Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 3, 2019

R. Barnes
Cisco
J. Millican
Facebook
E. Omara
Google
K. Cohn-Gordon
University of Oxford
R. Robert
Wire

July 02, 2018

The Messaging Layer Security (MLS) Protocol
draft-barnes-mls-protocol-01

Abstract

Messaging applications are increasingly making use of end-to-end security mechanisms to ensure that messages are only accessible to the communicating endpoints, and not to any servers involved in delivering messages. Establishing keys to provide such protections is challenging for group chat settings, in which more than two participants need to agree on a key but may not be online at the same time. In this document, we specify a key establishment protocol that provides efficient asynchronous group key establishment with forward secrecy and post-compromise security for groups in size ranging from two to thousands.

Status of This Memo

Table of Contents

1. Introduction

DISCLAIMER: This is a work-in-progress draft of MLS and has not yet seen significant security analysis. It should not be used as a basis for building production systems.

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH The source for this draft is maintained in GitHub. Suggested changes should be submitted as pull requests at https://github.com/ekr/mls-protocol. Instructions are on that page as well. Editorial changes can be managed in GitHub, but any substantive change should be discussed on the MLS mailing list.

A group of agents who want to send each other encrypted messages needs a way to derive shared symmetric encryption keys. For two parties, this problem has been studied thoroughly, with the Double Ratchet emerging as a common solution [doubleratchet] [signal]. Channels implementing the Double Ratchet enjoy fine-grained forward secrecy as well as post-compromise security, but are nonetheless efficient enough for heavy use over low-bandwidth networks.

For a group of size greater than two, a common strategy is to unilaterally broadcast symmetric "sender" keys over existing shared symmetric channels, and then for each agent to send messages to the group encrypted with their own sender key. Unfortunately, while this improves efficiency over pairwise broadcast of individual messages and (with the addition of a hash ratchet) provides forward secrecy, it is difficult to achieve post-compromise security with sender keys. An adversary who learns a sender key can often indefinitely and passively eavesdrop on that sender's messages. Generating and distributing a new sender key provides a form of post-compromise security with regard to that sender. However, it requires

computation and communications resources that scale linearly as the size of the group.

In this document, we describe a protocol based on tree structures that enable asynchronous group keying with forward secrecy and post-compromise security. This document describes two candidate approaches, one using "asynchronous ratcheting trees" [art], the other using an asynchronous key-encapsulation mechanism for tree structures called TreeKEM. Both mechanisms allow the members of the group to derive and update shared keys with costs that scale as the log of the group size. The use of Merkle trees to store identity information allows strong authentication of group membership, again with logarithmic cost.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

[TODO: The architecture document uses "Client" instead of "Participant". Harmonize terminology.]

Participant: An agent that uses this protocol to establish shared cryptographic state with other participants. A participant is defined by the cryptographic keys it holds. An application may use one participant per device (keeping keys local to each device) or sync keys among a user's devices so that each user appears as a single participant.

Group: A collection of participants with shared cryptographic state.

Member: A participant that is included in the shared state of a group, and has access to the group's secrets.

Initialization Key: A short-lived Diffie-Hellman key pair used to introduce a new member to a group. Initialization keys can be published for both individual participants (UserInitKey) and groups (GroupInitKey).

Leaf Key: A short-lived Diffie-Hellman key pair that represents a group member's contribution to the group secret, so called because the participants leaf keys are the leaves in the group's ratchet tree.

Identity Key: A long-lived signing key pair used to authenticate the sender of a message.

Terminology specific to tree computations is described in Section 5.

We use the TLS presentation language [I-D.ietf-tls-tls13] to describe the structure of protocol messages.

3. Basic Assumptions

This protocol is designed to execute in the context of a Messaging Service (MS) as described in [I-D.omara-mls-architecture]. In particular, we assume the MS provides the following services:

- A long-term identity key provider which allows participants to authenticate protocol messages in a group. These keys MUST be kept for the lifetime of the group as there is no mechanism in the protocol for changing a participant's identity key.

- A broadcast channel, for each group, which will relay a message to all members of a group. For the most part, we assume that this channel delivers messages in the same order to all participants. (See Section 9 for further considerations.)

- A directory to which participants can publish initialization keys, and from which participant can download initialization keys for other participants.

4. Protocol Overview

The goal of this protocol is to allow a group of participants to exchange confidential and authenticated messages. It does so by deriving a sequence of keys known only to group members. Keys should be secret against an active network adversary and should have both forward and post-compromise secrecy with respect to compromise of a participant.

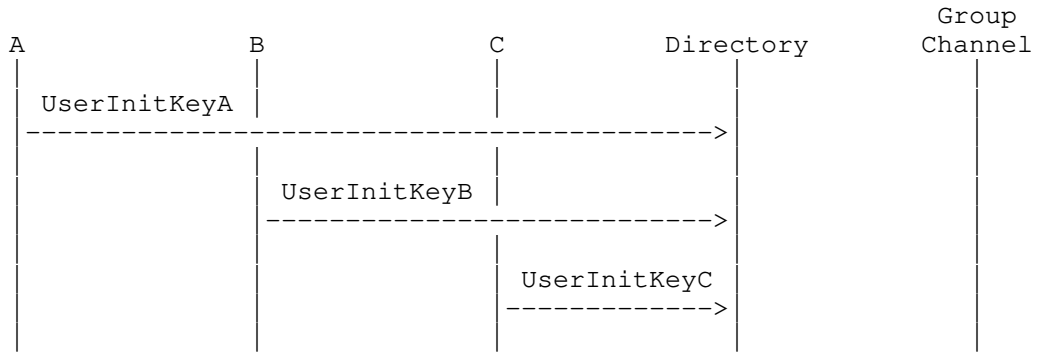
We describe the information stored by each participant as a _state_, which includes both public and private data. An initial state, including an initial set of participants, is set up by a group creator using the _Init_ algorithm and based on information pre-published by the initial members. The creator sends the _GroupInit_ message to the participants, who can then set up their own group state and derive the same shared key. Participants then exchange messages to produce new shared states which are causally linked to their predecessors, forming a logical Directed Acyclic Graph (DAG) of states. Participants can send _Update_ messages for post-compromise secrecy and new participants can be added or existing participants removed from the group.

The protocol algorithms we specify here follow. Each algorithm specifies both (i) how a participant performs the operation and (ii) how other participants update their state based on it.

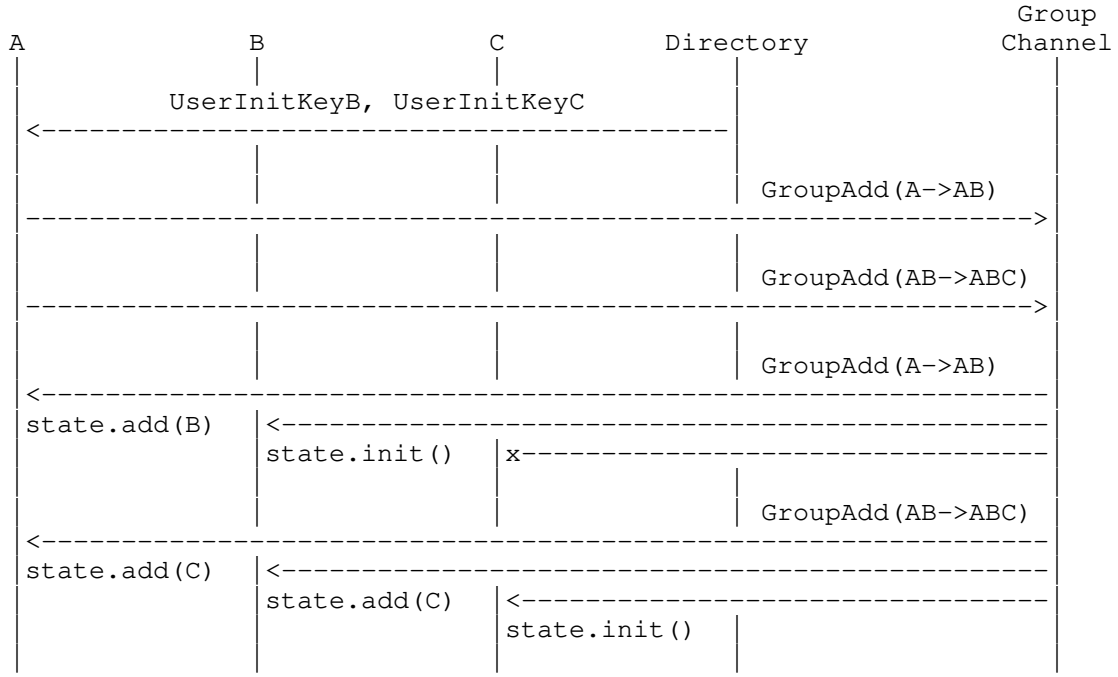
There are four major operations in the lifecycle of a group:

- Adding a member, initiated by a current member
- Adding a member, initiated by the new member
- Key update
- Removal of a member

Before the initialization of a group, participants publish UserInitKey objects to a directory provided to the Messaging Service.



