Internet Engineering Task Force                                   Syslog
Internet Draft                                               John Kelsey
draft-ietf-syslog-auth-00.txt                       Expires:  June 2001

# Syslog-Auth Protocol

ABSTRACT

This document describes syslog-auth, an attempt to add origin authentication, message integrity, replay-resistance, message sequencing, and missing message detection to syslog, while continuing to allow the use of UDP for syslog messages. Syslog-auth supports many different setups for syslog devices, relays, and collectors. Syslog-auth should be implemented on all devices, relays, and collectors being used together.

## 1. Introduction

The current syslog protocol sends syslog messages over various unsecured networks, with no mechanisms for ensuring anything about the messages that eventually arrive and are stored. It also provides no mechanisms for storage security.

In this note, I describe syslog-auth, a lightweight mechanism for providing authentication of syslog messages sent over an insecure network, between a sender and receiver which share a secret key. The goals of this mechanism are:

a. To provide the best over-the-wire security possible between pairs of syslog-auth machines that share a key. This means allowing the receiver to detect alterations, deletions, insertions, and replays in the sequence of syslog-auth messages from a sender in real time if resources permit, and during offline analysis if resources are too tight on the receiver.

b. To build in mechanisms to provide some level of storage security for logs generated by syslog-auth machines, when they are stored anywhere but on the machine that generated them.

c. To build in mechanisms to allow a ``best try'' of providing storage security on syslog-syslog receivers. (That is, on old-style receivers doing nothing but plain old syslog.)

d. To NEVER cause a syslog-syslog receiver to crash or drop messages because they're too long, and to NEVER leave syslog-auth data at the end of a message that can't be unambiguously identified as either syslog-auth data or original message text data.

e. To combine with reliable delivery mechanisms (e.g., syslog-reliable in RAW mode) to provide even stronger guarantees for both over-the-wire and storage security.

## 1.1. Resource Requirements

Our resource assumptions are as follows:

a. The sender's machine may be very limited in its resources. It must have a small amount of persistent storage to keep its shared key with the receiver, and a small amount of RAM which is kept for reasonable lengths of time. It must also be capable of computing a cryptographic hash function over the data it sends out as syslog messages. Senders that lose all their RAM state (e.g., reboot) once per message and have no clock, persistent storage, or reboot counter are not likely to benefit much from this mechanism.

b. The receiver's machine is assumed to be much less limited. In general, its problem will be assumed to be keeping up with many senders at once.

c. We assume that we have a shared secret key between the sender and receiver. How the key got shared is beyond the scope of this note, but the key did, in fact, get shared. This key includes both raw key data, and some indication of what cryptographic authentication scheme is to be used with that key.

d. In this note, I assume no cryptographic mechanisms available except md5.

## 2. Syslog-Auth Message Format

The syslog-auth message format consists of the text of the original syslog message (called the message text from now on), concatenated with an authentication block. This block contains all the information necessary to verify the origin, integrity, and freshness of this message. It may also contain additional useful information.

## 2.1. General Formatting Issues

A syslog-auth message is intended to go from a syslog-auth enabled sender to a syslog-auth enabled receiver.  A receiver that doesn't understand syslog-auth will have no way to use the authentication block, and for very long message texts, won't even see the authentication block.  It may be that a sender sends out all messages using syslog-auth.  In this case, syslog-auth receivers that share a key with it can make use of the authentication blocks, and old-style syslog receivers will have no harm done by the messages, other than receiving longer messages.

All information in syslog-auth is kept in ASCII printable format, and all cryptographic information appears at the end of the syslog message, after the original message text.  Cryptographic information is always contained in an "authentication block."  A syslog-auth message sent to a syslog-auth receiver may have either one or two authentication blocks.  A syslog-auth message sent to an old style receiver may have zero, one, or two authentication blocks.  Note that whenever a message is truncated for sending to an old-style receiver, each authentication block is either left intact or entirely deleted.

When a new syslog-auth message is generated and sent to a syslog-auth receiver, it will have one authentication block.  If the message is forwarded to another syslog-auth receiver, it will end up with both the original authentication block and a block by the forwarder. However many times the message is forwarded, it always keeps its original authentication block and an authentication block from the most recent forwarding machine.  As will be discussed below, when a machine forwards a syslog-auth message, it strips off the previous forwarder's block, generates a new authentication block, and appends its new authentication block.  The forwarder often borrows some fields from the previous forwarder's block to build its own.

The format of a syslog-auth message is thus

```
  syslog-auth message = message-text + authentication blocks
```

## 2.2. Base 64 Encoding

In the following description, we assume base-64 encoding for various fields.  This is to be done according to RFC-2045.

## 2.3. The Authentication Block Format

The authentication block always appears at the end of the syslog-auth message.  Its fields always appear in the following order, including optional fields that may not appear at all in most messages. I've listed the range of lengths of these fields using base 64 encoding.

a. Transmission MAC Block (Required; 27 bytes)
b. Storage MAC Block (Optional; 27 bytes)
c. Forwarding Block (Optional; 9 bytes)
d. Destination Message Counter (Optional; 8 bytes)
e. Global Message Counter (Required; 8 bytes)
f. Reboot Session ID (Required; 8 or 16 bytes)
g. Flags (Required; 2 bytes)
h. Version (Required; 2 bytes)
i. Cookie (Required; 8 bytes)

Note that the block is intended to be parsed from last field backwards.  Also note that the maximum possible length for a single authentication block is 123 bytes.

### 2.2.1. Transmission MAC Block (27 bytes)

The transmission MAC block consists of two parts: the key ID and the MAC.  All syslog-auth key IDs are 96 bits wide (encoded as 16 characters in base 64), and are computed by the formula:

```
Key ID = low96(hmac_md5_{K}("KEYID"))
```

All syslog-auth MACs are 64 bits wide (encoded as 11 characters) and are computed by the formula:

```
MAC_{K}(X) = low64(hmac-md5_{K}(X))
```

In the transmission MAC block, the key ID used is (naturally) that of the key used to generate the MAC.  The MAC is computed over the whole message text concatenated with the whole authentication block, exactly as it will appear in the final message except with the transmission MAC field set to eleven characters of ASCII zeros.

The purpose of the transmission MAC block is to:

a. Identify which key is being used for the receiver.

b. Authenticate the contents of the message, as well as its freshness and its position in the sequence of messages being sent by this sender.

### 2.2.2. Storage MAC Block (27 bytes)

The storage MAC block consists of the storage MAC key ID and the storage MAC.  The key ID used is (naturally) that of the key used for storage security.  The MAC is always computed over the original message text concatenated with the first authentication block held by the message, with all MAC fields set to eleven characters of ASCII zeros. Note that this happens even if the storage MAC is being computed by a forwarding machine several hops after the message was generated; the MAC is only computed over that original message and first authentication block, with all its MAC fields set to blocks of ASCII zeros.

The purpose of the storage MAC block is to allow the use of a second key for storage security. This is necessary since a syslog-auth receiver with the key to a MAC stored can't use that key to provide any additional storage security.

