Intrusion Detection, Diagnosis, and Recovery with Self-Securing Storage

John D. Strunk, Garth R. Goodson, Adam G. Pennington,

Craig A.N. Soules, Gregory R. Ganger

May 2002

CMU-CS-02-140

School of Computer Science

Carnegie Mellon University

Pittsburgh, PA 15213

Abstract

Self-securing storage turns storage devices into active parts of an intrusion survival strategy. From behind a thin storage interface (e.g., SCSI or CIFS), a self-securing storage server can watch storage requests, keep a record of all storage activity, and prevent compromised clients from destroying stored data. This paper describes three ways self-securing storage enhances an administrator's ability to detect, diagnose, and recover from client system intrusions. First, storage-based intrusion detection offers a new observation point for noticing suspect activity. Second, post-hoc intrusion diagnosis starts with a plethora of normally-unavailable information. Finally, post-intrusion recovery is reduced to restarting the system with a pre-intrusion storage image retained by the server. Combined, these features can improve an organization's ability to survive successful digital intrusions.

We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Network Appliance, 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. 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. This work is partially funded by DARPA/ITO's OASIS program (Air Force contract number F30602-99-2-0539-AFRL). Craig Soules is supported by a USENIX Fellowship. Garth Goodson is supported by an IBM Fellowship.

Keywords: Security, survivability, intrusion tolerance, storage systems, network-attached storage

1 Introduction

Digital intrusions are a fact of modern computing. While new security technologies may make them more difficult and less frequent, intrusions will occur as long as software is buggy and users are fallible. Once an intruder infiltrates the system, he can generally gain control of all system resources, including its storage access rights (complete rights, in the case of an OS accessing local storage). Crafty intruders can use this control to hide their presence, weaken system security, and manipulate sensitive data. Because storage acts as a slave to authorized principles, evidence of such actions can generally be hidden. In fact, so little of the system state is trustworthy after an intrusion that a common "recovery" approach is to reformat local storage, re-install the OS from scratch, and restore users' data from a pre-intrusion backup.

Self-securing storage is an exciting new technology for enhancing intrusion survival by enabling the storage device to safeguard data even when the host OS is compromised. It capitalizes on the fact that storage servers (whether file servers, disk array controllers, or even IDE disks) run separate software on separate hardware. This opens the door to server-embedded security that cannot be disabled by any software (even the OS) running on client systems (Figure 1). Of course, such servers have a narrow view of system activity, so they cannot distinguish legitimate users from clever impostors. But, from behind the thin storage interface, a self-securing storage server can actively look for suspicious behavior, retain an audit log of all storage requests, and prevent both destruction and undetectable tampering of stored data. The latter goals are achieved by retaining all versions of all data; instead of over-writing old data when a WRITE command is issued, the storage server "simply" creates a new version and keeps both. Together with the audit log, the server-retained versions represent a complete history of system activity from the storage system's point of view.

This paper describes how self-securing storage improves intrusion survival by safeguarding stored data and providing new information regarding storage activities before, during, and after the intrusion. Specifically, it focuses on three ways that self-securing storage contributes: helping to more quickly detect intrusions, providing easily accessible information for diagnosing intrusions, and simplifying and speeding up post-intrusion recovery.

First, a self-securing storage server can assist with intrusion detection by watching for suspi-

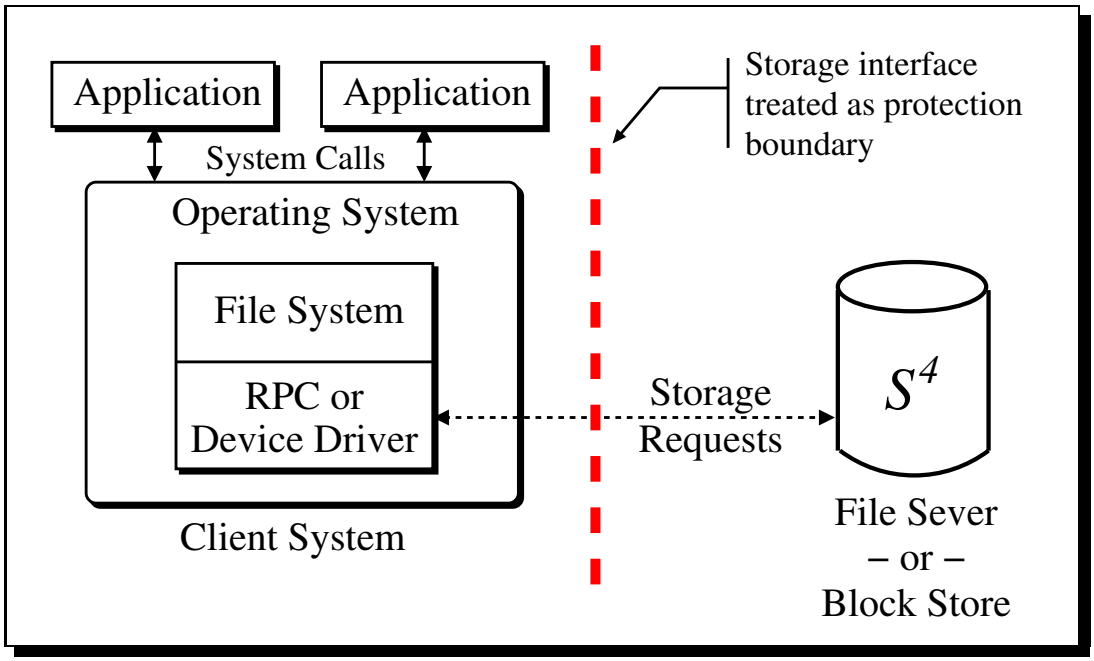

Figure 1: **Self-securing storage.** The storage interface provides a thin perimeter behind which a storage server can observe requests and safeguard data. Note that this same picture works for both block protocols, such as SCSI or IDE/ATA, and distributed file system protocols, such as NFS or CIFS. Thus, self-securing storage could be realized within many storage servers, including file servers, disk array controllers, and even disk drives.

cious storage activity. By design, a storage server sees all requests and stored data, so it can issue alerts about suspicious storage activity as it happens. Such *storage-based intrusion detection* can quickly and easily notice several common intruder actions, such as manipulating system utilities (e.g., to add backdoors) or tampering with audit log contents (e.g., to conceal evidence). Such activities are exposed to the storage system even when the client system's OS is compromised.

Second, after an intrusion has been detected and stopped, self-securing storage provides a wealth of information to security administrators who wish to analyze an intruder's actions. In current systems, little information is available for estimating damage (e.g., what information the intruder might have seen and what data was modified) and determining how he gained access. Because intruders can directly manipulate data and metadata in conventional systems, they can remove or obfuscate traces of such activity. With self-securing storage, intruders lose this ability—in fact, attempts to do these things become obvious red flags for intrusion detection and diagnosis efforts. Although technical challenges remain in performing such analyses, they will start with

much more information than forensic techniques can usually extract from current systems [32].

