Cluster Security with NVisionCC: The Forseti Distributed File Integrity Checker

Adam J. Lee[†‡], Gregory A. Koenig[†‡], and William Yurcik[†]
[†]National Center for Supercomputing Applications
[‡]Department of Computer Science
University of Illinois at Urbana-Champaign
{adamlee, koenig, byurcik}@ncsa.uiuc.edu

Abstract

Attackers who are able to compromise a single node in a high performance computing cluster can use that node as a launch point for a number of malicious actions. In many cases, the password used to log into a single node can be used to access a large number of nodes in the system, allowing the attacker to utilize the vast computing and storage capabilities of the compromised cluster to sniff network traffic, carry out brute-force password cracking, launch distributed denial of service attacks, or serve illegal digital content. Often, these types of attackers modify important system files to collect passwords to other accounts, disable certain logging facilities, or create back-doors into the system.

In this paper, we present Forseti, a distributed file integrity checker designed specifically for the high performance computing cluster environment. Forseti was designed to address the shortcomings exhibited by existing host-based intrusion detection systems when used in the cluster environment and to provide a means of detecting changes to critical system files made by root-level adversaries. We discuss the design and implementation of the Forseti system, present a security analysis of Forseti, examine the performance of the system, and explore how Forseti can be used in concert with other security monitoring techniques to enhance the security of the HPC cluster environment.

1 Introduction

Over the course of the last two years, the security of large-scale commodity clusters has become a topic of increasing research and practical interest. Attackers who are able to compromise a single node in one of these clusters can use that node as a launch point for a number of malicious actions. In many cases, the password used to log into a single node can be used to access a large number of nodes in the system, allowing the attacker to utilize the vast computing and storage capabilities of the compromised cluster to carry out brute-force password cracking, launch distributed denial of service attacks, or serve illegal digital content. Additionally, an attacker can listen to network traffic for other users to log into their compromised nodes in hopes of learning passwords for other accounts on these systems.

The recent rise in popularity of grid computing has exacerbated this problem by increasing the amount of damage that can be caused with a single compromised account. Often times, a password used to log into one cluster node can be used not only to log into other nodes in the same cluster, but also to log into other clusters located throughout the computational grid. This implies that the compromise of a single account on a single cluster node could give an attacker the ability to access many thousands of geographically distributed cluster nodes. The high utilization of clusters participating in grid computing systems allows an attacker sniffing for passwords on a compromised node to gain access to an extremely high number of other user accounts. This technique is not conjecture and was, in fact, used during the TeraGrid compromises that affected a large number of clusters world-wide during the Spring of 2004 [11]. During this time period, attackers installed Trojan-horse `sshd` processes designed to capture the passwords of users logging into compromised nodes so that these passwords could then be used to compromise other nodes throughout the grid.

Host-based file integrity checkers (*e.g.*, [2, 4, 6, 10, 12, 15, 18, 24]) have been used to help detect these types of attacks on hosts in enterprise computing systems for a number of years. These tools are activated periodically (usually by means of a scheduler such as `cron`), to check various properties of critical system files, including access permissions, timestamps, and content hashes. These properties are compared to reference values stored in a read-only database; files whose values differ from the reference cause an alert to be raised, such as an entry in the system log or an email being sent to the system administrator. Though these systems have worked well in enterprise computing, they suffer several shortcomings when used in large-scale commodity clusters. Installing and managing these tools on the hundreds or thousands of nodes in a large cluster is a time-consuming process involving changes to each node when system software is upgraded. Additionally, if the signature database is stored on each node, attackers can relatively easily change the the reference copy of the file information. Lastly, effectively logging the access violations can pose a problem if an attacker has gained root access to the machine, as they could disrupt the logging mechanism and easily cover their tracks.

To address the shortcomings of host-based file integrity checkers while retaining their strengths, we have designed Forseti, a distributed file integrity checker that meets the unique needs of the HPC cluster environment. In our tool, the file integrity database is separated from the nodes to be monitored and by leveraging the emergent properties of the HPC cluster computing environment [25], we can significantly reduce the size of this database. Cluster nodes cannot access the database, meaning that an attacker who has compromised some number of nodes in the system cannot alter the stored file signatures. Our system relies on a collector node (not necessarily part of the cluster) to carry out the integrity scans by making remote connections to the nodes in the system, pushing the necessary software to the system, and carrying out the scan. In this way, Forseti has no software footprint on the nodes to be monitored and adding a new node to be scanned is as simple as updating the database to inform the collector node of this change. This system is robust to individual node compromises and, though not foolproof, is significantly harder to defeat than other host-based systems performing similar function; had Forseti been deployed during the Spring of 2004, the TeraGrid attacks could have been detected and stopped much earlier than they were.

The rest of this paper is organized as follows. In Section 2 we describe our threat model and discuss the conditions in which integrity monitoring can aid in the security monitoring of large-scale commodity clusters. Section 3 briefly discusses the design of NVisionCC, the cluster monitoring tool with which Forseti interacts. In Section 4, we discuss the implementation of our distributed integrity checker and detail its interactions with cluster nodes and NVisionCC. We revisit our threat model and discuss the performance of this scanner in Section 5, and address related work in Section 6. We then present our conclusions and directions for future work in Section 7.

2 Threat Model

Throughout this paper, we focus on the use of file integrity checking as a single tool at the disposal of system administrators tasked with monitoring the security state of large-scale commodity clusters. As such, we do not advocate the use of Forseti to detect all forms of cluster intrusion, but rather to locate only a particular class of attacks. Any anomalies detected through the use of file integrity checking are meant to be cross-referenced and correlated with the findings of other security monitoring techniques—a task which our comprehensive cluster monitoring tool, NVisionCC, is designed to carry out (*e.g.*, by examining running system processes [13], open network ports [14], or detecting privilege escalations [20]).

