Appears in Proceedings of the 12th USENIX Security Symposium
Washington, DC. August 2003.

Storage-based Intrusion Detection: Watching storage activity for suspicious behavior

Adam G. Pennington, John D. Strunk, John Linwood Griffin,
Craig A.N. Soules, Garth R. Goodson, Gregory R. Ganger
Carnegie Mellon University

Abstract

Storage-based intrusion detection allows storage systems to watch for data modifications characteristic of system intrusions. This enables storage systems to spot several common intruder actions, such as adding backdoors, inserting Trojan horses, and tampering with audit logs. Further, an intrusion detection system (IDS) embedded in a storage device continues to operate even after client systems are compromised. This paper describes a number of specific warning signs visible at the storage interface. Examination of 18 real intrusion tools reveals that most (15) can be detected based on their changes to stored files. We describe and evaluate a prototype storage IDS, embedded in an NFS server, to demonstrate both feasibility and efficiency of storage-based intrusion detection. In particular, both the performance overhead and memory required (152 KB for 4730 rules) are minimal.

1 Introduction

Many intrusion detection systems (IDSs) have been developed over the years [1, 23, 29], with most falling into one of two categories: network-based or host-based. Network IDSs (NIDS) are usually embedded in sniffers or firewalls, scanning traffic to, from, and within a network environment for attack signatures and suspicious traffic [5, 25]. Host-based IDSs (HIDS) are fully or partially embedded within each host's OS. They examine local information (such as system calls [10]) 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 [7, p. 218][34, 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 must 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 paper 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 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 [18]. In general, compromises cannot hide their changes from the storage device if they wish to persist across reboots; to be re-installed upon reboot, the 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 perhaps 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 [27], but with fewer varieties and structural changes over time.

As a concrete example with which to experiment, we have augmented an NFS server with a storage IDS that supports online, rule-based detection of suspicious modifications. This storage IDS supports the detection of four categories of suspicious activities. First, it can detect unexpected changes to important system files and binaries, using a rule-set very similar to Tripwire's. Second, it can detect patterns of changes like non-append modification (e.g., of system log files) and reversing of inode times. Third, it can detect specifically proscribed content changes to critical files (e.g., illegal shells inserted into `/etc/passwd`). Fourth, it can detect the appearance of specific file names (e.g., hidden "dot" files) or content (e.g., known viruses or attack tools). An administrative interface supplies the

detection rules, which are checked during the processing of each NFS request. When a detection rule triggers, the server sends the administrator an alert containing the full pathname of the modified file, the violated rule, and the offending NFS operation. Experiments show that the runtime cost of such intrusion detection is minimal. Further analysis indicates that little memory capacity is needed for reasonable rulesets (e.g., only 152 KB for an example containing 4730 rules).

The remainder of this paper 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 an NFS server. Section 6 uses this prototype to evaluate the costs of storage-based intrusion detection. Section 7 presents related work. Section 8 summarizes this paper's contributions and discusses continuing work.

2 Storage-based Intrusion Detection

Storage-based intrusion detection enables storage devices to examine the requests they service for suspicious client behavior. Although the world view that a storage server 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 on 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 non-technical 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 more difficult. Host-based IDSs can be rendered ineffective by intruder software that disables them or feeds them misinformation, for which many tools exist. 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 [11]. 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.

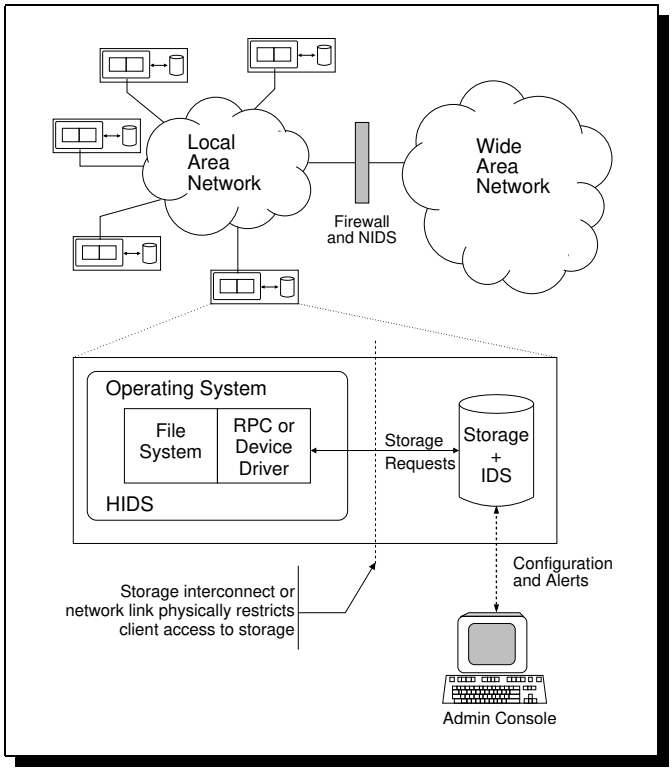

Figure 1: **The compromise independence of a storage IDS.** The storage interface provides a physical boundary behind which a storage server can observe the requests it is asked to service. Note that this same picture works for block protocols, such as SCSI or IDE/ATA, and distributed file system protocols,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