### 2.2.3. Forwarding Block (25 bytes)

This block is used only when the message is being forwarded, and the sender wishes to inform the receiver of this fact. It has three components:

a. Flags (1 byte, encoding 6 bits) numbered 5-0:

   (i) Secure Path bit (bit 5) -- set if message has traveled over syslog-auth for its whole life.

   (ii) Replay Resistant bit (bit 4) -- set if every forwarder that has forwarded this message is sure this is not a replayed message. (See the section on forwarding issues to see why this is important.)

   (iii) Reserved (bits 2-0) -- reserved for later use, MUST be zeros now.

b. The IP address of the first syslog-auth sender or forwarder, 128 bits, base-64 encoded as 22 characters.

c. The number of times this message has been forwarded, 0-4095, encoded as two base-64 characters.

The forwarding block is only used when a message is being forwarded, and it can only appear when it is indicated by the right flag.

### 2.2.4. Destination Message Counter (8 bytes, base 64 encoded)

This is a counter incremented only for messages the sender knows are going to the same destination. It is 48 bits wide in its raw format.

### 2.2.5. Global Message Counter (8 bytes, base 64 encoded)

This is a counter which is incremented and included for each syslog-auth message sent out since its last reboot.  It is 48 bits wide in its raw format.

### 2.2.6. Reboot Session ID (8-16 bytes, base 64 encoded)

The reboot session ID can have one of three sources:

a.  It can be a fixed value that never changes, though this typically allows no replay resistance.

b.  It can be generated pseudorandomly, but this means that it needs to be 96 bits long.

c.  It can be based on something guaranteed always to increase in value, such as the timestamp of the last reboot.  It's very important to note that this allows a receiver to immediately detect any replay attempt.  This value is always 48 bits long. Its only requirement is that a later reboot session ID MUST always be greater than an earlier reboot session ID.

### 2.2.7. Flags (2 bytes, base 64 encoded)

The flags take up 12 bits in their raw format, and are represented by two base-64 characters.  Numbering from 11-0, these are:

a.  Destination Counter (bit 11):
    This indicates whether the optional destination message counter is included in this message.  A zero bit means there's no desitination message counter.

b.  Superincreasing Session ID (bit 10):
    This indicates whether the session ID is guaranteed to always increase over time (so that replayed session IDs can be recognized by the fact that they have lower values than recently seen session IDs), or whether the session ID is pseudorandom.  A one bit means the session ID is superincreasing.

d.  Temporary Reboot Session ID (bit 9):
    This indicates whether this session ID is a temporary one.  This is used when a sender has to wait for its PRNG to become available, or to accumulate the running hash of some limited number of messages, before it can generate a reboot session ID that it overwhelmingly likely to be unique.  When this bit is set to one, the reboot session ID MUST be 0.

e.  Replay Vulnerable (bit 8):
    This bit indicates whether the replay session ID may be repeated in different sessions.  If this bit is set, then the message has no replay protection right now.  This may be a temporary situation (e.g., when we're using a temporary reboot session ID) or permanent (e.g., when the sender can't generate any kind of

unique session ID).

f. Forwarding Block (bit 7):
   This bit indicates whether the forwarding block is present in this message.

g. Storage MAC (bit 6):
This bit indicates whether the storage MAC is present in this message.

h. Old Style Receiver (bit 5):
This bit is set when the sender believes it is sending to an old-style receiver.

i. Reserved (bits 4-0):
These bits must be zero at present, and may be used in the future to indicate new things.

#### 2.2.8. Version (2 bytes, base 64 encoded)

The version is twelve bits raw. I don't expect the version to *ever* change, but if it does, then all the other fields may change. The current version is 1. A receiver at version M MAY be able to understand lower versions, but this is not required, since lower versions may be known to be insecure. A receiver below the version of the sender cannot make any use of the syslog-auth authentication block at all.

#### 2.2.9. Cookie (8 bytes)

The cookie is an eight character string that occurs at the *end* of the syslog-auth message, signaling that it is a syslog-auth message, rather than an old-style syslog message. There's no special reason to use "authAUTH", and the choice has no security relevance--its purpose is simply to give receivers who must process both old style and syslog-auth messages a simple way to distinguish them, with a very low rate of false identification. Note that falsely identifying an old style message as a syslog-auth message will never cause the receiver to misidentify the message for long, since there won't be a valid MAC for this message.

#### 2.2.10. Length of Messages

The longest possible syslog-auth message (including forwarding, as described in detail below) will include a 1024-byte message text, and two 109-byte authentication blocks, and thus a total length of 1133 bytes. A normal syslog-auth message should have about 57 bytes of authentication block when there is no forwarding going on, and 123 bytes with forwarding.

## 3. Key Management Issues

In this document, managing the keys outside the senders, forwarders, and receivers is the responsibility of the user of the system. However, in this section, we discuss:

a. Our general philosophy of how keys ought to be managed for this system.

b. Some ideas of ways that keys may be managed that will scale reasonably well.

c. Some tools for managing keys on syslog-auth devices.

## 3.1. The Basic Scheme: One Key Per Sender/Receiver Pair.

This is the cleanest way to manage the keys. Somehow, we distribute a shared key to each sender and receiver, in such a way that each receiver has a different key for each sender it's receiving from, and each sender has a different key for each receiver it's sending to.

This has good security properties, since compromise of one sender leads only to the compromise of logs sent by that sender, and compromise of one receiver leads only to the compromise of logs received by that sender. However, we now have the problem of distributing all those keys securely.

Consider a set of syslog-auth machines, which are all "owned" by the same entity and communicate with one another. Let's call the entity that owns all these machines the administrator. Also, let's assume that each machine has an IP address, and knows the IP address of all machines it communicates with over syslog-auth. We now have the following trick for simplifying key management:

a. The administrator has a master key, K_{master}, which is stored securely somehow. This key could be a passphrase, though it should be a very long one. The key is an hmac-md5 key.

b. Each syslog-auth machine has its own device key, K_{device}. This key has to be given to the device by the administrator. The administrator generates a device's key as:

```
K_{device} = hmac-md5_{K_{master}}(Device's IP address).
```

c. Each machine knows the IP addresses of machines it is going to be sending to. The keys of these machines are generated as:

```
K_{sender,receiver} = hmac-md5_{K_{sender}}(receiver's IP address)
```

where K_{sender} is K_{device} for the sender.

d. Storage MAC keys are derived as

```
K_{sender,STORAGE} = hmac-md5_{K_{sender}}("STORAGE")
```

e. This reduces the key management problems to:

(i)  Loading the secret key for a device onto that
device securely during installation or change of IP
address.

(ii) Loading K_{sender,receiver} onto a receiver
securely.

Note that the administrator can always generate any key in the system in this scheme.  This means that if the administrator's key is compromised, the whole set of keys have to be changed, and until they are, all syslog-auth security is lost.  Similarly, if the administrator's key is lost, the whole system must be rekeyed.