File integrity checking allows system administrators to be notified when the contents or permissions of key system files is changed. Many times, attackers who compromise nodes in a cluster do so with the intent of carrying out password gathering attacks to escalate their privilege set and gain access to other nodes or clusters. These attacks are usually carried out through the use of rootkits or Trojan-horse system utilities. The tool presented in this paper focuses on attacks executed by a root-level adversary that involve the modification of system files. In particular, we assume our adversary has the ability to insert, reorder, and replay messages sent on the network, arbitrarily modify files stored on the compromised nodes, and perform any number of administrative tasks (*e.g.*, starting and stopping system services) on the nodes which have been compromised. Throughout this paper, we assume that any individual node in the cluster can be compromised and used to carry out arbitrary tasks on behalf of an attacker. We relax this assumption and discuss the use of Forseti in single-image clusters (*e.g.*, Clustermatic systems) in Section 5.4.

3 Design

In this section, we comment briefly on the emergent properties of large-scale commodity clusters and describe the ways in which these properties can be used

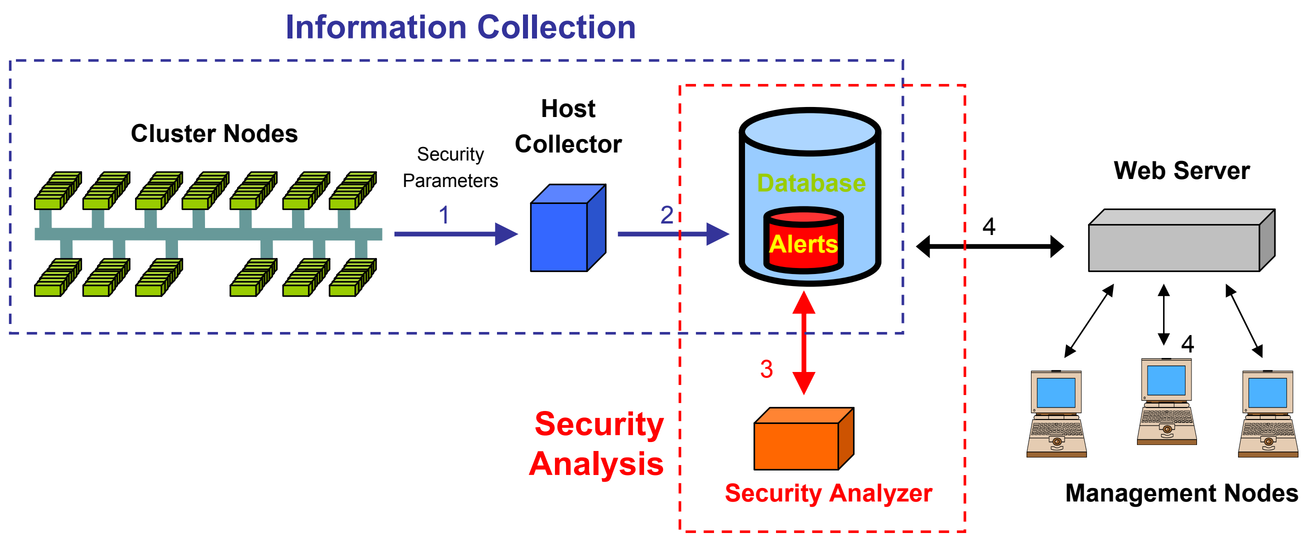

Figure 1. NVisionCC system architecture

during security monitoring. We then discuss the architecture of NVisionCC, a cluster security monitoring system designed to leverage these emergent properties. Lastly, we overview the architecture of the Forseti distributed file integrity checker and discuss how it fits into the NVisionCC monitoring framework.

3.1 Emergent Properties

Emergent properties are characteristics of a system that are only visible when the system is viewed as a whole, rather than as simply the sum of its parts. In large-scale commodity clusters, many individual nodes are collected together to form a single logical computing environment. When this larger computing environment is examined as a whole, a number of useful properties can be observed; the tool presented in this paper leverages one such observation.

The predominant emergent property present in large-scale commodity clusters is the ability to partition the hundreds or thousands of individual nodes into a small number of equivalence classes. Nodes within the same equivalence class are configured similarly and exhibit like behaviors. For instance, nodes within the same equivalence class are typically built from the same system images and will thus be configured to run the same sets of system services, contain the same system binaries, and exhibit similar communication patterns with other nodes in the system. A list of commonly observed equivalence classes is provided in Table 1.

In a previous work [25], we showed that emergent properties can be leveraged to increase security situational awareness in the cluster environment. Rather than being concerned with managing thousands of possible node configurations, as in an enterprise computing environment, the emergent structure of large-scale commodity clusters allows us to focus only on whether the configuration and behavior of a particular node is consistent with other nodes in its equivalence class (or classes). We next describe how our cluster monitoring tool, NVisionCC, leverages the emergent structure of the cluster environment to provide a framework for effective security monitoring.

3.2 NVisionCC

NVisionCC is an extension to the Clumon [9] cluster monitoring package which has been enhanced to monitor the security-state of large-scale commodity clusters by leveraging their emergent properties [26]. As shown in Figure 1, the NVisionCC architecture consists of four distinct parts: collection nodes, security analyzers, a database, and a web server. Security monitoring within the NVisionCC framework takes place in three phases: *information collection*, *security analysis*, and *visualization*.