Third, self-securing storage can speed up and simplify the intrusion recovery process. In today's systems, full recovery usually involves reformatting, reinstalling the OS from scratch, and loading user data from back-up tapes. These steps are taken to remove backdoors or Trojan horses that may have been left behind by the intruder. Given server-maintained versions, on the other hand, an administrator can simply *copy-forward* the pre-intrusion state (both system binaries and user data) in a single step. Further, all work done by the user since the security breach remains in the *history pool*, allowing incremental (albeit potentially dangerous) recovery of important data.

In a previous paper [29], we introduced self-securing storage and evaluated its feasibility. We showed that, under a variety of workloads, a small fraction of the capacity of modern disk drives is sufficient to hold several weeks of complete storage history. With a prototype implementation, we also demonstrated that the performance overhead of keeping the complete history is small. This paper builds on the previous work by exploring the intrusion survival features of self-securing storage.

The remainder of this paper is organized as follows. Section 2 overviews self-securing storage and its feasibility. Section 3 describes how it enables storage-based intrusion detection. Section 4 describes how it contributes to post-hoc analysis of intrusions. Section 5 describes how it simplifies the post-intrusion recovery process. Section 6 discusses open challenges in realizing the full potential of self-securing storage. Section 7 approaches self-securing storage from the attacker's viewpoint by discussing how they can modify their behavior in response. Finally, Section 8 summarizes this paper's contributions.

2 Self-Securing Storage

The goal of self-securing storage is to enhance intrusion survivability by embedding security functionality into storage servers. Running in a separate system, this functionality remains in place even when client OSes or user accounts are compromised. That is, self-securing storage utilizes the storage interface as a protection boundary. From behind this boundary, a self-securing storage device can watch for suspicious activity, retain an audit log of storage requests, and prevent the destruction and undetectable tampering of stored data. Data survival is ensured by retaining

all previous versions of files for a guaranteed amount of time, called the *detection window*. For intrusions detected within this window of time, the complete storage history is available for intrusion diagnosis and recovery. This section describes the architecture in more detail, presents some security-related administration issues, and summarizes results from feasibility analyses and experiments with a prototype implementation.

2.1 Additional security perimeter

In conventional systems, the storage system acts as a slave to host operating systems, giving the OSes complete responsibility for the protection of stored data. As a result, compromised client systems can destroy data either directly or by masquerading as a legitimate user.

Instead of relying on the OS for protection, the storage device (whether a local disk or a remote server) should protect the data that it stores, independent of any protection that the client OS provides. The architecture of a computer system supports this idea since the storage system is a separate piece of hardware, running separate code from the OS, and the two entities communicate via a simple interface of well-defined commands (e.g., SCSI, NFS, CIFS). This places the storage device in an ideal position to provide protection for the system's data. Additionally, since it already stores the system's data and sees all requests, it has sufficient information to safeguard the data.

Modern storage systems also have significant resources that they can draw from to implement this protection. They have very large capacities that can be leveraged to increase security by maintaining multiple versions of stored data. These devices also have memory and processing power that can be utilized to examine and act upon storage requests independent of the current state of the host CPU and OS. These resources can be used to protect the system's data from unauthorized changes in a way that will not interfere with a valid, running system. To accomplish this, the storage device must view all storage operations with suspicion, yet not attempt to second-guess the intentions of the OS.

2.2 Data protection

To accomplish the goal of enhancing intrusion survivability, the storage device takes both an active and passive role. The storage device can use its processing power and memory to actively watch

the storage requests and issue alerts (or take corrective action) if it notices suspect activity. This active role can decrease the time it takes to detect intrusions because it adds a new point in the computer system from which to perform intrusion detection.

The self-securing storage device sees all storage requests and passively maintains an audit log of these requests. This audit log allows the system administrator to view the type, order, and potentially the originator of each request. The storage device maintains the audit log internally and exports it to authorized clients in a read-only manner. This allows the administrator (or authorized entity) to view the storage requests, but not change or modify the log.

In addition to the audit log of storage requests, the device keeps all versions of all data that are written. With each write or delete request, the current version of the file or directory is updated to reflect the new state, and the old version is saved in the device's *history pool* for a guaranteed amount of time. The old versions of files and directories cannot be modified or removed from the device until they expire from the *detection window*, an administrator-configured amount of time. Space on the storage device is logically divided into three sections: current data, history pool, and free space. The size of the history pool grows or shrinks, as necessary, to hold the data that has not yet expired from the detection window. The maintenance of the history information is handled entirely within the storage device, transparent to the rest of the system. This historical information can be used to view (and recover) any version of any file that still resides within the history pool, allowing quick post-intrusion recovery. Recovery from this historical data is done via a *copy forward* operation (as opposed to rolling back the state of the file or device). This prevents the destruction of intermediate versions, allowing incremental recovery and safe use of copy forward by normal users.

To implement the capabilities described above, a storage device must have a large capacity (common on modern disk drives) and the processing power necessary to maintain the history pool and audit log. These characteristics are present, and self-securing storage can be embedded in, smart disk drives [35], storage arrays, and file servers. The latter two (storage arrays and file servers) are in wide use today and require only the appropriate software to become self-securing storage devices. There is a spectrum of systems that could benefit from this self-securing technology, and any storage device that has the required processing capabilities and exists as a largely single-purpose device (to reduce exposure to intrusion of the storage device) would be a good

candidate.

Liu, et al. have proposed a method of isolating attackers, via versioning, at the file system level [15]. Their method relies upon an intrusion detection system to identify suspicious users in real-time. The file system forks the version trees to sandbox suspicious users until the administrator verifies the legitimacy of the users' actions. If done within the file server, this isolation method would be a form of self-securing storage, although it does endanger our goal of not interfering with a valid, running system, unless the intrusion detection mechanism yields no false positives. Specifically, since suspicious users modify different versions of files from regular users, it creates a difficult reintegration [18, 31] problem, should the updates be judged legitimate.

2.3 Administration issues

While self-securing storage is able to protect data in a nearly transparent way, it introduces several administrative issues. These additional complexities must be properly managed for self-securing storage to be effective.

Secure administrative control: Self-securing storage devices must have a secure, out-of-band interface for handling administrative tasks such as configuration and intrusion recovery. This interface must use strong authentication to ensure that requests cannot be forged by the OS or other processes on potentially compromised client systems. This precaution is necessary because the administrative interface (necessarily) permits commands that are destructive to data. For example, setting the length of the detection window is handled via this interface. If an intruder were able to change the length of the detection window, he could erase data from the history pool by shrinking it.

Although strong authentication is critical for this interface, confidentiality may be less so. Since history data can be viewed and the audit log read with this interface, some privacy concerns arise. However, most organizations still employ file systems that do not encrypt their normal traffic, thus it is unclear whether there is a need to obscure the data at this point.

Administrative alerts (for intrusion detection) pose another interesting problem. The channel that is used for this device-to-administrator communication must, at a minimum, be tamper-evident. This is necessary to prevent the intruder from successfully preventing the alerts from

reaching the administrator. The administrator may also wish to hide the fact that an alert was issued at all. She may even wish to hide the presence of the communication channel completely. These concerns play a significant role in the implementation of the administrative control channel.

