Secure Audit Logs to Support Computer Forensics

BRUCE SCHNEIER and JOHN KELSEY
Counterpane Systems

In many real-world applications, sensitive information must be kept in log files on an untrusted machine. In the event that an attacker captures this machine, we would like to guarantee that he will gain little or no information from the log files and to limit his ability to corrupt the log files. We describe a computationally cheap method for making all log entries generated prior to the logging machine's compromise impossible for the attacker to read, and also impossible to modify or destroy undetectably.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms: Security

Additional Key Words and Phrases: Auditing, audit Logs, authentication, computer forensics, hash chains, intrusion detection

1. INTRODUCTION

1.1 A Description of the Problem

We have an untrusted machine, $\mathcal{U}$, which is not physically secure or sufficiently tamper-resistant to guarantee that it cannot be taken over by some attacker. However, this machine needs to be able to build and maintain a file of audit log entries of some processes, measurements, events, or tasks.

With a minimal amount of interaction with a trusted machine, $\mathcal{T}$, we want to make the strongest security guarantees possible about the authenticity of the log on $\mathcal{U}$. In particular, we do not want an attacker who gains control of $\mathcal{U}$ at time t to be able to read log entries made before time t, and we do not want him to be able to alter or delete log entries made before

A version of this paper appeared in Schneier and Kelsey [1998].

Authors' address: Counterpane Systems, 101 E Minnehaha Parkway, Minneapolis, MN 55419; email: schneier@counterpane.com.

time t in such a way that his manipulation will be undetected when $\mathcal{U}$ next interacts with $\mathcal{T}$.

It is important to note that $\mathcal{U}$, while "untrusted," isn't generally expected to be compromised. However, we must be able to make strong statements about the security of previously generated log entries even if $\mathcal{U}$ is compromised.

In systems where the owner of a device is not the same person as the owner of the secrets within the device, it is essential that audit mechanisms be in place to determine if there has been some attempted fraud. These audit mechanisms must survive the attacker's attempts at undetectable manipulation. This is not a system to prevent all possible manipulations of the audit log; this is a system to detect such manipulations after the fact.

Applications for this sort of mechanism abound. Consider $\mathcal{U}$ to be an electronic wallet—a smart card, a calculator-like device, a dongle on a PC—that contains programs and data protected by some kind of tamper resistance. The tamper resistance is expected to keep most attackers out, but it is not 100% reliable [Anderson and Kuhn 1996; McCormac 1996]. However, the wallet occasionally interacts with trusted computers ($\mathcal{T}$) in banks. We would like the wallet to keep an audit log of both its actions and data from various sensors designed to respond to tampering attempts. Moreover, we would like this log to survive successful tampering, so that when the wallet is brought in for inspection it will be obvious that the wallet has been tampered with.

There are other examples of systems that could benefit from this protocol:

—A computer that logs various kinds of network activity needs to have log entries of an attack undeletable and unalterable, even in the event that an attacker takes over the logging machine over the network.[1]

—An intrusion-detection system that logs the entry and exit of people into a secured area needs to resist attempts to delete or alter logs, even after the machine on which the logging takes place has been taken over by an attacker [Schneier and Kelsey 1999].

—A secure digital camera needs to guarantee the authenticity of pictures taken, even if it is reverse-engineered sometime later [Kelsey et al. 1996].

—A computer under the control of a marginally trusted person or entity needs to keep logs that can't be changed after the fact, despite the intention of the person in control of the machine to "rewrite history" in some way. This also comes up when a secure coprocessor, or "dongle," is attached to an untrusted computer [Kelsey and Schneier 1996; Schneier and Kelsey 1997b].

[1] In Stoll [1989], Cliff Stoll attached a printer to a network computer for just this purpose.

—A computer that is keeping logs of confidential information needs to keep that information confidential even if it is taken over for a time by some attacker.

—Mobile computing agents could benefit from the ability to resist alteration of their logs even when they're running under the control of a malicious adversary [Riordan and Schneier 1998].

1.2 Limits on Useful Solutions

A few moments' reflection will reveal that no security measure can protect the audit log entries written *after* an attacker has gained control of $\mathcal{U}$. At that point, $\mathcal{U}$ will write to the log whatever the attacker wants it to write. All that is possible is to refuse the attacker the ability to read, alter, or delete log entries made *before* he compromised the logging machine.

If there is a reliable, high-bandwidth channel constantly available between $\mathcal{T}$ and $\mathcal{U}$, then this problem won't come up. $\mathcal{U}$ will simply encrypt each log entry as it is created and send it to $\mathcal{T}$ over this channel. Once logs are stored by $\mathcal{T}$, we are willing to trust that no attacker can change them.

Finally, no cryptographic method can be used to actually prevent the deletion of log entries: solving that problem requires write-only hardware such as a writable CD-ROM disk, a WORM disk, or a paper printout. The only thing these cryptographic protocols can do is to guarantee detection of such deletion, and that is assuming $\mathcal{U}$ eventually manages to communicate with $\mathcal{T}$.

These three statements define the limits of useful solutions to this problem. We are able to make strong statements only about log entries $\mathcal{U}$ made before compromise; we cannot prevent deletion of log entries stored in read/write media, and our audit log protocol is interesting only when there is no communications channel of sufficient reliability, bandwidth, and security to simply continuously store the logs on $\mathcal{T}$.

In essence, this technique is an implementation of an engineering tradeoff between how "on-line" $\mathcal{U}$ is and how often we expect $\mathcal{U}$ to be compromised. If we expect $\mathcal{U}$ to be compromised very often—once a minute, for example—then we should send log entries to $\mathcal{T}$ at least once or twice every minute; hence, $\mathcal{U}$ will need to be on-line nearly all the time. In many systems, $\mathcal{U}$ is not expected to be compromised nearly as often, and is also not on-line nearly as continuously. Therefore, we only need $\mathcal{U}$ to communicate log entries to $\mathcal{T}$ infrequently, at some period related to the frequency with which you expect that $\mathcal{U}$ may be compromised. The audit log technique in our paper enables this trade-off. It provides a "knob" that the system architect can adjust based on his judgment of this trade-off; furthermore, the knob can be adjusted during the operation of the system as expectations of the rate of compromise change.

