Cryptographic Support for Secure Logs on Untrusted Machines

Bruce Schneier John Kelsey
{schneier,kelsey}@counterpane.com
Counterpane Systems, 101 East Minnehaha Parkway, Minneapolis, MN 55419

Abstract

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 undetectably modify or destroy.

1 Introduction

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 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, etc.—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. 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 undeleteable 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. A secure digital camera needs to guarantee the authenticity of pictures taken, even if it is reverse-engineered sometime later [KSH96]. 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. 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

[1] In [Sto89], Cliff Stoll attached a printer to a network computer for just this purpose.

benefit from the ability to resist alteration of their logs even when they're running under the control of a malicious adversary.

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 cryptographichic 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, and a solution to do so 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 essense, this technique is an implementation of an engineering tradeoff between how "online" $\mathcal{U}$ 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 online nearly all the time. In many systems, $\mathcal{U}$ is not expected to be compromised nearly as often, and is also not online 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 T may be compromised. The audit log technique in our paper enables this tradeoff. It provides a "knob" that the system architect can adjust based on his judgement of this tradeoff; furthermore, the knob can be adjusted during the operation of the system as expectations of the rate of compromise change.

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. Finally, in Section 5 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 [RSA78] or ElGamal [ElG85].
3. $SIGN_{SK_x}(Z)$ is the digital signature, under x's private key, of Z, using an algorithm such as RSA or DSA [NIST94].
4. $E_{K_0}(X)$ is the symmetric encryption of X under key K_0, using an algorithm such as DES [NBS77], IDEA [LMM91], or Blowfish [Sch94].
5. $MAC_{K_0}(X)$ is the symmetric message authentication code (HMAC or NMAC [BCK96]), under key K_0, of X.
6. $hash(X)$ is the one-way hash, using an algorithm such as SHA-1 [NIST93] or RIPE-MD [DBP96], of X.
7. X, Y represents the concatenation of X with Y.

Descriptions of most of these algorithms are in [Sti95, Sch96, MOV97].

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 [And95, KSW97]. 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. 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 machine, etc.
2. $\mathcal{U}$ is the untrusted machine, on which the log is to be kept.
3. $\mathcal{V}$ is a moderately-trusted verifier, who will be trusted to review certain kinds of records, but not trusted with the ability to change records. Note that not all of our implementations will be able to support $\mathcal{V}$.

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. Methodologies for all of these are described in great detail elsewhere: see [Sti95, Sch96, MOV97].

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 [HS91, SK97]. 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.
4. Each log entry contains its own permission mask. This permission mask defines roles in a role-based security scheme; partially-trusted users can be given access to only some 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

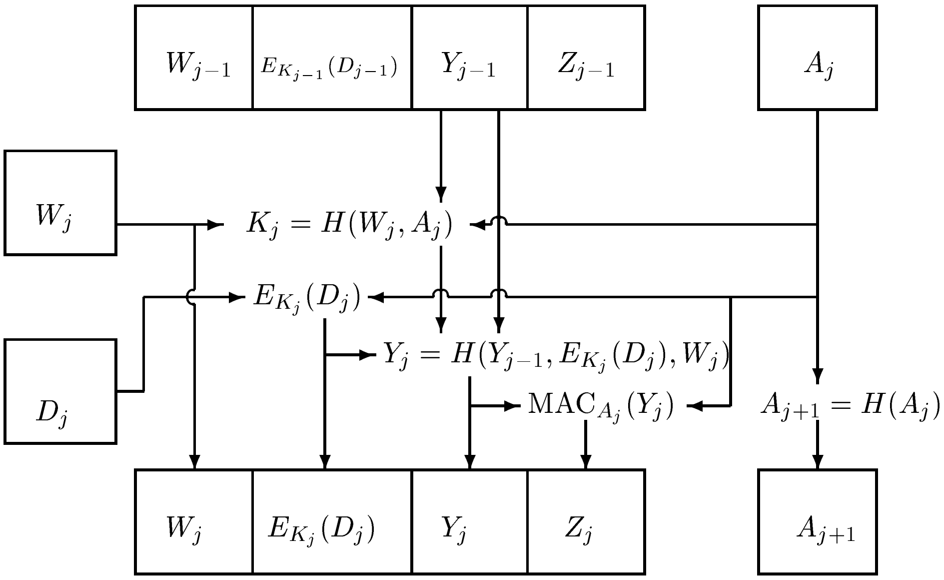

Figure 1: Adding an entry to the log.

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 Verifier 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 which 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 twenty-byte block of binary zeros.[2]

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. See Figure 1 for 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 entry. He can delete a block of entries (or the entire log file), but he cannot create new entries, either past entries to replace them or future entries. 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. [3]

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

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

[3] 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.

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 $A_1 = hash(\text{“Increment Hash”}, A_0)$.

