

2.3.1 Prevention and Detection

This mechanism relies on specification of access control policies for rootkit detection. The policies are tailored to protect memory areas and system files that are a target of rootkit attacks. These are categorized into file access control and memory access control policies as depicted in Figure 2.2. File access control policies protect the system utilities from being replaced by their trojaned counterparts. Memory access control policies protect the kernel code and data structures from being overwritten in memory, which is a common method utilized by kernel rootkits.

Figure 2.3 shows a sample policy file used by our system. Several directories are made non-writable. These policies can be tailored as per the system requirement. For example, Figure 2.3 shows that the program `/usr/bin/passwd` is allowed to write into `/etc/passwd`, which is readonly to other programs. Such policies are easy to specify and are commonly used with other tools, such as Tripwire [6] and AIDE [7]. Memory access control policies for rootkit detection currently include protecting the kernel system call table, the interrupt descriptor table and the kernel text. These are the three most common hooking places in memory for rootkits. As rootkit authors find newer data structures to manipulate, the access control policies can be extended to protect the newer data regions. Certain legitimate applications may need to write into kernel memory or hook to certain protected areas. For example, an anti-virus program may need to place a hook into the system

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

Figure 2.3: Sample Paladin policies

call table to scan for files before they are opened by the user. The access control policies can be tailored to allow only these applications exclusive access to the required protected zones. These applications, in turn, need to be protected on disk by using appropriate access control policies to prevent rootkits from modifying them.

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

2.3.2 Tracking

The tracking mechanism generates a dependency tree by maintaining parent-child relationships between processes and relationships between processes and generated files. This information is updated from the system call events. The dependency tree is used by the containment algorithm to identify possible malicious processes. Figure 2.8 shows a sample representation of a dependency tree. Processes are represented by ellipses and files by rectangles. In a *process* \rightarrow *process* relationship, a directed edge from one process to another represents a parent-child relationship. A *process* \rightarrow *file* dependency, shows that the file is created by the process.

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

1. Upon process creation, a link is created between the parent and the newly created child process. This is represented as a directed edge from the parent to the child in the dependency tree.
2. When a process image is overlayed for execution, we store the filename from where the process was executed. This filename, shown within the ellipse, represents the process name in the dependency tree.
3. When a process exits, if it has created other files or child processes, the process is not deleted from the dependency tree but simply marked for deletion. If the process has not spawned any child processes nor created any files, the process is deleted.
4. When a file is created, a link is created from the process to the file.
5. When a file is deleted, any process that becomes childless and has been previously marked for deletion is also deleted.

2.3.3 Containment

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

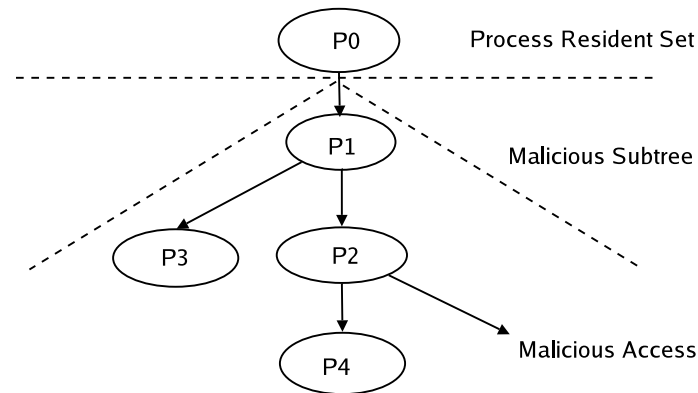


Figure 2.4: Automated containment in Paladin

formatting the hard disk, an attacker from stealing confidential data, or a worm from spreading.

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

Figure 2.4 shows process P2 performing malicious access. P2 is identified as the offending process. The system automatically identifies the subtree that is considered malicious using the information in the dependency tree. A path is traced back from the malicious process P2 to the root of the dependency tree until a process in the PRS is encountered, all of whose ancestors are also in the PRS. The previously visited node becomes the root of the subtree. In this case, P0 belongs to the PRS and hence, P1 becomes the root of the subtree identified as malicious. All processes identified in this subtree are considered malicious and will be killed by our system. The pseudo code for the containment algorithm is shown in Figure 2.5.

```

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

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

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

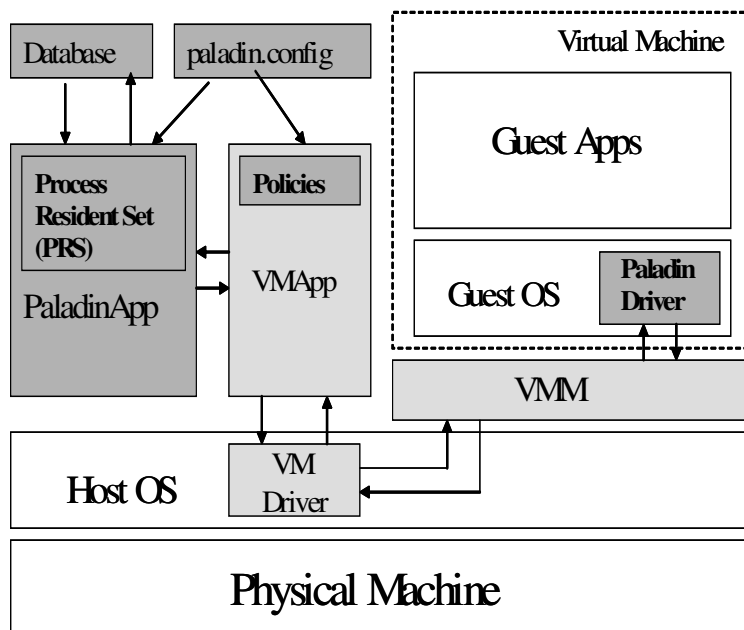
```

Figure 2.5: The containment algorithm

2.4 Design and Implementation

In this section, we describe the design of the prototype Paladin. As shown in Figure 2.6, Paladin comprises of several components: the modified form of VMware Workstation, PaladinApp, the driver and the database. VMAApp and the VMM are a part of the VMware Workstation software. We added hooks into these to enable communication with PaladinApp, which is an application process in the host OS. Arrows indicate the communication paths between the different components of the system. The dashed box in the figure represents a virtual machine.

VMware workstation is a type II virtual machine [46], which is installed on a host operating



Components shown in dark gray are the ones added by us. Components shown in light gray belong to the VMware Workstation software.

Figure 2.6: The Paladin architecture

system and relies on the host to fulfil I/O requests from the Guest OS. The VMApp appears as a process on the host OS. VMApp runs the VMM, which in turn runs the guest operating systems.

2.4.1 Design overview

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

In our prototype, VMM forwards file and process related system calls to PaladinApp. These are used by PaladinApp to update the dependency tree, stored in the database. If a violation of a given access control policy is intercepted by the VMM, it notifies the PaladinApp, which in turn

initiates the containment procedure. The Paladin prototype works in the following three phases: Initialization, Normal Operation and Containment.

Initialization

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

Normal operation

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

Containment

This phase is initiated when a specified access control policy is violated. In both cases, the process performing the malicious access is identified by the VMM and the information is passed to the PaladinApp. In case the illegal access is performed from a kernel module, the process inserting the module is considered malicious. The PaladinApp refers to the dependency tree and runs the containment algorithm. It relies on the driver loaded in the guest OS to kill malicious processes. The driver code itself is protected by the VMM and cannot be tampered with by the attacker.

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

2.4.2 Implementation

Our prototype was developed for VMware Workstation. The host machine and the virtual machine were running the 2.4 Linux kernel. The database used was MySQL. The driver is a Linux kernel module (LKM). The driver looks up the *System.Map* file. It finds the symbols for kernel text segment, system call table and interrupt descriptor table (IDT) and returns the physical addresses of these symbols².

System call information consists of the system call number, arguments and the virtual CPU registers. When a fork is encountered, in a process, a relationship is created between the parent and the child process. The child process has a different page table root (*cr3*) than the parent, but the same stack and instruction pointer upon exit from the fork system call. Thus, a fork system call return with the same stack pointer and instruction pointer but a different *cr3* indicates a return to a child process. Moreover, when comparing stack pointer values, we use the physical address converted from the virtual address to avoid ambiguity due to two identical processes making fork system calls at the same point in the program. This assumes that the OS implementation of process creation uses copy-on-write for the user stack pages.

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

2.5 Evaluation

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

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

2.5.1 Linux rootkits

We analyzed 36 significant rootkits available for Linux. Figure 2.7 lists these rootkits categorized according to the hiding mechanisms used. Rootkits classified as Category 1 use trojaned system binaries to hide from the users. These rootkits are easily portable and can be quickly installed. Category 2-4 rootkits change the kernel to hide themselves and are highly sophisticated compared to their user-level counterparts. Many of the rootkits use the Linux kernel loadable module interface to load their code in the kernel. More sophisticated rootkits write directly in the kernel memory using the `/dev/kmem` and `/dev/mem` interfaces and are effective even when the module support is disabled. Category 2 rootkits hook to the system call table, still a widely used technique. Category 3 rootkits change the kernel text and Category 4 rootkits hook to the interrupt descriptor table (IDT). Often, kernel rootkits use user-space programs to perform the actual malicious job of installing backdoors, sniffers and keyloggers. Rootkits bundled with other malware, such as viruses, worms and spyware, have almost no limits on the damage that they can stealthily do to the system.

2.5.2 Experimental methodology

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

As shown in Figure 2.7, we could successfully test 27 rootkits (indicated by a check mark in column *Test Set*) against Paladin, and Paladin detected and contained all of them. We were unable to get a functional version for the nine others and hence, they were excluded from our test set. An examination of their source code revealed that they use similar techniques as the others within the same category. Therefore, we contend that Paladin will be able to counter these rootkits as well. We use simple policies shown in Figure 2.3 for our experiments.

Category 1		Category 2		Category 3	
Rootkit	Test Set	Rootkit	Test Set	Rootkit	Test Set
0x333openssh-3.7.1	-	adore-0.42	✓	enyelkm v1.1	-
ark 1.0.1	✓	all-root	✓	phantasmagoria	✓
balaur 2.0	✓	linspy2	✓	suckit	✓
cbr00tkit	✓	kbd v3	✓	suckit2priv	✓
devNull v0.9	-	kis 0.9	✓	superkit	✓
dica	✓	knark 2.4.3	✓		
fk v0.4	✓	modhide	✓		
flea	✓	maxty	-		
lrk5 and variants	-	override	-		
sm4ck	✓	phalanx-b6	-		
tl0gin	✓	phide	✓		
tnet-tools v1.55	-	rial	✓		
torn 6.66	✓	rkit 1.01	✓		
trNkit v1.0	✓	synapsys	✓		
troier v 1.0	✓	taskigt	✓		

Category 4	
Rootkit	Test Set
backdoor-caca	-

✓	Included in test set
-	Not included in test set

category 1: User-level - Install trojaned system binaries

category 2: Kernel-level - Modify the system call table

category 3: Kernel-level - Modify kernel text

category 4: Kernel-level - Modify interrupt descriptor table (IDT)

Figure 2.7: Linux rootkits publicly available, categorized by hiding techniques

We pick one sample rootkit from each category to describe the effects of our mechanism in detail. Additionally, to demonstrate the strength of the containment mechanism in case of fast spreading automated attacks, we evaluate it with a worm called *Lion* that carried a rootkit [47].

2.5.3 Experimental results

Category 1

The *Tornkit* rootkit trojans the following system binaries : *du*, *find*, *ifconfig*, *in.fingerd*, *login*, *ls*, *netstat*, *pg*, *ps*, *pstree*, *sz* and *top*. It also installs a log cleaner (*t0rnsc*), a standard linux sniffer (*torns*) and a sniffer log parser (*t0rnp*). The kit creates a hidden directory called */usr/src/puta*, where it stores all the hidden information.

PD mode. In this mode, the system binaries mentioned above are prevented from being overwritten

since they violate the specified access control policies. However, this mode cannot prevent running of the sniffer and the log eraser processes.

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

Category 2

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

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

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

Category 3

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

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

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

Category 4

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

Stealthy worm

Lion is a notorious stealth worm [47] that spread very quickly and evaded detection for a long time due to the presence of a rootkit. It installs the *t0rn* rootkit to hide its files. Lion worm has several variants including some without the rootkit. Since we could not find the variant of the Lion worm with the rootkit, we created a home-grown version of this worm. We combined the variant available on the Internet that did not contain the rootkit and modified it to include the rootkit.

Lion affects DNS servers that have the BIND TSIG vulnerability [48]. Figure 2.8 shows the Lion worm in action. Once the worm has gained access to a machine, it scans for vulnerable hosts with class B Internet addresses. The program *pscan* is used to scan the network while *randb* generates random class B network addresses. If the worm finds a machine with a given IP address, it checks if the machine is susceptible to the BIND attack. If the target machine has a vulnerable version of BIND running, it uses the vulnerability to get root privileges on the system and continue propagating. It installs the *t0rn* rootkit to hide the compromise. The files *scan.sh* and *hack.sh* execute the worm algorithm. *getip.sh* tries to find more victims while *li0n.sh* and *star.sh* are the controlling processes [49]. The worm finds all the files named *index.html* present on the system and defaces the pages by replacing it with its own version.