During the *information collection* phase, the NVisionCC plug-ins installed on each collector node gather information from the nodes in the cluster through the use of either push or pull mechanisms. Once collected, this information is logged to the NVisionCC database where it can be accessed by other components in the

Equivalence Class	Description
Head	Head nodes provide an interface for end users to access the cluster
Compute	Nodes in this class carry out computations on behalf of scheduled jobs
Storage	Storage nodes provide and interface to the cluster's storage subsystems
Management	These nodes host the cluster management software
Monitor	Nodes in this class may host cluster performance monitoring software

Table 1. Common node equivalence classes

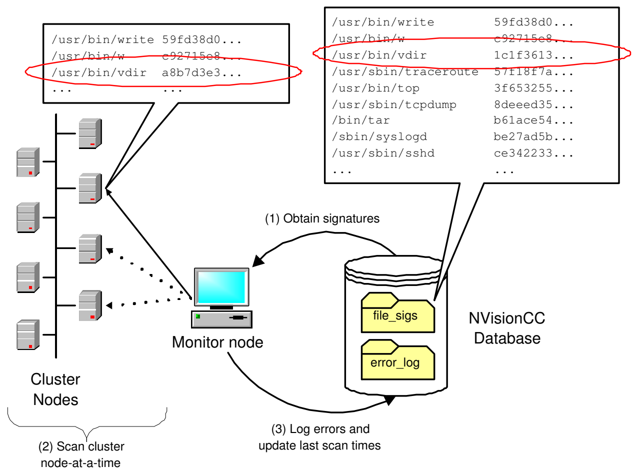
/usr/bin/write 59fd38d0...
~~/usr/bin/w c92715e8...~~
/usr/bin/vdir a8b7d3e3...
... ...
/usr/bin/write 59fd38d0...
~~/usr/bin/w c92715e8...~~
/usr/bin/vdir 1c1f3613...
/usr/sbin/traceroute 57f18f7a...
/usr/bin/top 3f653255...
/usr/sbin/tcpdump 8deeed35...
/bin/tar b61ace54...
/sbin/syslogd be27ad5b...
/usr/sbin/sshd ce342233...
... ...
(1) Obtain signatures
file_sigs
error_log
NVisionCC Database
Monitor node
Cluster Nodes
(2) Scan cluster node-at-a-time
(3) Log errors and update last scan times

Figure 2. Forseti architecture diagram

system. In addition to containing the observations gathered during the information collection phase, the NVisionCC database is also used to store expected-value profiles for each of the data items gathered by its plug-ins. Due to the emergent structure of the cluster environment, these profiles can usually be stored on a per-equivalence-class basis, rather than a per-node basis, thereby greatly reducing the effort required to manage the cluster node profiles.

During *security analysis*, one or more security analyzer nodes parse the information stored in the NVisionCC database to search for unusual patterns. The information returned by the collector nodes is cross-referenced with the appropriate equivalence-class profiles stored in the database and any deviations from the expected are logged to an alerts table in the database. Though the information collection and security analysis phases of NVisionCC's operation are logically two separate processes, it is sometimes the case that a particular plug-in executes these two logical processes using a single system process.

The alerts table in the database contains various data about the possible security violations detected by the NVisionCC plug-ins, including the time of the observation, the type of discrepancy detected, and the severity of the deviation. The NVisionCC front-end is a PHP-enabled web-page that allows the current status of the cluster, including all current alerts, to be *visualized* on a single screen. Security administrators can access this interface from their workstations to assess the security-state of their cluster and take reactive measures to the security alerts that are raised by NVisionCC.

3.3 Forseti

We now highlight how the Forseti distributed file integrity checker fits into the NVisionCC monitoring paradigm. Figure 2 shows an overview of the Forseti architecture. Essentially, the components of the system can be broken up into three groups: the monitor node (or nodes), the NVisionCC database, and the cluster itself. In addition to containing the configuration information for NVisionCC, the NVisionCC database is also augmented to contain configuration information needed by Forseti, including node last scan times and stored signatures for the files being monitored.

By utilizing the information stored in the NVisionCC database, the "monitor node" shown in Figure 2 actually embodies the functionality of both the "collector node" and "security analyzer" shown in Figure 1. This node retrieves equivalence class profiles from the database, carries out a connection-oriented scan of each node in the cluster and reports any deviations from the stored profile to the `error_log` table in the NVisionCC database. Detailing and analyzing this scan process in depth is the focus of the remainder of this paper.

4 Implementation

In this section, we discuss the design of Forseti, our distributed file integrity checker. In particular, we overview the setup and configuration of Forseti and discuss the monitoring process in full detail.

Field	Format	Description
host_type	char(60)	The equivalence class of hosts that this row applies to. This is a foreign key linking to the host_types table in the NVisionCC database.
filename	char(255)	The name of a file that should be monitored by the integrity checker.
sig1	char(128)	The signature of the file filename computed using hash algorithm 1.
...	...	...
sig$N$	char(128)	The signature of the file filename computed using hash algorithm N.

Table 2. The file_sigs table

4.1 Setup

Prior to configuring NVisionCC to make use of Forseti, the NVisionCC database must be augmented to support the integrity checker. The first step of this process entails creating the file_sigs and forseti_last_scan database tables required by Forseti. The file_sigs table is used to hold the baseline file signature information for nodes in the cluster. The format of the this table is described in Table 2. The forseti_last_scan table is used by Forseti to keep track of the date and time that each node in the cluster was last successfully integrity checked. We discuss the importance of this table further in Section 5.2.

To facilitate the creation of the file_sigs table, we have provided config.pl, a Perl script that is to be run from the collector node during the installation process. This script takes as parameters the paths to any number of programs which can be used to compute the hash value of the contents of a file. These programs must conform to the input argument format and digest output format of the Gnu utilities md5sum and sha1sum. Given this list of hash programs, config.pl creates an instance of the file_sigs table in the NVisionCC database containing one sig column for each hash program (*e.g.*, if N hash programs are supplied as arguments to config.pl the columns sig1, ..., sig$N$ are created). This allows multiple hash values to be stored for each file being monitored; the importance of storing multiple hash values will be discussed in detail in Section 5.2. The config.pl script also generates the other scripts needed by the distributed integrity checker: update_sigs.pl and forseti.pl.