The administrative interface may be implemented in a number of ways. For example, there could be a dedicated administrative terminal connection. For network-attached storage (e.g., file servers), one can use cryptography and well-protected administrative keys. For disks attached to host systems, a secure communication channel between a smart disk drive and a remote administration console can be created by cryptographically tunneling commands through the host's OS. No matter which interface is chosen, it must be the case that obtaining "superuser" privileges on a client computer system is not sufficient to gain administrative access to the storage device.

Setting the detection window: The storage device's detection window is the configurable parameter that determines the duration of history available for detecting, diagnosing, and recovering from intrusions. For this reason, it is desirable that the window be very long. On the other hand, more history requires more disk capacity. Our studies indicate that, with current disk technology, configuring the detection window for between one week and one month of history provides a reasonable balance for a large variety of workloads. The size of the resulting history pool is up to the total number of bytes written or deleted during the detection window. Cross-version differencing and compression can reduce the history pool size [27].

Denial of service: As with any system, an intruder can fill up the storage capacity and prevent forward progress by the system. Such denial-of-service is easy to detect once it occurs, but with self-securing storage, it requires careful administrative intervention. Specifically, restoring the system after such an attack requires true deletion of files or versions from the history pool in order to free space. Such deletion interferes with the basic goals of self-securing storage, so administrative privileges and care are required.

Additionally, conventional file systems use space quotas to prevent a single user from using a disproportionate amount of storage. This concept can be utilized for self-securing storage as well, but it would need to be modified to also contain a rate quota. This second quota would bound the average rate at which a user can write data. Such rate quotas are not likely to completely solve the problem. However, they could allow the device to detect a likely attack and throttle the offending user or client machine. This might provide sufficient time for an administrator to react before all

free space is consumed.

Privacy and inability to delete: Some users will object to being unable to delete files whenever they want, but allowing users to permanently delete their files (without waiting for the detection window to elapse) would open a path for intruders to destroy data. The chosen compromise between the need for security and the users' desire for privacy is to allow users to mark files as "unrecoverable." Such files would be retained in the device like normal, but could only be retrieved from the history pool by the administrator, not the user. This prevents an intruder (who could masquerade as a normal user) from recovering the file, but it still allows the administrator to completely recover after an intrusion. While some users may object to it being retained at all, it is often possible for an administrator to recover a deleted file in a conventional system as well [7]. Also, if the file was stored on the system for any reasonable length of time, it is likely that it was backed up for disaster recovery, making it unlikely that a normal user could force deletion [11].

2.4 Prototype system

To evaluate the concepts and feasibility of self-securing storage, a prototype system was designed and implemented. The prototype acts as a self-securing NFS server. It maintains an audit log and history pool as described above. The prototype runs as a user-level process on the Linux operating system. Unmodified client systems can use the server as a standard NFS version 2 [30] server, while the self-securing storage features are provided transparently.

Our previous work [29] evaluated the two main costs of self-securing storage: space consumption of the history pool and performance overhead of comprehensive versioning. To evaluate the space consumption, the write bandwidths of several file system workloads were examined to determine the length of time that the history can be retained in real environments. The workloads studied were a set of AFS servers [28], a single NFS server used by a dozen researchers [24], and local and remote file system activity of a collection of Windows NT computers used for scientific processing, development, and administrative tasks [34]. Based on the volume of write traffic in these environments, it is possible to keep between 10 and 90 days worth of history in 20% of a 50 GB disk drive.

The performance of our prototype system was compared to several other NFS implementations

and found to perform comparably on macro-benchmarks. More detailed benchmarking showed that maintaining the audit log results in a 1%–3% performance overhead, and maintaining the historical versions of data can be done with less than 10% performance overhead.

We view these small performance and capacity costs as acceptable, given the benefits described in the remainder of this paper.

3 Storage-based Intrusion Detection

Most intrusion detection systems concentrate on host-based and network-based detection of security compromises [2, 16, 19, 22]. Host-based detection, from its vantage point within the system, examines host-specific information (such as system calls [10]) for signs of intrusion or suspicious behavior. Network-based detection concerns itself with the network traffic to, and from, the hosts being watched. The storage system is an additional point in the computer system that can be used as a vantage point for intrusion detection.

Many intruders modify stored data. For instance, they may wish to cover any traces left by the system penetration, install Trojan programs to capture passwords, install a back door for future re-entry, or tamper with data files for the purpose of sabotage. These modifications will be visible to the storage system since the intruder is using and/or modifying the data that resides there. The storage system can watch this stream of requests for abnormalities and issue alerts.

Although the world view that a storage server sees is incomplete, two features combine to provide a well-positioned platform for enhancing intrusion detection efforts. First, since storage servers are independent of client OSes, they can continue to look for intrusions after the initial compromise, whereas a host-based system would normally be subverted. Second, since most computer systems depend on persistent storage to function, it will often be difficult for an intruder to avoid causing storage activity that can be captured and analyzed.

This section discusses four behaviors that could be detected via a storage-based intrusion detection system. It then discusses how one of these classes (watching for data or attribute modification) fits into our prototype system. The section concludes with the topic of how a storage device might respond to a suspicious event.

3.1 Detection capabilities

There are several forms of storage activity that a self-securing storage system might monitor for malicious behavior. These range from watching metadata operations, such as permission or timestamp modification, to verifying the internal consistency of data files. Of course, each comes with some cost in processing and memory resources. In configuring a self-securing storage system, one must balance detection efforts with performance costs for the particular operating environment.

This section describes several signals that storage-based intrusion detection can monitor: data and attribute modifications, update patterns, structural changes to content, and content changes and additions.

3.1.1 Data/attribute modification

As the simplest example, a self-securing storage server can watch for suspect, yet authorized, modifications to data or attributes. Such modifications can be detected on-the-fly, before the storage device processes each request. For suspect operations, the device can issue an alert immediately. It could even delay the update until the alert is processed. Such monitoring can easily detect changes to system executables, modification of static access permissions or ownership, and backwards changes to timestamps of sensitive files.

In conventional systems, similar functionality can be implemented via a checksumming utility (e.g., Tripwire [12]) that periodically compares the current storage state against a pre-generated reference database stored elsewhere. Self-securing storage goes beyond this current approach in four ways: (1) it allows real-time detection, (2) it avoids the complexity of constructing, maintaining, and protecting a reference database, (3) for local storage, it avoids relying on the client OS to do the checks, which a successful intruder could disable or subvert, and (4) it can notice short-term changes, such as timestamp rollback, which would not be seen if they occurred between two periodic checks. Although detection of file modification still requires creation of a rule set to designate allowed/prohibited modifications, the above four benefits are significant.

3.1.2 Update patterns

Some important files are manipulated by a single application or system component, in a well-defined way. For instance, system log files are usually treated as append-only during normal operation, 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 either the current log file or the previous log files is suspicious. Similar behavior is sometimes characteristic of system password files (the editing of **`/etc/passwd`** paired with editing of **`/etc/shadow`**) and even word processing files (their use of backup and temporary files).

