

Messaging Layer Security  
Internet-Draft  
Intended status: Informational  
Expires: 1 January 2026

K. Kohbrok  
Phoenix R&D  
30 June 2025

Decentralized Messaging Layer Security  
draft-kohbrok-mls-dmls-01

## Abstract

Messaging Layer Security (MLS) provides strong end-to-end security guarantees for group messaging including Forward Secrecy (FS) and Post-Compromise Security (PCS). MLS requires a Delivery Service (DS) component to facilitate agreement between group members on the order of Commit messages. In decentralized settings without an authoritative entity to enforce ordering, group members will likely have to retain key material so they can process commits out-of-order.

Retaining key material, however, significantly reduces the FS of the protocol. This draft specifies Decentralized MLS (DMLS), based on the the Fork-Resilient Continuous Group Key Agreement protocol FREEK proposed by Alwen et al. [FRCGKA]. In essence, DMLS extends MLS such that key material can be retained to process Commits out-of-order with recuded impact to FS, thus allowing safer deployment in decentralized environments.

## 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://phnx-im.github.io/dmls-spec/draft-kohbrok-mls-dmls.html>. Status information for this document may be found at <https://datatracker.ietf.org/doc/draft-kohbrok-mls-dmls/>.

Discussion of this document takes place on the Messaging Layer Security 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/phnx-im/dmls-spec>.

## 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 1 January 2026.

#### Copyright Notice

Copyright (c) 2025 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

#### Table of Contents

1. Introduction . . . . .	3
2. Epoch identifiers . . . . .	3
3. DMLS Messages . . . . .	4
4. DMLS key schedule . . . . .	4
5. Puncturable pseudorandom function . . . . .	5
6. State management . . . . .	5
6.1. Example: Federated scenario with server support . . . . .	6
6.1.1. Server behaviour . . . . .	6
6.1.2. Client behaviour . . . . .	7
7. Security Considerations . . . . .	7
8. IANA Considerations . . . . .	8
9. References . . . . .	8
9.1. Normative References . . . . .	8
9.2. Informative References . . . . .	8
Acknowledgments . . . . .	8
Author's Address . . . . .	8

## 1. Introduction

...

DMLS allows group members to keep around old group state a little more safely, because the init secret of old epoch states is punctured. However, keeping an old epoch state around is still not as safe as deleting it in the first place. See Section 7 for more details.

While DMLS is thus safer to use in scenarios where members must be able to process old commits, it is still not as safe as the use of vanilla MLS with its strict deletion schedule.

Even when using DMLS, applications should take care that group state forks are short-lived and group members (and/or assisting servers) endeavour to resolve forks as soon as possible.

In contrast scenarios should be avoided where multiple forks are long-lived. For example, if two or more parts of a group are not in contact with one-another and effectively run their own fork of the same group.

## 2. Epoch identifiers

In MLS, each epoch is identified by a 64 bit unsigned integer, with the epoch increasing by one with each commit. The integer identifies epochs uniquely as long as there is only one chain of Commits. However, in a decentralized context there can be multiple commits for the same epoch, which means that an integer is not sufficient to uniquely identify an epoch. For example, if two group member send a commit at the same time with different subsets of group members receiving a different commit first. After processing the newly arrived Commit, all group members would be in the same epoch, but in different group states. For subsequently arriving messages, it is unclear from the integer designating the epoch, which state the message belongs to. In such scenarios it is important that epochs are uniquely identifiable.

The `dmls_epoch` can be used for this purpose.

```
pseudocode dmls_epoch = DeriveSecret(epoch_secret, "epoch")
```

A `dmls_epoch` is represented by byte strings of length `KDF.Nh` (thus depending on the group's ciphersuite). The byte string identifying an epoch is derived from the epoch's `epoch_secret`.

### 3. DMLS Messages

As regular MLSMessages only contain integer-based epoch identifiers, this section introduces DMLSMessages, a simple wrapper that adds a `dmls_epoch` header to an MLSMessage.

```
struct {
    MLSMessage message;
    opaque dmls_epoch<V>;
} DMLSMessage
```

### 4. DMLS key schedule

DMLS uses a modified version of the MLS key schedule that allows the derivation of multiple `init_secrets`, where each init secret can be used to initialize a subsequent epoch.