After the file_sigs table has been created by config.pl, it must be populated with the names of the files that are to be monitored and their corresponding signatures. To this end, the update_sigs.pl script is to be run once for each equivalence class of nodes in the cluster. At the start of its execution, update_sigs.pl is provided with the name of an equivalence class and a list of files that should be monitored. The script then connects to a representative node of the specified equivalence class and copies over the specified hash computation binaries. At this point, each of the N signatures that will be stored in the file_sigs table are generated for each of the files that are to be monitored by Forseti. The information gathered during this process is then inserted into the file_sigs table and forms the basis of comparison used each time that Forseti is invoked by NVisionCC. For obvious reasons, it is imperative that the entirety of this setup phase takes place without interference; we fully address this requirement in Section 5.1.

4.2 Monitoring

After the NVisionCC database has been updated to support Forseti, the Forseti plug-in can be activated; this process occurs at an interval which can be configured from within NVisionCC. Once activated, Forseti follows a very basic algorithm to check for modifications to the files whose signatures were imported to the file_sigs table during the previously described setup phase. This process is presented in detail in Figure 3.

Forseti takes a very methodical approach to checking the integrity of files on the nodes of the cluster. The set of nodes is first partitioned based on equivalence class and then scanned one partition at a time (Figure 3, lines 2–6). This allows us to minimize database transactions and prevents the integrity scanner from slowing down other NVisionCC plug-ins. Prior to scanning a node, Forseti randomly chooses a hash algorithm and copies a randomly-renamed binary implementing that algorithm over to the cluster node using SCP (Figure 3, lines 8–11). An SSH session is then opened to the node which is used to execute the hash program on the files that are to be monitored for that equivalence class and gather the corresponding signatures (Figure 3, lines 12–13).

In addition to performing the tasks associated with the NVisionCC processing stage of *data collection*, the forseti.pl script also performs the *data analysis* stage by comparing the signatures returned during the scan to their expected values and reporting any errors to the NVisionCC database (Figure 3, lines 14–16). We

```
 1: {# The forseti.pl script}
 2: Let $\mathcal{C}$ be the set of node equivalence classes
 3: for all $c \in \mathcal{C}$ do
 4:   Let $N$ be the set of nodes of type $c$ stored in
      the NVisionCC database
 5:   Retrieve $\mathcal{F}_c$, the list of files to be integrity
      checked for equivalence class $c$, from the NVi-
      sionCC database
 6:   Retrieve $\mathcal{S}_c$, the signatures of the files in $\mathcal{F}_c$,
      from the NVisionCC database
 7:   for all $n \in N$ do
 8:     Choose a hash algorithm, $alg$, at random
 9:     Choose a random string, $str$
10:     Rename the binary implementing $alg$ to $str$
11:     SCP $str$ to node $n$
12:     Establish an SSH connection to $n$
13:     Compute $str(\mathcal{F}_c)$, the hash of each file in
        $\mathcal{F}_c$ using $str$
14:     if any $h \in str(\mathcal{F}_c)$ differs from its corre-
        sponding $h' \in \mathcal{S}_c$ then
15:       Log an alert to the NVisionCC database
16:     end if
17:     Update last_scans table for node $n$
18:   end for
19: end for
```

Figure 3. Pseudocode describing the integrity checking process

have found that in this case, it is more efficient to combine the information gathering and signature mapping as one step and thus take advantage of NVisionCC's flexibility to do so. After this process has been completed, the `last_scan_time` for the current node is updated to reflect the current time (Figure 3, line 17).

It is important to note that the account used by Forseti to log into the cluster nodes and carry out its scans should be an unprivileged (*i.e.*, non-root) account. Forseti requires only read access to the files that it should monitor, thus it is highly recommended that administrators create an extremely low-privilege "forseti" user with which to carry out these scans.

5 Discussion

In this section, we discuss our implementation of Forseti. In particular, we draw attention to how Forseti address the threats identified in our threat model, discuss possible attacks against Forseti, comment its performance, and summarize the benefits of the Forseti distributed file integrity checker.

5.1 A Key Assumption

Prior to addressing the security of Forseti, we must make one key assumption regarding the state of the NVisionCC database.

Assumption. *The NVisionCC database contains accurate information. This implies that the database was set up correctly and was not tampered with after installation.*

For obvious reasons, it is imperative that the entirety of the setup phase takes place using an uncompromised collector node which builds its signature profiles by contacting uncompromised representative nodes. This is easily accomplished in practice. The collector nodes used by NVisionCC should be configured to refuse all incoming connections, effectively "hardening" them against attack. In addition, rather than profiling nodes that exist in the cluster, the image for the equivalence class being profiled should be installed on a fresh node which is disconnected from the rest of the cluster (*e.g.*, it could be connected to the collector node via a separate management network). In this way, a careful administrator can ensure that the `file_sigs` table contains correct values at the time that it is created.

To protect the integrity of the NVisionCC database after it is configured, it is important to ensure that nodes in the cluster have no means of accessing the NVisionCC database. This is easily accomplished by disabling all access to the database server aside from requests originating from known collector nodes and the management consoles of system administrators. As the collector nodes should be configured to block all incoming access requests, it is highly likely that they will remain uncompromised and will not corrupt the NVisionCC database. For these reasons, we do not feel that assuming an accurate NVisionCC database is unreasonable.