While the log file behavior above can be watched to thwart an intruder trying to cover his tracks, even less tamper-revealing scenarios such as the word processor example are still beneficial. Any measure that serves to restrict the non-exposing actions of a system intruder increases the likelihood of the intruder making a mistake and exposing their presence. The examples above accomplish this by forcing the intruder to use the same temporary files and backup files in a manner that is consistent with the behavior of the actual application. This "raises the bar" for intruders wishing to remain undetected.

In general, many applications access storage in a well-defined manner. These patterns of access at the storage device will be a reflection of the application's requests. This presents an opportunity for more general anomaly detection based on how a given file is normally accessed. This could be done in a manner similar to watching system calls [10] or having rules regarding the expected behavior of applications [13]. Deviation from the normal pattern could indicate an intruder or malicious user attempting to subvert the normal method of accessing a given file. Anomaly detection within storage access patterns is an interesting topic for subsequent research.

3.1.3 Content integrity

Some important files have a well-defined structure. For instance, a UNIX system password file, such as **`/etc/passwd`**, consists of a set of records. Each record is delimited by a line-break, and the records each consist of exactly seven fields, colon separated. Since a storage system has access to all data that is written, it can verify that the data contained within write requests is consistent with

the rules governing the internal structure of the file. Of course, to perform this type of verification, the device must understand the format of each file it needs to verify. Thus, it could only be used for a small set of files—those files that are critical to the system's operation, but are allowed to change during normal operation.

For "simple" file structures, such as **`/etc/passwd`**, performing this verification with each write is a low-overhead operation. Data file formats, however, can be arbitrarily complex. Complex structures may necessitate accessing adjacent file blocks (other than those currently being written) to have sufficient context to verify the file's structure, causing a significant performance impact if it is performed during every write. This creates a performance vs. security tradeoff 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.

3.1.4 Suspicious content

As the content repository, the storage device can watch for the appearance of suspicious storage content. The most obvious suspicious content to look for is the appearance of a known virus, detectable via its signature. Several high-end storage servers (e.g., from EMC [17] and Network Appliance [21]) 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, and their use may indicate that an intruder is using the system as a storage repository. A large number of empty files or directories may indicate an attempt to exploit a race condition by inducing a time-consuming directory listing, search, or removal [3, 23].

3.2 Example: modification detection

As a concrete example, our prototype self-securing NFS server had been extended to support rule-based detection of suspect modifications. This work focuses on metadata changes and detecting data modification, enforcing a rule-set very similar to Tripwire [12]. The administrator adds rules via the secure administrative interface, and these rules are verified during the processing of each

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

Table 1: **Attribute list –** Rules can be established to watch these attributes in real-time on a file-by-file basis.

storage request. The current list of attributes that can be watched are shown in Table 1.

The administrator-supplied rules are of the form: {*pathname, attribute-list*}—designating which attributes should not change for the particular file. These rules are then added to the storage device's internal rule table. As a performance optimization, the aggregate set of rules that apply to a particular file are stored in that file's inode. This allows efficient verification of the rules since the inode is read prior to any file access, thus reducing the rule verification to a simple flag comparison. In addition to watching the file, all parent directories (up to the root) must be watched for name space operations that could affect the watched file (e.g., RENAME of a parent directory).

When a violation is detected, the global rule table is consulted and the full pathname of the file as well as the offending operation are sent to the administrator. In this situation, the global rule table must be consulted to reconstruct the pathname since multiple rules (due to hard or soft links) may apply to the same file.

3.3 Detection response

There are several actions that a self-securing storage device could perform upon detecting suspicious activity. Possible responses range from issuing an administrative alert to full-scale intervention. When choosing the proper response, the administrator must weigh the benefits of an active response against the inconvenience and potential damage caused by false alarms.

In addition to alerting the administrator, the device can take steps to minimize the potential damage by attempting to slow down the intruder. This is possible because the device can artificially increase the request latency and decrease the data throughput to the client or user that is suspected

of foul play. This can provide increased time for a more thorough response, and, while it can cause some annoyance in false alarm situations, it is unlikely to cause damage in most scenarios. The device could even deny a request entirely when it would violate one of the rules, although this action must be weighed carefully since a false alarm would likely cause applications to fail.

3.4 Additional benefits

Self-securing storage's characteristics can also help administrators configure rules and investigate alerts. When creating rules about storage activity for use in detection, the audit log and version history can be used to test new rules for false-positives. They can also be used to investigate alerts of suspicious behavior (i.e., check for supporting evidence within the history).

As well, because the history is retained, all forms of detection described above can be delayed until the device is idle. This would allow the device to avoid performance penalties for expensive checks by accepting a potentially longer detection latency.

Self-securing storage provides a new approach to intrusion detection that is complementary to current approaches, yet can be effective as a stand-alone system.

4 Diagnosis of Intrusions

Once an intrusion is detected and stopped, the administrator would like to understand what happened. There are several goals of post-intrusion diagnosis. These goals include determining how the intruder gained access to the system, when they gained access, and what they did once they got in.

In current systems, the administrator is poorly equipped to answer these questions because, once an intruder gains control of the computer system, no information can be trusted; the intruder has the ability to erase and obfuscate incriminating evidence. As a result, administrators usually perform only a cursory review of the post-intrusion system, hoping that the intruder overlooked some obviously incriminating evidence.

The administrator who wishes to dig deeper into the system in search of answers to these questions is faced with a daunting task. First, she must scour the free space of the storage system in search of disk blocks from deleted data and log files that have not yet been overwritten; simplifying

this task has been the focus of several forensic tool developers [9, 14, 20]. Second, and far more difficult, she must then piece together this incomplete information and form hypotheses about the details of the intrusion; this is, at best, a black art.

Self-securing storage has the ability to brighten this dismal picture. It makes available a large amount of information that was previously very difficult or impossible to obtain. This information provides several new diagnosis opportunities by highlighting system log file tampering, exposing modifications made by the intruder, and potentially allowing the capture of the intruder's exploit tools.

4.1 Post-mortem information

Self-securing storage maintains an audit log of all requests and keeps the old versions of files. Therefore, the administrator is no longer relegated to working with just the remaining fragments after an intrusion. The administrator now has the ability to view the sequence of storage events as well as the entire state of storage at any point in time during the intrusion. This history means that the pre-diagnosis forensics effort is no longer needed because the storage system retains this information automatically.

Because self-securing storage removes the need for the forensics effort, performing post-intrusion diagnosis is no longer an all-or-nothing proposition. The forensics effort that was previously required meant that only in extreme situations would an intrusion be investigated seriously. With all of the storage information immediately available, the administrator can spend an appropriate amount of effort interpreting post-mortem information, with near-zero invested in pre-diagnosis forensics.