1.3 Organization of This Paper

The remainder of this paper is divided into sections as follows: In Section 2, we discuss notation and tools. In Section 3, we present our general scheme. Then, in Section 4, we discuss some extensions and variations on the scheme. In Section 5, we discuss uses of this system as a forensic tool. Finally, in Section 6, we provide a summary of what we've done and interesting directions for further research in this area.

2. NOTATION AND TOOLS

In the remainder of this paper, we will use the following notation:

(1) ID_x represents a unique identifier string for an entity, x, within this application.

(2) $PKE_{PK_x}(K)$ is the public-key encryption, under x's public key, of K, using an algorithm such as RSA [Rivest et al. 1978] or ElGamal [ElGamal 1985].

(3) $SIGN_{SK_x}(Z)$ is the digital signature, under x's private key, of Z, using an algorithm such as RSA or DSA [NIST 1994].

(4) $E_{K_0}(X)$ is the symmetric encryption of X under key K_0, using an algorithm such as DES [NBS 1977], IDEA [Lai et al. 1991], or Blowfish [Schneier 1994].

(5) $MAC_{K_0}(X)$ is the symmetric message authentication code (HMAC or NMAC [Bellare et al. 1996]), under key K_0, of X.

(6) $hash(X)$ is the one-way hash, using an algorithm such as SHA-1 [NIST 1993] or RIPE-MD [Dobbertin et al. 1996], of X.

(7) X, Y represents the concatenation of X with Y.

Descriptions of most of these algorithms are in Stinson [1995], Schneier [1996], and Menezes et al. [1997].

Note that all authenticated protocol steps in this paper should include some nonce identifying the specific application, version, protocol, and step. This nonce serves to limit damaging protocol interactions, either accidental or intentional [Anderson 1995; Kelsey et al. 1998]. In our protocols, we will use p to represent this unique step identifier.

Additionally, many of the protocols require the two parties to establish a secure connection, using an authentication and key-agreement protocol that has perfect forward secrecy, such as authenticated Diffie-Hellman [Diffie et al. 1992]. The purpose of this is for the two parties to prove their identity to each other, and to generate a shared secret with which to encrypt subsequent messages in the protocol.

In the remainder of this paper, we will use the following players:

(1) $\mathcal{T}$ is the trusted machine. It may typically be thought of as a server in a secure location, though it may wind up being implemented in various ways: a tamper-resistant token, a bank ATM, etc.

(2) $\mathcal{U}$ is the untrusted machine, on which the log is to be kept.

(3) $\mathcal{V}$ is a moderately trusted verifier, which will be trusted to review certain kinds of records on a log, but not trusted with the ability to change records. Note that $\mathcal{V}$ must interact with $\mathcal{T}$ in order to access these records.

In this paper, we assume that $\mathcal{U}$ has both short-term and long-term storage available. The long-term storage will store the audit log, and we assume that it is sufficiently large that filling it up is not a problem. We assume that $\mathcal{U}$ can irretrievably delete information held in short-term memory, and that this is done each time a new key is derived. We also assume that $\mathcal{U}$ has some way of generating random or cryptographically strong pseudorandom values. Finally, we assume the existence of several cryptographic primitives, and a well-understood way to establish a secure connection across an insecure medium.[2] Methodologies for all of these are described in great detail elsewhere—see Stinson [1995], Schneier [1996], and Menezes et al. [1997].

3. A DESCRIPTION OF OUR METHOD

Our system leverages the fact that the untrusted machine creating the logfile initially shares a secret key with a trusted verification machine. With this key, we create the logfile.

The security of our logfile comes from four basic facts:

(1) The log's authentication key is hashed, using a one-way hash function, immediately after a log entry is written. The new value of the authentication key overwrites and irretrieveably deletes the previous value.

(2) Each log entry's encryption key is derived, using a one-way process, from that entry's authentication key. This makes it possible to give encryption keys for individual logs out to partially trusted users or entities (so that they can decrypt and read entries), without allowing those users or entities to make undetectable changes.

(3) Each log entry contains an element in a hash chain that serves to authenticate the values of all previous log entries [Haber and Stornetta 1991; Schneier and Kelsey 1997a]. It is this value that is actually authenticated, which makes it possible to remotely verify all previous log entries by authenticating a single hash value.

[2] This secure connection could use any of several established protocols—IPSec, SSL—or a proprietary system.

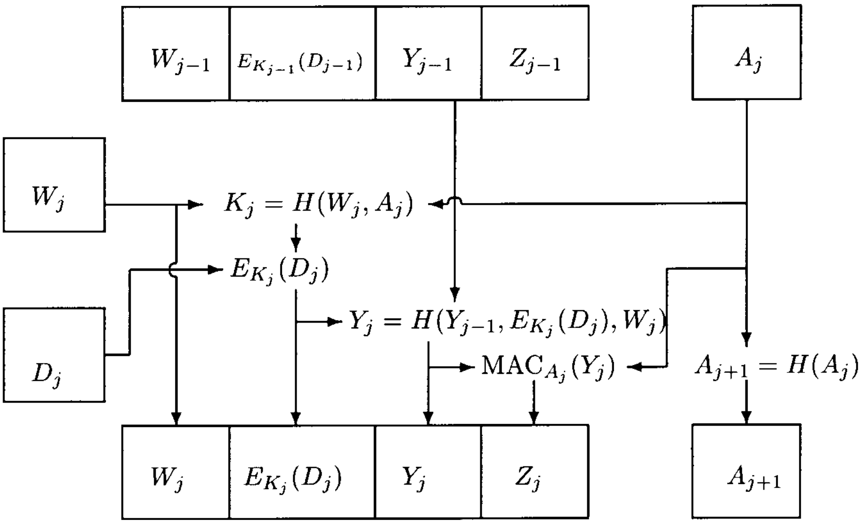

Fig. 1. Adding an entry to the log.