PD mode. In this mode, only the system binaries are prevented from being corrupted. Malicious access is detected when the rootkit is activated and tries to overwrite the system binary */bin/ps*. Since

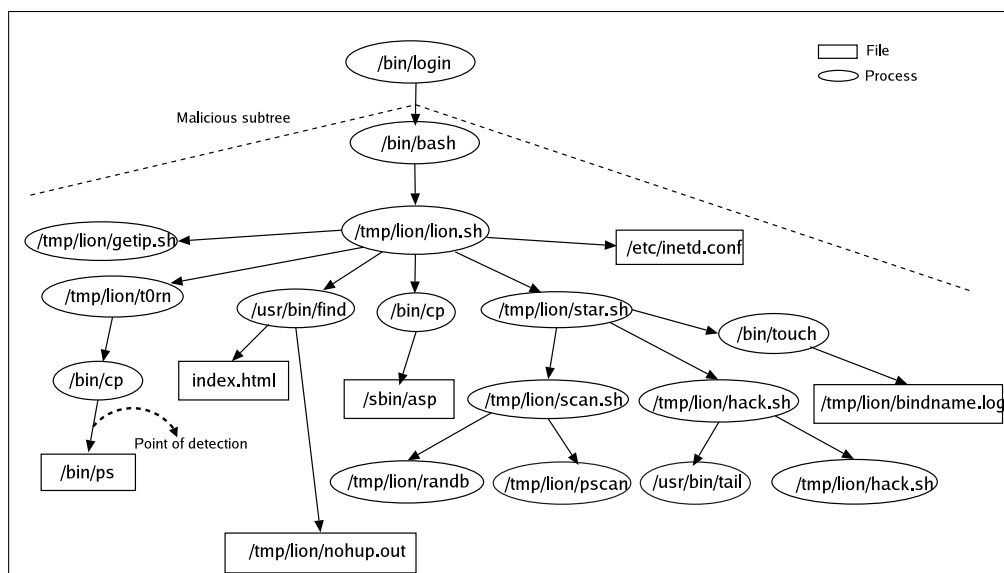


Figure 2.8: Dependency tree shows the Lion worm attack

the prevention part stops the corruption of the binaries, the hiding behavior is effectively disabled. The worm activities are visible to the administrator. However, these activities continue to occur. Lion defaces all the *index.html* files found on the machine. It also tries to propagate to other hosts running the vulnerable version of the BIND service.

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

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

System call	Paladin	
	Disabled	Enabled
fork	1.5 μ s	3.5 μ s
exit	1.5 μ s	1.6 μ s
open	0.8 μ s	1.5 μ s
close	0.5 μ s	0.7 μ s

Table 2.1: Per system call performance

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

Table 2.2: Performance overhead

2.5.4 Performance

To test the overhead of Paladin, we performed two system call intensive tasks. First, we copied a large set of files from one directory to another. The second task was to compile the Linux kernel. We measured the time taken for both these tasks with and without Paladin support. Table 2.2 shows the performance overhead incurred by applications that run with and without Paladin prototype. Paladin adds about 12% overhead to the execution time for applications in the Guest OS.

Table 2.1 shows the per system call overhead for the four most common file and process related system calls. Paladin incurs relatively larger overhead for open and fork system calls. This is due to the fact that the application performs bookkeeping and matching of parent/child system calls respectively.

2.5.5 Dependency tree size

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

the database is small and storage requirements are modest.

2.6 Discussion

In this section, we discuss some related issues, counter attacks on Paladin and the limitations of our solution.

2.6.1 Handling Loadable Modules

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

2.6.2 Counter attacks

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

Multiple control processes

An immediate attack that comes to mind is the use of multiple control processes to carry out the attack. Here, the controlling process for the hiding part is separated from the controlling process that performs other malware activities (non-hiding). This is shown in Figure 2.9. In this figure, P1 is the controlling process that performs the hiding. P1 may spawn other processes or write into the kernel directly. P3 is the controlling process for carrying out other activities, such as installing keyloggers, scanning network packets, sending passwords etc. Here, since P1 and P3 share a parent P0 (this process could be *sshd* for example), which resides in the PRS, our containment mechanism

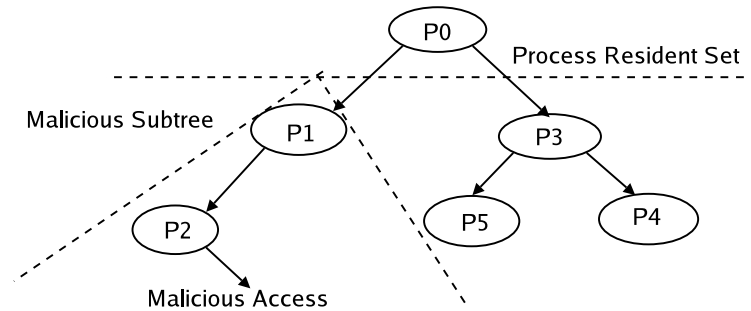


Figure 2.9: Multiple control processes

cannot automatically link the process P3 and its children to P1. The hiding processes, P1 and its children will be killed, while P3 and its children will continue to run.

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

Overwriting disk blocks

The attacker can directly attempt to change system binaries by overwriting disk blocks. Our approach tracks processes using the system call interface to access files. While this is generally true for most processes, it is possible for an attacker to perform a write directly to disk blocks. We can handle this by disabling raw writes to disks by augmenting the access control policies. A more effective solution is to use other approaches, such as storing data in a separate data VM [50] to force file access through a well-defined interface.

In-memory corruption of resident processes

Rootkits may be able to backdoor resident processes by overwriting code in memory. This attack is very hard to carry out on Linux due to the absence of APIs to perform such actions. We can however, defend against this attack by simply protecting the code pages of these processes from the

VMM.

2.6.3 Limitations

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

Killing legitimate processes

Our containment algorithm kills all processes inside the malicious subtree, which might involve other genuine processes run by the user. We contend that this is a better option, rather than allowing the malware to continue running, which can cause the owner financial or legal distress.

Accidental modifications

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

System upgrades

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

Other types of stealthy behavior

Other types of stealthy behavior may include modifying the user's environment variables, so that the user executes corrupted binaries without his knowledge. Though this is stealthy behavior, it does

not strictly fall under the umbrella of the hiding characteristics unique to rootkits. In this case, the corrupted binaries are visible and can be detected by the user by installing other tools like Tripwire and AIDE.

2.7 Summary

Attack containment is critical in halting the ongoing damage in progress. In this chapter, we discussed our containment algorithm, which upon detection of a rootkit, identifies a set of processes as malicious and kills them, by referring to a process dependency tree. This algorithm is effective in containing rootkits that provide most of the malicious functionality as user space programs. We discussed the design and implementation of a prototype, Paladin, built as part of VMware Workstation. Paladin uses virtual machine introspection for rootkit detection and runs the containment algorithm when a rootkit is found. Paladin was found to be effective in containing all publicly available Linux rootkits at the time. It also effectively countered the Lion worm bundled with a rootkit, that we used in our experiments. In the next chapter, we discuss a new class of stealth attacks that are different from the conventional rootkits that we discussed here.

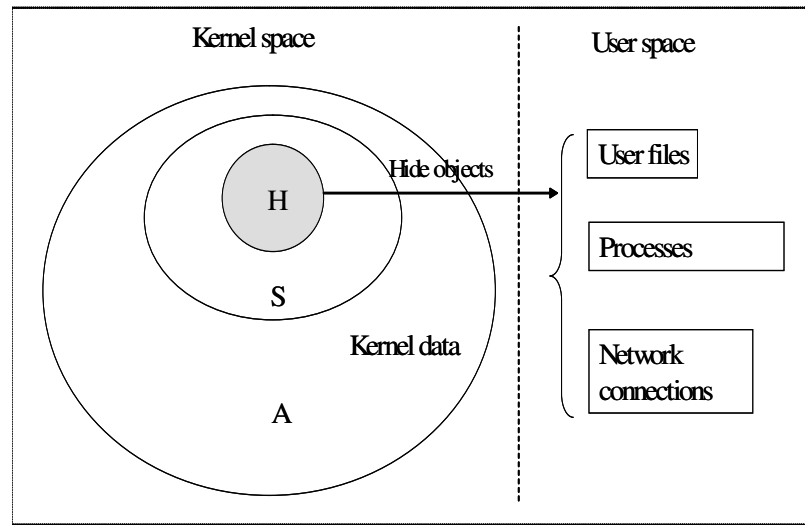
Chapter 3

Stealth Attacks on Kernel Data

Conventional kernel rootkits provide most of the malicious functionality as user space programs. Therefore, they are designed to hide user level objects such as files, processes and network connections belonging to the attacker. This hiding behavior is typically achieved by tampering kernel code or data reachable from the system call control paths. For example, when the user looks at a directory listing, several system calls are invoked, which display the contents of the directory. To be able to hide a malicious file that exists in the directory, the attack either places a hook available in the control paths of the corresponding system calls, or modifies non-control data to achieve the same. The most common data structure exploited by rootkits to intercept control from user applications and eventually hide objects is the system call table itself. Attacks have also targeted other data structures in the virtual file system layer and the process tables that exist at different layers in the kernel.

3.1 Problem Statement

Implementing malicious functionality as user space programs provides the rootkit authors with lot of flexibility and ease of writing code. Therefore, they need to manipulate the kernel to hide these user space programs. These attack techniques that primarily focus on hiding objects have two weaknesses. First, detection methods that use a cross view based approach, such as Strider Ghostbuster



Set A represents a set of all kernel data structures. Set S represents only those data structures reachable from the system call control paths ($S \subset A$). Set H represents the data structures within set S that play a role in creating user level views ($H \subset S$).

Figure 3.1: Kernel data structure affecting user level views

[17], utilize the hiding behavior as a symptom for detection. Therefore, these attacks can be easily detected, the moment they try to hide. Second, they are limited to exploiting only a small set of data structures that play a role in creating user level views.

Figure 3.1 depicts this scenario. Set A is a set of all data structures in the kernel. Set S is a subset of set A, which comprises of data structures that are reachable from the system call paths. Set H is a subset of set S, consisting of data structures that specifically allow for altering user level views. Rootkits that attempt to hide malicious objects on the system, alter data structures that are present only in the set H.

In this chapter, we discuss the design of a new class of attacks that are stealthy by design. The attacks achieve their malicious objectives solely by manipulating kernel data. Since they do not depend on user level counterparts to provide malicious functionality, these attacks do not explicitly attempt to hide any object. Therefore, these attacks are not limited to manipulating data only in the

subset H as shown in Figure 3.1. They can affect any data structure belonging to the set A. Further, they cannot be detected by tools that check for hiding behavior as a symptom for detection. These attacks demonstrate innovative ways in which an attacker can alter the behavior of different kernel subsystems in the kernel and are indicative of a more systemic threat posed by future rootkits to kernel data.

3.2 Disable Firewall

This attack hooks into the *netfilter* framework of the Linux kernel and stealthily disables the firewall installed on the system. The user cannot determine this fact by inspecting the system using *iptables*. The rules still appear to be valid and the firewall appears to be in effect. In designing this attack, the goal of the attacker is to disable the network defense mechanisms employed by the target systems, thereby making them vulnerable to other attacks over the network.

3.2.1 Background

Netfilter is a packet filtering framework in the Linux kernel. It provides hooks at different points in the networking stack. This was designed for kernel modules to hook into and provide different functionality such as packet filtering, packet mangling and network address translation. These hooks are provided for each protocol supported by the system. The *netfilter* hooks for the IP protocol are shown in Figure 3.2. Each of the hooks, *Pre-routing*, *Input*, *Forward*, *Output* and *Post-routing*, are hooks at different points in the packets traversal. When the packets come in, after a few sanity

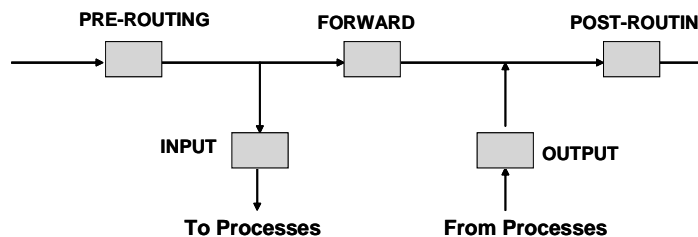


Figure 3.2: Hooks provided by the Linux netfilter framework

```

Chain INPUT (policy ACCEPT)
target prot opt source destination
ACCEPT tcp -- anywhere anywhere tcp dpt:ssh
ACCEPT tcp -- anywhere anywhere tcp dpt:telnet
ACCEPT tcp -- anywhere anywhere tcp dpt:24
REJECT tcp -- anywhere anywhere tcp dpt:http reject-with
icmp-port-unreachable

Chain FORWARD (policy ACCEPT)
target prot opt source destination

Chain OUTPUT (policy ACCEPT)
target prot opt source destination

```

Figure 3.3: Firewall rules deny admission to the web server port

checks, they are passed to the *netfilter* framework's *Pre-routing* hook. Next, they enter the routing code, which decides whether the packet is destined for another interface, or a local process. If the packet belongs to the local system, the *netfilter* framework's *Input* hook is invoked, before being passed to the process. If the packet is intended for another interface instead, the *netfilter* framework's *Forward* hook is invoked. The packet then passes a final *netfilter Post-routing* hook, before being put on the wire again. The *Output* hook is called for packets that are created locally.

Iptables is a firewall management command line tool available on Linux. *Iptables* can be used to set the firewall rules for incoming and outgoing packets. *Iptables* uses the *netfilter* framework to enforce the firewall rules. Packets are filtered according to the rules provided by the firewall.

3.2.2 Attack description

The pointers to the *netfilter* hooks are stored in a global table called *nf_hooks*. This is an array of pointers that point to the handlers registered by kernel modules to handle different protocol hooks. We modified the hook corresponding to the IP protocol and redirected it to our dummy code, effectively disabling the firewall. The firewall rules that we used during this experiment are shown in

Figure 3.3. The INPUT rules deny admission for incoming traffic to the web server running on the system. Before the attack, we were unable to access this web server externally. After we inserted the attack module, we could access the web content hosted by the web server running on http port (port 80). Running *iptables* command to list the firewall rules still shows that the same rules are in effect (as shown in Figure 3.3). The user has no way of knowing that the firewall is disabled as the rules appear to be in effect.

3.2.3 Impact

A stealthy attack such as the one described cannot be detected by the existing set of tools. Since our attack module is able to filter all packets without passing it to the firewall, it can run other commands upon receipt of a specially crafted packet sent by the remote attacker.

