LOGCRYPT: FORWARD SECURITY AND PUBLIC VERIFICATION FOR SECURE AUDIT LOGS

by

Jason E. Holt

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

April 2005

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Jason E. Holt

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Kent E. Seamons, Chair
Date	Rodney Forcade
Date	Scott Woodfield

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Jason E. Holt in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Kent E. Seamons
Chair, Graduate Committee

Accepted for the Department

David W. Embley
Department Chair

Accepted for the College

G. Rex Bryce
Associate Dean, College of Physical and Mathematical Sciences

ABSTRACT

LOGCRYPT: FORWARD SECURITY AND PUBLIC VERIFICATION FOR SECURE AUDIT LOGS

Jason E. Holt

Department of Computer Science

Master of Science

Logcrypt provides strong cryptographic assurances that data stored by a logging facility before a system compromise cannot be modified after the compromise without detection. We build on prior work by showing how log creation can be separated from log verification, and describing several additional performance and convenience features not previously considered.

ACKNOWLEDGMENTS

Thanks to my advisor, Dr. Kent Seamons, for giving me so much space to explore and learn both in my life and at work. Hilarie Orman and Rich Schroeppel gave much needed feedback and guidance throughout all my research, while Hans Reiser and Roger Dingledine provided the impetus for reinventing this particular construction. Coila Graham, Brett Rasmussen, Sonny Cook and Catherine Meyers all helped me understand and enjoy my life enough to get work done, and the BYU Ballroom Dance program kept me sane. Thanks to my family for always encouraging me to get an education.

Financial support for this research was provided by DARPA through AFRL contract number F33615-01-C-0336 and through Space and Naval Warfare Systems Center San Diego grant number N66001-01-1-8908. Support was also provided by the National Science Foundation under grant number CCR-0325951 and prime cooperative agreement number IIS-0331707, and The Regents of the University of California.

Contents

List of Tables

List of Figures

Chapter 1 — Introduction

The popular application Tripwire keeps cryptographic fingerprints of all files on a computer, allowing administrators to detect when attackers compromise the system and modify important system files. But Tripwire is unsuitable for system logs and other files that change often, since the fingerprints it creates apply to files in their entirety. Several people have proposed cryptographic systems which allow each new log entry to be fingerprinted, preventing attackers from removing evidence of their attacks from system logs.

In the systems described first by Bellare and Yee [3] and then Schneier and Kelsey [13], a small secret is established at log creation time and stored somewhere safe, such as on a slip of paper locked in a safe or on a separate, trusted computer. The secret stored on the computer is the head of a hash chain, changing via a cryptographic one-way function every time an entry is written to the log. This secret is used to compute a cryptographic message authenticaion code (MAC) for the log each time an entry is added, and optionally to encrypt the log as well.

If the system is compromised, the attacker has no way to recover the secrets used to create the MACs or decryption keys for entries in the log which have already been completed. He can delete the log entirely, but can't modify it without detection. Later, the administrator can use the original secret to recreate the hash chain and check whether the logs are still intact. To keep an attacker from interfering with this process, this should happen on a separate, secure machine.

MACs may also be sent to another machine as they're written; then they can serve as commitments to log entries. A radiologist, for instance, could send MACs for each MRI image she creates to an auditing agency. Later, she could produce the

images in court and the auditor could vouch that the images she presented match the MACs she sent out. But otherwise, the auditor would have no way of knowing what the images were. The radiologist is protected from accusations of fraud, and the patient's privacy is protected.

Logcrypt builds on the Schneier and Kelsey system, adding several significant improvements. The most significant is the ability to use public key cryptography with Logcrypt. Using the symmetric techniques just described, any entity who wishes to verify a log must possess the secret used to create the MACs. This secret gives the entity the ability to falsify log entries as well, which could be a major drawback in many applications. Public key cryptography allows signatures to be created with one key and verified with a different one. Such signatures can be used in place of MACs to allow verification of a log without the ability to modify it, as well as allowing publication of the initial key used to create the log, since only the public key is needed for verification.

Other improvements we describe include a method of securing multiple, concurrently maintained logs using a single initial value, and a method of aggregating multiple log entries to reduce latency and computational load.