(4) Each log entry contains its own permission mask. This permission mask defines which log entries can be accessed by partially trusted users. Different partially trusted users can be given access to different kinds of entries. Because the encryption keys for each log entry are derived partly from the log entry type, lying about what permissions a given log entry has ensures that the partially trusted user simply never gets the right key.

3.1 Log Entry Definitions and Construction Rules

All entries in the log file use the same format, and are constructed according to the following procedure:

(1) D_j is the data to be entered in the jth log entry of ID_{log}. The specific data format of D is not specified in our scheme: it must merely be something that the reader of the log entries will unambiguously understand, and that can be distinguished in virtually all cases from random gibberish. (If we are dealing with raw binary data here, we may add some structure to the data to make detection of garbled information likely, though this is seldom going to be important.)

(2) W_j is the log entry type of the jth log entry. This type serves as a permissions mask for $\mathcal{V}$; $\mathcal{T}$ will be allowed to control which log entry types any particular $\mathcal{V}$ will be allowed to access.

(3) A_j is the authentication key for the jth entry in the log. This is the core secret that provides all of this scheme's security. Note that $\mathcal{U}$ must generate a new A_0 before starting the logfile; A_0 can be given to $\mathcal{U}$ by $\mathcal{T}$

at startup, or $\mathcal{U}$ can randomly generate it and then securely transmit it to $\mathcal{T}$.

(4) $K_j = hash$(“Encryption Key”, W_j, A_j). This is the key used to encrypt the jth entry in the log. Note that W_j is used in the key derivation to prevent the partially trusted verifier, $\mathcal{V}$, from getting decryption keys for log entry types to which he is not permitted access.

(5) $Y_j = hash(Y_{j-1}, E_{K_j}(D_j), W_j)$. This is the hash chain that we maintain to allow partially trusted users, $\mathcal{V}$s, to verify parts of the log over a low-bandwidth connection with the trusted machine, $\mathcal{T}$. Y_j is based on $E_{K_j}(D_j)$ instead of D_j so that the chain can be verified without knowing the log entry. At startup, Y_{-1} is defined as a 20-byte block of binary zeros.[3]

(6) $Z_j = MAC_{A_j}(Y_j)$.

(7) $L_j = W_j, E_{K_j}(D_j), Y_j, Z_j$, where L_j is the jth log entry.

(8) $A_{j+1} = hash$(“Increment Hash”, A_j).

Note that when A_{j+1} and K_j are computed, the previous A_j and K_{j-1} values are irretrieveably destroyed; under normal operation, there are no copies of these values kept on $\mathcal{U}$. Additionally, K_j is destroyed immediately after use in Step (4). (Naturally, an attacker will probably store A_j values after he takes control of $\mathcal{U}$.)

The above procedure defines how to write the jth entry into the log, given A_{j-1}, Y_{j-1}, and D_j. Figure 1 gives an illustration of this process.

If an attacker gains control of $\mathcal{U}$ at time t, he will have a list of valid log entries, $L_1, L_2, \ldots, L_t$, and the value A_{t+1}. He cannot compute A_{t-n} for any $n \leq t$, so he cannot read or falsify any previous entries. He can delete a block of entries (or the entire log file), but he cannot create new entries to replace the ones he deleted, regardless of whether he deleted a block of entries in the middle of the log or a block of entries at the end of the log. The next time $\mathcal{U}$ interacts with $\mathcal{T}$, $\mathcal{T}$ will realize that entries have been deleted from the log and that (1) $\mathcal{U}$ may have committed some invalid actions that have not been properly audited and (2) $\mathcal{U}$ may have committed some valid actions whose audit record has been deleted.[4]

Even if the attacker does not delete entries, the audit system should include entries that point to a successful intrusion by the attacker. Again, since the attacker cannot read past entries, he will have no way of knowing if his intrusion was noted by the log system or not.

[3] There is no security reason for this; it has to be initialized as something.

[4] If the attacker gains control of $\mathcal{U}$ before Step (8), he can learn A_t. In this case, the tth log entry is not secured from deletion or manipulation.

3.2 Startup: Creating the Logfile

In order to start the logfile, $\mathcal{U}$ must irrevocably commit A_0 to $\mathcal{T}$. Once A_0 has been committed to, there must be a valid log on $\mathcal{U}$, properly formed in all ways, or $\mathcal{T}$ will suspect that someone is tampering with $\mathcal{U}$.

In the simplest case, $\mathcal{T}$ is able to reliably receive a message (but perhaps not in realtime) from $\mathcal{U}$. $\mathcal{U}$ knows $\mathcal{T}$'s public key, and has a certificate of her own public key from $\mathcal{T}$. The protocol works as follows:

(1) $\mathcal{U}$ forms:

K_0, a random session key.
d, a current timestamp.
d^+, timestamp at which $\mathcal{U}$ will time out.
ID_{log}, a unique identifier for this logfile.
$C_{\mathcal{U}}$, $\mathcal{U}$'s certificate from $\mathcal{T}$.
A_0, a random starting point.
$X_0 = p, d, C_U, A_0$.

She then forms and sends to $\mathcal{T}$:
$M_0 = p, ID_{\mathcal{U}}, PKE_{PK_{\mathcal{T}}}(K_0), E_{K_0}(X_0, SIGN_{SK_{\mathcal{U}}}(X_0))$,

where p is the protocol step identifier.

