Adding Availability to Log Services of Untrusted Machines

A. Arona, D. Bruschi, E. Rosti
Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
Via Comelico 39/41, 20135 Milano – Italy
{*arona, bruschi, rosti*}*@dsi.unimi.it*

Abstract

Uncorrupted log files are the critical system component for computer forensics in case of intrusion and for real time system monitoring and auditing. Protection from tampering with information can be achieved using cryptographic functions that provide authenticity, integrity, and confidentiality. However, they cannot provide the prerequisite for any further information processing, i.e., information availability. *In this case, fault tolerant strategies can be of great help improving information availability in case of accidental or deliberate deletion.*

In this paper we propose a system that increases log files availability in case of software deletion by reliably and efficiently distributing the logs on multiple independent machines. The proposed scheme is more efficient than simple replication, both from the storage space and the network bandwidth points of view. The proposed system has been implemented and its impact on performance has been measured. Since it operates as a postprocessor after log generation, the proposed system can be easily integrated with logging systems that provide various cryptographic functions for forensic purposes.

1. Introduction

One of the most common and serious consequences of computer system compromise is data deletion, in particular log files deletion in order to cancel the intruder's traces. Adopting all the countermeasures to increase information availability against users' malicious behavior is one of the basic tasks of any security system. Protection from tampering with information can be achieved using cryptographic functions that provide authenticity, integrity, and confidentiality. However, they cannot provide the prerequisite for any further information processing, i.e., *information availability*. Since information availability is the necessary precondition to any further processing aimed at preserving higher level properties such as integrity, authenticity, or confidentiality, many efforts have been put forth on improving data availability. The most popular solution, i.e., any type of RAID system [2], however, is of no help for the case we consider in this paper. RAID systems operate at file system level and are completely transparent at user level. Therefore, executing an "`rm -r /`" command would simply delete *all* files and the related redundant information that would allow to reconstruct lost data in case of single disk failures.

In this paper we address the problem of preserving information availability even after files are deleted using commands such as "`rm -r /`" in Unix system. The solution we propose is an efficient answer to the problem of recovering data after a successful and disastrous intrusion. From this point of view, the approach proposed in this paper increases availability in case of *software* and *hardware* malfunctioning, as opposed to the RAID approach, which is only intended to overcome *hardware* malfunctioning. Note that the proposed solution cannot prevent an intruder from turning off the logging facilities, but can limit the chances that the intruder removes the logs of the activities performed in order to gain access to the system.

Any reasonable proposal in the direction of increasing data hw/sw availability requires the use of a set of independent computers, hence of file systems, different from the one whose data must be preserved. Replicating the data on an independent machine, that is a machine that does not share any hardware or software component with the one where the data originally resides, is the simplest way of increasing availability. Note that replication on independent machines differs from RAID1 systems, i.e., RAID systems that mirror all the information on a duplicate set of disks, as both sets of disks in the latter provide a single logical file system, while the former involves logically and physically distinct systems. Replication on independent machines is expensive in terms of disk space and network bandwidth usage. With n hosts, where the value n depends upon the importance of the stored information, the storage and band-

width requirements are n-fold the original one. Thus, the higher the level of availability required in the presence of (possibly) untrusted hosts, the higher the number of machines, which implies larger amounts of disk space and network bandwidth. An alternative solution to increase data availability, which at the same time guarantees also confidentiality, is using Shamir's Secret Sharing strategy [7]. However, Shamir's algorithm is no better than simple replication as for the disk space occupation and bandwidth requirement, since it leads to an n-fold increase in storage space and network bandwidth requirement.