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 paper could also be spotted from within a 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 [39]; such deployment would enjoy compromise independence from the OSes running in its virtual machines [4].

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.

2.3.1 Data/attribute modification

In managed computing environments, the simplest (and perhaps most effective) category of warning signs consists of data or meta-data 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 [18]) 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 be useful in watching. The clearest is client system audit logs; these audit logs are critical to both intrusion detection [6] and diagnosis [35], 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." This 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 post-intrusion diagnosis of which files were manipulated [9]. 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 activ-

ity 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 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 10 GB of multimedia files). When the system reaches predetermined thresholds of unallocated resources and allocation rate, warning the administrator is appropriate even in non-intrusion 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 inter-related 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 [10] or starting with rules regarding the expected behavior of individual applications [19]. 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 vs. 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 line-break, 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 DEF CON conference [21], 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 down-time 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, non-empty password field, legal default shell, legal home directory, non-overlapping user IDs, etc. The attack described above, 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 [24] and Network Appliance [28]) now include 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 [7, 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 [2, 30] 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 modify the OS but change no 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 one side takes away an avenue of attack, the other 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 those modifications do not propagate to storage. A reboot, however, will reset the system and remove the intrusion, which argues for proactive restart [3, 16, 43]. To counter this, attackers must have their changes re-established automatically after a reboot, such as by manipulating the various boot-time (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 non-trivial rules without adding new inter-device 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 1) designed to be run on compromised systems. All were downloaded from public websites, most of them from Packet Storm [26].

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. The second provides a mechanism for reentry into a system. Twelve of the tools operate by running various binaries on the host system and overwriting existing binaries to continue gaining 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 logfile 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. However, they are removed if the system reboots; all three modify the kernel, but would have to be combined with system file changes to be re-inserted upon reboot.

Non-append 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

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

Name	Description	Syscall redir.	Log scrub	Hidden dirs	Watched files	Total alerts
Ramen	Linux worm			X	2	3
1ion	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

Table 1: **Visible actions of several intruder toolkits.** 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 alerts generated by a given tool.

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`.

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 [33] 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 non-executable 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 above. 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 the writing of this paper, one of the authors happened to be asked to analyze a system that had been recently compromised. Several modifications similar to those made by the above 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 setup to hide the existence of an IRC bot, running out of the 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 those rules should be as simple and straightforward as possible. Each of the four categories 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.

Less guidance exists for the other two categories of warning signs: update patterns and content integrity. We do not currently know how to specify general rules for these categories. Our approach has been to fall back on Tripwire-style rules; we hard-code checking functions (e.g., for non-append update or a particular content integrity violation) and then allow an administrator to specify on which files they should be checked (or that they should be checked for every file). More general approaches to specifying detection rules for these categories of warning signs are left for future work.

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 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 above 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 ex-

ist [14]. 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.

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 insecure environments, cryptography is the only viable solution.

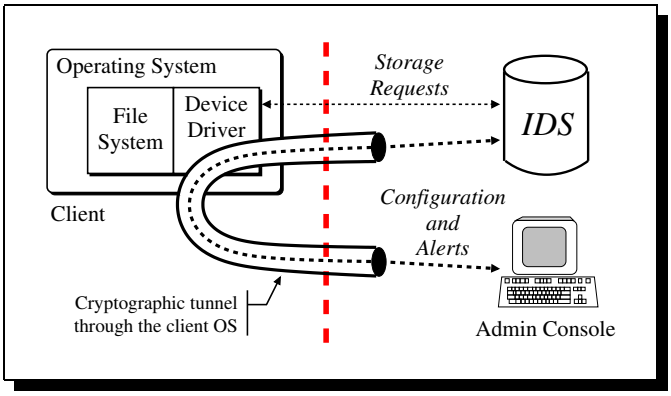

Figure 2: **Tunneling administrative commands through client systems.** For storage devices attached directly to client systems, a cryptographic tunnel can allow the administrator to securely manage a storage IDS. This tunnel uses the untrusted client OS to transport administrative commands and alerts.