Self-securing storage also takes intrusion diagnosis out of the critical path. Since the intrusion state is saved within the history data, diagnosis can be started after post-intrusion recovery. The administrator can return the system to operation quickly, and then utilize the history for actual diagnosis. This relieves some of the pressure that constrains the amount of time and effort that an administrator is able to put into diagnosis.

4.2 New diagnosis opportunities

Following an intrusion, the administrator is left with many questions. Her goal is to determine exactly what happened so that she can assess the damage and ensure that the intrusion is not repeated. Some of the many questions she seeks to answer are:

1. How did the intruder get in?
2. When did the intrusion start?
3. What tools were used?
4. What data was changed?
5. What data was seen?
6. What tainted information was propagated through the system?
7. Why was the system attacked?

Self-securing storage assists the administrator in answering the above questions by providing previously unavailable insight into storage activity. The rest of this section provides examples of information that is now available.

Highlighting audit log tampering: It is common for intruders to tamper with log files in an attempt to cover their tracks. With self-securing storage, not only is it possible to tell that the tampering occurred [25, 26], but the administrator can locate and retrieve the exact entries that were erased or modified. This means that when an intruder attempts to conceal their actions by doctoring log files, they are, in fact, doing just the opposite.

Capturing exploit tools: It is also common for intruders to load and run exploit tools locally after they initially gain entry. This can be done for several reasons, such as subsequent phases of a multi-stage intrusion, Trojan programs to capture passwords, or exploit tools that can be used to attack other machines. Normally written to the file system prior to being executed, these tools are automatically captured and preserved by self-securing storage. The capture of such exploit tools and "root kits" makes it easier to find the intruder's point of entry into the system and the weakness(es) they exploited [33].

Exposing modifications: In addition to capturing exploit tools, any changes to system files and executables are obvious based on the storage device's audit log. This allows the administrator to see the scope of damage to the OS and other sensitive system software. The storage device tracks and "exposes" even legitimate software updates made by the administrator unless an out of band method is used to coordinate those specific modifications (e.g., via the secured administrative interface). Additionally, the device applies its own timestamps to modifications, in addition to those supplied by client systems. This allows a clear picture of the sequence of events even when the intruder may have manipulated the creation, modification, or access times of files [8].

Recording reads: Since the server's audit log records all READ and WRITE operations, the administrator can estimate the real damage that may have been done by the intruder. For example, the storage log allows one to bound the set of files read by a system and the likelihood that the intruder has read specific confidential files. Additionally, the log can assist the administrator in determining whether intruder-modified files were read by legitimate users. This allows her to gauge the potential spread of mis-information, planted by the intruder for sabotage purposes. It is important to note that this is only an approximation since the client system's cache can obscure some storage requests. It does, however, provide a way of gauging intrusion damage that previously could not be measured at all. This even provides information for inferring the intruder's motivation for attacking the system. With a more complete view of the intruder's actions, the administrator has a greater chance of determining whether the intent was espionage, sabotage, or merely "entertainment."

Self-securing storage provides a new window into the scope of damage and the intentions of a digital intruder. This information can be invaluable in determining the impact of the compromise, preventing future intrusions, and catching those responsible. Clearly, much future research and experience will be needed to create post-mortem diagnosis tools that exploit the new wealth of information provided by self-securing storage.

5 Recovery from Intrusions

Intrusion detection and post-intrusion diagnosis are parts of a good computer security strategy. An efficient and effective plan for recovery, however, is a necessity. Maintaining well-administered

and up-to-date systems will minimize the occurrence of intrusions, but they will inevitably happen, so it is critical that recovery be efficient and thorough.

In conventional systems, intrusion recovery is difficult and time-consuming, in terms of both system down-time and administrator time. It requires a significant amount of the administrator's time because there are many, error-prone steps involved in returning a compromised computer system to a safe state. Self-securing storage, with its history information, facilitates this task.

5.1 Restoration of pre-intrusion state

All storage in a conventional system is suspect after an intrusion has occurred. As a result, full recovery necessitates wiping all information via a reformat of the storage device, re-installing the operating system from its distribution media, and restoring users' data from the most recent pre-intrusion backup. Shortcutting these steps can result in tainted information remaining in the system, yet following these steps results in significant down-time and inconvenience to users.

Self-securing storage addresses this unfortunate situation in several ways. First, all pre-intrusion state is preserved on the device. Therefore, once diagnosis yields an approximate time for the intrusion, the restore described above can be a single step for the administrator (issuing the *copy forward* command to bring forward the system state from before the intrusion). Second, the granularity of the restoration is not limited to the most recent backup; any or all files can be restored from arbitrarily close to the time of the intrusion. Third, restoration in self-securing storage is non-destructive. The administrator can quickly return the system to a safe state so that users may utilize the computer system, while preserving all storage history (including any intrusion evidence therein) in the storage server. The administrator may then perform detailed diagnosis at her leisure.

5.2 Performance of restoration

To investigate the time required to return a system to operation, we gathered traces of all NFS activity to our local NFS server and replayed them against our prototype system. We then measured the amount of time required to copy forward the entire state of the device from various points in the past.

The prototype self-securing NFS server used for this experiment was a dual 600 MHz Pen-

Days	Files	MB	Time w/ rsync (s)	Time w/ audit log (s)
1	7	7.3	3208	26.4
2	349	34.4	3311	139.0
3	359	66.8	3324	167.5
4	361	76.2	3350	198.4
5	362	66.8	3329	208.3
6	362	66.8	3325	212.6

Table 2: **Recovery statistics –** This table summarizes the results of completely restoring the state of the NFS server as a function of the detection latency. The times shown are based on using either **`rsync`** or the data in the device's audit log to create the list of files that must be copied forward. The large difference is due to **`rsync`** executing a STAT on every file at both the current time and the recovery time—a total of approximately 545,000 calls.

tium III system, running RedHat Linux 6.1 using kernel version 2.2.20. The traces were collected from an NFS version 2 server, running Linux 2.2. The traced NFS server supports the research efforts of approximately 30 graduate students, faculty and staff. The server contained approximately 66.5 GB of capacity of which 33.5 GB was consumed by $272,521$ files. The workload on the server was mainly generated by code development and word processing activities of the supported users.

To evaluate both the number of files and bytes of data (due to legitimate use) that the administrator may have to recover, an initial snapshot of the file system was copied onto the prototype system, followed by the first day's worth of activity. The state of the storage system was returned to the original condition by copying forward the snapshot state. During this process, the time to recover the initial state, the number of affected files, and the total size of the affected files were recorded. This experiment was conducted for each of one through six days of activity, each time beginning with the original snapshot.