If would also be possible to use this kind of a scheme in a network where some devices have variable IP addresses.  In this case, we'd give each device a unique serial number, and use the serial number in place of the IP address.  So long as "STORAGE" remains an invalid serial number, the scheme will work fine.

Along with making the key management simpler, this scheme decreases the memory needed for a single sender who must send to many receivers.  The sender can compute the key for each receiver on the fly based on their IP address.

## 3.2. Some Less Useful Key Management Options

There are several alternatives for key managemtent in syslog-auth. Here, I describe three methods that may be useful in some limited environments, but which have major security problems.

### 3.2.1. One Key per Sender

A simpler way to manage keys is simply to give each sender a unique key, and share that key with all receivers that must receive messages from the sender.  This is slightly simpler to manage than the scheme described above, but it has a major security problem:  If one machine that receives messages from a given sender is compromised, the attacker can alter or make up new messages from that sender to all its other receivers, without being caught by syslog-auth.

In this case, each sender that also includes a storage MAC in its messages MUST have a unique storage MAC key, which is not shared by anyone else on the network.

### 3.2.2. One Key per Receiver

Another simple way to manage keys is simply to give each receiver a unique key, and share that key with all senders who will send to a given receiver.  This has the same kind of security problem as the above scheme--compromise of one machine sending to a given receiver lets the attacker successfully impersonate all other senders to this receiver.  Even worse, if the sender is sending messages to many receivers, then an attacker who compromises that sender will be able

to generate fake messages or alter messages in transit from any
sender to any of those receivers.

In this case, each sender that does storage MACing MUST have its own
unique storage MAC key.

### 3.2.3. Global Shared Key

The simplest method of doing key management is to give all syslog-auth devices under the same administrator the same key.  This has the advantage that key management is made very simple, and the disadvantage that compromising a single machine with syslog-auth installed allows an attacker to make up or alter messages to and from all other syslog-auth machines using that key.

In this case, each sender that does storage MACing MUST have its own unique storage MAC key.

## 4. Sending Syslog-Auth Messages

### 4.1. Overview

In this section, we discuss the operations of a syslog-auth sender. We expect the majority of machines implementing syslog-auth to be doing nothing but sending messages out.  The requirements for the process of sending messages:

a.  It must be possible to send messages out properly based only on information the sender has available.

b.  It must be possible to send messages out with minimal available resources on the sender.

c.  It must be reasonably computationally cheap to generate and send syslog-auth messages.

#### 4.1.1. Resources

The sender MUST have the following resources:

a.  Shared keys with all its intended syslog-auth receivers.

b.  RAM in which to store a reboot session ID and a counter.

c.  The ability to compute MACs and hashes.

d.  Some way to generate reboot session IDs, which may include:
     (i)  Pseudorandom generation
     (ii) Superincreasing generation
     (iii)Fixed session ID

## 4.2. Setup and Configuration

The sender is told somehow where to send each kind of log message, and is somehow given shared keys to use with any available syslog-auth recipients in its list of machines to send messages to. The mechanisms to do this are generally outside the scope of this note.
In the remainder of this section, we will assume the following:

a. Any storage MAC keys that are to be used have been loaded onto the sender somehow.

b. The sender knows where to send each kind of syslog message, and has shared keys for any of the designated receivers that are equipped to handle syslog-auth messages.

## 4.3. Reboot Sessions

Occasionally, the operations of the sender are disrupted somehow, and it loses the memory of its message counters. We call this event a reboot, though it may or may not correspond to the physical operation of a reboot. Some senders may *never* have a reboot of this kind, while others may reboot several times a day.

A reboot session ID must be generated in one of the following ways, with the specified implications for the message flags.

a. Superincreasing -- session IDs of 48 bits chosen in a way that guarantees that the session ID will never repeat or decrease in successive reboot sessions.

b. Pseudorandom -- session IDs of 96 bits chosen pseudorandomly, and very unlikely to ever repeat in the lifetime of a sender and his key. (If the sender has 2^{32} reboots in its lifetime, the probability of a single collision is about 2^{-33}. If the sender reboots ten times a day for a lifetime of about eighteen years, it will reach 2^{16} reboots in its lifetime, and thus will have a probability of about 2^{-65} of colliding.)

c. Fixed -- session IDs that are 96 bits of binary zeros. This kind of session ID is chosen when a pseudorandom session ID hasn't yet been generated, and may be used for some senders that can't generate any variable session ID.

### 4.3.1. Superincreasing Session IDs

The best way to generate reboot session IDs is to guarantee that if we have two IDs, X and Y, and X was generated before Y, then X < Y. This allows the receiver to put reboot session IDs in sequence, and thus to very efficiently determine whether a message with a new session ID is a replay attempt.

Many devices have an internal clock. Others have some persistent read/write storage in which to store a persistent counter. Still others have a reboot counter available. Any of these can be used to generate superincreasing reboot session IDs. Syslog-auth has no requirement for how these resources (or others) are used to set the reboot session ID. The only requirements are:

a. The Superincreasing Session ID flag MUST be set, and the Replay Vulnerable flag MUST be cleared.

b. The session ID MUST be 48 bits.

c. If there are two session IDs generated at different times, the one generated later MUST be greater than the one generated earlier, when the two session IDs are treated as 48 bit unsigned integers.

I don't know whether there will be any devices that use this, but it's legitimate for a system which never "reboots" (that is, never loses track of its message counters) to make use of a permanent fixed session ID, and still set the Superincreasing Session ID flag. This represents a situation in which we have replay resistance despite a fixed reboot session ID. In this case, the reboot session ID can be any fixed value except a block of binary zeros. The messages sent will always satisfy the requirement that no later message will have a smaller reboot session ID.

### 4.3.2. Pseudorandom Session IDs

Senders that do not have access to a clock, persistent storage across reboots, or a reboot counter may have to generate reboot session IDs pseudorandomly. It is legitimate to do this by generating a cryptographically strong 128 bit random number, but I don't expect there to be any sender devices in practice that don't have any of those other things, but which do have a really good RNG or PRNG.

I emphasize: C compiler PRNGs MUST NOT be used to generate these reboot session IDs.

The standard way I expect for senders to generate reboot session IDs in this case is as follows:

a. Initialize the reboot session ID to a fixed 128-bit block of binary zeros.

b. Set the Temporary Reboot Session bit and the Replay Vulnerable bit and clear the Superincreasing Session ID bit.

c. Decide on a number of messages, N, after which it is overwhelmingly likely that we will have a unique message sequence --that is, that this precise sequence of messages has never been generated from this sender before, and never will again.

d. Initialize an md5 hashing context.

e. For each message sent, feed the message text into the md5 hashing context. If there is an internal clock (e.g., clock ticks since reboot) available on the sender, hash that value in as well.

f. After the sender has sent its Nth message, it does the following:

(i) Hashes in any additional information that might be unique for this reboot session, such as an internal clock, status flags, etc.