3.3 Resource Wastage Attack

This attack causes resource wastage and performance degradation on applications by generating artificial memory pressure. The goal of this attack is to show that it is possible to stealthily influence the kernel algorithms by simply manipulating data values. This attack targets the zone balancing logic, which ensures that there are always enough free pages available in the system memory.

3.3.1 Background

Linux divides the total physical memory installed on a machine into nodes. Each node corresponds to one memory bank. A node is further divided into three zones: *zone_dma*, *zone_normal* and *zone_highmem*. *Zone_dma* is the first 16MB reserved for direct memory access (DMA) transfers. *Zone_normal* spans from 16MB to 896MB. This is the zone that is used by user applications and dynamic data requests within the kernel. This zone and *zone_dma* are linearly mapped in the kernel virtual address space. *Zone_highmem* is memory beyond 896MB. This zone is not linearly mapped and is used for allocations that require a large amount of contiguous memory in the virtual address

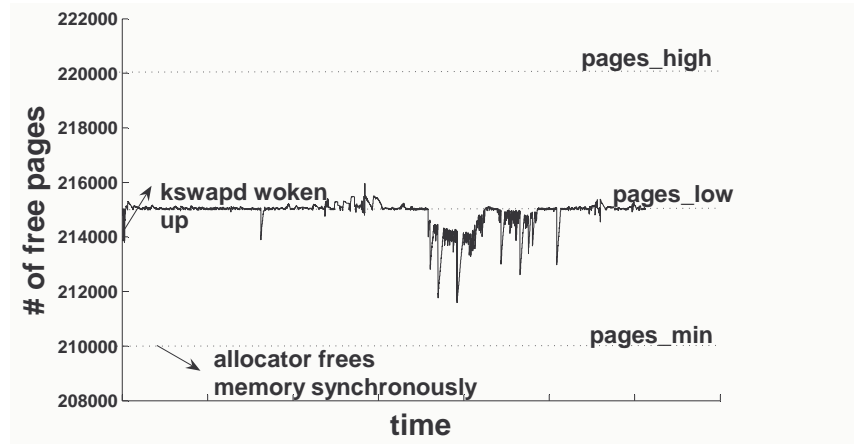


Figure 3.4: Zone balancing logic and the use of zone watermarks

space.

Each zone is always kept balanced by the kernel memory allocator called the *buddy allocator* and the page swapper *kswapd*. The balance is achieved using zone watermarks, which are basically indicators for gauging memory pressure in the particular zone. The zone watermarks have different values for all the three zones. These are initialized on startup depending on the number of pages present in the zones. These three watermarks are called *pages_min*, *page_low* and *pages_high* respectively as shown in Figure 3.4. When the number of free pages in the zones drops below *pages_low* pages, *kswapd* is woken up. *kswapd* tries to free pages by swapping unused pages to the swap store. It continues this process until the number of pages reaches *pages_high* and then, goes back to sleep. When the number of pages reaches *pages_min*, the buddy allocator tries to synchronously free pages. Sometimes, the number of free pages can go below the *pages_min*, due to atomic allocations requested by the kernel.

3.3.2 Attack description

The zone watermarks for each zone are stored in a global data structure called *zone_table*. *Zone_table* is an array of *zone_t* data structures that correspond to each zone. Zone watermarks are stored inside this data structure. The location of this table can be found by referring to the `System.map`

Watermark	Original Value	Modified Value
pages_min	255	210000
pages_low	510	215000
pages_high	765	220000
total free pages	144681	210065
Total number of pages in zone: 225280		

Table 3.1: Watermark values and free page count before and after the resource wastage attack

file. We wrote a simple kernel module to corrupt the zone watermarks for the *zone_normal* memory zone. The original and new values for these watermarks are shown in Table 3.1. We push the *pages_min* and the *pages_low* watermarks very close to the *pages_high* watermark. We also make the *pages_high* watermark very close to the total number of pages in that zone. This forces the zone balancing logic to maintain the number of free pages close to the total number of pages in that zone, essentially wasting a big chunk of the physical memory. Table 3.1 shows that 210065 (820.56 MB) pages are maintained in the free pool. This attack can be similarly carried out for other zones as well, wasting almost all memory installed on the system. The table indicates that only about 60MB are used and the rest is maintained in the free pool, causing applications to constantly swap to disk. This attack also imposes a performance overhead on applications as shown in Table 3.2. The three tasks that we used to measure the performance overhead are file copy of a large number of files, compilation of the Linux kernel and file compression of a directory. The table shows the time taken when these tasks were carried out on a clean kernel and after the kernel was tampered.

Application	Before	After	Degradation (%)
	Attack	Attack	
file copy	49s	1m, 3s	28.57
compilation	2m, 33s	2m, 56s	15.03
file compression	8s	23s	187.5

Table 3.2: Performance degradation in applications after the resource wastage attack

The performance degradation imposed by this attack is considerable.

3.3.3 Impact

This attack resembles a stealthier version of the resource exhaustion attack, which traditionally has been carried out over the network [51–53]. We try to achieve a similar goal, i.e to overwhelm the compromised system subtly by creating artificial memory pressure. This leads to a considerable performance overhead on the system. It also causes a large amount of memory to stay unused all the time to maintain the high number of pages in the free pool, leading to resource wastage. The attacker could keep the degradation subtle enough to escape detection over extended periods.

3.4 Entropy Pool Contamination

This attack contaminates the entropy pool and the polynomials used by the Pseudo-Random Number Generator (PRNG) to stir the pools. The goal of this attack is to degrade the quality of the pseudo random numbers that are generated by the PRNG. The kernel depends on the PRNG to supply good quality pseudo random numbers, which are used by all security functions in the kernel, as well as by applications for key generation, generating secure session id's, etc. All applications and kernel functions that depend on the PRNG are in turn open to attack.

3.4.1 Background

The PRNG provides two interfaces to user applications namely */dev/random* and */dev/urandom* as shown in Figure 3.5. The PRNG depends on three pools for its entropy requirements: the primary pool, the secondary pool and the urandom pool. The */dev/random* is a blocking interface and is used for very secure applications. The device maintains an entropy count and blocks if there is insufficient entropy available. Entropy is added to the primary pool from external events such as keystrokes, mouse movements, disk activity and network activity. When a request is made for random bytes, bytes are moved from the primary pool to the secondary and the urandom pools. The */dev/urandom*

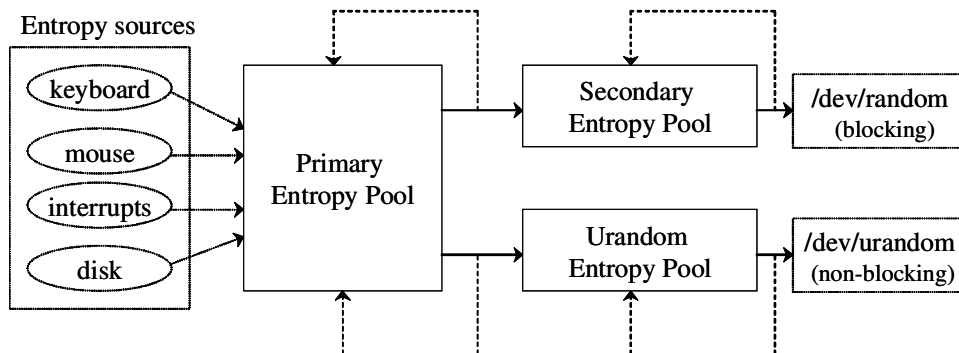


Figure 3.5: The Linux random number generator

interface, on the other hand, is non-blocking. The contents of the pool are stirred when the bytes are extracted from the pools. A detailed analysis of the Linux random number generator is available in [54].

3.4.2 Attack description

This attack constantly contaminates the entropy pool by writing zeroes into all the pools. This is done by loading an attack module that consists of a kernel thread. The thread constantly wakes up and writes zeroes into the entropy pools. It also attacks the polynomials that are used to stir the pool. Zeroing out these polynomials nullifies a part of the extraction algorithm used by the PRNG. The location of the entropy pool is not exported by the Linux kernel. We can find the location by simply scanning kernel memory. Entropy pool has the cryptographic property of being completely random [55]. Since we know the size of the entropy pools, this can be found by running a sliding window of the same sizes through memory and calculating the entropy of the data within the window. Kernel code and data regions are more ordered than the entropy pools and have a lower entropy value. The pool locations can therefore be successfully located.

We measured the quality of the random numbers generated by using the diehard battery of tests [56]. The results are summarized in Table 3.3. Diehard is the suite of tests used to measure the quality of random numbers generated. Any test that generates a value extremely close to 0 or 1

File #	bday	operm	binrnk6x8	cnt1s	parkinglot	mindist	sphere	squeeze	osum	craps
1	0.765454	0.497607	0.197306	0.000000	0.159241	0.000000	0.893287	0.423572	0.641313	0.147407
2	0.044118	0.180747	0.143452	0.000000	0.012559	0.000000	0.055361	0.769919	0.002603	0.066102
3	0.079672	0.999996	0.467953	0.000000	0.132155	0.000000	0.001550	0.190808	0.032007	0.468605
4	0.009391	0.000334	0.010857	0.000000	0.400118	0.000000	0.000258	0.573443	0.051299	0.057709
5	0.059726	0.996908	0.754544	0.000000	0.065416	0.000000	0.212797	0.276961	0.009343	0.389614
6	0.384023	0.975071	0.003450	0.000000	0.004431	0.000000	0.021339	0.047575	0.139662	0.082087
7	0.002450	0.458676	0.014060	0.000000	0.002061	0.000000	0.000010	0.044232	0.068223	0.836221
8	0.001195	0.840548	0.115478	0.000000	0.192544	0.000000	0.001535	0.024058	0.000078	0.214631
9	0.427721	0.553566	0.138635	0.000000	0.311526	0.000000	0.071177	0.296367	0.003107	0.679244
10	0.654884	0.106287	0.212463	0.000000	0.072483	0.000000	0.212785	0.338967	0.122016	0.710536

Table 3.3: Results of running the Diehard battery of tests after contamination of the entropy pool

represents a failing sequence. More about the details of these tests can be found in [56]. We run the tests over ten different 10MB files that were generated by reading from the `/dev/random` device. The table shows that the sequence that is generated after attack, fails miserably in two of the tests: *cnt1s* and *mindist* and partially in the others. A failure in any one of the tests means that the PRNG is no longer cryptographically secure.

3.4.3 Impact

After the attack, the generated pseudo random numbers are of poor quality, leaving the system and applications vulnerable to cryptanalysis attacks.

3.5 Disable Pseudo-Random Number Generator (PRNG)

This attack overwrites the addresses of the device functions registered by the PRNG with the function addresses of the attack code. The original functions are never invoked. These functions always return a zero when random bytes are requested from the `/dev/random` or `/dev/urandom` devices. Although this appears similar to the attack by traditional rootkits that hook into function pointers, there is a subtle difference. Since this particular device does not affect user-level view of objects, this is not a target for achieving hiding behavior and hence, not monitored by kernel integrity monitors.

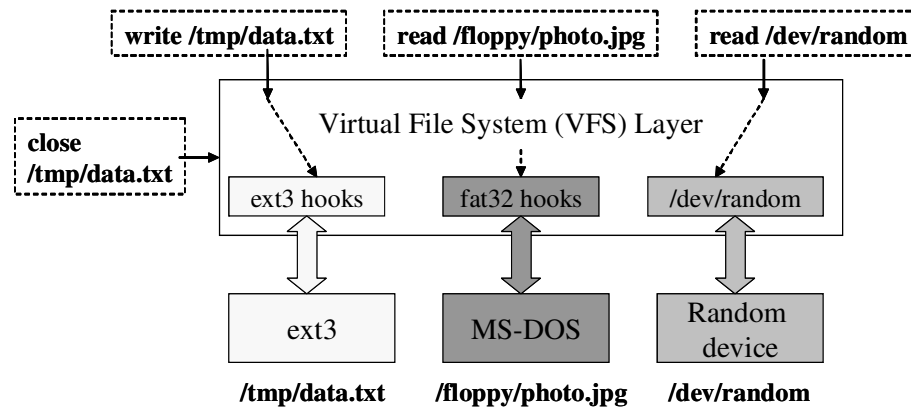


Figure 3.6: File and device hooks in the Linux virtual file system layer

3.5.1 Background

Linux provides a flexible architecture and a common interface for different file systems and devices. This interface is provided by a layer called the virtual file system (VFS) layer. A new file system or a device provides a set of hooks when registering with the VFS layer. Figure. 3.6 depicts two file systems *ext3* and *MS-DOS* and one device */dev/random* that are registered with the VFS layer. This enables user applications to access files residing on both file systems and the access to the device file with a common set of system calls. The system call is first handled by the VFS code. Depending on where the file resides, the VFS layer invokes the appropriate function registered by the file system or device during registration. Some system calls such as the *close* system call are directly handled by the VFS layer, which simply requires release of resources.

3.5.2 Attack description

The kernel provides functions for reading and writing to the */dev/random* and */dev/urandom* devices. The data structures used to register the device functions are called *random_state_ops* and *urandom_state_ops* for the devices */dev/random* and */dev/urandom*, respectively. These symbols are

exported by the 2.4 kernel but are not exported by the 2.6 kernel. We could find this data structure by first scanning for function opcodes of functions present within *random_state_ops* and *urandom_state_ops*. We then used the function addresses in the correct order to find the data structure in memory. Once these data structures are located in memory, the attack module replaces the genuine function provided by the character devices with the attack function. The attack function for reading from the device simply returns a zero when bytes are requested. After the attack, every read from the device returns a zero.

3.5.3 Impact

All security functions within the kernel and other security applications rely on the PRNG to supply pseudo random numbers. This attack stealthily compromises the security of the system, without raising any suspicions from the user.

3.6 Intrinsic Denial of Service

This attack causes performance degradation on applications by throttling the number of processes that an application can create to perform tasks in parallel. It achieves this by corrupting data used by the clone system call in Linux. This attack stealthily causes a measured degree of denial of service because resources beyond a certain threshold become temporarily unavailable to applications, which therefore experience a slowdown.

