

Storage-Based Intrusion Detection

ADAM G. PENNINGTON, JOHN LINWOOD GRIFFIN, JOHN S. BUCY,
JOHN D. STRUNK, and GREGORY R. GANGER
Carnegie Mellon University

30

Storage-based intrusion detection consists of storage systems watching for and identifying data access patterns characteristic of system intrusions. Storage systems can spot several common intruder actions, such as adding backdoors, inserting Trojan horses, and tampering with audit logs. For example, examination of 18 real intrusion tools reveals that most (15) can be detected based on their changes to stored files. Further, an Intrusion Detection System (IDS) embedded in a storage device continues to operate even after client operating systems are compromised. We describe and evaluate a prototype storage IDS, built into a disk emulator, to demonstrate both feasibility and efficiency of storage-based intrusion detection. In particular, both the performance overhead (< 1%) and memory required (1.62MB for 13995 rules) are minimal.

Categories and Subject Descriptors: K.6.5 [Management of Computing and Information Systems]: Security and Protection—*Invasive software (e.g., viruses, worms, Trojan horses); unauthorized access (e.g. hacking, phreaking)*

General Terms: Security

Additional Key Words and Phrases: Storage, intrusion detection

ACM Reference Format:

Pennington, A. G., Griffin, J. L., Bucy, J. S., Strunk, J. D., and Ganger, G. R. 2010. Storage-based intrusion detection. *ACM Trans. Inf. Syst. Secur.* 13, 4, Article 30 (December 2010), 27 pages. DOI = 10.1145/1880022.1880024. <http://doi.acm.org/10.1145/1880022.1880024>.

1. INTRODUCTION

Many Intrusion Detection Systems (IDSs) have been developed over the years [Axelsson 1998; Lunt and Jagannathan 1988; Porras and Neumann 1997], with most falling into one of two categories: network-based or

This material is based on research sponsored in part by the Air Force Research Laboratory, under agreement number F49620-01-1-0433, and by the Army Research Office, under agreement number DAAD19-02-1-0389, APC, IBM, Intel, NetApp, and Seagate provided hardware grants which supported our research.

Authors' address: A. G. Pennington (corresponding author), J. L. Griffin, J. S. Bucy, J. D. Strunk, and G. R. Ganger, School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213; email: adamp@andrew.cmu.edu.

Permission to make digital or hard copies part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org. © 2010 ACM 1094-9224/2010/12-ART30 \$10.00 DOI: 10.1145/1880022.1880024. <http://doi.acm.org/10.1145/1880022.1880024>.

ACM Transactions on Information and System Security, Vol. 13, No. 4, Article 30, Pub. date: December 2010.

host-based. Network IDSs (NIDSs) are usually embedded in sniffers or firewalls, scanning traffic to, from, and within a network environment for attack signatures and suspicious traffic [Cheswick and Bellovin 1994; NFR 2002]. Host-based IDSs (HIDSs) are fully or partially embedded within each host's Operating System (OS). They examine local information (such as system calls [Forrest et al. 1996]) for signs of intrusion or suspicious behavior. Many environments employ multiple IDSs, each watching activity from its own vantage point.

The storage system is another interesting vantage point for intrusion detection. Several common intruder actions [Denning 1999, p. 218; Scambray et al. 2001, pp. 363–365] are quite visible at the storage interface. Examples include manipulating system utilities (e.g., to add backdoors or Trojan horses), tampering with audit log contents (e.g., to eliminate evidence), and resetting attributes (e.g., to hide changes). By design, a storage server sees all changes to persistent data, allowing it to transparently watch for suspicious changes and issue alerts about the corresponding client systems. Also, like a NIDS, a storage IDS can be compromise-independent of the host OS, meaning that it cannot be disabled by an intruder who only successfully gets past a host's OS-level protection.

This article motivates and describes storage-based intrusion detection. It presents several kinds of suspicious behavior that can be spotted by a storage IDS. Using sixteen “rootkits” and two worms as examples, we describe how fifteen of them would be exposed rapidly by our storage IDS. (The other three do not modify any stored files.) Most of them are exposed by modifying system binaries, adding files to system directories, scrubbing the audit log, or using suspicious file names. Of the fifteen detected, three modify the kernel to hide their presence from host-based detection, including FS integrity checkers like Tripwire [Kim and Spafford 1994]. In general, intruders cannot hide their changes from the storage device if they wish to persist across reboots; to be reinstalled upon reboot, their tools must manipulate stored files.

A storage IDS could be embedded in many kinds of storage systems. The extra processing power and memory space required should be feasible for file servers, disk array controllers, and even augmented disk drives. Most detection rules will also require FS-level understanding of the stored data. Such understanding exists trivially for a file server and may be explicitly provided to block-based storage devices. This understanding of a file system is analogous to the understanding of application protocols used by a NIDS [Paxson 1998], but with fewer varieties and structural changes over time.

As a concrete example with which to experiment, we have augmented a disk emulator with a storage IDS that supports online, rule-based detection of suspicious modifications. This Intrusion Detection on Disk (IDD) prototype supports the detection of two main types of suspicious activities. First, it can detect unexpected changes to important system files and binaries, using a rule-set similar to Tripwire's. Second, it can detect patterns of changes like nonappend modification (e.g., of system log files) and reversing of inode times. An administrative interface supplies the detection rules, which are checked during the processing of each block request. When a detection rule triggers, the IDD sends

the administrator an alert containing the full pathname of the modified file and the violated rule(s). Experiments show that the runtime overheads of such intrusion detection is minimal. Further analysis indicates that little memory capacity is needed for reasonable rule sets (e.g., only 1.62MB for a realistic example containing 13995 rules).

The remainder of this article is organized as follows. Section 2 introduces storage-based intrusion detection. Section 3 evaluates the potential of storage-based intrusion detection by examining real intrusion tools. Section 4 discusses storage IDS design issues. Section 5 describes a prototype storage IDS embedded in a disk emulator. Section 6 uses this prototype to evaluate the costs of storage-based intrusion detection. Section 7 discusses related work. Section 8 summarizes this article's contributions.

2. STORAGE-BASED INTRUSION DETECTION

Storage-based intrusion detection augments storage devices to examine the requests they service for suspicious client behavior. Although the computer system state that a storage device sees is incomplete, two features combine to make it a well-positioned platform for enhancing intrusion detection efforts. First, since storage devices are independent of host OSes, they can continue to look for intrusions after the initial compromise, whereas a host-based IDS can be disabled by the intruder. Second, since most computer systems rely heavily on persistent storage for their operation, many intruder actions will cause storage activity that can be captured and analyzed. This section expands on these two features and identifies limitations of storage-based intrusion detection.

2.1 Threat Model and Assumptions

Storage IDSs focus on the threat of an attacker who has compromised a host system in a managed computing environment. By “compromised,” we mean that the attacker subverted the host's software system, gaining the ability to run arbitrary software on the host with OS-level privileges. The compromise might have been achieved via technical means (e.g., exploiting buggy software or a loose policy) or nontechnical means (e.g., social engineering or bribery). Once the compromise occurs, most administrators wish to detect the intrusion as quickly as possible and terminate it. Intruders, on the other hand, often wish to hide their presence and retain access to the machine.

Unfortunately, once an intruder compromises a machine, intrusion detection with conventional schemes becomes much less effective. Host-based IDSs can be rendered ineffective by intruder software that disables them or feeds them misinformation, as many such tools do. Network IDSs can continue to look for suspicious behavior, but are much less likely to find an already successful intruder; most NIDSs look for attacks and intrusion attempts rather than for system usage by an existing intruder [Ganger et al. 2003]. A storage IDS can help by offering a vantage point on a system component that is often manipulated in suspicious ways *after* the intruder compromises the system.

A key characteristic of the described threat model is that the attacker has software control over the host, but does not have physical access to its hardware. We are not specifically trying to address insider attacks, in which the intruder would also have physical access to the hardware and its storage components. Also, for the storage IDS to be effective, we assume that neither the storage device nor the admin console are compromised.

2.2 Compromise Independence

A storage IDS will continue watching for suspicious activity even when clients' OSes are compromised. It capitalizes on the fact that storage devices (whether file servers, disk array controllers, or even IDE disks) run different software on separate hardware, as illustrated in Figure 1. This fact enables server-embedded security functionality that cannot be disabled by any software running on client systems (including the OS kernel). Further, storage devices often have fewer network interfaces (e.g., RPC+SNMP+HTTP or just SCSI) and no local users. Thus, compromising a storage server should be more difficult than compromising a client system. Of course, such servers have a limited view of system activity, so they cannot distinguish legitimate users from clever impostors. But, from behind the physical storage interface, a storage IDS can spot many common intruder actions and alert administrators.

Administrators must be able to communicate with the storage IDS, both to configure it and to receive alerts. This administrative channel must also be compromise-independent of client systems, meaning that no user (including "root") and no software (including the OS kernel) on a client system can have administrative privileges for the storage IDS. Section 4 discusses deployment options for the administrative console, including physical consoles and cryptographic channels from a dedicated administrative system.