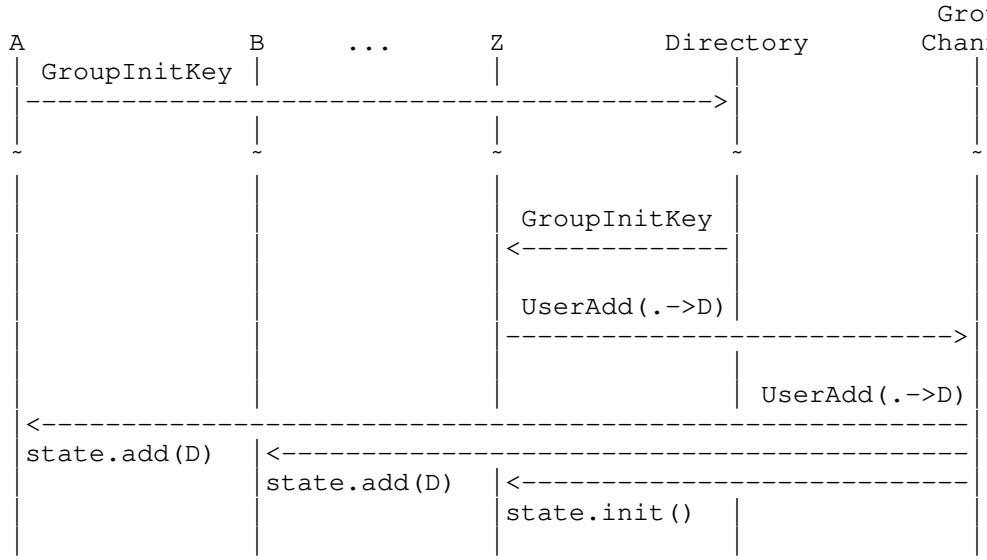
When a participant A wants to establish a group with B and C, it first downloads InitKeys for B and C. It then initializes a group state containing only itself and uses the InitKeys to compute GroupAdd messages to add B and C, in a sequence chosen by A. These messages are broadcasted to the Group, and processed in sequence by B and C. Messages received before a participant has joined the group are ignored. Only after A has received its GroupAdd messages back from the server does it update its state to reflect their addition.



Subsequent additions of group members proceed in the same way. Any member of the group can download an InitKey for a new participant and broadcast a GroupAdd which the current group can use to update their state and the new participant can use to initialize its state.

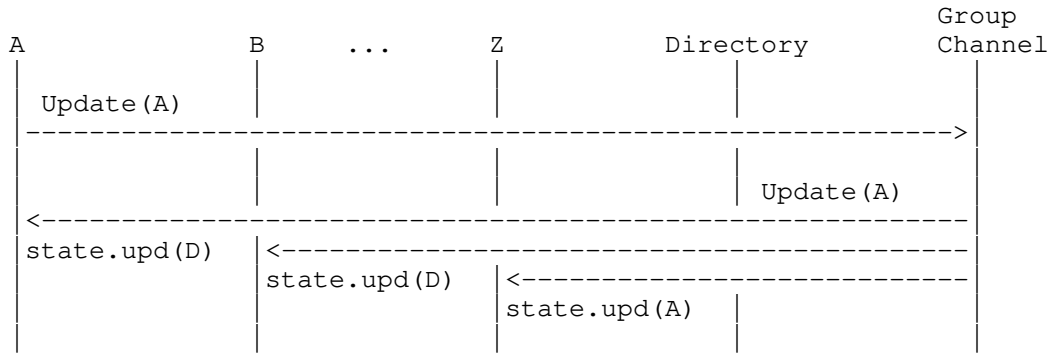
It is sometimes necessary for a new participant to join without an explicit invitation from a current member. For example, if a user that is authorized to be in the group logs in on a new device, that device will need to join the group as a new participant, but will not have been invited.

In these "user-initiated join" cases, the "InitKey + Add message" flow is reversed. We assume that at some previous point, a group member has published a GroupInitKey reflecting the current state of the group (A, B, C). The new participant Z downloads that GroupInitKey from the directory, generates a UserAdd message, and broadcasts it to the group. Once current members process this message, they will have a shared state that also includes Z.



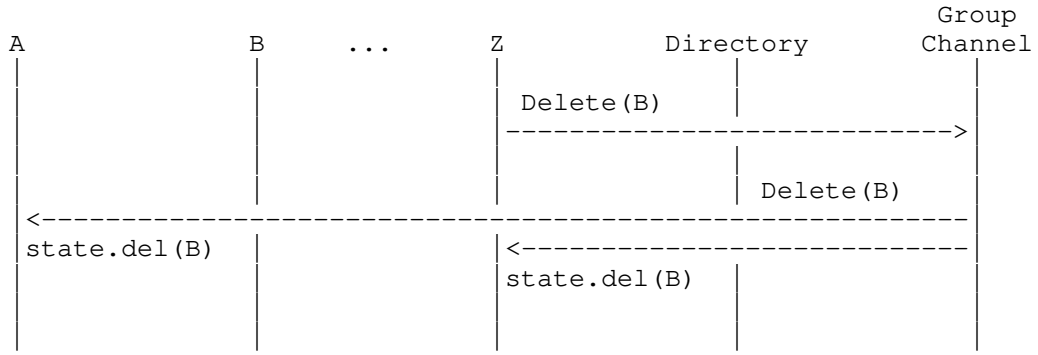
To enforce forward secrecy and post-compromise security of messages, each participant periodically updates its leaf key, the DH key pair that represents its contribution to the group key. Any member of the group can send an Update at any time by generating a fresh leaf key pair and sending an Update message that describes how to update the group key with that new key pair. Once all participants have processed this message, the group's secrets will be unknown to an attacker that had compromised the sender's prior leaf private key.

It is left to the application to determine the interval of time between Update messages. This policy could require a change for each message, or it could require sending an update every week or more.



Users are deleted from the group in a similar way, as a key update is effectively removing the old leaf from the group. Any member of the

group can generate a Delete message that adds new entropy to the group state that is known to all members except the deleted member. After other participants have processed this message, the group's secrets will be unknown to the deleted participant. Note that this does not necessarily imply that any member is actually allowed to evict other members; groups can layer authentication-based access control policies on top of these basic mechanism.



5. Binary Trees

The protocol uses two types of binary tree structures:

- Merkle trees for efficiently committing to a set of group participants.
- Ratchet trees for deriving shared secrets among this group of participants.

The two trees in the protocol share a common structure, allowing us to maintain a direct mapping between their nodes when manipulating group membership. The "nth" leaf in each tree is owned by the "nth" group participant.

5.1. Terminology

We use a common set of terminology to refer to both types of binary tree.

Trees consist of various different types of _nodes_. A node is a _leaf_ if it has no children, and a _parent_ otherwise; note that all parents in our Merkle or ratchet trees have precisely two children, a _left_ child and a _right_ child. A node is the _root_ of a tree if it has no parents, and _intermediate_ if it has both children and parents. The _descendants_ of a node are that node, its children, and the descendants of its children, and we say a tree _contains_ a

node if that node is a descendant of the root of the tree. Nodes are _siblings_ if they share the same parent.

A _subtree_ of a tree is the tree given by the descendants of any node, the _head_ of the subtree The _size_ of a tree or subtree is the number of leaf nodes it contains. For a given parent node, its _left subtree_ is the subtree with its left child as head (respectively _right subtree_).

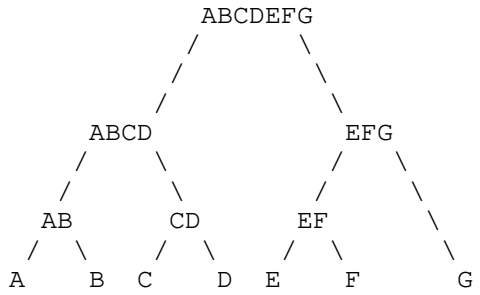
All trees used in this protocol are left-balanced binary trees. A binary tree is _full_ (and _balanced_) if it its size is a power of two and for any parent node in the tree, its left and right subtrees have the same size. If a subtree is full and it is not a subset of any other full subtree, then it is _maximal_.

A binary tree is _left-balanced_ if for every parent, either the parent is balanced, or the left subtree of that parent is the largest full subtree that could be constructed from the leaves present in the parent's own subtree. Note that given a list of "n" items, there is a unique left-balanced binary tree structure with these elements as leaves. In such a left-balanced tree, the "k-th" leaf node refers to the "k-th" leaf node in the tree when counting from the left, starting from 0.

The _direct path_ of a root is the empty list, and of any other node is the concatenation of that node with the direct path of its parent. The _copath_ of a node is the list of siblings of nodes in its direct path, excluding the root, which has no sibling. The _frontier_ of a tree is the list of heads of the maximal full subtrees of the tree, ordered from left to right.

For example, in the below tree:

- The direct path of C is (C, CD, ABCD)
- The copath of C is (D, AB, EFG)
- The frontier of the tree is (ABCD, EF, G)



We extend both types of tree to include a concept of "blank" nodes; which are used to replace group members who have been removed. We expand on how these are used and implemented in the sections below.

(Note that left-balanced binary trees are the same structure that is used for the Merkle trees in the Certificate Transparency protocol [I-D.ietf-trans-rfc6962-bis].)

5.2. Merkle Trees

Merkle trees are used to efficiently commit to a collection of group members. We require a hash function, denoted H, to construct this tree.

Each node in a Merkle tree is the output of the hash function, computed as follows:

- Leaf nodes: "H(0x01 || leaf-value)"
- Parent nodes: "H(0x02 || left-value || right-value)"
- Blank leaf nodes: "H(0x00)"

The below tree provides an example of a size 2 tree, containing identity keys "A" and "B".

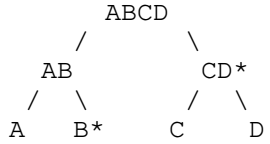
```
             * H(2 || H(1 || A) || H(1 || B))
            / \
           /   \
H(1 || A) *     * H(1 || B)
```

In Merkle trees, blank nodes appear only at the leaves. In computation of intermediate nodes, they are treated in the same way as other nodes.