4.3 Checking the detection rules

Checking detection rules can be non-trivial, because rules generally apply to full pathnames rather than inodes. Additional complications arise because 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 (usually a simple flag comparison, sometimes a content check). For operations that affect the file system's namespace, verification is more complicated. For example, a rename of a directory tree 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 the case of rules pertaining to files that do not currently exist, this list of rules must be consulted 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 [22]. 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 [20, 41] problem, should the updates be judged legitimate. Still, it is interesting to consider em-

bedding 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 post-intrusion 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. [38] argue for always doing these things (just in case). Still, IDS-triggered employment of this functionality may be a useful trade-off point.

Metadata	
• inode modification time	• data modification time
• access time	• file permissions
• link count	• device number
• file owner	• inode number
• file type	• file owner group
• file size	
Data	
• any modification	• append only
• password structure	

Table 2: **Attribute list.** Rules can be established to watch these attributes in real-time on a file-by-file basis.

5 Storage-based intrusion detection in an NFS server

To explore the concepts and feasibility of storage-based intrusion detection, we implemented a storage IDS in an NFS server. Unmodified client systems access the server using the standard NFS version 2 protocol [40][2], while storage-based intrusion detection occurs transparently. This section describes how the prototype storage IDS handles detection rule specification, the structures and algorithms for checking rules, and alert generation.

The base NFS server is called S4, and its implementation is described and evaluated elsewhere [38]. It internally performs file versioning and request auditing, using a log-structured file system [32], but these features are not relevant here. For our purposes, it is a convenient NFS file server with performance comparable to the Linux and FreeBSD NFS servers. Secure administration is performed via the server's console, using the physical access control approach.

5.1 Specifying detection rules

Our prototype storage IDS is capable of watching for a variety of data and metadata changes to files. The administrator specifies a list of Tripwire-style rules to configure the detection system. Each administrator-supplied rule is of the form: {*pathname, attribute-list*}—designating which attributes to monitor for a particular file. The list of attributes that can be watched is shown in Table 2. In addition to standard Tripwire rules, we have added two additional functions that can be specified on a per-file basis. The first watches for non-append changes, as described earlier; any truncation or write anywhere but at the previous end of a file will generate an alert. The second checks a file's integrity against the password file integrity rule discussed earlier. After every write, the file must conform to the rigid structure of a password file (7 colons per line), and all of the shells must be contained in the "acceptable" list.

In addition to per-file rules, an administrator can choose to enable any of three system-wide rules: one that matches on any operation that rolls-back a file's modification time, one that matches on any operation that creates a "hidden" directory (e.g., a directory name beginning with '.' and having spaces in it), and one that looks for known (currently hard-coded) intrusion tools by their sizes and SHA-1 digests. Although the system currently checks the digests on every file update, periodic scanning of the system would likely be more practical. These rules apply to all parts of the directory hierarchy and are specified as simply ON or OFF.

Rules are communicated to the server through the use of an administrative RPC. This RPC interface has two commands (see Table 3). The `setRule()` RPC gives the IDS two values: the path of the file to be watched, and a set of flags describing the specific rules for that file. Rules are removed through the same mechanism, specifying the path and an empty rule set.

5.2 Checking the detection rules

This subsection describes the core of the storage IDS. It discusses how rules are stored and subsequently checked during operation.

5.2.1 Data structures

Three new structures allow the storage IDS to efficiently support the detection rules: the reverse lookup table, the inode watch flags, and the non-existent names table.

Reverse lookup table: The reverse lookup table serves two functions. First, it serves as a list of rules that the server

[2]The use of the NFSv2 protocol is an artifact of the server implementation the IDS is built into, but makes no difference in the areas we care about.

Command	Purpose	Direction
`setRule(path, rules)`	Changes the watched characteristics of a file. This command is used to both set and delete rules.	admin⇒server
`listRules()`	Retrieves the server's rule table as a list of {`pathname`, `rules`} records.	admin⇒server
`alert(path, rules, operation)`	Delivers a warning of a rule violation to the administrator.	server⇒admin

Table 3: **Administrative commands for our storage IDS.** This table lists the small set of administrative commands needed for an administrative console to configure and manage the storage IDS. The first two are sent by the console, and the third is sent by the storage IDS. The pathname refers to a file relative to the root of an exported file system. The *rules* are a description of what to check for, which can be any of the changes described in Table 2. The *operation* is the NFS operation that caused the rule violation.

is currently enforcing. Second, it maps an inode number to a pathname. The alert generation mechanism uses the latter to provide the administrator with file names instead of inode numbers, without resorting to a brute-force search of the namespace.