(2) $\mathcal{U}$ forms the first log entry, L_0, with $W_0 =$ **LogfileInitializationType** and $D_0 = d, d^+, ID_{log}, M_0$. Note that $\mathcal{U}$ does not store A_0 in the clear, as this could lead to replay attacks: an attacker gets control of $\mathcal{U}$ and forces a new audit log with the same A_0. $\mathcal{U}$ also stores $hash(X_0)$ locally while waiting for the response message.

(3) $\mathcal{T}$ receives and verifies the initialization message. If it is correct (i.e., it decrypts correctly, $\mathcal{U}$'s signature is valid, $\mathcal{U}$ has a valid certificate), then $\mathcal{T}$ forms:
$X_1 = p, ID_{log}, hash(X_0)$.

$\mathcal{T}$ then generates a random session key, K_1, and forms and sends:
$M_1 = p, ID_{\mathcal{T}}, PKE_{PK_{\mathcal{U}}}(K_1), E_{K_1}(X_1, SIGN_{SK_{\mathcal{T}}}(X_1))$.

(4) $\mathcal{U}$ receives and verifies M_1. If all is well, then $\mathcal{U}$ forms a new log entry, L_j, with W_j=**ResponseMessageType** and $D_j = M_1$. $\mathcal{U}$ also calculates $A_1 = hash$("Increment Hash", A_0).

If $\mathcal{U}$ doesn't receive M_1 by the time-out time d^+, or if M_1 is invalid, then $\mathcal{U}$ forms a new log entry with $W_1 =$ **AbnormalCloseType** and $D_1 =$ the current timestamp and the reason for closure. The log file is then closed.

Depending on the application, we may or may not allow $\mathcal{U}$ to log anything between the time it sends M_0 and the time it receives M_1 (or by the time-out time). In high-security applications, we might not want to take the chance that there are any problems with $\mathcal{T}$ or the communications. In other applications, it might be allowable for $\mathcal{U}$ to perform some actions while waiting for $\mathcal{T}$ to respond.

The purpose of writing the abort-startup message is simply to prevent there ever being a case where, due to a communications failure, $\mathcal{T}$ thinks there is a logfile being used even though none exists. Without this protection, an attacker could delete $\mathcal{U}$'s whole logfile after compromise, and claim to simply have failed to receive M_1 during the startup. In implementations where $\mathcal{U}$ waits for the response message before writing any log entries, M_1 will be the second message written in the log as well as the last. Otherwise, when $\mathcal{T}$ sees this message, he will believe either that $\mathcal{U}$ didn't receive the response message, or that $\mathcal{U}$ was compromised before the response message.

3.3 Closing the Logfile

Closing the logfile involves three operations: Writing the final-record message, D_f (the entry code is **NormalCloseMessage** and the data is a timestamp); irretrieveably deleting A_f and K_f; and (in some implementations) actually closing the file. Note that after the file has been properly closed, an attacker who has taken control of $\mathcal{U}$ cannot make any alteration to the logfile without detection. Nor can an attacker delete some entries (and possibly add others) and then create a valid close-file entry earlier in the log. Finally, the attacker cannot delete the whole log file, because of the earlier interaction between $\mathcal{U}$ and $\mathcal{T}$. Any of these changes will be detected when $\mathcal{T}$ sees the final logfile.

3.4 Validating the Log

When $\mathcal{T}$ receives the complete and closed log, he can validate it using the hash chain and Z_f (since it already knows D_0). He can also derive all the encryption keys used, and thus read the whole audit log.

3.5 Verification, Verifiers, and Querying Entries

At times, a moderately trusted person or machine, called $\mathcal{V}$, may need to verify or read some of the logfile's records while they are still on $\mathcal{U}$. This is made possible if $\mathcal{T}$ has sent M_1 to $\mathcal{U}$ (see Section 3.3), and if $\mathcal{V}$ has a suitable channel available to and from $\mathcal{U}$. Note that this can occur before $\mathcal{T}$ has received a copy of the log from $\mathcal{U}$, and before $\mathcal{U}$ has closed the logfile.

(1) $\mathcal{V}$ receives a copy of the audit log, $L_0, L_1, L_2, \ldots, L_f$, where f is the index value of the last record, from $\mathcal{U}$. Note that $\mathcal{U}$ does not have to send $\mathcal{V}$ a complete copy of the audit log, but it must send $\mathcal{V}$ all entries from L_0 through some L_f, including the entry with M_1.

(2) $\mathcal{V}$ goes through the hash chain in the log entries (the Y_i values), verifying that each entry in the hash chain is correct.

(3) $\mathcal{V}$ establishes a secure connection with $\mathcal{T}$.

(4) $\mathcal{V}$ generates a list of all the log entries she wishes to read, from 0 to n. This list contains a representation of the log entry type and index of each entry to which the verifier is requesting access. (Typically, only some log entry types will be allowed, in accordance with $\mathcal{V}$'s permission mask.) This list is called $Q[0..n]$, where $Q_i = j, W_j$.

(5) $\mathcal{V}$ forms and sends to $\mathcal{T}$:
$M_2 = p, ID_{log}, f, Y_f, Z_f, Q[0..n]$.

(6) $\mathcal{T}$ verifies that the log has been properly created on $\mathcal{U}$, and that $\mathcal{V}$ is authorized to work with the log. $\mathcal{T}$ knows A_0 so he can calculate A_f; this allows him to verify that $Z_f = MAC_{A_f}(Y_f)$. If there is a problem, $\mathcal{T}$ sends the proper error code to $\mathcal{V}$ and records that there is a problem with ID_{log} on $\mathcal{U}$ or a problem with $\mathcal{V}$.