5.2.1. Merkle Proofs

A proof of a given leaf being a member of the Merkle tree consists of the value of the leaf node, as well as the values of each node in its copath. From these values, its path to the root can be verified; proving the inclusion of the leaf in the Merkle tree.

In the below tree, we denote with a star the Merkle proof of membership for leaf node "A". For brevity, we notate "Hash(0x02 || A || B)" as "AB".



5.3. Ratchet Trees

Ratchet trees are used for generating shared group secrets. In this section, we describe the structure of a ratchet tree, along with two ways to manage a ratchet tree, called ART and TreeKEM.

To construct these trees, we require:

- A Diffie-Hellman finite-field group or elliptic curve
- A Derive-Key-Pair function that produces a key pair from an octet string
- A hash function (TreeKEM only)

A ratchet tree is a left-balanced binary tree, in which each node contains up to three values:

- A secret octet string (optional)
- An asymmetric private key (optional)
- An asymmetric public key

The private key and public key for a node are derived from its secret value using the Derive-Key-Pair operation.

The relationships between nodes are different for ART and TreeKEM. In either case, the ratchet tree structure ensures the following property: A party can compute the secret value for the root of the tree if and only if that party holds the secret value for another

node lower in the tree (together with public information). Each participant holds one leaf secret; each participant can update the root secret by changing their leaf secret.

5.3.1. Ratchet Trees for ART

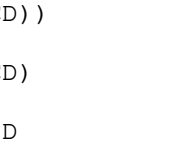
In ART the contents of a parent node are computed from its children as follows:

- parent_secret = DH(left_child, right_child)

- parent_private, parent_public = Derive-Key-Pair(parent_secret)

Ratchet trees are constructed as left-balanced trees, defined such that each parent node's key pair is derived from the Diffie-Hellman shared secret of its two child nodes. To compute the root secret and private key, a participant must know the public keys of nodes in its copath, as well as its own leaf private key.

For example, the ratchet tree consisting of the private keys (A, B, C, D) is constructed as follows:



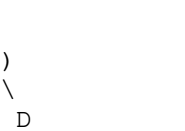
5.3.2. Ratchet Trees for TreeKEM

In TreeKEM, the contents of a parent node are computed from one of its children as follows:

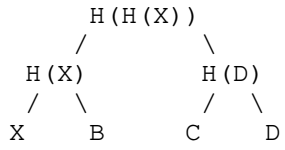
- parent_secret = Hash(child_secret)

- parent_private, parent_public = Derive-Key-Pair(parent_secret)

The contents of the parent are based on the latest-updated child. For example, if participants with leaf secrets A, B, C, and D join a group in that order, then the resulting tree will have the following structure:



If the first participant subsequently changes its leaf secret to be X, then the tree will have the following structure.



5.3.3. Ratchet Tree Updates

In order to update the state of the group such as adding and removing participants, MLS messages are used to make changes to the group's ratchet tree. While the details of update processing differ between ART and TreeKEM (as described below), in both cases the participant proposing an update to the tree transmits a representation of a set of tree nodes along the direct path from a leaf to the root. Other participants in the group can use these nodes to update their view of the tree, aligning their copy of the tree to the sender's.

In ART, the transmitted nodes are represented by their public keys. Receivers process an update with the following steps:

1. Replace the public keys in the cached tree with the received values

2. Whenever a public key is updated for a node whose sibling has a private key populated:

   * Perform a DH operation and update the node's parent

   * Repeat the prior step until reaching the root

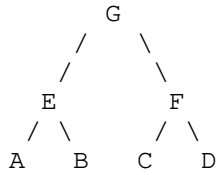
In TreeKEM, the sender transmits a node by sending the public key for the node and an encrypted version of the secret value for the node. The secret value is encrypted in such a way that it can be decrypted only by holders of the private key for one of its children, namely the child that is not in the direct path being transmitted. (That is, each node in the direct path is encrypted for holders of the private key for a node in the corresponding copath.) For leaf nodes, no encrypted secret is transmitted.

A TreeKEM update is processed with the following steps:

1. Compute the updated secret values * Identify a node in the direct path for which the local participant has the private key * Decrypt the secret value for that node * Compute secret values for ancestors of that node by hashing the decrypted secret

2. Merge the updated secrets into the tree * Replace the public keys for nodes on the direct path with the received public keys * For nodes where an updated secret was computed in step 1, replace the secret value for the node with the updated value

For example, suppose we had the following tree:



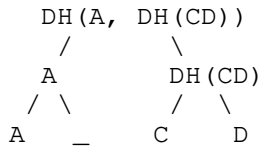
If an update is made along the direct path B-E-G, then the following values will be transmitted (using pk(X) to represent the public key corresponding to the secret value X and E(K, S) to represent public-key encryption to the public key K of the secret value S):

Public Key	Ciphertext
pk(G)	E(pk(F), G)
pk(E)	E(pk(A), E)
pk(B)	

5.3.4. Blank Ratchet Tree Nodes

Nodes in a ratchet tree can have a special value "_", used to indicate that the node should be ignored during path computations. Such nodes are used to replace leaves when participants are deleted from the group.

If any node in the copath of a leaf is _, it should be ignored during the computation of the path. For example, the tree consisting of the private keys (A, _, C, D) is constructed as follows for ART:



Replacing a node by _ in TreeKEM, means performing an update on any leaf without sending the new key to the the blanked leaf. In the

following example, participant A update its key to A' and derive the new sequence of keys up-to the path. Here A only send H(H(A')) to the parent node of C and D but does not send H(A') to B which evicts it from the Group. ~~~~~ H(H(A')) / \ H(A') H(C) / \ / \ A' _ C D ~~~~~

If two sibling nodes are both _, their parent value also becomes _.

Blank nodes effectively result in an unbalanced tree, but allow the tree management to behave as for a balanced tree for programming simplicity.

6. Group State

The state of an MLS group at a given time comprises:

- A group identifier (GID)
- A ciphersuite used for cryptographic computations
- A Merkle tree over the participants' identity keys
- A ratchet tree over the participants' leaf key pairs
- A message master secret (known only to participants)
- An add key pair (private key known only to participants)
- An init secret (known only to participants)

Since a group can evolve over time, a session logically comprises a sequence of states. The time in which each individual state is used is called an "epoch", and each state is assigned an epoch number that increments when the state changes.

MLS handshake messages provide each node with enough information about the trees to authenticate messages within the group and compute the group secrets.

Thus, each participant will need to store the following information about each state of the group:

1. The participant's index in the identity/ratchet trees
2. The private key associated with the participant's leaf public key

3. The private key associated with the participant's identity public key

4. The current epoch number

5. The group identifier (GID)

6. A subset of the identity tree comprising at least the copath for the participant's leaf

7. A subset of the ratchet tree comprising at least the copath for the participant's leaf

8. The current message encryption shared secret, called the master secret

9. The current add key pair

10. The current init secret

6.1. Cryptographic Objects

Each MLS session uses a single ciphersuite that specifies the following primitives to be used in group key computations:

- A hash function
- A Diffie-Hellman finite-field group or elliptic curve
- An AEAD encryption algorithm (TreeKEM only) [RFC5116]

The ciphersuite must also specify an algorithm "Derive-Key-Pair" that maps octet strings with the same length as the output of the hash function to key pairs for the asymmetric encryption scheme.

Public keys and Merkle tree nodes used in the protocol are opaque values in a format defined by the ciphersuite, using the following four types:

```
uint16 CipherSuite;
opaque DHPublicKey<1..2^16-1>;
opaque SignaturePublicKey<1..2^16-1>;
opaque MerkleNode<1..255>
```

[[OPEN ISSUE: In some cases we will want to include a raw key when we sign and in others we may want to include an identity or a certificate containing the key. This type needs to be extended to accommodate that.]]

6.1.1. ART with Curve25519 and SHA-256

This ciphersuite uses the following primitives:

- Hash function: SHA-256
- Diffie-Hellman group: Curve25519 [RFC7748]
- AEAD: N/A

Given an octet string X, the private key produced by the Derive-Key-Pair operation is SHA-256(X). (Recall that any 32-octet string is a valid Curve25519 private key.) The corresponding public key is X25519(SHA-256(X), 9).

Implementations SHOULD use the approach specified in [RFC7748] to calculate the Diffie-Hellman shared secret. Implementations MUST check whether the computed Diffie-Hellman shared secret is the all-zero value and abort if so, as described in Section 6 of [RFC7748]. If implementers use an alternative implementation of these elliptic curves, they SHOULD perform the additional checks specified in Section 7 of [RFC7748]

6.1.2. ART with P-256 and SHA-256

This ciphersuite uses the following primitives:

- Hash function: SHA-256
- Diffie-Hellman group: secp256r1 (NIST P-256)
- AEAD: N/A

Given an octet string X, the private key produced by the Derive-Key-Pair operation is SHA-256(X), interpreted as a big-endian integer. The corresponding public key is the result of multiplying the standard P-256 base point by this integer.

P-256 ECDH calculations (including parameter and key generation as well as the shared secret calculation) are performed according to [IEEE1363] using the ECKAS-DH1 scheme with the identity map as key derivation function (KDF), so that the shared secret is the x-coordinate of the ECDH shared secret elliptic curve point represented as an octet string. Note that this octet string (Z in IEEE 1363 terminology) as output by FE2OSP, the Field Element to Octet String Conversion Primitive, has constant length for any given field; leading zeros found in this octet string MUST NOT be truncated.