The reverse lookup table is populated via the `setRule()` RPC. Each rule's full pathname is broken into its component names, which are stored in distinct rows of the table. For each component, the table records four fields: *inode-number*, *directory-inode-number*, *name*, and *rules*. Indexed by *inode-number*, an entry contains the *name* within a parent directory (identified by its *directory-inode-number*). The *rules* associated with this *name* are a bitmask of the attributes and patterns to watch. Since a particular inode number can have more than one name, multiple entries for each inode may exist. A given inode number can be translated to a full pathname by looking up its lowest-level name and recursively looking up the name of the corresponding directory inode number. The search ends with the known inode number of the root directory. All names for an inode can be found by following all paths given by the lookup of the inode number.

Inode `watchflags` field: During the `setRule()` RPC, in addition to populating the reverse lookup table, a rule mask of 16 bits is computed and stored in the `watchflags` field of the watched file's inode. Since multiple pathnames may refer to the same inode, there may be more than one rule for a given file, and the mask contains the union. The inode `watchflags` field is a performance enhancement designed to co-locate the rules governing a file with that file's metadata. This field is not necessary for correctness since the pertinent data could be read from the reverse lookup table. However, it allows efficient verification of detection rules during the processing of an NFS request. Since the inode is read as part of any file access, most rule checking becomes a simple mask comparison.

Non-existent names table: The non-existent names table lists rules for pathnames that do not currently exist. Each entry in the table is associated with the deepest-level (existing) directory within the pathname of the original rule. Each entry contains three fields: *directory-inode-number*, *remaining-path*, and *rules*. Indexed by *directory-inode-number*, an entry specifies the *remaining-path*. When a file or directory is created or removed, the non-existent names table is consulted and updated, if necessary. For example, upon creation of a file for which a detection rule exists, the *rules* are checked and inserted in the `watchflags` field of the inode. Together, the reverse lookup table and the non-existent names table contain the entire set of IDS rules in effect.

5.2.2 Checking rules during NFS operations

We now describe the flow of rule checking, much of which is diagrammed in Figure 3, in two parts: changes to individual files and changes to the namespace.

Checking rules on individual file operations: For each NFS operation that affects only a single file, a mask of rules that might be violated is computed. This mask is compared, bitwise, to the corresponding `watchflags` field in the file's inode. For most of the rules, this comparison quickly determines if any alerts should be triggered. If the "password file" or "append only" flags are set, the corresponding verification function executes to determine if the rule is violated.

Checking rules on namespace operations: Namespace operations can cause watched pathnames to appear or disappear, which will usually trigger an alert. For operations that create watched pathnames, the storage IDS moves rules from the non-existent names table to the reverse lookup table. Conversely, operations that delete watched pathnames cause rules to move between tables in the opposite direction.

When a name is created (via CREATE, MKDIR, LINK, or SYMLINK) the non-existent names table is checked. If there are rules for the new file, they are checked and placed in the `watchflags` field of the new inode. In addition, the corresponding rule is removed from the non-existent

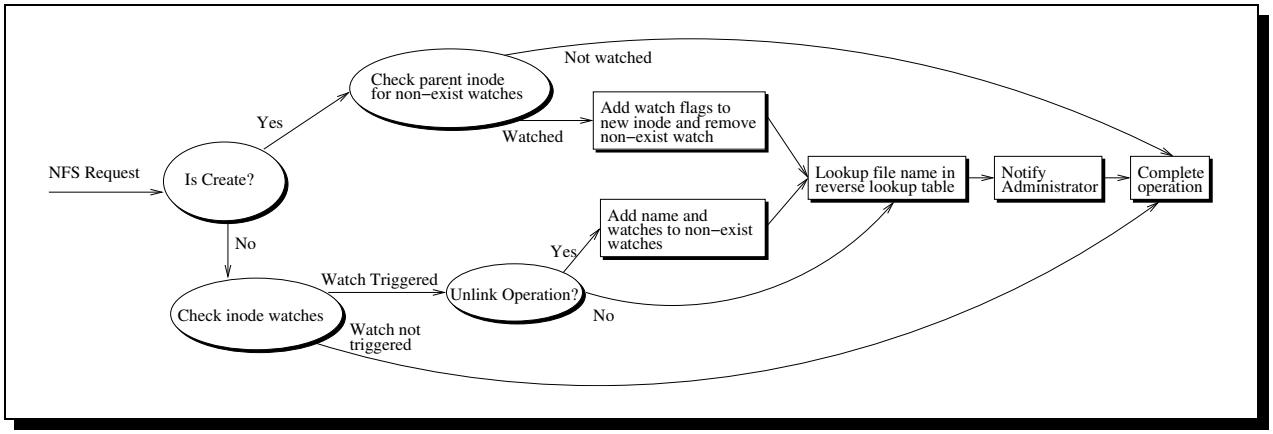

Figure 3: **Flowchart of our storage IDS.** Few structures and decision points are needed. In the common case (no rules for the file), only one inode's `watchflags` field is checked. The picture does not show RENAME operations here due to their complexity.