Chapter 2 — Applications

Logcrypt can provide data integrity and secrecy in a wide range of applications. The most obvious and simple application is in protecting system logs on Internet servers. Such servers are always in danger of compromise, and skilled attackers can generally make detection after the fact quite difficult[15] for system administrators. Logcrypt makes detection much more feasible in systems which can successfully record compromises as they happen by removing the secret used to record each entry as soon as it is used. If the logging facility is properly configured, attack attempts may be recorded before the attacker even completes the system compromise. Entries made concurrently with the intrusion can be logged within milliseconds, giving attackers a very small window in which to subvert the logging system.

The data integrity offered by Logcrypt is particularly useful for allowing auditors outside a system to make sure no tampering takes place within the system. For instance, handguns used by police officers could be fitted with a camera which takes a picture each time the gun is fired, then stores the image with a Logcrypt MAC. Later, the pictures can be used for forensic analysis of a crime scene. Officers are protected against accusations of tampering, since MACs cannot be forged later, even with access to the internals of the device.

Logcrypt secrecy allows data to be stored "write-only". For instance, photographers employed by news agencies sometimes take pictures which are embarassing to the government of the country in which they take them. This can place photographers in significant personal danger. A photographer using a Logcrypt-equipped camera, however, could establish a secret with the home office before leaving for his assignment. Logcrypt then encrypts each image a few seconds after it is recorded.

Even the photographer himself will be unable to view the pictures he takes until he transmits the data to the home office.

Audio recorders can make use of both the secrecy and integrity provided by Logcrypt. A journalist taking voice notes or recording interviews could benefit from secrecy in the same way as the photographer in the last example. Secrecy also protects audio entries from thieves and unscrupulous officials. Message integrity ensures that the interview isn't edited later without detection.

Publicly verifiable logs can be used for systems which need to be publicly audited, such as financial books for publicly held companies and voting systems in democratic countries. When such logs are properly created and their initial public keys sent to external auditors, not even their creators can go back and change entries once they're entered. For example, an honest system administrator could set up a log to record all financial bookkeeping entries for a company, sending the initial public key to external auditors before the first entry arrives. After each entry is recorded, the private key used to create it is destroyed automatically. Later, the CFO approaches the administrator and demands that certain entries be replaced to hide poor quarterly results. But the administrator has no power to do so – the private keys for the existing entries have been deleted, and the auditing agency will be able to detect any missing or modified entries if it ever verifies the log. Of course, the CFO can prevent *future* entries from being recorded properly (or even reaching the system), but existing entries are irrevocably tamper-evident.

Chapter 3 — Related Work

In 1995, Futoransky and Kargieman [8][7] proposed the fundamental technique on which Logcrypt is built. In 1997, Bellare and Yee [3] published a more theoretical paper based on a very similar technique. Their systems closely relate to the simple system we present in section 5. Futoransky and Kargieman's work led to MSyslog [11], an open source implementation of the Unix syslog service with integrity protection. Bellare and Yee mentioned the idea of forward security using signatures, and in 1999 proposed a forward secure signature scheme [1], but did not further discuss its application to secure audit logs.

Schneier and Kelsey proposed another similar system [13][14][10]. Their system uses the same fundamental construct, but gives a precise protocol for its implementation in a distributed system, describing how messages are sent to external machines upon log creation and closing. Their system also closely relates to the simple system we present in section 5, but neither system addresses public verifiability, metronome entries, multiple concurrent logs, or high load conditions.

Chong, Peng and Hartel discussed the possibilities offered by tamper-resistant hardware in conjunction with a system like Schneier and Kelsey's in [5], and implemented their system on an iButton.

Waters et al described how identity-based encryption can be used to make audit logs efficiently searchable in [16]. Keywords which relate to each log entry are used to form IBE public keys with which the entry's key is encrypted. Administrators allow searching and retrieval of entries matching a given set of keywords by issuing clients the corresponding IBE private keys.

Chapter 4 — Overview

Assuming Logcrypt's design, construction and underlying cryptographic primitives are sound, then if a system is secure from tampering at a time t, and the computational overhead from Logcrypt's cryptographic operations takes l milliseconds, then log entries created until time $t-l$ will have forward secrecy in the sense that tampering will be detectable with overwhelming probability.

Three elements of Logcrypt form the foundation of its security:

1. Logs begin in a known state which is recorded in a secure external system.

2. The security of an earlier state can be used to verify the integrity of a later state, assuming the system is secure in both states.