(Note that this use of the identity KDF is a technicality. The complete picture is that ECDH is employed with a non-trivial KDF because MLS does not directly use this secret for anything other than for computing other secrets.)

Clients MUST validate remote public values by ensuring that the point is a valid point on the elliptic curve. The appropriate validation procedures are defined in Section 4.3.7 of [X962] and alternatively in Section 5.6.2.3 of [keyagreement]. This process consists of three steps: (1) verify that the value is not the point at infinity (O), (2) verify that for Y = (x, y) both integers are in the correct interval, (3) ensure that (x, y) is a correct solution to the elliptic curve equation. For these curves, implementers do not need to verify membership in the correct subgroup.

6.1.3. TreeKEM with Curve25519, SHA-256, and AES-128-GCM

This ciphersuite uses the following primities:

- Hash function: SHA-256
- Diffie-Hellman group: Curve25519 [RFC7748]
- AEAD: AES-128-GCM

DH and Derive-Key-Pair operations are performed in the same way as the corresponding ART ciphersuite.

Encryption keys are derived from shared secrets by taking the first 16 bytes of H(Z), where Z is the shared secret and H is SHA-256.

6.1.4. TreeKEM with P-256, SHA-256, and AES-128-GCM

This ciphersuite uses the following primities:

- Hash function: P-256
- Diffie-Hellman group: secp256r1 (NIST P-256)
- AEAD: AES-128-GCM

DH and Derive-Key-Pair operations are performed in the same way as the corresponding ART ciphersuite.

Encryption keys are derived from shared secrets by taking the first 16 bytes of H(Z), where Z is the shared secret and H is SHA-256.

6.2. Direct Paths

As described in Section 5.3.3, each MLS message needs to transmit node values along the direct path from a leaf to the root. In ART, this simply entails sending the public key for each node. In TreeKEM, the path contains a public key for the leaf node, and a public key and encrypted secret value for intermediate nodes in the path. In both cases, the path is ordered from the leaf to the root; each node MUST be the parent of its predecessor.

```
DHPublicKey ARTPath<0..2^16-1>;

struct {
    DHPublicKey ephemeral_key;
    opaque nonce<0..255>;
    opaque ciphertext<0..255>;
} ECIESCiphertext;

struct {
    DHPublicKey public_key;
    ECIESCiphertext ciphertext;
} TreeKEMNode;

struct {
  DHPublicKey leaf;
  TreeKEMNode intermediates<0..2^16-1>;
} TreeKEMPath;

struct {
    select (mode) {
        case ART: ARTPath;
        case TreeKEM: TreeKEMPath;
    };
} DirectPath;
```

When using TreeKEM, the ECIESCiphertext values encoding the encrypted secret values are computed as follows:

- Generate an ephemeral DH key pair (x, x*G) in the DH group specified by the ciphersuite in use
- Compute the shared secret Z with the node's other child
- Generate a fresh nonce N
- Encrypt the node's secret value using the AEAD algorithm specified by the ciphersuite in use, with the following inputs:

    * Key: A key derived from Z as specified by the ciphersuite

    * Nonce: A random nonce N of the size required by the algorithm

    * Additional Authenticated Data: The empty octet string

    * Plaintext: The secret value, without any further formatting

  o Encode the ECIESCiphertext with the following values:

    * ephemeral_key: The ephemeral public key x*G

    * nonce: The random nonce N

    * ciphertext: The AEAD output

Decryption is performed in the corresponding way, using the private key of the non-updated child and the ephemeral public key transmitted in the message.

6.3. Key Schedule

Group keys are derived using the HKDF-Extract and HKDF-Expand functions as defined in [RFC5869], as well as the functions defined below:

```
Derive-Secret(Secret, Label, ID, Epoch, Msg) =
     HKDF-Expand(Secret, HkdfLabel, Length)
```

Where HkdfLabel is specified as:

```
struct {
    uint16 length = Length;
    opaque label<7..255> = "mls10 " + Label;
    opaque group_id<0..2^16-1> = ID;
    uint32 epoch = Epoch;
    opaque message<1..2^16-1> = Msg
} HkdfLabel;
```

The Hash function used by HKDF is the ciphersuite hash algorithm. Hash.length is its output length in bytes. In the below diagram:

  o HKDF-Extract takes its Salt argument from the top and its IKM argument from the left

  o Derive-Secret takes its Secret argument from the incoming arrow

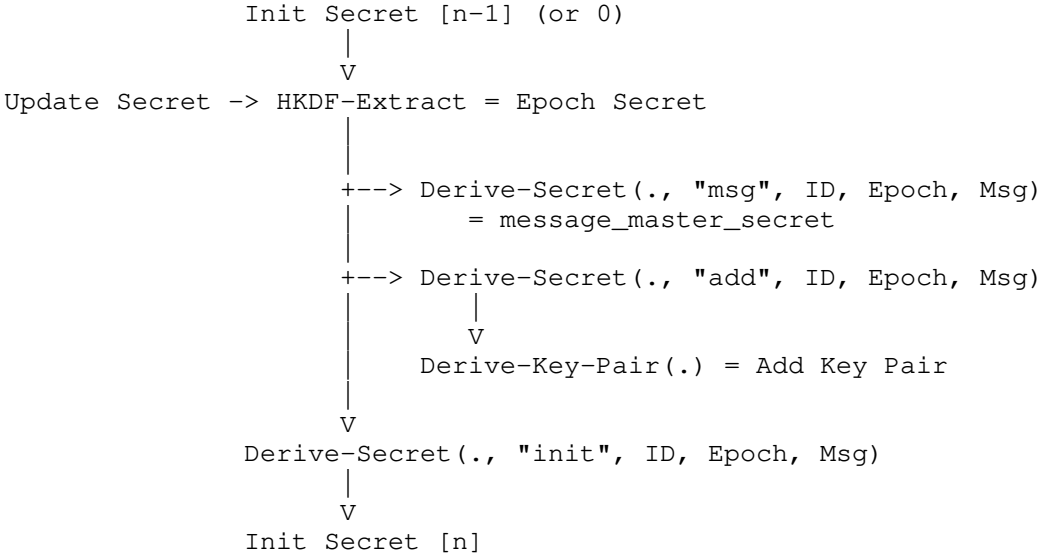
When processing a handshake message, a participant combines the following information to derive new epoch secrets:

- The init secret from the previous epoch
- The update secret for the current epoch
- The handshake message that caused the epoch change
- The current group identifier (GID) and epoch

The derivation of the update secret depends on the change being made, as described below.

For UserAdd or GroupAdd, the new user does not know the prior epoch init secret. Instead, entropy from the prior epoch is added via the update secret, and an all-zero vector with the same length as a hash output is used in the place of the init secret.

Given these inputs, the derivation of secrets for an epoch proceeds as shown in the following diagram:



7. Initialization Keys

In order to facilitate asynchronous addition of participants to a group, it is possible to pre-publish initialization keys that provide some public information about a user or group. UserInitKey messages provide information about a potential group member, that a group

member can use to add this user to a group asynchronously. GroupInitKey messages provide information about a group that a new user can use to join the group without any of the existing members of the group being online.

7.1. UserInitKey

A UserInitKey object specifies what ciphersuites a client supports, as well as providing public keys that the client can use for key derivation and signing. The client's identity key is intended to be stable throughout the lifetime of the group; there is no mechanism to change it. Init keys are intended to be used a very limited number of times, potentially once. (see Section 11.4).

The init_keys array MUST have the same length as the cipher_suites array, and each entry in the init_keys array MUST be a public key for the DH group or KEM defined by the corresponding entry in the cipher_suites array.

The whole structure is signed using the client's identity key. A UserInitKey object with an invalid signature field MUST be considered malformed. The input to the signature computation comprises all of the fields except for the signature field.

```
struct {
    CipherSuite cipher_suites<0..255>;
    DHPublicKey init_keys<1..2^16-1>;
    SignaturePublicKey identity_key;
    SignatureScheme algorithm;
    opaque signature<0..2^16-1>;
} UserInitKey;
```

7.2. GroupInitKey

A GroupInitKey object specifies the aspects of a group's state that a new member needs to initialize its state (together with an identity key and a fresh leaf key pair).

- The current epoch number
- The number of participants currently in the group
- The group ID
- The cipher suite used by the group
- The public key of the current update key pair for the group

- The frontier of the identity tree, as a sequence of hash values

- The frontier of the ratchet tree, as a sequence of public keys

GroupInitKey messages are not themselves signed. A GroupInitKey should not be published "bare"; instead, it should be published by constructing a handshake message with type "none", which will include a signature by a member of the group and a proof of membership in the group.

```
struct {
    uint32 epoch;
    uint32 group_size;
    opaque group_id<0..2^16-1>;
    CipherSuite cipher_suite;
    DHPublicKey add_key;
    MerkleNode identity_frontier<0..2^16-1>;
    TreeNode ratchet_frontier<0..2^16-1>;
} GroupInitKey;
```

8. Handshake Messages

Over the lifetime of a group, its state will change for:

- Group initialization

- A current member adding a new participant

- A new participant adding themselves

- A current participant updating its leaf key

- A current member deleting another current member

In MLS, these changes are accomplished by broadcasting "handshake" messages to the group. Note that unlike TLS and DTLS, there is not a consolidated handshake phase to the protocol. Rather, handshake messages are exchanged throughout the lifetime of a group, whenever a change is made to the group state. This means an unbounded number of interleaved application and handshake messages.