names table and is added to the reverse lookup table. During a MKDIR, any entries in the non-existent names table that include the new directory as the next step in their remaining path are replaced; the new entries are indexed by the new directory's inode number and its name is removed from the remaining path.

When a name is removed (via UNLINK or RMDIR), the `watchflags` field of the corresponding inode is checked for rules. Most such rules will trigger an alert, and an entry for them is also added to the non-existent names table. For RMDIR, the reverses of the actions for MKDIR are necessary. Any non-existent table entries parented on the removed directory must be modified. The removed directory's name is added to the beginning of each remaining path, and the directory inode number in the table is modified to be the directory's parent.

By far, the most complex namespace operation is a RENAME. For a RENAME of an individual file, modifying the rules is the same as a CREATE of the new name and a REMOVE of the old. When a directory is renamed, its subtrees must be recursively checked for watched files. If any are found, and once appropriate alerts are generated, their rules and pathname up to the parent of the renamed directory are stored in the non-existent names table, and the `watchflags` field of the inode is cleared. Then, the non-existent names table must be checked (again recursively) for any rules that map into the directory's new name and its children; such rules are checked, added to the inode's `watchflags` field, and updated as for name creation.

5.3 Generating alerts

Alerts are generated and sent immediately when a detection rule is triggered. The alert consists of the original detection rule (pathname and attributes watched), the specific attributes that were affected, and the RPC operation that triggered the rule. To get the original rule information, the reverse lookup table is consulted. If a single RPC operation triggers multiple rules, one alert is sent for each.

5.4 Storage IDS rules in a NIDS

Because NFS traffic goes over a traditional network, the detection rules described for our prototype storage IDS could be implemented in a NIDS. However, this would involve several new costs. First, it would require the NIDS to watch the LAN links that carry NFS activity. These links are usually higher bandwidth than the Internet uplinks on which most NIDSs are used.[3] Second, it would require that the NIDS replicate a substantial amount of work already performed by the NFS server, increasing the CPU requirements relative to an in-server storage IDS. Third, the NIDS would have to replicate and hold substantial amounts of state (e.g. mappings of file handles to their corresponding files). Our experiences checking rules against NFS traces indicate that this state grows rapidly because the NFS protocol does not expose to the network (or the server) when such state can be removed. Even simple attribute updates cannot be checked without caching the old values of the attributes, otherwise the NIDS could not distinguish modified attributes from reapplied values. Fourth, rules cannot always be checked by looking only at the current command. The NIDS may need to read file data and attributes to deal with namespace operations, content integrity checks, and update pattern rules. In addition to the performance penalty, this requires giving the NIDS read permission for all NFS files and directories.

Given all of these issues, we believe that embedding storage IDS checks directly into the storage component is more appropriate.

[3] Tapping a NIDS into direct-attached storage interconnects, such as SCSI and FibreChannel, would be more difficult.

6 Evaluation

This section evaluates the costs of our storage IDS in terms of performance impact and memory required—both costs are minimal.

6.1 Experimental setup

All experiments use the S4 NFS server, with and without the new support for storage-based intrusion detection. The client system is a dual 1 GHz Pentium III with 128 MB RAM and a 3Com 3C905B 100 Mbps network adapter. The server is a dual 700 MHz Pentium III with 512 MB RAM, a 9 GB 10,000 RPM Quantum Atlas 10K II drive, an Adaptec AIC-7896/7 Ultra2 SCSI controller, and an Intel EtherExpress Pro 100 Mb network adapter. The client and server are on the same 100 Mb network switch. The operating system on all machines is Red Hat Linux 6.2 with Linux kernel version 2.2.14.

SSH-build was constructed as a replacement for the Andrew file system benchmark [15, 36]. It consists of 3 phases: The unpack phase, which unpacks the compressed tar archive of SSH v. 1.2.27 (approximately 1 MB in size before decompression), stresses metadata operations on files of varying sizes. The configure phase consists of the automatic generation of header files and makefiles, which involves building various small programs that check the existing system configuration. The build phase compiles, links, and removes temporary files. This last phase is the most CPU intensive, but it also generates a large number of object files and a few executables. Both the server and client caches are flushed between phases.