All of the warning signs discussed in this article could also be spotted from within an HIDS, but host-based IDSs do not enjoy the compromise independence of storage IDSs. A host-based IDS is vulnerable to being disabled or bypassed by intruders that compromise the OS kernel. Another interesting place for a storage IDS is the virtual disk module of a virtual machine monitor [Sugerman et al. 2001]; such deployment would enjoy compromise independence from the OSes running in its virtual machines [Chen and Noble 2001].

2.3 Warning Signs for Storage IDSs

Successful intruders often modify stored data. For instance, they may overwrite system utilities to hide their presence, install Trojan horse daemons to allow for reentry, add users, modify startup scripts to reinstall kernel modifications upon reboot, remove evidence from the audit log, or store illicit materials. These modifications are visible to the storage system when they are made persistent. This section describes four categories of warning signs that a storage IDS can monitor: data and attribute modifications, update patterns, content integrity, and suspicious content.

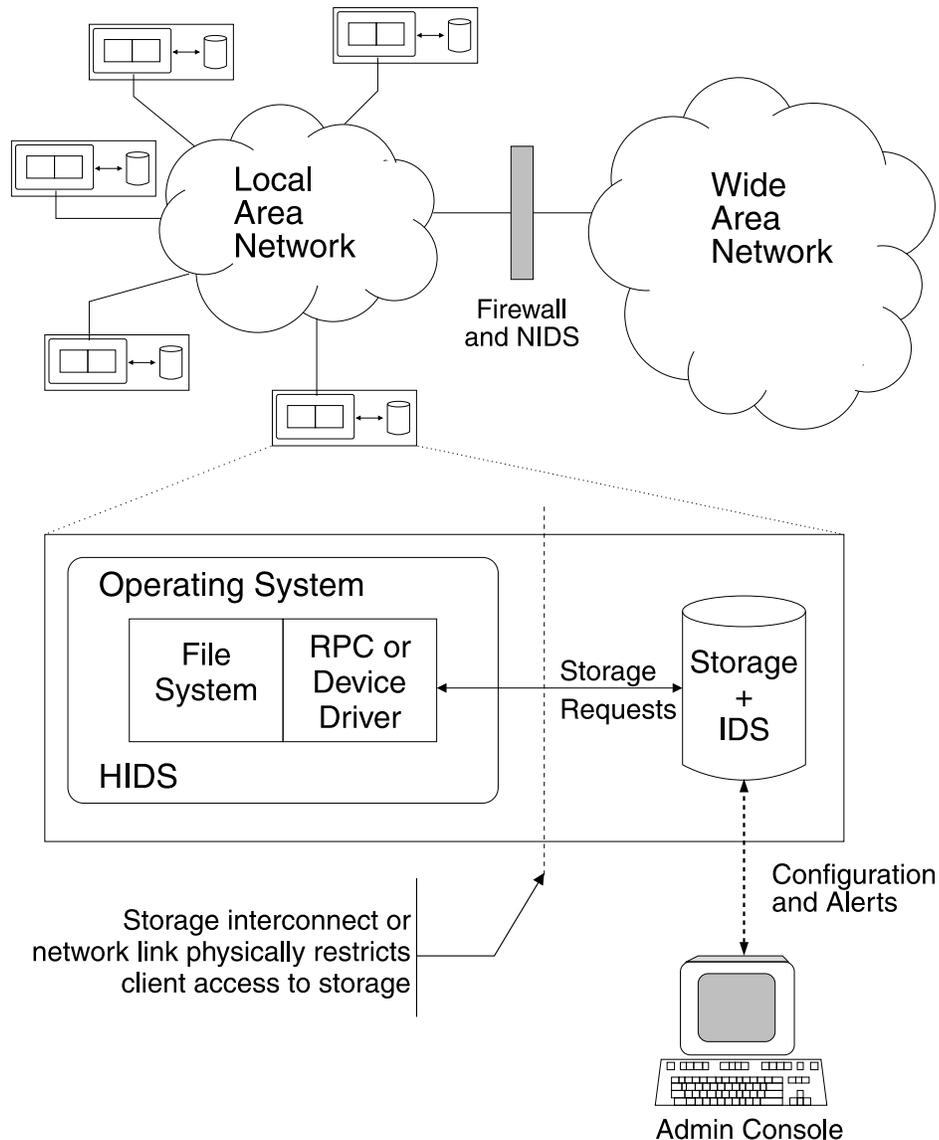


Fig. 1. The compromise independence of a storage IDS. The storage interface provides a physical boundary behind which a storage server can observe the requests it is asked to service. Note that this same picture works for block protocols, such as SCSI or IDE/ATA, and distributed file system protocols, such as NFS or CIFS. Also note that storage IDSs do not replace existing IDSs, but do offer an additional vantage point from which to detect intrusions.

2.3.1 Data /Attribute Modification. In managed computing environments, the simplest (and perhaps most effective) category of warning signs consists of data or metadata changes to files that administrators expect to remain unchanged except during explicit upgrades. Examples of such files include system executables and scripts, configuration files, and system header files and

libraries. Given the importance of such files and the infrequency of updates to them, any modification is a potential sign of intrusion. A storage IDS can detect all such modifications on-the-fly, before the storage device processes each request, and issue an alert immediately.

In current systems, modification detection is sometimes provided by a checksumming utility (e.g., Tripwire [Kim and Spafford 1994]) that periodically compares the current storage state against a reference database stored elsewhere. Storage-based intrusion detection improves on this current approach in three ways: (1) it allows immediate detection of changes to watched files; (2) it can notice short-term changes, made and then undone, which would not be noticed by a checksumming utility if the changes occurred between two periodic checks; and (3) for local storage, it avoids trusting the host OS to perform the checks, which many rootkits disable or bypass.

2.3.2 Update Patterns. A second category of warning signs consists of suspicious access patterns, particularly updates. There are several concrete examples for which storage IDSs can usefully watch. The clearest is client system audit logs; these audit logs are critical to both intrusion detection [Denning 1987] and diagnosis [Schneier and Kelsey 1999], leading many intruders to scrub evidence from them as a precaution. Any such manipulation will be obvious to a storage IDS that understands the well-defined update pattern of the specific audit log. For instance, audit log files are usually append-only, and they may be periodically “rotated.” Such rotation consists of renaming the current log file to an alternate name (e.g., logfile to logfile.0) and creating a new “current” log file. Any deviation in the update pattern of the current log file or any modification of a previous log file is suspicious.

Another suspicious update pattern is timestamp reversal. Specifically, the data modification and attribute change times commonly kept for each file can be quite useful for postintrusion diagnosis of which files were manipulated [Farmer 2000]. By manipulating the times stored in inodes (e.g., setting them back to their original values), an intruder can inhibit such diagnosis. Of course, care must be taken with IDS rules, since some programs (e.g., tar) legitimately set these times to old values. One possibility would be to only set off an alert when the modification time is set back long after a file’s creation. This would exclude tar-style activity but would catch an intruder trying to obfuscate a modified file. Of course, the intruder could now delete the file, create a new one, set the date back, and hide from the storage IDS; a more complex rule could catch this, but such hide-and-seek escalation is the nature of intrusion detection.

Detection of storage Denial-of-Service (DoS) attacks also falls into the category of suspicious access patterns. For example, an attacker can disable specific services or entire systems by allocating all or most of the free space. A similar effect can be achieved by allocating inodes or other metadata structures. A storage IDS can watch for such exhaustion, which may be deliberate, accidental, or coincidental (e.g., a user just downloaded 1 TB of multimedia files). When the system reaches predetermined thresholds of unallocated resources and allocation rate, warning the administrator is appropriate even in

nonintrusion situations; attention is likely to be necessary soon. A storage IDS could similarly warn the administrator when storage activity exceeds a threshold for too long, which may be a DoS attack or just an indication that the server needs to be upgraded.

Although specific rules can spot expected intruder actions, more general rules may allow larger classes of suspicious activity to be noticed. For example, some attribute modifications, like enabling “set UID” bits or reducing the permissions needed for access, may indicate foul play. Additionally, many applications access storage in a regular manner. As two examples: word processors often use temporary and backup files in specific ways, and UNIX password management involves a pair of interrelated files (`/etc/passwd` and `/etc/shadow`). The corresponding access patterns seen at the storage device will be a reflection of the application’s requests. This presents an opportunity for anomaly detection based on how a given file is normally accessed. This could be done in a manner similar to learning common patterns of system calls [Forrest et al. 1996] or starting with rules regarding the expected behavior of individual applications [Ko et al. 1997]. Deviation from the expected pattern could indicate an intruder attempting to subvert the normal method of accessing a given file. Of course, the downside is an increase (likely substantial) in the number of false alarms. Our focus to date has been on explicit detection rules, but anomaly detection within storage access patterns is an interesting topic for future research.

2.3.3 Content Integrity. A third category of warning signs consists of changes that violate internal consistency rules of specific files. This category builds on the previous examples by understanding the application-specific semantics of particularly important stored data. Of course, to verify content integrity, the device must understand the format of a file. Further, while simple formats may be verified in the context of the write operation, file formats may be arbitrarily complex and verification may require access to additional data blocks (other than those currently being written). This creates a performance versus security trade-off made by deciding which files to verify and how often to verify them. In practice, there are likely to be few critical files for which content integrity verification is utilized.