An MLS handshake message encapsulates a specific message that accomplishes a change to the group state. It also includes two other important features:

- A GroupInitKey so that a new participant can observe the latest state of the handshake and initialize itself

- A signature by a member of the group, together with a Merkle inclusion proof that demonstrates that the signer is a legitimate member of the group.

Before considering a handshake message valid, the recipient MUST verify both that the signature is valid, the Merkle inclusion proof is valid, and the sender is authorized to make the change according to group policy. The input to the signature computations comprises the entire handshake message except for the signature field.

The Merkle tree head to be used for validating the inclusion proof MUST be one that the recipient trusts to represent the current list of participant identity keys.

```
enum {
    none(0),
    init(1),
    user_add(2),
    group_add(3),
    update(4),
    delete(5),
    (255)
} HandshakeType;

struct {
    HandshakeType msg_type;
    uint24 inner_length;
    select (Handshake.msg_type) {
        case none:      struct{};
        case init:      Init;
        case user_add:  UserAdd;
        case group_add: GroupAdd;
        case update:    Update;
        case delete:    Delete;
    };

    uint32 prior_epoch;
    GroupInitKey init_key;

    uint32 signer_index;
    MerkleNode identity_proof<1..2^16-1>;
    SignaturePublicKey identity_key;

    SignatureScheme algorithm;
    opaque signature<1..2^16-1>;
} Handshake;
```

[[OPEN ISSUE: There will be a need to integrate credentials from an authentication service that associate identities to the identity keys used to sign messages. This integration will enable meaningful authentication (of identities, rather than keys), and will need to be done in such a way as to prevent unknown key share attacks.]]

[[OPEN ISSUE: The GroupAdd and Delete operations create a "double-join" situation, where a participants leaf key is also known to another participant. When a participant A is double-joined to another B, deleting A will not remove them from the conversation, since they will still hold the leaf key for B. These situations are resolved by updates, but since operations are asynchronous and participants may be offline for a long time, the group will need to be able to maintain security in the presence of double-joins.]]

[[OPEN ISSUE: It is not possible for the recipient of a handshake message to verify that ratchet tree information in the message is accurate, because each node can only compute the secret and private key for nodes in its direct path. This creates the possibility that a malicious participant could cause a denial of service by sending a handshake message with invalid values for public keys in the ratchet tree.]]

8.1. Init

[[OPEN ISSUE: Direct initialization is currently undefined. A participant can create a group by initializing its own state to reflect a group including only itself, then adding the initial participants. This has computation and communication complexity O(N log N) instead of the O(N) complexity of direct initialization.]]

8.2. GroupAdd

A GroupAdd message is sent by a group member to add a new participant to the group.

```
struct {
    PublicKey ephemeral;
    DirectPath add_path<1..2^16-1>;
} GroupAdd;
```

A group member generates this message using the following steps:

- Requesting from the directory a UserInitKey for the user to be added

- Generate a fresh ephemeral DH key pair

- Generate the leaf secret for the new node as the output of a DH operation between the ephemeral key pair and the public key in the UserInitKey

- Use the ratchet frontier and the new leaf secret to compute the direct path between the new leaf and the new root

The public key of the ephemeral key pair is placed in the "ephemeral" field of the GroupAdd message. The computed direct path is placed in the "add_path" field.

The new participant processes the message and the private key corresponding to the UserInitKey to initialize his state as follows:

- Compute the participant's leaf secret by combining the init key in the UserInitKey with the prior epoch's add key pair

- Use the frontiers in the GroupInitKey of the Handshake message to add its keys to the trees

An existing participant receiving a GroupAdd message first verifies the signature on the message, then verifies its identity proof against the identity tree held by the participant. The participant then updates its state as follows:

- Compute the new participant's leaf key pair by combining the leaf key in the UserInitKey with the prior epoch add key pair

- Update the group's identity tree and ratchet tree with the new participant's information

The update secret resulting from this change is the output of a DH computation between the private key for the root of the ratchet tree and the add public key from the previous epoch.

8.3. UserAdd

A UserAdd message is sent by a new group participant to add themself to the group, based on having already had access to a GroupInitKey for the group.

```
struct {
    DirectPath add_path;
} UserAdd;
```

A new participant generates this message using the following steps:

- Fetch a GroupInitKey for the group

- Use the frontiers in the GroupInitKey to add its keys to the trees
- Compute the direct path from the new participant's leaf in the new ratchet tree (the add_path).

An existing participant receiving a UserAdd first verifies the signature on the message, then verifies its identity inclusion proof against the updated identity tree expressed in the GroupInitKey of the Handshake message (since the signer is not included in the prior group state held by the existing participant). The participant then updates its state as follows:

- Update trees with the descriptions in the new GroupInitKey
- Update the local ratchet tree with the information in the UserAdd message, replacing any common nodes with the values in the add path

The update secret resulting from this change is the output of a DH computation between the private key for the root of the ratchet tree and the add public key from the previous epoch.

8.4. Update

An Update message is sent by a group participant to update its leaf key pair. This operation provides post-compromise security with regard to the participant's prior leaf private key.

```
struct {
    DirectPath update_path;
} Update;
```

The sender of an Update message creates it in the following way:

- Generate a fresh leaf key pair
- Compute its direct path in the current ratchet tree

An existing participant receiving a Update message first verifies the signature on the message, then verifies its identity proof against the identity tree held by the participant. The participant then updates its state as follows:

- Update the cached ratchet tree by replacing nodes in the direct path from the updated leaf using the information contained in the Update message

The update secret resulting from this change is the secret for the root node of the ratchet tree.

8.5. Remove

A Remove message is sent by a group member to remove one or more participants from the group.

```
struct {
    uint32 deleted;
    DirectPath path;
} Remove;
```

The sender of a Remove message generates it as as follows:

- Generate a fresh leaf key pair
- Compute its direct path in the current ratchet tree, starting from the deleted leaf (Note: In ART, this requires knowing the deleted node's copath)

An existing participant receiving a Delete message first verifies the signature on the message, then verifies its identity proof against the identity tree held by the participant. The participant then updates its state as follows:

- Update the cached ratchet tree by replacing nodes in the direct path from the deleted leaf using the information in the Delete message
- Update the cached ratchet tree and identity tree by replacing the deleted node's leaves with blank nodes

The update secret resulting from this change is the secret for the root node of the ratchet tree after both updates.

9. Sequencing of State Changes

[[OPEN ISSUE: This section has an initial set of considerations regarding sequencing. It would be good to have some more detailed discussion, and hopefully have a mechanism to deal with this issue.]]

Each handshake message is premised on a given starting state, indicated in its "prior_epoch" field. If the changes implied by a handshake messages are made starting from a different state, the results will be incorrect.

This need for sequencing is not a problem as long as each time a group member sends a handshake message, it is based on the most current state of the group. In practice, however, there is a risk that two members will generate handshake messages simultaneously, based on the same state.

When this happens, there is a need for the members of the group to deconflict the simultaneous handshake messages. There are two general approaches:

- Have the delivery service enforce a total order
- Have a signal in the message that clients can use to break ties

In ART, in either case, there is a risk of starvation. In a sufficiently busy group, a given member may never be able to send a handshake message, because he always loses to other members. The degree to which this is a practical problem will depend on the dynamics of the application.

In TreeKEM, because of the non-contributivity of intermediate nodes update messages can be applied one after the other without the Delivery Service having to reject any handshake message which makes TreeKEM more resilient regarding the concurrency of handshake messages. The Messaging system can decide to choose the order for applying the state changes. Note that there are certain cases (if no total ordering is applied by the Delivery Service) where the ordering is important for security, ie. all updates must be executed before deletes.

Regardless of how messages are kept in sequence, implementations MUST only update their cryptographic state when valid handshake messages are received. Generation of handshake messages MUST be stateless, since the endpoint cannot know at that time whether the change implied by the handshake message will succeed or not.

9.1. Server-Enforced Ordering

With this approach, the delivery service ensures that incoming messages are added to an ordered queue and outgoing messages are dispatched in the same order. The server is trusted to resolve conflicts during race-conditions (when two members send a message at the same time), as the server doesn't have any additional knowledge thanks to the confidentiality of the messages.

Messages should have a counter field sent in clear-text that can be checked by the server and used for tie-breaking. The counter starts at 0 and is incremented for every new incoming message. In ART, if

two group members send a message with the same counter, the first message to arrive will be accepted by the server and the second one will be rejected. The rejected message needs to be sent again with the correct counter number. In TreeKEM, the message does not necessarily need to be resent.

To prevent counter manipulation by the server, the counter's integrity can be ensured by including the counter in a signed message envelope.

This applies to all messages, not only state changing messages.

9.2. Client-Enforced Ordering

Order enforcement can be implemented on the client as well, one way to achieve it is to use a two step update protocol: the first client sends a proposal to update and the proposal is accepted when it gets 50%+ approval from the rest of the group, then it sends the approved update. Clients which didn't get their proposal accepted, will wait for the winner to send their update before retrying new proposals.

While this seems safer as it doesn't rely on the server, it is more complex and harder to implement. It also could cause starvation for some clients if they keep failing to get their proposal accepted.

9.3. Merging Updates

When TreeKEM is in use, it is possible to partly address the problem of concurrent changes by having the recipients of the changes merge them, rather than having the senders retry. Because the value of intermediate node is determined by its last updated child (as opposed to both its children in ART), TreeKEM updates can be merged by recipients as long as the recipients agree on an order - the only question is which node was last updated.