3. Once a secret is used to secure a log entry, it is erased from memory as soon as possible.

Hash chains make it easy to fulfill these requirements. In a hash chain, a secret s is hashed by a cryptographically strong one-way function to produce the next links in the chain, $s_1 = h(s)$, $s_2 = h(s_1)$, etc. One-wayness means it is assumed to be computationally infeasible to find s given s_1, even though calculating $s_1 = h(s)$ is generally quite efficient.

The simple system we propose is essentially a simplification of the system described in [13]. We first define our simple system, then give the public-key variant, and then describe several other features relevant to real systems.

We have to be careful in describing the assurances our system makes. Once an attacker gains complete control over a system, he can control virtually everything that happens from that point. Consequently, our system provides tamper-evidence

by removing secrets from the system as soon as possible: once a secret has been destroyed, that secret can effectively protect information through cryptography.

It is also worthwhile to consider the difference between logs which merely reside on a system and logs which are used to detect attempts at compromising the system. In the latter case, the latency l can make the difference between an attack being recorded in a tamper-evident log and an attack in which all the evidence is removed. Logcrypt can have values of l in low milliseconds on modern PCs, and may be low enough to prevent even automated, targeted attacks which anticipate use of Logcrypt and attempt to prevent incriminating entries from being recorded. For logs which don't record breakin attempts, l primarily determines how quickly an attacker must decide to manipulate an entry once it is received, which will generally be on a long, human timescale, as in the case of a CFO who wishes to modify financial bookkeeping entries.

Chapter 5 — Simple System

Our fundamental system is illustrated in figure 5.1. An initial secret is used to start a hash chain in which each link is used to derive keys for a single log entry by hashing with an additional constant (0 for MAC keys, 1 for encryption keys).

As soon as a key is used, it is erased from memory. Likewise, the link in the hash chain used to create each entry must be erased as soon as it has been used. Consequently, only the bottom link in the hash chain and the keys it generates exist in memory while the logging system awaits a new entry.

Figure 5.3 illustrates how Logcrypt can provide confidentiality as well as integrity. A second key is derived from each link in the hash chain and used to encrypt the entry. In a system without encryption, each entry L_i can briefly be described as follows, where $s_i = h(s_{i-1})$ and s_0 is the initial secret and $|$ denotes concatenation:

$$L_i = \langle MAC(h(0|s_i), log_i), log_i \rangle$$

In a system using encryption, each entry additionally encrypts log_i:

$$L_i = \langle MAC(h(0|s_i), log_i), E(h(1|s_i), log_i) \rangle$$

Figure 5.1: Simple forward security using Message Authentication Codes.

Figure 5.2: Verifying entries in the simple scheme.

Figure 5.3: Forward security plus secrecy.

More formally, the Logcrypt algorithm for MAC-based integrity protection with optional encryption is as follows:

Given

A secure cryptographic hash function $h(input)$

A secure message authentication code $MAC(secret, plaintext)$

A secure encryption function $E(secret, plaintext)$

Begin

Randomly generate or accept as input an initial unique secret s

Store s securely (generally accomplished by sending it to a separate machine)

Loop

Ensure that the next 3 steps completely destroy the prior values:

Calculate the next link: $s = h(s)$

Derive the MAC key: $mkey = h(0|s)$

Derive the encryption key: $ekey = h(1|s)$

Wait for the next log entry: log_n

Let $MAC_n = MAC(mkey, log_n)$

If encrypting,

$ciphertext_n = E(ekey, log_n)$

Output $\langle MAC_n, ciphertext_n \rangle$

Else

Output $\langle MAC_n, log_n \rangle$

Verifying a log later proceeds as follows:

Given

The initial secret s

The decryption function D corresponding to E

If encryption was used, a list of log entries such that $L_i = \langle MAC_i, ciphertext_i \rangle$

Otherwise, $L_i = \langle MAC_i, log_i \rangle$

Begin

Loop for $L_1..L_{|L|}$:

Calculate the next link: $s = h(s)$

Derive the MAC key: $mkey = h(0|s)$

Derive the decryption key: $dkey = h(1|s)$

If encryption was used, set $log_i = D(dkey, ciphertext_i)$

Abort unless $MAC_i == MAC(mkey, log_i)$

(optionally output log_i)

Indicate success.