(ii) Computes the final hash value from all this information that's been fed into that md5 hashing context.

(iii)Sets the reboot session ID to that md5 output value.

(iv) Clears the Temporary Reboot Session flag and the Replay Vulnerable flag.

(v) Leaves the global message counter and destination counters to increment just as they would for any other message.

We don't specify N in this document. However, N MUST be chosen in such a way that the reboot session ID is different for every reboot session.

### 4.3.3. Temporary Session IDs

This is simple enough that there is little to discuss about it. The session ID is set to 96 bits of binary zeros, the Replay Vulnerable flag is set, the Temporary Session ID is set, and the Superincreasing Session ID flag is cleared.

## 4.4. Building a Syslog-Auth Message

The sender builds a Syslog-Auth message one field at a time, in order, as follows:

### 4.4.1. Cookie (Required; 8 bytes)

The cookie value is set to ``authAUTH'' for all authentication blocks.

### 4.4.2. Version (Required; 2 bytes)

The version field is filled in. Note that this field is base-64 encoded, as described above, and so represents a number between 0 and

4095. The current version is version 1.

### 4.4.3. Flags (Required; 2 bytes)

The flags are set as follows:

a.  Destination Counter (bit 11):
    If this sender is including the optional Destination Counter
    field, then this bit is set, otherwise it is cleared.

b.  Superincreasing Session ID (bit 10)

c.  Temporary Reboot Session ID (bit 9)

d.  Replay Vulnerable (bit 8):
    These bits are set as indicated earlier in this section, in the
    discussion of different ways to derive reboot session IDs.

e.  Forwarding Block (bit 7):
    For messages being sent (not forwarded), this bit will always be
    cleared.  In the section on forwarding, below, we will discuss
    cases where this bit is set.

f.  Storage MAC (bit 6):
    This bit is set if the sender is including the optional storage
    MAC block in this message.

g.  Old Style Receiver (bit 5):
    This bit is set if the sender believes it is sending to an
    old-style receiver.

h.  Reserved (bits 4-0):
    These bits are all set to zero at present.

### 4.4.4. Reboot Session ID (Required; 8 or 16 bytes)

This value is set as described above.  The raw value of the session
ID is either 48 or 96 bits; that value is then base 64 encoded.

### 4.4.5. Global Message Counter (Required; 8 bytes)

This value is set to the number of all messages that have been sent
so far in this reboot session by this sender.  Note that the current
reboot session and global message counter on a sender are used for
all receivers.  It is a 48-bit number that is base-64 encoded as 8
bytes.

### 4.4.6. Destination Message Counter (Optional; 8 bytes)

If this value is included, it is set to the number of messages in this reboot session sent to this receiver so far.  It is a 48-bit number that is base-64 encoded into 8 bytes.

### 4.4.7. Forwarding Block (Optional; 9 bytes)

This field is never included in a message that's just being generated.  The format and settings in this field are discussed in the section on forwarding, below.

### 4.4.8. Storage MAC Block (Optional; 27 bytes)

If this field is included, it will carry two subfields:

a.  A 96-bit key ID, base-64 encoded as 16 bytes.

b.  A 64-bit MAC, base-64 encoded as 11 bytes.

The key ID is a hash of the secret storage MAC key used by this sender.  It is computed by taking

```
  Key ID = low 96 bits( hmac_md5_{K}("KEYID") ).
```

The MAC is computed as:

```
  MAC = hmac-md5_{Key}( message-text, A)  where
```

A = the authentication block, with all MAC values set to ASCII "00000000000", but key IDs and all other stuff left unchanged.  Note that in forwarded messages, storage MACs are computed using the *original* authentication block.  In forwarded non-authenticated messages, no authentication block is included in the MAC.

### 4.4.9. Transmission MAC Block (Required; 27 bytes)

This field also carries two subfields:

a.  A 96-bit key ID, base-64 encoded as 16 bytes.

b.  A 64-bit MAC, base-64 encoded as 11 bytes.

The key ID is a hash of the MAC key used by this sender when sending to this receiver.  It is computed by taking

```
  Key ID = low 96 bits( hmac_md5_{K}("KEYID") ).
```

The MAC is computed as:

```
  MAC = hmac-md5_{Key}( message-text, A')  where
```

A' = the authentication block, with the transmission MAC value set to ASCII "00000000000", but key IDs and all other stuff left unchanged.

Note that the storage MAC value will be authenticated by this MAC.

## 4.5. Sending to Old-Style Receivers

If possible, a sender will know whether it shares a key with a given receiver. If not, it MUST assume that this receiver is an old-style receiver. When a syslog-auth sender is sending to an old-style receiver, it MUST make sure all messages sent are less than or equal to 1024 characters in length.

There are only two ways a message may be shortened before being sent off:

a. If it has two authentication blocks (e.g., if it's a forwarded message), we can cut off the last one and try to fit the resulting message.

b. If that doesn't cut the message down to 1024 or fewer bytes total, we cut off all authentication blocks. This will cause some messages to arrive at their final destinations with no authentication messages, when the last one or more receivers of a message like this are old-style receivers.

Note that in no case do we ever leave part of an authentication block. Either the whole authentication block is left, or the whole authentication block is deleted. (The alternative would be to leave random meaningless blobs of bits at the end of long syslog messages, which could not be distinguished from the original message text except by context.)

## 5. Forwarding

Syslog-auth has a mechanism to make it clear when we're dealing with forwarded messages, and what we can guarantee about them cryptographically. These lead to a fairly simple set of mechanisms for handling forwarded messages, which retain important context about these messages.

Whenever a syslog-auth message is forwarded, the forwarder appends its own authentication block to the message. To avoid having message keep expanding as they're forwarded, the original sender's authentication block is left untouched, but when an already forwarded message is forwarded again, the previous forwarder's authentication block is stripped off, and replaced with the current forwarder's authentication block.

The goal here is to allow future receivers of the message to know exactly what can and cannot be promised about the message. This means:

a. Whether the message has been sent and forwarded only through syslog-auth machines, or whether some of the hops the message has traveled have been unauthenticated.

b. Whether all the forwarders have been able to implement online replay detection on this message.

c. The entry point of the message into the chain of one or more syslog-auth forwarders.

d. How many hops the message has made so far.

All of this is included in the forwarding block, which is contained inside the forwarder's authentication block. A forwarding block MUST be included in the forwarder's authentication block, and MUST NOT be included in the message's original authentication block.

Note that when a message is forwarded to an old-style receiver, all its authentication information is treated as just part of the message text. Similarly, when a message is received from an old-style sender or forwarder, the receiver treats the whole message text like nothing but message text; it does not treat it as being a syslog-auth message, even if that message text contains an authentication block! This is necessary, since the receiver will generally have no shared key with the original sender, and so can't trust anything it would find in a previous sender's authentication block.