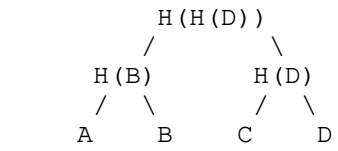
Recall that the processing of a TreeKEM update proceeds in two steps:

1. Compute updated secret values by hashing up the tree

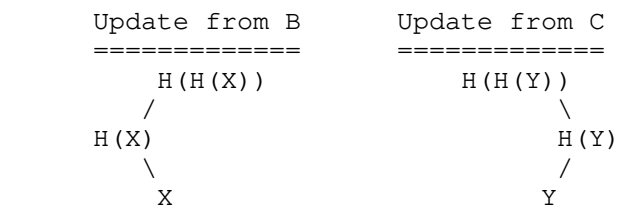
2. Update the tree with the new secret and public values

To merge an ordered list of updates, a recipient simply performs these updates in the specified order.

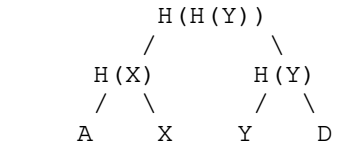
For example, suppose we have a tree in the following configuration:



Now suppose B and C simultaneously decide to update to X and Y, respectively. They will send out updates of the following form:



Assuming that the ordering agreed by the group says that B's update should be processed before C's, the other participants in the group will overwrite the root value for B with the root value from C, and all arrive at the following state:



10. Message Protection

[[OPEN ISSUE: This section has initial considerations about message protection. This issue clearly needs more specific recommendations, possibly a protocol specification in this document or a separate one.]]

The primary purpose of this protocol is to enable an authenticated group key exchange among participants. In order to protect messages sent among those participants, an application will need to specify how messages are protected.

For every epoch, the root key of the ratcheting tree can be used to derive key material for symmetric operations such as encryption/AEAD and MAC; AEAD or MAC MUST be used to ensure that the message originated from a member of the group.

In addition, asymmetric signatures SHOULD be used to authenticate the sender of a message.

In combination with server-side enforced ordering, data from previous messages is used (as a salt when hashing) to:

- add freshness to derived symmetric keys
- cryptographically bind the transcript of all previous messages with the current group shared secret

Possible candidates for that are:

- the key used for the previous message (hash ratcheting)
- the counter of the previous message (needs to be known to new members of the group)
- the hash of the previous message (proof that other participants saw the same history)

The requirement for this is that all participants know these values. If additional clear-text fields are attached to messages (like the counter), those fields MUST be protected by a signed message envelope.

Alternatively, the hash of the previous message can also be included as an additional field rather than change the encryption key. This allows for a more flexible approach, because the receiving party can choose to ignore it (if the value is not known, or if transcript security is not required).

11. Security Considerations

The security goals of MLS are described in [[the architecture doc]]. We describe here how the protocol achieves its goals at a high level, though a complete security analysis is outside of the scope of this document.

11.1. Confidentiality of the Group Secrets

Group secrets are derived from (i) previous group secrets, and (ii) the root key of a ratcheting tree. Only group members know their leaf private key in the group, therefore, the root key of the group's ratcheting tree is secret and thus so are all values derived from it.

Initial leaf keys are known only by their owner and the group creator, because they are derived from an authenticated key exchange protocol. Subsequent leaf keys are known only by their owner. [[TODO: or by someone who replaced them.]]

Note that the long-term identity keys used by the protocol MUST be distributed by an "honest" authentication service for parties to authenticate their legitimate peers.

11.2. Authentication

There are two forms of authentication we consider. The first form considers authentication with respect to the group. That is, the group members can verify that a message originated from one of the members of the group. This is implicitly guaranteed by the secrecy of the shared key derived from the ratcheting trees: if all members of the group are honest, then the shared group key is only known to the group members. By using AEAD or appropriate MAC with this shared key, we can guarantee that a participant in the group (who knows the shared secret key) has sent a message.

The second form considers authentication with respect to the sender, meaning the group members can verify that a message originated from a particular member of the group. This property is provided by digital signatures on the messages under identity keys.

[[OPEN ISSUE: Signatures under the identity keys, while simple, have the side-effect of preclude deniability. We may wish to allow other options, such as (ii) a key chained off of the identity key, or (iii) some other key obtained through a different manner, such as a pairwise channel that provides deniability for the message contents.]]

11.3. Forward and post-compromise security

Message encryption keys are derived via a hash ratchet, which provides a form of forward secrecy: learning a message key does not reveal previous message or root keys. Post-compromise security is provided by Update operations, in which a new root key is generated from the latest ratcheting tree. If the adversary cannot derive the updated root key after an Update operation, it cannot compute any derived secrets.

11.4. Init Key Reuse

Initialization keys are intended to be used only once and then deleted. Reuse of init keys is not believed to be inherently insecure [dhreuse], although it can complicate protocol analyses.

12. IANA Considerations

TODO: Registries for protocol parameters, e.g., ciphersuites

13. Contributors

- Benjamin Beurdouche
  INRIA
  benjamin.beurdouche@ens.fr

- Karthikeyan Bhargavan
  INRIA
  karthikeyan.bhargavan@inria.fr

- Cas Cremers
  University of Oxford
  cas.cremers@cs.ox.ac.uk

- Alan Duric
  Wire
  alan@wire.com

- Srinivas Inguva
  Twitter
  singuva@twitter.com

- Albert Kwon
  MIT
  kwonal@mit.edu

- Eric Rescorla
  Mozilla
  ekr@rtfm.com

- Thyla van der Merwe
  Royal Holloway, University of London
  thyla.van.der@merwe.tech

14. References

14.1. Normative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-28 (work in progress), March 2018.

[IEEE1363] "IEEE Standard Specifications for Password-Based Public-Key Cryptographic Techniques", IEEE standard, DOI 10.1109/ieeestd.2009.4773330, n.d..

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/info/rfc5116>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/info/rfc7748>.

[X962] ANSI, "Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62, 1998.

14.2. Informative References

[art] Cohn-Gordon, K., Cremers, C., Garratt, L., Millican, J., and K. Milner, "On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees", January 2018, <https://eprint.iacr.org/2017/666.pdf>.

[dhreuse] Menezes, A. and B. Ustaoglu, "On reusing ephemeral keys in Diffie-Hellman key agreement protocols", International Journal of Applied Cryptography Vol. 2, pp. 154, DOI 10.1504/ijact.2010.038308, 2010.

[doubleratchet] Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., and D. Stebila, "A Formal Security Analysis of the Signal Messaging Protocol", 2017 IEEE European Symposium on Security and Privacy (EuroS&P), DOI 10.1109/eurosp.2017.27, April 2017.

[I-D.ietf-trans-rfc6962-bis]
Laurie, B., Langley, A., Kasper, E., Messeri, E., and R. Stradling, "Certificate Transparency Version 2.0", draft-ietf-trans-rfc6962-bis-28 (work in progress), March 2018.

[keyagreement]
Barker, E., Chen, L., Roginsky, A., and M. Smid, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-56ar2, May 2013.

[signal] Perrin(ed), T. and M. Marlinspike, "The Double Ratchet Algorithm", n.d., <https://www.signal.org/docs/specifications/doubleratchet/>.

Authors' Addresses

Richard Barnes
Cisco

Email: rlb@ipv.sx

Jon Millican
Facebook

Email: jmillican@fb.com

Emad Omara
Google

Email: emadomara@google.com

Katriel Cohn-Gordon
University of Oxford

Email: me@katriel.co.uk

Raphael Robert
Wire

Email: raphael@wire.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: August 11, 2018

E. Omara
Google
B. Beurdouche
INRIA
E. Rescorla
Mozilla
S. Inguva
Twitter
A. Kwon
MIT
A. Duric
Wire

February 07, 2018

Messaging Layer Security Architecture
draft-omara-mls-architecture-01

Abstract

This document describes the architecture and requirements for the Messaging Layer Security (MLS) protocol. MLS provides a security layer for group messaging applications with from two to a large number of clients. It is meant to protect against eavesdropping, tampering, and message forgery.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 11, 2018.

Copyright Notice

Table of Contents

1. Introduction

End-to-end security is a requirement for instant messaging systems and is commonly deployed in many such systems. In this context, "end-to-end" captures the notion that users of the system enjoy some

level of security - with the precise level depending on the system design - even when the messaging service they are using performs unsatisfactorily.

Messaging Layer Security (MLS) specifies an architecture (this document) and an abstract protocol [MLSPROTO] for providing end-to-end security in this setting. MLS is not intended as a full instant messaging protocol but rather is intended to be embedded in a concrete protocol such as XMPP [RFC3920]. In addition, it does not specify a complete wire encoding, but rather a set of abstract data structures which can then be mapped onto a variety of concrete encodings, such as TLS [I-D.ietf-tls-tls13], CBOR [RFC7049], and JSON [RFC7159]. Implementations which adopt compatible encodings should be able to have some degree of interoperability at the message level, though they may have incompatible identity/authentication infrastructures.

This document is intended to describe the overall messaging system architecture which the MLS protocol fits into, and the requirements which it is intended to fulfill.