The solution to the problem of increasing data availability we propose in this paper is both disk space and network bandwidth efficient, as it is based on the Information Dispersal Algorithm [5]. The Information Dispersal Algorithm allows to distribute a file F in n pieces, each of size $|F|/m$, to multiple hosts such that it can be reconstructed using any $m < n$ such pieces, with $n/m \sim 1$. Because of the dimensions of each piece, the overhead introduced in this case, both at network and storage level, is $|F|n/m$. In this paper we apply the Information Dispersal Algorithm to log files, as they are the critical component of computer forensics and post mortem analysis after computer intrusions as well as "live" activities such as auditing and monitoring. Their availability depends upon hardware reliability and system security. Log files deletion may be caused by a system crash or a successful system compromise. While using WORM devices or printers is a viable solution against the latter [3], it is of no use against hardware faults to which such devices are as vulnerable as any other writable storage device. It is well known that the level of hardware reliability can be improved but can never reach 100%. Security on untrusted machines is usually low and log files hardly ever survive intrusions. On the other hand, trusted machines are equally subject to hardware faults, which may cause the loss of the log files so that the actions preceding the fault cannot be analyzed. Guaranteeing log files survival to either software or hardware malicious events, that is guaranteeing log files availability, is the critical property any secure logging service implicitly relies upon. Once availability is guaranteed, integrity, authenticity, and confidentiality can be added to the service in order to make sure the information it provides is reliable/trustworthy.

We realized a filter that increases data availability, based on an optimized version of the Information Dispersal Algorithm, as it is more efficient with respect to encoding time, storage space, and network bandwidth usage than the original one presented in [5]. When n hosts are used, the number m of hosts, $1 \leq m \leq n$, that are sufficient to reconstruct the original information can be set depending upon the critical nature of the log files. However, unlike the Secret Sharing scheme, $m - 1$ pieces may yield some information about the original one, although heuristics can be adopted that minimize the probability of such an event. The filter can be applied to any log generating process, whose output can be piped as input to the filter, or by applying it to the log files after they are being written, as in the case of web servers, ftp servers, routers, or system processes. A prototype has been implemented that operates with the system general purpose logging facility `syslogd`. Percentage space reductions with respect to simple replication in the range of 38% to 71% have been measured depending upon the message types and degrees of availability, i.e., minimum number of pieces necessary to reconstruct the original information.

Once service availability is guaranteed, the logging service can be enhanced in order to add integrity, authenticity, and secrecy. Cryptographic functions, such as encryption, digital signature, and (keyed) digest, can be easily added to the system proposed in this paper, together with an adequate key management system, either as a part of the application or as an autonomous system, e.g., a Certification Authority.

This paper is organized as follows. Section 2 briefly recalls the Information Dispersal Algorithm in the optimized version we developed. Section 3 presents the proposed filter and Section 4 illustrates its implementation. Extensions for forensic use are investigated in Section 5. Section 6 summarizes our contribution and concludes the paper.

2. The Information Dispersal Algorithm

In this section we illustrate the mathematical aspects of the Information Dispersal Algorithm, with particular emphasis on the optimized version we propose. Readers that are more interested in system details can skip this section, as it does not aid system understanding.

The Information Dispersal Algorithm (IDA) proposed by Rabin [5] distributes, after a suitable encoding, a piece of information I of length $|I|$ into n parts, any m of which are sufficient in order to reconstruct the original information I. Thus, even if $k = n - m$ systems are compromised and the pieces stored there are deleted, the original information can still be reconstructed correctly and completely. IDA is space efficient as the overall space occupation is $|I|n/m$, which is close to $|I|$ if n and m are chosen so that $n/m \sim 1$.

In this work we designed and implemented a more efficient version than the original one, which we call accelerated IDA (aIDA). In aIDA the encoding scheme is based on a Galois Field over 2^8 instead of the finite field Z_{257} used in [5]. Such a choice impacts on the protocol execution performance as summations and bit-wise XOR on bytes replace multiplications and summations on half words, respectively, which are used in the original description. Furthermore, it also impacts on the storage space required by the encoding scheme. While the original scheme based on Z_{257} encodes each byte in a two byte string, our scheme

maps bytes to bytes, thus halving the size of the dispersed information produced by the original implementations. The implemented version is described in what follows.