The individual `init_secrets` are derived through a puncturable pseudorandom function (PPRF, see Section 5) keyed by the `base_init_secret`.

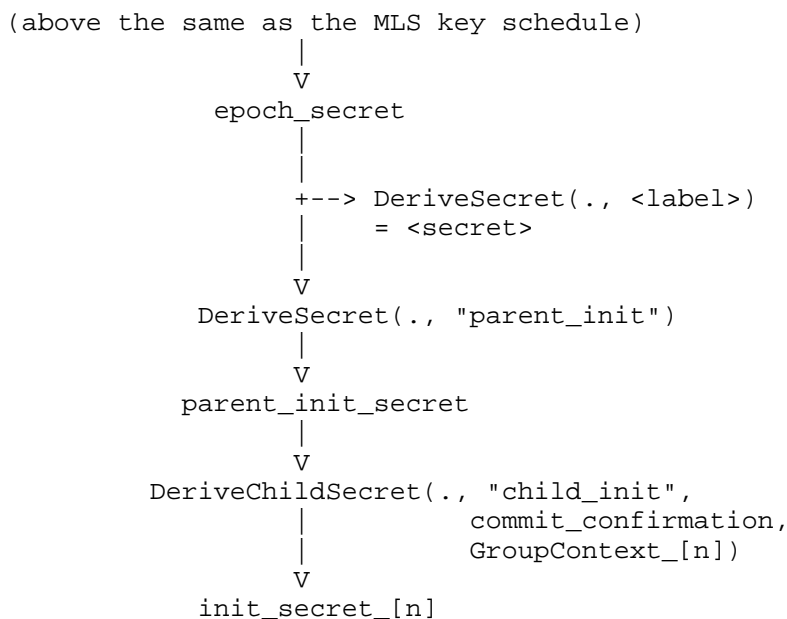


Figure 1: The DMLS Key Schedule

```
commit_confirmation = DeriveSecret(path_secret[n], "conf")

DeriveChildSecret(prf_key, label, input_secret, context) =
    DeriveFSSecret(prf_key, ExpandWithLabel(input_secret, label, context, KDF.Nh))
```

## 5. Puncturable pseudorandom function

A PPRF allows the derivation of keys in a forward secure way. In particular, a PRF that was evaluated with a given key and input can't be evaluated with those same parameters again. Storing the original input key thus doesn't harm the forward secrecy of (deleted) output keys.

The MLS Secret Tree as defined in [RFC9420] already represents a PPRF and needs to be modified only slightly for the purpose of this document.

In the context of MLS, the Secret Tree has as many leaves as the group has members. To derive child init secrets, the same tree is used but with KDF.Nh leaves.

The function `DeriveFSSecret(secret, input)` thus follows these steps:

- \* Check if secret and input are of length KDF.Nh
- \* With secret as the root node secret and input as the leaf index, derive the direct path nodes and the copath nodes as defined in Section 9 of [RFC9420]
- \* With leaf\_node\_secret as the resulting secret compute the final output using `DeriveSecret(leaf_node_secret, "pprf")`

## 6. State management

As outlined in Section 7 DMLS makes it safer to retain old MLS group states. As such, it enables an `_eventually consistent_` delivery service as described in Section 5.2.2. of [RFC9750], i.e. one that tolerates out-of-order delivery of messages. This in turn allows the use of DMLS in applications with highly decentralized architectures.

The lack of a strong agreement on message order, however, leads to the various state-agreement problems inherent to distributed systems and independent of (D)MLS.

More concretely, applications need to specify the following.

- \* What fork should a client choose when sending a message?

- \* When can an old epoch state be safely deleted?
- \* When can a fork be safely consolidated and deleted?
- \* How should two forks be consolidated?

The answers to these questions depend on the application's specific architecture and other requirements and are thus outside of the scope of this document.