5.1 Security Proofs

Theorem 5.1.1 *Assume h is a random oracle and MAC is a secure message authentication code. Assume the secrets $s_i, mkey_i$ corresponding to every log entry L_i have been securely deleted at time $t_j, j > i$, and that no adversary has information about the initial secret s. Then with overwhelming probability, no adversary who gains access to the system after t_j can modify any entry L_i without detection.*

Proof: In the random oracle model [2], $h(x)$ provides no information about x. Then knowledge of $s_i, i \geq j$ provides no information about any $s_k, k < j$ since $\forall i > 0, s_i = h(s_{i-1})$. Assuming $MAC()$ is secure, then knowledge of $L_i = \langle MAC_i, log_i \rangle$ where $MAC_i = MAC(mkey_i, log_i)$ provides no information about $mkey_i$. Since the

system consists only of values s_i, $mkey_i$, log_i and MAC_i, with overwhelming probability no valid MAC for $log'_i \neq log_i$ can be created without $mkey_i$ $\square$

Theorem 5.1.2 *Given the assumptions in the previous theorem, and additionally assuming that E is a semantically secure encryption function, then no adversary who gains access to the system after t_j can distinguish any $E(ekey_i, log_i)$ from $E(ekey_i, log'_i)$ where $|log_i| = |log'_i|$ in any log entry $L_i = \langle MAC_i, E(ekey_i, log_i)\rangle, i < j$.*

Proof: If E is semantically secure, then by definition any adversary without knowledge of $ekey_i$ cannot distinguish encryptions of equal length plaintexts or gain information about $ekey_i$ from $E(ekey_i, log_i)$. The previous theorem shows that the adversary cannot learn any $s_i, i < j$, so knowledge of all $s_i, i \geq j$ also provides no information about $ekey_i$ $\square$

Note that these theorems do not address the situation in which adversaries delete or truncate a log, effectively making values of L_i unavailable to the verifier. We address this issue in section 8.

Proofs for the public key variants of Logcrypt are easy to construct for secure signature schemes with protection against existential forgery, and are sufficiently similar to the ones presented here that we omit them.

Chapter 6 — Public Key System

The primary disadvantage of the symmetric system just described is that verification of a MAC requires the same key that was used to create it. This means that anyone with the ability to verify a particular log entry could create arbitrary alternative entries which would also appear correct.

Public key cryptography provides the ability to separate signing from verification and encryption from decryption. This section describes how the signing/verification separation can be used to create logs which can be verified by anyone. We omit discussion of creative applications of the encryption/decryption separation, although several such applications are possible, particularly when using identity based encryption.

Bellare and Miner proposed a public key counterpart to hash chains in [1] which could be used with our simple system. Here we propose a system which is less elegant, but which can be used with any signature scheme. Then we present an optimization which works with any identity-based signature scheme.

Figure 6.1 shows the public key variant of Logcrypt. A signature replaces the MAC, and we add a special log meta-entry listing the next n public keys which will be used for signing. Then the next $n-1$ entries can be described as follows ($i \in (1..n-1)$):

$$L_i = \langle log_i, Sign(private_i, log_i) \rangle$$

The last private key, $private_n$, is used to sign the meta-entry listing the next n

Figure 6.1: Forward security with public verifiability.

Figure 6.2: Verifying entries in the public key scheme.

public keys.

More formally, the Logcrypt algorithm for signature-based integrity protection is as follows:

Given

A public-key signature function $Sign(private, message)$

A value n describing how many public/private keypairs will be stored in memory.

Begin

Generate an initial random public/private keypair, $(pub_0, private_0)$

Store pub_0 securely (generally, by sending it to a separate machine)

Loop

Create random keypairs $\langle (pub_1, private_1)..(pub_n, private_n) \rangle$

Create the meta-entry listing the public keys: $meta = \langle pub_1..pub_n \rangle$

Generate the signature on the meta-entry: $sig_0 = Sign(private_0, meta)$

Securely delete $private_0$. (pub_0 may also be removed).

Output $\langle meta, sig_0 \rangle$

Set $i = 0$

Loop

Increment i

If $i == n$, exit the inner loop

Wait for the next log entry: log_i

Calculate $sig_i = Sign(private_i, log_i)$

Securely delete $private_i$. (pub_i may also be removed).