PostMark was designed to measure the performance of a file system used for electronic mail, netnews, and web based services [17]. It creates a large number of small randomly-sized files (between 512 B and 16 KB) and performs a specified number of transactions on them. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. The default configuration used for the experiments consists of 100,000 transactions on 20,000 files, and the biases for transaction types are equal.

6.2 Performance impact

The storage IDS checks a file's rules before any operation that could possibly trigger an alert. This includes READ operations, since they may change a file's last access time. Additionally, namespace-modifying operations require further checks and possible updates of the non-existent names table. To understand the performance consequences of the storage IDS design, we ran PostMark and SSH-Build tests. Since our main concern is avoiding a performance loss in the case where no rule is violated, we ran these benchmarks with no relevant rules set. As long as no rules match, the results are similar with 0 rules, 1000 rules on existing files, or 1000 rules on non-existing files. Table 4 shows that the performance impact of the storage IDS is minimal. The largest performance difference is for the configure and build phases of SSH-build, which involve large numbers of namespace operations.

Benchmark	Baseline	With IDS	Change
SSH untar	27.3 (0.02)	27.4 (0.02)	0.03%
SSH config.	42.6 (0.68)	43.2 (0.37)	1.3%
SSH build	85.9 (0.18)	86.8 (0.17)	1.0%
PostMark	4288 (11.9)	4290 (13.0)	0.04%

Table 4: **Performance of macro benchmarks.** All benchmarks were run with and without the storage IDS functionality. Each number represents the average of 10 trials in seconds (with the standard deviation in parenthesis).

Benchmark	Baseline	With IDS	Change
Create	4.32	4.35	0.7%
Remove	4.50	4.65	3.3%
Mkdir	4.36	4.38	0.5%
Rmdir	4.52	4.59	1.5%
Rename file	3.81	3.91	2.6%
Rename dir	3.91	4.04	3.3%

Table 5: **Performance of micro benchmarks.** All benchmarks were run with and without the storage IDS functionality. Each number represents the average of 1000 trials in milliseconds.

Microbenchmarks on specific filesystem actions help explain the overheads. Table 5 shows results for the most expensive operations, which all affect the namespace. The performance differences are caused by redundancy in the implementation. The storage IDS code is kept separate from the NFS server internals, valuing modularity over performance. For example, name removal operations involve a redundant directory lookup and inode fetch (from cache) to locate the corresponding inode's `watchflags` field.

Rules take very little time to generate alerts. For example, a write to a file with a rule set takes 4.901 milliseconds if no alert is set off. If an alert is set off the time is 4.941 milliseconds. These represent the average over 1000 trials, and show a .8% overhead.

6.3 Space efficiency

The storage IDS structures are stored on disk. To avoid extra disk accesses for most rule checking, though, it is important that they fit in memory.

Three structures are used to check a set of rules. First, each inode in the system has an additional two-byte field for the bitmask of the rules on that file. There is no cost for this, because the space in the inode was previously unused. Linux's ext2fs and BSD's FFS also have sufficient unused space to store such data without increasing their inode sizes. If space were not available, the reverse lookup table can be used instead, since it provides the same information. Second, for each pathname component of a rule, the reverse lookup table requires $20 + N$ bytes: a 16-byte inode number, 2 bytes for the rule bitmask, and $N+2$ bytes for a pathname component of length N. Third, the non-existent names table contains one entry for every file being watched that does not currently exist. Each entry consumes 274 bytes: a 16-byte inode number, 2 bytes for the rule bitmask, and 256 bytes for the maximum pathname supported.

To examine a concrete example of how an administrator might use this system, we downloaded the open source version of Tripwire [42]. Included with it is an example rule file for Linux, containing (after expanding directories to lists of files) 4730 rules. We examined a Red Hat Linux 6.1 [31] desktop machine to obtain an idea of the number of watched files that actually exist on the hard drive. Of the 4730 watched files, 4689 existed on our example system. Using data structure sizes from above, reverse lookup entries for the watched files consume 141 KB. Entries in the non-existent name table for the remaining 41 watched files consume 11 KB. In total, only 152 KB are needed for the storage IDS.

6.4 False positives

We have explored the false positive rate of storage-based intrusion detection in several ways.