(7) If there are no problems, $\mathcal{T}$ forms a list, $R[1..n]$, of responses to the requests in Q. Each entry in Q gets a corresponding entry in R: either giving the verifier the decryption key for that record, or else giving it an error code describing the reason for the refusal of access. (Typically, this will be because the log entry type isn't allowed to be read by this verifier.) Note that $\mathcal{T}$ computes these keys based on the log entry type codes given. If $\mathcal{V}$ provides incorrect codes to $\mathcal{T}$, the keys will be incorrect and $\mathcal{V}$ will be unable to decrypt the log entries. Additionally, $\mathcal{V}$ will not be able to derive the right key from the key he has been given.

(8) $\mathcal{T}$ forms and sends to $\mathcal{V}$:
$M_3 = p, R[0..n]$.

(9) $\mathcal{V}$ is now able to decrypt and read, but not to change, the log entries whose keys were sent in M_4. $\mathcal{V}$ is also convinced that the log entries are authentic, since a correct MAC on any hash-chain value is essentially a MAC on all previous entries as well.

(10) $\mathcal{V}$ deletes the key it established with $\mathcal{T}$ in Step (3). This guarantees that an attacker cannot read $\mathcal{U}$'s logfile if $\mathcal{V}$ is compromised later.

Of course, if $\mathcal{V}$ is compromised at the start of this protocol with $\mathcal{T}$, it will be able to read $\mathcal{U}$'s logfile. Presumably, $\mathcal{V}$ will have its own authenticated logfiles and will be regularly audited to reduce the likelihood of this problem occurring.

4. APPLICATIONS AND EXTENSIONS

4.1 Abnormal Shutdowns

In some systems, abnormal shutdowns (e.g., caused by a system crash or power outage) are a normal part of operations. The mechanism for closing the log file doesn't work in this situation—it will be impossible to distinguish the log of a system that crashed at 12:10 PM and the log of a system that was taken over at 2:00 PM, and then had all records entered since 12:10 PM truncated.

To deal with this, we change our routine after writing a log entry slightly: After we compute the new A_j key value, we generate a special Abnormal Shutdown message and store it in nonvolatile storage. At the same time as we write the next log entry, we irretrieveably delete this message from nonvolatile storage. (It is critical that it be irretrieveably deleted; otherwise, an attacker can silently truncate the log.)

When $\mathcal{U}$ "wakes up" after a system crash, it copies the Abnormal Shutdown message to the log file. An attacker who takes $\mathcal{U}$ over is able to do the same thing, but is *not* able to delete any log entries in this way. (He can always *delete* the log entries, but he cannot do so without the fact being detected later.)

Note that in real-world systems, there will probably not be an atomic operation to delete the previous Abnormal Shutdown message, write the next log entry, and generate and store a new Abnormal Shutdown message. However, this can be made to happen in a very short time window, so that the likelihood of a system crash during the operation is very small.

Real-world systems may also have to deal with the difficulty of really deleting data from some nonvolatile media. If this is a major issue, we may end up storing these Abnormal Shutdown messages encrypted, and also storing the keys. We can then attempt to overwrite the stored key many times in succession after each new log entry is written.

4.2 An Off-Line Variant

In the protocols and message formats given in Section 3.3, we left the specifics of the timing of M_1 from $\mathcal{T}$ to $\mathcal{U}$ open. This allows us to create a completely off-line variant using couriers as the only communications medium. Thus, M_0 and M_1 can be sent via courier on diskettes. If a voice channel is available, someone can also read the hash of M_0 from $\mathcal{U}$ over the line for additional security.

4.3 Voice Line Only

It is also possible to reduce the protocols for starting up a log file and to verify messages that can be sent over a voice line directly, either by reading them over the line or by using DTMF tones. For the startup protocol (Section 3.2), ID_U, ID_{log}, $hash(M_0)$ must be read over the line. In practice, this can probably be reduced down to 22 digits (with the SHA-1 hash reduced to only 20 digits).[5] This will provide resistance against practical attacks that do not involve compromise of $\mathcal{U}$ before the logfile is created. For the log-entry protocol (Section 3.1), we must send the last entry of the hash chain (about 40 digits) and 20 digits of the MAC. This would be impractical for a human to handle via a phone keypad, but might be done by voice-recognition, perhaps involving some additional correction digits.

This variant might be useful for applications in which some piece of equipment doesn't have any direct access to communications with the outside world, but which has a keyboard and display. A human user using a telephone, cellphone, or satellite phone could then start up the audit logs. For example, we might have the processor inside a vending machine maintain such logs. It might be useful to allow a user to verify the contents of the logfile with a personal digital assistant (such as a PalmPilot) and a cellphone.

4.4 Cross-Peer Linkages: Building a Hash Mesh

If there are multiple instances of $\mathcal{U}$ executing this same protocol, they can cross-link their audit logs with each other. In applications where there are many instances of $\mathcal{T}$ and with different instances of $\mathcal{U}$ authenticating their log files with different instances of $\mathcal{T}$, this cross-linking can make back-alteration to audit logs impractical, even with one or more compromised instances of $\mathcal{U}$ or even (in some cases) $\mathcal{T}$. This will also decrease the freedom of any compromised $\mathcal{U}$ machine to alter logs, because it keeps having to commit to its log's current state on other uncompromised machines.

This cross-authentication is done in addition to the rest of the scheme as described above. To cross-authenticate between two untrusted machines, $\mathcal{U}_0$ and $\mathcal{U}_1$, we execute the following protocol.

(1) $\mathcal{U}_0$ and $\mathcal{U}_1$ exchange identities and establish a secure connection.

(2) $\mathcal{U}_0$ forms and enters into its own log an entry, in position j, with:

$W_j =$ **"Cross Authentication Send"**,

$D_j =$ "Cross Authentication Send", $ID_{\mathcal{U}_1}$, d_0,

where d_0 is $\mathcal{U}_0$'s timestamp.