This is the only way a syslog-auth message could end up with more than two authentication blocks. Consider the really weird case where a message goes from a syslog-auth sender to a syslog-auth forwarder to an old-style forwarder, and then to another syslog-auth forwarder. Assuming the original message text is not too long, the last forwarder will receive an unauthenticated message whose text contains two authentication blocks. However, the whole message text will be treated as unauthenticated, and a new authentication block will be appended at the end.

It's possible to contrive cases where a very short syslog message is forwarded through alternating syslog-auth and old-style forwarders, and thus ends up with eleven authentication blocks. However, note that this leaves no ambiguity; each authentication block will clearly specify where it came from, and since long messages are never sent to old-style receivers, we will never allow messages to get longer than our maximum syslog-auth length.

## 5.1. Building the New Authentication Block

A forwarder is forwarding messages, using syslog-auth. In this subsection, we describe how the forwarder builds the forwarding block, and thus the authentication blocks, which it uses to forward this message along.

We can break the forwarder's task into three cases: Forwarding unauthenticated messages, forwarding authenticated messages that haven't been forwarded before, and forwarding authenticated messages that were forwarded to us.

Note that the forwarder must parse, verify, and process authentication blocks just as the receiver must.  In the rest of this section, we assume the forwarder has already done everything it would have to do as a receiver, including drawing conclusions about received messages before forwarding them.

### 5.2.2. Unauthenticated Message

When an unauthenticated message arrives at the forwarder, the whole message text is treated as the original message text for this message. If the sender complies with the syslog standard, this message will be less than 1024 bytes long; if not, the message text is truncated to 1024 bytes before any further processing is done on it.

The forwarding block has four fields, and they are set as follows:

a. The Secure Path bit, set if the message has traveled over syslog-auth for its whole life. This bit is set to zero, since the message arrived here without any authentication. This is true even if the message was previously forwarded, and has a forwarding block; none of that information can be verified at this forwarder, so it is not relevant or useful.

b. Replay Resistant bit, set if every forwarder that has forwarded this message is sure this is not a replayed message. This bit is set to zero, since there's no way to verify that a message arriving unauthenticated isn't a replayed or spoofed message.

c. The IP address of the first syslog-auth sender or forwarder in this sequence of forwards. This is filled in with the forwarder's IP address, since this is the first IP address which can actually be known to be correct.

d. The number of times this message has been forwarded, 0-4095, encoded as two base-64 characters. This is set to 1, by definition, since this is the first time this particular message text has been forwarded.

#### 5.2.1.1. The Rest of the Authentication Block

The rest of the authentication block is set exactly as though this message were being originally sent by the forwarder, with only two exceptions: The Forwarding Block bit in the Flags field is set to one, and the Forwarding Block is included in the authentication block. The whole original message is treated as message text.

### 5.2.2. Authenticated Messages on First Hop

When an authenticated message arrives at the forwarder without a forwarding block, we build its forwarding block as follows:

a. The Secure Path bit, set if the message has traveled over syslog-auth for its whole life.  This bit is set to one.

b. Replay Resistant bit, set if every forwarder that has forwarded this message is sure this is not a replayed message. This bit is set to a one if:

(i) The Replay Vulnerable bit in the arriving message's authentication block is 0.

AND

(ii) The forwarder is implementing online replay detection for this kind of message.

Otherwise, the bit is cleared.

c. The IP address of the first syslog-auth sender or forwarder in this sequence of forwards. This field is set to the IP address of the sender.

d. The number of times this message has been forwarded, 0-4095, encoded as two base-64 characters. This is set to 1, by definition. (When the receiver gets this message, it will have been forwarded once.)

#### 5.2.2.1. The Rest of the Authentication Block

The rest of the authentication block is set exactly as though this message were being originally sent by the forwarder, with only three exceptions:

a. The Forwarding Block bit in the Flags field is set to one.

b. The Forwarding Block is included in the authentication block.

c. If the forwarder generates a storage MAC, it is done exactly as described in the section on sending syslog-auth messages, above, but the authentication block used is the original message's authentication block, not the one generated by the forwarder. Because forwarders' authentication blocks are stripped off by subsequent forwarders, the storage MAC cannot be based on any part of the forwarder's authentication block.

### 5.2.3. Authenticated Message, Additional Forwards

There is no inherent limit to how many times a message may be forwarded, though syslog-auth supports only allowing a message to be forwarded 4095 times. I don't expect this limit to ever become relevant in practice unless messages are being forwarded in an

endless loop.  However, syslog-auth forwarders MUST NOT forward a
message more than 4095 hops.

When we forward a message that has already been forwarded to us through syslog-auth, we build a new forwarding block based heavily on the old forwarding block. We do this as follows. Note that we always reference the fields of the last authentication block appended to the message.

a. The Secure Path bit, set if the message has traveled over syslog-auth for its whole life. This bit is set to the value of the Secure Path bit in the message we received.

b. Replay Resistant bit, set if every forwarder that has forwarded this message is sure this is not a replayed message. If this forwarder is able to guarantee that this message is not being replayed from the previous forwarder (e.g., if it is able to implement online replay detection), and if the previous forwarder set this bit, then the current forwarder sets it. Otherwise, the bit is cleared.

c. The IP address of the first syslog-auth sender or forwarder in this sequence of forwards. This value is copied to the new forwarding block unchanged.

d. The number of times this message has been forwarded, 0-4095, encoded as two base-64 characters. The forwarder takes this value, increments it by one, and checks to see if the result is greater than 4095. If so, the message MUST NOT be forwarded, but may be stored, discarded, or otherwise processed. Otherwise, the new value is copied into the same field in the new forwarding block.

### 5.2.3.1. The Rest of the Authentication Block

In the case of multiply-forwarded messages, we *replace* the authentication block appended by the previous forwarder with our own. In doing this, we generate an authentication block exactly like we would if we were sending this message ourselves, with three exceptions:

a. The Forwarding Block bit in the Flags field is set to one.

b. The Forwarding Block is included in our authentication block.

c. If there is a Storage MAC field in the authentication block we're replacing, we copy that field into our own authentication block. This is true even in the case that we would normally generate a Storage MAC of our own; in that case, there's simply no room for our Storage MAC, and so it isn't included.

Note that when a storage MAC field is computed on a forwarded message, it is computed over only the original message text and the first authentication block; its computation never includes any of the forwarder's authentication block, since that block may be stripped off.  The storage MAC is computed exactly as described in the section on sending syslog-auth messages, above, with the authentication block involved being the first authentication block.

## 6. Receiver-Side Issues

Everything the receiver does with regard to syslog-auth is intended to accomplish the following goals:

a. Provide as much real-time information about the logs being received as possible. Whenever possible, we would like to be able to give applications processing these log messages in real time enough information to decide whether this message might be a replay, or whether we can even promise the identity of the sender and the integrity of the message. This is particularly important to allow receivers to resist flooding attacks, by discarding replay-vulnerable messages when the disk is nearly full.

b. Provide enough information in the stored logs that a person or program reviewing these logs

   (i) Has as much information as possible about the log messages stored here, and any missing messages.

   (ii) Knows unambiguously what can be determined from the log messages stored here about these things.