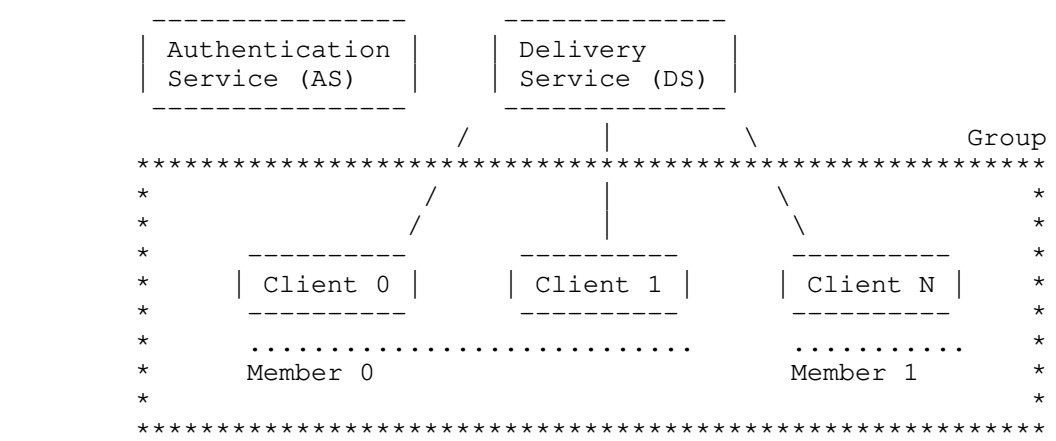
2. General Setting

A Group using a Messaging Service (MS) comprises a set of participants called Members where each Member is typically expected to own multiple devices, called Clients. A group may be as small as two members (the simple case of person to person messaging) or as large as thousands. In order to communicate securely, Group Members initially use services at their disposal to obtain the necessary secrets and credentials required for security.

The Messaging Service (MS) presents as two abstract services that allow Members to prepare for sending and receiving messages securely:

- An Authentication Service (AS) which is responsible for maintaining user long term identities, issuing credentials which allow them to authenticate each other, and potentially allowing users to discover each others long-term identity keys.

- A Delivery Service (DS) which is responsible for receiving and redistributing messages between group members. In the case of group messaging, the delivery service may also be responsible for acting as a "broadcaster" where the sender sends a single message to a group which is then forwarded to each recipient in the group by the DS. The DS is also responsible for storing and delivering initial public key material required in order to proceed with the group secret key establishment process.



In many systems, the AS and the DS are actually operated by the same entity and may even be the same server. However, they are logically distinct and, in other systems, may be operated by different entities, hence we show them as being separate here. Other partitions are also possible, such as having a separate directory server.

A typical group messaging scenario might look like this:

1. Alice, Bob and Charlie create accounts with a messaging service and obtain credentials from the AS.

2. Alice, Bob and Charlie authenticate to the DS and store some initial keying material which can be used to send encrypted messages to them for the first time. This keying material is authenticated with their long term credentials.

3. When Alice wants to send a message to Bob and Charlie, she contacts the DS and looks up their initial keying material. She uses these keys to establish a new set of keys which she can use to send encrypted messages to Bob and Charlie. She then sends the encrypted message(s) to the DS, which forwards them to the recipients.

4. Bob and/or Charlie respond to Alice's message. Their messages might trigger a new key derivation step which allows the shared group key to be updated to provide post-compromise security Section 3.2.2.1.

Clients may wish to do the following:

- create a group by inviting a set of other members;
- add one or more members to an existing group;
- remove one or more members from an existing group;
- join an existing group;
- leave a group;
- send a message to everyone in the group;
- receive a message from someone in the group.

At the cryptographic level, Clients in groups (and by extension Members) are peers. For instance, any Client should be able to add a member to a group. This is in contrast so some designs in which there is a single group controller who can modify the group. MLS is compatible with having group administration restricted to certain users, but we assume that those restrictions are enforced by authentication and access control. Thus, for instance, while it might be technically possible for any member to send a message adding a new member to a group, the group might have the policy that only certain members are allowed to make changes and thus other members can ignore or reject such a message from an unauthorized user.

2.1. Group, Members and Clients

In MLS a Group is defined as a set of Members who possibly use multiple endpoint devices (Clients) to interact with the Messaging Service. These Clients will typically correspond to end-user devices such as phones, web clients or other devices running MLS.

Each member device owns a long term identity key pair that uniquely defines its identity to other Members of the Group. Because a single Member may operate multiple devices simultaneously (e.g., a desktop and a phone) or sequentially (e.g., replacing one phone with another), the formal definition of a Group in MLS is the set of Clients that has legitimate knowledge of the shared (Encryption) Group Key established in the group key establishment phase of the protocol.

In some messaging systems, Clients belonging to the same Member must all share the same identity key pair, but MLS does not assume this. The MLS architecture considers the more general case and allows for important use cases, such as a Member adding a new Client when all their existing clients are offline.

MLS has been designed to provide similar security guarantees to all Clients, for all group sizes, even when it reduces to only two Clients.

2.2. Authentication Service

The basic function of the Authentication Service is to provide a trusted mapping from user identities (usernames, phone numbers, etc.), which exist 1:1 with Members, to identity keys, which may either be one per Client or may be shared amongst the Clients attached to a Member.

- A certificate authority or similar service which signs some sort of portable credential binding an identity to a key.

- A directory server which provides the key for a given identity (presumably this connection is secured via some form of transport security such as TLS).

By definition, the AS is invested with a large amount of trust. A malicious AS can impersonate - or allow an attacker to impersonate - any user of the system. This risk can be mitigated by publishing the binding between identities and keys in a public log such as Key Transparency (KT) [KeyTransparency]. It is possible to build a functional MLS system without any kind of public key logging, but such a system will necessarily be somewhat vulnerable to attack by a malicious or untrusted AS.

2.3. Delivery Service

The Delivery Service (DS) is expected to play multiple roles in the Messaging Service architecture:

- To act as a directory service providing the keying material (authentication keys and initial keying material) for Clients to use. This allows a Client to establish a shared key and send encrypted messages to other Clients even if the other Client is offline.

- To route messages between Clients and to act as a message broadcaster, taking in one message and forwarding it to multiple Clients (also known as "server side fanout")

Depending on the level of trust given by the Group to the Delivery Service, the functional and security guarantees provided by MLS may differ.

2.3.1. Key Storage

Upon joining the system, each Client stores its initial cryptographic key material with the DS. This key material represents the initial contribution from each member that will be used in the establishment of the shared group key. This initial keying material MUST be authenticated using the Client's identity key. Thus, the Client stores:

- A credential from the Authentication service attesting to the binding between the Member and the Client's identity key.
- The member's initial keying material signed with the Client's identity key.

As noted above, Members may have multiple Clients, each with their own keying material, and thus there may be multiple entries stored by each Member.

2.3.2. Key Retrieval

When a Client wishes to establish a group and send an initial message to that group, it contacts the DS and retrieves the initial key material for each other Member, verifies it using the identity key, and then is able to form a joint key with each other Client, and from those forms the group key, which it can use for the encryption of messages.

2.3.3. Delivery of messages and attachments

The DS's main responsibility is to ensure delivery of messages. Specifically, we assume that DSs provide:

- Reliable delivery: when a message is provided to the DS, it is eventually delivered to all group members.
- In-order delivery: messages are delivered to the group in the order they are received from a given Client and in approximately the order in which they are sent by Clients. The latter is an approximate guarantee because multiple Clients may send messages at the same time and so the DS needs some latitude in reordering between Clients.
- Consistent ordering: the DS must ensure that all Clients have the same view of message ordering.

Note that the DS may provide ordering guarantees by ensuring in-order delivery or by providing messages with some kind of sequence information and allowing clients to reorder on receipt.

The MLS protocol itself should be able to verify these properties. For instance, if the DS reorders messages from a Client or provides different Clients with inconsistent orderings, then Clients should be able to detect this misconduct. However, MLS need not provide mechanisms to recover from a misbehaving DS.

Note that some forms of DS misbehavior are still possible and difficult to detect. For instance, a DS can simply refuse to relay messages to and from a given Client. Without some sort of side information, other Clients cannot generally distinguish this form of Denial of Service (DoS) attack from the Client being actually offline.

2.3.4. Membership knowledge

Group membership is itself sensitive information and MLS is designed so that neither the DS nor the AS need have static knowledge of which Clients are in which Group. However, they may learn this information through traffic analysis. For instance, in a server side fanout model, the DS learns that a given Client is sending the same message to a set of other Clients. In addition, there may be applications of MLS in which the Group membership list is stored on some server associated with the MS.

2.3.5. Membership and offline members

Because Forward Secrecy (FS) and Post-Compromise Security (PCS) rely on the deletion and replacement of keying material, any Client which is persistently offline may still be holding old keying material and thus be a threat to both FS and PCS if it is later compromised. MLS doesn't inherently defend against this problem, but MLS-using systems should enforce some mechanism for doing so. Typically this will consist of evicting Clients which are idle for too long, thus containing the threat of compromise. The precise details of such mechanisms are a matter of local policy.

3. System Requirements

3.1. Functional Requirements

MLS is designed as a large scale group messaging protocol and hence aims to provide performance and safety to its users. Messaging systems that implement MLS must provide support for conversations involving two or more participants, and aim to scale to approximately

50,000 clients, typically including many Members using multiple devices.

3.1.1. Asynchronous Usage

No operation in MLS should require two distinct users to be online simultaneously. In particular, clients participating in conversations protected using MLS must be able to update shared keys, add or remove new members, and send messages and attachments without waiting for another user's reply.

Messaging systems that implement MLS must provide a transport layer for delivering messages asynchronously and reliably.

3.1.2. Recovery After State Loss

Conversation participants whose local MLS state is lost or corrupted must be able to reinitialize their state and continue participating in the conversation. This may entail some level of message loss, but should not result in permanent exclusion from the group.