(3) $\mathcal{U}_0$ forms and sends to $\mathcal{U}_1$:

$M_4 = p, Y_j, d_0.$

Recall that Y_j is the current value of $\mathcal{U}_0$'s hash chain.

(4) $\mathcal{U}_1$ receives this message, and verifies that the timestamp is approximately correct. If so, $\mathcal{U}_1$ writes a log entry in position i with:

$W_i =$ **"Cross Authentication Receive"**,

$D_i =$ "Cross Authentication Receive", $ID_{\mathcal{U}_0}$, d_0, Y_j.

[5] We get moderate resistance to targeted collision attacks, but not to free collision atacks, with messages of 20 decimal digits.

Then $\mathcal{U}_1$ forms and sends to $ID_{\mathcal{U}_0}$
$M_5 = p, Y_i$.

If $\mathcal{U}_1$ doesn't agree with the timestamp, it writes a log entry in position i with:
$W_i =$ **"Cross Authentication Receive Error**,
$D_i =$ "Cross Authentication Receive Error", $ID_{\mathcal{U}_0}$, d_0, d_1, Y_j,

where d_1 is $\mathcal{U}_1$'s timestamp. Then $\mathcal{U}_1$ forms and sends to $\mathcal{U}_0$:
$M_6 = p, Y_i, ErrorCode$.

The protocol is then terminated.

(5) $\mathcal{U}_0$ receives M_6 and processes it. If there was no error in receiving M_6, and if M_6 was not an error message, then $\mathcal{U}_0$ writes an entry to the log at position $j + 1$ with:
$W_{j+1} =$ **"Cross Authentication Reply"**,
$D_{j+1} =$ "Cross Authentication Reply", $ID_{\mathcal{U}_1}$, Y_i.

If it was an error, or if the expected message doesn't arrive before time-out, then $\mathcal{U}_0$ writes an entry to the log at position $j + 1$ with:
$W_{j+1} =$ **"Cross Authentication Reply Error"**,
$D_{j+1} =$ "Cross Authentication Reply Error", $ID_{\mathcal{U}_1}$, ErrorCode.

If mutual cross-peer linking is required, $\mathcal{U}_1$ could then initiate this same protocol with $\mathcal{U}_0$.

Cross linkages are a way to interweave the audit logs of different $\mathcal{U}$s, so that it would be harder to erase the history of a particular $\mathcal{U}_i$. For example, consider a network of electronic wallets: smart cards, PC-Cards), PalmPilot-like PDAs, etc. The wallets, $\mathcal{U}$s, would each keep their own audit log, and download it to a banking terminal, $\mathcal{T}$, whenever the two interacted. (Presumably, the wallets would only interact with the banking terminals occasionally: to upload or download money, as part of a regular audit cycle, etc.)

A wallet-to-wallet transfer protocol could make use of this system of cross-peer linkages, and exchange audit entries every time two wallets interacted. Pieces of the resulting hash mesh would be downloaded to $\mathcal{T}$ whenever a wallet interacted with $\mathcal{T}$. Even if a particular wallet, $\mathcal{U}_0$, did not interact with $\mathcal{T}$ for a long time, the other wallets that it did interact with would interact with $\mathcal{T}$, allowing $\mathcal{T}$ to at least partially reconstruct what happend to $\mathcal{U}_0$. And if the tamper-resistance in $\mathcal{U}_0$ was defeated, the other entries in the hash lattice would help the bank reconstruct any fraudulent transactions and trace what had happened.

This system of cross-linking audit trails also has applications where multiple audit trails are on the same physical device. For example, a smart card might run several different commerce applications, or a firewall might run several different security applications. In this situation, it would make sense for each application to keep its own audit log (the hardware itself might keep a separate audit log), and that they would all interact using the hash lattice protocol. In this way, the interacting audit logs could provide evidence even in the event of one audit log being altered or destroyed.

4.5 Replacing $\mathcal{T}$ with a Network of Insecure Peers

We can run this whole scheme using multiple untrusted machines, $\mathcal{U}_0$, $\mathcal{U}_1$, ..., $\mathcal{U}_n$, to do all the tasks of $\mathcal{T}$. Since $\mathcal{T}$ is a huge target, this could potentially increase security. Basically, this involves an extension of the hash lattice ideas from the previous section.

Let $\mathcal{U}_0$ and $\mathcal{U}_1$ both be untrusted machines, with $\mathcal{U}_0$ about to start an audit log. $\mathcal{U}_1$ will serve as the trusted server for this audit log.

(1) $\mathcal{U}_0$ and $\mathcal{U}_1$ establish a secure connection.

(2) $\mathcal{U}_0$ forms:

d, a current timestamp.
d^+, timestamp at which $\mathcal{U}_0$ will time out.
ID_{log}, a unique identifier for this logfile.
A_0, a random starting point.
$ID_{\mathcal{U}_0}$, $ID_{\mathcal{U}_1}$ are unique identifiers for U_0 and U_1, respectively.
$X_0 = p, ID_{\mathcal{U}_0}, ID_{\mathcal{U}_1}, d, ID_{log}, A_0$.

She then forms and sends to $\mathcal{U}_1$:
$M_0 = X_0$.

(3) $\mathcal{U}_0$ forms the first log entry with:
$W_0 =$ **LogfileInitializationType**,
$D_0 = d, d^+, ID_{log}, M_0$.

Again, $\mathcal{U}_0$ does not store A_0 in the clear, to protect against a replay attack. $\mathcal{U}_0$ also calculates and stores $hash(X_0)$ locally while waiting for the response message from $\mathcal{U}_1$.

(4) $\mathcal{U}_1$ receives and verifies that M_0 is well formed. If it is, then $\mathcal{U}_1$ forms:
$X_1 = p, ID_{log}, hash(X_0)$.