These two goals are the only reason to bother with security for log files in the first place. In this section, we discuss providing online information where possible, and storing auxillary information about a stored log file to aid in offline reviewing of the logs. In the next section, we discuss techniques for using that auxillary information to review stored logs offline.

As a rule, message origin and integrity can be detected in real time for all syslog-auth messages, replays can be detected in near real time for some syslog-auth messages by some syslog-auth receivers, and gaps in messages can sometimes be detected offline using one receiver's logs, and in special cases, may be detectable in near real time for some senders and receivers. Even more critical than allowing receivers and reviewers of log messages to detect gaps and replays and such, is making it absolutely clear to a receiver or reviewer (which may be a computer program) what can and can't be determined from available data.

### 6.1. Description of Steps of Receiver Processing

This section discusses the steps of processing a received syslog-auth message:

a. Detect the syslog-auth message.

   b.  Parse it into its fields, and verify its integrity.

   c.  Process the fields, drawing conclusions about the status and
       security of the message.

d. Apply those conclusions, either through writing them to an auxillary log file, noting them in some other way associated with the log file, or changing some of the flags and fields used in forwarding this message.

Note that forwarded messages will generally have two authentication blocks. The last authentication block allows us to verify information from the last forwarder to see the message if it is forwarded, or the original sender if it is not forwarded. In general, we will share a key only with the last forwarder, not with the original sender of a forwarded message. If we share a key with the original sender of a forwarded message, we can check the original authentication block of the message as well as the last authentication block. In general, though, we will only check the last authentication block, since that's the only one we will have the key material for.

### 6.1.1. Notation

We will use the following notation in the rest of this section:

M is the syslog message.

M[i] is the ith byte in the message.

M[-i] is the ith byte from the end of the message.

M[i..j] is the string from the ith to the jth byte of M.

M[-i..-j] is the string from i characters before the end of the string to j characters before the end of the string.

Thus, if M = "Hello, world!", then:

```
M[0] = 'H'
M[4] = 'o',
M[-1] = '!'
M[-5] = 'o'.
M[2..4] = "llo,"
M[-3..-1] = "ld!"
```

A is the authentication block data structure, which has fields for all the possible fields in the authentication block.

The fields are denoted as follows:

```
A.cookie
A.version
A.flags
A.flags.destinationCounter
A.flags.superincreasingSessionID
A.flags.temporaryRebootSession
A.flags.replayVulnerable
A.flags.forwardingBlock
A.flags.storageMAC
A.flags.oldStyleReceiver
A.transmissionMAC
A.transmissionMAC.keyID
A.transmissionMAC.MAC
A.sessionID
A.globalCounter
A.destinationCounter
A.forwardingBlock
A.forwardingBlock.flags
A.forwardingBlock.flags.securePath
A.forwardingBlock.flags.replayResistant
A.forwardingBlock.entryPoint
A.forwardingBlock.hopCount
A.storageMAC
A.storageMAC.keyID
A.storageMAC.MAC
```

## 6.2. Detecting, Parsing and Verifying

Before anything else may be done, we must efficiently detect syslog-auth messages, parse them into their constituent fields, and verify their signatures.  This is done as follows:

### 6.2.1. Detecting Syslog-Auth Messages

To detect a syslog-auth message, we simply look at the end of the message for the cookie, ``authAUTH''.  If the cookie is missing, we assume this is not a syslog-auth message; if it is there, we assume it is a syslog-auth message, but don't make any assumptions that it's a valid one.

### 6.2.2. Parsing

The next step is parsing the authentication block into its constituent fields. To do this, we must:

a. Determine whether this message is long enough to safely process. That means first verifying that it's as long as the minimum length for an authentication block. In what follows, we assume that the receiver takes care never to allow buffer overruns.

b. Read the version of the authentication block, and verify that we can understand it. To get the version, we read M[-10..-9], and process it as a base-64 encoded string. The result is put into A.version. If A.version is not any of the versions understood by the receiver, then the message cannot be processed, and must be stored in a log file which is specified not to have been processed or discarded.

c. Read the flags of the authentication block, and use those flags to determine what fields will exist and what their size will be. The flags are at M[-12..-11], and are also base 64 encoded.

d. Read and store each field from the authentication block for verification and later processing. Based on the specific values of the flags, the right blocks of characters are placed into each field.

### 6.2.3. Verifying the Authentication Block

Before any processing can be done on the contents of these fields, we need to verify that the block and message have arrived intact. To do this, we do the following:

a. Check the keyIDs of all keys currently available to the receiver against the A.transmissionMAC.keyID. If there is a match, then use the corresponding key for the following operations. If there is no match, this message must be written to a log file which is specified as not having been processed at all by syslog-auth, or must be discarded.

b. Let A' be A with A.transmissionMAC.MAC = "00000000000".

c. Compute hmac-md5_{Key}(message-text,A') where , denotes concatenation of strings. Note the use of A', which is A with the transmission MAC field set to a block of ASCII zeros.

d. Compare the result of the hmac-md5 computation against A.transmissionMAC.MAC. If the two are equal, then the message

has been verified, and processing can contine.  If the two are
not equal, the message must either be stored in a log file
specified as not having been processed, or must be discarded.

### 6.3. Drawing Conclusions about Message Security

The message flags and receiver configuration, together, determine what fields are in the message, and thus what can be determined about the message. In this subsection, we describe how the receiver will draw conclusions about what can be guaranteed about the messages.

#### 6.3.1. Online Replay Detection

The receiver sees a sequence of syslog-auth messages coming from each sender. It needs to be able to detect replay attempts in real time, so that it can't be swamped by replayed messages and have its disk filled up.

There are three steps necessary to do this:

a. Verify the authentication block and MAC. If that is not valid, discard the message.

b. Check to see whether the Replay Vulnerable bit is set; if so, treat this message like an unauthenticated message for purposes of online replay detection. It may still be worthwhile to try to discard accidental replays, but there's no point in spending any time trying to detect intentional ones.

c. Otherwise, treat this message as a syslog-auth message with replay resistance.

##### 6.3.1.1. Syslog-auth Messages with Replay Resistance

Messages in this category contain the information necessary to *always* detect replays. Whether and how this is done depends on the implementation and resources available. The basic idea is to accept messages arriving slightly out of order, but not to accept messages which have already been seen, or whose reboot session ID indicates a replay. The main difficulty here is with changes of session ID from a sender.

When a syslog-auth message arrives with the Replay Vulnerable bit clear, it means that this combination of reboot session ID and global message counter has only been generated and used in a message once. The sender generates these two numbers in such a way that this is somehow assured.

##### 6.3.1.2. The Replay Window

A simple way to detect replays is the "replay window."  This divides the problem of deciding whether the current message has been seen before into two simpler problems:

a. Is this message even in the range of possible non-replayed messages?

b. If so, is this message in the list of recently-received messages in that range?