Let I be a sequence of characters $b_1 \ldots b_N$, $N = |I|$, where each character is represented as an integer over a range $[0, B]$ and $B = 255$ if the usual 8-bit byte character representation is used. Each character b_i can then be represented as an element g_j of the Galois Field over 2^8, $GF(2^8)$. Note that, because an 8-bit representation is used, the proposed system can be used also with logging systems whose output has any binary format. Choose n vectors $a_i = (a_{i1}, \ldots, a_{im}) \in GF(2^8)^m$ for $1 \leq i \leq n$ such that every subset of m different vectors are linearly independent [5]. I can be divided in $\frac{N}{m}$ sequences, each of length m,

$$I = S_1, S_2, \ldots, S_{N/m}$$

where

$$S_i = g_{1+(i-1)m} \cdots g_{m+(i-1)m}, \quad i = 1, \ldots, \frac{N}{m}$$

and $g_j \in GF(2^8)$. I can then be encoded as the sequence $I_1, I_2, \ldots, I_n$ with $I_i = c_{i1}c_{i2} \ldots c_{iN/m}$ and

$$c_{ik} = (a_{i1} + g_{1+(k-1)m}) \oplus \cdots \oplus (a_{im} + g_{km}) \quad (1)$$

where $+$ and $\oplus$ are the summation over bytes and bit-wise XOR operations, respectively. Note that since $|I_i| = |I|/m$, the sum of the lengths $|I_i|$ is $|I|n/m \sim |I|$ if m is selected such that $n/m \sim 1$.

We now show how, with any m pieces out of the entire sequence of n, it is possible to reconstruct the complete information I. For ease of notation, let $I_1 \ldots I_m$ be the sequence of available uncorrupted pieces and let $\mathbf{A} = [a_{ij}], 1 \leq i, j \leq m$, be the $m \times m$ square matrix whose i-th row is the vector a_i defined before. Compute the inverse matrix $\mathbf{A}^{-1}$ and indicate the i-th row of $\mathbf{A}^{-1}$ with $\alpha_i = (\alpha_{i1} \ldots \alpha_{im})$. Then in general, for $1 \leq k \leq N/m$,

$$b_j = h^{-1}[(\alpha_{i1} + c_{1k}) \oplus \cdots \oplus (\alpha_{im} + c_{mk})], \quad 1 \leq j \leq N \quad (2)$$

where $i = j \bmod m$, $k = \lceil j/m \rceil$ and h^{-1} is the function that maps each element in $GF(2^8)$ to the corresponding byte, which can be efficiently implemented with a table lookup using the argument as index. I is encoded and split using Eq. 1 and reconstructed using Eq. 2 after inverting matrix $\mathbf{A}$ once and for all. Each equation requires $2m$ operations for each character, thus reconstructing I requires $N2m$ operations.

Table 1 shows a sample of performance measurements of aIDA against IDA with $n = 5$ and $m = 2$ and $m = 4$ for various input strings. For each input size, the piece size and the encoding times of the input string are given. Measurements were collected on the SGI Origin 2000 used for the experiments described in Section 4. The values reported are the average over ten runs, in order to account for possible measurement instabilities. However, negligible standard deviations were observed, so they are not reported. As the tables show, the advantage of using aIDA becomes greater as the input size increases or as the redundancy level increases.

Table 1. Encoding times in $m$secs (a) and piece size in bytes (b) using IDA and aIDA for various input sizes with $n = 5$ and $m = 2$ and $m = 4$.

	INPUT	$m = 2, n = 5$		$m = 4, n = 5$	
	SIZE	IDA	aIDA	IDA	aIDA
(a)	70	0.226	0.176	0.216	0.172
	140	0.365	0.298	0.326	0.275
	210	0.498	0.419	0.444	0.384
	280	0.633	0.541	0.563	0.492
	INPUT	$m = 2, n = 5$		$m = 4, n = 5$	
	SIZE	IDA	aIDA	IDA	aIDA
(b)	70	70	35	36	18
	140	140	70	70	35
	210	210	105	106	53
	280	280	140	140	70

3. The Log Availability Filter

In this section we describe the software fault-tolerant system we designed in order to increase log files availability at low storage and network bandwidth cost. Our system is a module that transparently operates on the system generating the log information and processes it before it is sent to the independent storing systems. Since the processing phase is decoupled from log generation, the module can be combined with any log generating routine. Because a set of independent machines is used to store the processed information, using such a filter protects both against hardware failures of the individual storing machines and against illegitimate software misuse thereof, such as deletion commands executed with superuser privileges by a malicious user. Appropriate security measures are assumed to be in place on the machine generating the log files. If such measures were to be broken, i.e., the logging machine were to be compromised, nothing could prevent the intruder to turn off the Filter or the logging facility itself.