Output $\langle log_i, sig_i \rangle$

Set $pub_0 = pub_n$

Set $private_0 = private_n$

Recall that the second principle listed as foundational to Logcrypt's security in section 4 is the ability to validate a later log state using the state at an earlier entry. Hash chains actually *derive* later secrets from earlier secrets, even though all we need is validation. Consequently, it suffices to sign the public key that will be used at a later time using an earlier key, then throw away the signing key to fulfill the requirement that secrets be erased after they're used.

Verification proceeds as follows:

Given

The initial public key pub_0

A public-key signature verification function $Verify(pub, sig, message)$ which returns true iff sig is a signature for $message$ made with pub

A list of log entries such that $L_i = \langle log_i, sig_i \rangle$

Begin

Set $i = 0$

Loop

Set $meta = log_i$

Abort unless $Verify(pub_0, sig_0, meta)$ returns true

Extract public keys $pub_1..pub_n$ from $meta$

Set $j = 0$

Loop

Increment i and j

If $j == n$, exit the inner loop

Abort unless $Verify(pub_j, sig_i, log_i)$ returns true

Set $pub_0 = pub_n$

Indicate success.

6.1 Elliptic Curve Cryptography

Elliptic curve cryptography (ECC) is becoming increasingly popular as an alternative to cryptosystems like RSA because its structure allows shorter key lengths to provide an equivalent degree of security. For example, an RSA key of 1620 bits is estimated by [6] to have the same security as an ECC key with only 256 bits, while more recent estimates [12] specify even longer RSA key lengths.

This property can be particularly useful for Logcrypt, since a new key must be generated and stored for each log entry. When log entries are short, this overhead can consume more than 50% of the total space required for the log. Since the specification in the last section makes no distinction between traditional and ECC cryptosystems, ECC-based signature algorithms can be used without modification to the algorithm. However, ECC is commonly used for identity-based encryption, which has further advantages in storage space which we consider in the next section.

6.2 Identity Based Signatures

Identity-based Signatures (IBS) allow public keys to be derived from arbitary bit strings and the public key of a Private Key Generator (PKG). Private keys can only be extracted from that string and the PKG private key. The construction given by Cha and Cheon [4] uses elliptic curves as the underlying mathematical construction for an IBS sceme, allowing Logcrypt to retain the advantage of short ECC keys while eliminating the need to list the individual public keys to be used for upcoming log entries. That is, public keys 1 through n can simply be derived from the strings “1”, “2”, etc., while the corresponding private keys are created with a function called

Extract, cached in memory and deleted after use. A new private key generator (PKG) key is generated for each key block, since it is used to generate all the private keys, and must therefore be erased as soon as all n private keys have been created. The $n-1$ entries corresponding to a PKG key can be described as follows, just as before ($i \in (1..n-1)$):

$$L_i = \langle log_i, Sign(private_i, log_i) \rangle, \ where \ private_i = Extract(PKGprivate, i)$$

The last private key, $private_n$, is then used to sign the meta-entry listing just the next PKG public key.

Formally, here is the Logcrypt algorithm for IBS integrity protection:

Given

An IBS function $IBSign(private, message)$

A private key extraction function $private = Extract(PKGprivate, i)$

Begin

Generate an initial random PKG keypair, $\langle PKGpub, PKGprivate \rangle$

Store $PKGpub$ securely (generally, by sending it to a separate machine)

Loop

Calculate private keys for the strings $1..n$: $private_i = Extract(PKGprivate, i)$

Securely delete $PKGprivate$.

Set $i = 0$

Loop

Increment i

If $i == n$, exit the inner loop

Wait for the next log entry: log_i

Calculate $sig_i = IBSign(private_i, log_i)$

Securely delete $private_i$. (pub_i may also be removed)

Output $\langle log_i, sig_i \rangle$

Generate a new PKG keypair, $\langle PKGpub, PKGprivate \rangle$

Generate the meta-entry listing the new PKG key: $meta = \langle PKGpub \rangle$

Generate the signature on the meta-entry: $sig_n = IBSign(private_n, meta)$

Securely delete $private_n$.

Output $\langle meta, sig_n \rangle$

Since the public keys are always simply the strings 1 through n, they don't need to be stored in the meta entry. Verification proceeds as follows:

Given

The initial PKG public key $PKGpub$

An IBS verification function $IBVerify(PKGpub, ID, sig, message)$ which returns true iff sig is a valid signature for $message$ using the private key derived from $PKGpub$ and the string ID

A list of log entries such that $L_i = \langle log_i, sig_i \rangle$

Begin

Set $i = 0$

Loop

Set $j = 0$

Loop

Increment i and j

If $j == n$, exit the inner loop

Abort unless $Verify(PKGpub, j, sig_i, log_i)$ returns true

Set $meta = log_i$

Abort unless $Verify(PKGpub, n, sig_i, meta)$ returns true

Figure 6.3: Forward security with public verifiability using Identity Based Signatures.