5.2 Security Analysis

In the threat model presented in Section 2, we indicated that we assume that our adversary has gained root-level access to some number of nodes in the cluster that Forseti is monitoring. In these types of attacks, important system files are often modified; Section 4 focused on how Forseti can be used to detect such attacks. However, we cannot consider Forseti to be a comprehensive solution until we have considered scenarios in which the root-level adversary attacks the

Forseti system directly in an effort to cover his or her tracks. We now address several such scenarios.

Configuration Attacks

One common class of attacks against file integrity monitors involves altering the configuration of the integrity scanning software itself. In general, these types of configuration changes can take one of three forms: (1) altering the signature database, (2) stopping the scanner altogether so that failed integrity checks are never reported, or (3) altering the reporting process which logs integrity violations. We claim that Forseti is resistant to each of these classes of configuration attack.

With respect to attack (1), we have made the assumption in Section 5.1 that the cluster monitoring network has been configured in such as way as to prevent unauthorized entities from modifying the NVisionCC database. Since the file integrity signatures are stored in this database rather than on the nodes themselves, Forseti is in fact resistant to this class of attacks.

Forseti is resistant to attack classes (2) and (3) because these processes are not under the control of the compromised host. The file integrity scans are initiated by a remote collector node, not a local process such as `cron`. In this way, a compromised cluster node cannot alter the frequency with which integrity checks are initiated. In addition, the reporting process which inserts alerts into the NVisionCC database runs on the collector node and thus is not subject to tampering from a compromised cluster node. As mentioned in Section 5.1, the NVisionCC database should only be accessible to collector nodes and the management consoles of system administrators, Forseti is resistant against log tampering by a compromised node. This is in sharp contrast to other systems in which the reporting process takes place locally on the node being monitored and can thus be tampered with by the attacker.

Result Spoofing

In result spoofing attacks, attackers modify the file hash binary to return a correct hash value despite the fact that the file being integrity checked has been altered. This is often accomplished by modifying the hash binary so that it hashes files that the attacker has not modified, but returns a cached result for files that have been tampered with by the attacker. Other times, hashes are redirected to unmodified "reference" copies of modified files so that a hash computation actually takes place, though an outdated file is hashed. Forseti offers two lines of defense against this particular attack.

Algorithm	Output length
MD5 [17]	128 bit
SHA-1 [1]	160 bit
SHA-256	256 bit
SHA-384	384 bit
SHA-512	512 bit
TIGER-192 [3]	192 bit
RIPEMD-160 [7]	160 bit

Table 3. Possible hash algorithms for use by Forseti

The first line of defense against this attack addresses the threat of a modified hash binary. To mitigate this threat, Forseti generates a random filename for the hash binary that is to be used during the integrity checking process and then copies this binary to the node being monitored at scan time. Copying a fresh binary to the node being monitored decreases the risk of the binary being modified before the integrity check is run and the random filename increases the difficulty that an attacker would have to carry out a execution-path attack, in which an old binary could be run despite the fact that a new binary was just copied. In short, an attacker cannot easily attack a binary that they cannot find.

The second defense against this attack is Forseti's support for multiple hash algorithms. Rather than using the supported hash algorithms in some preset order, Forseti randomly chooses the hash algorithm to be used to check the file integrity of a particular node at runtime. This random choice implies that any attacker who attempts to pass old scan results to the collector node has a probability of success of $1/N$ where N is the number of scan algorithms configured during the installation process discussed in Section 4.1. Table 3 lists a number of candidate algorithms for use by Forseti. A secondary benefit of Forseti being agnostic with respect to the hash algorithms used is that as collisions and other insecurities are discovered in some algorithms (*e.g.*, [22, 21]), they are easily replaced.

Availability Attacks

Another way in which an attacker can prevent Forseti from reporting changes to the files that it monitors is by preventing the integrity checks initiated by the collector node from completing successfully. For instance, an easy way to accomplish this task is to prevent the collector node from logging into the cluster nodes to complete the requisite integrity checks. This can be accomplished by disabling the account used to log into the cluster nodes during the integrity checking process.

Forseti takes a conservative approach to solving this problem. Recall from Figure 3 that Forseti keeps track of the last time that each node was successfully integrity checked. It is trivial to write another NVisionCC plug-in that checks the `forseti_last_scans` table and raises an alert if a certain node has not been successfully integrity checked for some period of time, thereby alerting administrators to potential attacks on the Forseti system. This, however, is at the cost of increased attack false-positives in the event that a node is down for maintenance. However, in many cases the person monitoring the NVisionCC alert display will be aware of the maintenance operation and can thus dismiss the alert. In the future, we plan to incorporate this check of the `forseti_last_scans` table into Forseti itself.

Intercepted System Calls

One serious attack against any file integrity checking program can occur when the attacker has the ability to modify the system call table in the operating system kernel. If the attacker has this ability, he can intercept calls to the hash binary executed by the Forseti integrity checker and redirect these calls to unmodified reference copies of the files to be integrity checked[1]. This is very obviously a serious attack, as the user-level process running the integrity check will have no way of detecting that the results were falsified.

In general, a user-level process has very little means of recourse when running on a compromised operating system, as the trusted computing base can no longer be trusted. As a result, this attack could be launched against *any* file integrity checker, not just Forseti. One line of defense that Forseti has against this attack is the fact that the name of the binary that is executed to carry out the integrity checking procedure varies at each run, so the attacker must effectively guess which calls made by Forseti are invoking hash programs and which are executing other processes. In some cases the attacker will want Forseti to access the modified copies of programs (such as a modified shell), but other times access should be redirected to the older reference copies. As difficult as this may make attacking Forseti, it does not solve the problem that an attacker who can intercept and modify system calls can wreak havoc with Forseti.