The Log Availability Filter is based on the distribution of the log file on a set of possibly untrusted machines using aIDA. By splitting the log file on different machines using aIDA, both transmission and disk writing times decrease. Increasing the number of devices the log file is written to reduces the probability of hardware faults that will make the

log files unavailable, i.e., the probability that all the device will break down, both at network and host level. Since the storing machines are independent servers, the probability of a hardware fault that will crash all of them decreases with the product of the individual fault probabilities. Furthermore, when a set of independent machines different from the one that generates the logs is used for storing the log file, a number thereof, from which the log file can be reconstructed, is reasonably expected to survive an intrusion. To improve security, the storing machines can be connected to the one generating the log entries on a serial line, so that they are usually not visible to the Internet.

The system architecture comprises a server, whose activities must be logged, that generates the log file and a set of satellite independent machines that individually store a piece of the log file. The Log Availability Filter transparently operates between the two, as illustrated in Fig. 1.
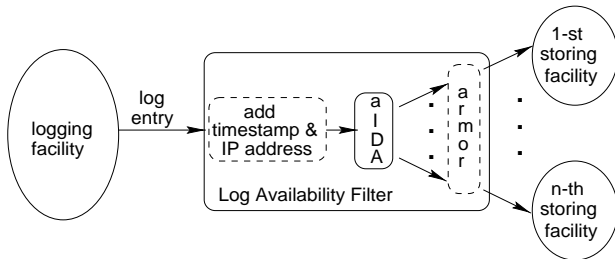

Figure 1. A schematic view of the Log Availability Filter architecture. Dashed lines indicate optional Filter phases.

The Filter operates in three steps, as log entries are generated. First, each log entry is time-stamped with the current local time and IP address of the host that generated it. The addition of such information is instrumental to ordered log reconstruction, as different logging machines may send their files to the same set of storing machines. However, the time-stamp and IP address might be eliminated if the logging service provides either of them or both already. In the second step the time-stamped entry is processed with aIDA producing a set of pieces, each one to be sent to a different machine. Each such piece is identified by a sequence number that is used for log reconstruction. Note that availability is achieved at log entry level, as log entries are processed individually as they are generated, thus improving the granularity of fault-tolerance. In the third step each piece is armored using only printing ASCII characters, e.g., with the Unix command `uuencode` [4] or the standard encapsulation format MIME [1], as a protection against corruption as the piece travels through intersystem gateways and character interpretation at the receiving host. Such a phase is optional and can be skipped when it is guaranteed that neither packet corruption nor early character interpretation may occur. Finally, the storing machines receive the pieces addressed to each of them and store them on disks.

With each entry being divided into as many pieces as there are storing machines, for any given number n of storing machines the number of pieces m that are needed in order to reconstruct the original entry is a configuration parameter of the Filter. The trade-off existing between the level of availability required and the level of confidentiality offered as a by-product by the Filter translates into conflicting requirements on the parameters of the encoding scheme. High availability implies small m, i.e., a small number of servers is sufficient to reconstruct the file. In this case, a large number of servers must be compromised in order to completely delete the log file but access to a small number of servers allows an intruder to gain access to the log file. On the other hand, with a large m, a large number of servers is needed to reconstruct the file and to be able to read it. In this case, a large number of servers must be compromised in order to have access to the file but it is enough to compromise few hosts to make file reconstruction impossible. Thus, large m implies higher confidentiality and lower availability. Note that, although aIDA does not offer encryption protection, the encoding scheme used can be adapted such that fewer than m pieces yield almost no information about the original content. Therefore, confidentiality is higher than with simple replication. Furthermore, a large m contributes to the space and transmission time efficiency of the protocol, while a small m increases the size of the individual pieces distributed on the machines, thus reducing the space efficiency of the protocol. The value $m = n/2 + 1$ strikes a balance between the two issues. If the log entries are encrypted, the level of availability can be increased, thus reducing m, without exposing the log file to confidentiality violations. Table 2 summarizes the properties of the encoding scheme as a function of m.