Whenever a new message arrives, we first check to see if it is automatically disqualified by being outside our allowable range. The allowable range is defined in terms of both session IDs and global message counters, in a way that will be described further below. If the message isn't automatically disqualified, then we check to see whether we've already seen it. If it passes both checks, then it is accepted, the message is added to the list of in-range messages that have recently been received (the "window"), and the range may be adjusted based on this message.

Note that the replay window needs to be larger for messages with long forwarding paths, since each hop in the forwarding path can introduce additional delays or re-orderings.

A message with the Replay Vulnerable flag not set comes with a promise from the sender: its reboot session ID and global message counter, taken together, are guaranteed to be unique among messages ever sent by this sender. Together, these two fields make up a unique identifier for each message sent by a sender.

At any given time, the replay window has zero, one, or two session IDs active. (It would have zero active if the sender hadn't sent a message in a long time; it would have two active if the sender had recently changed session IDs, as it might after a reboot. In nearly all cases, we expect it to have one active reboot session ID.)

In the discussion below, we assume there is some number, N, such that we would be very surprised to see a message sent before N other messages that arrived after all of them.

When a new message arrives with its Replay Vulnerable flag not set, there are three categories into which we can immediately place it:

a. Old -- for example, if it has an active session ID, but a message counter much lower than the highest message counter received for that session ID, then it will be discarded. Similarly, if it has a session ID that's not currently active, but which is determined to be old, then the message is discarded. Note that even if we don't discard messages whose MACs fail, we SHOULD discard messages that are unambigously replayed.

b. Current session ID and counter -- in this case, it has a currently active session ID and a counter that's within the range of counter values we're expecting. The session ID and message counter of this message are added to the list of recently-received messages, and the range of messages to be accepted may be updated as a result.

c. New session ID -- in this case, a new session ID has occurred, and the new session ID is not known to be a replayed session ID. In this case, a new session ID becomes active in our window. We have to deal with this in various ways, as will be discussed below.

### 6.3.1.3. New Session IDs and Discontinuities

Because syslog-auth doesn't assume reliable delivery, it is possible to get "discontinuities." A discontinuity is a gap between two blocks of messages, about which the receiver can usually tell nothing except that a block of messages sent between these two blocks was never received. For example, when a message arrives with a new session ID, this implies that the sender has rebooted recently. We will never know whether we saw the last message sent from the old session ID, since there is no later message to show a gap. Further, when a message arrives with a new session ID from a sender that generates those session IDs pseudorandomly, it's impossible to know whether there were other session IDs generated in-between which were never seen. (This could happen if the network between the sender and receiver went down for a long time.) Such situations are noted as discontinuities, and MUST be reported in the authenticated log file.

### 6.3.1.4. Detecting Replayed Session IDs

When the Superincreasing Session ID flag is on, detecting a replayed session ID is easy: we simply check to see if the session ID of the new message is greater than all currently active session IDs, and if not, we reject it.

When this flag is off, we must check the list of all session IDs ever sent by this sender since it's been using its current key. This sounds like searching through a long list, but there are two things that make it less demanding than it sounds:

a. New session IDs should occur very rarely; very few machines reboot or otherwise lose all context all that often. A message with a new session ID should in practice not arrive from a sender more than a few times a day in the worst case. This means that it's acceptable for the test of a session ID to take a few seconds. It might even be reasonable to set up one machine on a local network to keep track of the pseudorandom session IDs for all the senders.

b. The list of previously seen pseudorandom session IDs from a given sender is expected to be pretty small. Suppose in the worst case that we see one new session ID from some sender per hour, and that this sender continues using the same key for five years. The result will be a list of less than 44,000 session IDs. A hash table of 65,536 entries will handle this list efficiently, and the last sixteen bits of the pseudorandom session IDs can be used as the key for the hash table. A syslog server handling many senders will presumably have a lot of disk space; even five years' session IDs for 1000 senders in the worst case will take up only about 64 MB.

Despite this, there will be some receivers that cannot do online replay detection for syslog-auth messages with pseudorandom session IDs.  This is acceptable.  However, the authenticated logs MUST indicate that replay detection isn't done on these session IDs, in the same field that shows discontinuities for new session IDs.

### 6.3.1.5. Unauthenticated Messages and Replay-Vulnerable Messages

Unauthenticated messages can never be kept from being used to flood a receiver.  That means that a receiver that handles unauthentication messages SHOULD take precautions to resist flooding attacks.  These include keeping much more storage than is necessary, and keeping separate storage for authenticated and unauthenticated senders, so that an attacker can't fill up the receiver's storage with unauthenticated garbage messages, and thus prevent the receipt of syslog-auth messages.

Similar steps SHOULD be taken for syslog-auth messages that are vulnerable to replay, as described above.

### 6.3.1.6. Syslog-Auth Forwarded Messages

Forward messages present a special problem, because even when they're forwarded through syslog-auth, they may not have always been protected by syslog-auth, or some forwarding machine may not have implemented online replay detection.

To deal with this, the forwarding block carries a flag (the Replay Resistant bit) which indicates whether the forwarder can guarantee that this message isn't a replay.

If a forwarded message arrives at a receiver for storage, and the Replay Resistant bit is not set, the fact that this message (and presumably the whole log file from this sender via this forwarding pattern) is not replay resistant MUST be specified in the log.

### 6.3.2. Detecting Gaps

If the authentication block contains the field A.destinationCounter, then we can detect any gaps in the messages we were sent.  The destination counter is incremented by one each time a message is sent to a given destination, so if there are missing destination counter values in the received messages, we know that those messages didn't make it to us.  Cryptographic methods can't tell us why the messages didn't arrive (e.g., whether this is due to random packets being dropped or malicious action).  If syslog-auth is being used over reliable delivery mechanisms, then gaps in the sequence of received messages is strong evidence of malicious tampering.

Note that we can use the replay window mechanism to look for gaps. As each message falls out of the replay window, we can check to see whether it's leaving any gaps more than N messages back, where N is the replay window size.  If so, we note the gap.

Also note that we don't actually need to note gaps in the auxiliary log; we merely have to note whether or not sufficient information exists in the original log to note gaps, and if so, what information that is.

## 6.4. Applying Conclusions

Once the receiver has drawn conclusions about what can be guaranteed about a given message cryptographically, it needs to have a way of applying these conclusions.  There are three ways this may be done:

a.  Writing conclusions out as an auxillary log file.

b.  Using the observations to change flags and such in forwarded messages.

c.  Using the observations to alter processing based on these messages.

### 6.4.1. Writing Out Conclusions about Syslog-Auth Messages

When the messages are being stored on the receiver, they need to be stored in such a way that a reviewer doesn't need the secret key shared between the sender and receiver to verify a message.  For this purpose, we propose an auxillary log file.

This log file is related to a given log file, and includes additional information about what can is known from syslog-auth about a given message or range of messages.

Entries in this auxillary log file are of the form:

a.  Key ID, Reboot Session ID (identify the machine and session)

b.  startGlobalCounter, endGlobalCounter (identify the range of messages; * is valid on either end).

c.  blob (32-bit encoding of promises being made)

d.  textDescription ASCII text of the promises being made.