Figure 6.4: Verifying entries in the IBS scheme.

Extract the new $PKGpub$ from $meta$

Indicate success.

Chapter 7 — Cumulative Verification

In the construction given in [10], verification values for log entries validate the current log entry as well as the verification value for the previous entry. This is an advantage in that verifying an entry ensures that all prior entries were correct; the last value in such a log can be sent to an external auditor as a commitment to the entire log up until that point. In fact, the last value is the only value which needs to be stored – rather than storing each entry with its signature or MAC, log entries could be kept in one file and another could store only the most recent verification value.

This feature can trivially be added to the simple Logcrypt algorithm by adding the MAC of the previous entry as a third parameter to the $MAC()$ function, and to the other two algorithms by including the cumulative hash of all prior entries with the current entry for the signing process. However, we chose to omit this feature in our specification because it can hamper forensic analysis in some situations.

Consider an attacker who deletes some number of log entries (not including a public key meta-entry) from the middle of a Logcrypt log which uses public key or identity based signatures and stores the signature for each entry. The verification process will detect that the log has been modified in either case. However, if cumulative hashes are being maintained, intact entries after the deletion cannot be verified since the cumulative hash of prior entries cannot be reconstructed without knowledge of the missing entries. Without cumulative hashing, the later entries can still be checked for validity. Of course, if a meta-entry is lost, then there will be no public key available for verifying the later entries.

Logcrypt logs using MACs are not so heavily impacted by this drawback, and users may find it worthwhile to use cumulative MACs by default. Entries after a

deleted block can still be checked once the number of deleted entries are known as long as the MAC for each entry is kept (rather than storing only the last one), except for the entry immediately after the deleted block. That entry cannot be verified in a system using cumulative MACs since the previous MAC is unknown and thus prevents calculation of the current MAC.

Of course, an attacker aware of how Logcrypt works will tend to remove a Logcrypt log entirely, giving the analyst no information about the log, or leave it unchanged hoping that administrators won't notice any incriminating entries. But especially in situations where verification values are kept separately from a traditional-looking log, attackers may be unaware that Logcrypt is in use, and may remove only a few incriminating entries from the log. In such cases, forensic analysis can benefit from the finer granularity offered by not using cumulative verification.

Chapter 8 — Detecting Truncation

Consider what happens if an attacker chooses to simply delete or truncate a Logcrypt log rather than attempting to modify existing entries without detection. Of course, no new valid entries can be added once a log has been truncated, since intermediate secrets will have been lost, and this will be detected during verification. But in the case of a log which only records breakin attempts, for example, a lack of new entries suggests that the system is still secure.

Logcrypt cannot prevent an attacker in control of a system from deleting and truncating files. But it can be used to let an administrator know when the log is no longer functioning normally by using what we call metronome entries. Metronome entries are simply special log entries which are made at regular intervals to indicate that the log is still accepting new data. If an attacker truncates the log just before an incriminating entry was made, he also truncates any metronome entries entered after that entry, and must prevent any future entries from being recorded, since they will fail verification. The verification process can be augmented to ensure that all metronome entries are present at the time of verification. If any are missing, then the last valid entry indicates the earliest time at which the log could have been truncated.

A related idea was described by Schneier and Kelsey[10]. They describe how a log can be securely closed by creating a special final entry and storing the final MAC off-site. Such a technique would be easy to use with Logcrypt in situations where this is needed.

Chapter 9 — Multiple Logs

Perhaps the most inconvenient part of Logcrypt is the requirement that initial values be securely stored at log creation time. Most systems maintain logs for multiple services concurrently, and regularly prune out older entries by rotating log files. Unless initial values can be automatically, securely shipped to an external machine via a network, this quickly becomes an unwieldy task for system administrators.

To simplify this key distribution issue, we create a treelike structure of logs in which parent nodes store the initial secrets for their children. New children can be added at any time, and only the initial value created for the root node needs to be kept securely off the machine.