Table 2. Properties of the aIDA encoding scheme with respect to the number of pieces m for a system with n, $n > m$, storing machines.

m	availab.	secrecy	storage sp.	bandwidth
small	high	low	high	high
large	low	high	low	low

4. Experimentation

In this section we present the results of an implementation of the Log Availability Filter in conjunction with the general purpose logging facility `syslogd` available in Unix systems. A porting to NT environments is also

planned, where the Filter must be adapted to interact with the `event logging service`, which collects system, applications, and security logs.

We focus here on the log file generated by the system routine `syslogd` and illustrate how our Filter can be integrated with such a routine. However, the Filter is general enough to be integrated with other logging routines either at system or at application level. Interface adjustments may be necessary in order to read log entries with non-textual format or that cannot be directly sent to the Filter instead or before they are written to disk. Possible improvements to our Filter are also discussed.

4.1. System Platform

The experimental platform comprises one log generating host and five storing hosts, as depicted in Fig. 2. The logging machine is an SGI Origin 2000 with four MIPS R10000 processors equipped with 1GB main memory, 32KB on chip data cache and 32 KB on chip instruction cache, 4 MB level 2 data/instruction cache. The five storing machines are SGI O2 workstations connected on a 100 Mbps Ethernet LAN.
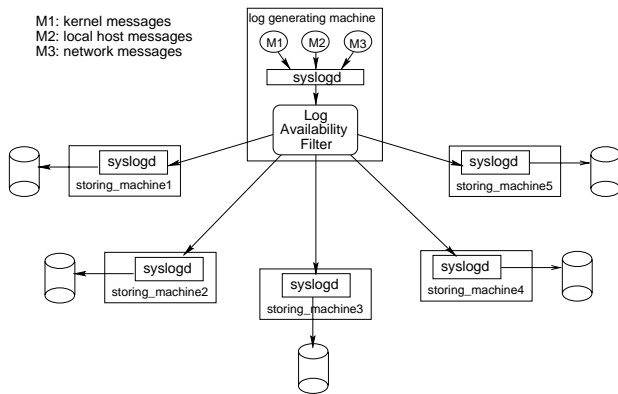

Figure 2. Experimental setting for the Log Availability Filter with 5 storing machines.

The `syslogd` command is a general purpose logging facility widely available on Unix systems [3]. Programs that need to have information logged can generate a syslog message. Syslogd reads the received message and logs it as a single-line character string into a set of files or devices (such as printers) or other hosts as described by the configuration file `/etc/syslog.conf`. Log messages can be received from one of three standard sources, indicated as M1, M2, and M3 in Fig. 2: a special device (`/dev/klog`) for messages generated by the kernel, a Unix domain socket for messages generated by processes running on the local host, and the UDP port 514 for messages generated over the (local area) network by other machines. Five actions may be

specified in the configuration file: log to a file or device, send a message to a user, send a message to all users, pipe a message to a program, and send the message to the syslog of another host.

Our system is configured as the program to which the selected entry types are piped, according to the configuration file specification. In the more general case, when piping is not possible like with the `event logging service` in NT environments, the Filter can be adjusted so as to read log records from the log files used by the application, as they are written. The Filter takes care of sending the generated pieces for each entry to the destination hosts, where the local syslogd will receive them and store them on disk. Because the syslogd on the receiving host reads a character at a time and aIDA uses an 8-bit code, the third step, i.e., the armoring phase, is necessary in our case to prevent character interpretation ahead of time.

4.2. Experimental Results

Results for experiments with 5 storing machines are presented in this section. For our case study, we have considered three types of syslog entries, namely those generated by `rlogind` for a remote login connection attempt, `named` for a DNS query, and `sendmail` for an e-mail message transmission or reception. Entry length varies depending upon the host names that appear in it and whether their symbolic or numerical name is resolved locally. The `rlogin` entry we considered is 51 bytes long, the `named` entry is 158 bytes long and the `sendmail` entry is 205 bytes long.