3.6.1 Background

The kernel relies on the process creation mechanisms to satisfy user requests. Especially servers are designed to be multi process or multi threaded; they constantly create new processes/threads to service requests obtained from clients. Each process is scheduled to handle a single client request to improve server efficiency by utilizing concurrent execution. New processes are created using the clone system call in Linux. The flags passed to clone determine the degree of sharing between the

parent and the child processes.

3.6.2 Attack description

This attack changes the *max_threads* variable used by the clone system call. This variable is used to check if the total number of processes on the system created exceeds the total number of processes that are allowed to be created. This check within the *clone* system call is incorporated to curtail fork bombs. The *max_threads* variable is in the static kernel region and its address is available in the kernel symbol file. By default, an upper limit of 14,336 is set on the total number of processes that can exist on a system. The total number of processes existing at the time of attack was 33. The attack changes this value to the number of processes running on the system to 35, severely limiting the number of new processes that can be created on the system. Subsequent system calls to create new processes receive an error message, once the number of processes exceeds 35, indicating the temporary unavailability of the resource. Applications are programmed to handle this error code and therefore simply function at less than full capacity.

3.6.3 Impact

Applications experience a slowdown because it is not able to concurrently execute multiple tasks. This attack resembles an intrinsic denial of service attack, where the service is unable to function at its full capacity. The level of stealth can range from moderate performance loss to a much severe one simply by controlling the value of the *max_threads* kernel variable.

3.7 Altering Real Time Clock Behavior

The real time clock (RTC) on the system provides the system time and features such as setting an alarm clock for scheduled execution of applications at later points in time. This attack alters the behavior of the real time clock in such a way that alarms registered by certain select applications, such as anti-virus and other intrusion detection systems running on the systems are never triggered.

This disables scheduled virus scans and other defense activities carried out on the system, making it vulnerable to attacks.

3.7.1 Background

The real time clock in a computer system is powered by a small battery or accumulator and continues to tick even when the system is turned off. It can be programmed to issue periodic interrupts or issue an interrupt when the clock reaches a certain value. Linux uses the RTC to retrieve the date and time. The RTC driver provides the device */dev/rtc* to applications, which they can program and use. The system time can be set by the administrator using the clock system program.

3.7.2 Attack description

The RTC is used by applications that rely on periodic execution of tasks. A classic example of such an application is the anti-virus software. A user typically schedules complete disk scans for viruses and worms when the system is not in use because it is a time consuming process that slows down the system significantly, while the activity is in progress. For example, a periodic scan of the system might be scheduled to run every Sunday morning at 3:00 am. The anti-virus program relies on the RTC to issue an interrupt when the clock reaches this time.

The goal of this attack is to disable the scheduled execution of the anti-virus program. Applications set such an alarm by using the *ioctl* system call on the */dev/rtc* device. This attack works by overwriting the function pointer for the *ioctl* system call, which is stored within the data structure *rtc_fops*. The malicious function can selectively disable alarms only for certain applications of interest, such as the anti-virus software, thereby other regular applications function flawlessly.

3.7.3 Impact

The system continues to be vulnerable to attacks as anti-virus and intrusion detection systems do not run at their scheduled times. This attack successfully lowers the system defenses.

3.8 Routing Cache Pollution

The goal of this attack is to pollute the routing cache on a routing system and divert traffic meant for a certain systems through a different gateway. The alternative route might consist of malicious systems that log all network traffic to perform traffic analysis and extract sensitive information from the victim systems.

3.8.1 Background

The kernel refers to the forwarding information base, also known as the static routing table, to determine the next hop of an IP packet. The *main_table* variable points to this complex array of data structures that store the static routing information. This information is used by the kernel for routing packets. Looking up this information by walking through the data structures is quite a slow process. To avoid this delay, the kernel maintains a routing cache for the most recently discovered routes. This cache includes several entries in sorted order such that more frequently accessed entries can be accessed more quickly. The cache is available in the variable *rt_hash_table* and each entry in the cache is of type *rtable*. The *rtable* data structure stored information about the source and destination IP address, the gateway IP address and other data relative to the route specified by the entry.

3.8.2 Attack Description

This attack corrupts the concerned entry in the routing cache by setting the gateway to point to a malicious node on the inter network. This could corrupt the gateway for packets meant for a certain destination. For example, the attacker might be interested in redirecting traffic generated for a specific bank website. The cache entry corresponding to the IP address of this website can be corrupted so that the traffic is routed through the malicious gateway. The malicious gateway can analyze all traffic and extract sensitive information. Since the entry for the destination in the routing cache is available, the kernel does not refer to the static routing table but instead uses the

information from the routing cache to forward packets.

3.8.3 Impact

The compromised systems faithfully forwards packets to a malicious gateway, giving the attacker access to all the packets that the attacker is interested in. The attacker can extract sensitive information by performing traffic and packet analysis.

3.9 Defeating In Memory Signature Scans

The goal of this attack is to defeat malware detectors that use in-memory signature scans running on the same system, by providing them with a fake view of memory. The attack achieves this goal by installing malicious read functions for the */dev/kmem* and the */dev/mem* devices, which provide interfaces for reading and writing to the kernel virtual address space and the system physical memory respectively.

3.9.1 Background

The */dev/mem* and */dev/kmem* character special files on a Linux system provide access to a device driver that allows read and write access to system memory. */dev/kmem* Only privileged users are allowed to read or write to these files. The device */dev/kmem* accesses data from the kernel virtual memory. The device */dev/mem* reads data from the system physical memory. Reading from these files returns the memory contents existing at the respective memory locations. Writing allows for patching memory with the required data. Rootkits also use the */dev/kmem* interface to patch the running kernel.

3.9.2 Attack description

This attack is similar in objectives to the rootkit Shadow Walker, which makes use of the Pentium split TLB architecture and modification to the kernel page fault handler to fake memory contents. This attack achieves the same by overwriting the function pointers registered by the devices */dev/mem* and */dev/kmem*. These are stored in the virtual file system layer in the data structure *kmem_fops* and *mem_fops*, of type *struct file_operations*. The malicious handlers for the read function can present a counterfeit view of the memory pages, thus thwarting all detection software that uses these interfaces to scan memory for malware signatures.

3.9.3 Impact

Rootkit detectors that run on the same system and scan system memory for attack signatures. This attack thwarts such detectors by providing them with a fake view of memory, doctored as desired by the attack. Therefore it successfully evades detection.

3.10 Attack Categorization

We have identified several attack categories based on the tampering techniques employed to achieve rootkit functionality. These categories are derived from the techniques used by publicly available rootkits as well as the advanced stealth attacks on kernel data. Broadly classified, attacks manipulate either control data or non-control data in the kernel. Modifying control data allows the rootkit to redirect control flow to its malicious function. Modification of non-control data alters the behavior of the kernel algorithms and in some cases also alters the control flow to execute the malicious code injected by the attacker. This attack categorization spans across static as well as dynamic data in the kernel.

Our motivation behind creating this categorization is to inspire the building of novel defense

techniques that are generic. These can apply to an entire class of attacks when applied comprehensively to all data structure in the kernel, rather than individual attacks themselves.

3.10.1 Control data modifications

We have identified two main categories that rootkits use to modify control data, namely control hijacking and control interception. In both cases, control flow data (function pointers) is substituted by some other value. The function pointer data is typically immutable data, which does not change after initialization. Both sub-categories differ in terms of the functionality. All function pointers in the kernel are susceptible to control data modification attacks.

Control hijacking.

Control hijacking attack is a form of manipulating the control flow within a kernel control path. This type of attack redirects the control flow to the attack code and the original code is never invoked or it sets the function pointer to NULL, effectively disabling the function that the pointer points to.

Control Interception.

Control interception is a technique used by most conventional rootkits. These attacks intercept the kernel control path in such a way that control first flows to the attack code. The attack code then invokes the original code. In this fashion, the attacker is able to filter requests to and responses from the original code. Control interception is typically used for hiding the attacker's files, processes and network connections.

3.10.2 Non-control data modifications

Two main categories exist in modifying non-control data namely control tapping and data value manipulation.

Control Tapping.

This type of attack effectively diverts the control flow by simply modifying non-control data in the kernel, leading to the invocation of the malicious function installed by the attacker. In other words, the control flow is transferred in such a way that the attack code is not able to manipulate the arguments being passed to the original function or the return values. It is only invoked on every call to the original function.

Data Value Manipulation.

These attacks rely on manipulating values of critical variables or aggregate data structures such as arrays and linked lists, which in turn directly or indirectly alter the behavior of the kernel algorithms. This category of attacks can further be classified based on the type of variables that they change, namely, mutable variables changed during the normal operation of the kernel or immutable data that never changes once the system is initialized. Sometimes attacks might need to modify both kinds of variables and therefore, might use a hybrid approach.

3.10.3 Attacks in this chapter

We classify the attacks discussed in this chapter based on the tampering technique, according to the categorization that we proposed. We also show other aspects of the attack i.e. the type of data modified and the location of the data modified. Table 3.7 summarizes this information.

3.11 Summary

The operating system kernel exposes a large number of data structures that can be subtly exploited by the attacker. In this chapter, we demonstrated a new class of attacks that exploit kernel data and achieve specific malicious goals. The attacks are stealthy by design and attack several kernel subsystems, effectively altering the behavior of the kernel algorithms. Since these attacks achieve

Attack	Location of data		Type of Data		Attack category
	Static	Dynamic	Control	Non-control	
Disable firewall		✓	✓		Control hijacking
Resource wastage	✓			✓	Data value manipulation
Entropy pool contamination		✓		✓	Data value manipulation
Disable PRNG	✓		✓		Control hijacking
Intrinsic denial of service	✓			✓	Data value manipulation
Altering RTC behavior	✓		✓		Control interception
Routing cache pollution		✓		✓	Data value manipulation
Defeating in-memory scans	✓		✓		Control interception

Figure 3.7: Attack categorization

the malicious functionality by modifying data structures in the kernel, they do not exhibit hiding behavior and therefore, cannot be detected by tools that use hiding behavior as a symptom for detection. These attacks serve to demonstrate that the threat to kernel data is realistic and systemic, thus requiring a comprehensive protection scheme. We discuss one such detection scheme that we developed for detecting advanced stealth attacks in the next chapter.

Chapter 4

Attack Detection via Invariant Inference

Control data within the kernel has been a common target of kernel rootkits because it allows for easy redirection of control flow to malicious code. This form of control interception has been especially popular with conventional rootkits because it allows the rootkit to filter requests and responses, subsequently affecting application level views of the system. Researchers have lately proposed solutions that allow for automatic and comprehensive validation of function pointers within the kernel [12, 57].

4.1 Problem Statement

Rootkits have already evolved to thwart detectors that validate control data. These rootkits alter non-control data structures in the kernel to achieve their goals. Though non-control data attacks are harder to conceive, the kernel presents a much larger attack surface for them compared to control data. Checking the integrity of non-control data is much more complex because a large part of it is routinely modified by the kernel. To verify the integrity of non-control data, architectures have been proposed that require manually written specifications of constraints that hold on data structures [15]. These specifications are supplied by an expert who has a detailed understanding of kernel data structure semantics. Kernel data structures are continuously monitored during runtime against these specifications, and violations are used as indicators of rootkit behavior. While this approach has

the advantage of detecting sophisticated rootkits, developing specifications is currently a manual procedure. Because the kernel maintains several hundred data structures, the specification writer could either fail to supply certain integrity specifications, *e.g.*, because he is unaware that they exist, or may fail to realize how a rootkit could exploit them.

In this chapter, we present a novel approach that automatically and uniformly checks the integrity of control and non-control data. This approach is based on the hypothesis that several data structures in the kernel exhibit invariants at runtime during the normal operation of a clean kernel. A rootkit that alters the behavior of the kernel by modifying kernel data, violates some of these invariants.

4.2 Approach

Our approach is based upon *automatic inference of data structure invariants* that can uniformly detect rootkits that modify both control and non-control data. The key idea is to monitor the values of kernel data structures during a training phase, and hypothesize invariants that are satisfied by these data structures. These invariants include properties of both control and non-control data structures that serve as specifications of data structure integrity. For example, an invariant could state that the values of elements of the system call table are a constant (an example of a control data invariant). Similarly, an invariant could state that all the elements of the *running-tasks* linked list (used by the kernel for process scheduling) are also elements of the *all-tasks* linked list that is used by the kernel for process accounting (an example of a non-control data invariant) [15]. These invariants are then checked during an enforcement phase; violation of an invariant indicates the presence of a rootkit. Because invariants are inferred *automatically* and *uniformly across both control and non-control data structures*, our approach overcomes the shortcomings of prior rootkit detection techniques.

This approach is very similar to that used for dynamic invariant inference in application programs by tools such as Daikon [58] and Diduce [59]. These tools instrument application programs to generate variable values at certain program points. The instrumented application is run several times during the training period to collect variable traces. The tool infers invariants over program variables from the data traces. In contrast, our system observes kernel data structure values asynchronously, and these values are converted and fed into Daikon for invariant inference.

To evaluate the viability of our approach, we built *Gibraltar*, a rootkit detection tool that automatically infers invariants on kernel data structures. Gibraltar periodically captures snapshots of kernel memory via an external PCI card. It uses these snapshots to reconstruct kernel data structures, and adapts Daikon [58], an invariant inference tool for application programs, to infer invariants on kernel data structures. Gibraltar has automatically extracted around 718,000 invariants on kernel data. In experiments with twenty rootkits, including those that modify non-control data, we found that Gibraltar detected *all* rootkits with a false positive rate of just 0.65%, and imposed a runtime monitoring overhead of 0.49%.