As a concrete example, consider a UNIX system password file (`/etc/passwd`), which consists of a set of well-defined records. Records are delimited by a linebreak, and each record consists of seven colon-separated fields. Further, each of the fields has a specific meaning, some of which are expected to conform to rules of practice. For example, the seventh field specifies the “shell” program to be launched when a user logs in, and (in Linux) the file `/etc/shells` lists the legal options. During the “Capture the Flag” information warfare game at the 2002 DEFCON conference [Lemos 2002], one tactic used was to change the root shell on compromised systems to `/sbin/halt`; once a targeted system’s administrator noted the intrusion and attempted to become root on the machine (the common initial reaction), considerable downtime and administrative effort was needed to restore the system to operation. A storage IDS can monitor changes to `/etc/passwd` and verify that they conform to a set of basic integrity rules:

7-field records, nonempty password field, legal default shell, legal home directory, nonoverlapping user IDs, etc. The attack described before, among others, could be caught immediately.

2.3.4 Suspicious Content. A fourth category of warning signs is the appearance of suspicious content. The most obvious suspicious content is a known virus or rootkit, detectable via its signature. Several high-end storage servers (e.g., from EMC [Strom 2008] and Network Appliance [Sureshkumar 2009]) now offer support for internal virus scanning. By executing the scans within the storage server, viruses cannot disable the scanners even after infecting clients.

Two other examples of suspicious content are large numbers of “hidden” files or empty files. Hidden files have names that are not displayed by normal directory listing interfaces [Denning 1999, p. 217], and their use may indicate that an intruder is using the system as a storage repository, perhaps for illicit or pirated content. A large number of empty files or directories may indicate an attempt to exploit a race condition [Bishop and Dilger 1996; Purczynski 2002] by inducing a time-consuming directory listing, search, or removal.

2.4 Limitations, Costs, and Weaknesses

Although storage-based intrusion detection contributes to security efforts, of course it is not a silver bullet.

Like any IDS, a storage IDS will produce some false positives. With very specific rules, such as “watch these 100 files for any modification,” false positives should be infrequent; they will occur only when there are legitimate changes to a watched file, which should be easily verified if updates involve a careful procedure. The issue of false alarms grows progressively more problematic as the rules get less exact (e.g., the time reversal or resource exhaustion examples). The far end of the spectrum from specific rules is general anomaly detection.

Also like any IDS, a storage IDS will fail to spot some intrusions. Fundamentally, a storage IDS cannot notice intrusions whose actions do not cause odd storage behavior. For example, three of the eighteen intrusion tools examined in the next section manipulate the OS but change no stored files. Also, an intruder may manipulate storage in unwatched ways. Using network-based and host-based IDSs together with a storage IDS can increase the odds of spotting various forms of intrusion.

Intrusion detection, as an aspect of information warfare, is by nature a game of escalation. As soon as the defender takes away an avenue of attack, the attacker starts looking for the next. Since storage-based intrusion detection easily sees several common intruder activities, crafty intruders will change tactics. For example, an intruder can make any number of changes to the host’s memory, so long as these modifications do not propagate to storage. A reboot, however, will reset the system and remove the intrusion, which argues for proactive restart [Castro and Liskov 2000; Huang et al. 1996; Vaidyanathan et al. 2002]. To counter this, attackers must have their changes reestablished automatically after a reboot, such as by manipulating the various boot-time

Table I. Visible Actions of Several Intruder Toolkits

Name	Description	Syscall redir.	Log scrub	Hidden dirs	Watched files	Total alerts
Ramen	Linux worm			X	2	3
lion	Linux worm				10	10
FK 0.4	Linux LKM rootkit and trojan ssh	X			1	1
Taskigt	Linux LKM rootkit				1	1
SK 1.3a	Linux kernel rootkit via /dev/kmem	X				-
Darkside 0.2.3	FreeBSD LKM rootkit	X				-
Knark 0.59	Linux LKM rootkit	X		X	1	2
Adore	Linux LKM rootkit	X				-
lrk5	User level rootkit from source		X	X	20	22
Sun rootkit	SunOS rootkit with trojan rlogin				1	1
FreeBSD Rootkit 2	User level FreeBSD rootkit		X	X	15	17
t0rn	Linux user level rootkit		X	X	20	22
Advanced Rootkit	Linux user level rootkit			X	10	11
ASMD	Rootkit w/SUID binary trojan			X	1	2
Dica	Linux user level rootkit		X	X	9	11
Flea	Linux user level rootkit		X	X	20	22
Ohara	Rootkit w/PAM trojan		X	X	4	6
TK 6.66	Linux user level rootkit		X	X	10	12

For each of the tools, the table shows which of the following actions are performed: redirecting system calls, scrubbing the system log files, and creating hidden directories. It also shows how many of the files watched by our rule set are modified by a given tool. The final column shows the total number of storage IDS alerts that would be generated by that intruder tool.

(e.g., `rc.local` in UNIX-like systems) or periodic (e.g., `cron` in UNIX-like systems) programs. Doing so exposes them to the storage IDS, creating a traditional intrusion detection game of cat and mouse.

As a practical consideration, storage IDSs embedded within individual components of decentralized storage systems are unlikely to be effective. For example, a disk array controller is a fine place for storage-based intrusion detection, but individual disks behind software striping are not. Each of the disks has only part of the file system's state, making it difficult to check nontrivial rules without adding new interdevice communication paths.

Finally, storage-based intrusion detection is not free. Checking rules comes with some cost in processing and memory resources, and more rules require more resources. In configuring a storage IDS, one must balance detection efforts with performance costs for the particular operating environment.

3. CASE STUDIES

This section explores how well a storage IDS might fare in the face of actual compromises. To do so, we examined eighteen intrusion tools (Table I) designed

to be run on compromised systems. All were downloaded from public Web sites, most of them from Packet Storm [Packetstorm 2009]. While not all of the tested intrusion tools would have successfully compromised our testbed system, we still looked at how well their activity would have been detected.

Most of the actions taken by these tools fall into two categories. Actions in the first category involve hiding evidence of the intrusion and the rootkit's activity. Actions in the second category provide mechanisms for reentry into a system. Twelve of the tools operate by running various binaries on the host system and overwriting existing binaries to broaden their control. The other six insert code into the operating system kernel.

For the analysis in this section, we focus on a subset of the rules supported by our prototype storage-based IDS described in Section 5. Specifically, we include the file/directory modification (Tripwire-like) rules, the append-only log-file rule, and the hidden filename rules. We do not consider any “suspicious content” rules, which may or may not catch a rootkit depending on whether its particular signature is known.¹ In these eighteen toolkits, we did not find any instances of resource exhaustion attacks or of reverting inode times.

3.1 Detection Results

Of the eighteen toolkits tested, storage IDS rules would immediately detect fifteen based on their storage modifications. Most would trigger numerous alerts, highlighting their presence. The other three make no changes to persistent storage. Thus, they are removed if the system reboots; all three modify the kernel, but would have to be combined with system file changes to be reinserted upon reboot.

Nonappend changes to the system audit log. Seven of the eighteen toolkits scrub evidence of system compromise from the audit log. All of them do so by selectively overwriting entries related to their intrusion into the system, rather than by truncating the logfile entirely. All cause alerts to be generated in our prototype.

System file modification. Fifteen of the eighteen toolkits modify a number of watched system files (ranging from 1 to 20). Each such modification generates an alert. Although three of the rootkits replace the files with binaries that match the size and CRC checksum of the previous files, they do not foil cryptographically strong hashes. Thus, Tripwire-like systems would be able to catch them as well, though the evasion mechanism described in Section 3.2 defeats Tripwire.

Many of the files modified are common utilities for system administration, found in `/bin`, `/sbin`, and `/usr/bin` on a UNIX machine. They are modified to hide the presence and activity of the intruder. Common changes include modifying `ps` to not show an intruder's processes, `ls` to not show an intruder's files, and `netstat` to not show an intruder's open network ports and connections. Similar modifications are often made to `grep`, `find`, `du`, and `pstree`.

¹An interesting note is that rootkit developers reuse code: four of the rootkits use one audit log scrubbing program (`sauber`) and another three use a second (`zap2`).

The other common reason for modifying system binaries is to create backdoors for system reentry. Most commonly, the target is `telnetd` or `sshd`, although one rootkit added a backdoored PAM module [Samar and Schemers III 1995] as well. Methods for using the backdoor vary and do not impact our analysis.

Hidden file or directory names. Twelve of the rootkits make a hard-coded effort to hide their nonexecutable and working files (i.e., the files that are not replacing existing files). Ten of the kits use directories starting in a “.” to hide from default `ls` listings. Three of these generate alerts by trying to make a hidden directory look like the reserved “.” or “..” directories by appending one or more spaces (“.” or “..”). This also makes the path harder to type if a system administrator does not know the number of spaces.