We first analyze the bandwidth requirements in terms of message size. The time-stamp and host IP address add 19 bytes to each entry, thus the Filter actually operates on 70, 177, and 224 bytes long entries, respectively. Each of the n packets generated by aIDA for an entry is then piggybacked with the packet sequence number and the Filter process ID, for a total of eight additional bytes. Because of the way syslogd works, the Filter is executed every time an entry is generated on the logging host, thus each entry can be uniquely associated with the process ID of the Filter execution that processed it. This information, together with the packet sequence number, is necessary to correctly reconstruct the original entry at the storing host, as syslogd uses the UDP protocol, which does not guarantee order packet delivery. Armoring each packet expands its size by a further 35%. The original entry length, the time-stamped entry length, and the size of the packets actually sent over the network with and without armoring are reported in Table 3 for each entry type. Note that, for the same input string length, packet sizes differ from those reported in Table 1 because of the eight bytes added to each packet that specify the packet sequence number and the Filter process ID.

Table 3. Overall log entry size and individual packet size in bytes for various availability degrees for the `syslog` entry type considered when 5 storing machines are used. The size of the armored packets is given in parenthesis.

SIZE	`rlogind`	`named`	`sendmail`
ORIGINAL	51	158	205
W/T_ST & IP	70	177	224
$m = 2, n = 5$	43 (61)	97 (133)	120 (161)
$m = 3, n = 5$	32 (45)	67 (93)	83 (113)
$m = 4, n = 5$	26 (37)	53 (73)	64 (89)

Table 4 compares the total amount of bandwidth required by our system for various availability degrees and by simple replication using 5 hosts in both cases. In case of replication, we present results for time-stamped entries, in order to compare the two systems on homogeneous data sets. As the table shows, the proposed Filter is always more efficient than replication, even if armoring is used. The advantage of using our scheme with respect to replication is greater when armoring is not necessary, as in this case the percentage space reduction of our system ranges from 38% to 71%. However, when armoring is needed, the percentage space reduction of our system is still worth its cost since it ranges from 13% to 60%. In both cases, the minimum gain is for the high availability case (i.e., $m = 2, n = 5$) for short entries (i.e., `rlogind`) and the maximum gain is for the low availability case (i.e., $m = 4, n = 5$) for long entries (i.e., `sendmail` and `named`).

Table 4. Total bandwidth requirements in bytes for replication on 5 hosts and for three availability degrees using the Log Availability Filter, for plain and armored packets, with time-stamped entries in all cases. The percentage storage reduction is given in parenthesis.

STRATEGY	`rlogind`	`named`	`sendmail`
5 REPL.	350	885	1120
$m = 2, n = 5$, PL.	215 (38%)	485 (45%)	600 (46%)
$m = 2, n = 5$, AR.	305 (13%)	665 (24%)	805 (28%)
$m = 3, n = 5$, PL.	160 (54%)	335 (62%)	415 (63%)
$m = 3, n = 5$, AR.	225 (35%)	465 (47%)	565 (49%)
$m = 4, n = 5$, PL.	130 (62%)	265 (70%)	320 (71%)
$m = 4, n = 5$, AR.	185 (47%)	365 (58%)	445 (60%)

We now consider the execution time overhead introduced by the Filter. Since some phases of the Filter are optional, we have separately measured the times of each of them, so as to be able to correctly identify the system bottleneck, when the Filter is in place. The times (in $m$seconds) to add the time-stamp and IP address, to process the entry with aIDA, to armor all pieces, and finally to distribute the various pieces over the network for the various entry types and availability degrees are reported in Table 5. In order to account for possible measurement instabilities, each value is the average over ten runs. However, negligible standard deviations were observed, so they are not reported.

As the table shows, the time to add the time-stamp and IP address is basically constant, as expected since it does not depend on the input size. Armoring the packets introduces a very limited overhead, as the coding is a simple sequence of bitwise operations. The relatively large execution times of the aIDA phase[1] is however one order of magnitude smaller than the time to send out the packets. In fact, packet distribution is the system bottleneck and it is greater than the sum of the other components. Comparable, if not larger, times would be spent in case of replication, and in case of remote storing of log files, as suggested as a good security practice. Therefore, we conclude that the cost of adding availability to logging services in terms of processing overhead is not an issue.