$\mathcal{U}_1$ then forms and sends to $\mathcal{U}_0$:
$M_1 = X_1$.

(5) $\mathcal{U}_0$ receives and verifies M_1. If all is well, then $\mathcal{U}_0$ forms a new log entry with:
$W_0 =$ **ResponseMessageType**,
$D_j = M_1$.

If $\mathcal{U}_0$ doesn't receive M_1 by the time-out time d^+, or if M_1 is invalid, then $\mathcal{U}_0$ forms a new log entry with:
$W_0 =$ **AbnormalCloseType**,
$D_j =$ the current timestamp and the reason for closure.

The log file is then closed.

One potential issue here is that an attacker may compromise $\mathcal{U}_1$, allowing the wholesale alteration of entries in the logfile. There are two imperfect solutions to this:

$\mathcal{U}_0$ should log the same data in several parallel logfiles, with each logfile using a different untrusted server as its trusted server.

$\mathcal{U}_0$ may commit, in the first protocol message, to a number N that will be the index number of its final log entry. $\mathcal{U}_1$ then computes A_N and $K_{0..N}$, stores these values, and deletes A_0. If an attacker compromises $\mathcal{U}_1$, he will learn what he needs to read and to close the logfile on $\mathcal{U}_0$, but not what he needs to alter any other log entries.

It is worth noting that these measures do not protect the secrecy of a logfile once an attacker has compromised both $\mathcal{U}_0$ and $\mathcal{U}_1$. An improved solution is for $\mathcal{U}_0$ to use a secret-sharing scheme to store A_0 among n untrusted machines, any m of which could then recover it.

Another solution is for $\mathcal{U}_0$ to keep parallel log files in the manner described above on n machines, but generating (for each log entry that needed to be kept secret) $n - 1$ random bit strings, P_i of length equal to that of D_j. $\mathcal{U}_0$ then stores $D_j \oplus P_0 \oplus P_1 \oplus \ldots \oplus P_{N-2}$ in one logfile, and each pad value in another logfile.

In practice, these kinds of distributed schemes seem to work better for authenticating the log entries than for protecting their secrecy.

This kind of redundancy has considerable benefit in applications where even the trusted computers are not very trusted: e.g., electronic wallet applications where the $\mathcal{U}$s are customer cards and the $\mathcal{T}$s are merchant terminals, or vending applications where the $\mathcal{U}$s are customer cards and the $\mathcal{T}$s are vending machines (or parking garage machines, or public transportation terminals, or pay telephones) that may be in remote locations. It also can serve as an extra measure of security in applications where $\mathcal{T}$s are trusted.

5. USING THE AUDIT LOG AS A FORENSIC TOOL

The primary benefit of our complicated protocol for generating audit logs and audit-log entries is that it aids in forensic analysis [Wilding 1997; Schneier and Kelsey 1999]. The following discussion assumes that audit log entries detect an intrusion (record the opening of a door, removal of a tamper-resistant coating, access of a normally secret file, and so on). If an attacker can gain control of $\mathcal{U}$ *without* triggering an alarm condition and associated audit-log entry, then this system cannot help.

Audit logs are useless unless someone reads them. Hence, we first assume that there is a software program whose job it is to scan all audit logs and look for suspicious entries. In our system, there are two types of suspicious entries. First, there are valid entries that indicate an intrusion (e.g., any of the alarm conditions mentioned in the previous paragraph). Second, there are invalid entries that indicate that the audit log has been tampered with. Since an attacker who gains access to a device with this type of logging has only two options—leave the incriminating log entries in the log or delete them and ensure that the deletion will be noticed—one of these two suspicious entry types will indicate a break-in.

After that, the details are completely dependent on the particular log entries. If there is an invalid entry, one can immediately assume that all entries afer the last valid one are suspect and that all entries before the first invalid one are genuine.

6. SUMMARY AND CONCLUSIONS

Many security systems, whether they protect privacy, secure electronic-commerce transactions, or use cryptography for something else, do not directly prevent fraud. Rather, they detect attempts at fraud after the fact, provide evidence of that fraud in order to convict the guilty in a court of law, and assume that the legal system will provide a "back channel" to deter further attempts. We believe that fielded systems should recognize this fundamental need for detection mechanisms, and provide audit capabilities that can survive both successful and unsuccessful attacks. Additionally, an unalterable log should make it difficult for attackers to cover their tracks, meaning that the victims of the attack can quickly learn that their machine has been attacked, and take measures to contain the damage from that attack. The victims could then revoke some public-key certificates, inform users that their data may have been compromised, wipe the machine's storage devices and restore it from a clean backup, or improve physical and network security on the machine to prevent further attacks.

In this paper, we have presented a general scheme that allows keeping an audit log on an insecure machine, so that log entries are protected even if the machine is compromised. We have shown several variations of this scheme, including one that is suitable for multiple electronic wallets interacting with each other but not connected to a central secure network. This scheme, combined with physical tamper-resistance and periodic inspection of the insecure machines, could form the basis for highly trusted auditing capabilities. Our technique is strictly more powerful than simply periodically submitting audit logs and log entries to a trusted time-stamping server [Haber and Stornetta 1991]: the per-record encryption keys and permission masks permit selective disclosure of log information, and there is some protection against denial-of-service attacks against the communications link between the insecure machine and the trusted server.

The primary limitation of this work is that an attacker can seize control of an insecure machine and simply continue creating log entries, without trying to delete or change any previous log entries. In any real system, we envision log entries for things like "Tamper-resistance breach attempt" that any successful attacker will want to remove. Even so, the possibility of an unlogged successful attack makes it impossible to be certain that a machine was uncompromised before a given time. A sufficiently sneaky attacker might even create log entries for a phony attack hours after the real, unlogged, compromise.