Figure 9.1 shows a simple case of a single encrypted master log which maintains the secrets for other system logs. Since the first child uses MACs instead of signatures and therefore has a secret initial value, the parent must encrypt all its entries. However, if all the children of a node use public key or identity based signatures, all the initial values will be public keys and need not be encrypted. Such a subtree can then be verified by anyone who can verify the integrity of the initial value of the root node.

Storing multiple logs in a tree structure has other advantages as well. If all system logs were merged into a single log instead of being kept separately and maintained in a tree, then the entire log will need to be traversed in order to check the validity of the last entry. With a tree structure, however, any individual log can be verified by starting at the root node and iteratively verifying the entries corresponding to the nodes which lead to the log in question.

Figure 9.1: Maintaing concurrent logs with a single initial value.

Chapter 10 — High Volume Logging

One way an attacker might try to compromise Logcrypt is to generate a large number of spurious events to be logged by the system or otherwise bog down the system's logging facility. If the attacker can generate the events more quickly than the system can process them, the system could end up with a backlog of entries all vulnerable to attack since they haven't yet been signed.

Recall that Logcrypt only offers strong protection for events which occur before time $t - l$, where t is the time of attack and l is the time required for Logcrypt to use and then destroy the secret corresponding to a particular entry. Increasing the demands on the system allows the attacker to increase the overhead l, which defines the window in which he can compromise entries which have already been received. To some extent, this cannot be avoided, particularly if we assume an attacker that can thoroughly overwhelm the system.

However, we can improve the system's resilience to such attacks as well as improve its overall capacity tremendously by observing that some of the cryptographic operations which Logcrypt uses are much more efficient than others. In particular, public key and identity based signature operations have a relatively high overhead that varies very little with the size of the log entry being signed. This is because signatures are performed on a constant-sized hash of the input, so that the only extra cost to lengthening the input comes from the hash function. For example, a 1.4Ghz Pentium workstation running OpenSSL requires 56 ms to perform an RSA signature using a 2048-bit key, whereas it can process in excess of 100 bytes of input to the SHA-1 hash per *microsecond*.

Consequently, if multiple new log entries arrive while the present entry is being

processed, it makes sense to create a single signature for all of them combined rather than creating one signature for each entry, decreasing the average l across all the entries. Of course, this requires changes to the output format so that the entries can still be identified as distinct, and the verification process will have to consider the entries as a single unit.

This creates new possibilities for attackers which have the ability to overwhelm even the hash function, since they will be able to create increasing numbers of entries which the system will try to process before creating any signatures at all, whereas a system which processes entries one at a time would still create signatures on individual entries even as the queue filled up. Consequently, an upper limit could be set on the number of entries which may be aggregated into a single signature, providing an upper limit on latency for the head entry in the queue while still allowing much higher performance in high-load situations. However, in many systems the hash function will have higher throughput than the network and disk subsystems, so that external limitations will be reached before hash function performance even becomes an issue.

Also note that in the public key system, computing public/private keypairs can be CPU-intensive. Consequently, systems which have plenty of memory and regular intervals with low system load may find it beneficial to compute keypairs during these available intervals, storing them until needed. This is similar to increasing the parameter n which determines how many keys are listed per meta-entry, and comes at no security penalty as long as keys are still held securely until used and then immediately destroyed.

Chapter 11 — Implementation Considerations

Gutmann describes in [9] how data written to magnetic storage or even just stored in RAM may later be recoverable by a determined adversary. Some of the techniques he describes involve exotic equipment and processes destructive to hardware, but others can be performed entirely in software, such as scanning swap space for memory pages containing cryptographic keys. Fortunately, Gutmann also describes techniques for mitigating at least the software-based attacks, such as locking pages in memory and regularly inverting bits of stored keys.

Logcrypt implementors should take the recommendations in [9] into account along with the threat model of the target system in order to ensure that deleted secrets are truly unavailable to attackers. The implementation we provide uses the *mlock*() system call to ensure that memory pages containing secret values are not paged to disk, and the *volatile* keyword in C to prevent the compiler from using optimizations which might make unexpected copies or relocations of secret values.

In particular, systems which precompute keypairs as described in the last section must be sensitive to the issues raised by Gutmann, since such systems will need to store relatively large amounts of data for longer periods than other Logcrypt implementations.