Two different methods were tested for selecting the files to recover. The first method utilized the **`rsync`** [4] program to synchronize the current system state with the pre-intrusion copy of data. The **`rsync`** application was configured to select files based on their attributes, causing it to retrieve the attributes of all files on the device at both the current time and at the recovery time, leading to a large overhead for just building the list of files to recover. Creating this initial list required 3184 seconds—and by only checking the attributes, this method is vulnerable to manipulation of the file modification times. The second method used to create this list of files to recover is by using the device's audit log. This second method is much faster, requiring only 71 seconds to determine the

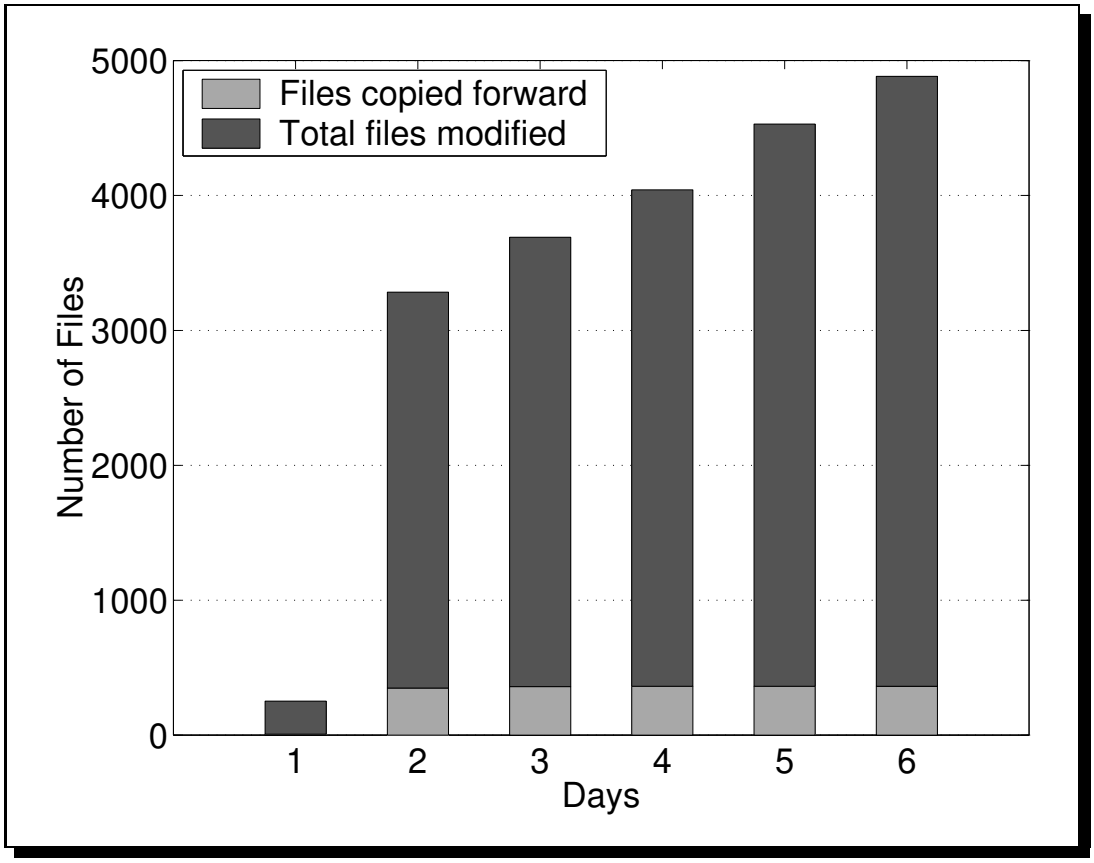

Figure 2: **Files modified –** Shows the total number of user files that must be restored to return to the pre-intrusion state as a function of the detection latency.

list of files for the complete six days worth of changes. In addition to being faster, this second method is not susceptible to timestamp manipulation. The results are summarized in Table 2.

Based on Figure 2, it is obvious that very few files need to be restored relative to the total number of files that were modified. This is because a large number of files are transient. Files that were completely created then deleted between the recovery time and the current time need not be copied forward.

To project the amount of data that would need to be recovered for longer detection latencies, we examined a snapshot of over 10000 file systems from desktop computers at Microsoft Corporation [6]. The snapshot contains a listing of all files, their size, and modification times for each of the systems. Based on the last-modified time of the files, we can project the number of files that would need to be restored as a function of the detection latency. The results are shown in Figure 3.

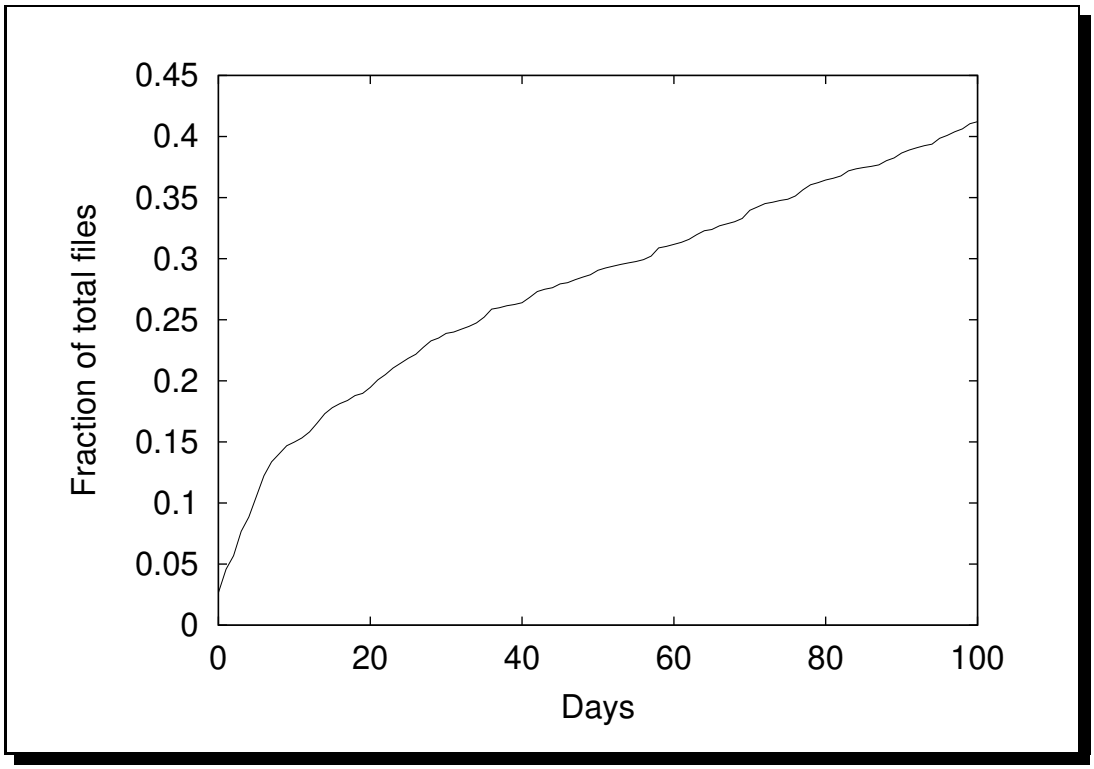

Figure 3: **Fraction of files modified –** Shows the fraction of files that were modified in the examined Microsoft systems, as a function of the number of days. The average number of files per file system was approximately $13,000$.

The results are shown as a fraction of the total number of files stored.