To evaluate the file watch rules, two months of traces of all file system operations were gathered on a desktop machine in our group. We compared the files modified on this system with the watched file list from the open source version of Tripwire. This uncovered two distinct patterns where files were modified. Nightly, the user list (`/etc/passwd`) on the machine was overwritten by a central server. Most nights it does not change but the create and rename performed would have triggered an alert. Additionally, multiple binaries in the system were replaced over time by the administrative upgrade process. In only one case was a configuration file on the system changed by a local user.

For alert-triggering modifications arising from explicit administrative action, a storage IDS can provide an added benefit. If an administrator pre-informs the admin console of updated files before they are distributed to machines, the IDS can verify that desired updates happen correctly. Specifically, the admin console can read the new contents via the admin channel and verify that they are as intended. If so, the update is known to have succeeded, and the alert can be suppressed.

We have also performed two (much) smaller studies. First, we have evaluated our "hidden filename" rules by examining the entire filesystems of several desktops and servers—we found no uses of any of them, including the '.' or '..' followed by any number of spaces discussed above. Second, we evaluated our "inode time reversal" rules by examining lengthy traces of NFS activity from our environment and from two Harvard environments [8]—we found a sizable number of false positives, caused mainly by unpacking archives with utilities like `tar`. Combined with the lack of time reversal in any of the toolkits, use of this rule may be a bad idea.

7 Additional Related Work

Much related work has been discussed within the flow of the paper. For emphasis, we note that there have been many intrusion detection systems focused on host OS activity and network communication; Axelsson [1] recently surveyed the state-of-the-art. Also, the most closely related tool, Tripwire [18], was used as an initial template for our prototype's file modification detection ruleset.

Our work is part of a recent line of research exploiting physical [12, 44] and virtual [4] protection boundaries to detect intrusions into system software. Notably, Garfinkel et al. [13] 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.

Perhaps the most closely related work is the original proposal for self-securing storage [38], 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 post-intrusion 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 and Future Work

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 within a file server. 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.

In continuing work, we are developing a prototype storage IDS embedded in a device exporting a block-based interface (SCSI). To implement the same rules as our augmented NFS server, such a device must be able to parse and traverse the on-disk metadata structures of the file system it holds. For example, knowing whether `/usr/sbin/sshd` has changed on disk requires knowing not only whether the corresponding data blocks have changed, but also whether the inode still points to the same blocks and whether the name still translates to the same inode. We have developed this translation functionality for two popular file systems, Linux's ext2fs and FreeBSD's FFS. The additional complexity required is small (under 200 lines of C code for each), simple (under 3 days of programming effort each), and changes infrequently (about 5 years between incompatible changes to on-disk structures). The latter, in particular, indicates that device vendors can deploy firmware and expect useful lifetimes that match the hardware. Sivathanu et al. [37] have evaluated the costs and benefits of device-embedded FS knowledge more generally, finding that it is feasible and valuable.

Another continuing direction is exploration of less exact rules and their impact on detection and false positive rates. In particular, the potential of pattern matching rules and general anomaly detection for storage remains unknown.

Acknowledgments

We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. We thank IBM and Intel for hardware grants supporting our research efforts. This material is based on research sponsored by the Air Force Research Laboratory, under agreement number F49620-01-1-0433, and by DARPA/ITO's OASIS program, under Air Force contract number F30602-99-2-0539-AFRL.[4] Craig Soules was supported by a USENIX Fellowship. Garth Goodson was supported by an IBM Fellowship.

References

[1] S. Axelsson. *Research in intrusion-detection systems: a survey*. Technical report 98–17. Department of Computer Engineering, Chalmers University of Technology, December 1998.

[2] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, **9**(2):131–152, Spring 1996.

[3] M. Castro and B. Liskov. Proactive recovery in a Byzantine-fault-tolerant system. Symposium on Operating Systems Design and Implementation, pages 273–287. USENIX Association, 2000.

[4] P. M. Chen and B. D. Noble. When virtual is better than real. Hot Topics in Operating Systems, pages 133–138. IEEE Comput. Soc., 2001.

[5] B. Cheswick and S. Bellovin. *Firewalls and Internet security: repelling the wily hacker*. Addison-Wesley, Reading, Mass. and London, 1994.

[6] D. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, **SE-13**(2):222–232, February 1987.

[7] D. E. Denning. *Information warfare and security*. Addison-Wesley, 1999.

[8] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of an email and research workload. Conference on File and Storage Technologies, pages 203–217. USENIX Association, 2003.

[9] D. Farmer. What are MACtimes? *Dr. Dobb's Journal*, **25**(10):68–74, October 2000.

[10] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for UNIX processes. IEEE Symposium on Security and Privacy, pages 120–128. IEEE, 1996.