3.2 Kernel-Inserted Evasion Techniques

Six of the eighteen toolkits modified the running operating system kernel. Five of these six “kernel rootkits” include Loadable Kernel Modules (LKMs), and the other inserts itself directly into kernel memory by use of the `/dev/kmem` interface. Most of the kernel modifications allow intruders to hide as well as reenter the system, similarly to the file modifications described earlier. Especially interesting for this analysis is the use of `exec()` redirection by four of the kernel rootkits. With such redirection, the `exec()` system call uses a replacement version of a targeted program, while other system calls return information about or data from the original. As a result, any tool relying on the accuracy of system calls to check file integrity, such as Tripwire, will be fooled.

All of these rootkits are detected using our storage IDS rules; they all put their replacement programs in the originals’ directories (which are watched), and four of the six actually move the original file to a new name and store their replacement file with the original name (which also triggers an alert). However, future rootkits could be modified to be less obvious to a storage IDS. Specifically, the original files could be left untouched and replacement files could be stored someplace not watched by the storage IDS, such as a random user directory; neither would generate an alert. With this approach, file modification can be completely hidden from a storage IDS unless the rootkit wants to reinstall the kernel modification after a reboot. To accomplish this, some original files would need to be changed, which forces intruders to make an interesting choice: hide from the storage IDS or persist beyond the next reboot.

3.3 Anecdotal Experience

During this research, one of the authors was asked to analyze a system that had been recently compromised. Several modifications similar to those made by the preceding rootkits were found on the system. Root’s `.bash_profile` was modified to run the zap2 log scrubber, so that as soon as root logged into the system to investigate the intrusion, the related logs would be scrubbed. Several binaries were modified (`ps`, `top`, `netstat`, `pstree`, `sshd`, and `telnetd`). The binaries were set up to hide the existence of an IRC bot, running out of the “hidden” directory `‘/dev/.. /’`. This experience helps validate our choice of

“rootkits” for study, as they appear to be representative of at least one real-world intrusion. This intrusion would have triggered at least 8 storage IDS rules.

4. DESIGN OF A STORAGE IDS

To be useful in practice, a storage IDS must simultaneously achieve several goals. It must support a useful set of detection rules, while also being easy for human administrators to understand and configure. It must be efficient, minimizing both added delay and added resource requirements; some user communities still accept security measures only when they are free. Additionally, it should be invisible to users at least until an intrusion detection rule is matched.

This section describes four aspects of storage IDS design: specifying detection rules, administering a storage IDS securely, checking detection rules, and responding to suspicious activity.

4.1 Specifying Detection Rules

Specifying rules for an IDS is a tedious, error-prone activity. The tools an administrator uses to write and manipulate these rules should be as simple and straightforward as possible. Each category of suspicious activity presented earlier will likely need a unique format for rule specification.

The rule format used by Tripwire seems to work well for specifying rules concerned with data and attribute modification. This format allows an administrator to specify the pathname of a file and a list of properties that should be monitored for that file. The set of watchable properties are codified, and they include most file attributes. This rule language works well, because it allows the administrator to manipulate a well-understood representation (pathnames and files), and the list of attributes that can be watched is small and well-defined.

The methods used by virus scanners work well for configuring an IDS to look for suspicious content. Rules can be specified as signatures that are compared against files’ contents. Similarly, filename expression grammars (like those provided in scripting languages) could be used to describe suspicious filenames.

4.2 Secure Administration

The security administrator must have a secure interface to the storage IDS. This interface is needed for the administrator to configure detection rules and to receive alerts. The interface must prevent client systems from forging or blocking administrative requests, since this could allow a crafty intruder to sneak around the IDS by disarming it. At a minimum, it must be tamper-evident. Otherwise, intruders could stop rule updates or prevent alerts from reaching the administrator. To maintain compromise independence, it must be the case that obtaining “superuser” or even kernel privileges on a client system is insufficient to gain administrative access to the storage device.

Two promising architectures exist for such administration: one based on physical access and one based on cryptography. For environments where the

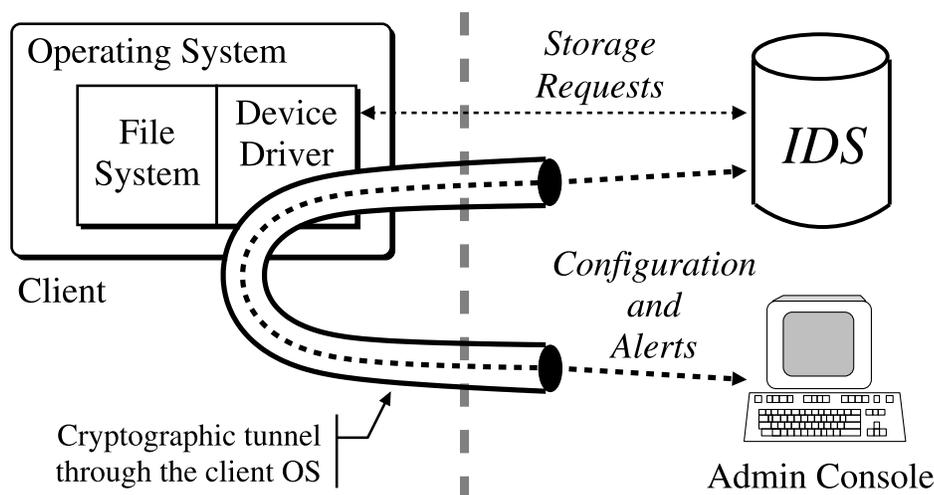


Fig. 2. Tunneling administrative commands through client systems. For storage devices attached directly to client systems, a cryptographic tunnel can allow the administrator to securely manage a storage IDS. This tunnel uses the untrusted client OS to transport administrative commands and alerts.

administrator has physical access to the device, a local administration terminal that allows the administrator to set detection rules and receive the corresponding alert messages satisfies the preceding goals.

In environments where physical access to the device is not practical, cryptography can be used to secure communications. In this scenario, the storage device acts as an endpoint for a cryptographic channel to the administrative system. The device must maintain keys and perform the necessary cryptographic functions to detect modified messages, lost messages, and blocked channels. Architectures for such trust models in storage systems exist [Gobiuff 1999]. This type of infrastructure is already common for administration of other network-attached security components, such as firewalls or network intrusion detection systems. For direct-attached storage devices, cryptographic channels can be used to tunnel administrative requests and alerts through the OS of the host system, as illustrated in Figure 2. Such tunneling simply treats the host OS as an untrusted network component. An attacker may be able to disrupt communication passing through the host OS but this will be visible to both an administrator and the storage device.

For small numbers of dedicated servers in a machine room, either approach is feasible. For large numbers of storage devices or components operating in physically distributed environments, cryptography is the only practical solution.

4.3 Checking the Detection Rules

Checking detection rules during individual storage accesses can be nontrivial, because rules generally apply to full pathnames rather than inode or block

numbers. Additional complications arise when rules can watch for files that do not yet exist.

For simple operations that act on individual files (e.g., READ and WRITE), rule verification is localized. The device need only check that the rules pertaining to that specific file are not violated. For operations that affect the file system's namespace, verification is more complicated. For example, a rename of a directory may impact a large number of individual files, any of which could have IDS rules that must be checked. Renaming a directory requires examining all files and directories that are children of the one being renamed. In addition to rules on affected files, rules pertaining to files that do not currently exist must be checked when operations change the namespace. For example, the administrator may want to watch for the existence of a file named /a/b/c even if the directory named /a does not yet exist. However, a single file system operation (e.g., `mv /z /a`) could cause the watched file to suddenly exist, given the appropriate structure for z's directory tree.

4.4 Responding to Rule Violations

Since a detected "intruder action" may actually be legitimate user activity (i.e., a false alarm), our default response is simply to send an alert to the administrative system or the designated alert log file. The alert message should contain such information as the file(s) involved, the time of the event, the action being performed, the action's attributes (e.g., the data written into the file), and the client's identity. Note that, if the rules are set properly, most false positives should be caused by legitimate updates (e.g., upgrades) from an administrator. With the right information in alerts, an administrative system that also coordinates legitimate upgrades could correlate the generated alert (which can include the new content) with the in-progress upgrade; if this were done, it could prevent the false alarm from reaching the human administrator while simultaneously verifying that the upgrade went through to persistent storage correctly.

There are more active responses that a storage IDS could trigger upon detecting suspicious activity. When choosing a response policy, of course, the administrator must weigh the benefits of an active response against the inconvenience and potential damage caused by false alarms.

One reasonable active response is to slow down the suspected intruder's storage accesses. For example, a storage device could wait until the alert is acknowledged before completing the suspicious request. It could also artificially increase request latencies for a client or user that is suspected of foul play. Doing so would provide increased time for a more thorough response, and, while it will cause some annoyance in false alarm situations, it is unlikely to cause damage. The device could even deny a request entirely if it violates one of the rules, although this response to a false alarm could cause damage and/or application failure. For some rules, like append-only audit logs, such access control may be desirable.