Based on the playback data, we can estimate the amount of time that would be required to restore a "typical" one of these desktop systems. Looking at only the time required to restore the files, the copy forward executed at a rate of 2.55 files per second on our prototype system. This means that one week worth of changes could be copied forward in: $\frac{.134 \cdot 13000}{2.55} = 683$ seconds. Likewise, the system could be restored to its state as of one month ago in: $\frac{.239 \cdot 13000}{2.55} = 1219$ seconds. We believe that the "in-time" performance of out prototype can be improved, significantly increasing the rate of recovery beyond 2.55 files per second.

5.3 Preservation of user data

In addition to removing the intruder and restoring the system to a safe state, an administrator is often under considerable pressure to retain a recent version of users' work—a version from after the system was compromised. In conventional systems, the mechanics of doing this involves

finding separate media on which to temporarily store the users' data, then performing the normal reformat, reinstall, restore, and finally restoring this recent version of the users' data. There also exists the challenge of determining whether this data is safe to keep at all since the intruder could have modified it.

Self-securing storage obviates the mechanical steps by automatically retaining the users' recent work. While this provides the opportunity for the administrator to easily restore it, there is still some question about the authenticity of this data. The question of intruder tampering, while still very difficult to answer, is one step closer due to the availability of the storage system's version history. The version history can show the sequence of modifications to the data as well as, in the case of a network-attached storage device, the client and user who made the modifications. This provides additional information for tampering investigations, though they will remain very difficult.

Additionally, in current systems, the administrator is left with only two choices when restoring a given file: use the current, potentially-tainted data or use the most recent backup, losing all intermediate work. Self-securing storage provides the opportunity to restore any version in between those two extremes as well. This can be beneficial in situations where the data was (or might have been) modified by the intruder; the administrator is able to restore the version from just prior to the tampering and need not lose all changes. Careful validation of user data is still too time-consuming to use on all data, but it can be performed on an as-needed basis for important data files.

5.4 Application-specific recovery

With only the above information, it is possible to accomplish recovery of data in a reasonable fashion. However, without examining the contents of files, it is difficult to determine what application level change was made.

Given a tool that understands the contents of a file, it would be possible to, in some situations, untangle changes made by a legitimate user and an intruder within a single file. For instance, the intruder may have planted a macro virus within a word processing document. A tool that understands the format of such files could remove the virus while leaving the other data intact. Modern virus detectors are able to handle this specific case, but in general, this sounds like a

daunting task—creating application-specific recovery tools for data files. However, a small number of applications create most files in a given system. Therefore, a few such utilities will cover most of the data.

An additional use for these application-specific recovery tools is for bootstrapping recovery and diagnosis of complex programs, such as databases. Consider a database application that maintains a log of operations, capable of undoing or redoing its operations. A recovery tool could potentially validate the contents of the database against the database's log, ensuring that any changes made to the database contents were made through the proper interface. The database's built-in validation and auditing could then be used to dig deeper into the changes, knowing that the database log information is consistent with the actual database contents. Ammann, et al. have already approached the problem of removing undesirable but committed transactions from databases [1], assuming that any malicious transactions have been inserted via the normal database interfaces (as opposed to accessing raw storage).

6 Discussion

This section discusses some remaining issues involved in using self-securing storage to detect, diagnose, and recover from intrusions.

Audit log accuracy: The information stored in the device's audit log is derived from the information contained in the storage protocol. As a result, information about the requester's identity is only as good as the guarantees provided by the storage protocol itself. For instance, if protocol requests can be forged, these forged requests will be added to the audit log as seen by the storage server. When present, this limitation is inherent in the choice of storage configuration. This is only relevant to network-attached self-securing storage devices. Locally-attached disks receive all of their commands from a single OS, which does not provide any user-specific information in storage requests.

Tracking tainted data: The audit log maintained on the device shows not only the files that were written after an intrusion, but also which clients or users subsequently read (potentially) intruder-tainted data. While it is possible to consider subsequent writes (of different files) by those clients and users as suspect as well, the possible set of tainted data is likely to grow very large.

Also, the probability that a file is affected via a specific WRITE decreases with each iteration. Additionally, it is not possible to completely track tainted data, since it may have been transmitted to other files in the system via external methods (e.g., printouts or word-of-mouth among users).

Additional OS information: The addition of a small number of additional fields inside each storage request would greatly increase the utility of the audit log. For instance, if each request were tagged with the process ID and name of the process that is making the request, it is much easier to determine (during both detection and diagnosis) whether a given access was benign. Additional, useful information would be an indication of the purpose of a READ operation. In current file systems, a READ operation for data retrieval looks nearly identical to a read for the purpose of executing.

Client caches: File system caches on client systems obscure traffic that would otherwise be seen by the storage server. The caches effectively act as a filter causing the audit log to only have a partial view of the OS's storage activity. For example, read caches can obscure the propagation of intruder-tainted data since it is likely to be in the client's cache. This danger is smaller in some file systems, such as NFS version 2, that perform aggressive "freshness" checks prior to returning the cached contents of a file. This freshness check is visible to the storage system, and the window of vulnerability during which the freshness check is not necessary is small (a few seconds). Client-side write caches are also a problem since short-lived files that are written, read, and deleted quickly may never be transmitted to the storage device. In this case, the existence of the file may never be known to the storage system. It would not have any associated entries in the audit log nor versions in the history pool. This presents a larger problem for diagnosis, since it means that temporary files may be lost.

7 Hiding From Self-Securing Storage: The Game Continues

Self-securing storage has the potential to expose intruders and their actions by explicitly watching at a new point in the system. We expect it to work very well initially, and less so as intruders learn about self-securing storage and its capabilities. That is, we expect clever intruders (and those that borrow their tools) to modify their behavior in an attempt to mitigate the benefits that self-securing storage provides. This section explores some potential actions that intruders might take

in an attempt to avoid detection or thwart attempts at diagnosis and recovery. We note that those in the white hats gain ground when intruders must change tactics and avoid convenient actions. Thus, we also discuss some ramifications of these new behaviors on the intruders themselves.

Minimize log file evidence: In a system protected by self-securing storage, it is not possible for an intruder to tamper with log files once they are written. Using the self-securing storage device for system log files functions much like a remote logging server. If intruders wish to hide their presence and actions, this limits the types of attacks and exploits that they can use in their initial compromise of the system. Once they gain control of the system, however, they can manipulate the logging facilities to prevent information from entering the log. This can be made a non-trivial activity. If they wish to avoid detection, they must filter log entries that they generate (as a result of their malicious activities), but they must allow normal entries to continue to be logged. Filtering too much or too little will be detectable. Additionally, it may be possible to correlate system log entries and storage log entries, further complicating the filtering process.

Use RAM-based file systems: One way of preventing capture of exploit tools and utilities is to store them in a RAM disk instead of the usual file system that is kept on self-securing storage. The problem with this approach is that the data that is written is not persistent, and a simple reboot of the machine will wipe it out. While this will erase the evidence, it will also erase any backdoor or Trojan executables that were left behind.