To address this problem, we propose two solutions as future work. One area of research which our group is pursuing is that of node *virtualization*. Rather than executing code on the actual cluster nodes, users will log into virtual nodes running on the physical cluster. In this way, the virtual nodes could periodically be destroyed and reinstalled from read only media, thereby removing any Trojan-horse programs. As a second avenue of defense, we have considered constructing another NVisionCC plug-in that monitors results returned by a hardware-based kernel integrity checker (*e.g.*, [16]) to detect compromised operating system kernels, though this solution has the downfall of increased hardware costs.

[1]Note that this attack is similar to the result spoofing attacks previously mentioned with the exception that the compromise is happening at the kernel level, rather than the user level.

5.3 Performance

Now that we have discussed the security properties of the Forseti scanner, we wish to discuss its performance. The timing measurements that we present were measured on a cluster with dual-processor nodes running Red Hat Linux with kernel 2.4.21-15.ELsmp. The collector node used to perform the scans was located on the same gigabit Ethernet as the cluster nodes as to minimize round-trip times. This configuration seems plausible in practice, as the entity performing the security analysis of the cluster would likely have the ability to attach collector nodes to the Ethernet used by the cluster. Measurements reported are averages over 10 trials.

We tested Forseti by checking the compute nodes on our cluster for changes to 37 files totaling 2.9 megabytes in size. On average, we found that Forseti took .9397 seconds per node to carry out the entirety of the integrity checking process; this includes SCPing the hash binary (`sha1sum`) to the node being monitored, integrity checking the 37 files on that node, comparing the results returned to the compute node equivalence class profile, and logging any errors to the NVisionCC database. Slightly more than half of this time (.4978 seconds on average) was spent copying the `sha1sum` binary to the node being scanned. The majority of the remaining .4419 seconds is overhead associated with establishing the SSH connection over which the hash computation is carried out—integrity checking the 37 files took only .0845 seconds on average, once the SSH connection was established.

Overall, we feel that these performance numbers are reasonable. At a scan rate of .9397 seconds per node, a single collector node can integrity check a 512 node cluster in just over 8 minutes, while two collector nodes sharing this task can reduce the scan time to 4 minutes. In addition, watching a larger number of files also poses little problem to Forseti. With `sha1sum`, we measured a throughput of approximately 35.21 MB/second, meaning that the number of files monitored can be increased greatly without reducing the overall performance of Forseti. At these rates, administrators should have no problems configuring Forseti to track a large number of critical system files at very reasonable scan intervals.

5.4 Use in Single-Image Clusters

We now comment on the use of Forseti on single-image clusters, such as those built using the Clustermatic [5] or Scyld [19] systems. In these types of clusters, the compute nodes tend to be lightweight rather than full system images which essentially reduces the task of integrity checking the cluster to the task of integrity checking the head node. Though it seems that this is a task that could be easily carried out by a host-based file integrity checker, we contend that the use of Forseti still has several advantages over such a solution. Forseti has the benefit of keeping the file signature database stored separately from the head node, which makes the task of corrupting the signature database more difficult for malicious users or attackers. In addition, if nodes other than the head node are used to store important system files or executables, these nodes can easily be added to Forseti's scan path with minimal configuration changes. In short, though Forseti's main strengths lie in scanning clusters in which each node runs a complete system image, it can still be useful in single-image clusters.

5.5 Summary of Benefits

We now conclude our discussion of the Forseti distributed file integrity checker with a summary of its benefits.

Ease of configuration Initial configuration of the Forseti system is extremely straight-forward and almost entirely automated. By providing a few parameters to the supplied scripts, the configuration process for an entire cluster can be completed in seconds. It should also be noted that only a few nodes need to be contacted and integrity checked during this process—one node from each equivalence class. This is in contrast to installing a file integrity checker designed for the enterprise computing environment, in which each node being monitored would need to be configured separately.

Ease of maintenance Configuring new cluster nodes to be integrity checked by Forseti is as easy as updating a single table in the NVisionCC database. Since Forseti does not require any software other than `sshd` to be running on the nodes that it monitors, simply informing NVisionCC of the existence of the new node is the only step that needs to be taken for Forseti to integrity check the node. In addition, as software on the cluster nodes is upgraded, the `update_sigs.pl` script needs to be run only once per modified node equivalence class to ensure that Forseti functions correctly. Other file integrity checking systems require the database to be updated for each modified node.

Robustness against attack As discussed in Section 5.2, Forseti is robust against large classes of attacks against the system. Any number of cluster nodes can become compromised with minimal effect on the accuracy of Forseti's integrity checking process.

Efficiency The per-node costs of integrity checking cluster nodes using the Forseti system are very reasonable. In our tests, nodes could be integrity checked in under one second. In the future, we plan to examine parallel integrity checking algorithms that could reduce the overall execution time of a cluster-wide Forseti invocation.

In the following section, we discuss related work in the area of file integrity checking and discuss how Forseti differs from existing systems.

6 Related Work

File integrity checkers can serve an important role in enforcing a security policy and currently there is a wide spectrum of enterprise file integrity checkers that have been developed, each with different capabilities. To better understand where the contribution of the Forseti file integrity checker for HPC cluster security falls within this spectrum, we briefly survey enterprise file integrity checkers as summarized in Table 4. Our survey intention is to highlight representative enterprise file integrity checkers across the spectrum, not to be exhaustive.