4.3 Invariants and Stealth Attacks

This section motivates the use and effectiveness of data structure invariants at detecting rootkits by presenting six previously demonstrated attacks that employ stealth techniques [14, 15, 30]. These attacks either modify non-control kernel data (cf. Attacks 1-4) or modify kernel control data without affecting user-level objects in the system (cf. Attacks 5 and 6), which are typically monitored by rootkit detection tools. Each of these attacks is successfully detected by Gibraltar; where applicable, we also discuss existing tools that can detect each attack.

For each attack presented below, we also describe a data structure invariant (automatically inferred by Gibraltar by observing the execution of an uncompromised kernel) that is violated by the

attack. In addition, we also describe the semantic meaning of each invariant, *i.e.*, the reason why a data structure satisfies the property specified by the invariant in an uncompromised kernel. The invariants listed in this section are examples drawn from several thousand invariants that are automatically inferred by Gibraltar. Particularly noteworthy in the examples below is the heterogeneity of the data structures over which Gibraltar infers invariants. Although these invariants can be examined, interpreted and refined by a security expert, Gibraltar, by default, automatically uses these invariants as specifications of data structure integrity. As we show in Section 4.5, Gibraltar’s default approach effectively detects rootkits with a low false positive rate.

4.3.1 Attack 1: Entropy pool contamination

The kernel uses the pseudo random number generator (PRNG) to obtain randomness needed to seed several other security-critical applications. The goal of the entropy pool contamination attack [14] is to contaminate entropy pools and associated polynomials used by the PRNG, so as to degrade the quality of random numbers that it generates.

- **Attack.** The PRNG uses the primary and secondary entropy pools, to generate random numbers. The primary pool derives entropy from external events such as keystrokes, mouse movements, disk and network activity. As a request arrives for a random number, the kernel extracts bytes from the primary pool and moves them to the secondary pool. Bytes extracted from the secondary pool are in turn used to provide random numbers to kernel functions and user-level applications.

To ensure that the numbers generated by the PRNG are pseudo random, the contents of the pools are updated using a stirring function each time bytes are extracted from the pools. The stirring function uses a polynomial whose coefficients are specified in five integer fields of a *struct poolinfo* data structure, namely *tap1*, *tap2*, *tap3*, *tap4* and *tap5*. This attack zeroes the coefficients of the polynomial, which renders ineffective part of the algorithm used to extract bytes from the pool. It also writes zeroes constantly into the entropy pools. Consequently, the numbers generated by the

<code>poolinfo.tap1</code>	\in	{26, 103}
<code>poolinfo.tap2</code>	\in	{20, 76}
<code>poolinfo.tap3</code>	\in	{14, 51}
<code>poolinfo.tap4</code>	\in	{7, 25}
<code>poolinfo.tap5</code>	$==$	1

The invariants satisfied by the coefficients of the polynomial used by the stirring function in the PRNG. The coefficients are fields of a struct `poolinfo` data structure, shown above as `poolinfo`. These invariants are violated by the entropy pool contamination attack (Section 4.3.1).

Figure 4.1: Invariants violated by the entropy pool contamination attack

PRNG are no longer random.

- **Invariants.** Figure 4.1 shows the invariants that Gibraltar identifies for the coefficients of the polynomial that is used to stir entropy pools in an uncompromised kernel (the `poolinfo` data structure shown in this figure is represented in the kernel by one of `random_state->poolinfo` or `sec_random_state->poolinfo`). The coefficients are initialized upon system startup, and must never be changed during the execution of the kernel. The attack violates these invariants when it zeroes the coefficients of the polynomial. Gibraltar detects this attack when the invariants are violated.

Attack 2: Process hiding

The goal of this attack is to hide a (possibly malicious) user-space process from the system utilities, such as `ps`. The attack operates by modifying the contents of the kernel linked lists used for process accounting and scheduling [13, 15].

- **Attack.** This attack relies on the fact that process accounting utilities, such as `ps`, and the kernel's task scheduler consult different process lists. The process descriptors of all tasks running on a system belong to a linked list called the `all-tasks` list (represented in the kernel by the data structure `init_tasks->next_task`). This list contains process descriptors headed by the first process created on the system. The `all-tasks` list is used by process accounting utilities. In contrast, the scheduler uses a second linked list, called the `run-list` (represented in the kernel by `run_queue_head->next`), to

$$\text{run-list} \subseteq \text{all-tasks}$$

The invariant that detects the process hiding attack (Section 4.3.1). In this attack, a task that is not in the all-tasks linked list appears in the run-list linked list, which is used by the kernel's task scheduler.

Figure 4.2: Invariant violated by the hidden process attack

schedule processes for execution.

The process hiding attack removes the process descriptor of a malicious user-space process from the *all-tasks* list (but not from the *run-list* list). This ensures that the process is not visible to process accounting utilities, but that it will still be scheduled for execution.

- **Invariants.** Figure 4.2 presents the invariant automatically discovered by Gibraltar. When a rootkit attempts to remove a task from the *all-tasks* list, this invariant is violated, and is therefore, detected by Gibraltar. We note that this attack was previously described by Petroni *et al.* [15] as an example of a non-control data attack. They also describe an invariant enforcement tool to detect such attacks; however, in contrast to Gibraltar, their enforcement tool requires data structure invariants, such as the one in Figure 4.2, to be supplied manually by a security expert.

Attack 3: Adding binary formats

The goal of this attack is to invoke malicious code each time a new process is created on the system [30]. While rootkits typically achieve this form of hooking by modifying kernel control data, such as the system call table, this attack works by inserting a new binary format into the system.

- **Attack.** This attack operates by introducing a new binary format into the list of formats supported by the system. The handler provided to support this format is malicious in nature. The binary formats supported by a system are maintained by the kernel in a global linked list called *formats*. The binary handler, specific to a given binary format, is also supplied when a new format is registered.

A new process is created on the system via the system call *sys_execve*. This system call creates

length(formats) == 2

Invariant inferred on the formats list; the attack discussed in Section 4.3.1 modifies the length of this list.

Figure 4.3: Invariant violated by the adding binary formats attack

the process address space, sets up credentials and in turn calls the function *search_binary_handler*, which is responsible for loading the binary image of the process from the executable file. The function *search_binary_handler* iterates through the *formats* list to look for an appropriate handler for the binary that it is attempting to load. As it traverses this list, it invokes each handler in it. If a handler returns an error code `ENOEXEC`, the kernel considers the next handler on the list; it continues to do so until it finds a handler that returns the code `SUCCESS`.

This attack works by inserting a new binary format in the *formats* list and supplying the kernel with a malicious handler that returns the error code `ENOEXEC` each time it is invoked. Because the new handler is inserted at the head of the *formats* list, the malicious handler is executed each time a new process is executed.

- **Invariants.** Gibraltar infers the invariant shown in Figure 4.3 on the *formats* list on our system, which has two registered binary formats, namely *a.out* and *elf*. The size of the list is constant after the system starts, and changes only when a new binary format is installed. Because this attack inserts a new binary format, thereby changing the length of the *formats* list, it violates the invariant in Figure 4.3; consequently, Gibraltar detects this attack.

Attack 4: Resource wastage

This attack creates artificial pressure on the memory subsystem [14], thereby forcing the memory management algorithms to constantly free memory by swapping pages to disk. In spite of the availability of free memory, this memory is not used either by the kernel or by user-space applications.

```
zone_table[1].pages_min == 255
zone_table[1].pages_low == 510
zone_table[1].pages_high == 765
```

(a) Invariants inferred for watermarks.

```
zone_table[1].pages_min = 210,000
zone_table[1].pages_low = 215,000
zone_table[1].pages_high = 220,000
```

(b) Watermark values after the attack.

Part (a) shows the invariants that Gibraltar inferred for `zone_table[1]`, a data structure of type `struct zone_struct` (Gibraltar infers similar invariants for other elements of the `zone_table` array). Part (b) shows the values of the watermarks after a resource wastage attack. The total number of pages on the system was 225,280.

Figure 4.4: Invariants violated by the resource wastage attack

Continuous swapping to disk also affects the performance of the system.

- **Attack.** The kernel’s memory management unit ensures that there are always free pages in memory to fulfil allocation requests made both from the kernel and user-space applications. To do so, it employs *memory balancing algorithms* that use three watermarks to gauge memory pressure, namely, the fields `pages_min`, `pages_low` and `pages_high`, of a `struct zone_struct` data structure. When the number of free pages in the system drops below the `pages_low` watermark, the kernel asynchronously swaps unused pages to disk. This process continues until the number of pages reaches the `pages_high` watermark. In contrast, if the number of free pages available drops below the `pages_min` watermark, the kernel synchronously swaps pages to disk.

This attack manipulates the three watermarks and sets their values close to the number of free pages in the system. Consequently, the number of free pages frequently drops below the `pages_min` and `pages_low` watermarks, forcing the kernel to continuously swap pages to disk, thereby creating synthetic memory pressure in the system.

- **Invariants.** Gibraltar identifies the invariants shown in Figure 4.4(a) for the three watermarks. These values are initialized upon system startup, and typically do not change in an uncompromised kernel. Figure 4.4(b) shows the values of these watermarks after the attack; the values of these watermarks are close to 225,280, which is the number of pages available on our system. Gibraltar

detects this attack because the values of the watermarks shown in Figure 4.4(b) violate the invariants shown in Figure 4.4(a).

Attack 5: Disabling firewalls

The goal of this attack is to stealthily disable firewalls installed on the system [14]; a user is unable to determine that firewalls have been disabled using the *iptables* utility. Instead, *iptables* shows the firewall rules that were created for the system, and the firewall appears to be enabled.

- **Attack.** This attack overwrites hooks in the Linux *netfilter* framework, which is a packet filtering framework in the Linux kernel. It provides hooks at multiple points in the networking stack, and was designed for kernel modules to register callbacks for packet filtering, packet mangling and network address translation. The *iptables* command line utility enforces firewall rules through the *netfilter* framework. Pointers to the *netfilter* hooks are stored in a global table called *nf_hooks*. This attack overwrites the hooks for the IP protocol, and instead sets them to point to the attack function, thereby effectively disabling the firewall. The table where the firewall rules are stored is unaltered and therefore, displayed by *iptables* when the user manually inspects the firewall.
- **Invariants.** Gibraltar inferred the invariant shown in Figure 4.5 for *netfilter*. The attack overwrites the hook with the attack function, thereby violating the invariant that function pointer *nf_hooks[2][1].next.hook* is a constant.

Because this attack modifies kernel function pointers, it can also be detected by SBCFI [12], which automatically extracts and enforces kernel control flow integrity. In fact, function pointer

```
nf_hooks[2][1].next.hook == 0xc03295b0
```

An invariant inferred for the netfilter hook. Firewalls are disabled by modifying the function pointer, thereby violating the invariant.

Figure 4.5: Invariant violated by the disable firewalls attack

invariants inferred by Gibraltar implicitly determine a control flow integrity policy that is equivalent to SBCFI. However, in contrast to SBCFI, Gibraltar can also detect non-control attacks, such as Attacks 1-4, discussed above.

Attack 6: Disabling the PRNG

This attack overwrites the addresses of the functions registered with the virtual file system layer by the PRNG [14]. The overwritten values point to functions that always return zero or an attacker-defined sequence when random bytes are requested from the PRNG; the PRNG's functions are never executed.

- **Attack.** The kernel provides two devices */dev/random* and */dev/urandom* from which random numbers can be read. The data structures used to register the device functions are *random_fops* and *urandom_fops*, both of which are variables of type *struct file_operations*. These data structures have function pointers to the various functions provided by the PRNG. The attack replaces the genuine function pointers to the various functions provided by the PRNG. The attack replaces the genuine function pointers for the *read* function within these data structures. After the attack has infected the kernel, every byte read from the two devices simply returns a zero. The original PRNG functions are never called.

- **Invariants.** The invariants inferred by Gibraltar on our system for the *random_fops* and *urandom_fops* are shown in Figure 4.6. The attack code changes the values of the above two function pointers, violating the invariants. As with Attack 5, this attack can also be detected using SBCFI.

<pre>random_fops.read == 0xc028bd48 urandom_fops.read == 0xc028bda8</pre>

Invariants inferred for the PRNG function pointers. These are replaced to point to attacker-specified code, thereby disabling the PRNG.

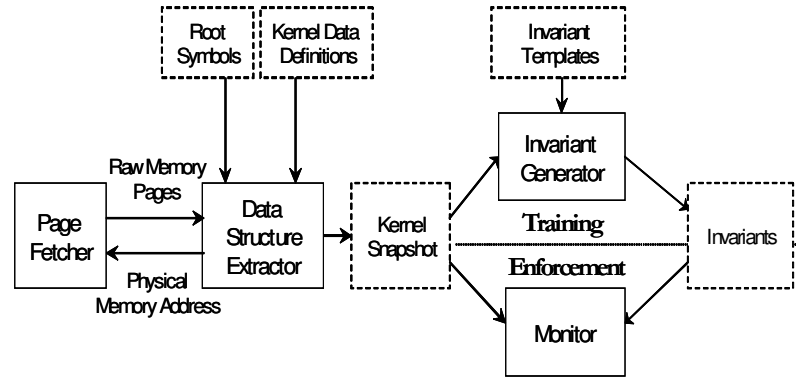
Figure 4.6: Invariant violated by the disable PRNG attack.

4.4 Design and Implementation

As Gibraltar aims to detect rootkits, it must execute on an entity that is outside the control of the kernel of the monitored machine, such as a virtual machine monitor [10], or on a coprocessor [8, 9]. In our architecture, Gibraltar executes on a separate machine (the *observer*) and monitors the execution of the target machine (the *target*). Both the observer and the target are interconnected via a secure back-end network using the Myrinet PCI intelligent network cards [60]. The back end network allows Gibraltar to remotely access the target kernel's physical memory. Gibraltar is built to infer data structure invariants when supplied with raw kernel memory as input. Therefore, it can be adapted to work with other infrastructures such as virtual machine monitors and coprocessors.