Liu et al. proposed a more radical response to detected intrusions: isolating intruders, via versioning, at the file system level [Liu et al. 2000]. To do so,

the file system forks the version trees to sandbox suspicious users until the administrator verifies the legitimacy of their actions. Unfortunately, such forking is likely to interfere with system operation, unless the intrusion detection mechanism yields no false alarms. Specifically, since suspected users modify different versions of files from regular users, the system faces a difficult reintegration [Kumar and Satyanarayanan 1995; Terry et al. 1995] problem, should the updates be judged legitimate. Still, it is interesting to consider embedding this approach, together with a storage IDS, into storage systems for particularly sensitive environments.

A less intrusive storage-embedded response is to start versioning all data and auditing all storage requests when an intrusion is detected. Doing so provides the administrator with significant information for postintrusion diagnosis and recovery. Of course, some intrusion-related information will likely be lost unless the intrusion is detected immediately, which is why Strunk et al. [2000] argue for always doing these things (just in case). Still, IDS-triggered employment of this functionality may be a useful trade-off point.

5. PROTOTYPE IMPLEMENTATION

This section describes the architecture and implementation of a prototype disk-based IDS called *IDD* (Intrusion Detection on Disk).

5.1 Architecture

IDD takes the form of a PC acting as a Fibre Channel SCSI disk [Griffin 2004], enhanced with storage-based intrusion detection. From the perspective of the host computer's software and hardware, IDD looks and behaves like an actual disk. Figure 3 shows the high-level interaction between a workstation or server computer ("Host") and IDD components. Most of the IDD components are located in the host's locally attached "disk" (shown as the "Disk Emulator" for our prototype system). A component not shown, the remote administrator's console, receives all alerts and would likely also serve as a central point of control for other intrusion detection systems running at other hosts.

There are two primary IDS functions in IDD. The first function, *storage traffic monitoring*, is implemented by the Policy Monitor. It maps administrative policies into violating interface actions, monitors all ordinary storage traffic for real-time violations, and generates alerts. The Policy Monitor uses the IDD Block Cache to reduce the cost of checking rules that require examination of existing disk blocks. The second function, *administrative communication*, is implemented by the Administrative Bridge process (which runs in the host being watched) and the Request Demultiplexer. The Administrative Bridge process forwards commands from the administrator to IDD and conveys alerts from IDD to the administrator. The Request Demultiplexer identifies incoming SCSI requests that contain administrative data (rather than normal disk requests) and handles the transmission of alerts.

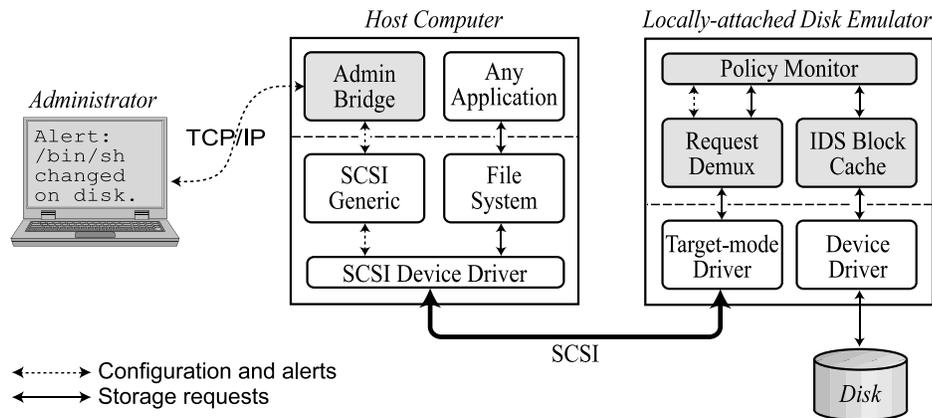


Fig. 3. Disk-based IDS prototype architecture. This figure shows the communications flow between a host computer and the locally attached IDD. The shaded boxes are key components that support disk-based IDS operation. Ordinary storage traffic is initiated by application processes, passes across the SCSI bus, is checked by the policy monitor, and is finally serviced by the disk. Administrative traffic is initiated by the administrator, passes across a TCP/IP network, is received by the bridge process on the host computer, passes across the SCSI bus, and is finally serviced by the policy monitor. The sample alert displayed on the administrator's console originated in the policy monitor.

5.2 Storage Traffic Monitoring

The policy monitor bridges the semantic gap between the administrator's policy statements and individual SCSI READ and WRITE requests, checking each request from the host computer before it is delivered to IDD's real disk for service. Administrative policies are specified in terms of files, so IDD must understand the on-disk file system structures. Such understanding is not difficult to embed, given access to the OS's definitions (e.g., in header files), and the structures change very infrequently for existing file systems [Pennington et al. 2003].

IDD currently understands the Second Extended (ext2) file system used by Linux-based host computers [Card et al. 1994]; to support this, we hard-coded the structure of on-disk metadata into the policy manager. For ext2, this includes the ext2 superblock, inode, indirect block, and directory entry structures. Such file system-specific detail, either hard-coded or module-based, is feasible under an economic model of cooperation between host OS vendors and disk manufacturers. During initial configuration, the administrator would specify the file system used by the host computer. It may also be possible to use grey-box techniques [Sivathanu et al 2003] determine the file system being used or even to deduce a file system's structures.

When administrative policy is specified, IDD receives a list of files whose contents should be watched in particular ways (e.g., for any change, for reads, or for nonappend updates). For each watched file, IDD traverses the on-disk directory structure to determine which metadata and data blocks are associated with the file. These blocks are labeled "watched blocks" and are added

to a table of watched blocks. Each such block is then associated with a number of access check functions that evaluate whether a block access violates a given rule.

The mapping from file-system-level policy to potential block access violations must be thorough. Consider the example rule *Warn me if the file /bin/netstat in partition 2 changes*. For ext2, the mapping expands to individual access check functions associated with: (1) the second entry in the disk's partition table, (2) certain fields in the ext2 superblock, (3) fields in the root inode, (4) any indirect blocks—including doubly- and triply-indirect—for the root inode's directory entries, (5) the inode number associated with `bin` in the root directory, (6) fields in the `bin` inode, (7) block pointers for `bin`'s directory entries, (8) the inode number for `netstat`, (9) fields in the `netstat` inode, (10) indirect blocks for `netstat`, and (11) each of the data blocks for `netstat`.

5.2.1 The Watched Block Table (WBT). The Watched Block Table (WBT) is the IDD's primary data structure. Each WBT entry contains a monitored block number (LBN) and a list of the associated Access Check Functions (ACF) for the block along with associated parameters. IDD's WBT is indexed by LBN via a B*-tree of contiguous block extents.

When a new storage request arrives from the host computer, IDD checks whether any of the request's blocks appear in the WBT. In the common case, no matches are found. This means there are no possible policy violations caused by the request, so no further IDS-related processing is required. (The other case is discussed in Section 5.2.2.) It is imperative that this WBT lookup be efficient, as it must be performed in the critical path for every request. The WBT lookup could be overlapped with disk positioning time, but it cannot safely proceed in parallel with the media transfer for writes. Not only might the write generate an alert or even be blocked, but IDD may need to fetch the old version of the LBN for comparison purposes.

To reduce the memory footprint used by the IDS functionality, portions of the WBT can be demand paged. The list of monitored block numbers should always remain in-core, but the remaining information (the access check function pointer and alert-time explanation) is only accessed after a monitored block is accessed. IDD keeps enough data in-core to positively determine that a given update affects a watched file without paging in any additional data other than the old data in question.

5.2.2 Access Check Functions (ACFs). If one or more of the blocks accessed by a request appear in the WBT, IDD must perform extra processing to determine whether the request actually causes a rule violation. This is necessary because multiple file system objects can appear in a single block (e.g., directory entries), only some of which may set off rules when updated. IDD executes ACFs in two steps. First, it determines whether specifically watched byte ranges within the blocks have changed by comparing the new contents with the old, which it may need to fetch from the media. If watched bytes change, the ACF is invoked.

Checking whether a write request violates rules may require that stored data be read from the disk by the ACF. We call these IOs “interposed reads.” IDD uses a simple LRU block cache to reduce the number of extra IOs caused by interposed reads. Note that this cache is designed to quickly execute ACFs on block updates that will ultimately not generate an alert, rather than satisfy host I/O requirements.

As examples from our ext2-based implementation, the ACF for a data block with the “any change” policy would simply compare the contents of the block on disk with the new block being written. The ACF for a directory entry block with the “any change” policy would check the old directory block and new directory block to determine if a particular filename-to-inode-number mapping changed (e.g., `mv file1 file2` when `file2` is watched). The ACF for an inode block with the “nonappend updates” policy would compare the old and new inode contents to ensure that the access time field and the file size field only increased. Complicating the logic for this rule, the last valid block pointer in the inode is allowed to change, but only portions of the last block of a file beyond the current file size can change.

5.3 Alert Generation and Administrative Communication

The administrative communications channel is implemented jointly by the bridge process and the request demultiplexer. The administrator sends its traffic directly to the bridge process over a TCP/IP-based network connection. The bridge process immediately repackages that traffic in the form of specially marked SCSI requests and sends those across the SCSI bus. When IDD receives these marked requests, they are identified and intercepted by the request demultiplexer. The administrative console encrypts its traffic, creating a secure channel with the administrator’s computer and the request demultiplexer as endpoints.