   If $\mathcal{U}$ doesn't receive M_1 by the timeout time d^+, or if M_1 is invalid, then $\mathcal{U}$ forms a new log entry with W_j = **AbnormalCloseType** and D_j = 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. 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 high-bandwidth 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 log. $\mathcal{T}$ knows A_0 so he can calculate A_f; this allows him to verifies 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 responding 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 occuring.

4 Applications and Extensions

An Offline 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 offline 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.

Voice Line Only It is also possible to reduce the protocol in Section 3.3 to messages that can be sent over a voice line directly, either by reading them over the line or by using DTMF tones. In this case, $ID_U, ID_{log}, hash(M_0)$ 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).[4] This will provide resistance against practical attacks that do not involve compromise of $\mathcal{U}$ before the logfile is created.

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

4.1 Cross-Peer Linkages: Building a Hash Lattice

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.

   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 timeout, 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 $\mathcal{U}_0$.

This protocol could be useful in a network of electronic wallets. The wallets, $\mathcal{U}$s, would exchange money with each other regularly and occasionally come in contact with a banking terminal $\mathcal{T}$. This hash lattice could help the bank reconstruct fradulant transactions and trace wallets whose tamper-resistance has been defeated.

4.2 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 timeout 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 of length equal to that of D_j. $\mathcal{U}_0$ then stores $D_j \oplus Pad_0 \oplus Pad_1 \oplus ... \oplus Pad_{N-2}$ 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.

5 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 to 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 [HS91]: the per-record encryption keys and permission masks permit selective disclosure of log information, and there is some protection against dential-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 sieze 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 attmpt" that any successful attacker will want to remove. Even so, the possibility of an unlogged successful attack make 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 to use ideas from the Rampart system [Rei96] 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.

6 Acknowledgments

The authors would like to thank Hilarie Orman and David Wagner, as well as the anonymous USENIX Security Sumposium 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

[And95] R. Anderson, "Robustness Principles for Public Key Protocols," *Advances in Cryptology — CRYPTO '95,* Springer-Verlag, 1995, pp. 236–247.

[BCK96] M. Bellare, R. Canetti, and H. Krawcyzk, "Keying Hash Functions for Message Authentication," *Advances in Cryptology — CRYPTO '96,* Springer-Verlag, 1996, pp. 1–15.

[DBP96] H. Dobbertin, A. Bosselaers, and B. Preneel, "RIPEMD-160: A Strengthened Version of RIEPMD," *Fast Software Encryption, Third International Workshop,* Springer-Verlag, 1996, pp. 71–82.

[ElG85] T. ElGamal, "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *IEEE Transactions on Information Theory,* V. IT-31, n. 4, 1985, pp. 469–472.

[HS91] S. Haber and W.S. Stornetta, "How to Time Stamp a Digital Document," *Advances in Cryptology — Crypto '90*, Springer-Verlag, 1991, pp. 437–455.

[KSW97] J. Kelsey, B. Schneier, and D. Wagner, "Protocol Interactions and the Chosen Protocol Attack," *1997 Protocols Workshop*, Springer-Verlag, 1997, to appear.

[KSH96] J. Kelsey, B. Schneier, and C. Hall, "An Authenticated Camera," *Twelfth Annual Computer Security Applications Conference,* IEEE Computer Society Press, 1996, pp. 24–30.

[LMM91] X. Lai, J. Massey, and S. Murphy, "Markov Ciphers and Differential Cryptanalysis," *Advances in Cryptology — CRYPTO '91*, Springer-Verlag, 1991, pp. 17–38.

[MOV97] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.

[NBS77] National Bureau of Standards, NBS FIPS PUB 46, "Data Encryption Standard," National Bureau of Standards, U.S. Department of Commerce, Jan 1977.

[NIST93] National Institute of Standards and Technology, NIST FIPS PUB 180, "Secure Hash Standard," U.S. Department of Commerce, May 1993.

[NIST94] National Institute of Standards and Technologies, NIST FIPS PUB 186, "Digital Signature Standard," U.S. Department of Commerce, May 1994.

[Rei96] M. K. Reiter, "Distributing trust with the Rampart toolkit," *Communications of the ACM*, v. 39, n. 4, Apr 1996, pp. 71–74.

[RSA78] R. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, v. 21, n. 2, Feb 1978, pp. 120–126.

[Sch94] B. Schneier, "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)," *Fast Software Encryption, Cambridge Security Workshop Proceedings*, Springer-Verlag, 1994, pp. 191-204.

[Sch96] B. Schneier, *Applied Cryptography, Second Edition*, John Wiley & Sons, 1996.

[SK97] B. Schneier and J. Kelsey, "Automatic Event-Stream Notorization Using Digital Signatures," *Security Protocols, International Workshop, Cambridge, United Kingdom, April 1996 Proceedings*, Springer-Verlag, 1997, pp. 155–169.

[Sto89] C. Stoll, *The Cuckoo's Egg*, Doubleday, New York, 1989.

[Sti95] D.R. Stinson, *Cryptography: Theory and Practice*, CRC Press, 1995.