Chapter 12 — Conclusion and Future Work

In this paper, we showed several innovative ways of achieving forward security for logs. We showed how to allow forward secrecy as well as tamper evidence in the simple system, as has been done in previous work, and then added a public key variant allowing verifiability without the ability to forge entries. We showed how multiple logs can be maintained concurrently and verified using a single initial value. We suggested optimizations which resist flooding attacks and dramatically improve performance under high load conditions. We described how logs can be made resistant to truncation, and closed to further additions. These features all address significant needs in systems administration as well as other disciplines such as finance and medicine which deal with tamper-sensitive data.

Future work may address further improvements to the public key variant of Logcrypt. Hierarchical IBS systems and meta-entries storing multiple PKG public keys can be used to further improve performance while keeping overhead low. Bellare and Miner's contribution [1] also provides an obvious avenue for space-efficient public verification.

An initial implementation of our simple and public key systems is available under the GPL at $http://isrl.cs.byu.edu/logcrypt/$. It implements our simple system as well as the public key variant using RSA signatures. Future work will include performance refinements and increased convenience features in the form of both library functions and sample applications.

Bibliography

[1] M. Bellare, S. Miner. A Forward-Secure Digital Signature Scheme. In Proc. of Crypto, pp. 431–448, 1999. http://citeseer.ist.psu.edu/bellare99forwardsecure.html

[2] M. Bellare and P. Rogaway, Random Oracles are Practical: A Paradigm for Designing Efficient Protocols, ACM Conference on Computer and Communications Security 1993, pp62-73.

[3] M. Bellare and B. Yee. Forward Integrity for Secure Audit Logs. Technical Report, Computer Science and Engineering Department, University of California at San Diego, November 1997.

[4] J. Cha, J. Cheon. An ID-based signature from Gap-Diffie-Hellman Groups. Proceedings of PKC 2003, Lecture Notes in Computer Science, Vol. 2567, pp. 18-30 (2003). http://www.math.snu.ac.kr/~jhcheon/Published/2003_PKC/PKC03_CC.pdf

[5] C. N. Chong, Z. Peng, and P. H. Hartel. Secure audit logging with tamper resistant hardware. Technical report TR-CTIT-02-29, Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands, Aug 2002. http://citeseer.nj.nec.com/chong02secure.html

[6] A Cost-Based Security Analysis of Symmetric and Asymmetric Key Lengths, RSA Laboratories Bulletin #13, April 2000. http://www.rsasecurity.com/rsalabs/node.asp?id=2088

[7] A. Futoransky and E. Kargieman. PEO Revised. DISC 98 (Dıá Intrenacional de la Seguridad en Cómputo). DF, Mexico. 1998. http://www1.corest.com/files/files/11/PEO.pdf

[8] A. Futoransky and E. Kargieman. VCR y PEO, dos protocolos criptográficos simples. 25 Jornadas Argentinas de Informática e Investigación Operativa, July 1995. http://www1.corest.com/common/showdoc.php?idx=86&idxseccion=11

[9] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. Sixth USENIX Security Symposium, July 1996. http://www.cs.auckland.ac.nz/p̃gut001/pubs/secure_del.html

[10] J. Kelsey, B. Schneier. Minimizing Bandwidth for Remote Access to Cryptographically Protected Audit Logs. Recent Advances in Intrusion Detection, 1999. http://citeseer.nj.nec.com/kelsey99minimizing.html

[11] MSyslog (Unix syslogd with integrity protection). http://oss.coresecurity.com/projects/msyslog.html

[12] H. Orman and P. Hoffman, Determining Strengths For Public Keys Used For Exchanging Symmetric Keys, Internet Engineering Task Force RFC 3766, April 2004. http://www.ietf.org/rfc/rfc3766.txt

[13] B. Schneier, J. Kelsey. Cryptographic Support for Secure Logs on Untrusted Machines. In Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, Jan. 1998.

[14] B. Schneier, J. Kelsey. Secure Audit Logs to Support Computer Forensics. ACM Transactions on Information and System Security 2(2): 159-176, 1999.

[15] K. Thompson. Reflections on Trusting Trust. Communications of the ACM, Vol. 27, No. 8, August 1984, pp. 761-763. http://www.acm.org/classics/sep95/

[16] B. R. Waters, D. Balfanz, G. Durfee, D. K. Smetters. Building an Encrypted and Searchable Audit Log. ACM Annual Symposium on Network and Distributed System Security, 2004.