In future work, we intend to expand this scheme to deal with the multiparty situation more cleanly. For example, we might like to be able to specify any three of some group of five nodes to play the part of the trusted machine. While this is clearly possible, we have not yet worked out the specific protocols. We also might want to use ideas from the Rampart system [Reiter 1996] to facilitate distributed trust. Also, it would be useful to anonymize the communications and protocols between an untrusted machine and several of its peers, which are playing the part of the trusted machine in our scheme. This would make it harder for an attacker to compromise one machine, and then learn from it which other machines to compromise in order to be able to violate the log's security on the first compromised machine.

ACKNOWLEDGEMENTS

The authors would like to thank Hilarie Orman and David Wagner, as well as the anonymous USENIX Security Symposium and TISSEC referees, for their helpful comments; and David Wagner, again, for the LaTeX figure. This work is patent pending in the United States and other countries.

REFERENCES

ANDERSON, R. AND NEEDHAM, R. 1995. Robustness principles for public key protocols. In *Proceedings of the Conference on Advances in Cryptology* (CRYPTO '95). Springer-Verlag, New York, NY, 236–247.

ANDERSON, R. AND KUHN, M. 1996. Tamper resistance: A cautionary note. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce* (Nov.). USENIX Assoc., Berkeley, CA, 1–11.

BELLARE, M., CANETTI, R., AND KRAWCYZK, H. 1996. Keying hash functions for message authentication. In *Advances in Cryptology* (CRYPTO '96, Santa Barbara, Calif.), N. Koblitz, Ed. Springer-Verlag, New York, 1–15.

DIFFIE, W., VAN OORSCHOT, P. C., AND WIENER, M. J. 1992. Authentication and authenticated key exchanges. *Des. Codes Cryptography 2*, 2 (June 1992), 107–125.

DOBBERTIN, H., BOSSELAERS, A., AND PRENEEL, B. 1996. RIPEMD-160: A strengthened version of RIPEMD. In *Proceedings of the 3rd International Workshop on Fast Software Encryption*. Springer-Verlag, New York, NY, 71–82.

ELGAMAL, T. 1985. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inf. Theor. IT-31*, 4, 469–472.

HABER, S. AND STORNETTA, W. S. 1991. How to time stamp a digital document. In *Advances in Cryptology* (CRYPTO '90). Springer-Verlag, New York, NY, 437–455.

KELSEY, J. AND SCHNEIER, B. 1996. Authenticating outputs of computer software using a cryptographic coprocessor. In *Proceedings of the 1996 CARDIS* (Sept.). 11–24.

KELSEY, J., SCHNEIER, B., AND HALL, C. 1996. An authenticated camera. In *Proceedings of the 12th Annual Conference on Computer Security Applications*. IEEE Computer Society Press, Los Alamitos, CA, 24–30.

KELSEY, J., SCHNEIER, B., AND WAGNER, D. 1998. Protocol interactions and the chosen protocol attack. In *Proceedings of the 1997 Workshop on Protocols*. Springer-Verlag, New York, NY, 91–104.

LAI, X., MASSEY, J., AND MURPHY, S. 1991. Markov ciphers and differential crytanalysis. In *Advances in Cryptology* (CRYPTO '91). Springer-Verlag, New York, NY, 17–38.

MCCORMAC, J. 1996. *European Scrambling Systems*. Waterford University Press.

MENEZES, A. J., VAN OORSCHOT, P. C., AND VANSTONE, S. A. 1997. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL.

NBS, 1977. NBS FIPS PUB 46, Data Encryption Standard. U.S. Department of Commerce.

NIST, 1993. *NIST FIPS PUB 180, Secure Hash Standard*. U.S. Department of Commerce.

NIST, 1994. *NIST FIPS PUB 186, Digital Signature Standard*. U.S. Department of Commerce.

REITER, M. 1996. Distributing trust with the Rampart toolkit. *Commun. ACM 39*, 4, 71–74.

RIORDAN, J. AND SCHNEIER, B. 1998. Environmental key generation towards clueless agents. In *Mobile Agents and Security*, G. Vigna, Ed. Springer-Verlag, New York, NY, 15–24.

RIVEST, R., SHAMIR, A., AND ADELMAN, L. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM 21*, 2 (Feb.), 120–126.

SCHNEIER, B. 1994. Description of a new variable-length key, 64-bit block cipher (blowfish): Fast software encryption. In *Proceedings of the Cambridge Security Workshop*. Springer-Verlag, New York, NY, 191–204.

SCHNEIER, B. 1995. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. 2ND John Wiley & Sons, Inc., New York, NY.

SCHNEIER, B. AND KELSEY, J. 1997. Automatic event-stream notarization using digital signatures. In *Proceedings of the International Workshop on Security Protocols* (Cambridge, U.K., Apr.). Springer-Verlag, New York, NY, 155–169.

SCHNEIER, B. AND KELSEY, J. 1997. Remote auditing of software outputs using a trusted coprocessor. *Future Gener. Comput. Syst. 13*, 1, 9–18.

SCHNEIER, B. AND KELSEY, J. 1998. Cryptographic support for secure logs on untrusted machines. In *Proceedings of the 7th on USENIX Security Symposium* (Jan.). USENIX Assoc., Berkeley, CA, 53–62.

SCHNEIER, B. AND KELSEY, J. 1999. Tamperproof audit logs as a forensics tool for intrusion detection systems. *Comput. Networks ISDN Syst. 31*.

STINSON, D. R. 1995. *Cryptography: Theory and Practice*. 1st CRC Press, Inc., Boca Raton, FL.

STOLL, C. 1989. *The Cuckoo's Egg: Tracking a Spy through the Maze of Computer Espionage*. Doubleday, New York, NY.

WILDING, E. 1997. Computer forensics: Trends and concerns. *Inf. Sec. Bull. 2*, 6 (Dec.), 15–18.

Received: August 1997; revised: January 1999; accepted: February 1999