[11] G. R. Ganger, G. Economou, and S. M. Bielski. *Finding and Containing Enemies Within the Walls with Self-securing Network Interfaces*. Carnegie Mellon University Technical Report CMU-CS-03-109. January 2003.

[12] G. R. Ganger and D. F. Nagle. Better security via smarter devices. Hot Topics in Operating Systems, pages 100–105. IEEE, 2001.

[4] The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

[13] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. NDSS. The Internet Society, 2003.

[14] H. Gobioff. *Security for a high performance commodity storage subsystem*. PhD thesis, published as TR CMU–CS–99–160. Carnegie-Mellon University, Pittsburgh, PA, July 1999.

[15] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, **6**(1):51–81, February 1988.

[16] Y. N. Huang, C. M. R. Kintala, L. Bernstein, and Y. M. Wang. Components for software fault-tolerance and rejuvenation. *AT&T Bell Laboratories Technical Journal*, **75**(2):29–37, March-April 1996.

[17] J. Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.

[18] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: a file system integrity checker. Conference on Computer and Communications Security, pages 18–29. ACM, 1994.

[19] C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in distributed systems: a specification-based approach. IEEE Symposium on Security and Privacy, pages 175–187. IEEE, 1997.

[20] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. USENIX Annual Technical Conference, pages 95–106. USENIX Association, 1995.

[21] R. Lemos. Putting fun back into hacking. ZDNet News, 5 August 2002. `http://zdnet.com.com/2100-1105-948404.html`.

[22] P. Liu, S. Jajodia, and C. D. McCollum. Intrusion confinement by isolation in information systems. IFIP Working Conference on Database Security, pages 3–18. IFIP, 2000.

[23] T. F. Lunt and R. Jagannathan. A prototype real-time intrusion-detection expert system. IEEE Symposium on Security and Privacy, pages 59–66. IEEE, 1988.

[24] McAfee NetShield for Celerra. EMC Corporation, August 2002. `http://www.emc.com/pdf/partnersalliances/einfo/McAfee_netshield.pdf`.

[25] NFR Security. `http://www.nfr.net/`, August 2002.

[26] Packet Storm Security. Packet Storm, 26 January 2003. `http://www.packetstormsecurity.org/`.

[27] V. Paxson. Bro: a system for detecting network intruders in real-time. USENIX Security Symposium, pages 31–51. USENIX Association, 1998.

[28] J. Phillips. *Antivirus scanning best practices guide*. Technical report 3107. Network Appliance Inc. `http://www.netapp.com/tech_library/3107.html`.

[29] P. A. Porras and P. G. Neumann. EMERALD: event monitoring enabling responses to anomalous live disturbances. National Information Systems Security Conference, pages 353–365, 1997.

[30] W. Purczynski. GNU fileutils – recursive directory removal race condition. BugTraq mailing list, 11 March 2002.

[31] Red Hat Linux 6.1, 4 March 1999. `ftp://ftp.redhat.com/pub/redhat/linux/6.1/`.

[32] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, **10**(1):26–52. ACM Press, February 1992.

[33] V. Samar and R. J. Schemers III. *Unified login with pluggable authentication modules (PAM)*. Open Software Foundation RFC 86.0. Open Software Foundation, October 1995.

[34] J. Scambray, S. McClure, and G. Kurtz. *Hacking exposed: network security secrets & solutions*. Osborne/McGraw-Hill, 2001.

[35] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security*, **2**(2):159–176. ACM, May 1999.

[36] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. USENIX Annual Technical Conference, 2000.

[37] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. Conference on File and Storage Technologies, pages 73–89. USENIX Association, 2003.

[38] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: protecting data in compromised systems. Symposium on Operating Systems Design and Implementation, pages 165–180. USENIX Association, 2000.

[39] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. USENIX Annual Technical Conference, pages 1–14. USENIX Association, 2001.

[40] Sun Microsystems. *NFS: network file system protocol specification*, RFC–1094, March 1989.

[41] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **29**(5), 1995.

[42] Tripwire Open Souce 2.3.1, August 2002. `http://ftp4.sf.net/sourceforge/tripwire/tripwire-2.3.1-2.tar.gz`.

[43] K. Vaidyanathan, R. E. Harper, S. W. Hunter, and K. S. Trivedi. Analysis and implementation of software rejuvenation in cluster systems. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. Published as *Performance Evaluation Review*, **29**(1):62–71. ACM Press, 2002.

[44] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer. Secure Coprocessor-based Intrusion Detection. ACM SIGOPS European Workshop. ACM, 2002.