Messaging Layer Security
Internet-Draft
Intended status: Informational
Expires: 7 August 2026

K. Kohbrok
R. Robert
Phoenix R&D

3 February 2026

A two-party profile for MLS
draft-kohbrok-mls-two-party-profile-00

Abstract

TODO Abstract

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://kkohbrok.github.io/draft-kohbrok-mls-two-party-profile/draft-kohbrok-mls-two-party-profile.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-kohbrok-mls-two-party-profile/.

Discussion of this document takes place on the Messaging Layer Security Working Group mailing list (mailto:mls@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/mls/. Subscribe at https://www.ietf.org/mailman/listinfo/mls/.

Source for this draft and an issue tracker can be found at https://github.com/kkohbrok/draft-kohbrok-mls-two-party-profile.

Status of This Memo

Table of Contents

1. Introduction

MLS is primarily designed to be a continuous group key agreement (and messaging) protocol. This document explores the special case of groups of two and thus MLS's use as a (non-group) continuous key agreement protocol with a focus on the use in synchronous scenarios.

The two-party profile of MLS described in this document can, for example, be used as a replacement for other key exchange protocols that don't provide post-compromise security and/or post-quantum security. This is the case with the key exchange components for many secure channel protocols today.

TODO: For now, this document specifies a simple two party profile based on vanilla MLS that assumes synchronous communication. In the future, we should explore the asynchronous case and add a more specialized version that trims unnecessary things like signatures in key update messages, etc.

2. Protocol overview

The synchronous case assumes that both parties are online. The party sending the first message is the initiator and the other party the responder. The protocol can be split into three phases.

1. Initial key agreement

2. Continuous key agreement

3. Resumption

In Phase 1, the initiator and responder exchange messages until they agree on a shared key and thus enter Phase 2.

In Phase 2, both parties can perform key-updates to achieve forward-secrecy (FS) and post-compromise security (PCS). To avoid de-synchronization, both parties have to await confirmation of each update by the other party before they can perform another one. This phase continues until the connection between the parties is interrupted.

Phase 3 allows either party to resume the connection. During resumption, initiator and responder exchange messages to renew their key material before they (re-)enter Phase 2.

While the synchronous two-party case is significantly simpler when it comes to the responsibilities of the MLS Delivery Service, it still requires both parties to interface with an Authentication Service to validate both initial credentials and any credential updates.

3. Initial key agreement

```
struct {
  MLSMessage key_package;
} ClientHello

struct {
  MLSMessage welcome;
} ServerHello
```

The initiator starts the key agreement part of the protocol by creating a KeyPackage and sending a ClientHello to the responder.

The responder inspects the KeyPackage and checks whether it supports the offered ciphersuite and whether the initiator has sufficient capabilities to support the connection.

The responder MUST interface with the AS to ensure that the credential in the KeyPackage is valid.

The responder then locally creates an MLS group and commits to an Add proposal containing the initiator’s KeyPackage. The responder sends the resulting Welcome message back to the initiator as part of a ServerHello message.

The initiator uses the Welcome to create its local group state. The initiator then inspects the group state and MUST interface with the AS to ensure that the credential of the responder is valid.

4. Continuous key agreement

```
struct {
  MLSMessage update
} ConnectionUpdate

struct {
  uint64 epoch;
} EpochKeyUpdate
```

After the initial key agreement phase, both parties can send an MLS commit with UpdatePath to update their key material. To ensure that both agree on the order of such commits and thus on the currently used key material, they must follow the following rules.

* Each party may send a ConnectionUpdate if they are not currently waiting for an EpochKeyUpdate to confirm a previous ConnectionUpdate

* If either party receives a ConnectionUpdate and they’re not currently waiting for an EpochKeyUpdate, they MUST validate and apply the commit and respond with an EpochKeyUpdate, where epoch is the group’s new epoch

* If the initiator receives a ConnectionUpdate while waiting for an EpochKeyUpdate, it MUST ignore the ConnectionUpdate and resume waiting

* If the responder receives a ConnectionUpdate while waiting for an EpochKeyUpdate, it MUST drop its locally pending commit and validate and apply the commit as if it hadn’t been waiting for an EpochKeyUpdate

* A party receiving a ConnectionUpdate MUST start using the key material of the new epoch after sending the EpochKeyUpdate

* A party sending a ConnectionUpdate MUST wait until they receive the corresponding EpochKeyUpdate before they start using the key material of the new epoch

5. Resumption

Either party may resume a previously interrupted protocol session based on that session’s group state. The party initiating the resumption becomes the initiator.

```
struct {
  MLSMessage commit;
} ResumptionRequest

struct {
  MLSMessage commit;
} ResumptionResponse
```

The initiator sends a Resumption message to the responder. If the initiator was waiting for an EpochKeyUpdate while the connection was interrupted, it MUST include the commit from the last ConnectionUpdate in the Resumption message. The initiator MUST then wait for a ResumptionResponse.

The responder receiving a ResumptionRequest MUST validate and apply the commit in the ResumptionRequest and create a commit with UpdatPath to send back as part of a ResumptionResponse.

If one of the parties receives a ResumptionRequest while waiting for a ResumptionResponse, their reaction depends whether they were the initial initiator or responder when the connection was first established. The initial initiator MUST drop the ResumptionRequest and continue waiting. The initial responder MUST drop its pending commit and instead validate and apply the incoming commit before responding with a fresh commit as part of a ResumptionResponse.

6. Security Considerations

TODO Security

7. IANA Considerations

This document has no IANA actions.

Acknowledgments

TODO acknowledge.

Contributors

Russ Housley
Vigil Security
Email: housley@vigilsec.com

Authors' Addresses

Konrad Kohbrok
Phoenix R&D
Email: konrad@ratchet.ing

Raphael Robert
Phoenix R&D
Email: ietf@raphaelrobert.com