The bridge process assists with communication in both directions. For a message from the administrator to the IDD (e.g., sending new rules) the bridge creates a single SCSI WRITE request containing the entire message. The request is marked as containing administrative data by setting an unused flag in the SCSI command descriptor block. The request is then sent over the bus using the Linux SCSI Generic passthrough driver interface.

To poll for messages from IDD to the administrator (e.g., alerts), the bridge creates either one or two SCSI READ requests. The first request is always of fixed size (we used 8KB) and is used to determine the number of bytes of message data waiting in IDD to be fetched: The first 32 bits received from IDD indicate the integer number of pending bytes. The remaining space in the first request is filled with waiting message data. If there is more data waiting than fits in the first request, a second READ request immediately follows. This second request is of appropriate size to fetch all the remaining message data. These READ requests are marked as “administrative” by the same unused SCSI flag as described before. Once the bridge process has fetched all the waiting data, it forwards the data to the administrative console over the network.

The administrative communication channel must be reliable in the face of message duplication or omission due to network problems or malicious attack. Our implementation uses per-message sequence and acknowledgement numbers to ensure that such errors are detected. The IDD administrative process initiates one pair of messages (sending one and expecting a response) per second by default. In order to reduce administrator-perceived lag, this frequency is temporarily increased whenever recent messages contained policies or alerts.

6. PROTOTYPE EVALUATION

This section examines the performance and memory overheads incurred by IDD, our prototype disk-based IDS. We find that these overheads are not unreasonable for inclusion in workstation disks.

6.1 Experimental Setup

In our setup, both the host computer and its locally attached disk emulator are 2 GHz Pentium 4-based computers with 512MB RAM. The disk emulator runs FreeBSD 5.2 and makes use of FreeBSD's target-mode SCSI support in order to capture SCSI requests initiated by the host computer. The host computer runs Red Hat Linux 8.0. The host and emulator are connected point-to-point using QLogic QLA2100 Fibre Channel adapters. The backing store for the emulator is an 72 GB Seagate ST373405LC disk connected to an Adaptec 29160 Ultra160 SCSI controller. In an effort to exercise the worst-case storage performance, the file system stored on the disk emulator was mounted synchronously by the host computer and caching was turned off inside the backing store disk.

We do not argue that embedded disk processors will have a 2 GHz clock frequency; this is perhaps an order of magnitude larger than one might expect. However, an actual disk-based IDS would be manually tuned to the characteristics of the platform it runs on (e.g., the disk's SCSI ASICs would obviate much of the IDS communication and interposition overheads) and would therefore run more efficiently than IDD, perhaps by as much as an order of magnitude. To compensate for this uncertainty, we report processing overheads both in elapsed time and in processor cycle counts. The latter provides a reasonably portable estimate of the amount of CPU work performed by IDD.

To approximate the conditions of a production file system deployed in a real environment, we created a disk image of a freshly installed Red Hat Linux 8.0 desktop system. This image is loaded into the emulator at the start of each experiment, after which the emulated disk is mounted synchronously in a root-level directory on the host computer. For each experiment with IDD "fully enabled," we set the administrative policy to match the default Tripwire rule-set for Red Hat Linux [Tripwire 2002].

Our experiments use a microbenchmark and two macrobenchmarks. Our alert-generating microbenchmark cycles 1000 times over a single file operation on each of 1000 files (out of 1,000,000 spread over 1,000 directories), all of which will generate alerts when IDD is enabled. We use Linux kernel build and PostMark as nonalert-generating macrobenchmarks. Linux kernel build is an example of a disk-intensive workload that creates, operates on, and deletes

many temporary files. The benchmark unpacks a bzip2 compressed tarball of the Linux 2.6.9 kernel, and then builds a kernel image using the kernel's default configuration. It can be considered a successor to the Andrew Benchmark [Howard et al. 1988]. PostMark was designed to exercise the file system as would a server for the small files associated with electronic mail, netnews, or Web-based services [Katcher 1997]. Each macrobenchmark result represents the average of 10 runs of the test program on a warm system, where the system was warmed by loading the rules into IDD and running the test program once.

6.2 Base Resource Requirements

Expanding out the default Tripwire rule set for Red Hat Linux on our freshly installed disk image results in rules being set on 13995 files that IDD records in 17627 extents. This rule coverage represents watching 15% of the total number of files in the base (no user files) system.

For our IDD prototype, the fully enabled rule set results in a baseline memory footprint of 1.62MB. This includes 36KB for the IDD executable image size and 1.57MB for internal structures that track rules and the lists of watched blocks. This works out to approximately 120B per file watched. In addition to this footprint, IDD can have a cache to temporarily hold rule structures in memory as well as recently checked blocks on disk.

When the rule-set is first loaded, IDD makes an initial pass through the quiescent file system to initialize the lists of watched blocks and other structures. In our implementation, this time is dominated by the number of disk I/Os required to load the relevant inode and directory structures and to save a copy of the rule data structure to disk. By caching certain frequently used blocks (such as the superblock, root inode, and root directory blocks, as well as recently touched inodes and directory entries), this takes a total of 529 seconds, or about 37ms per rule. This initialization isn't necessary every time the disk is powered up; once the rule-set is loaded, IDD's internal state can be saved to disk media and restored quickly when restarted. It may be desirable for a deployed disk-based IDS to periodically repeat this process in the background, just to verify the consistency of its internal state.

6.3 Common-Case Performance

As discussed in Section 5.2.1, WBT lookup must be efficient, since it is performed in the critical path of every request. We examine the impact of the common-case WBT lookup both in the aggregate using the macrobenchmarks and at the individual request level using the microbenchmark.

The macrobenchmark results are shown in Figure 4. Linux kernel build generates 241,718 disk requests in 1590 seconds, and PostMark generates 364310 requests in 1440 seconds. These graphs show both that the overhead of the IDS infrastructure itself can be small (as shown by case (b), 0.01–0.1% for our implementation) and that the WBT lookup time is insignificant (case (c), 0.02–1.3%) relative to the disk request times. Our microbenchmark results, shown in Table II, show that the overhead for a WBT lookup is 1–4 μ s. These results

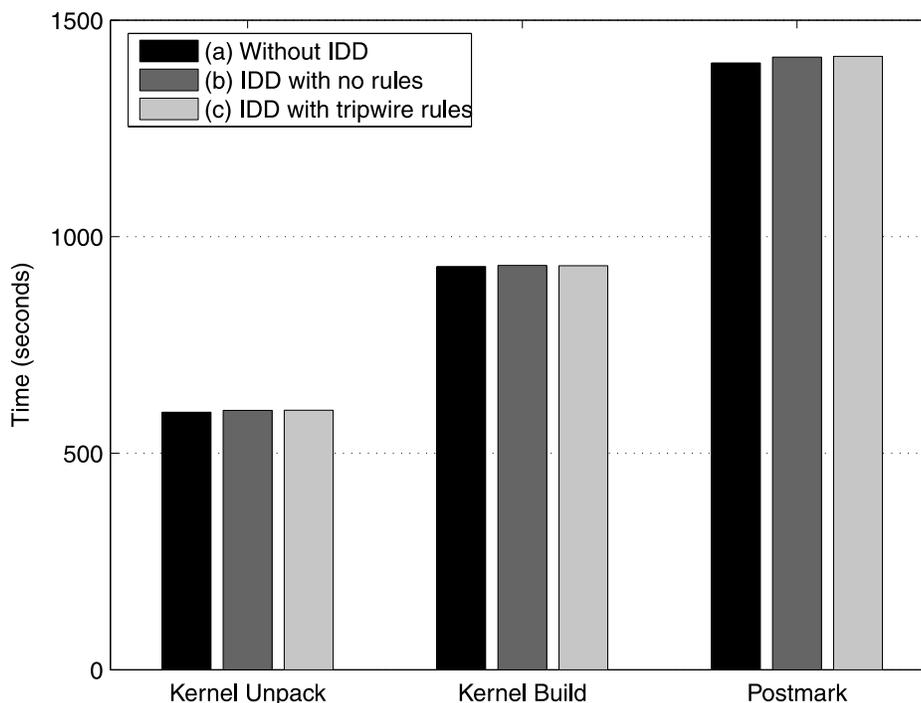


Fig. 4. Application benchmarks. These graphs show the impact of the initial WBT check on application performance. The “without IDD” bars represent using our IDD disk simulator with all IDS functionality bypassed; the remaining experiments were run with the IDS engaged and watching different amounts of data. None of the benchmark’s disk accesses commits policy violations, but IDD must ensure this for the IDD no rules and IDD Tripwire rules cases. The data indicate virtually no scaling of the overhead as a function of the amount of watched data.

show that it is indeed possible to do the requisite common-case IDS processing with no discernible effect on application performance.

6.4 Updates to Watched Blocks

If one or more of the blocks of a request appear in the WBT, IDD must perform extra work to determine if the request actually causes a rule violation. Unlike the common-case analysis given earlier, it is not paramount that these checks have no performance impact on the host’s request stream; since an update to a watched block may indicate an intruder action, it is imperative for the disk to determine whether the associated rules have been violated. We examine the impact of watched block checking using the alert-generating microbenchmark. The results are shown in Table II.