The goal of an enterprise file integrity checker is to detect and report on directory and file system changes such as existence, ownership, permissions, and last access timestamp; these changes that may either be either accidental or malicious. The technique for determining file system changes has evolved from comparing an entire file against a saved entire version to comparing a single value calculated from the entire contents of a file against a saved single value. For this reason, file integrity checkers have always had some form of database integration to hold baseline data for comparison. It

Integrity Checker	**Unique Features**	**Supported Operating Systems**
AIDE [2]	Advanced Intrusion Detection Environment. Freeware using regular expressions, requires GNU utilities to compile.	Linux/Unix
chkrootkit [15]	Shell script collection of utilities focusing on checking system binaries for rootkit modification. Independent application with a large number of rootkit signatures.	Linux/Unix
BinAudit	a.k.a. RIACS Auditing Package. Designed to scan many hosts serially. Can collapse entire output into Email, configurable to ignore file/directory changes	Unix
Data Sentinel [6]	Commercial software. Monitors registry, two-fish encrypted reports stored in XML, CSV, and HTML.	Windows
Fcheck	Reports deviations from system baseline snapshot. Requires Perl, dependent on external executables.	Linux/Unix, Windows
GFI LANguard System Integrity Monitor [10]	Freeware and commercial application.	Windows, Linux
Integrit [4]	Small memory footprint during runtime. Included with Debian Linux distributions. Modular for database and cryptographic algorithm support. Designed for multiple scans as opposed to one complex scan.	Linux/Unix
Osiris [24]	Uses OpenSSL for encryption/authentication, monitors resident kernel extensions and changes to local user/group databases.	Windows
Samhain [18]	Daemon remembers file changes to minimize change reports (as opposed to a process invoked from cron). Monitors rootkits and login activities. Centralized administration using secure TCP and PGP-signed database support.	Linux/Unix, Mac OS X
Sentinel	Secure, signed logfiles with RIPEMD 160 bit hash. Monitors registry, options for autoload scan, and secure shutdown and closing processes.	Windows
Tripwire [12]	Commercial licensing for Linux/Unix and Windows. Can roll authorized changes into baseline for future monitoring.	Linux/Unix, Windows

Table 4. Summary of enterprise file integrity checkers

is usually the case that the file integrity database is (1) self-contained without dependence on external programs that may become compromised and (2) in ASCII format in order to provide human-readable access for reading and printing observation.

The message digest or hash value calculated from the contents of a file for integrity comparison should be both computationally efficient and infeasible to reverse. An example of an integrity checker which does not follow this design rule is the COPS tool [8]. COPS uses CRC signatures designed for streaming error detection rather than a cryptographic hash function for file integrity checking. The CRC reversal process is widely available, meaning that files can easily be altered in ways that COPS cannot detect. The first widespread file integrity checker based on cryptographic hash value comparisons was Tripwire, which was introduced in 1992. While comparing entire files with previous versions has the advantage of showing exactly what changes were made to files, in most cases knowing only that a change has been detected is enough along with the obvious resource efficiency gained from not having to save entire file systems for comparison.

Scalability is another major factor in comparing file integrity checkers, in terms of number of host file systems tested, human monitoring effort, and file sizes (which is the easiest metric to quantify and compare). All enterprise file integrity checkers exhibit non-linear speed behavior to varying degrees (slower speeds with larger files) [23]. Integrity checkers written in C tend to be the fastest (*e.g.*, AIDE, BinAudit, Integrit, Osiris, and Samhain) followed by those checkers written in Perl (*e.g.*, Fcheck) and C++ (*e.g.*, Tripwire). Osiris and Samhain collect reports and data from clients to a centralized server where the system can be more easily administered as opposed to distributed administration across many hosts. Other specialized features that distinguish enterprise file integrity checkers include warnings about incorrect configurations, handling race conditions in dynamic environments, and handling corner file system cases (e.g. file size equal to zero).

Since very few enterprise installations are uniform, enterprise file integrity checkers require significant continuous configuration. Also, since file system changes on enterprise systems may occur for a multitude of reasons, regular human analysis of reports is essential. There is no standard monitoring interface, syslog format, or message exchange system for enterprise file integrity checkers. File system change alarms are not typically prioritized across enterprise systems such that alarms that may be more important (e g. on shared production servers) are often obscured by sheer volume.

Monitoring for file system changes on an enterprise

LAN containing hundreds or thousands of hosts is not the same as monitoring for file system changes on a HPC cluster with a similar number of nodes—both the security posture and underlying detection techniques need to be significantly different for clusters. With this in mind, the Forseti file integrity checker and the NVisionCC cluster security monitoring system have been specifically designed for the unique HPC cluster environment. Forseti scans within HPC cluster equivalence classes producing uniform reports from many hosts that can be quickly evaluated against common profiles. Automated security analysis validates file integrity and presents changes to human operators using the standard Clumon visual interface (via the NVisionCC plug-in). Human management capability does not scale linearly as cluster size in nodes grows increasingly larger. For this reason the NVisionCC/Clumon visual interface is designed to be scalable in human terms to represent large HPC clusters graphically (in the thousands of nodes) while simultaneously consolidating alarms on individual cluster nodes to prevent information overload (for instance one file change with many corresponding directory changes is shown as a single node alert as opposed to many alerts). The Forseti/NVisionCC visual interface also presents file integrity changes along with other security events (process monitoring, privilege escalation, port scans) so a file change can be correlated and prioritized with other security events in the HPC cluster context. Lastly, Forseti/NVisionCC file integrity checking in the HPC cluster environment context is enhanced in contrast to enterprise systems since the HPC cluster environment is more constrained with generally fewer critical system files that change less frequently.

7 Summary

In large-scale commodity clusters, it is often the case that the compromise of a single node can lead to the compromise of an entire cluster. In computational grids, a single compromised node can easily lead to the compromise of multiple clusters, as was the case with the TeraGrid break-ins that occurred during the Spring of 2004. During these types of attacks, important system files are often times modified by attackers to help them gather passwords, avoid logging mechanisms, or open back-doors into the system. To detect these types of attacks, we have designed the Forseti distributed file integrity scanner.