Manipulate memory images: Since overwriting of system executables is easily detectable, one way to create a backdoor version of a (long running) program would be to directly modify it's memory image. This would leave no traces on the storage system, but is likely to be difficult to do. Additionally, the modifications could be erased by restarting the application [5].

Tamper slowly: If the intruder is able to avoid having his action(s) detected until after the detection window elapses, self-securing storage will be of little help in diagnosing and repairing the damage. An intruder that is willing to make small changes over a large amount of time can increase his chances of success. The problem that this creates for an attacker is that it takes a large amount of time to tamper with a significant amount of data.

Redirect file system requests: Once the intruder is able to compromise the OS, he can manipulate the file system code in the kernel in such a way that requests for one file are redirected to another. For example, he can use this to redirect requests from a legitimate system executable

to a Trojan version of the same program. While self-securing storage would not view this as a change to the system executable (and not issue an alert), the Trojan executable will be captured, and the true executable remains intact. Again, since this strategy does not modify any of the true executables, a reboot will solve the problem.

Encrypt tools: To prevent capture of exploit tools, the intruder can use tools that are written to disk in an encrypted form and decrypted just prior to execution (in a similar manner to some viruses). The decryption key can be held in memory of the affected system (at a well-known location). As long as the key is present and correct, the tools can be used, but by removing or changing the key, it would not be possible to recover the true contents of the exploit tools stored on the compromised system. While this prevents capture, the key is necessarily stored in a volatile location, hence a restart of the system will clear it, rendering the tools useless.

Use a network loader: The attacker could utilize a network loading utility that would read an executable directly into memory and execute it. This avoids the file system all together, but is, again, not persistent.

These examples show that it is possible to dodge some aspects of self-securing storage. However, they also show that doing so requires a level of expertise and effort not necessary for conventional systems.

8 Conclusion

This paper describes how storage servers can be used as effective tools for intrusion survival. Self-securing storage contributes to the system administrator's ability to effectively deal with digital intrusions by providing a new location for intrusion detection, preserving evidence to help with diagnosis, and safeguarding data to allow rapid, effective post-intrusion recovery.

References

[1] Paul Ammann, Sushil Jajodia, and Peng Liu. Rewriting histories: recovering from undesirable committed transactions. *Distributed and Parallel Databases*, **8**(1):7–40. Kluwer Academic Publishers, January 2000.

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

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

[4] Randal C. Burns. *Differential compression: a generalized solution for binary files*. Masters thesis. University of California at Santa Cruz, December 1996.

[5] Miguel Castro and Barbara Liskov. Proactive recovery in a Byzantine-fault-tolerant system. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 273–287. USENIX Association, 2000.

[6] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Atlanta, GA, 1–4 May 1999). Published as *ACM SIGMETRICS Performance Evaluation Review*, **27**(1):59–70. ACM Press, 1999.

[7] Dan Farmer. Bring out your dead: the ins and outs of data recovery. *Dr. Dobb's Journal*, **26**(1):102–108, January 2001.

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

[9] Dan Farmer and Wietse Venema. Forensic computer analysis: an introduction. *Dr. Dobb's Journal*, **25**(9):70–75, September 2000.

[10] Stephanie Forrest, Setven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for UNIX processes. *IEEE Symposium on Security and Privacy* (Oakland, CA, 6–8 May 1996), pages 120–128. IEEE, 1996.

[11] Norman C. Hutchinson, Stephen Manley, Mike Federwisch, Guy Harris, Dave Hitz, Steven Kleiman, and Sean O'Malley. Logical vs. physical file system backup. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22–25 February 1999), pages 239–249. ACM, Winter 1998.

[12] Gene H. Kim and Eugene H. Spafford. The design and implementation of Tripwire: a file system integrity checker. *Conference on Computer and Communications Security* (Fairfax, VA, 2–4 November 1994), pages 18–29, 1994.

[13] Calvin Ko, Manfred Ruschitzka, and Karl Levitt. Execution monitoring of security-critical programs in distributed systems: a specification-based approach. *IEEE Symposium on Security and Privacy* (Oakland, CA, 04–07 May 1997), pages 175–187. IEEE, 1997.

[14] Warren G. Kruse II and Jay G. Heiser. *Computer forensics: incident response essentials.* Addison-Wesley, 2002.

[15] Peng Liu, Sushil Jajodia, and Catherine D. McCollum. Intrusion confinement by isolation in information systems. *IFIP Working Conference on Database Security* (Seattle, WA, 25–28 July 1999), pages 3–18, 2000.

[16] Teresa F. Lunt and R. Jagannathan. A prototype real-time intrusion-detection expert system. *IEEE Symposium on Security and Privacy* (Oakland, CA, 18–21 April 1988), pages 59–66. IEEE, 1988.

[17] McAfee NetShield for Celerra. EMC Corporation. http://www.emc.com/pdf/partnersalliances/einfo/McAfee_netshield.pdf.

[18] Lily B. Mummert. *Exploiting weak connectivity in a distributed file system.* PhD thesis, published as CMU–CS–96–195. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, December 1996.

[19] NFR Security. http://www.nfr.net/.

[20] Stephen Northcutt, Mark Cooper, Matt Fearnow, and Karen Frederick. *Intrusion Signatures and Analysis*. New Riders, 2001.

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

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

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

[24] Douglas J. Santry, Michael J. Feeley, and Norman C. Hutchinson. Elephant: the file system that never forgets. *Hot Topics in Operating Systems* (Rio Rico, AZ, 29–30 March 1992). IEEE Computer Society, 1999.

[25] Bruce Schneier and John Kelsey. Cryptographic support for secure logs on untrusted machines. *USENIX Security Symposium* (San Antonio, TX, 26–29 January 1998), pages 53–62. USENIX Association, 1998.

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

[27] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. *Metadata efficiency in a comprehensive versioning file system.* Technical report CMU–CS–02–145. Carnegie Mellon University, 2002.

[28] M. Spasojevic and M. Satyanarayanan. An empirical-study of a wide-area distributed file system. *ACM Transactions on Computer Systems*, **14**(2):200–222. ACM Press, May 1996.

[29] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: protecting data in compromised systems. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 165–180. USENIX Association, 2000.

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

[31] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO). Published as *Operating Systems Review*, **29**(5), 3–6 December 1995.

[32] Wietse Venema. File recovery techniques. *Dr. Dobbs's Journal*, **25**(12):74–80, December 2000.

[33] Wietse Venema. Strangers in the night. *Dr. Dobbs Journal*, **25**(11):82–88, November 2000.

[34] Werner Vogels. File system usage in Windows NT 4.0. *ACM Symposium on Operating System Principles* (Kiawah Island Resort, Charleston, South Carolina, 12–15 December 1999). Published as *Operating System Review*, **33**(5):93–109. ACM, December 1999.

[35] Randolph Y. Wang, David A. Patterson, and Thomas E. Anderson. Virtual log based file systems for a programmable disk. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22–25 February 1999), pages 29–43. ACM, 1999.