Table 5. Execution times in $m$seconds of the phases of the Log Availability Filter for various degrees of availability.

AVAIL. DEG.	PHASE	`rlogind`	`named`	`sendmail`
all cases	ADD T_ST & IP	0.16	0.17	0.17
$m = 2, n = 5$	aIDA	0.2	0.40	0.49
	ARMOR	0.018	0.030	0.037
	SEND	2.3	2.3	2.3
$m = 3, n = 5$	aIDA	0.2	0.39	0.47
	ARMOR	0.02	0.031	0.037
	SEND	2.4	2.3	2.3
$m = 4, n = 5$	aIDA	0.2	0.40	0.46
	ARMOR	0.021	0.033	0.039
	SEND	2.1	2.0	2.0

4.3. Observations

We analyze here the proposed implementation and consider possible improvements to our system. We concentrate on the transmission aspects and on the logging mechanism.

Because `syslogd` uses the UDP protocol when transmitting the log to a machine other than the one that gen-

---

[1] The execution times of aIDA are generally larger than those reported in Table 1 because we account to aIDA the time for the allocation of some data structures, which are used in subsequent phases and which are not considered in the IDA vs aIDA comparison.

erated it, it is subject to packet loss. The use of our Filter provides tolerance also against UDP packet loss. The degree of availability, which protects from log file deletion in case of compromise of up to a certain number of machines, applies to packet loss as well. However, UDP packet loss may combine maliciously with system compromise so that an entry may not be recoverable although fewer than $n - m$ systems have been compromised. When the set of compromised hosts is disjoint or partially disjoint from the set of hosts that have not received their portion of log entry because of UDP packet loss, so that the union of the two sets contains more than $n - m$ machines, log file reconstruction becomes impossible because fewer than m machines have reliable data. Using a reliable transport protocol, namely TCP, would guarantee the system against packet loss, thus solving the problem of the combined effect of machine compromise and packet loss. The packet size would also decrease, as the packet sequence number and Filter process ID would not be necessary. In this case, `syslogd` could not be used directly on the storing machine as the packets recipient, since it listens on UPD port 514 by definition. Ad hoc clients should be developed and installed on the storing hosts that would receive the packets and pass them along to the local `syslogd`. The price of using a reliable transport protocol is paid with a performance degradation, as TCP is slower than UDP. However, in a local area network, where packet loss and retransmission are not frequent, performance degradation is expected to be limited.

As Table 5 shows, transmission is the Filter bottleneck. In order to optimize such a phase, we considered the possibility of using the UDP based multicast service. With multicast communication, a single message is sent to a registered group of hosts participating in the multicast instead of as many messages as there are hosts in the group. In this case too, a receiving client on the storing machines would be needed, as `syslogd` is not multicast enabled. Because of the nature of the communications in the Filter, where each recipient is sent a different message, multicast is not the correct solution from a logical point of view. However, it turns out to be a good solution from an implementation point of view, since the time to send all packets reduces to 0.55 $m$sec on the average, about one fourth of the unicast case, regardless of the availability degree, entry type, and number of recipients, i.e., participating hosts. Although the overall bandwidth requirement does not change with respect to the unicast communication case, when using multicast all messages are sent from a single socket, which explains why the send time is roughly constant in this case. Thus, if multicast were used, all of the storing machines would see the entire traffic for each log entry and would have to discard the packets destined to other machines. If they kept them all instead of dropping them, a further level of fault-tolerance would be introduced as a by-product that would reduce the chances of malicious combinations of UDP packet loss and packet deletion due to machine compromise.

The measures reported in the previous section are relative to the Filter execution time but do not take into account the time to launch its execution. The way `syslogd` works requires that, in case log entries are piped to a program, such a program be executed every time an entry occurs. A more efficient implementation that optimizes system overhead for program startup every time it must be executed would consider running the program (our Filter in this case) as a daemon. Such an alternative would require to modify the standard `syslogd`.