This analysis assumes that the entire WBT is kept in-core but that no data blocks are cached by the IDD block cache. Depending on the system state, the actual results could be worse or better. If the WBT were fully paged to disk, the request time would increase by approximately 30% as an additional interposed request would occur to page in the appropriate entry in the WBT.

Table II. Alert-Generating Microbenchmark

Experiment	Cache Hit			Cache Miss		
	data	inode	same-block	data	inode	same-block
WBT Lookup (ms)	0.00321 (0.000332)	0.00172 (0.000211)	0.00151 (0.000227)	0.00310 (0.000461)	0.00343 (0.000424)	0.00407 (0.000548)
clock cycles	6396	3428	3002	6190	6840	8120
Read Old Data	0.244 (0.0105)	0.121 (0.0141)	0.176 (0.0266)	5.33 (1.51)	5.28 (1.47)	5.24 (1.48)
Page rule data	0.111 (0.00407)	0.104 (0.00342)	n/a	10.2 (1.39)	10.2 (1.39)	n/a
ACF check	0.0625 (0.00670)	0.00653 (0.00266)	n/a	0.0365 (0.00188)	0.0101 (0.00738)	n/a
clock cycles	125000	13000		72800	20100	
Final Write	5.86 (1.42)	5.77 (1.56)	5.85 (1.47)	3.83 (1.43)	6.72 (1.37)	3.86 (0.0621)
Total	6.11	5.89	6.02	19.4	22.2	9.11

This table decomposes the service time of write requests in our alert-generating microbenchmark. Mean times are in milliseconds with standard deviation given in parentheses. “data” overwrites a single data block. “inode” changes an inode parameter (mtime). “same-block” changes an inode not being watched in the same inode block as another inode which is watched. The “cache hit” and “cache miss” cases represent whether all interposed blocks were already cached in the IDD Block Cache.

Conversely, if all the relevant data blocks were cached (requiring 512KB for this experiment), the interposed requests would be unnecessary, reducing the overhead for updates to watched blocks to 6–9%.

For some updates that generate an alert, it may be necessary to modify the internal IDD structures to reflect the update. For example, given the rule *Warn me if anything changes in the directory /sbin*, if the file */sbin/newfile* is created, it is perhaps appropriate to both generate an alert and start watching the new file for subsequent changes. This can either be done by reconstructing the exact change caused by the alert (which may require additional interposed reads), or by reinvoking the full initialization process.

These results show that the overhead involved with determining whether to generate alerts during updates to watched blocks is not unreasonable, especially if it is assumed that such updates occur infrequently. In the following two subsections, we analyze desktop traces from a university laboratory environment to give some insight into the validity of this assumption.

6.4.1 Frequency of Alert-Generating Updates. To understand the frequency of overheads beyond the common-case performance, we examined 11 months worth of local file system traces from managed desktop machines in a mid-sized research group. The traces included 820,145,133 file operations, of which 1.8% translated into modifications to the disk. Using these traces, we quantify the frequency of two cases: actual rule violations and nonrule-violating updates to watched blocks (specifically, incidental updates to shared inode blocks or directory blocks).

We examine the traces for violations of the Tripwire’s default rule set for Red Hat Linux. This expanded out to rules watching 29,308 files, which in turn translates to approximately 225,000 blocks being watched. In the traces, 5350 operations (0.0007% of the total) impacted files with rules set on them. All but 10 of these were the result of nightly updates to configuration files such as */etc/passwd* and regular updates to system binaries. If the IDD is made aware of expected updates (perhaps by receiving the expected change first over

the administrative channel) the IDD could convert its response from a false positive to a helpful confirmation. As there were no known intrusions on any of the traced systems, the remaining 10 alerts appear to be file changes by users.

6.4.2 Frequency of Nonalert-Generating Updates. The first class of nonrule-violating updates that require ACF execution is shared inode blocks. In the case of the ext2 file system [Card et al. 1994], 32 inodes are stored in each inode block. Our prototype notices any changes to an inode block containing a watched inode, so it must also determine if any such modification impacts an inode being watched. If any inode in a given block is watched, an update to one of the 31 remaining inodes will incur some additional overhead. To quantify this effect, we looked at the number of changes to inodes that were in the same block as a watched inode. For this analysis, the local file systems of 15 computers were used as examples of which inodes share blocks with watched files. In our traces, 1.9% of writes resulted in changes to inode blocks. Of these, 8.1% update inode blocks that are being watched (with a standard deviation of 2.9% over the 15 machines), for a total of 0.15% of writes requiring ACF execution. Most of the inode block overlap resulted from these machines' `/etc/passwd` being regenerated nightly. This caused its inode to be in close proximity with many short-lived files in `/tmp`. On the one machine that had its own partition for `/tmp`, we found that only 0.013% of modifications caused writes to watched inode blocks. Using the values from Table II and the hit frequency over the 15 machines, we compute that the extra work would result in a 0.01–0.04% overhead (depending on the IDD cache hit rate).

Similarly, the IDD needs to watch directories between a watched file and the root directory to make sure that relevant entries do not change. We looked at the number of changes to unmatched entries in watched directories that the IDD would have to process given our traces. Using the same traces, we found that 0.22% of modifications to the file system result in directory changes that an ACF would need to process in order to verify that no rule was violated. Based on the Table II measurements, these ACF invocations would result in a 0.02–0.06% performance impact, depending on IDD cache hit rate.

7. ADDITIONAL RELATED WORK

Much related work has been discussed within the flow of the article. For emphasis, we note that there have been many intrusion detection systems focused on host OS activity and network communication. Axelsson [1998] surveyed and laid out classifications for many of these IDS types. Also, the most closely related tool, Tripwire [Kim and Spafford 1994], was used as an initial template for our prototype's file modification detection rule set.

Our work is part of a line of research exploiting physical [Ganger and Nagle 2001; Zhang et al. 2002] and virtual [Chen and Noble 2001; Payne et al. 2007] protection boundaries to detect intrusions into system software. Notably, Garfinkel and Rosenblum [2003] explore the utility of an IDS embedded in a Virtual Machine Monitor (VMM), which can inspect machine state while being

compromise-independent of most host software. Storage-based intrusion detection rules could be embedded in a VMM's storage module, rather than in a physical storage device, to identify suspicious storage activity.

After our initial explorations of the field of storage-based intrusion detection [Pennington et al. 2003; Griffin 2004], other researchers pursued complementary projects that advanced the field and demonstrated the versatility of active monitoring behind the storage interface.

Banikazemi et al. at IBM Research explored the commercial viability of storage-based intrusion detection [Banikazemi et al. 2005]. First, they extended our IDD concept beyond a single disk and into a managed storage area network, demonstrating a concrete realization of a real-time block-based storage device inside a commercially viable storage platform. Second, they observed the utility of using delayed execution an IDS rule-set over file system snapshots as an efficient complement to real-time IDS activity.

Paul et al. at the University of Virginia explored the architectural issues that will be faced in stand-alone semantically smart disk systems [Paul 2008; Paul et al. 2005]. They identified observable disk-level behaviors that are characteristic of malware and explored disk detection algorithms tuned to operate in the limited-resource embedded computational environments that will likely be characteristic of programmable storage devices.

Butler et al. at the Pennsylvania State University introduced the idea of using a removable administrative token to perform safe programming and administration of a storage-based IDS [Butler et al. 2008]. They identified an elegant empirical alternative to selecting which blocks should be treated as immutable by the IDS rule-set: with some exceptions, all blocks written to storage whenever the administrative token is present are considered immutable.

The results from these three independent projects collectively underscore our claims of the feasibility and efficacy of locating independent security monitoring and response utilities behind the storage interface.

Somewhere between block-based storage and file-based storage lies the emerging concept of object-based storage [Gibson et al. 1998; Weber 2004]. Storage-based intrusion detection is easier for storage objects than for blocks, since files often map directly to one or more objects. One such system has been demonstrated by Zhang and Wang [2006] who created a storage-based IDS running on an early object-based storage implementation.

Perhaps the most closely related work is the original proposal for self-securing storage [Strunk et al. 2000], which argued for storage-embedded support for intrusion survival. Self-securing storage retains every version of all data and a log of all requests for a period of time called the detection window. For intrusions detected within this window, security administrators have a wealth of information for postintrusion diagnosis and recovery.

Such versioning and auditing complements storage-based intrusion detection in several additional ways. First, when creating rules about storage activity for use in detection, administrators can use the latest audit log and version history to test new rules for false alarms. Second, the audit log could simplify implementation of rules looking for patterns of requests. Third, administrators

can use the history to investigate alerts of suspicious behavior (i.e., to check for supporting evidence within the history). Fourth, since the history is retained, a storage IDS can delay checks until the device is idle, allowing the device to avoid performance penalties for expensive checks by accepting a potentially longer detection latency.