The remainder of this section shows an example of how these questions can be answered in a decentralized scenario.

#### 6.1. Example: Federated scenario with server support

The architecture of the first application consists of a federation of servers, where each server serves one or more clients.

The servers can queue messages for their clients and determine the order of handshake messages for the groups their clients are in.

In this system, forks can only occur if a subset of servers lose connectivity from the rest.

##### 6.1.1. Server behaviour

The exact nature of the algorithm used by the servers to agree on a commit for the next epoch is out of scope for this example. However, the requirement of such an algorithm are:

- \* Agreement: If one or more commits reach one or more servers at the same time the algorithm should facilitate agreement between the servers that can currently reach one-another.
- \* Netsplit detection: Servers should be able to detect if the federation of servers has split. If connections between individual servers fail, other servers can act as forwarding proxies between the disconnected servers.

If a netsplit is detected, servers will fack back on agreement within their subgroup of servers. Agreed-upon commits and other group messages are buffered for later fan-out to the rest of the federation.

If a netsplit ends, all messages buffered during the netsplit are delivered to their respective destinations.

If a fork has occurred during the netsplit, the fork with the lexicographically higher confirmation tag in the first differing commit wins and the servers restart their agreement algorithm with respect to that fork.

If there was another netsplit during a netsplit, the same rules apply.

When fanning out group messages to their clients, servers include a flag to indicate whether the old commit state should be retained due to an ongoing netsplit.

#### 6.1.2. Client behaviour

Clients generally only have one fork of a given group. If they send a message, that's the fork they choose.

Epoch states are generally deleted immediately after processing the next commit, except when the server indicates that the epoch state should be retained.

When a client receives a commit for an old epoch (i.e. if a fork occurs), it retains the fork where the first differing commit has the lexicographically higher confirmation tag.

To keep the scenario simple, losing forks are simply deleted.

### 7. Security Considerations

The use of a PPRF to derive init secrets for new epochs significantly improves forward secrecy in scenarios where clients need to be able to process multiple commits for a given epoch.

However, PPRF only improves forward secrecy for the init secret. Group members must still delay the deletion of other secrets such as the (private) decryption keys for the nodes in the ratchet tree. This delay in deletion compromises the forward secrecy of the protocol. Conversely, the fact that other group members might encrypt to those keys in turn weakens the protocol's post-compromise security.

It is thus still advisable to delete old epoch states as soon as the functional requirements of the application allows it.

A rule that will be safe for most applications, for example, is that an old epoch state can be deleted once each group member has sent a commit on at least one fork "upstream" of that epoch state. This signals that all group members have agreed to continue using this particular fork of the group state.

For effective forward secrecy and post-compromise security it is thus advisable to choose a state management algorithm where members converge on a shared fork rather than continuously using different forks of the same group.

## 8. IANA Considerations

This document has no IANA actions.

## 9. References

### 9.1. Normative References

- [RFC9420] Barnes, R., Beurdouche, B., Robert, R., Millican, J., Omara, E., and K. Cohn-Gordon, "The Messaging Layer Security (MLS) Protocol", RFC 9420, DOI 10.17487/RFC9420, July 2023, <<https://www.rfc-editor.org/rfc/rfc9420>>.
- [RFC9750] Beurdouche, B., Rescorla, E., Omara, E., Inguva, S., and A. Duric, "The Messaging Layer Security (MLS) Architecture", RFC 9750, DOI 10.17487/RFC9750, April 2025, <<https://www.rfc-editor.org/rfc/rfc9750>>.

### 9.2. Informative References

- [FRCGKA] Alwen, J., Mularczyk, M., and Y. Tselekounis, "Fork-Resilient Continuous Group Key Agreement", 22 February 2024, <<https://eprint.iacr.org/2023/394.pdf>>.

## Acknowledgments

TODO acknowledge.

## Author's Address

Konrad Kohbrok  
Phoenix R&D  
Email: [konrad.kohbrok@datashrine.de](mailto:konrad.kohbrok@datashrine.de)