Figure 4.7 presents the architecture of Gibraltar. Like other anomaly detection tools, Gibraltar operates in two modes, namely, a *training* mode and an *enforcement* mode. In the training mode, Gibraltar infers invariants on data structures of the target's kernel. Training happens in a controlled environment on an uncompromised target (*e.g.*, a fresh installation of the kernel on the target machine). In the enforcement mode, Gibraltar ensures that the data structures on the target's kernel satisfy the invariants inferred during the training mode.

As shown in Figure 4.7, Gibraltar consists of four key components (shown in the boxes with solid lines). The *page fetcher* responds to requests by the *data structure extractor* to fetch kernel memory pages from the target. The data structure extractor, in turn, extracts values of data structures on the target's kernel by analyzing raw physical memory pages. The data structure extractor also accepts as input the data type definitions of the kernel running on the target machine and a set of root symbols that it uses to traverse the target's kernel memory pages. Both these inputs are obtained via an offline analysis of the source code of the kernel version executing on the target machine. The output of the data structure extractor is the set of kernel data structures on the target. The



Boxes with solid lines show components of Gibraltar. Boxes with dashed lines show data used as input or output by the different components.

Figure 4.7: The Gibraltar Architecture

invariant generator processes these data structures and infers invariants. These invariants represent properties of both individual data structures, *also known as objects* (e.g., scalars, such as integer variables and arrays and aggregate data structures, such as structs), as well as collections of data structures (e.g., linked lists). During enforcement, the *monitor* uses the invariants as specifications of kernel data structure integrity, which raises an alert when an invariant is violated by a kernel data structure. The following sections elaborate on the design of each of these components.

4.4.1 The page fetcher

Gibraltar executes on the observer, which is isolated from the target system. However, the observer must be able to faithfully reconstruct the state of the target's memory. Gibraltar achieves this goal by fetching physical memory pages and reconstructing the target's kernel data structures.

Gibraltar's page fetcher is a component that takes a physical memory address as input, and obtains the corresponding memory page from the target. The target runs a Myrinet PCI card to which the page fetcher issues a request for a physical memory page. Upon receiving a request, the firmware on the target initiates a DMA request for the requested page. It sends the contents of the physical page to the observer upon completion of the DMA. The Myrinet card on the target

system runs an enhanced version of the original firmware. Our enhancement ensures that when the card receives a request from the page fetcher, the request is directly interpreted by the firmware and serviced (rather than forwarding the request to a user-space application running on the target system). The page fetcher sends the physical memory address of the page to be fetched. If the required address is present in the linearly-mapped region of the kernel, it subtracts a fixed offset to get the physical address. If the address is not within this region, it refers to the kernel page tables to acquire the physical address of the page. (The page tables themselves reside in the linearly-mapped region of the target at a known physical memory location. Consequently, the page tables are first fetched and interpreted by the page fetcher.)

4.4.2 The data structure extractor

This component reconstructs snapshots of the target kernel's data structures from raw physical memory pages. During this process of reconstruction, it may issue requests to the page fetcher to fetch physical memory pages from the target.

The data structure extractor processes raw physical memory pages using two inputs to locate data structures within these pages. First, it uses a set of *root symbols*, which denote kernel data structures whose physical memory locations are fixed, and from which all data structures on the target's heap are reachable. In our implementation, we use the symbols in the *System.map* file of the target's kernel as the set of roots. Second, it uses a set of *type definitions* of the data structures in the target's kernel. Type definitions are used as described below to recursively identify all reachable data structures. We automatically extracted 1292 type definitions by analyzing the source code of the target kernel using a CIL module [61].

The data structure extractor uses the roots and type definitions to recursively identify data structures in physical memory using a standard worklist algorithm (see Figure 4.8). The extractor first adds the addresses of the roots to a worklist; it then issues a request to the page fetcher for memory

<p>Input: (a) R: addresses of roots; (b) Data structure definitions. Output: Set of all data structures reachable from R.</p> <ol style="list-style-type: none"> 1. $worklist = R$; 2. $visited = \phi$; 3. $snapshot = \phi$; 4. while $worklist$ is not empty do 5. $addr =$ remove an entry from $worklist$; 6. $visited = visited \cup \{addr\}$; 7. $M =$ physical memory page containing $addr$; 8. $obj =$ object at address $addr$ in M; 9. $snapshot = snapshot \cup$ value of obj; 10. foreach pointer p in obj do 11. if $p \notin visited$ 12. $worklist = worklist \cup \{p\}$; 13. return $snapshot$;

Figure 4.8: Algorithm used by the data structure extractor.

pages containing the roots. It extracts the values of the roots from these pages, and uses their type definitions to identify pointers to more (previously-unseen) data structures. For example, if a root is a C *struct*, the data structure extractor adds all pointer-valued fields of this *struct* to the worklist to locate more data structures in the kernel's physical memory. This process continues in a recursive fashion until all the data structures in the target kernel's memory (reachable from the roots) have been identified. *A complete set of data structures reachable from the roots is defined to be a snapshot.* The data structure extractor periodically probes the target and outputs snapshots.

When the data structure extractor finds a pointer-valued field, it may require assistance in the form of code annotations to clarify the semantics of the pointer. In particular, the data structure extractor requires assistance when it encounters linked lists, implemented in the Linux kernel using the *list_head* structure. In Linux, other kernel data structures that must be organized as a linked list (called *containers*) simply include the *list_head* data structure. Figure 4.9 shows an example of a *task_struct*, in which the field *run_list* is of type *list_head*. Objects of type *task_struct* are linked together as a list using the *next* and *prev* fields, which are members of the *list_head* structure.

```

struct task_struct {
    ...
    struct list_head CONTAINER(struct task_struct,run_list) run_list;
    ...
}

```

The field `run_list` within the structure `task_struct` points to the `run_list` field of another `task_struct` object.

Figure 4.9: An example showing the `CONTAINER` annotation

The kernel provides functions to add, delete, and traverse `list_head` data structures. To traverse and process a list of `task_struct` structures, the kernel would use locate and traverse to the `list_head` structures within the `task_struct` structure; it would then identify the corresponding `task_struct` objects in the list using pointer arithmetic (e.g., the `container_of` macro provided by the Linux kernel).

Linked lists within container objects are problematic for the data structure extractor. In particular, when it encounters a `list_head` structure, it will be unable to identify the container data structure (e.g., the `task_struct` data structure in Figure 4.9). To handle such linked lists, we use the `CONTAINER` annotation, as shown in Figure 4.9. The annotation explicitly specifies the type of the container data structure (`struct task_struct`) and the field within this type (`run_list`), which the `list_head` pointers point to. The extractor uses this annotation when it encounters the `run_list` field, and locates the container `task_struct` data structure. Therefore, the `CONTAINER` annotations disambiguate the semantics of the `list_head` pointer to the data structure extractor. In our experiments, we annotated all 163 annotations of the `list_head` data structure in the Linux-2.4.20 kernel.

In addition to linked lists, Gibraltar also requires assistance to disambiguate opaque pointers (`void *`), dynamically-allocated arrays and untagged unions. For example, the extractor would require the length of a dynamically-allocated arrays in order to traverse and locate objects in the array. These are not annotated in the current prototype.

Because the page fetcher obtains pages from the target asynchronously (without halting the target), it is likely that the data structure extractor will encounter inconsistencies, such as pointers to non-existent objects. Such invalid pointers are problematic because the data structure extractor will incorrectly fetch and parse the memory region referenced by the pointer (which will result in more invalid pointers being added to the worklist of the traversal algorithm). To remedy this problem, we currently place an upper bound on the number of objects traversed by the extractor. In our experiments, we found that on an idle system, the number of data structures in the kernel varies between 40,000 and 65,000 objects. We therefore, place an upper bound of 150,000; the data structure extractor aborts the collection of new objects when this threshold is reached. In our experiments, this threshold was rarely reached, and even so, only when the system was under heavy load.

4.4.3 The invariant generator

In the training mode, the output of the data structure extractor is used by the invariant generator, which infers likely data structure invariants. These invariants are used as specifications of data structure integrity.

To extract data structure invariants, we adapted Daikon [58], a state of the art tool invariant inference tool. Daikon attempts to infer likely program invariants by observing the values of variables during multiple executions of a program. Daikon first instruments the program to emit a trace that contains the values of variables at selected *program points*, such as the entry points and exits of functions. It then executes the program on a test suite, and collects the traces generated by the program. Finally, Daikon analyzes these traces and hypothesizes invariants—properties of variables that hold across all the executions of the program. The invariants produced by Daikon conform to one of several *invariant templates*. For example, the template `x == const` checks whether the value of a variable `x` equals a constant value `const` (where `const` represents a symbolic constant; if

x has the constant value 5, Daikon will infer $x == 5$ as the invariant). Daikon also infers invariants over *collections* of objects. For example, if it observes that the field *bar* of all objects of type *struct foo* at a program point have the value 5, it will infer the invariant “The fields *bar* of all objects of type *struct foo* have value 5.”

We had to make three key changes to adapt Daikon to infer invariants over kernel data structures.

- **Inference over snapshots.** Daikon is designed to analyze multiple execution traces obtained from instrumented programs and extract invariants that hold across these traces. We cannot use Daikon directly in this mode because the target’s kernel is not instrumented to collect execution traces. Rather, we obtain values of data structures by asynchronously observing the memory of the target kernel. To adapt Daikon to infer invariants over these data structures, we represent all the data structures in one snapshot of the target’s memory as a *single* Daikon trace. As described in Section 4.4.2, the data structure extractor periodically reconstructs snapshots of the target’s memory. Multiple snapshots, therefore, yield multiple traces. Daikon processes all these traces and hypothesizes properties that hold across all traces, thereby yielding invariants over kernel data structures
- **Naming data structures.** Because Daikon analyzes instrumented programs, it represents invariants using global variables and the local variables and formal parameters of functions in the program. However, because Gibraltar aims to infer invariants on data structures reconstructed from snapshots, the invariants output by Gibraltar must be represented using the root symbols. Gibraltar represents each data structure in a snapshot using its name relative to one of the root symbols. For example, Gibraltar represents the head of the *all-tasks* linked list, described in Section 4.3.1, using the name *init_tasks->next_task* (here, *init_tasks* is a root symbol). The extractor names each data structure as it is visited for the first time (in Lines 11

and 12 of Figure 4.8).

In addition, Gibraltar also associates each name with the virtual memory address of the data structure that it represents in the snapshot. These addresses are used during invariant inference, where they help identify cases where the same name may represent different data structures in multiple snapshots. This may happen because of deallocation and reallocation. For example, suppose that the kernel deallocates (and reallocates, at a different address) the head of the *all-tasks* linked list. Because the name *init_tasks->next_task* will be associated with different virtual memory addresses before and after allocation, it represents different data structures; Gibraltar ignores such objects during invariant inference.

- **Linked data structures.** Linked lists are ubiquitous in the kernel and, as demonstrated in Section 4.3, can be exploited subtly by rootkits. It is therefore important to preserve the integrity of kernel linked lists. Daikon, however, does not infer invariants over linked lists. To overcome this shortcoming, we represented kernel linked lists as arrays in Daikon trace files, and leveraged Daikon’s ability to infer invariants over arrays. We then converted the invariants that Daikon inferred over these arrays to invariants over linked lists.

Daikon infers invariants that conform to 75 different templates [58], and infers several thousand invariants over kernel data structures using these templates. In the discussion below, and in the experimental results reported in Section 4.5, we focus on five templates; in the templates below, *var* denotes either a scalar variable or a field of a structure.

- **Membership template ($\text{var} \in \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$).** This template corresponds to invariants that state that *var* only acquires a fixed set of values (in this case, *a*, *b* or *c*). If this set is a singleton *{a}*, denoting that *var* is a constant, then Daikon expresses the invariant as *var* == *a*.

- **Non-zero template ($\text{var} \neq \mathbf{0}$).** The non-zero template corresponds to invariants that determine that a `var` is a non-NULL value (or not 0, if `var` is not a pointer).
- **Bounds template ($\text{var} \geq \text{const}$), ($\text{var} \leq \text{const}$).** This template corresponds to invariants that determine lower and upper bounds of the values that `var` acquires.

The three example templates discussed above correspond to invariants over variables and fields of C *struct* data structures. These invariants can be inferred over individual objects, as well as over collections of data structures (*e.g.*, the fields *bar* of all objects of type *struct foo* have value 5). Invariants over collections describe a property that hold for *all members* of that collection across *all snapshots*.

- **Length template ($\text{length}(\text{var}) == \text{const}$).** This template describes invariants over lengths of linked lists.
- **Subset template ($\text{coll}_1 \subset \text{coll}_2$).** This template represents invariants that describe that the collection `coll1` is a subset of collection `coll2`. This is used, for instance, to represent invariants that describe that every element of one linked list is also an element of another linked list.

The last two example templates are used to describe properties of kernel linked lists. As reported in Section 4.5, in our experiments, invariants that conformed to the Daikon templates sufficed to detect all the control and non-control data attacks that we tested. However, to accommodate for rootkits that only violate invariants that conform to other kinds of templates, we may need to extend Gibraltar with more templates in the future. Fortunately, Daikon supports an extensible architecture. Newer invariant templates can be supplied to Daikon, thereby allowing Gibraltar to detect more attacks.

4.4.4 The monitor

During enforcement, the monitor ensures that the data structures in the target's memory satisfy the invariants obtained during training. As with the invariant generator, the monitor obtains snapshots from the data structure extractor, and checks the data structures in each snapshot against the invariants. This ensures that any malicious modifications to kernel memory that cause the violation of an invariant are automatically detected.

4.4.5 Persistent v/s Transient Invariants

The invariants inferred by Gibraltar can be categorized as either *persistent* or *transient*. Persistent invariants represent properties that are valid across reboots of the target machine, provided that the target's kernel is not reconfigured or recompiled between reboots. All the examples in Figures 4.1-4.6 are persistent invariants.