The fields are separated by commas.  The result is something like

```
     3f48d17787acce4437836b6e,00000348a490,*,*,c1f13839,
                 No replay or gap detection possible online.

  9d1eac9bcc3ab07657eb35ee,0000898de7bd,*,000000000048,09dee718,
                 No replay detection possible.

  8837174d47328e71937014e2,*,*,091726aa,
                 Session-ID is pseudorandoml cannot be sequenced with
                         other session IDs
```

### 6.4.2. Applying Conclusions in Forwarded Messages

The conclusions drawn in subsection 6.3 are also useful when the message in question is being forwarded. Knowledge about replay and gap detection must be passed on to the next receiver, and this can be done by setting appropriate bits in the flags of the authentication block and its forwarding block. In particular, the following flags may be affected:

```
A.flags.replayVulnerable
A.forwardingBlock.flags.replayResistant
```

### 6.4.3. Applying Conclusions in Online Processing

These conclusions may also be of great value for systems doing online processing of log files, e.g., for intrusion detection. Such systems can, for example, discard replayable messages when they're overloaded with other messages, or signal a problem when some authenticated device's messages are showing a suspicious number of missing messages.

## 7. Reviewing Syslog-Auth Logs Offline

We expect that in most applications, people or programs will review logs offline, perhaps hours or days after they have been written. Offline review of logs gives several potential advantages:

a. It's possible to postprocess log data, e.g., by putting all of the messages from the same reboot session ID in order of increasing global counter, and thus in order of original transmission.

b. It's possible to gather information from other receivers, and thus combine information from the same sender that went to different receivers, or from different senders that ought to be analyzed together.

c. Sometimes, an administrator will want to have an additional storage security key which is not available online. This might be used to resist attacks in which someone compromises the receiver machine, and alters logs on it.

This section describes offline review of logs received and stored by a syslog-auth receiver, including an auxillary log file.

## 7.1. Offline Replay Detection

Online replay detection is preferable, but offline replay detection is often good enough in practice.  Offline replay detection uses exactly the same techniques as online replay detection, but isn't nearly as time critical.

Offline replay detection uses the same techniques as online replay detection, described above.

## 7.2. Storage Security

Syslog-auth is mainly concerned with ensuring transmission security of log messages--that is, ensuring that they aren't altered on the wire between the sender and receiver. However, someone doing offline analysis of logs may also want to ensure that the logs haven't been tampered with since they've arrived. These two goals are rather distinct. It is important to note: storage security cannot be provided using a key held by the machine on which the logs are to be stored. If that machine isn't compromised, then an attacker can't alter the logs once they've arrived at it. If the machine is compromised, then an attacker can recover the key used, and can alter logs undetectably.

### 7.3.1. Syslog-Auth Storage MACs

Storage MACs are the only way that syslog-auth supports storage security directly. A syslog-auth message may, under normal conditions, contain zero, one, or two storage MACs. The key used for the storage MAC should be known to no online machine except the sender.

The storage MAC is intended to be verified offline. The owner of the log uses the storage MAC key, specified by the key ID in the storage MAC block, to verify that the message hasn't changed. There are two important points to consider, here:

a. If there is ever a disagreement about authenticity of a message between the receiver and the storage MAC (e.g., a message is stored as a valid message, but its storage MAC is not valid), this implies either a malfunction or a compromise somewhere on the forwarding path of the message, since the storage MAC was added to the machine.

b. The storage MAC prevents the receiver, if compromised, from altering stored log messages, but does not prevent it from deleting inconvenient log messages. (This is inevitable, since all that a storage MAC can authenticate is its message, not a sequence of messages that arrived.)

## 7.4. Detecting Gaps by Reassembling All Paths of Forwarded Messages

Forwarded messages and messages without destination counters can't be checked for meaningful gaps in the message sequence online.  (In this context, a meaningful gap is a gap caused by a message being lost or blocked, rather than by the sender having simply sent the message to a different receiver instead of this one.)  To check those logs for gaps that represent lost messages, we must reassemble the logs from a given sender, using the logs stored on all the ultimate receivers (possibly at the end of long chains of forwards).  We can then find any gaps in global message counters and identify them.8.    Acknowledgements

The author wishes to thank Alex Brown, Chris Calabrese, Jon Callas, Carson Gaspar, Drew Gross, Chris Lonvick, Darrin New, Marshall Rose, Holt Sorenson, and the whole bunch of Counterpane engineering and operations people who commented on early or late versions of this proposal.

## 9.   Bibliography

## A. Defining the Key ID.

The md5 hash function is known to have some weaknesses, particularly in terms of collision resistance.  However, it is also believed to be secure when used in the hmac construction, which is

```
  hmac_{K}(X) = md5(K xor Pad1,md5(K xor Pad2,X))
```

This applies the hash twice between the initial use of the key and the output.  All our security in this scheme is based on hmac-md5, so far.  One nice feature of this is that we can always change the hash function used, e.g., to SHA1 or SHA-256, with no real difficulties in terms of these definitions.  Our key IDs and MACs are both based on taking the low-order 64 or 96 bits of the hash value, so changing the output size of the hash should have no impact on them.

We use the following formula for Key ID:

```
  Key ID = low 96 bits( hmac_md5_{K}("KEYID") ).
```

The reasoning behind this is:

a.  We are already trusting hmac_md5 with all our security in this scheme, so it introduces no new trust requirements.

b.  There is no valid syslog-auth message with only the text "KEYID", since it does not include an authentication block.  A sender or

forwarder who follows the standard will never generate a MAC for
such a message.  Therefore, we need not worry about someone
trying to forge a syslog-auth message using the key ID as a
valid MAC, because there's no valid message that could be forged
this way.

c. The key ID is 96 bits based on the following logic: We never expect to see any receiver dealing with more than 2^{32} different senders' keys at once. In fact, we'd be surprised to see even one receiver on earth have to deal with that many senders. For that receiver, however, the probability that a pair of keys will collide is

$$choose(2^{32},2) * 2^{-96} \sim= 2^{63} * 2^{-96} = 2^{-33}.$$

This means that in this worst-case environment, the probability that we will get a pair of keys with the same key ID is still astronomically small.

Why is this last point important? The software implementing syslog-auth is going to use this key ID to decide which key to use when checking the message's MAC. So if there were ever a pair of keys that had the same key ID, and a given receiver had to deal with both at once, it would almost certainly fail to authenticate messages from one of the two colliding keys. (It's possible to implement this so that it detects unequal keys with colliding key IDs, and deals intelligently with them. But the code to do this will never be used, and so will probably almost never be tested. Even generating a test case for this situation requires a prohibitive amount of processing power (we expect it to take about 2^{48} operations)! So it will never be used, will never be tested, and so it just shouldn't be counted on.)

## B Author's Address

John Kelsey
Counterpane Internet Security
kelsey.j@ix.netcom.com

## C Full Copyright Statement