3.1.3. Support for Multiple Devices

It is typically expected for Members of the Group to own different devices.

A new device can join the group and will be considered as a new Client by the protocol. This Client will not gain access to the history even if it is owned by someone who is already a Member of the Group. Restoring history is typically not allowed at the protocol level but applications may elect to provide such a mechanism outside of MLS.

3.1.4. Extensibility / Pluggability

Messages that don't affect the group state can carry an arbitrary payload with the purpose of sharing that payload between group members. No assumptions are made about the format of the payload.

3.1.5. Privacy

The protocol is designed in a way that limits the server-side (AS and DS) metadata footprint. The DS must only persist data required for the delivery of messages and avoid Personally Identifiable Information (PII) or other sensitive metadata wherever possible. A Messaging Service provider that has control over both the AS and the DS, will not be able to correlate encrypted messages forwarded by the DS, with the initial public keys signed by the AS.

3.1.6. Federation

The protocol aims to be compatible with federated environments. While this document does not specify all necessary mechanisms required for federation, multiple MLS implementations should be able to interoperate and to form federated systems.

3.1.7. Compatibility with future versions of MLS

It is important the multiple versions of MLS be able to coexist in the future. Thus, MLS must offer a version negotiation mechanism; this mechanism must prevent version downgrade attacks where an attacker would actively rewrite messages messages with a lower protocol version than the ones originally offered by the endpoints. When multiple versions of MLS are available, the negotiation protocol must guarantee that the version agreed upon will be the highest version supported in common by the group.

3.2. Security Requirements

3.2.1. Connections between Clients and Servers (one-to-one)

We assume that all transport connections are secured via some transport layer security mechanism such as TLS [I-D.ietf-tls-tls13]. However, as noted above, the security of MLS should generally survive compromise of the transport layer.

3.2.2. Message Secrecy and Authentication

The trust establishment step of the MLS protocol is followed by a conversation protection step where encryption is used by clients to transmit authenticated messages to other clients through the DS. This ensures that the DS doesn’t have access to the Group’s private content.

MLS aims to provide Secrecy, Integrity and Authentication for all messages.

Message Secrecy in the context of MLS means that only intended recipients (current group members), should be able to read any message sent to the group, even in the context of an active adversary as described in the threat model.

Message Integrity and Authentication mean that an honest Client should only accept a message if it was sent by a group member and that one Client must not be able to send a message which other Clients accept as being from another Client.

A corollary to this statement is that the AS and the DS can't read the content of messages sent between Members as they are not Members of the Group. MLS is expected to optionally provide additional protections regarding traffic analysis so as to reduce the ability of adversaries, or a compromised member of the messaging system, to deduce the content of the messages depending on (for example) their size. One of these protections includes padding messages in order to produce ciphertexts of standard length. While this protection is highly recommended it is not mandatory as it can be costly in terms of performance for clients and the MS.

Message content can be deniable if the signature keys are exchanged over a deniable channel prior to signing messages.

3.2.2.1. Forward and Post-Compromise Security

MLS provides additional protection regarding secrecy of past messages and future messages. These cryptographic security properties are Forward Secrecy (FS) and Post-Compromise Security (PCS).

FS means that access to all encrypted traffic history combined with an access to all current keying material on clients will not defeat the secrecy properties of messages older than the oldest key of the client. Note that this means that clients have the extremely important role of deleting appropriate keys as soon as they have been used with the expected message, otherwise the secrecy of the messages and the security for MLS is considerably weakened.

PCS means that if a group member is compromised at some time t but subsequently performs an update at some time t', then all MLS guarantees should apply to messages sent after time t'. For example, if an adversary learns all secrets known to Alice at time t, including both Alice's secret keys and all shared group keys, but Alice performs a key update at time t', then the adversary should be unable to violate any of the MLS security properties after time t'.

Both of these properties must be satisfied even against compromised DSs and ASs.

3.2.2.2. Membership Changes

MLS aims to provide agreement on group membership, meaning that all group members have agreed on the list of current group members.

Some applications may wish to enforce ACLs to limit addition or removal of group members, to privileged users. Others may wish to require authorization from the current group members or a subset

thereof. Regardless, MLS does not allow addition or removal of group members without informing all other members.

Once a Member is part of a Group, the set of devices controlled by the member should only be altered by an authorized member of the group. This authorization could depend on the application: some applications might want to allow certain other members of the group to add or remove devices on behalf of another member, while other applications might want a more strict policy and allow only the owner of the devices to add or remove them at the potential cost of weaker PCS guarantees.

Members who are removed from a group do not enjoy special privileges: compromise of a removed group member should not affect the security of messages sent after their removal.

3.2.2.3. Security of Attachments

The security properties expected for attachments in the MLS protocol are very similar to the ones expected from messages. The distinction between messages and attachments stems from the fact that the typical average time between the download of a message and the one from the attachments may be different. For many reasons (a typical reason being the lack of high bandwidth network connectivity), the lifetime of the cryptographic keys for attachments is usually higher than for messages, hence slightly weakening the PCS guarantees for attachments.

3.2.2.4. Denial of Service

In general we do not consider Denial of Service (DoS) resistance to be the responsibility of the protocol. However, it should not be possible for anyone to perform a trivial Denial of Service (DoS) attack from which it is hard to recover.

3.2.2.5. Deniability

As described in Section 4.4, MLS aims to provide data origin authentication within a group, such that one group member cannot send a message that appears to be from another group member. Additionally, it is a requirement of some services that a recipient be able to prove to the messaging service that a message was sent by a given Client, in order to report abuse. MLS should support both of these use cases. In some deployments, these services may be provided by mechanisms which allow the receiver to prove a message's origin to a third party (this if often called "non-repudiation"), but it should also be possible to operate MLS in a "deniable" mode where such proof

is not possible. [[OPEN ISSUE: Exactly how to supply this is still a protocol question.]]

4. Security Considerations

MLS adopts the Internet threat model [RFC3552] and therefore assumes that the attacker has complete control of the network. It is intended to provide the security services described in in the face of such attackers. In addition, these guarantees are intended to degrade gracefully in the presence of compromise of the transport security links as well as of both Clients and elements of the messaging system, as described in the remainder of this section.

4.1. Transport Security Links

[TODO: Mostly DoS, message suppression, and leakage of group membership.]

4.2. Delivery Service Compromise

MLS is intended to provide strong guarantees in the face of compromise of the DS. Even a totally compromised DS should not be able to read messages or inject messages that will be acceptable to legitimate Clients. It should also not be able to undetectably remove, reorder or replay messages.

However, a DS can mount a variety of DoS attacks on the system, including total DoS attacks (where it simply refuses to forward any messages) and partial DoS attacks (where it refuses to forward messages to and from specific Clients). As noted in Section 2.3.3, these attacks are only partially detectable by clients. Ultimately, failure of the DS to provide reasonable service must be dealt with as a customer service matter, not via technology.

Because the DS is responsible for providing the initial keying material to Clients, it can provide stale keys. This doesn't inherently lead to compromise of the message stream, but does allow it to attack forward security to a limited extent. This threat can be mitigated by having initial keys expire.

4.3. Authentication Service Compromise

A compromised AS is a serious matter, as the AS can provide incorrect or adversarial identities to clients. As noted in Section 2.2, mitigating this form of attack requires some sort of transparency/ logging mechanism. Without such a mechanism, MLS will only provide limited security against a compromised AS.

4.4. Client Compromise

In general, MLS only provides limited protection against compromised Clients. When the Client is compromised, then the attacker will obviously be able to decrypt any messages for groups in which the Client is a member. It will also be able to send messages impersonating the compromised Client until the Client updates its keying material (see Section 3.2.2.1). MLS attempts to provide some security in the face of client compromise.

In addition, a Client should not be able to send a message to a group which appears to be from another Client with a different identity. Note that if Clients from the same Member share keying material, then one will be able to impersonate another.

Finally, Clients should not be able to perform denial of service attacks Section 3.2.2.4.

5. Contributors

- Katriel Cohn-Gordon
  University of Oxford
  me@katriel.co.uk

- Cas Cremers
  University of Oxford
  cas.cremers@cs.ox.ac.uk

- Thyla van der Merwe
  Royal Holloway, University of London
  thyla.van.der@merwe.tech

- Jon Millican
  Facebook
  jmillican@fb.com

- Raphael Robert
  Wire
  raphael@wire.com

6. Informative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-23 (work in progress), January 2018.

[KeyTransparency]
Google, ., "Key Transparency", n.d., <https://KeyTransparency.org>.

[MLSPROTO]
Barnes, R., Millican, J., Omara, E., Cohn-Gordon, K., and R. Robert, "Messaging Layer Security Protocol", 2018.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <https://www.rfc-editor.org/info/rfc3552>.

[RFC3920] Saint-Andre, P., Ed., "Extensible Messaging and Presence Protocol (XMPP): Core", RFC 3920, DOI 10.17487/RFC3920, October 2004, <https://www.rfc-editor.org/info/rfc3920>.

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <https://www.rfc-editor.org/info/rfc7049>.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <https://www.rfc-editor.org/info/rfc7159>.

Authors' Addresses

Emad Omara
Google

Email: emadomara@google.com

Benjamin Beurdouche
INRIA

Email: benjamin.beurdouche@inria.fr

Eric Rescorla
Mozilla

Email: ekr@rtfm.com

Srinivas Inguva
Twitter

Email: singuva@twitter.com

Albert Kwon
MIT

Email: kwonal@mit.edu

Alan Duric
Wire

Email: alan@wire.com