An invariant is persistent if and only if the names of the variables in the invariant persist across reboots *and* the property represented by the invariant holds across reboots. Thus, a transient invariant either expresses a property of a variable whose name does not persist across reboots or represents a property that does not hold across reboots. For example, consider the invariant in Figure 4.10, which expresses a property of a *struct file_operations* object. This invariant is transient because it does not persist across reboots. The name of this object changes across reboots as it appears at different locations in kernel linked lists; consequently, the number of *next* and *prevs* that appear in the name of the variable differ across reboots.

```
init_fs->root->d_sb->s_dirty.next->i_dentry.next->d_child.prev->d_inode->i_fop.read == 0xeff9bf60
```

Figure 4.10: Example of a transient invariant.

The distinction between persistent and transient invariants is important because it determines

the number of invariants that must be inferred each time the target machine is rebooted. In our experiments, we found that out of a total of approximately 718,000 invariants extracted by Gibraltar, approximately 40,600 invariants persist across reboots of the target system.

Although it is evident that the number of persistent invariants is much smaller than the total number of invariants inferred by Gibraltar (thus necessitating a training each time the target is rebooted), we note that this does not reflect poorly on our approach. In particular, the persistent invariants can be enforced as Gibraltar infers transient invariants after a reboot of the target machine, thus providing protection during the training phase as well. The cost of retraining to obtain transient invariants can potentially be ameliorated with techniques such as live-patching [62, 63], which can be used to apply patches to a running system.

4.5 Experimental Results

This section presents the results of experiments to test the effectiveness and performance of Gibraltar at detecting rootkits that modify both control and non-control data structures. We focus on three concerns:

- **Detection accuracy.** We tested the effectiveness of Gibraltar by using it to detect both publicly-available rootkits as well as those proposed in the research literature [14, 15, 30]. Gibraltar detected all these rootkits (Section 4.5.2).
- **False positives.** During enforcement Gibraltar raises an alert when it detects an invariant violation; if the violation was not because of a malicious modification, the alert is a false positive. Our experiments showed that Gibraltar has a false positive rate of 0.65% (Section 4.5.3).
- **Performance.** We measured three aspects of Gibraltar’s performance and found that it imposes a negligible monitoring overhead (Section 4.5.4).

All our experiments are performed on a target system with a Intel Xeon 2.80GHz processor with

Attack Name	Data Structures Affected
Rootkits from Packet Storm [64].	
Adore-0.42	System call table
Adore-ng	Vfs hooks, Udp recvmmsg
All-root	System call table
Kbd v3	System call table
Kis 0.9	System call table
Knark 2.4.3	System call table, Proc hooks
Linspy2	System call table
Modhide	System call table
Phide	System call table
Rial	System call table
Rkit 1.01	System call table
Shtroj2	System call table
Synapsys-0.4	System call table
THC Backdoor	System call table

This table shows the data structures modified by the rootkit. Gibraltar successfully detects all the above rootkits. The invariants violated are all Object invariants detected by the Membership(constant) template.

Table 4.1: Rootkits that modify control data

1GB RAM, running a Linux-2.4.20 kernel (infrastructure limitations prevented us from upgrading to the latest version of the Linux kernel). The observer also has an identical configuration.

4.5.1 Experimental methodology

Our experiments with Gibraltar proceeded as follows. We first ran Gibraltar in training mode and executed a workload that emulated user behavior (described below) on the target system. We configured Gibraltar to collect fifteen snapshots during training. Gibraltar analyzes these snapshots and infers invariants. We then configured Gibraltar to run in enforcement mode using the invariants obtained from training. During enforcement, we installed rootkits on the target system, and observed the alerts generated by Gibraltar. Finally, we studied the false positive rate of Gibraltar by executing a workload consisting of benign applications.

Workload.

We chose the Lmbench [65] benchmark as the workload that runs on the target system. This workload consists of a micro benchmark suite that is used to measure operating system performance. These micro benchmarks measure bandwidth and latency for common operations performed by applications, such as copying to memory, reading cached files, context switching, networking, file system operations, process creation, signal handling and IPC operations. This benchmark therefore, exercises several kernel subsystems and modifies several kernel data structures as it executes.

4.5.2 Detection accuracy

In this section, we report the detection accuracy of Gibraltar. We test Gibraltar with fourteen publicly available rootkits and six other attacks proposed by research literature.

Publicly available rootkits

We used fourteen publicly-available rootkits [64] to test the effectiveness of Gibraltar. Each of these rootkits modifies kernel data structures (in particular, we did not use rootkits that modify kernel code; these rootkits can trivially be detected by Gibraltar by ensuring that the invariant that kernel code area is an invariant). Table 4.1 summarizes the list of rootkits that modify kernel control data that we used in our experiments.

Gibraltar successfully detects all the above rootkits. Each of these rootkits violated a persistent invariant that conformed to the template `var == constant`. Because these rootkits modify kernel control flow, they can also be detected by SBCFI. The invariants on control data structures inferred by Gibraltar implicitly determine a control flow integrity policy that is equivalent to SBCFI.

Attack Name	Data Structures Affected	Invariant Type	Template
Entropy Pool Contamination	struct poolinfo	Collection	Membership
Hidden Process	all-tasks list	Collection	Subset
Linux Binfmt	formats list	Collection	Length
Resource Wastage	struct zone_struct	Object	Membership (constant)
Disable Firewall	struct nf_hooks[]	Object	Membership (constant)
Disable PRNG	struct random_state_ops struct urandom_state_ops	Object	Membership (constant)

Rootkits from research literature [14, 15, 30]. This table also shows the data structure modified by the attack, the type of the invariant violated and the template that this invariant conforms to.

Table 4.2: Rootkits from research literature.

Attacks from research literature

We used six attacks discussed in prior work [14, 15, 30] to test Gibraltar. These attacks, and the invariants that they violate were discussed in detail in Section 4.3. Table 4.2 summarizes these attacks, and shows the data structures modified by the attack, the invariant type (collection/object) violated, and the template that classifies the invariant. Each of the invariants that was violated was a persistent invariant, which survives a reboot of the target machine.

4.5.3 Invariants and false positives

We report the number of invariants inferred by Gibraltar, and the evaluation of false positives in this section.

Invariants

As discussed in Section 4.4, Gibraltar uses Daikon to infer invariants; these invariants express properties of both individual objects, as well as collections of objects (*e.g.*, all objects of the same type; invariants inferred over linked lists are also classified as invariants over collections). Table 4.3 reports the number of invariants inferred by Gibraltar on individual objects as well as on collections

Template	Object	Collection
Membership	643,622	422
Non-zero	49,058	266
Bounds	16,696	600
Length	NA	4,696
Subset	NA	3,580
Total	709,376	9,564

Invariants inferred by Gibraltar. These invariants are used as data structure integrity specifications during enforcement.

Table 4.3: Number of invariants inferred by Gibraltar.

Template	Object	Collection
Membership	0.71%	1.18%
Non-zero	0.17%	2.25%
Bounds	0%	0%
Length	NA	0.66%
Subset	NA	0%
Average false positive rate: 0.65%		

False positive rates, classified by the type of invariant and the template that classifies the invariant.

Table 4.4: Gibraltar false positive rate

of objects. Table 4.3 also presents a classification of invariants by templates; the length and subset invariants apply only to linked lists. As this table shows, Gibraltar *automatically* infers several thousand invariants on kernel data structures.

False positives

To evaluate the false positive rate of Gibraltar, we designed a test suite consisting of several benign applications, which performed the following tasks: **(a)** copying the Linux kernel source code from one directory to another; **(b)** editing a text document (an interactive task); **(c)** compiling the Linux kernel; **(d)** downloading eight video files from the Internet; and **(e)** perform file system read/write and meta data operations using the IOZone benchmark [66]. This test suite ran for 42 minutes on the target. We enforced the invariants inferred using the workload described in Section 4.5.1.

The false positive rate is measured as the ratio of the number of invariants for which violations are reported and the total number of invariants inferred by Gibraltar. Table 4.4 presents the false positive rate, further classified by the type of invariant (object/collection) that was erroneously violated by the benign workload, and the template that classifies the invariant. As this table shows, the overall false positive rate of Gibraltar was 0.65%. Improving the false positive rate significantly to make the system practical is an important direction for future work [67]. An automated filtering

strategies for classes of data structures could be identified and eliminated. Alternatively, a similar process could be carried out by a manual security expert.

4.5.4 Performance

We measured three aspects of Gibraltar’s performance: **(a)** training time, *i.e.*, the time taken by Gibraltar to observe the target and infer invariants; **(b)** detection time, *i.e.*, the time taken for an alert to be raised after the rootkit has been installed; and **(c)** performance overhead, *i.e.*, the overhead on the target system as a result of the DMA requests issued by the Myrinet PCI card.

Training time.

The training time is calculated as the cumulative time taken by Gibraltar to gather kernel data structure values and infer invariants when executing in training mode. Overall, the process of gathering 15 snapshots of the target kernel’s memory requires approximately 25 minutes, followed by 31 minutes to infer invariants, resulting in a total of 56 minutes for training.

Training is currently a time-consuming process because our current prototype invokes Daikon to infer invariants after collecting all the kernel snapshots. Training time can potentially be reduced by adapting Daikon to use an incremental approach to infer invariants. In this approach, Daikon would hypothesize invariants using the first snapshot, in parallel with the execution of the workload to produce more snapshots. As more snapshots are produced, Daikon can incrementally refine the set of invariants. We leave this enhancement for future work.

Detection time.

Gibraltar raises an alert when an invariant over a kernel data structure is violated. We measure the detection time as the interval between the installation of the rootkit and Gibraltar detecting that an invariant has been violated. Because Gibraltar traverses the data structures in a snapshot and checks invariants over each data structure, detection time is proportional to the number of objects in each

snapshot. Detection time also depends upon the order in which its algorithms traverse objects in the snapshot, and the data structure whose invariant is violated.

Gibraltar’s detection time varied from a minimum of fifteen seconds (when there were 41,254 objects in the snapshot) to a maximum of 132 seconds (when there were 150,000 objects in the snapshot). On average, we observed a detection time of approximately 20 seconds.

Monitoring overhead.

The Myrinet PCI card fetches raw physical memory pages from the target using DMA; because DMA increases contention on the memory bus, the target’s performance will potentially be affected. We measured this overhead using the Stream benchmark [68], a simple, synthetic benchmark that measures sustainable memory bandwidth. Measurement is performed four vector operations, namely, copy, scale, add and triad. The vectors are chosen so that they clear the last-level cache in the system, forcing data to be fetched from main memory.

Table 4.5 presents the bandwidth measurements for these four vector operations, both with Gibraltar’s monitoring turned off, and turned on. Bandwidth measurements and time taken for the four vector operations are shown. This table shows the maximum and minimum time taken for each operation, and the average over 100 executions. As this table shows, Gibraltar imposes a negligible overhead of 0.49% on the operation of the target system.

Function	Time (Monitoring OFF)			Time (Monitoring ON)			Overhead
	Avg.	Min	Max	Avg.	Min	Max	
Copy	0.2260	0.2259	0.2271	0.2272	0.2269	0.2277	0.48%
Scale	0.2239	0.2237	0.2242	0.2251	0.2248	0.2254	0.49%
Add	0.3316	0.3313	0.3321	0.3329	0.3326	0.3336	0.39%
Triad	0.3295	0.3292	0.3298	0.3308	0.3304	0.3314	0.37%

Table 4.5: Results from the Stream microbenchmark averaged over 100 iterations.

4.6 Summary

In this chapter, we presented a novel approach that uniformly detects rootkits that violate control as well as non-control data by identifying violations of automatically-inferred kernel data structure invariants. Our approach is based on extracting invariants by observing the behavior of a clean kernel at runtime. We present Gibraltar, a prototype tool that uses the above approach for rootkit detection. We presented the design and implementation of Gibraltar. We also present a comprehensive evaluation of Gibraltar on twenty rootkits that affect both control and non-control data structures and show that Gibraltar can detect all of them, with a low false positive rate and negligible monitoring overhead.

Chapter 5

Conclusions and Future Work

5.1 Concluding Remarks

Kernel-level rootkits pose a significant and growing threat to computer systems. Operating systems expose a large attack surface because they employ a diverse set of complex data structures, which can be manipulated in very subtle ways. Rootkit attacks that alter kernel data can be highly stealthy and are capable of causing long term damage to the system, in the absence of appropriate detection tools. Losses to the end user or corporations, victim to such attacks, might be in the form of exfiltration of sensitive information, performance degradation, wastage of system resources or system involvement in other malicious activities.

This dissertation has made several contributions in understanding the threat model and proposed automated techniques for the detection and containment of kernel rootkits. To contain the ongoing damage done to the system by conventional rootkits, this dissertation proposes a containment algorithm, built upon the virtualization architecture. This dissertation also identifies a novel class of attacks on the kernel, which are stealthy by design and achieve their malicious objectives by solely modifying kernel data. Finally, this dissertation also proposes a novel comprehensive detection method for kernel rootkits, based on automatic mining of kernel data structure invariants.

Conventionally, rootkits tamper with the kernel to achieve stealth, while most of the malicious functionality is provided by accompanying user space programs. Therefore, stealth is achieved by trying to hide the objects, such as files, processes and network connections present in user space belonging to the attacker. Since user space programs can access or modify user space objects using system calls, the rootkit is limited to manipulating code or data structures that are reachable from the system call paths alone. While rootkit detectors are able to detect such attacks effectively, they cannot stop the user space programs from continuing with their malicious activities. While an administrator can shutdown the system upon detection of a rootkit; such responses adversely affect the productivity of commercial systems and do not scale with growing rate of attacks. In this dissertation, we propose a containment algorithm that uses virtual machine introspection for detection. To perform containment, it keeps track of process dependencies and identifies processes at runtime that might be malicious and immediately kills them. This algorithm is also effective in containing worms and viruses that use rootkits to hide.