5. Logging as Evidence

A possible use of log files is to supply evidence in legal proceedings in case of computer security incidents. Legal acceptability of tracks as evidence depends upon admissibility, i.e., the conformity to the jurisdiction's legal rules, and weight, i.e., the degree of understanding and convincement for the court and judge(s). Admissibility strictly depends upon the country's body of laws regarding electronic documents and their generation and storage. Availability is the necessary prerequisite for legal evidence, although it is not sufficient. Authentication and integrity, which are usually among the requirements for legally valid electronic documents, are also necessary, to guarantee the origin of the logs and their non-modification [8]. Authentication of the log generating machine and integrity checksums to detect possible alterations of the logs, contribute to increase the legal weight of log files. Confidentiality may be required for privacy reasons, e.g., when the owner of the logging system is not the owner of the information being stored as in case of outsourced systems, but it does not add weight to the evidence.

The Filter we have presented guarantees log availability by distributing each log entry on a set of possibly untrusted hosts so as to hamper log file destruction. The logs processed with our Filter have no value as evidence in court, as much as the standard ones. However, they can be given legal weight if authentication of the logging machine and integrity checks of the log entry are added in the Filter processing, according to the jurisdiction's legal rules. Regardless of the specific solution adopted, a trusted machine is needed if log entry authentication is required. Different strategies can be devised in order to obtain a secure, in terms of confidentiality, integrity, and availability, log service. Two are the scenarios we can envision.

We believe that the most elegant and complete solution combines our Filter with the logging system such described in [6]. Schneier's system guarantees that in case of intrusion, an attacker will not be able to read nor alter or delete undetectably log entries made before the intrusion.

A trusted host that interacts with the logging machine is necessary for key management, which is limited to the initial sharing of a secret key with the logging machine from which the chain of keys used for authentication, MAC computation, and encryption is constructed, and the exchange of the first session key of the chain. Note that the sophisticated system proposed by Schneier guarantees the log confidentiality and allows to grant log access on a role-based security scheme, which is not a requirement for legal evidence.

The other scenario consists of a network of servers implementing the IPSEC communication protocol where our Filter is simply deployed as a log postprocessor. In this environment, log authentication and integrity can be achieved at network level since cryptographic functions can be computed using the session key established between the two hosts during the authentication phase. The deployment of our Filter in such an environment would complete the requirements for log legal evidence with availability, as authentication and integrity are provided by the environment itself. The role of the trusted host of Schneier's solution is played by the CA that manages the host public keys. Encryption, although not necessary for legal purposes, could be added either at application level or at network level.

6. Conclusions

The first step towards providing auditing capabilities that can be exercised after system compromise is preventing log file destruction, i.e., adding availability to log services. Mechanisms that can help detect file alteration and prevent unauthorized file reading complete the requirements for effective intrusion detection and auditing systems.

In this paper we have presented a software fault-tolerant system that adds availability to any system logging facility. The proposed system is based on an original implementation of the Information Dispersal Algorithm, which provides an efficient encoding scheme that significantly reduces storage space and network bandwidth requirements with respect to the simple replication strategy. A prototype was developed and the impact of its deployment in a real environment was measured. A variety of long entry types and degrees of availability have been considered and possible optimizations have been discussed. The application of the proposed system to the design of a legally valid logging system has been outlined and possible scenarios have been illustrated.

References

[1] N. Borenstein and N. Freed. Mime (multipurpose internet mail extensions): Mechanisms for specifying and describing the format of internet message bodies. *Internet RFC 1341*, June 1992.

[2] P. Chen, E. Lee, G. Gibson, and D. Patterson. Raid: High-performance, reliable secondary storage. *ACM Computing Survey*, 26(2):145, February 1994.

[3] S. Garfinkel and E. Spafford. *Practical Unix and Internet Security*. O'Reilly & Associates, 1996.

[4] G. Glass. *Unix for Programmers and Users: A Complete Guide*. Prentice Hall Int., 1993.

[5] M. Rabin. Efficient dispersal of information for security, load balancing, and fault-tolerance. *Journal of the ACM*, 36(2):335–348, February 1989.

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

[7] A. Shamir. How to share a secret. *Communication of the ACM*, 22(11):612–613, November 1979.

[8] P. Sommer. Intrusion detection systems as evidence. *Recent Advances in Intrusion Detection, RAID 98*, September 1998.