Unlike other file integrity monitoring tools and host-based intrusion detection systems, Forseti was designed to meet the needs of the cluster computing environment. Rather than treating a cluster as a collection of hundreds or thousands of independent nodes, Forseti leverages the emergent structure of large-scale commodity clusters to simplify its configuration and maintenance and increase the security of the system. Forseti was designed as a plug-in to the NVisionCC cluster monitoring package, which provides a convenient framework for scheduling the plug-in and providing visual feedback to security analysts.

In this paper, we have presented the details of NVisionCC and the Forseti system and examined the security of Forseti. We highlighted several attacks against the system and showed that even a root-level adversary can do very little to falsify results or prevent the system from carrying out its integrity checks without being detected. In addition, we presented preliminary performance results which show that Forseti can monitor our test cluster at a rate of just under one second per node.

We currently have several directions for future work. We propose to make minor modifications to Forseti to make additional checks on the files that it monitors, including modification timestamps and the file's current permission set. We also plan to provide a mechanism through which the node scheduler can invoke Forseti on-demand prior to allocating a node to a user, thereby allowing users to have confidence in the freshness of the integrity scan results for the nodes on which their jobs will run. Our more ambitious plans for future work include further investigating defenses against an attacker who is able to intercept and modify system calls on the nodes being monitored, and investigating parallel node scanning algorithms.

Acknowledgments

The authors would like to thank Nadir Kiyanclar, Forrest Xin Meng, Dmitry Mogilevsky, and Michael Treaster for their thoughts and comments during the design and implementation of Forseti and for their assistance in maintaining our test cluster.

References

[1] Secure hash standard. Federal Information Processing Standards Publication 180-2, Aug. 2002. ⟨`http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf`⟩.

[2] AIDE—advanced intrusion detection environment. Web Page, Jun. 2005. ⟨`http://sourceforge.net/projects/aide`⟩.

[3] R. Anderson and E. Biham. Tiger: A fast new hash function. Technical report, Cambridge University, 1995.

[4] E. L. Cashin. integrit file verification system. Web Page, Jun. 2005. ⟨`http://integrit.sourceforge.net`⟩.

[5] Clustermatic: A complete cluster solution. Web Page, Aug. 2004. ⟨`http://www.clustermatic.org`⟩.

[6] Data sentinel. Web Page, Jun. 2005. ⟨`http://www.ionx.co.uk/html/products/data_sentinel/index.php`⟩.

[7] H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160, a strengthened version of RIPEMD. In *Fast Software Encryption*, number 1039 in Lecture Notes in Computer Science, pages 71–82, 1996.

[8] D. Farmer and E. H. Spafford. The COPS security checker system. In *USENIX Summer*, pages 165–170, 1990.

[9] J. Fullop. Clumon. Web Page, Jun. 2005. ⟨`http://clumon.ncsa.uiuc.edu/`⟩.

[10] GFiLANguard system integrity monitor. Web Page, Jun. 2005. ⟨`http://www.gfi.com/lansim/index.html`⟩.

[11] Grid attacks raise concerns among security experts. *Grid Today*, 3(17), Apr. 2004. ⟨`http://www.gridtoday.com/04/0426/103080.html`⟩.

[12] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 18–29. ACM Press, 1994.

[13] G. A. Koenig, A. J. Lee, M. Treaster, N. Kiyanclar, and W. Yurcik. Cluster security with NVisionCC: Process monitoring by leveraging emergent properties. In *5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, May 2005.

[14] A. J. Lee, G. A. Koenig, X. Meng, and W. Yurcik. Searching for open windows and unlocked doors: Port scanning in large-scale commodity clusters. In *5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, May 2005.

[15] N. Murilo and K. Steding-Jessen. chkrootkit—locally checks for signs of a rootkit. Web Page, Jun. 2005. ⟨`http://www.chkrootkit.org`⟩.

[16] N. L. Petroni, J. Molina, T. Fraser, and W. A. Arbaugh. Copilot: A coprocessor based runtime integrity monitor. In *13th USENIX Security Symposium*, Aug. 2004.

[17] R. Rivest. The MD5 message-digest algorithm. IETF Request for Comments 1321, Apr. 1992.

[18] The SAMHAIN file integrity / intrusion detection system. Web Page, Jun. 2005. ⟨`http://la-samhna.de/samhain/`⟩.

[19] Scyld software. Web Page, Aug. 2004. ⟨`http://www.scyld.com/`⟩.

[20] M. Treaster, G. A. Koenig, X. Meng, and W. Yurcik. Detection of privilege escalation for linux cluster security. In *6th LCI International Conference on Linux Clusters*, Apr. 2005.

[21] X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full SHA-1. In *Crypto 2005*, Aug. 2005.

[22] X. Wang, H. Yu, and Y. L. Yin. Efficient collision search attacks on SHA-0. In *Crypto 2005*, Aug. 2005.

[23] R. Wichman. A comparison of several host/file integrity checkers (scanners). Web Page, Oct. 2004. ⟨`http://www.la-samhna.de/library/scanners.html`⟩.

[24] B. Wotring. Osiris — host integrity monitoring. Web Page, Jun. 2005. ⟨`http://www.hostintegrity.com/osiris/`⟩.

[25] W. Yurcik, G. A. Koenig, X. Meng, and J. Greenseid. Cluster security as a unique problem with emergent properties: Issues and techniques. In *The 5th LCI International Conference on Linux Clusters: The HPC Revolution 2004*, May 2004.

[26] W. Yurcik, X. Meng, and N. Kiyanclar. NVisionCC: A visualization framework for high performance cluster security. In *CCS Workshop on Visualization and Data Mining for Computer Security (VizSEC/DMSEC)*, Oct. 2004.