To better understand the threat model, we demonstrated a new class of stealth attacks that do not employ the traditional hiding behavior used by rootkits but are stealthy by design. They manipulate data within several different subsystems in the kernel to achieve their malicious objectives. They are based upon the observation that kernel rootkits need not necessarily be limited to manipulation of data structures that lie within the system call paths. Other subsystems within the kernel are also vulnerable to such attacks. To demonstrate this threat, we built several new attacks. We have designed attack prototypes to demonstrate that such attacks are realistic and indicative of a more systemic problem in the kernel.

Previously proposed rootkit detection techniques largely detect attacks that modify kernel control data; techniques that detect non-control data attacks, especially on dynamically-allocated data

structures, require specifications of data structure integrity to be supplied manually. In this dissertation, we present a novel rootkit detection technique that detects rootkits uniformly across control and non-control data. The approach is based on the hypothesis that several invariants are exhibited by kernel data structures at runtime during its correct operation. A rootkit that modifies the behavior of the kernel algorithms violates some of these invariants. To validate this hypothesis, this dissertation presents Gibraltar, a tool that automatically infers and enforces specifications of kernel data structure integrity. Gibraltar infers invariants uniformly across control and non-control kernel data, and enforces these invariants as specifications of data structure integrity. Our experiments showed that Gibraltar successfully detects rootkits that modify both control and non-control data structures, and does so with a low false positive rate and negligible performance overhead.

5.2 Future Work

Research over the past few years has made significant strides in the development of stealth attacks and tools and techniques for monitoring the integrity of the kernel. Numerous novel research challenges have also emerged that show promise towards building more robust and comprehensive kernel integrity monitors. Below, we discuss some interesting directions for future work in this area.

5.2.1 Data Structure Repair

Detection of rootkits that tamper with the kernel data structures has received a lot of attention over the past five years [9, 10, 12, 15, 29]. Detection techniques are able to identify the data structures that are modified by the attack. While some work has been done in containment of ongoing attacks [28] and offline recovery of select data structures, such as the system call table [42], the commonly employed approach in the face of such attacks is to format the disk and install a new operating system image. The current response procedure besides being tedious and time consuming, does not scale with the current attack growth rate.

Kernel integrity monitors such as Gibraltar discussed in Chapter 4, monitor invariants exhibited by kernel data. These are used as integrity specifications and are checked during runtime. The monitor can therefore, identify the data structure and the invariant that is violated when an alert is raised by the system. In such cases, repair of the data structure comprises of restoring the invariant that is violated. For example, if a data structure exhibits the constancy invariant, then a violation occurs when the rootkit replaces this value with a different one. The repair action comprises of restoring the old value. While restoring other more complex invariants might require sophisticated methods, we believe that data structure repair is a promising research direction.

To secure the monitor, current approaches isolate it from the system that it monitors [9, 10, 12, 15, 29]. As a result, the monitor is limited to external asynchronous memory based scans. It is unable to acquire locks from the operating system that is concurrently executing and modifying the data structures that are monitored. Repairing data structures requires the monitor to be able to make modifications to kernel data structures without affecting the correctness of kernel code. This also requires the invention of better mechanisms for realizing inline data structure repairs.

5.2.2 Mining Complex Invariants

Complex invariants that express conjunction or disjunction between simple invariants discussed in this dissertation might express interesting properties. It is also possible to mine more complex invariants that express relationships between different data structure fields or between different data structures altogether. Invariants might also be mined using more complex invariant templates. Verifying a large number of invariants has performance implications for the monitor. Therefore a careful study of the kind of invariants that are more likely to be violated by attacks will provide some insight into the type of invariants that are more interesting than others.

5.2.3 Transient Kernel Attacks

The advancement in techniques to detect attacks that modify persistent control flow data or persistent non-control data, are most likely to lead attackers to explore transient ways of exploiting the kernel. Some attacks have been demonstrated to this effect already [69]. Transient kernel attacks target data structures that are mutable and are modified on a regular basis by authentic kernel code. No detection technique currently exists to identify such attacks. Verification of the integrity of transient kernel attacks requires innovation in terms of techniques to be used for detection as well as mechanisms.

5.2.4 Stealth Kernel Attacks on Mobile Devices

Mobile devices such as smart phones, which run full-fledged operating systems are already being used by millions of users. Smart phones, especially the ones equipped with VoIP features, have been identified to be susceptible to hundreds of viruses. Attacks carried out on these devices, such as voice spam, smishing and denial of service attacks, jeopardize not just the user but the whole communication infrastructure. In one example of a DoS attack, a hacker could program 50 million mobile and/or VoIP phones to call 911 simultaneously in order to disable the Enhanced 911 system.

An emerging threat that researchers have completely ignored so far is the threat posed by rootkits to the phone operating system kernel. An attacker who has complete control of the phone kernel can launch particularly stealthy attacks. Since users rely on the correct operation of these devices, stealth attacks on them that are executed without the user's knowledge, can cause severe hardship and financial loss. For example, attacks might stealthily record voice conversations, send expensive messages without the user's consent, spoof location information to cause inconvenience or steal usernames and passwords. Currently, this area is largely uncharted.

References

- [1] Rootkits, part 1 of 3: A growing threat, April 2006. MacAfee AVERT Labs Whitepaper.
- [2] Anti rootkit software, news, articles and forums. <http://antirootkit.com/>.
- [3] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. Cloaker: Hardware supported rootkit concealment. In *SP '08: Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.
- [4] Samuel King, Peter Chen, Yi-Min Wang, Chad Verblowski, Helen J. Wang, and Jacob R. Lorch. Subvirt: Implementing malware with virtual machines. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006.
- [5] Joanna Rutkowska. The blue pill. <http://bluepillproject.org/>.
- [6] Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *CCS '94: Proceedings of the 2nd ACM Conference on Computer and Communications Security*, Fairfax, VA, November 1994.
- [7] Advanced intrusion detection environment. <http://sourceforge.net/projects/aide>.
- [8] Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ronald Perez, and Reiner Sailer. Secure coprocessor-based intrusion detection. In *EW '02: Proceedings of the 10th ACM SIGOPS European Workshop: Beyond the PC*, Saint-Emilion, France, July 2002.
- [9] Nick L. Petroni Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Security '04: Proceedings of the USENIX Security Symposium*, San Diego, CA, August 2004.
- [10] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS '03: Proceedings of the 10th Network and Distributed Systems Security Symposium*, San Diego, CA, February 2003.
- [11] Bryan D. Payne, Martim Carbone, Monirul I. Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.
- [12] 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*, Alexandria, VA, October 2007.
- [13] Fu rootkit. <http://www.rootkit.com/project.php?id=12>.

- [14] Arati Baliga, Pandurang Kamat, and Liviu Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2007.
- [15] Jr. Nick L. Petroni, Timothy Fraser, Aaron Walters, and William A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Security '06: Proceedings of the 15th USENIX Security Symposium*, Vancouver, Canada, July 2006.
- [16] Yi-Min Wang, Roussi Roussev, Chad Verbowski, Aaron Johnson, Ming-Wei Wu, Yennun Huang, and Sy-Yen Kuo. Gatekeeper: Monitoring auto-start extensibility points (aseps) for spyware management. In *LISA '04: Proceedings of the 18th USENIX Conference on System Administration*, Atlanta, GA, November 2004.
- [17] Doug Beck, Binh Vo, and Chad Verbowski. Detecting stealth software with strider ghostbuster. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, Yokohoma, Japan, June 2005.
- [18] Chkrootkit - rootkit detection tool. <http://www.chkrootkit.org/>.
- [19] Rootkit hunter - rootkit detection tool. <http://www.rootkit.nl/>.
- [20] System virginity verifier. <http://www.antirootkit.com/software/System-Virginity-Verifier.htm>.
- [21] F-secure blacklight. <http://www.f-secure.com/blacklight/>.
- [22] Klister rootkit detector. <http://www.rootkit.com/project.php?id=14>.
- [23] James Butler. Vice rootkit detector. <http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-butler/bh-us-04-butler.pdf>.
- [24] Patch finder. <http://www.rootkit.com/project.php?id=15>.
- [25] The st. jude intrusion detection system. <http://freshmeat.net/projects/stjude/>.
- [26] The linux intrusion detection system. <http://www.lids.org/>.
- [27] Sherri Sparks and Jamie Butler. Shadow walker. <http://www.phrack.org/issues.html?id=8&issue=63>.
- [28] Arati Baliga, Liviu Iftode, and Xiaoxin Chen. Automated containment of rootkit attacks. *Elsevier Journal on Computers and Security*, 27:323–334, August 2008.
- [29] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Automatic inference and enforcement of kernel data structure invariants. In *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference*, Anaheim, CA, December 2007.
- [30] Shellcode Security Research Team. Registration weakness in linux kernel's binary formats. <http://goodfellas.shellcode.com.ar/own/binfmt-en.pdf>, September 2006.
- [31] Amd pacifica virtualization technology. <http://enterprise.amd.com/Downloads/Pacifica.en.pdf>.
- [32] Lionel Litty and David Lie. Manitou: a layer-below approach to fighting malware. In *ASID*, San Jose, CA, October 2006.

- [33] Jeffrey Wilhelm and Tzi cker Chiueh. A forced sampled execution approach to kernel rootkit identification. In *RAID '07: Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection*, Queensland, Australia, September 2007.
- [34] 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*, Anaheim, CA, December 2004.
- [35] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Swatt: Software-based attestation for embedded devices. In *SP '04: Proceedings of the 2004 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004.
- [36] Rick Kennell and Leah H. Jamieson. Establishing the genuinity of remote computer systems. Washington, DC, August 2003. *Security '03: Proceedings of the 12th USENIX Security Symposium*.
- [37] Elaine Shi, Adrian Perrig, and Leendert van Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2005.
- [38] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep K. Khosla. Pioneer: Verifying vode integrity and enforcing untampered code execution on legacy systems. In *SOSP '05: Proceedings of the 20th ACM Symposium on Operating System Principles*, Brighton, United Kingdom, October 2005.
- [39] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a tcb-based integrity measurement architecture. In *Security '04: Proceedings of the 2004 USENIX Security Symposium*, San Diego, CA, August 2004.
- [40] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating System Principles*, Bolton Landing, New York, October 2003.
- [41] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert van Doorn. Attestation-based policy enforcement for remote access. In *Proceedings of the ACM Conference on Computer and Communications Security*, Alexandria, VA, October 2004.
- [42] Julian B. Grizzard, John G. Levine, and Henry L. Owen. Re-establishing trust in compromised systems: Recovering from rootkits that trojan the system call table. In *ESORICS '04: Proceedings of the 9th European Symposium On Research in Computer Security*, French Riviera, France, September 2004.
- [43] Intel virtualization technology specification for the ia-32 intel architecture. <ftp://download.intel.com/technology/computing/vptech/C97063-002.pdf>.
- [44] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Operating System Review*, 36(SI):181–194, 2002.
- [45] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, New York, October 2003.

- [46] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *UTC '01: Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.
- [47] Lion worm attack advisory. <http://www.ciac.org/ciac/bulletins/l-064.shtml>.
- [48] Bind tsig vulnerability. <http://www.sans.org/resources/idfaq/tsig.php>.
- [49] Dan Ellis. Worm anatomy and model. In *WORM '03: Proceedings of the 2003 ACM Workshop on Rapid Malcode*, Washington, DC, October 2003.
- [50] Xin Zhao, Kevin Borders, and Atul Prakash. Towards protecting sensitive files in a compromised system. In *Proceedings of the IEEE Security in Storage Workshop*, San Fransisco, CA, December 2005.
- [51] H. Wang, D. Zhang, and K. Shin. Detecting syn flooding attacks. New York, NY, June 2002.
- [52] Christoph L. Schuba, Ivan V. Krsul, Markus G. Kuhn, Eugene H. spafford, Aurobindo Sundaram, and Diego Zamboni. Analysis of a denial of service attack on tcp. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997.
- [53] David Moore, Colleen Shannon, Douglas J. Brown, Geoffrey M. Voelker, and Stefan Savage. Inferring internet denial-of-service activity. *ACM Transactions on Computer Systems*, 24(2):115–139, 2006.
- [54] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the linux random number generator. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006.
- [55] Adi Shamir and Nicko van Someren. Playing "hide and seek" with stored keys. In *FC '99: Proceedings of the Third International Conference on Financial Cryptography*, Anguilla, British West Indies, February 1999.
- [56] G Marsaglia. The marsaglia random number cdrom including the diehard battery of tests of randomness. <http://stat.fsu.edu/pub/diehard>.
- [57] Zhi Wang, Xuxian Jiang, Weidong Cui, and Xinyuan Wang. Raid '08: Countering persistent kernel rootkits through systematic hook discovery. In *RAID*, Cambridge, MA, September 2008.
- [58] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.
- [59] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida, May 2002.
- [60] Myricom: Pioneering high performance computing. <http://www.myri.com>.
- [61] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, Grenoble, France, April 2002.

- [62] Jeffrey Brian Arnold. Ksplice: An automatic system for rebootless linux kernel security updates. <http://web.mit.edu/ksplice/doc/ksplice.pdf>.
- [63] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *VEE '06: Proceedings of the 2nd International Conference on Virtual Execution Environments*, Ottawa, Canada, 2006.
- [64] Packet storm. <http://packetstormsecurity.org/UNIX/penetration/rootkits/>.
- [65] Larry McVoy and Carl Staelin. Lmbench: portable tools for performance analysis. In *UTC '96: Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, January 1996.
- [66] W. Norcott. Iozone benchmark. <http://www.iozone.org>, 2001.
- [67] Stefan Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security*, 3(3):186–205, 2000.
- [68] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. In *IEEE Technical Committee on Computer Architecture newsletter*, December 1995.
- [69] Jinpeng Wei, Bryan D. Payne, Jonathon Giffin, and Calton Pu. Soft-timer driven transient kernel control flow attacks and defense. In *ACSAC '08: Proceedings of the 24th Annual Computer Security Applications Conference*, Anaheim, CA, December 2008.