8. CONCLUSIONS

A storage IDS watches system activity from a new viewpoint which immediately exposes some common intruder actions. Running on separate hardware, this functionality remains in place even when client OSES or user accounts are compromised. Our prototype storage IDS demonstrates both feasibility and efficiency of adding IDS capabilities to a workstation disk. Analysis of real intrusion tools indicates that most would be immediately detected by a storage IDS. After adjusting for storage IDS presence, intrusion tools will have to choose between exposing themselves to detection or being removed whenever the system reboots.

ACKNOWLEDGMENTS

We thank the members and companies of the PDL Consortium (including APC, Data Domain, EMC, EqualLogic, Facebook, Google, HP, HGST, Hitachi, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Panasas, Seagate, Sun, Symantec, VMware) for their interest, insights, feedback, and support.

REFERENCES

- AXELSSON, S. 1998. Research in intrusion-detection systems: A survey. Tech. rep. 98-17, Department of Computer Engineering, Chalmers University of Technology.
- BANIKAZEMI, M., POFF, D., AND ABALI, B. 2005. Storage-based intrusion detection for storage area networks (SANs). In *Proceedings of the IEEE Symposium on Mass Storage Systems*. IEEE Computer Society, 118-127.
- BISHOP, M. AND DILGER, M. 1996. Checking for race conditions in file accesses. *Comput. Syst.* 9, 2, 131-152.
- BUTLER, K. R. B., MCLAUGHLIN, S., AND MCDANIEL, P. D. 2008. Rootkit-resistant disks. In *Proceedings of the Conference on Computer and Communications Security (CCS'08)*. ACM, 403-416.
- CARD, R., TS'O, T., AND TWEEDIE, S. 1994. Design and implementation of the second extended file system. In *Proceedings of the 1st Dutch International Symposium on Linux*.
- CASTRO, M. AND LISKOV, B. 2000. Proactive recovery in a byzantine-fault-tolerant system. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. USENIX Association, 273-287.
- CHEN, P. M. AND NOBLE, B. D. 2001. When virtual is better than real. In *Proceedings of the Conference on Hot Topics in Operating Systems*. IEEE Computer Society, 133-138.
- CHESWICK, B. AND BELLOVIN, S. 1994. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, Reading, MA.
- DENNING, D. 1987. An intrusion-detection model. *IEEE Trans. Softw. Engin.* SE-13, 2, 222-232.
- DENNING, D. E. 1999. *Information Warfare and Security*. Addison-Wesley, Reading, MA.
- FARMER, D. 2000. What are MACtimes? *Dr. Dobbs's J.* 25, 10, 68-74.
- FORREST, S., HOFMEYR, S. A., SOMAYAJI, A., AND LONGSTAFF, T. A. 1996. A sense of self for UNIX processes. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 120-128.

- GANGER, G. R. AND NAGLE, D. F. 2001. Better security via smarter devices. In *Proceedings of the Conference on Hot Topics in Operating Systems*. IEEE, 100–105.
- GANGER, G. R., ECONOMOU, G., AND BIELSKI, S. M. 2003. Finding and containing enemies within the walls with self-securing network interfaces. Tech. rep. CMU-CS-03-109, Carnegie Mellon University.
- GARFINKEL, T. AND ROSENBLUM, M. 2003. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS'03)*. The Internet Society.
- GIBSON, G. A., NAGLE, D. F., AMIRI, K., BUTLER, J., CHANG, F. W., GOBIOFF, H., HARDIN, C., RIEDEL, E., ROCHBERG, D., AND ZELENKA, J. 1998. A cost-effective, high-bandwidth storage architecture. *SIGPLAN Not.* 33, 11, 92–103.
- GOBIOFF, H. 1999. Security for a high performance commodity storage subsystem. Ph.D. thesis, School of Computer Science, Carnegie Mellon University.
- GRIFFIN, J. L. 2004. Timing-accurate storage emulation: Evaluating hypothetical storage components in real computers. Ph.D. thesis, Carnegie Mellon University.
- HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. 1988. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.* 6, 1, 51–81.
- HUANG, Y. N., KINTALA, C. M. R., BERNSTEIN, L., AND WANG, Y. M. 1996. Components for software fault-tolerance and rejuvenation. *AT&T Bell Lab. Tech. J.* 75, 2, 29–37.
- KATCHER, J. 1997. Postmark: A new file system benchmark. Tech. rep. TR3022, Network Appliance.
- KIM, G. H. AND SPAFFORD, E. H. 1994. The design and implementation of Tripwire: A file system integrity checker. In *Proceedings of the Conference on Computer and Communications Security (CCS'94)*. ACM, 18–29.
- KO, C., RUSCHITZKA, M., AND LEVITT, K. 1997. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 175–187.
- KUMAR, P. AND SATYANARAYANAN, M. 1995. Flexible and safe resolution of file conflicts. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 95–106.
- LEMONS, R. 2002. Putting fun back into hacking. <http://zdnet.com/100-1105-948404.html>.
- LIU, P., JAJODIA, S., AND MCCOLLUM, C. D. 2000. Intrusion confinement by isolation in information systems. In *Proceedings of the IFIP Working Conference on Database Security*. IFIP, 3–18.
- LUNT, T. F. AND JAGANNATHAN, R. 1988. A prototype real-time intrusion-detection expert system. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 59–66.
- NFR 2002. Nfr security. <http://www.nfr.net/>.
- PACKETSTORM 2009. Packet storm security. <http://www.packetstormsecurity.org/>.
- PAUL, N., GURUMURTHI, S., AND EVANS, D. 2005. Towards disk-level malware detection. In *Proceedings of the CoBaSSA – Workshop on Code Based Software Security Assessments*.
- PAUL, N. R. 2008. Disk-level behavioral malware detection. Ph.D. thesis, University of Virginia.
- PAXSON, V. 1998. Bro: A system for detecting network intruders in real-time. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 31–51.
- PAYNE, B. D., DE A. CARBONE, M. D. P., AND LEE, W. 2007. Secure and flexible monitoring of virtual machines. In *Proceedings of the Computer Security Applications Conference (ACSAC'07)*. IEEE, 385–397.
- PENNINGTON, A. G., STRUNK, J. D., GRIFFIN, J. L., SOULES, C. A.N., GOODSON, G. R., AND GANGER, G. R. 2003. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *Proceedings of the USENIX Security Symposium*.
- PORRAS, P. A. AND NEUMANN, P. G. 1997. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the National Information Systems Security Conference*. 353–365.

- PURCZYNSKI, W. 2002. Gnu fileutils – Recursive directory removal race condition. <http://www.mail-archive.com/bug-fileutils@gnu.org/msg01537.html>.
- SAMAR, V. AND SCHEMERS III, R. J. 1995. Unified login with pluggable authentication modules (PAM). Tech. rep., Open Software Foundation RFC 86.0, Open Software Foundation.
- SCAMBRAY, J., MCCLURE, S., AND KURTZ, G. 2001. *Hacking Exposed: Network Security Secrets and Solutions*. Osborne/McGraw-Hill.
- SCHNEIER, B. AND KELSEY, J. 1999. Secure audit logs to support computer forensics. *ACM Trans. Inf. Syst. Secur.* 2, 2, 159–176.
- SIVATHANU, M., PRABHAKARAN, V., POPOVICI, F. I., DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 2003. Semantically smart disk systems. In *Proceedings of the Conference on File and Storage Technologies*. USENIX Association, 73–88.
- STROM, R. 2008. Emc Celerra family technical review. http://www.emc.com/pdf/partnersalliances/einfo/McAfee_netshield.pdf.
- STRUNK, J. D., GOODSON, G. R., SCHEINHOLTZ, M. L., SOULES, C. A. N., AND GANGER, G. R. 2000. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. USENIX Association, 165–180.
- SUGERMAN, J., VENKITACHALAM, G., AND LIM, B.-H. 2001. Virtualizing I/O devices on vmware workstation’s hosted virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 1–14.
- SURESHKUMAR, N. 2009. Antivirus scanning best practices guide. Tech. rep., Network Appliance Inc. <http://media.netapp.com/documents/tr-3107.pdf>
- TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. *Oper. Syst. Rev.* 29, 5.
- TRIPWIRE. 2002. Tripwire open source 2.3.1. <http://ftp4.sf.net/sourceforge/tripwire/tripwire-2.3.1-2.tar.gz>.
- VAIDYANATHAN, K., HARPER, R. E., HUNTER, S. W., AND TRIVEDI, K. S. 2002. Analysis and implementation of software rejuvenation in cluster systems. *Perform. Eval. Rev.* 29, 1, 62–71.
- WEBER, R. O. 2004. Scsi object-based storage device commands (osd). <ftp://ftp.t10.org/t10/drafts/osd/osd-r10.pdf>.
- ZHANG, X., VAN DOORN, L., JAEGER, T., PEREZ, R., AND SAILER, R. 2002. Secure coprocessor-based intrusion detection. In *Proceedings of the ACM SIGOPS European Workshop*. ACM.
- ZHANG, Y. AND WANG, D. 2006. Research on object-storage-based intrusion detection. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems (ICPADS’06)*. IEEE Computer Society, 68–78.

Received April 2008; revised July 2009; accepted August 2009